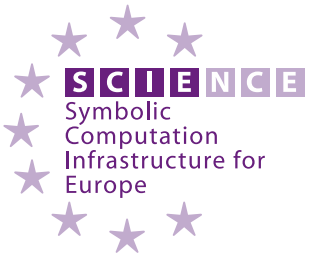

S. Freundt, P. Horn, A. Konovalov,
S. Linton, D. Roozemond

Symbolic Computation Software Composability Protocol

SCSCP

Specification, Version 1.3

March 27, 2009

The project 026133 "SCIEnce - Symbolic Computation Infrastructure in Europe"
is supported by the EU FP6 Programme

Contents

1. Introduction

This document specifies the requirements for the software to be developed by the NA3 activity of the SCIEnce project for the subsequent usage in the NA3 and JRA1 activities.

Specifically it describes a protocol by which a CAS may offer services and a client may employ them. This protocol is called the *Symbolic Computation Software Composability Protocol*, or **SCSCP**. We envisage clients for this protocol including:

- A Web server which passes on the same services as Web services using SOAP/HTTP to a variety of possible clients;
- Grid middleware;
- Another instance of the same CAS (in a parallel computing context);
- Another CAS running on the same system.

Note that the specification assumes two possible ways of implementation. One is the standard socket-based implementation, where a CAS can talk locally or remotely via ports, as described in the section 5, in fact an SCSCP service rather than a Web service. The other implementation is a proper Web service using standard SOAP/HTTP wrappings for SCSCP messages. In the SCSCP specification we will use the term *Web services* in the broad sense, meaning both kinds of symbolic computation services.

Our vision of the SCSCP usage is described in the following scheme.

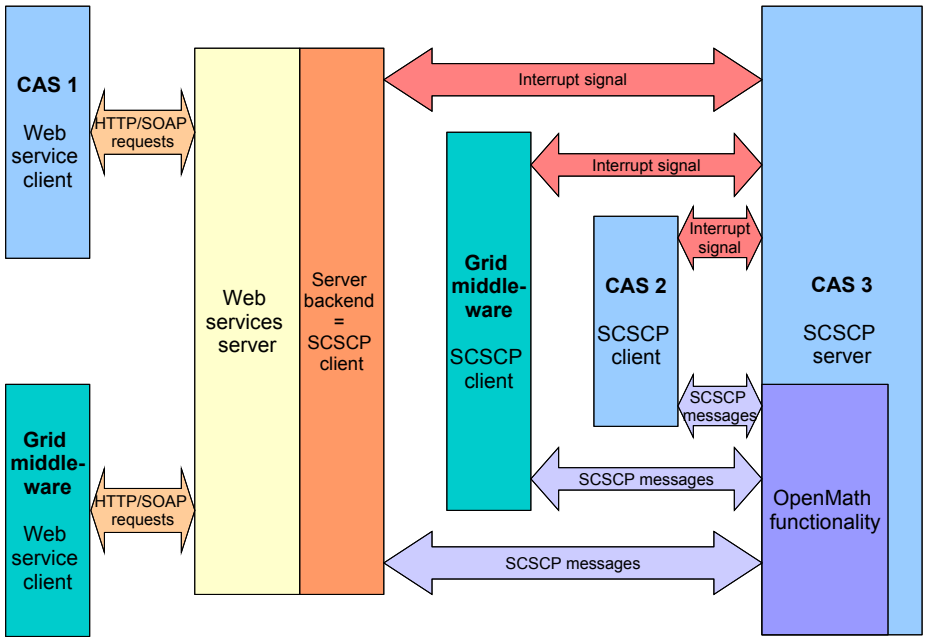

Figure 1. SCSCP usage.

In Section 2 we describe the meaning of the possible messages that can appear during communication between various software and the allowed sequences of such messages. In Section 4 we specify how these messages are encoded as OpenMath

objects. Finally in Section 5 we describe one solution (suitable for UNIX systems, at least) for the practical problems of establishing a connection and delivering these messages.

In Appendix A we specify the list of necessary OpenMath symbols. Examples of OpenMath messages are given in Appendices B and C. More examples may be found in the `scscp1` and `scscp2` OpenMath Content Dictionaries: [1], [2].

List of abbreviations used in the document:

CAS: Computer Algebra System
CD: Content Dictionary
OM: OpenMath
PI: processing instruction (in XML)
SCSCP: Symbolic Computation Software Composability Protocol
WS: Web Service
WSDL: Web Services Description Language

2. Semantic descriptions

2.1. **Messages from client to server.**

2.1.1. *Procedure call.* This is an OM message containing the following information:

- *Procedure name* - the name of the procedure registered as a web-service;
- *Arguments* - arguments that will be passed to the procedure being called; (Remark: we treat procedure options, such as guidance options for used algorithms, as arguments as well);
- *Options/Attributes* - attributes and options that specify the expected behaviour of the system. Mandatory options are:
  - call identifier (unique in the scscp session);
  - type of action performed with the result (precisely one must be given)
    - storing the result at the server side and returning a cookie referring to it;
    - returning the result of the procedure (that may involve the actual computation or the retrieval of a previously stored result);
    - not returning the result itself, only a message indicating whether the procedure completed successfully or not;

  Non-mandatory options are, for example:
  - procedure runtime limit;
  - minimal/maximal memory limits;
  - debugging level, determining degree of output detail;
  - other options that might appear during the development; may be system dependent.

There are some standard procedures predefined in Section 3. Besides this, in order to provide specific SCSCP services, the service provider should develop and make available their own customized procedures, which may range from generic services (e.g. we may imagine some procedure named like EVALUATE_OMOBJ to evaluate given OM object) to specific (e.g. compute the determinant of a matrix).

2.1.2. *Interrupt signal.* This signal implies that the results of one particular computation are not needed, so the CAS need not complete the computation. After receiving an interrupt signal, the server must, as always, reply to the procedure call

with the mentioned call ID. It must still reply to the procedure calls in the order they were received in. For example, if the client sends procedure calls A,B,C and then an interrupt signal for B while the server is still computing A, the server must still reply to A before it sends a procedure terminated message for B.

Note that it is always correct for a CAS to ignore an interrupt, and thus simply reply with a procedure completed message. This may be appropriate when procedure calls are quick.

Note also that in earlier versions of the protocol the Interrupt signal was implemented using POSIX signals, but from version 1.3 onwards the signal is implemented by the `<?scscp terminate?>` processing instruction. See Section 5.3.

2.2. **Messages from server to client.**

2.2.1. *Procedure completed.* This is an OM message containing the information about the result of the procedure:

- *Result value* - if the procedure returns a result, it must be contained in this section of the message. If the procedure only produces side-effects, such a section is not necessary since this message itself acts as a signal about its successful completion;
- *Mandatory additional information:*
  - call identifier;
- *Optional additional information:*
  - procedure runtime;
  - memory used;
  - other information that we might need (may be system dependent).

2.2.2. *Procedure terminated.* This is an OM message that acts as a signal about procedure termination.

- *Error*: in its body, the message must contain an appropriate error, as described for example in the `scscp1` OpenMath Content Dictionary.
- *Mandatory additional information:*
  - call identifier;
- *Optional additional information:*
  - procedure runtime before termination;
  - memory used;
  - other information that we might need (may be system dependent).

2.3. **Allowed sequences of messages.** Once a connection has been established and any initial technical information exchanged (the mechanism for which is part of an implementation of this protocol and addressed in Section 5) the SCSCP session is considered to be opened.

Until the end of the session, the communication from client to server is a stream of procedure calls and the response is a stream of procedure completed and/or procedure terminated messages. The client is permitted to send as many procedure calls as it likes, subject to the buffering capabilities of the channel used in a particular implementation. The server must process them in sequence and send *either* a procedure completed, *or* a procedure terminated message (but not both) for each procedure call received.

Apart from this, the client can send an interrupt message to the server. The interrupt message can be sent to the server at any time. It entitles the server

to stop processing one procedure call and respond to it with a suitable procedure terminated message.

3. Special Procedures

This section documents certain predefined procedures which every compliant client is expected to support. OpenMath symbols corresponding to these procedures are defined in the `scscp2` Content Dictionary [2].

3.1. **Determining the list of supported procedures.**

3.1.1. *Representing Collections of OpenMath Symbols.* A number of the procedures defined in this section return values which are intended to represent sets of OM symbols. For convenience, we define a standard way of representing these sets.

Such sets should be represented as applications of the symbol `symbol_set` which may take any number of arguments. Each of those arguments should be one of the following:

(1) An OpenMath symbol, representing itself,
(2) An application of one of symbols `CDName` or `CDURL` from the `meta` CD, representing all the symbols in the referenced CD;
(3) An application of one of the symbols `CDGroupName` or `CDGroupURL` from the `metagrp` CD; representing all the symbols in all CDs in the referenced CD group.

As an alternative, the symbol `symbol_set_all` can be used, to represent the set of all OpenMath symbols from any CD.

3.1.2. *Transient CDs.* In describing its allowed procedure calls according to the conventions of this section, a server is permitted to refer to symbols from CDs with names beginning `scscp_transient_` (note the lower case). These are content dictionaries defined by this server, and valid only for the duration of the session. If needed, the client can request these CDs using the `get_transient_cd` procedure (see below).

3.1.3. *Requesting the Allowed Procedure Names.* The first standard procedure defined in this section is `get_allowed_heads`, which takes no arguments. This returns, in the above format, the set of OpenMath symbols which might be allowed to appear as "head" symbol (ie first child of the outermost `<OMA>`) in an SCSCP procedure call to this server. These may be symbols from standard OpenMath CDs or from transient CDs as described above.

A second standard procedure is `is_allowed_head`. It takes only one argument, which is to be an OpenMath symbol. The server returns a boolean, `logic1.true` or `logic1.false`, depending on whether it accepts this symbol. This enables the client to check whether a particular symbol is allowed without requesting the full symbol list.

Note that it is acceptable (although not necessarily desirable) for a server to "overstate" the set of symbols it accepts and use standard OpenMath errors to reject requests later.

3.1.4. *Requesting Signature Information.* The standard procedure `get_signature` takes one argument: an OpenMath symbol. If the supplied symbol is one of those accepted as a head symbol by the server then it returns an application of the `signature` symbol. This symbol takes four arguments:

- an OpenMath symbol indicating which signature is described;
- a minimum number of children (*min*);
- a maximum number of children (*max*), which could be the `infinity` symbol from the `nums1` CD);
- a set of symbols, represented as an application of the symbol `symbol_set`, or a list of `symbol_set`s:
  - if the list of `symbol_set`s is given, then its i-th entry corresponds to the i-th argument;
  - if just one `symbol_set` is given, it represents that every argument should come from this set.

3.1.5. *Requesting Transient CDs.* The standard procedure `get_transient_cd` takes one argument – an application of the `CDName` symbol from `meta`, which should start with `scscp_transient_`, and returns the corresponding CD, encoded using symbols from the `meta` CD. If no such transient CD is defined by this server it returns the `procedure_terminated` message with OM error containing the symbol `no_such_transient_cd` (from the `scscp2` CD) and the name of the CD that caused the problem.

3.1.6. *Requesting general description of the service.* The service provider may have various parameters describing an offered service. The connection initiation message contains only the service name, version and identifier, and this is not enough. Instead of this, some meta-information about the service may be structured and retrieved by a special standard procedure `get_service_description`. It takes no arguments, since it is completely determined by the server to which it was sent, and returns the symbol `service_description` that takes the following three arguments as OM strings: server name, version and the description of the service. The latter may include for example functions exposed, resources, contact details of service provider, etc.

3.2. **Operations with remote objects.**

3.2.1. *Storing, retrieving and deleting remote objects.* The following symbols are defined in the `scscp2` CD [2] to work with remote objects:

`store_session:` store an object on the server side (possibly after computing or simplyfying it), returning a cookie (actually, an OM reference) pointing to that object. This cookie is then usable within the current session to get access to the actual object;

`store_persistent:` store an object on the server side (possibly after computing or simplyfying it), returning a cookie (actually, an OM reference) pointing to that object. This cookie is then usable in the foreseeable future, possibly from different sessions, to get access to the actual object;

`retrieve:` using the cookie that was obtained earlier by one of the store procedures or another procedure call, return an OM object representing the object referred by the cookie;

`unbind:` remove the object referred by the cookie from the server.

The service provider should be able to decide whether clients are allowed to create remote objects or not.

4. Technical descriptions

4.1. **Messages from client to server.**

4.1.1. *Procedure call.* The procedure call is an OM object having in general the following structure:

```
<OMOBJ>
    <OMATTR>
        <!-- call_id and option_return_... are mandatory -->
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>call_identifier</OMSTR>
            <OMS cd="scscp1" name="option_runtime" />
            <OMI>runtime_limit_in_milliseconds</OMI>
            <OMS cd="scscp1" name="option_min_memory" />
            <OMI>minimal_memory_required_in_bytes</OMI>
            <OMS cd="scscp1" name="option_max_memory" />
            <OMI>memory_limit_in_bytes</OMI>
            <OMS cd="scscp1" name="option_debuglevel" />
            <OMI>debuglevel_value</OMI>
            <OMS cd="scscp1" name="option_return_object" />
            <OMSTR></OMSTR>
        </OMATP>
        <!-- Attribution pairs finished, now the procedure call -->
        <OMA>
            <OMS cd="scscp1" name="procedure_call" />
            <OMA>
                <OMS cd="..." name="..."/>
                <!-- Argument 1 -->
                <!-- ... -->
                <!-- Argument M -->
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

Remarks:

(1) The first OM object in the argument of `procedure_call` is the name of the procedure, the remaining OM objects are its arguments. It may come from any (standard or non-standard) content dictionary, including transient CDs such as described in Section 3.1.2.

(2) OM symbols for options will be introduced accordingly to the list of options given in Section 2.1.1. If the need for new options is discovered, new OM symbols for them will be added to future CDs.

(3) Options (except the `call_id` and one of the options `option_return_cookie`, `option_return_nothing` or `option_return_object`) may be omitted in the procedure call, and in this case their default values must be used. The

default values of options may be determined by the service provider, and they are not regulated by the SCSCP specification.

4.1.2. *Interrupt signal.* See Section 5.3.

4.2. **Messages from server to client.**

4.2.1. *Procedure completed.* The procedure completion message is an OM object having in general the following structure:

```
<OMOBJ>
    <OMATTR>
        <!-- Attribution pairs (only call_id is mandatory) -->
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>call_identifier</OMSTR>
            <OMS cd="scscp1" name="info_runtime" />
            <OMI>runtime_in_milliseconds</OMI>
            <OMS cd="scscp1" name="info_memory" />
            <OMI>used_memory_in_bytes</OMI>
            <OMS cd="scscp1" name="info_message" />
            <OMSTR>some extra information</OMSTR>
        </OMATP>
        <!-- Attribution pairs finished, now the result -->
        <OMA>
            <OMS cd="scscp1" name="procedure_completed" />
            <!-- The result itself, may be OM symbol for cookie -->
            <!-- OM_object_corresponding_to_the_result -->
        </OMA>
    </OMATTR>
</OMOBJ>
```

In case the procedure returns a cookie, the returned OM object must have the following structure:

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>call_identifier</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_completed" />
            <OMR href="CAS_variable_identifier" />
        </OMA>
    </OMATTR>
</OMOBJ>
```

4.2.2. *Procedure terminated.* The procedure termination message is an OM object having in the most general case the following structure:

```
<OMOBJ>
    <OMATTR>
       <!-- Attribution pairs (only call_id is mandatory) -->
```

```
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>call_identifier</OMSTR>
            <OMS cd="scscp1" name="info_runtime" />
            <OMI>runtime_in_milliseconds</OMI>
            <OMS cd="scscp1" name="info_memory" />
            <OMI>used_memory_in_bytes</OMI>
        </OMATP>
        <!-- Attribution pairs finished, now the error message -->
        <OMA>
            <OMS cd="scscp1" name="procedure_terminated" />
            <OME>
                 <OMS cd="scscp1" name="name_of_standard_error"/>
                 <!-- Error description depends on error type -->
                 <OMSTR>Error_message</OMSTR>
            </OME>
        </OMA>
    </OMATTR>
</OMOBJ>
```

5. Reference Implementation

In this section we describe a simple implementation.

To facilitate the implementation, we use XML processing instructions (PI). All PIs defined by SCSCP specification have the form

`<?scscp [key] [attribute="value" [attribute="value" [...]]] ?>`

where:

- `key` is an alphanumerical identifier,
- `attribute` is an alphanumerical identifier,
- `value` is an arbitrary string with the same constraints as an XML URI, unless stated otherwise.

A single SCSCP processing instruction must not exceed 4094 bytes in total, including `<?` and `?>` elements. Furthermore, any processing instruction with an unrecognized key must be ignored.

5.1. **Connection Initiation.** The software wishing to provide an SCSCP service should listen on port 26133. This port has been assigned to SCSCP by the Internet Assigned Numbers Authority (IANA) in November 2007, see `http://www.iana.org/assignments/port-numbers`.

If the port is already in use, for example in a parallel computing context, the SCSCP service may listen on another port, for example by increasing the port number by one until an available port is found.

When a client connects, the server should send it a Connection Initiation Message, agree about the protocol version, and then commence an exchange of messages as desribed in Section 2.3.

5.1.1. *Connection Initiation Message.* This processing instruction is the first message that the client receives from the server in the beginning of the SCSCP session. Is has the following format:

```
<?scscp service_name="name" service_version="ver"
  service_id="id" scscp_versions="list_of_supported_versions" ?>
```

where

- `"name"` is a string containing the service name, e.g. `"GAPSmallGroups"`, `"KANT"`, `"MapleIntegration"`, `"MuPAD"`, etc.
- `"ver"` is a string containing the version of the service, e.g. `"11"`, `"3.1"`, `"4.1.5beta"`, etc.
- `"id"` is an identifier for the service, **unique** within the SCSCP session (e.g. a hostname-pid combination, a pid, etc.). Note that when the hostname is used, it is advised not to trim off any domain information from the hostname to avoid situations when the client connected to multiple hosts may get coinciding service identifiers.
- `scscp_versions` is a space-separated list of identifiers of SCSCP versions supported by the server. Identifier of every single version of SCSCP may contain digits, letters and dots. Versions can be listed in any order. For example, valid value of `scscp_versions` is `"1.0 3.4.1 1.2special"`.

The format of the control sequence is compulsory, and server implementations must not change the order of the attribute/value pairs nor omit some of them. This strict restriction makes it sure that even very simple clients should be able to parse this control sequence.

After receiving an incoming connection from a client, the SCSCP server must not send to the client anything else before the connection initiation message. The client must wait for this "welcome string" from the server, and must not send any data to the server before obtaining it.

5.1.2. *Version negotiation.* To ensure compatibility of various versions of the protocol both upwards and downwards the following version negotiation procedure should be performed.

After obtaining the connection initiation message from the server, the client informs the server about the version of the protocol used by the client, sending the message

```
<?scscp version="client_version" ?>
```

where `"client_version"` is a string denoting the SCSCP versions supported by the client (for example, `"1.0"` or `"3.4.1"` or `"1.2special"`).

As the current rule, we require that an official SCSCP client/server should always support at least the version `"1.0"`.

We expect that the client choose one of the versions listed in the connection initiation message received from the server. However, a simple client may ignore that information, because the server is able to reject any incompatible versions the client may propose. After receiving the client's SCSCP versions, the server replies with one of two possible messages:

- `<?scscp quit reason="not supported version" ?>`, if the server does not support the version communicated by the client. The server should subsequently close the connection.
- `<?scscp version="server_version" ?>` if the server does support the version communicated by the client. Here `"server_version"` is a string representing the preferred SCSCP version of the server. The server should subsequently wait for messages from the client.

Remember that in the connection initiation phase, the order of messages is very strict:

- after receiving an incoming connection, the SCSCP server must not send anything to the client before the Connection Initiation Message,
- the client must wait for the Connection Initiation Message from the server, and must not send any data to the server before obtaining this message,
- after receiving the Connection Initiation Message the client must send its version to the server,
- after receiving the client's version, the server either confirms this to the client and starts to wait for procedure calls, or rejects it by replying with the quit message,
- only after receiving the server's confirmation of the protocol version, the client is allowed to send procedure calls to the server.

Any other data sent in this phase may be ignored.

The only other allowed control sequence during this phase of the protocol is `<?scscp quit ?>` or `<?scscp quit reason="explanation" ?>`, issued, for example, when one side is not able to accept any version proposed by the other side, or issued under external circumstances, for example, sending messages that are not allowed in this negotiation phase. In either case, the rules of quitting (see below) are applied.

An example of successful version negotiation:

```
Server -> Client:
  <?scscp service_name="MuPADserver" service_version="1.1"
    service_id="host:26133" scscp_versions="1.0 3.4.1 1.2special" ?>
Client -> Server:
  <?scscp version="1.0" ?>
Server -> Client:
  <?scscp version="1.0" ?>
```

An example of failed version negotiation:

```
Server -> Client:
  <?scscp service_name="MuPADserver" service_version="1.1"
    service_id="host:26133" scscp_versions="1.0 3.4.1 1.2special" ?>
Client -> Server:
  <?scscp version="1.5beta" ?>
Server -> Client:
  <?scscp quit reason="non supported version 1.5beta" ?>
```

5.2. **Ongoing message exchange.** All messages except interrupts are delivered as XML processing instructions and OpenMath objects in either the XML or the binary OpenMath encoding, transmitted via the socket connection.

To locate OpenMath objects in the input/output streams, we put them into *transaction blocks*. The transaction block looks as follows:

```
<?scscp start ?>
[a valid OpenMath object]
<?scscp end ?>
```

Another kind of transaction block have the form

```
<?scscp start ?>
```

```
[something]
<?scscp cancel ?>
```

and the `<?scscp cancel ?>` instruction indicates that the started transaction block is cancelled. The receiver need not to wait for a matching `<?scscp end ?>` directive, and must not process/evaluate the data in this transaction block.

Another instruction is `<?scscp quit ?>`, which may also be used in the long form `<?scscp quit reason="explanation" ?>`, that indicates that the sender is about to leave the SCSCP session. Then each side may close the socket (and at least the receiver may safely disconnect). This instruction may appear anywhere throughout the session.

Any data after successful version negotiation which are located outside of transaction blocks (apart from PIs) may be safely ignored.

5.3. **Interrupt.** The client may send a processing instruction carrying an interrupt signal as follows:

```
<?scscp terminate call_id="someidentifier" ?>
```

This PI implies that the results of the computation with call ID `someidentifier` are not needed, so the server need not complete the computation. The server may still choose to complete the calculation.

The server must, as always, reply to the `procedure_call` with the mentioned call ID. It may do this either by sending a `procedure_terminated` or, if the result is already available, a `procedure_completed` message.

Note that it is always correct for a server to ignore an interrupt, and that this may be appropriate when procedure calls are quick.

5.4. **Info messages.** We finally describe PIs carrying some information that is not critical for the functioning of the protocol. These PIs have the form

```
<?scscp info="some interesting information" ?>
```

and may be sent by both the client and the server. They may aid in debugging a protocol implementation.

5.5. **Other ways of implementation.** We assume that in most cases it will be enough for the SCSCP-compliant server to support the scheme outlined above, since this provides the opportunity to "talk" in clean SCSCP with other servers supporting this scheme. It also allows communication with the SCSCP-proxy server, that accepts SOAP/HTTP requests from the outside and then sends their SCSCP content to servers. However, it is not unconceivable that the server itself is able to receive SOAP/HTTP requests without the mentioned proxy.

The same refers to the client functionality: if there is a server that accepts only SOAP/HTTP requests, then this server can send an SCSCP-request to the proxy which will then act as a proper Web services client, or might be able to wrap an SCSCP procedure call into a SOAP/HTTP envelope itself.

Appendix A. The scscp1 and scscp2 OM CDs

The list of OM symbols defined in the scscp1 CD

The scscp1 CD [1] defines OM symbols for main types of messages and attributes that may appear in them:

(1) Main messages:
- procedure_call
- procedure_completed
- procedure_terminated

(2) Call and response identifiers:
- call_id

(3) Options in procedure calls:
- option_max_memory
- option_min_memory
- option_runtime
- option_debuglevel
- option_return_cookie
- option_return_object
- option_return_nothing

(4) Information attributes:
- info_memory
- info_runtime
- info_message

(5) Standard errors:
- error_memory
- error_runtime
- error_system_specific

See the scscp1 CD [1] for the appropriate descriptions.

The list of OM symbols defined in the scscp2 CD

The scscp2 CD [2] defines OM symbols for special procedures, and also some symbols that may appear in their arguments and results:

(1) Procedures for work with remote objects:
- store_session
- store_persistent
- retrieve
- unbind

(2) Special procedures to obtain meta-information about SCSCP service:
- get_allowed_heads
- is_allowed_head
- get_transient_cd
- get_signature
- get_service_description

(3) Special symbols:
- signature
- service_description
- symbol_set
- symbol_set_all
- no_such_transient_cd

See the scscp2 CD [2] for the appropriate descriptions.

Appendix B. Examples of OM messages

B.1. **The Procedure Call message.** The `GroupIdentificationService` procedure accepts a permutation group G given by its generators and returns the `procedure_completed` message with the number of this group in the GAP Small Groups Library. If this fails, for example because groups of order $|G|$ are not contained in that library or because identification for groups of such order is not available in GAP, it sends a `procedure_terminated` message.

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:s2sYf1pg</OMSTR>
            <OMS cd="scscp1" name="option_runtime" />
            <OMI>300000</OMI>
            <OMS cd="scscp1" name="option_min_memory" />
            <OMI>40964</OMI>
            <OMS cd="scscp1" name="option_max_memory" />
            <OMI>134217728</OMI>
            <OMS cd="scscp1" name="option_debuglevel" />
            <OMI>2</OMI>
            <OMS cd="scscp1" name="option_return_object" />
            <OMSTR></OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call" />
            <OMA>
                <OMS cd="scscp_transient_1"
                   name="GroupIdentificationService" />
                <OMA>
                    <OMS cd="group1" name="group"/>
                    <OMA>
                        <OMS cd="permut1" name="permutation"/>
                        <OMI> 2</OMI> <OMI> 3</OMI>
                        <OMI> 1</OMI>
                    </OMA>
                    <OMA>
                        <OMS cd="permut1" name="permutation"/>
                        <OMI> 1</OMI> <OMI> 2</OMI>
                        <OMI> 4</OMI> <OMI> 3</OMI>
                    </OMA>
                </OMA>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

In the next example we retrieve the group [24,12] from GAP Small Groups Library, creating it on the GAP side and requesting a cookie for it:

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:GvnaGRLN</OMSTR>
            <OMS cd="scscp1" name="option_return_cookie" />
            <OMSTR></OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call" />
            <OMA>
                <OMS cd="scscp_transient_1" name="GroupByIdNumber" />
                <OMI>24</OMI><!-- Arg1 -->
                <OMI>12</OMI><!-- Arg2 -->
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

In the next example we are sending an OM object containing a MathML object:

```
<OMOBJ>
  <OMATTR>
    <OMATP>
      <OMS cd="scscp1" name="call_id"/>
      <OMSTR>symcomp.org:26133:18668:mDkmEqh9</OMSTR>
      <OMS cd="scscp1" name="option_return_cookie"/>
      <OMSTR></OMSTR>
    </OMATP>
    <OMA><OMS cd="scscp1" name="procedure_call"/>
      <OMA>
        <OMS cd="scscp_transient_M25" name="Evaluate" />
        <OMA>
          <OMS cd="altenc" name="MathML_encoding"/>
          <OMFOREIGN encoding="MathML-Presentation">
            <math xmlns="http://www.w3.org/1998/Math/MathML">
              <mrow>
                <mi>sin</mi>
                <mo>&ApplyFunction;</mo>
                <mfenced><mi>x</mi></mfenced>
              </mrow>
            </math>
          </OMFOREIGN>
        </OMA>
      </OMA>
    </OMA>
  </OMATTR>
</OMOBJ>
```

B.2. **The Procedure Completed message.** The `GroupIdentificationService` procedure from the previous example returns its successful output in the following form:

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:s2sYf1pg</OMSTR>
            <OMS cd="scscp1" name="info_runtime" />
            <!-- The runtime in milliseconds as OM integer -->
            <OMI>1234</OMI>
            <OMS cd="scscp1" name="info_memory" />
            <!-- Memory occupied by CAS in bytes as OM integer -->
            <OMI>134217728</OMI>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_completed" />
            <OMA>
                <OMS cd="linalg2" name="vector"/>
                <OMI> 24</OMI>
                <OMI> 12</OMI>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

In case a cookie is requested, the `procedure_completed` message may look as follows (we assume the minimal debugging level with no information about the runtime and memory used):

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:GvnaGRLN</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_completed" />
            <OMR href="scscp://somehost.somedomain:26133/q9t4eX" />
        </OMA>
    </OMATTR>
</OMOBJ>
```

B.3. **The Procedure Terminated message.** This is an example how the server may return an error message if the error arises at the CAS level (we assume the minimal debugging level with no information about the runtime and memory used):

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:f1dG4fcT</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_terminated" />
            <OME>
                 <OMS cd="scscp1" name="error_system_specific"/>
                 <OMSTR>Number of arguments must be 2</OMSTR>
            </OME>
        </OMA>
    </OMATTR>
</OMOBJ>
```

One of standard errors on the SCSCP level may look as follows:

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:Sf3Rc6cT</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_terminated" />
            <OME>
                 <OMS cd="scscp1" name="error_memory"/>
                 <OMSTR>Exceeded the permitted memory</OMSTR>
            </OME>
        </OMA>
    </OMATTR>
</OMOBJ>
```

Appendix C. Examples of special procedures

C.1. **List of supported procedures.** To find the list of procedures supported by the SCSCP server, the client sends to the server the following message:

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:sRe6Gwqk</OMSTR>
            <OMS cd="scscp1" name="option_return_object" />
            <OMSTR></OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call" />
            <OMA>
                <OMS cd="scscp2" name="get_allowed_heads" />
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

The server replies with the next message, demonstrating all possible arguments of the `symbol_set` OpenMath symbol: symbols from transient CD, created by the service provider, a symbol from a standard CD, a CD and a CD group.

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:KdeT54cV</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_completed" />
            <OMA>
                <OMS cd="scscp2" name="symbol_set"/>
                <OMS cd="scscp_transient_1"
                   name="GroupIdentificationService" />
                <OMS cd="group1" name="group" />
                <OMA>
                    <OMS cd="meta" name="CDName"/>
                    <OMSTR>permut1</OMSTR>
                </OMA>
                <OMA>
                    <OMS cd="metagrp" name="CDGroupName"/>
                    <OMSTR>scscp</OMSTR>
                </OMA>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

Now the client is interested in getting the transient CD `scscp_transient_1`, and sends to the server the procedure call displayed below.

Note that in this example we assume that the `scscp_transient_1` CD contains just one symbol `GroupIdentificationService`, so the server could have just returned `CDName` with `scscp_transient_1` as first argument instead of one symbol. In such a case the client may use `get_transient_cd` procedure to get the corresponding CD in order to find out which OM symbols are defined in it.

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:aWeRvK2a</OMSTR>
            <OMS cd="scscp1" name="option_return_object" />
            <OMSTR></OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call" />
            <OMA>
                <OMS cd="scscp2" name="get_transient_cd" />
                <OMA>
                    <OMS cd="meta" name="CDName"/>
                    <OMSTR>scscp_transient_1</OMSTR>
                </OMA>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

In response to this procedure call, the server sends to the client the transient CD in the following message:

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:dLrHd64G</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_completed" />
            <OMA>
                <OMS cd="meta" name="CD"/>
                <OMA>
                    <OMS cd="meta" name="CDName"/>
                    <OMSTR>scscp_transient_1</OMSTR>
                </OMA>
                <OMA>
                    <OMS cd="meta" name="CDDate"/>
                    <OMSTR>2007-08-24</OMSTR>
                </OMA>
                <OMA>
```

```
                    <OMS cd="meta" name="Description"/>
                    <OMSTR>CD created by the service provider</OMSTR>
                </OMA>
                <OMA>
                    <OMS cd="meta" name="CDDefinition"/>
                    <OMA>
                       <OMS cd="meta" name="Name"/>
                       <OMSTR>GroupIdentificationService</OMSTR>
                    </OMA>
                    <OMA>
                       <OMS cd="meta" name="Description"/>
                       <OMSTR>IdGroup(permgroup by gens)</OMSTR>
                    </OMA>
                </OMA>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

We assume that when the procedure is installed as an SCSCP procedure, the server assigns a name to the transient CD, saves the time of its creation, and puts some textual information, specified by the service provider, in its description. Also in this step the service provider must specify the signature of this procedure, which then can be retrieved by the client by sending the following message to the server:

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:sRsmc1e4</OMSTR>
            <OMS cd="scscp1" name="option_return_object" />
            <OMSTR></OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call" />
            <OMA>
                <OMS cd="scscp2" name="get_signature" />
                <OMS cd="scscp_transient_1"
                   name="GroupIdentificationService" />
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

The server then replies with the following message:

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:s8sFDnf6</OMSTR>
        </OMATP>
```

```
        <OMA>
            <OMS cd="scscp1" name="procedure_completed" />
            <OMA>
                <OMS cd="scscp2" name="signature"/>
                <OMS cd="scscp_transient_1"
                   name="GroupIdentificationService" />
                <OMI>1</OMI>
                <OMI>1</OMI>
                <OMA>
                    <OMS cd="list1" name="list"/>
                    <OMA>
                       <OMS cd="scscp2" name="symbol_set"/>
                       <OMS cd="group1" name="group"/>
                       <OMA>
                          <OMS cd="meta" name="CDName"/>
                          <OMSTR>permut1</OMSTR>
                       </OMA>
                    </OMA>
                </OMA>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

The list containing one `symbol_set` symbol in the signature means that its first element represents the set of accepted symbols for the first (and unique) argument. In case the maximum number of arguments is not specified, we will use just one `symbol_set` symbol not enclosed in the list. Also note that we allow procedure calls when the actual number of arguments is smaller than the maximal, and in this case extra entries of this list should be be ignored.

C.2. **Signature of a generic service.** Below we give an example of the signature of some generic service, that does not restrict the number of arguments, but exposes the list of supported CDs and CD groups:

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:Sf3Rc6cT</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_completed" />
            <OMA>
                <OMS cd="scscp2" name="signature"/>
                <OMS cd="scscp_transient_1" name="CAS_Service" />
                <OMI>0</OMI>
                <OMS cd="nums1" name="infinity"/>
```

```
                <OMA>
                    <OMS cd="scscp2" name="symbol_set"/>
                    <OMA>
                      <OMS cd="meta" name="CDGroupName"/>
                      <OMSTR>scscp</OMSTR>
                    </OMA>
                    <OMA>
                      <OMS cd="meta" name="CDName"/>
                      <OMSTR>scscp_transient_0</OMSTR>
                    </OMA>
                    <OMA>
                      <OMS cd="meta" name="CDName"/>
                      <OMSTR>scscp_transient_1</OMSTR>
                    </OMA>
                    <OMA>
                      <OMS cd="meta" name="CDName"/>
                      <OMSTR>arith1</OMSTR>
                    </OMA>
                    <OMA>
                      <OMS cd="meta" name="CDName"/>
                      <OMSTR>transc1</OMSTR>
                    </OMA>
                </OMA>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

The next example demonstrates another default "universal" signature that may be returned in case nothing was specified during the installation of the SCSCP procedure on the server:

```
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id" />
            <OMSTR>symcomp.org:26133:18668:tRAdnc76</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_completed" />
            <OMA>
                <OMS cd="scscp2" name="signature"/>
                <OMS cd="scscp_transient_1" name="Something" />
                <OMI>0</OMI>
                <OMS cd="nums1" name="infinity"/>
                <OMS cd="scscp2" name="symbol_set_all"/>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
```

Appendix D. Change Log

Version 1.3

- Various small clarifications and fixes;
- Section 2.1.1: Introduced/emphasized obligatoryness of `call_id` and precisely one of the options `option_return_object`, `option_return_cookie` and `option_return_nothing`
- Section 2.1.2: Complete overhaul of specification of interrupt signal. Has been changed into a processing instruction, mentioned in section 5.3;
- Clarified various arguments, options, etc being optional. See in particular Sections 2.1.1, 2.2.1, 2.2.2;
- Changed "CAS" to "server" in various places;
- Changed casing of name for "intermediate CDs": they should start with `scscp_transient_` instead of `SCSCP_transient_`;
- Changed `call_ID` to `call_id` in OpenMath snippets.
- added `scscp2.is_allowed_head` to easily check whether a particular OpenMath symbol is allowed as head of a procedure call;
- `scscp2.store` changes into two variants to distinguish the lifetime of remote objects: `store_persistent` and `store_session`;
- Removed explicit "we allow only those key and attribute identifiers" with respect to processing instructions (which also more or less contradicted requirement that after version negotiation everything outside processing instructions `<?scscp start?>` and `<?scscp end?>` should be ignored). The new situation is that (for future backwards compatibility) any processing instruction with an unrecognized key must be ignored.
- Added `<?scscp info="..." ?>` processing instructions;
- Added `scscp1.info_message` symbol to allow small pieces of information to accompany a `procedure_completed` or `_terminated` message;
- When `option_return_nothing` is given a server is supposed to reply with an empty `procedure_completed` or `_terminated` message.

Furthermore, the corresponding changes in the `scscp1` and `scscp2` content dictionaries were made. In particular, the descriptions of the standard errors `error_runtime` and `error_system_specific` were clarified.

Version 1.2

- added `option_return_nothing`
- extended bibliography
- reordered `scscp2` description
- moved store-retrieve-unbind triple to special procedures section
- removed outdated comment about `call_ID`
- `<OMSTR/>` replaced by `<OMSTR></OMSTR>` in examples

Version 1.1

- Updated list of authors
- Mentioned that port number 26133 is registered for SCSCP
- clarified that `no_such_transient_cd` is returned within an error message contained in `procedure_completed` message.

References

[1] D. Roozemond. *OpenMath Content Dictionary* `scscp1`. (`http://www.win.tue.nl/SCIEnce/cds/scscp1.html`).
[2] D. Roozemond. *OpenMath Content Dictionary* `scscp2`. (`http://www.win.tue.nl/SCIEnce/cds/scscp2.html`).

Sebastian Freundt
Fakultät II - Institut für Mathematik,
Technische Universität Berlin, Berlin, Germany
E-mail address: `freundt@math.tu-berlin.de`

Peter Horn
Fachbereich Mathematik, Universität Kassel, Kassel, Germany
E-mail address: `hornp@mathematik.uni-kassel.de`

Alexander Konovalov
School of Computer Science, University of St Andrews,
North Haugh, St Andrews, Fife, KY16 9SX, Scotland
E-mail address: `alexk@cs.st-and.ac.uk`

Steve Linton
School of Computer Science, University of St Andrews,
North Haugh, St Andrews, Fife, KY16 9SX, Scotland
E-mail address: `sal@cs.st-and.ac.uk`

Dan Roozemond
Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven,
HG 9.55, Postbus 513, 5600 MB Eindhoven, Netherlands
E-mail address: `d.a.roozemond@tue.nl`