© 2004 THREEPENNY SOFTWARE LLC

Title:	**XML Interface to Soar (SML) Software Specification**
Written By:	Douglas Pearson doug@threepenny.net
Client Contacts:	
Date:	June 9, 2005
File Name:	Soar XML Interface Specification.doc

REVISION HISTORY

Rev	Revision Description	Date	Modification by:	
00	ISSUED	7/26/2004	Douglas Pearson	
01	Added ClientSML and KernelSML section	8/4/2004	Douglas Pearson	
02	Redesigned the I/O model and WME representation to better fit how we expect the client to use it.	8/7/2004	Douglas Pearson	
03	Specified Connection interface	8/8/2004	Douglas Pearson	
04	Added time tags to WMEs	8/9/2004	Douglas Pearson	
05	Revised ElementXML and Connection interfaces. Added MessageGenerator, <agent> and <agents>	8/11/2004	Douglas Pearson	
06	Added SML Command language section	8/16/2004	Douglas Pearson	
07	Added gSKI commands	8/31/2004	Douglas Pearson	
08	Removing gSKI Commands, updating I/O commands.	10/26/2004	Douglas Pearson	

© 2004 THREEPENNY SOFTWARE LLC

XML Interface to Soar (SML) Software Specification

ThreePenny Software, LLC

Table of Contents

Table of Figures

1. Introduction

1.1. XML Interface to Soar (SML)

This specification is to define an XML interface for sending commands to and receiving information from Soar version 8.6 or later.

1.2. Environment

The SML interface will be intended to run on the Windows, Linux, Unix and Mac platforms.

1.3. Glossary

Term	Meaning
gSKI	Generic Soar Kernel Interface. The new (as of Soar 8.6) interface into the Soar kernel.
Soar Kernel	The central processing part of Soar, where productions are matched, working memory is managed and chunks are learned. In the latest version (8.6) it should not contain any logic for interpreting commands or generating trace output.
In process	Soar and a tool are said to be "in process" if they both execute within the same process (as defined by the operating system). In Windows, if you run Soar and the tool you would see only one button on your task bar.
Out of process	Soar and a tool are said to be "out of process" if they execute as separate processes. In Windows, if you run Soar and the tool you would see two buttons on your task bar.
SML	"Soar Markup Language". The new XML interface proposed in this document for connecting tools to Soar.
Soar Tool	A tool, such as a debugger or editor, that wishes to connect to Soar. The tool will issue commands such as "run", "print" etc. and receive notification of events from Soar such as "stopped-running".
Soar Simulation	A simulator, such as a flight simulator or game board, that is connected to Soar. A Soar agent will receive input from the simulation (e.g. where another plane is located) and can send commands to the simulation to take action (e.g. to land the plane).
Client	Generic term used to refer to either a Soar Tool or a Soar Simulation.

2. Overview

The XML interface is intended to allow tools, such as debuggers, to control Soar by sending commands packaged in small pieces of XML and then receive output from Soar structured as an XML file. In many ways, this resembles a SOAP interface into Soar. The interface can also be used to connect simulations to Soar, with input being sent to Soar as an XML stream and output received from Soar as XML.

The most basic model for this communication is shown in Figure 1, although the communication between Soar and its tools would not be limited to just this model. The tool issues a command by creating an XML object that contains command and parameters. That object is then converted into an XML stream (a sequence of ASCII characters) which is then sent in some manner to the Soar process where it is converted back into an object representing the command and the command is executed (by an appropriate call to gSKI). The results of the command are then packaged and returned in a similar manner.

A simple example is the command "print chunk-1". The XML generated by the tool might take the form (in XML pseudo code):

```
 <Command name="print" arg="chunk-1" Command></Command>
```

and the results send back by Soar might be:

```
 <Production name="chunk-1">
        <Condition>(state <s> ^object <obj>)</Condition>
        <Condition>(object <obj> ^color red)</Condition>
        …
</Production>
```

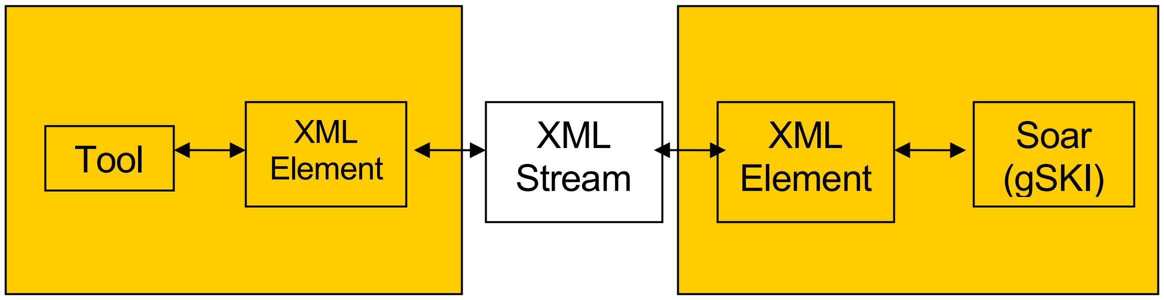

Figure 1 Basic Communication Model

It's important to recognize that the XML interface layer would connect to Soar through gSKI and is not intended to either replace gSKI or prevent tools or other users of the Soar kernel from connecting directly to gSKI if they wish. It is instead intended to provide additional functionality that may be useful for some tools and other users of the kernel.

2.1. Communication Models

2.1.1. Out-of-Process Model

Soar and the external tool or simulation each run in a separate process and communicate by sending an XML stream through a socket.

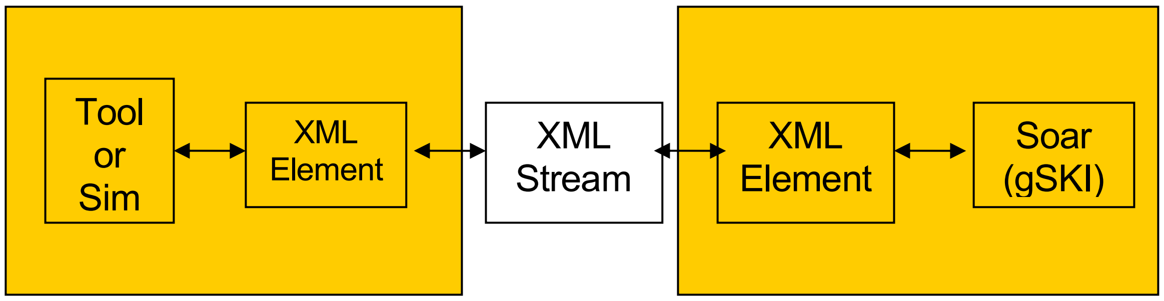

Figure 2 Out-of-Process Communication Model

2.1.2. In-Process Model

Soar and the tool or simulation exist in a single process, communicating by passing objects back and forth that represent XML nodes but without ever actually converting the objects to XML.

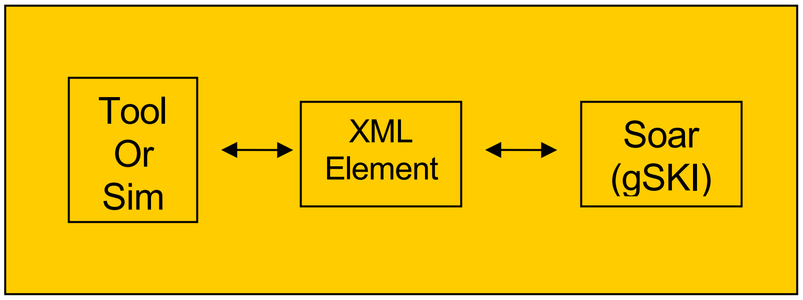

Figure 3 In-Process Communication Model

2.1.3. Remote Model

Soar and the external tool or simulation each run on a different machine, communicating via sockets to send and receive an XML stream.

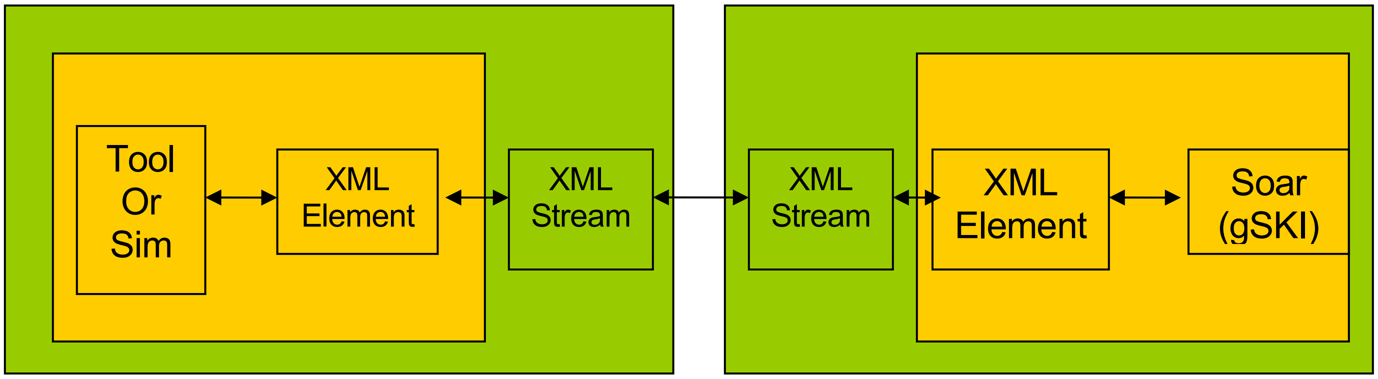

Figure 4 Remote Communication Model

2.2. XML Content and XML Element interfaces

The SML interface actually consists of two separate yet related software interfaces:

- XML Content Interface
- XML Element interface

The XML Content interface defines what XML documents should be sent and received between Soar and a Soar tool or simulation. The XML Element interface goes beyond that and also defines a C level interface for an object that represents an XML command (or piece of output) before it is converted into an actual XML Stream.

For example:

<Command name=”print” arg=”chunk-1” Command></Command> would be part of the XML content.

Int getNumberAttributes() ; would be part of the XML Element interface.

The reason for defining the XML Element as well as the XML Content is primarily for improved efficiency. If a tool and Soar share the same process, then it would be possible for them to transfer XML Elements without converting the object into an ASCII stream (as XML) and then parsing that stream to re-create the object.

A side benefit of defining the XML Element interface is to indicate which subset of XML functionality is sufficient to support the XML content. XML has grown to be a complex language with many capabilities and an implementer may wish to know which subset of XML is sufficient for talking with Soar. They may determine that either by reading this specification or by implementing the XML Element interface (which imposes a source code level constraint).

Any tool is free to ignore the XML Element interface completely (even if it is “in process”) and simply generate XML content and send that as an XML stream to Soar.

3. Motivations

The reasons for proposing the XML interface are explained here. The intention is that each of these reasons is sufficient motivation for creating the XML layer (as different motivations may be more relevant to different user groups).

3.1. Command line interface / Tcl Debugger (TSI)

The existing Tcl Debugger interacts with Soar by sending commands (encoded as strings) and parsing the results (also encoded as strings). To support this debugger with Soar 8.6 we need to write:

1. A command parser to convert the commands typed by the user (or otherwise sent by the Tcl interfaced) into calls to gSKI.

2. An output generator to convert from the gSKI representations of objects (e.g. the class representing a Soar production) to a string version that can be shown to the user.

Note that gSKI does not include either this command line parser or any functions to generate output. Part of the design of gSKI is that this functionality should lie outside of the kernel.

There is currently a project to write this parser and output generation code as a module called TgD. The functionality of this component is very similar to that of the XML interface layer and the work to write either should be similar, but the TgD effort will produce a component tied to the Tcl Debugger, while the XML interface effort would produce a much more general solution that could be used by any tool, including the Tcl Debugger.

We may decide at some point to drop support for the Tcl Debugger. However, if any tool is going to support a command line interface then it is still worthwhile to write this SML interface layer as it provides that command line support, but in a more structured manner than a traditional command line interface based around strings.

As a final consideration of the value of a command line interface it's worth noting that JESS (the Java Expert System Shell) offers an interface very similar to gSKI together with one additional function "executeCommand" which accepts any valid string of JESS commands (essentially a command line interface). JESS, like Soar, can be run either as an embedded process or standalone. This command line interface makes it possible to send the same set of commands to either an embedded system (which may be very hard to debug) or to a standalone instance of JESS (which is much more open to interrogation). Without the command line interface this would not be possible and embedded debugging would be much harder.

3.2. Support for logging

One desirable capability for a debugger is the ability to review output files logged from a running process when no debugger is attached. In order to support this we would need to define a file format. The XML interface layer gives us support for this file format and logging with minimal additional effort.

3.3. Loose coupling between tools and the Soar kernel

If a tool, such as a debugger, links directly to gSKI it will be tightly coupled to a particular version of the Soar kernel. This is because the gSKI interface is very large and any change to the kernel is likely to require some change to the interface. This in turn would mean that the debugger would need to be re-linked to the new interface.

If we define an XML interface this tight coupling does not need to occur. If the Soar kernel changes, the **content** sent over the XML interface will likely change but the ability to support reading and sending XML would not change. This means it becomes possible for a tool to support multiple versions of the Soar kernel simultaneously which could not be achieved with a direct link to gSKI.

This is a fairly subtle point but produces a very significant impact on the usefulness of a tool. The history of Soar shows that at any given time the members of the Soar community are using a wide range of different versions of Soar. It's important that they not be required to upgrade to the latest version of the kernel in order to use the latest version of the tools available for working with Soar. Also, the funding available for working on the kernel and the tools has historically not gone in lock step. This means in many cases the kernel may advance while the tools remain unchanged or vice versa. If the tools simply stop working when a new release of Soar comes out this will lead users to quickly become frustrated and stop using a tool. However, if the tool continues to work correctly in 99% of cases and only has problems in the area where the kernel was changed, users will be more likely to understand and accept that limitation and continue to use the tool. This is only possible if the tools and the Soar kernel are not tightly coupled.

3.4. Support for out of process and remote connections

There are potentially situations where we would like to run Soar in a different process from the tool that is connecting to it or even where Soar is running on a different machine. In order to do this it's necessary to define some format for the data that is sent between the tool and the Soar kernel. This can either be a binary format (producing a DCOM style solution for those familiar with Windows) or it can be a text format (such as XML).

The XML interface should give us the ability to cross process boundaries and even machine boundaries with little additional effort as it is only necessary to transfer the XML representation between the machines in order to connect them. There is currently no similar capability in the existing gSKI interface so Soar is required to run in the same process as the tool connecting to it.

This motivation is less compelling than the others to me as it has so far been possible to live with this limitation when using the Tcl debugger. Out of process and remote connectivity have always been achieved by having Soar's I/O link send and receive data over a socket allowing the environment to be moved to being out of process from Soar. However, if we can remove this limitation it's an added benefit of the XML interface and may make it possible to connect tools and Soar in new ways.

3.5. Language independence / simpler integration

XML is completely language independent, so tools can be written in any language and connect to Soar through this interface. That is a minimal gain as support for C level interfaces is so wide-spread. However, while any language **can** connect to a C language interface (such as gSKI offers) building and maintaining this interface can be a significant challenge. The current gSKI header files consist of over 2,500 lines of code which gives an indication of the size of the gSKI interface. In order to create an interface directly to gSKI from another language (e.g. a JNI interface from Java) one would need to support this entire interface and every change within gSKI would require a change to that cross-language interface.

By offering an XML layer, a tool builder has the choice of either integrating directly to gSKI (a simple operation if the tool is in C++) or the tool builder can choose to interface through the XML interface. In this case, the size of the programmatic interface will be much smaller (the number of functions that need to be written in a cross-language interface) because the bulk of the interface becomes content, making it only necessary to support reading and writing XML which can be done in about a dozen functions.

In short, the XML layer should make integration with other languages simpler and less prone to change as the Soar kernel changes.

3.6. Comprehensive naming

A small but significant motivation for the XML interface is that it will help to ensure objects within the Soar kernel continue to be named. Many of the names within the existing kernel are only present because of their role in helping users refer to objects within the kernel in a principled fashion. The most obvious example is the name of a production which has no role in the loading or firing of a production, but is clearly vital in explaining to a user which production has fired or allowing the user to indicate which production to excise.

It may seem ridiculous to imagine Soar without names, but gSKI now provides a well defined notion of maintaining a pointer to a Soar kernel object, so it would be possible to design an interface without names and still have it be well defined (whilst before gSKI maintaining pointers into the kernel would clearly be unsafe).

The most obvious example of how names can be lost along the way is the name of an agent. When multi-agent capabilities were added to Soar, the Tcl interface already existed and so each agent was tied to a unique Tcl interpreter. As a result, the agents were uniquely identified within Soar by that Tcl interpreter pointer rather than an agent name. (There was an agent name field within the kernel but it actually contained the address of this interpreter as a string). Needless to say, this implementation presented problems when other tools wished to connect to the kernel and send commands to a particular agent.

In order to build an XML interface there will need to be some way to refer to objects within the kernel using something other than a pointer, ensuring that naming remains as complete within the kernel as it is today.

3.7. XML will provide structure

One final consideration is whether it is necessary to provide XML communication in both directions. We could instead send simple strings in either one or both directions. Adding an XML layer certainly adds overhead, but there are several ways we can attempt to reduce this overhead (see below for details of passing data in-process without conversion to XML and the support for different levels of detail in the XML content).

Sending XML instead of simple strings allows both the command sender and the output parser to be less brittle and less sensitive to the exact format of output or commands.

4. XML Element Specification

4.1. Introduction and Design Principles

This section defines the C-level interface to an object that represents an element in an XML document. See Section 2.2 for an overview of this interface and motivation for its creation.

The design of this interface is intended to be:

1. Sufficient to meet the needs of the XML Content interface

2. As simple as possible while still meeting the needs of the XML Content interface.

By keeping the interface simple it will make integration with other tools and languages as easy as possible.

4.2. XML Elements Supported

We'll start by defining the types of XML elements that will be supported by the interface and then define the interface to provide those capabilities. To help make the definition more concrete, consider this example of an XML document:

```
<Production name=”chunk-1”>
        <Condition>(state &lt;s&gt; ^object &lt;obj&gt;)</Condition>
        <Condition>
                <![CDATA[ (object <obj> ^color red) ]]>
        </Condition>
</Production>
```

XML Feature Supported	Description
Element	Simple tagged component. An XML document consists of exactly one element (which then contains other elements as children). <Production>…</Production> is an example of an element.
Children	An element can contain an arbitrary number of children. In the examples, the <Condition>…</Condition> elements are the children.
Tag name	The name of the tag within an element. In the example, Production is a tag name. Tag names are case sensitive. Tag names consists of

	letters, numbers and "." "_" or "-".
Attribute	Attributes are name-value pairs where the value is always a string. In the example, name = "chunk-1" is an attribute.
Character data	The contents of an element that lies between its start and end tag. In the example, "state &lts&gt ^object &ltobj&gt" is the character data for the first Condition element.
Special characters	**&lt;** Less-than "<" **&gt;** Greater-than ">" **&amp;** Ampersand "&" **&quot;** Double quotation marks – " **&apos;** Apostrophe or single quotation mark – ' The first condition shows an example of using these escape sequences.
CDATA Section	A CDATA section is used to wrap complex character strings (such as occur within a Soar production). Values within a CDATA section are not interpreted by an XML parser so there is no need to use the special characters &gt; etc. A CDATA section starts with **<!CDATA** and ends with **]]>** For the SML subset of XML we will require that character data is either a single CDATA section or does not contain any CDATA sections. (So CDATA there will not be multiple CDATA sections and regular character data will not be interleaved with CDATA sections). <![CDATA[(object <obj> ^color red)]]> shows an example of using a CDATA section within the character data for the second Condition.

There is much more allowed by XML than this small subset but these are not required by SML and so are not required in the XML Object interface. They include:

- Processing instructions

- Comments
- Document Type Definitions (DTDs)
- Support for non-UTF8 character encodings

4.3. ElementXML Interface

Function	Description
ElementXML() ;	Default constructor for the element.
~ElementXML() ;	Destructor for the element. This is private. Call re-leaseRef() instead.
int addRef() ;	Add a new reference to the object. Returns new ref count.
int releaseRef() ;	Release a reference from the object. If ref count reaches 0 the object is deleted. Returns new ref count.
Static bool isValidID(char const* str) ;	Returns true if the string only contains letters, num-bers, “.”, “-“ and “_”.
setTagName(char* tagName) ;	Set the tag name for the element. Tag name can only contain letters, numbers, “.” “-“ and “_”. Tag names are case sensitive.
char const* getTagName() ;	Get the tag name for the element.
Void addChild(ElementXML* child) ;	Adds a child to the list of children of this element.
int getNumberChildren() ;	Returns the number of children of this element.
ElementXML const* getChild(int index) ;	Returns the n-th child of this element. The caller should not delete this element or attach it to other objects etc.
Void addAttribute(char* atttributeName, char* attributeValue);	Adds an attribute name-value pair. Attribute name can only contain letters, numbers, “.” “-“ and “_”. Attribute names are case sensitive.
int getNumberAttributes() ;	Get the number of attributes attached to this element.
const char* getAttributeName(int index) ;	Get the name of the n-th attribute of this element.
const char* getAttributeValue(int index) ;	Get the value of the n-th attribute of this element.
const char* getAttribute(const char* attName) ;	Get the value of the named attribute (or null if this attribute doesn’t exist).
Void setCharacterData(char* characterData) ;	Get and set the character data for this element. The character data passed in should *not* replace special characters such as “<” and “&” with the XML

char const* getCharacterData() ;	escape sequences &lt; etc. These values will be converted when the XML stream is created. Returns the character data for this element. This can return null if the element has no character data.
Void setBinaryCharacterData(char* buffer, int length) ; **IsCharacterDataBinary() ;** **Int getCharacterDataLength() ;**	Stores character data that can contain embedded nulls. The length is the length of the buffer. Returns true if character data was stored through a call to setBinaryCharacterData(). If the character data is a binary buffer, this returns the size of that buffer. If the data is a string, this returns the length of the string + 1 (to include the trailing null).
Void setUseCData(boolean useCData) ;	Setting this value to true indicates that this element's character data should be stored in a CDATA section. By default this value will be false.
Static void deleteString(char* string) ;	Utility function to release memory allocated by this element and returned to the caller.
Static char* allocateString(int length) ; **static char* copyString(const char* original);** **static char* copyBuffer(char* original, int length)**	Utility function to allocate memory that will then be passed to the ElementXML functions. The length is the number of characters in the string, so length+1 bytes will be allocated (so that a trailing null is always included). Thus passing length 0 is valid and will allocate a single byte. copyString performs an allocation and then copies the contents of the passed in string to the newly allocated string. Allocates a new buffer of size "length" and copies exactly that many bytes from original to the new buffer. This function is for use with binary character data (a rare situation).

4.3.1. C versus C++

Although the above interface is defined in C++ terms, it is likely that it will make more sense to define a C interface rather than a C++ interface. In that case the actual list of exported functions will be those shown below. The "ElementXML" type is just an integer which is passed to every function and should be treated by the client as a handle (i.e. an opaque reference to an object).

C Function definitions
typedef int ElementXML ;
ElementXML newElementXML() ;
ElementXML newElementXMLCopy(boolean copyStrings) ;
void deleteElementXML(ElementXML element) ;
setTagName(ElementXML element, char* tagName) ;
char const* getTagName(ElementXML element) ;
void addChild(ElementXML element, ElementXML child) ;
int getNumberChildren(ElementXML element) ;
ElementXML getChild(ElementXML element, int index) ;
void addAttribute(ElementXML element, char* atttributeName, char* attributeValue);
int getNumberAttributes(ElementXML element) ;
const char* getAttributeName(ElementXML element, int index) ;
const char* getAttributeValue(ElementXML element , int index) ;
const char* getAttribute(ElementXML element , const char* attName) ;
void setCharacterData(ElementXML element , char* characterData) ;
char const* getCharacterData(ElementXML element) ;
void setUseCData(ElementXML element , boolean useCData) ;
void deleteString(char* string) ;
char* allocateString(int length) ;
char* copyString(const char* original);

4.3.2. Memory allocation

The methods setTagName(), addAttribute() and setCharacterData() offer a range of options to do with ownership of the strings passed in. These forms allow a number of optimizations which can all be safely ignored by a casual user. However, if you are interested in the potential optimizations read on.

Each function is implemented as a series of C++ functions. E.g. for setTagName() we have:

```
setTagName(char* pName, bool copyName = true) ;

setTagName(char const* pName) ;

setTagNameFast(char const* pName) ;
```

In the first function, if copyName is true then pName will be copied by ElementXML. This is safe and the default behavior. If copyName is false, then ElementXML takes ownership of the pName string and will call deleteString() on this string when the ElementXML object is deleted. So if the caller passes “copyName = false” it must have allocated the string with either allocateString() or copyString() and must not delete it once passed in. The reason for allowing this, is that a smart client that is building up a string can do so with a call to allocateString() (e.g. while parsing XML) and then just pass that string into this class without an additional allocation and copy. For large strings this can be significant in both memory and time.

The second function is just a convenience. If a constant string is passed in, it must be copied (deleting a constant string is a bad idea). It simply calls “setTagName(CopyString(pName), false)”.

The third function exists, but is protected because it is must be used with care (note it’s name is also different, so accidentally use will not occur). This function requires that the constant string passed in remains in scope for the life of the ElementXML object. In practice, this usually means the constant is a static constant. The string is neither copied nor deleted. In SML there are many such constants (e.g. tag names, attribute names) but one cannot programmatically distinguish what the life of such a string will be, so the developer needs to use this one with care (which is why it’s not part of the public interface). To use it, you subclass from ElementXML.

4.3.3. Ownership and reference counting

The ElementXML objects will use a reference counting system, so when either the kernel or client has finished with an object it can call deleteElementXML() and the underlying object will in fact only be deleted if there are no other existing references to the object. Thus the normal use of an ElementXML object will be:

a) Create the ElementXML object

b) Call SendMessage to pass to another process

c) Delete the ElementXML object

5. XML Content Interface

5.1. Introduction

This section defines the XML documents that will be sent to and from the Soar kernel through the SML interface. Refer to section 4.2 for a list of the types of XML constructs used within the interface.

Let's start with some examples of SML documents and then follow it with a more precise specification:

5.1.1. Sample SML document to send "print chunk-1" command

```
<sml smlversion="1.0" doctype="call" soarVersion="8.4.2" id="1234" >
<command name="print">
        <arg param="agent">agent-1</arg>
        <arg type="string" param="name">chunk-1</arg>
</command>
</sml>
```

This is a request sent from a tool to the Soar kernel asking for the production named "chunk-1" to be printed. This command specifically indicates it is intended for "agent-1" (specifying the intended agent is optional).

5.1.2. Sample SML document to send "source towers.soar" command

```
<sml smlversion="1.0" doctype="call" soarVersion="8.4.2" id="1234" >
<command name="cmdline" output="raw">
        <arg param="agent">test-agent</arg>
        <arg param="line">source towers.soar</arg>
</command>
</sml>
```

This is a command to load a set of productions from the file "towers.soar". This command uses the command line interface, so the command is "source towers.soar" which is then parsed by the command line interpreter. This uses the same format as a user could type at the command prompt for Soar and allows a great number of commands to be sent easily.

5.1.3. Sample SML document containing "chunk-1" output

```
<sml smlversion="1.0" doctype="response" soarVersion="8.4.2" id="1275" ack="1234">
<result output="structured">
<production name="chunk-1">
        <condition><![CDATA[(state <s> ^problem-space <p>)]]></condition>
        <condition><![CDATA[(<p> ^name blocks-world)]]></condition>
</production>
</output>
</sml>
```

This is output generated from the kernel showing a production with two conditions (the rest are omitted for simplicity). Note that the ack field matches the id of the originating command. The original production was:

```
sp {chunk-1
        (state <s> ^problem-space <p>)
        (<p> ^name blocks-world)
}
```

5.1.4. Sample SML document containing "after decision cycle" event notification

```
<sml smlversion="1.0" doctype="notify" soarVersion="8.4.2" id="1248">
<command name="event">
        <arg param="agent">test</arg>
        <arg param="eventid">18</arg>
        <arg param="phase">4</arg>
</command>
</sml>
```

This is a notification that is sent from the kernel to notify a tool that Soar has completed a decision cycle. That this is an "after decision cycle" event is determined by the eventide. The "phase" in this case indicates where in the decision cycle Soar was when this event was fired.

5.1.5. Sample SML document containing input for agent-1

```
<sml smlversion="1.0" doctype="call" soarVersion="8.4.2" id="1234">
<command name="input">
        <arg param="agent">agent-1<.arg>
        <wme action="add" att="plane" id="i1" tag="-5" type="id" value="a4"></wme>
        <wme action="add" att="name" id="a4" tag="-6" type="string" value="plane1"></wme>
        <wme action="add" att="speed" id="a4" tag="-7" type="int" value="225"></wme>
        <wme action="add" att="position" id="a4" tag="-8" type="id" value="a5"></wme>
        <wme action="add" att="x" id="a5" tag="-9" type="int" value="65"></wme>
        <wme action="add" att="self" id="a4" tag="-10" type="id" value="a4"></wme> ; Note – same value
as id
</command>
</sml>
```

This represents: (I1 ^plane P4) (P4 ^name plane1 ^speed 225 ^position P5 ^self P4) (P5 ^x 65). For input, lower-case IDs (e.g. "a4") are used to refer to objects created by the client which have yet to have actual Soar IDs assigned to them. Upper-case IDs (e.g. "I1") are used to refer to objects that already exist within the kernel. Similarly, negative time tags (e.g. "-3") are used to refer to objects created by the client, while positive time tags (e.g. "5") are used to refer to objects created by the kernel. Time tags are used to identify wme's when we're removing values from working memory.

5.1.6. Sample SML document containing output

```
<sml smlversion=”1.0” doctype=”call” soarVersion=”8.4.2” id=”1234”>
<command name=”output”>
        <arg param=”agent”>agent-1</arg>
        <wme action=”add” att=”turn” id=”O1” tag=”7” type=”id” value=”T1”></wme>
        <wme action=”add” att=”heading” id=”T1” tag=”6” type=”int” value=”045”></wme>
        <wme action=”add” att=”speed” id=”T1” tag=”12” type=”int” value=”225”></wme>
</command>
</sml>
```

This represents: (O1 ^turn T1) (T1 ^heading 045 ^speed 225).

5.2. General document structure and the <sml> tag

Each XML command sent to Soar and each piece of XML output sent from Soar will be a valid XML document. Each valid SML document will always start with an <sml> element.. The children of this element will specify the details of the content being sent. (This specification is intended to be descriptive rather than proscriptive).

5.2.1. Example of this tag (if any of the examples accidentally contradict the spec, follow the spec):

```
<sml smlversion=”1.0” doctype=”call” soarVersion=”8.4.2” id=”1234”>
        [body of message goes here]
</sml>
```

SML Element	Description
Tag: sml **Required in a valid SML document.**	Tag for SML node. Has no parent node in the document.
Attribute: smlversion=”n.n” (n is one or more digits) **Required in a valid sml tag.**	Defines the version of SML used in this document. This version number should usually not be changed if there are additions made to the SML spec, only if something is removed or modified. Changes to the Soar version number do not require a change to the SML version number.
Attribute: soarversion=”n.n.n” (n is one or more digits) **Required.**	Messages from the kernel will always include the current kernel’s version in this field. Messages from a tool will include the latest version of Soar that the tool was designed for. This field allows either side of the communication to

	be smart in supporting older versions. A kernel that's version 8.5 might decide to accept 8.4 print commands that were formatted slightly differently than in 8.5. Similarly a tool built to support the 8.6 version of the kernel might accept output generated by the 8.5 kernel. Different SML documents could validly contain different soarversion fields (where one side essentially pretends it is a different version than it really is) although it'll be best to avoid this.
Attribute: doctype="call" | "response" **Required.**	Indicates the general nature of this document. "call" : The sender is issuing a command to the receiver. Generally calls will be from tools to the kernel but are not limited to this. The kernel could issue a command to a tool or a tool might send a command to another tool. "response": Result from a previous call being returned to original sender. Generally, responses will be from the kernel to the tools although again not limited to this. Calls are always matched to a response. The response may contain nothing beyond the header (indicating 'message received' and no more). Also, someone making a call is not required to wait for the response before continuing execution, so both synchronous and asynchronous message passing is possible.
Attribute: id=<id-string> (any string is valid) **Required.**	This string is a unique id for the message sender within a given session (a session being the life of the sending process).
Attribute: ack=<id-string> **Required for "response" messages. Not allowed in others.**	This string is used to indicate that this message is a response to another message. The value must equal the ID of the original message that is being responded to.
Children	The following tags can be children of the sml node: <command>,<result> Others may be added in future.
Character Data	Currently none.

5.3. Command tag

SML documents can contain one or more command element. Commands consist of a name followed by a series of arguments.

An SML document can only contain a single command. If you wish to send multiple commands, send multiple SML messages one after another. (We may wish to allow a tool to send multiple messages in a single string over a remote connection, thus ensuring that all commands are executed at the same time).

5.3.1. Example of this tag

```
<command name=”print”>
        <arg param=”agent”>test</arg>
        <arg type=”string” param=”name”>chunk-1</arg>
</command>
```

Command Element	Description
Tag: command **Optional in an SML document.**	Tag for command node. Has sml node as its parent.
Attribute: name=<command-name> **Required in a valid command tag.**	Specifies the command that is to be executed by the recipient.
Attribute: output=”raw” | “structured” **Optional.**	Specifies how the output from this command should be structured. “raw” indicates that the output should be returned using a raw XML node which will contain a single block of data. “structured” is the default, where the output is returned as a series of XML nodes. For example, when printing a production it could be returned as a single XML structure (with the entire production as a large string within the object) or as a series of <condition> elements which themselves contains <att> elements etc. The content of the output in each case is equivalent, but the ability to request different levels of structure in the output can allow the caller to be more efficient and can reduce the overhead in the communication. If the client has no interest in parsing the output, but just intends to display it to the user, then requesting raw output will make the client code much simpler as it won’t need to know how to convert an arbitrary SML document back into text to display to the user.

	Instead, it will only need to know how to do that for a few tags (raw, error etc.) The precise details of the content of the raw node depend on the command being executed.
Children	<arg> nodes can be children. Some commands may use other nodes as children. For example, a "send-production" command might include a <production> object as a child.
Character Data	Currently none.

5.4. Arg tag

An arg element is used to specify an argument for a command.

5.4.1. Example of this tag

```
<command name="print">
        <arg param="agent">agent-1</arg>
        <arg type="string" param="name">chunk-1</arg>
</command>
```

Arg Element	Description
Tag: arg **Optional in an SML command**	Tag for arg node. Has command node as its parent.
Attribute: param=<parameter-name> **Required.**	Specifies which parameter this is within the command.
Attribute: type="string" | "int" | "double" | "char" | "boolean" | "id" **Optional**	Specifies how the value should be interpreted. Optional because the recipient must already know how to interpret the argument (and presumably we don't define an API with multiple types in a single slot). Note that the value is always a string within the XML document, this just indicates how to parse it.
Children	Currently none.
Character Data	The value of this parameter.

Required	

5.4.2. Some common arguments

1. *Agent*

<arg param=”agent”>agent-3</arg>

This argument is used to pass the name of an agent.

2. *Eventid*

<arg param=”eventide”>18</arg>

This argument is used to pass notification of a specific event for an “event” command.

5.5. Error tag

An error element is used to report problems with the execution of a command. There will either be one error tag or the command is assumed to have completed successfully (in which case there may also be a result tag, although that is not required).

5.5.1. Example of this tag

```
<sml smlversion=”1.0” doctype=”response” soarVersion=”8.4.2” id=”id-1234”>
        <error code=”32” name=”not-implemented” source=”print-gds”>The print-gds command is not implemented in this version of Soar</error>
</sml>
```

Error Element	Description
Tag: error **Optional in an SML document**	Tag for error node. Has sml node as its parent.
Attribute: name=”not-implemented” | “invalid-arg” | “invalid-version” | “aborted” | “failed” **Optional.**	The class of error. “Failed” is a catch-all and the default value. This list may be extended later.
Attribute: source=<name> **Optional**	A description of the source of this error. For example, if an argument is invalid this could be used to report which argument was invalid.
Attribute: code=n **Optional**	A numeric code to identify the error. A particular command can define a series of error codes which it may return. The reason to define these is to allow the caller to make programmatic decisions based on the error (e.g. if file-not-found is “1” for a particular command, then the client can test this and decide it needs to create the missing file). The character data

	should never be used to control behavior (that's just a message to show to the user).
Children	Currently none.
Character Data **Required**	A description of the error.

5.6. Result tag

Result is used to report the results of a command. If a <result> tag is included, an <error> tag should not usually be included, so <result> is used to report successful output. A <result> tag is not required (so a successful command could return an empty <sml …></sml> message).

5.6.1. Example of this tag

```
<sml smlversion="1.0" doctype="response" soarVersion="8.4.2" id="id-1234">
<result output="raw">
        [Result goes here]
</result>
</sml>
```

Result Element	Description
Tag: result **Optional in an SML document**	Tag for result node. Has sml node as its parent.
Attribute: output="raw" | "structured" **Optional.**	Specifies how the output from this command should be structured. "raw" indicates that the output should be returned using a raw XML node which will contain a single block of data. "structured" is the default, where the output is returned as a series of XML nodes. For example, when printing a production it could be returned as a single XML structure (with the entire production as a large string within the object) or as a series of <condition> elements which themselves contains <att> elements etc. The content of the output in each case is equivalent, but the ability to request different levels of structure in the output can allow the caller to be more efficient and can reduce the overhead in the communication. If the client has no interest in parsing the output, but just intends to display it to the user, then requesting

	raw output will make the client code much simpler as it won't need to know how to convert an arbitrary SML document back into text to display to the user. Instead, it will only need to know how to do that for a few tags (raw, error etc.) The precise details of the content of the raw node depend on the command being executed.
Children	Depending on command
Character Data	Depending on command

5.7. Input tag

This element is used to describe a change to the input-link. The change can either be a complete copy of the input-link or it can be just the values that have changed since the last input message was sent. If the change is not a delta, the graph of WMEs sent must always start at the input-link. If the change is a delta, then the children are a list of WMEs that include the "action" attribute (to indicate if they are additions or deletions).

The input tag is a parameter to an "input" command.

IDs are values assigned by Soar to objects (e.g. "O4" is an ID). The client cannot know in advance what ID will be assigned by Soar. So when adding an object to the input link, this field is usually not specified by the client. Soar will assign the new WME an ID and maintain a record of the mapping between the client's ID and the Kernel's ID.

E.g.

```
<wme action="add" att="plane" id="i1" tag="-5" type="id" value="a4"></wme>
<wme action="add" att="name" id="a4" tag="-6" type="string" value="plane1"></wme>
<wme action="add" att="speed" id="a4" tag="-7" type="int" value="225"></wme>
```

Could be used to create (I1 ^plane P4) (P4 ^name plane1) (P4 ^speed 225), with Soar establishing a mapping from a4 to P4.

The client should attach time tags to the objects being added, so they can be removed like this:

```
<wme action="remove" tag="-5"></wme>
```

The client should use negative values for its time tags, so it's clear which type of tag is being used. Soar's time tags are always positive.

Note that Soar allows input to be added to both the input-link and the output-link. For example, additions to the output-link might be used to indicate when a command has completed or that it executed with an error. An "input" commands is used in either case.

Note: We need to include a command to let the kernel request a full input-link dump be sent over.

5.7.1. Example of this tag

```
<sml smlversion=”1.0” doctype=”call” soarVersion=”8.4.2” id=”1234”>
<command name=”input”>
        <arg param=”agent”>agent-1<.arg>
        <wme action=”add” att=”plane” id=”i1” tag=”-5” type=”id” value=”a4”></wme>
        <wme action=”add” att=”name” id=”a4” tag=”-6” type=”string” value=”plane1”></wme>
        <wme action=”add” att=”speed” id=”a4” tag=”-7” type=”int” value=”225”></wme>
        <wme action=”add” att=”position” id=”a4” tag=”-8” type=”id” value=”a5”></wme>
        <wme action=”add” att=”x” id=”a5” tag=”-9” type=”int” value=”65”></wme>
        <wme action=”add” att=”self” id=”a4” tag=”-10” type=”id” value=”a4”></wme> ; Note – same value
as id
</command>
</sml>
```

Input Element	Description
Tag: input **Optional in an SML document**	Tag for input node. Has sml node as its parent.
Attribute: update=”full | delta” **Optional**	Whether the WM description is a complete description of the input-link or how it should be modified from its current state. Defaults to “delta” if not specificied. This flag is not yet in use (all messages are deltas).
Children	List of WMEs to add or remove. For a full update, the WMEs must start at the input-link.
Character Data	None.

5.8. Output tag

This element is used to describe a change to the output-link and follows the same format as the input command.

The output tag is a parameter to an “output” command.

```
<sml smlversion=”1.0” doctype=”call” soarVersion=”8.4.2” id=”1234”>
<command name=”output”>
        <arg param=”agent”>agent-1</arg>
        <wme action=”add” att=”turn” id=”O1” tag=”7” type=”id” value=”T1”></wme>
        <wme action=”add” att=”heading” id=”T1” tag=”6” type=”int” value=”045”></wme>
        <wme action=”add” att=”speed” id=”T1” tag=”12” type=”int” value=”225”></wme>
</command>
```

```
</sml>
```

An output command informs the client that the Soar agent has added new structure to the output link. The client will then examine that new structure and determine what action the agent has taken and how to change the environment accordingly. Each output command reports just the changes to the output-link since the last time output was sent.

Output Element	Description
Tag: output **Optional in an SML document**	Tag for output node. Has sml node as its parent.
Attribute: update="full | delta" **Optional.**	Whether the WM description is a complete description of the output-link or how it has just changed.
Children	List of WMEs that have just been added or removed by the Soar agent. For a full update, the WMEs must start at the output-link.
Character Data	None.

5.9. Working Memory Element tag

This element is used to describe working memory elements, for example as output from a print command. A WME is represented as a triplet: (identifier, attribute, value) together with a time tag (a unique number for this specific working memory element).

5.9.1. Example of this tag

```
<wme action="add" att="plane" id="i1" tag="-5" type="id" value="a4"></wme>
<wme action="add" att="name" id="a4" tag="-6" type="string" value="plane1"></wme>
<wme action="add" att="speed" id="a4" tag="-7" type="int" value="225"></wme>
<wme action="add" att="position" id="a4" tag="-8" type="id" value="a5"></wme>
<wme action="add" att="x" id="a5" tag="-9" type="int" value="65"></wme>
<wme action="add" att="self" id="a4" tag="-10" type="id" value="a4"></wme> ; Note – same value as id
```

This example represents: (I1 ^plane P4) (P4 ^name plane1 ^speed 225 ^position P5 ^self P4) (P5 ^x 65). Note that the IDs created in Soar's working memory are different from those assigned by the client ("a4"->"P4"), but the structure is maintained correctly.

5.9.2. Definition

wme Element	Description
Tag: wme	Tag for wme node.
Attribute:id=<id of parent> **Required (unless referencing an existing wme by time tag)**	The ID for the parent of this WME.
Attribute: att=<attribute-name> **Required (unless referencing an existing wme by time tag)**	The attribute name of this wme.
Attribute: tag=<n> where n is a number. **Required if adding a wme to working memory.**	Used to report Soar's internal time tag for this WME. The time tag is sufficient to uniquely identify this WME to Soar and so may be more efficient to use in some situations. A positive value is a time tag that has meaning to the kernel. A negative value is a time tag that only has meaning to the client.
Attribute: type="string" | "int" | "double" | "id" **Optional. Defaults to string.**	Specifies how the value should be interpreted.
Attribute: value=<value> **Required.**	The value for this wme. If the value is an identifier then if the identifier was created by the client it will start with a lower case letter (e.g. "g3"). If the identifier was created by the kernel, it will start with an upper case letter (e.g. "K3").
Attribute: action="add | remove" **Optional.**	Only relevant when used in a command that changes working memory. Add is the default and indicates that this WME should be added. Remove indicates that this WME should be removed. For a remove command, only the time tag is required. (Specifying a unique WME through (id, attribute, value) is not possible as multiple wmes might share these same values, so the time tag must be used).
Children	None
Character Data	None

5.10. Condition tag

This element represents a condition in a production (for example (state <s> ^super-state <ss>)). It can be used to represent either a single condition or a series of conditions that share the same ID (e.g. state <s> ^name top ^color red).

5.10.1. Example of this tag

```
<production name="chunk-1">
<condition-set negated="false">
        <condition negated="false">
                <id-test> <test>
                                <symbol type="variable">&lt;s&gt;</symbol>     [id is test for <s>]
                </test> </id-test>
                <att-val-test>
                        <att-test> <test>
                                <symbol type="string">name</symbol>            [att is ^name]
                        </test> </att-test>
                        <val-test> <test>
                                <symbol type="string">Top-State</symbol>       [value is |Top-State|]
                        </test> </val-test>
                </att-val-test>
                <att-val-test>
                        <att-test> <test>
                                <symbol type="string">height</symbol>          [att is ^height]
                        </test> </att-test>
                        <val-test> <test-set>
                                <test>
                                <symbol type="int">52</symbol>                 [value is 52]
                                </test>
                                <test>
                                <symbol type="int">55</symbol>                 [or value is 55]
                                </test>
                        </test-set> </val-test>
                </att-val-test>
        </condition>
        [More conditions go here]
</condition-set>
</production>
```

This condition represents (<s> ^name |Top-State| ^height << 52 55 >>). Productions are complicated beasts, so there is a lot of structure here to cover the different possibilities.

Condition Element	Description
Tag: condition	Tag for condition node.

Attribute: negated="true" | "false" **Optional**	Indicates if this is a negated condition or not. If omitted the condition is not negated.
Children **Required**	One <id-test> and series of <att-val-test> nodes
Character Data	None

5.10.2. ID test tag

This tag represents the test for the identifier in a condition (e.g. <s> in (<s> ^name top-state)).

Id-test Element	Description
Tag: id-test	Tag for the id-test node. Has a condition element as its parent.
Children **Required**	Either <test> or <test-set> node.
Character Data	None

5.10.3. Att-val-test tag

This tag represents the test for an attribute and value in a condition (e.g. ^name top-state).

Att-val-test Element	Description
Tag: att-val-test	Tag for the att-val-test node. Has an id-test element as its parent.
Children **Required**	One <att-test> node and one <val-test> node. The first is for the attribute, the second for the value.
Character Data	None

5.10.4. Att-test tag

This tag represents the test for the attribute in a condition (e.g. name in (<s> ^name top-state)).

Id-test Element	Description
Tag: att-test	Tag for the att-test node. Has an att-val-test element as its parent.
Children **Required**	Either <test> or <test-set> node.
Character Data	None

5.10.5. Val-test tag

This tag represents the test for the value in a condition (e.g. top-state in (<s> ^name top-state).

Id-test Element	Description
Tag: val-test	Tag for the val-test node. Has an att-val-test element as its parent.
Children **Required**	Either <test> or <test-set> node.
Character Data	None

5.10.6. Test tag

This element represents a test in a condition.

Test Element	Description
Tag: test	Tag for the test node. Has either an id-test or an att-val-test as its parent (or possibly a test-set).
Attribute: type=”eq” | “gt” | “lt” | “gte” | “lte” | “ne” | “dis” | “con” **Optional**	Equal, greater than, less than, greater than or equal, less than or equal, not equal, disjunction, conjunction. If the field is omitted the test type is “eq”.

Children **Required**	A <symbol> node that represents the value being tested against.
Character Data	None

5.10.7. Test Set tag

A single test in a condition can actually be a test against a set of values (e.g. ^name << a b >>).

Test-set Element	Description
Tag: test-set	Tag for the test-set node. Has either an id-test or an att-val-test as its parent (or possibly a test or test-set).
Children **Required**	A series of <test> nodes (or I believe even <test-set> nodes are possibly valid).
Character Data	None

5.10.8. Symbol tag

This element represents a single symbol (an integer, a variable name etc.).

symbol Element	Description
Tag: symbol	Tag for the symbol node. Has a test element as its parent.
Attribute: type=”string” | “int” | “double” | “char” | “boolean” | “id” | “variable” **Optional**	The type of this symbol. If this field is omitted the type is “string”.
Children	None
Character Data	The value

5.10.9. Condition set tag

This element represents a grouped set of conditions. Productions can contain negated sets of conditions that are logically grouped together.

Condition-set Element	Description
Tag: condition-set	Tag for the condition-set node.
Attribute: negated="true" | "false" **Optional**	Indicates if this set of conditions are negated or not. If omitted the set of conditions is not negated.
Children **Required**	A series of <condition> nodes (or I believe <condition-set> nodes are also valid).
Character Data	None

5.11. Action tag

TODO: Need to specify similar things to the condition side.

5.12. Production tag

This element is used to represent a production. Interestingly, this XML structure exposes the Soar parser quite nicely. You could send a production as a raw string to Soar and get this back as output—a fully parsed and structured version of a Soar production.

Need to add the "state" symbol.

5.12.1. Example of this tag

```
<production name="chunk-1" source="c:\test\prods.soar" docs="My first production" type="chunk">
<condition-set negated="false">
        <condition negated="false">
                <id-test> <test>
                                <symbol type="variable">&lt;s&gt;</symbol>     [id is test for <s>]
                </test> </id-test>
                <att-val-test>
                        <att-test> <test>
                                <symbol type="string">name</symbol>            [att is ^name]
                        </test> </att-test>
                        <val-test> <test>
                                <symbol type="string">Top-State</symbol>       [value is |Top-State|]
                        </test> </val-test>
                </att-val-test>
                <att-val-test>
                        <att-test> <test>
                                <symbol type="string">height</symbol>          [att is ^height]
                        </test> </att-test>
                        <val-test> <test-set>
                        <test>
                        <symbol type="int">52</symbol>          [value is 52]
                        </test>
                        <test>
                        <symbol type="int">55</symbol>          [or value is 55]
                        </test>
                </test-set> </val-test>
            </att-val-test>
        </condition>
        [More conditions go here]
</condition-set>
<action-set>
        [Actions go here]
</action-set>
</production>
```

The condition represented here is (<s> ^name |Top-State| ^height << 52 55 >>).

Production Element	Description
Tag: production	Tag for the production node.
Attribute: name=<production-name> **Required**	The name of the production
Attribute: type="user" | "chunk" | "justification" | "default" **Required**	What type of production this is.
Attribute: docs=<documentation-string> **Optional**	The user's documentation of this production.
Attribute: source=<file-path> **Optional**	Path to the file from which this production was loaded.
Children **Required**	A <condition-set> node and an <action-set> node.
Character Data	None

5.13. Tags still to do

Need to support output from some functions such as matches, find etc. that return list of production names or match condition information etc. We'll probably end up defining these as we work through the command set. Once we have the basic structure, it should be simple enough to build up from it for these commands.

6. Overview of Client and Kernel Interfaces

6.1. Introduction

So far this spec has focused on the question of what will be communicated between a client (tool or simulation) and the Soar kernel at the XML level. To build a complete system, we also need to define the higher level interface that manages this communication on either the Soar kernel or client side. This interface is responsible for managing sockets, sending commands, receiving notifications, registering for events etc.

The basic structure for this higher level interface is shown below in Figure 4.

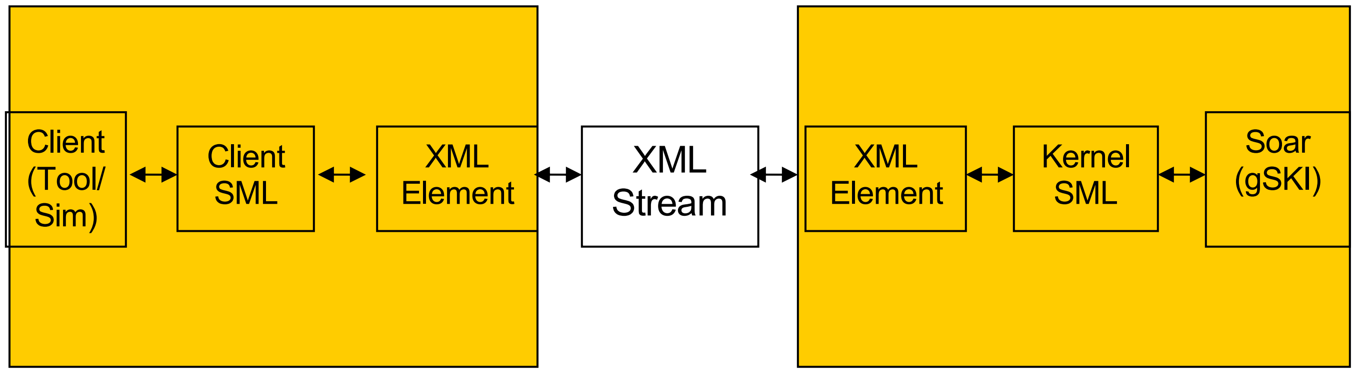

Figure 4 ClientSML and KernelSML interfaces

6.2. Kernel SML

The KernelSML component is responsible for:

- converting incoming commands into calls to the Soar kernel (gSKI)
- converting output from the Soar kernel into calls to the client
- accepting input wmes for Soar's input-link and sending output-link commands to the client
- allowing the client to register and un-register for events that occur in the kernel and sending that information to the client when the event occurs.
- supporting socket connections; synchronous and asynchronous calls and in-process communication

All of the information passed to and from Kernel SML will be encoded as SML messages and all SML messages sent to Soar will be interpreted by this layer. KernelSML will be a shared library (a DLL on Windows) and will be linked directly to a particular version of gSKI and Soar. The interface to this library should not change over time as Soar changes and evolves, making it possible for a tool to be designed to work with many different versions of Soar.

6.3. Client SML

The ClientSML component is a layer that clients may choose to use to make them simpler. However, it is not a requirement of the system that a client uses the ClientSML library and some tools or simulations may find it more convenient to simply generate XML directly or use the ElementXML library directly.

The ClientSML component is responsible for:

- sending input WMEs to Soar (via SML and the KernelSML layer) that are then attached to the input-link
- responding to output commands from Soar that are generated from the output-link
- supporting socket connections; synchronous and asynchronous calls and in-process communication
- providing some useful classes for constructing SML messages

The ClientSML library will be much smaller than the KernelSML library as it will not need to interpret the range of commands that can be sent and received by SML. The client itself is responsible for handling those messages in a manner that makes sense for the particular application.

Over time we will want to provide ClientSML implementations in a number of languages, notably C++ and Java and possibly others.

6.4. Multiple Clients, Multiple Kernels

While the base model shows a single client talking to a single kernel, in general we want to allow multiple clients to talk to a single kernel (e.g. Visual Soar and a Debugger might both wish to send information to an instance of Soar) and a single client to talk to multiple kernels (e.g. in a distributed setting we might run multiple instances of Soar on different machines and wish to control them all from a single control panel).

Furthermore, it would be good to allow, at the communication level, clients to talk to each other (e.g. one tool to another or a tool to a simulation) and to allow kernels to communicate with each other (e.g. to support inter-agent communication). This form of inter-kernel communication or inter-tool communication would require different SML message content than has been defined in this spec but that should be all.

One more complexity here is that to be truly general, we should allow Soar to be embedded in one tool and yet accessible by another tool through a socket. We won't focus on supporting this capability at the start, but by keeping it in mind I expect we can achieve it without extra effort if we're careful in the design.

7. Client SML and Kernel SML Interfaces

This section still needs to be updated to match the latest method definitions.

7.1. Connection Interface

This interface describes the connection between a client (tool or simulation) and the Soar kernel. Both ClientSML and KernelSML will use this connection interface to communicate with each other.

The functions here are described from the client's perspective and would be completely symmetrical in KernelSML from the kernel's perspective.

Function	Description
Connection* CreateEmbeddedConnection()	Creates a connection to the Soar kernel that is embedded within the same process as the client.
Connection* CreateRemoteConnection(char const* pIPaddress, int port)	Creates a connection to a different process. This could be on the same machine or on a different machine. If memory serves, the IP address to connect on the same machine should be 127.0.0.1. Need to confirm that. We'll allow the user to pass NULL for the address to indicate it's a local connection.
ListenerConnection* CreateListener(int port, ListenerCallback callback, void* pUserData) ;	Allows an external process to connect to us on the specified port. When a connection is made, the callback function is called and passed the user data.
void CloseConnection(Connection *pConnection)	Shutdown this connection.
void SendMessage(ElementXML* pMsg)	Send a message to the kernel (usually a command or input message). The connection records information about who we are talking to (what socket etc.). This should allow a client to talk to multiple kernels and we should certainly support clients talking to other clients. The error code that is returned indicates whether the command was successfully sent, not whether the command was interpreted successfully by Soar.
void ReceiveMessages(bool allMessages)	Retrieve any commands, notifications, responses etc. from the kernel. If "allMessages" is false processes at most one message. Messages that are received are routed to callback

	functions in the client for processing. This call never blocks. In an embedded situation this does nothing as messages from the kernel are sent directly to the callback functions. In a remote situation, the client must call this function periodically. We use a callback model (rather than retrieving each message in turn here) so that the embedded model and the remote model look the same to the client.
ElementXML* GetResponse(char const* pID, bool wait)	Retrieve the result of the last command sent. In an embedded situation this result is always immediately available and the "wait" parameter it ignored. In a remote situation if "wait" is true, this call will block waiting for the response (and will eventually timeout if no response comes in). If "wait" is false, the call returns NULL if the response is not available immediately. The ID is only required in the situation where the client is remote, sends a series of commands and then asks for the result of the last one. Without the ID it could not discriminate among the series of responses. This function will return NULL if called a second time for the same ID (i.e. the message can only be retrieved once) which just allows us to improve performance (by not keeping a message around longer than necessary). The client is not required to call to get the result of a command it has sent. The implementation of GetResponse() will call ReceiveMessages() to get messages one at a time and process them. Thus callbacks may be invoked while the client is blocked waiting for the particular response they requested. A response that is returned to the client through GetResponse() will not be passed to a callback function registered for response messages. This allows a client to register a general function to check for any error messages and yet retrieve specific responses

	to calls that it is particularly interested in.
void RegisterCallback(Function *pFunc, void* pUserData, char const* pType, bool addToEnd) **Function type accepts an ElementXML* and returns an ElementXML*.**	Register a callback function for a given type of message. When a message is received of the given type (based on the doctype attribute in the SML node) the callback function is called. If a message is returned by the callback it will be sent back to the kernel (and must be a "response"). We will maintain a list of callbacks for a given type of SML document and call each in turn. The first callback which returns a non-NULL response will be sent back to the kernel and no further callbacks will be called for that message. This ensures that only one response is generated for each incoming message. If "addToEnd" is true then the callback is added to the end of this list (i.e. will be called after all existing callbacks). If false, the callback is added to the front of the list. The user data parameter is passed to the callback function, allowing the caller to maintain context within the callback.
void UnregisterCallback(Function *pFunc, char const* pType)	Removes an existing callback from the list of callbacks. If pFunc is NULL remove all callbacks for this type of message.
ErrorCode GetLastError() ;	Reports any errors in the last function call made to this class. 0 indicates no errors. A user displayable error message can be had from GetErrorDescription(ErrorCode n) ;

I think KernelSML can be completely symmetric with ClientSML as regards message passing.

7.2. Sample code for an embedded connection

7.2.1. Initialization

First, create the connection object:

```
ErrorCode error = 0 ;
Connection* pConnection = Connection::CreateEmbeddedConnection(&error) ;
```

This example assumes that the client is interested in receiving notifications of events from the kernel, so it registers a notification callback. The callback function must itself be static, but for C++ code we'll typically want to handle the call within an object. To do this we register the function and pass in the "this" pointer:

```
pConnection->RegisterCallback(Notify1, this, sml_Names::kDocType_Notify, true) ;
```

Then the callback function looks like this:

```
static ElementXML* Notify1(Connection* pConn, ElementXML* pMsg, void* pUserData)
{
    // Switch from a static callback into one local to the sending object
    ClientClass* pMe = (ClientClass*)pUserData ;
    pMe->Notify(pConn, pMsg) ;

    // Since this is a notification, we always return NULL
    return NULL ;
}

void Notify(Connection* pConnection, ElementXML* pIncoming)
{
    // This is where we really handle the notification.
}
```

7.2.2. Usage

Commands are sent like this:

```
pConnection->SendMessage(pSML) ;
```

The response to a command is retrieved like this:

```
ElementXML* pResponse = pConnection->GetResponse(pSML->getID(), true) ;
```

Notifications of events occurring on the kernel (e.g. when Soar is interrupted for some reason or watch output while Soar is running) are received by the Notify function being called.

7.3. Sample code for a remote connection

7.3.1. Initialization

The only difference from the embedded example during initialization is how the Connection is created:

```
ErrorCode error = 0 ;
Connection* pConnection = Connection::CreateRemoteConnection("152.12.52.1", 572,
&error) ;
```

The methods for registering a callback are the same as for the embedded connection.

7.3.2. Usage

The only difference from the embedded example is that ReceiveMessages() must be called periodically.

So we have:

Commands are sent like this:

```
pConnection->SendMessage(pSML) ;
```

The response to a command is retrieved like this:

```
ElementXML* pResponse = pConnection->GetResponse(pSML->getID(), true) ;
```

Notifications of events occurring on the kernel (e.g. when Soar is interrupted for some reason or watch output while Soar is running) are received by the Notify function being called which will only happen as a result of either the GetResponse() call above or as a result of calling:

```
pConnection->ReceiveMessages(true) ;
```

7.4. MessageGenerator Interface (now folded into Connection class)

This class provides helper functions for creating SML messages. The client is not required to use these functions and can create ElementXML objects directly through that interface, but usually these will be helpful. The Connection class derives from the MessageGenerator class, so a Connection is always a MessageGenerator.

Function	Description
Int GenerateID()	Generates a new ID that is unique over the life of this generator.
ElementXML* CreateSMLMessage(char const* pType)	Creates an SML message with id, docType, smlVersion, soarVersion attributes defined. The type passed in should currently be one of “call”, “response” or “notify”.
ElementXML* CreateSMLCommand(char const* pAgent, char const* pCommandName, bool rawOutput) ;	Creates an SML message containing <agent> and <command> tags. After creating this object, you should add parameters to the command. If rawOutput is true then the result of the command will be a string wrapped in a <raw> tag, rather than full structured XML. (See the <raw> tag for more on this).
void AddParameterToSMLCommand (ElementXML* pCommand, char const* pName, char const* pValue, char const* pValueType) ;	Adds a parameter to an SML message that contains a single command object.
ErrorCode GetLastError() ;	Reports any errors in the last function call made to this class. 0 indicates no errors. A user displayable error message can be had from GetErrorDescription(ErrorCode n) ;

8. Communication Models for I/O

This whole section was useful during development, but is probably just confusing now. It should come out and be replaced by a simpler description of the different types of connections.

8.1. Simple Communication (Soar not executing)

When Soar is not executing, commands can be sent from the client to the kernel and the results are passed back, making for a simple communication model.

8.1.1. Embedded print "chunk-1" example

Here's an example showing the sequence of calls for an embedded client calling to the kernel to print a chunk.

Client	Client SML	Kernel SML	Notes
User command: print chunk-1			
Build SML command msg1 type="command"			
→	**SendMessage(msg1)**		This is a function call that will not return until kernel responds.
	→	**Call gSKI to get chunk-1 representation**	
		Convert production to SML msg2 type="response"	
		Return msg2 as result of "SendMessage"	This result is cached briefly until the GetResponse().
	ClientSMLretrieves result with **GetResponse(msg1)**	←	This call is just to maintain symmetry with the remote model (where a result is not immediately available).
Client is passed SML description of chunk-1	←		

8.1.2. Remote print "chunk-1" example

Here's an example showing the sequence of calls for a remote client calling to the kernel to print a chunk through a socket.

Client	Client SML	Kernel SML	Notes
User command: print chunk-1			
Build SML command msg1 type="command"			
→	**SendMessage(msg1)**		This function returns as soon as the message has been sent over the kernel.
	GetResponse(msg1)		Blocks waiting for kernel
		ReceiveMessages() **Calls to callback for handling commands.**	
		Call gSKI to get chunk-1 representation	
		Convert production to SML msg2 type="response"	
		SendMessage(msg2)	This function returns as soon as the message has been dispatched.
	GetResponse(msg1)		This call now unblocks and receives the response.
Client is passed SML description of chunk-1	←		

8.2. Input and Output during Soar execution

When Soar is executing together with a simulation, the relationship can be much more complex. These are the models I can see that we might wish to support:

	Embedded	Remote

Synchronous controlled by Soar **Soar generates output which changes the environment which is then sent to Soar as input. Environment only updates when Soar issues a command.** **Soar is the executable and the environment is loaded as a library in this case.**	It is not clear that this mode is very important or that we would ever load the environment as a library into Soar.	
Synchronous controlled by Client **Simulation runs Soar for a fixed amount of time during which Soar may receive input and generate output. Soar may issue multiple outputs or none during a given period and environment may update independently of Soar's actions (although not while Soar is reasoning).** **The simulation is the executable and Soar is loaded as a library in this case.**	This is the mode currently used in most simulations in either embedded form or remote form.	This is the mode currently used in most simulations in either embedded form or remote form.
Asynchronous **Both run as independent processes. Input is sent to Soar from the simulation whenever the environment changes.** **Output is sent from Soar to the simulation whenever Soar decides to act.** **Any synchronization will occur within the agent as it waits for events to happen before it takes action.** **Both Soar and the simulation are executables in this case.**	We do not plan to support asynchronous behavior for embedded processes. In order to do this, the environment and Soar would need to run in separate threads. If a user wishes to do this, they could use the remote model to send data over a socket between the two threads.	This is another popular mode for large scale simulations where Soar is just one of many actors in the environment.

8.2.1. Remote example – asynchronous model

Here's an example showing the sequence of calls to send over the current state from a simulation to the kernel and run for a while (processing I/O along the way). In this example, both processes run independently and communicate asynchronously. This model is appropriate when there are multiple agents in the world and each act in "real time" in the simulation.

Client	Client SML	Kernel SML	Notes

Build SML command msg1 type="input"			We'll have functions to help hide the details of the SML (so the client can work in turns of WMEs).
	SendMessage(msg1)		Returns immediately.
Simulation continues executing. In this model, may register a callback for responses and use that to check that the input message was handled correctly.			
		Assume Soar is not running at this point. **ReceiveMessages()**	We'll poll ReceiveMessages() periodically when Soar is not running.
		Convert msg1 into WMEs and add them to the input-link	For this example we'll ignore the caching issues and deciding what to add.
		Create response msg2 to indicate success.	
		SendMessage(msg2)	Because this is a response message, there is no reply to it, so we can't get stuck in an infinite sequence of responses to each other.
Now run <n> decisions to let Soar process the input.	**Build msg3: run <n> decisions.** **SendMessage(msg3)**		This command can be sent before Soar has processed the input command. That's not a problem. It returns immediately.
		ReceiveMessages()	
		Create response (msg4) to let simulation know call to run Soar succeeded. **SendMessage(msg4)**	

		Start running Soar.	
		Soar generates output. Create msg5 containing output actions.	
		SendMessage(msg5)	Returns immediately.
	ReceiveMessages() **(called periodically)**	**Soar continues executing.**	
Simulation updates to reflect agent's actions.	**Processes action from msg5.**		
Let's assume this action will take a long time to execute.	**Create response msg6 to indicate action was received and is now executing.** **SendMessage(msg6)**		
		ReceiveMessages() called. Soar is now aware that the output actions were successfully received by the simulation.	Response could be just "got your actions, thanks" or contain an <output> tag which is used to add, e.g. ^status executing to the output-link.
Action completes.	**Create output command msg7 to indicate action completed (and succeeded or failed)** **SendMessage(msg7)**		
		ReceiveMessages() to get msg7. During output-phase, Soar adds the result of the action to the output-link.	Details of what are added depend on what client sends in msg7. This is defined by the simulation, not by SML.
At some later point, the simulation will decide to send new input to Soar.	**Build new input state, msg8.**		
	SendMessage(msg8)		
	Simulation keeps running.	**ReceiveMessages() called during input-phase.**	

		Result of output command is now known and used to update output-link with result.	
		New input received and updates the input-link	
		Create response msg9 to indicate success for adding input. **SendMessage(msg9)**	
		Soar keeps executing.	

8.2.2. Embedded example – synchronous with client in control

Here's an example showing the sequence of calls to send over the current state from a simulation to the kernel and run for a while (processing I/O along the way).

Note: We are using a method here where the environment pushes the current state over to the kernel once before executing Soar for n-decisions. Thus the environment is not given the opportunity to update until Soar stops running. The feeling is that this is usually the correct model for these synchronous simulations. In order to have the environment update more rapidly, the client can choose to run Soar for just 1 decision at a time, in which case Soar ends up checking for changes in the environment every decision (rather than every n'th decision).

Client	Client SML	Kernel SML	Notes
Build SML command msg1 type="input"			We'll have functions to help hide the details of the SML (so the client can work in turns of WMEs).
→	**SendMessage(msg1)**		This function doesn't return until Soar has processed and responded to the message.
	→	**Convert msg1 into WMEs and add them to the input-link**	For this example we'll ignore the caching issues and deciding what to add.
		Create response msg2 to indicate success.	

		Return msg2 as result of "SendMessage"	
	GetResponse(msg1)	←	This call is just to maintain symmetry with the remote model (where a result is not immediately available).
Simulation has added initial input.	←		
Now run <n> decisions to let Soar process the input.			We should let the client run Soar any way they like: Run <d> decisions Run <t> seconds Run to next output or <d> decisions which is less Run forever etc.
Build SML command msg3 "run <n> decisions"			
→	**SendMessage(msg3)**		This function won't return until Soar stops running. This means that all of the message handling code must be fully re-entrant (no statics etc.).
	→	**Start running Soar.**	
		Soar generates output. Create msg4 containing output commands.	
		SendMessage(msg4)	This function will block until after the simulation has sent new input back to us.
	Callback to client through output callback function.	←	
Simulation updates to reflect agent's actions.	←		

This creates new input which should be returned to the agent.			
Build new input state: msg5			
	SendMessage(msg5)		
	→	**New input received and will update the input-link in next input phase.**	Changes cached until input phase.
		Create response msg6 to indicate success.	
		Return msg6 as result of "SendMessage"	
	GetResponse(msg6)	←	Check that input succeed-ed
Simulation has sent input. Can now return from initial output call from kernel.	←		
	Create response msg7 to indicate success of output call (msg4)		If the action will not com-plete immediately, msg7 could indicate this and then on a later call to the environment it would send an output message to signal that the action has now completed. This model is probably less common in situations where the simulation and Soar are running synchro-nously but it is still sup-ported.
	Return msg7 as result of "SendMessage"		
	→	Soar has completed the output phase and now continues running. It will receive the input from msg5 on the next input phase.	

This method ends up looking a lot like the embedded model where Soar is in control. I think the only difference is how long to run Soar for. If you run it forever, then this is the “Soar in control model”. If you run it for a fixed amount of time then it returns control to the simulation on a regular basis, which can then update independently of the Soar agent. Seems like this is the only difference.

9. SML Command Language

Up to this point, this spec has detailed the way information is passed between a client and the kernel. In this section, we will define the details of commands that can be sent to the kernel and the output it will generate.

1) All commands will be enclosed in a <command> tag.

2) All responses will either be enclosed in a <result> tag or an <error> tag will be returned. In the descriptions below these tags will usually be omitted to save space. If no failure condition is explicitly defined, then an error will just return an <error></error> tag.

3) The tags are shown to indicate the structure of the command or response. The tags themselves will often contain attributes or character data that is defined above (where the tag is defined). Those details are omitted here for simplicity.

This list needs to be completely updated as it used to refer to gSKI commands and those have all been replaced.

9.1. Agent Commands

9.1.1. create-agent

Type	Definition	Notes
Command	**“create-agent”**	Used to create a new agent.
Parameters	**name=<agent-name> (req)**	
Success	**<result><name>agent-name</name></result>**	The name of the newly created agent is returned.
Error	**<error>Why this call failed</error>**	Explanation for why. We’ll omit this field in most command definitions unless unusual error values are supported.
Description	Creates a new agent.	