dmgen User’s Manual

Dave Ray
ray@soartech.com
June 27, 2003

Introduction

The purpose of the Soar Datamap Generator (dmgen) is to generate a datamap through static analysis of Soar productions. The generated datamap can be used to better understand the structure of a Soar system, especially when used in conjunction with SoarDoc.

It should be noted that Visual Soar has a similar datamap generation facility. However, this implementation relies heavily on the analyzed code already being in Visual Soar format, which is a severe limitation for large systems such as TacAir-Soar that were developed prior to the creation of Visual Soar. dmgen can be applied to almost any Soar system without modifications to the source code.

See Sample Processing at the end of this document for an example of how dmgen works.

Requirements

dmgen requires an installation of Soar and its accompanying version of Tcl/Tk. It has been tested with Soar versions 7.3 and 8.3 on both Linux and Windows. The binary distribution of dmgen is compatible with Tcl version 8.0 and 8.3.

Identifiying Problem-Spaces

dmgen identifies problem-spaces by searching for particular tests on states. For TacAir-Soar, this test is `^problem-space.name`. For Visual Soar generated code, the test is simply `^name`.

This difference is held in the **ProblemSpaceSpec** configuration parameter. In your config file set **ProblemSpaceSpec** as follows:

```
Set ProblemSpaceSpec "problem-space name"
```

for TAS-style systems, or:

```
Set ProblemSpaceSpec "name"
```

for Visual Soar style systems.

The default value is `"name"`

From a Tcl shell, you can set this config parameter, or any other, as follows:

```
      DmGenCfg::Set ProblemSpaceSpec "problem-space name"
```

To get the value of a config parameter:

```
      set v [DmGenCfg::Get ProblemSpaceSpec]
```

Running from the Command-Line

In order to generate a datamap for your code, dmgen needs a Tcl/Soar file that, when executed by Tcl, will load the productions for it. Fortunately, many Soar systems already have this file in the form of **source.soar** or **load.soar**. It is specified with the **InputFile** config parameter. The default value for **InputFile** is **source.soar**.

dmgen also needs to know the path to the Soar library (i.e. /Soar-8.3/library).
This can be set with the **SOAR_LIBRARY** environment variable, or with the **SoarLibraryPath** config parameter. It is important that the Tcl version you are running dmgen with is the same version used by the Soar library. If they are not the same, the Tcl interpreter will crash when it tries to load the Soar library.

Assuming your **source.soar** file is in the current directory and **SOAR_LIBRARY** is set, here is a basic invokation of dmgen:

```
      $ tclsh /path/to/dmgen.tcl
```

or,

```
      $ wish /path/to/dmgen.tcl
```

These examples assume that the Tcl bin directory is on the system path. Note that on some systems, **tclsh** and **wish** include the Tcl version number, e.g. tclsh80 or wish83.

Using **wish** rather than **tclsh** will provide a simple Tk GUI for viewing the resulting datamap.

The above commands will generate a set of XML files representing the generated datamap in a new directory called 'xml'. This directory is suitable input for SoarDoc.

Calling dmgen from a Tcl Shell

dmgen can also be called from within an interactive Tcl shell such as the Soar TSI. This can come in handy when external dependencies such as simulation interfaces make it difficult to load your code straight from Tcl.

This is pretty straightforward, but definitely more work than calling dmgen from the commad-line. Here are the basic steps:

```
# Set up library paths...assumes that path to soar-x.x/library is in
# the SOAR_LIBRARY environment variable. These are only necessary
# if you're running in a plain-old TCL console. A Soar agent window
# should be ready to go.
lappend auto_path $env(SOAR_LIBRARY)
# Load Soar (if not already loaded)
package require Soar

# Load productions here... or anywhere else. If you have an agent
# window, your productions are probably already loaded.

# Load the datamap generator
lappend auto_path "path/to/dmgen"
# By adding dmgen to the auto_path, there is no need to source
# dmgen.tcl. Tcl will automatically find procs as necessary

# Generate a datamap, giving it a name. This will generate a datamap
# using all loaded productions. This command takes an optional
# second argument, a list of names of productions to process in case
# you want to leave some out. dmgen provides a function,
# GetSoarProductions, that will return a list of all loaded
# productions.
set dm [GenerateDatamap "My datamap"]
```

Now a reference to the datamap is available through **`$dm`**. If Tk is available, a datamap window will also appear with title "My datamap".

A Note on Performance

dmgen is implemented completely in Tcl, so it is *very* slow. Don't be surprised if it takes several minutes to generate a datamap for a large system (e.g. ~6000 TacAir-Soar productions on a 1.5 Ghz machine takes more than 15 minutes to generate a datamap). If you're using **tclsh**, you should be able to see dmgen's log output directly on the console. For **wish** or the TSI, it's not as easy to tell that anything is happening (since the console isn't updated until the interpreter is idle). Try looking at the log file (dmgenlog.txt); if it's size is changing, then dmgen is still running. ☺

Configuration Options

Command-Line Options

Details of dmgen's command-line options can be found by running dmgen with the **–h** option. For example:

```
    $ tclsh /path/to/dmgen.tcl -h
```

Configuration File

dmgen is currently implemented in Tcl and for convenience uses Tcl to specify its configuration file. When dmgen is run it loads default values for configuration options. Next, it checks whether an alternate configuration file has been provided with the **–f** command-line option. If so, the file is loaded. Otherwise dmgen looks for a file called

dmgenfile in the current directory and, if it exists, it is loaded. This behavior is similar to that of Doxygen (doxyfile) or make (makefile).

The **–g** command-line option will tell dmgen to print a default, well commented, configuration file to standard output. Thus the following command could be used to generate **dmgenfile**:

```
$ soardoc -g > soardocfile
```

When generated this file contains a complete list of configuration parameters with descriptions and default values. The comments in this file are the documentation for all of dmgen's configuration parameters.

Config parameters can be overridden at the command-line using the -C option:

```
$ tclsh /path/to/dmgen.tcl -C InputFile load.soar
```

This command will run dmgen with the **InputFile** parameter set to **load.soar**.

Configuration options in the config file are set using the **Set** command:

```
Set NameOfParameter Value
```

Note the capital "S" in **Set**. If **Value** is a string with any white space, it must be enclosed in quotes.

Output Options

Besides a simple graphical display, dmgen can output a datamap in XML or HTML format. XML output is suitable for use by SoarDoc. The HTML output is a very basic browsable datamap. The output format is controlled by the **OutputFormat** config parameter. The output directory is specified by **OutputDirectory**. The default values are:

```
Set OutputFormat xml
Set OutputDirectory xml
```

Thus, by default, dmgen will generate an XML datamap in a directory called xml, created in the current directory.

If OutputFormat is "none", then no output will be generated.

These output options are also available from the File menu of dmgen's datamap window if you're using wish.

Programmatic Datamap Output

If you're running dmgen from a Tcl shell or script, you can programmatically generate XML or HTML output as follows:

```
# Write $dm to XML
DmGenXmlOut::WriteXmlDatamap "destination/path" $dm
```

or,

```
# Write $dm to HTML
DmGenHtmlOut::WriteHtmlDatamap "destination/path" $dm
```

Datamap Viewer

As described above, if Tk is available, a simple viewer/editor window will appear that allows you to browse problem spaces and operators using the problem-space and operator menus.

You can do basic editing of the generated datamap here. When you have modified an attribute, it's name will be highlighted in the tree. To save changes, go back to the main datamap window and choose **File->Save Patches.** The next time you generate the datamap, you can do **File->Load Patches** to restore your changes. This functionality is fairly untested, so be forewarned.

The **File->Save To HTML/XML**... menu items will write out the datamap.

Tcl API

When use from a Tcl shell or script, dmgen exposes several useful Tcl procedures and methods:

GenerateDatamap [name] [productions]

> Generates a datamap called **name** from a list of production names (**productions**). If **productions** is omitted, then all loaded productions are processed. Returns a handle to the datamap.

DmGenXmlOut::WriteXmlDatamap path datamap

> Writes a datamap to a directory as a set of XML files. **path** is the target directory for writing. **datamap** is a datamap handle returned from **GenerateDatamap.**

DmGenHtmlOut::WriteHtmlDatamap path datamap

> Writes a datamap to a directory as a set of HTML files. **path** is the target directory for writing. **datamap** is a datamap handle returned from **GenerateDatamap.**

Trouble-Shooting

Here are problems/questions that may arise while using dmgen:

dmgen doesn't identify any problem-spaces (besides top-ps or any-ps)

This is almost always a case of **ProblemSpaceSpec** being set incorrectly.

dmgen seems to miss a problem-space or operator

It this isn't caused by an incorrect **ProblemSpaceSpec** config parameter (see above), the second most likely candidate is that there were errors while loading productions. The most likely causes of this are:

- A production fails to load because it uses an unknown RHS function. In this case you may have to run dmgen from a shell with the full simulation interface available to your code.
- A production fails to load because SOAR_LIBRARY points to a version of Soar that is incompatible with the source code. For example, attempting to process TankSoar with Soar 7.3 will cause several production to fail to load.

To find out what caused

Where does the top-ps problem-space come from?

top-ps comes from any attributes that are tested from the **^top-state** attribute. This behavior can be suppressed by setting the **FillTopPs** config parameter to **0**.

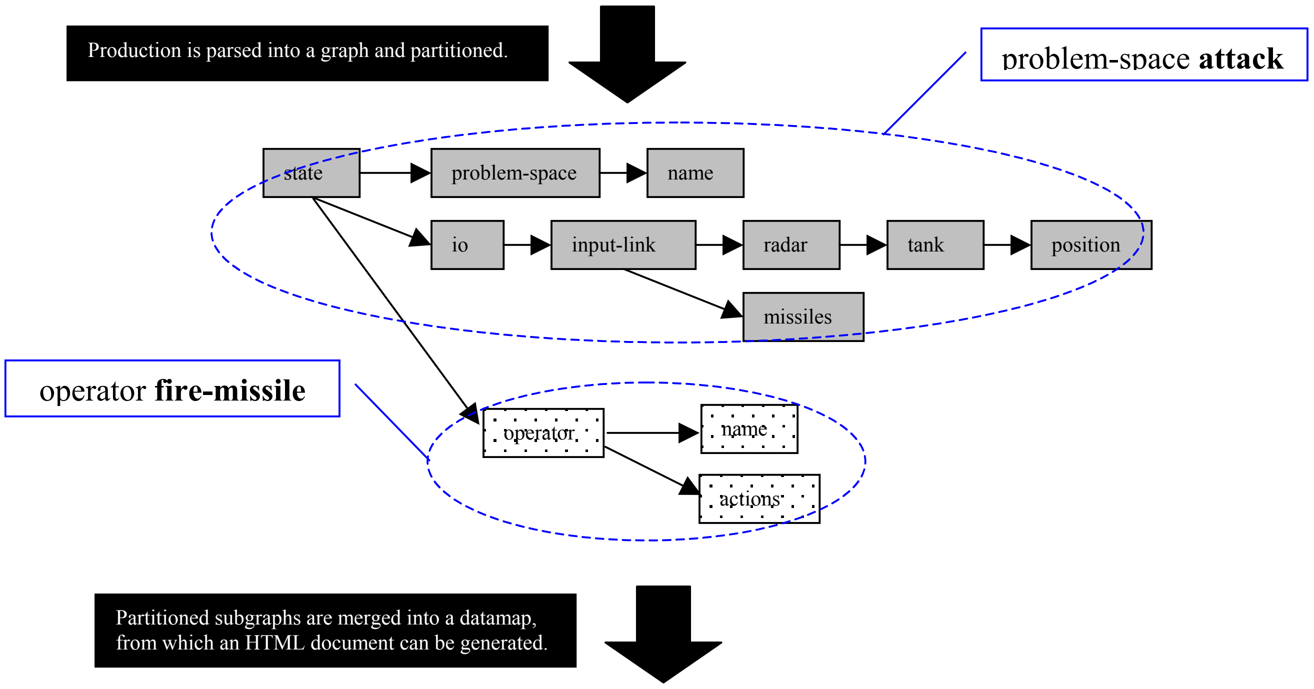
Where does the any-ps problem-space come from?

Whenever a production does not test for a particular problem-space or operator, dmgen will merge the attributes that that production references into a dummy problem-space called **any-ps**. This is basically everything that dmgen could not find a home for. This behavior can be suppressed by setting the **FillAnyPs** config parameter to **0**.

Sample Processing

Here is a sample of the processing that dmgen performs on a single production:

```
sp {attack*propose*fire-missile
   (state <s> ^problem-space.name attack
              ^io.input-link <il>)
   (<il> ^radar.tank.position center
         ^missiles > 0)
-->
   (<s> ^operator <o> +)
   (<o> ^name fire-missile)
   (<o> ^actions missile)
}
```


Production is parsed into a graph and partitioned.
problem-space **attack**
state
problem-space
name
io
input-link
radar
tank
position
missiles
operator **fire-missile**
operator
name
actions
Partitioned subgraphs are merged into a datamap, from which an HTML document can be generated.

Problem Space: attack

- **operator**, Type: *identifier*
  Links: fire-missile,
- **problem-space**, Type: *identifier*
  - **name**, Type: *string* , Value(s): attack
- **io**, Type: *identifier*
  - **input-link**, Type: *identifier*
    - **missiles**, Type: *int* , Value(s): 0
    - **radar**, Type: *identifier*
      - **tank**, Type: *identifier*
        - **position**, Type: *string* , Value(s): center

Operator: fire-missile

- **actions**, Type: *string* , Value(s): missile
- **name**, Type: *string* , Value(s): fire-missile