Easy Composition of Symbolic Computation Software: A New Lingua Franca for Symbolic Computation

S. Linton, K. Hammond, A. Konovalov,
University of St Andrews
{sal,kh,alexk}
@cs.st-and.ac.uk

A. D. Al Zain, P. Trinder
Heriot-Watt University
{ceeatia, trinder}
@macs.hw.ac.uk

P. Horn
Universität Kassel
horn@math.uni-kassel.de

D. Roozemond
Technical Universiteit Eindhoven
d.a.roozemond@tue.nl

ABSTRACT

We present the results of the first four years of the European research project SCIEnce (`www.symbolic-computation.org`), which aims to provide key infrastructure for symbolic computation research. A primary outcome of the project is that we have developed a new way of combining computer algebra systems using the Symbolic Computation Software Composability Protocol (SCSCP), in which both protocol messages and data are encoded in the OpenMath format. We describe SCSCP middleware and APIs, outline some implementations for various Computer Algebra Systems (CAS), and show how SCSCP-compliant components may be combined to solve scientific problems that can not be solved within a single CAS, or may be organised into a system for distributed parallel computations.

Categories and Subject Descriptors

I.1 [**Symbolic and Algebraic Manipulation**]: Miscellaneous

General Terms

Design, Standardization

Keywords

OpenMath, SCSCP, interface, coordination, parallelism

1. INTRODUCTION

A key requirement in symbolic computation is to efficiently combine computer algebra systems (CAS) to solve problems that cannot be addressed by any single system. Additionally, there is often a requirement to have CAS as a back-end of mathematical databases and web or grid services, or to combine multiple instances of the same or different CAS for parallel computations.

There are many possible combinations. Examples include: GAP and Maple in CHEVIE for handling generic character tables [21]; Maple and the PVS theorem prover to obtain more reliable results [1]; GAP and nauty in GRAPE for fast graph automorphisms [40]; and GAP as a service for the ECLiPSe constraint programming system for symmetry-breaking in search [22]. In all these cases, interfacing to a CAS with the required functionality is far less work than re-implementing the functionality in the "home" system.

Even within a single CAS, users may need to combine local and remote instances for a number of reasons, including: remote system features which are not supported in the local operating system; a need to access large (and changing) databases; remote access to the latest development version or to the configuration at the home institution; licensing restrictions permitting only online services, etc. A common quick solution is cut-and-paste from telnet sessions and web browsers. It would, however, be more efficient and more flexible to combine local and remote computations in a way such that remotely obtained results will be plugged immediately into the locally running CAS.

Moreover, individual CPUs have almost stopped increasing in power, but are becoming more numerous. A typical workstation now has 4-8 cores, and this is only a beginning. If we want to solve larger problems in future, it will be essential to exploit multiple processors in a way that gives good parallelism for minimal programmer/user effort.

CAS authors have inevitably started to face these issues, and have addressed them in various ways. For example, a CAS may write input files for another program and invoke it; the other program then will write CAS input to a file and exit; the CAS will read it and return a result. This works, but has fairly serious limitations. A better setup might allow the CAS to interact with other programs while they run and provide a separate interface to each possible external system. The SAGE system [39] is essentially built around this approach. However, achieving this is a major programming challenge, and an interface will be broken if the other system changes its I/O format, for example.

The EU Framework 6 SCIEnce project "SCIEnce – Symbolic Computation Infrastructure in Europe" is a major 5-year project that brings together CAS developers and experts in computational algebra, OpenMath, and parallel computations. It aims to design a common standard interface that may be used for combining computer algebra systems (and any other compatible software). Our vision is an easy, robust and reliable way for users to create and consume services implemented in any compatible systems, ranging from generic services (e.g. evaluation of a string or an OpenMath object) to specialised (e.g. lookup in the database; executing certain procedure). We have developed a simple lightweight XML-based remote procedure call protocol called **SCSCP** (*Symbolic Computation Software Composability Protocol*) in which both data and instructions are represented as OpenMath objects. SCSCP is now implemented in several computer algebra systems (see Section 2.2 for an overview) and has APIs making it easy to add SCSCP interfaces to more systems. Another important outcome of the project is the development of middleware for parallel computations, **SymGrid-Par**, which is capable of orchestrating SCSCP-compliant systems into a heterogeneous system for distributed parallel computations.

We will give an overview of these tools below. First we briefly characterise the underpinning OpenMath data encoding and the SCSCP protocol (Section 2). Then we outline SCSCP interfaces in two different systems, one open source and one commercial, and provide references for existing implementations in other systems (Section 3). After that we describe several examples that demonstrate the flexibility of the SCSCP approach and some SCSCP specific features and benefits (Section 4). We introduce several SCSCP-compliant tools for parallel computations in various environments (Section 5), before concluding (Section 6).

2. A COMPUTER ALGEBRA LINGUA FRANCA

2.1 OpenMath

In order to connect different CAS it is necessary to speak a common language, i.e., to agree on a common way of marshaling mathematical semantics. Here, the obvious choice was *OpenMath* [37], a well-established standard that has been used in similar contexts. OpenMath is a very flexible language built from only twelve language elements (integers, doubles, variables, applications etc.). The entire semantics is encapsulated in *symbols* which are defined in *Content Dictionaries* (CDs) and are strictly separate from the language itself. So, one finds the “normal” addition under the name *plus* in the CD *arith1*. A large number of CDs is available at the OpenMath website [37], such as *polyd1* for the definition and manipulation of multivariate polynomials, *group4* for cosets and conjugacy classes, etc. OpenMath was designed to be efficiently used by computers, and may be represented in several different encodings. The XML representation is the most commonly used, but there exist also a binary representation and a more human readable representation called Popcorn [29]. In the current draft of MathML 3, an OpenMath dialect (called *Strict Content MathML*) is used for the semantics layer.

2.2 SCSCP

In order to actually perform communication between two systems, it is necessary to fix a low-level communication protocol. The protocol developed in SCIEnce is called *SCSCP*. SCSCP [15] is used both to link systems directly with each other, and also as the foundation for more advanced cluster and grid infrastructures (see Section 5). The advantage of this approach is that any system that implements SCSCP can immediately connect to all other systems that already support it. This avoids the need for special cases and minimizes repeated effort. In addition, SCSCP allows remote objects to be handled by reference so that clients may work with objects of a type that do not exist in their own system at all (see the example in Section 4.2). For example, to represent the number of conjugacy classes of a group only knowledge of integers is required, not knowledge of groups. The SCSCP protocol (currently at version 1.3) is socket-based. It uses port number 26133, as assigned by the *Internet Assigned Numbers Authority* (IANA)), and XML-format messages.

3. BUILDING BLOCKS FOR CAS COMPOSITION

In this section, we briefly describe the implementation of the SCSCP protocol for two systems: GAP [18] and MuPAD [34]. The main aim of this section is to show that SCSCP is a standard that may be implemented in different ways by different CAS, taking into account their own design principles.

3.1 GAP

In the GAP system, support for OpenMath and SCSCP is implemented in two GAP packages called `OpenMath` and `SCSCP`, respectively. The `OpenMath` package [11] is an OpenMath phrasebook for GAP: it converts OpenMath to GAP and vice versa, and provides a framework that users may extend with their private content dictionaries. The `SCSCP` package [30] implements SCSCP, using the GAP `OpenMath`, `IO` [35] and `GAPDoc` [32] packages. This allows GAP to run as either an SCSCP server or client. The server may be started interactively from the GAP session or as a GAP daemon. When the server accepts a connection from the client, it starts the “accept-evaluate-return” loop:

- accepts the `"procedure_call"` message and looks up the appropriate GAP function (which should be declared by the service provider as an SCSCP procedure);
- evaluates the result (or produces a side-effect);
- replies with a `"procedure_completed"` message or returns an error in a `"procedure_terminated"` message.

The SCSCP client performs the following basic actions:

- establishes connection with server;
- sends the `"procedure_call"` message to the server;
- waits for its completion or checks it later;
- fetches the result from a `"procedure_completed"` message or enters the break loop in the case of a `"procedure_terminated"` message.

We have used this basic functionality to build a set of instructions for parallel computations using the SCSCP framework. This allows the user to send several procedure calls in parallel and then collect all results, or to pick up the first available result. We have also implemented the master-worker parallel skeleton in the same way (see Section 5.2).

A demo SCSCP server is available for test purposes at `chrystal.mcs.st-andrews.ac.uk`, port 26133. This runs the development version of the GAP system plus a selection of public GAP packages. Further details, downloads, and a manual with examples are available online [30].

3.2 MuPAD

There are two main aspects to the MuPAD SCSCP support: the MuPAD `OpenMath` package [26] and the SCSCP server wrapper for MuPAD. The former offers the ability to parse, generate, and handle OpenMath in MuPAD and to consume SCSCP services, the latter provides access to MuPAD's mathematical abilities as an SCSCP service. The current MuPAD end-user license agreement, however, does not generally allow providing MuPAD computational facilities over the network. We therefore focus on the open-source `OpenMath` package, which can be downloaded from [26].

3.2.1 OpenMath Parsing and Generation

Two functions are available to convert an OpenMath XML string into a tree of MuPAD `OpenMath::` objects, namely `OpenMath::parse(`*str*`)` which parses the string *str*, and `OpenMath::parseFile(`*fname*`)` which reads and parses the file named *fname*. Conversely, a MuPAD expression can be converted into its OpenMath representation using `generate::OpenMath`. Note that it is not necessary to directly use OpenMath in MuPAD if the SCSCP connection described below is used: the package takes care of marshalling and unmarshalling in a way that is completely transparent to the MuPAD user.

3.2.2 SCSCP Client Connection

The call `s := SCSCP(`*host*, *port*`)` creates an SCSCP connection object, that can subsequently be used to send commands to the SCSCP server. Note that the actual connection is initiated on construction by starting the Java program WUPSI [27] which is bundled with the `OpenMath` package. This uses an asynchronous message-exchange mode, and can therefore be used to introduce background computations. The command `s::compute(...)` can then be used to actually compute something on the server (`s(...)` is equivalent). Note that it may be necessary to wrap the parameter in `hold(...)` to prevent premature evaluation on the client side. In order to use the connection asynchronously, the `send` and `retrieve` commands may be used: `a := s::send(...)` returns an integer which may be used to identify the computation. The result may subsequently be retrieved using `s::retrieve(a)`. `retrieve` will normally return `FAIL` if the result of the computation is not yet computed, but this behaviour can be overridden using a second parameter to force the call to block.

3.3 Other Implementations of SCSCP

The SCIence project has produced a Java library [28] that acts as a reference implementation for systems developers who would like to implement SCSCP for their own systems. This is freely available under the Apache2 license. In addition to GAP and MuPAD, SCSCP has also been implemented in two other systems participating in the SCIEnce project: KANT [17] and Maple [33] (the latter implementation is currently a research prototype and not available in the Maple release). There are third-party implementations for TRIP [19, 20], Magma [8] (as a wrapper application), and Macaulay2 [24]. SCSCP thus, as intended, allows a large range of CAS to interact and to share computations.

4. EXAMPLES

In this section we provide a number of examples which demonstrate the features and benefits of SCSCP, such as flexible design, composition of different CAS, working with remote objects and speeding up computations. More examples can be found in e.g. [14, 16] and on the web sites for individual systems.

4.1 GAP

In order to illustrate the flexibility of our approach, we will describe three possible ways to set up a procedure for the same kind of problems.

The GAP Small Groups Library [7] contains all groups of orders up to 2000, except groups of order 1024. The GAP command `SmallGroup(n,i)` returns the `i`-th group of order `n`. Moreover, for any group G of order $1 \leq |G| \leq 2000$ where $|G| \notin \{512, 1024\}$, GAP can determine its *library number*: the pair `[n,i]` such that G is isomorphic to `SmallGroup(n,i)`. This is in particular the most efficient way to check whether two groups of "small" order are isomorphic or not.

Let us consider now how we can provide a group identification service with SCSCP. When designing an SCSCP procedure to identify small groups, we first need to decide how the client should transmit a group to the server. We will give three possible scenarios and outline simple steps needed for the design and provision of the SCSCP services within the provided framework.

Case 1. A client supports permutation groups (for example, a client is a minimalistic GAP installation without the Small Groups Library). In this case the conversion of the group to and from OpenMath will be performed straightforwardly, so that the service provider only needs to install the function `IdGroup` as an SCSCP procedure (under the same or different name) before starting the server:

```
gap> InstallSCSCPprocedure("IdGroup",IdGroup);
InstallSCSCPprocedure : IdGroup installed.
```

The client may then call this, obtaining a record with the result in its `object` component:

```
gap> EvaluateBySCSCP("IdGroup",[SymmetricGroup(6)],
>                    "scscp.st-and.ac.uk",26133);
rec( attributes := [ [ "call_id", "hp0SE18S" ] ],
  object := [ 720, 763 ] )
```

Case 2. A client supports matrices, but not matrix groups. In this case, the service provider may install the SCSCP procedure which constructs a group generated by its arguments and return its library number:

```
IdGroupByGens := gens -> IdGroup( Group( gens ) );
```

Note that validity of any input and the applicability of the `IdGroup` method to the constructed group will be automatically checked by GAP during the execution of the procedure on the SCSCP server, so there is no need to add such checks to this procedure (though they may be added to replace the standard GAP error message for these cases by other text).

Case 3. A client supports groups in some specialised representation (for example, groups given by pc-presentation in GAP). Indeed, for groups of order 512 the Small Groups Library contains all 10494213 non-isomorphic groups of this order and allows the user to retrieve any group by its library number, but it does not provide an identification facility. However, the GAP package ANUPQ [36] provides a function `IdStandardPresented512Group` that performs the latter task. Because the ANUPQ package only works in a UNIX environment it is useful to design an SCSCP service for identification of groups of order 512 that can be called from within GAP sessions running on other platforms (note that the client version of the SCSCP package for GAP does work under Windows).

Now the problem reduces to the encoding of such a group in OpenMath. Should it, for example, be converted into a permutation representation, which can be encoded using existing content dictionaries or should we develop a new content dictionary for groups in such a representation? Luckily, the SCSCP protocol provides enough freedom for the user to select his/her own data representation. Since we are interfacing between two copies of the GAP system, we are free to use a GAP-specific data format, namely the *pcgs code*, an integer that describes the *polycyclic generating sequence* (*pcgs*) of the group, to pass the data to the server (see the GAP manual and [6] for more details).

First we create a function that takes the *pcgs code* of a group of order 512 and returns the number of this group in the GAP Small Groups library:

```
gap> IdGroup512 := function( code )
>    local G, F, H;
>    G := PcGroupCode( code, 512 );
>    F := PqStandardPresentation( G );
>    H := PcGroupFpGroup( F );
>    return IdStandardPresented512Group( H );
>    end;;
```

After such a function is created on the server, it becomes "visible" as an SCSCP procedure under the same name:

```
gap> InstallSCSCPprocedure("IdGroup512",IdGroup512);
InstallSCSCPprocedure : IdGroup512 installed.
```

For convenience, the client may be supplied with a function that is specialised to use the correct server port, and which checks that the transmitted group is indeed of order 512:

```
gap> IdGroup512Remote:=function( G )
>    local code, result;
>    if Size(G)<>512 then Error("|G|<>512\n");fi;
>    code := CodePcGroup( G );
>    result := EvaluateBySCSCP("IdGroup512",[code],
>              "scscp.st-and.ac.uk", 26133);
>    return result.object;
>    end;;
```

Now the call to `IdGroup512Remote` returns the result in the standard `IdGroup` notation:

```
gap> IdGroup512Remote( DihedralGroup( 512 ) );
[ 512, 2042 ]
```

4.2 GAP and Macaulay2

We now consider interaction between GAP and Macaulay2 [24]. Macaulay2 is "a software system devoted to supporting research in algebraic geometry and commutative algebra," that is particularly well known for its efficient Gröbner bases procedures. We have implemented OpenMath and the SCSCP protocol as packages in the Macaulay2 system, and they have been available in the stable branch since late 2009. The OpenMath support includes basic arithmetic, matrices, finite fields elements, polynomials, Gröbner bases, etc. All support for OpenMath symbols was implemented directly in the Macaulay2 language, to allow for easy maintenance and extensibility.

Macaulay2 is fully SCSCP 1.3 compatible and can act both as a server and as a client. The server is multithreaded so it can serve many clients at the same time, and supports storing and retrieving of remote objects. The client was designed in such a way as to disclose remote computation using SCSCP with minimal interaction from the user. It supports convenient creation and handling of remote objects, as demonstrated below.

An example of a GAP client calling a Macaulay2 server for the Gröbner basis computation, can be found [16]. Although this 2008 implementation used a prototype wrapper implementation of an SCSCP server for Macaulay2, rather than the full internal implementation that we have now, it nicely demonstrates the possible gain of connecting computer algebra systems using SCSCP.

The next example of a Macaulay2 SCSCP client calling a remote GAP server was produced using the current implementation. First, we load the OpenMath and SCSCP packages and establish a connection to the GAP server that accepts and evaluates OpenMath objects.

```
i1 : loadPackage "SCSCP"; loadPackage "OpenMath";

i3 : GAP = newConnection "127.0.0.1"
o3 = SCSCP Connection to GAP (4.dev) on
                             scscp.st-and.ac.uk:26133
o3 : SCSCPConnection
```

We demonstrate the conversion of an arithmetic operation to OpenMath syntax (note the abbreviated form Macaulay2 uses to improve legibility of XML expressions), and evaluate the expression in GAP.

```
i4 : openMath 1+2
o4 = <OMA
       <OMS cd="arith1" name="plus"
       <OMI "1"
       <OMI "2"
o4 : XMLnode

i5 : GAP <== openMath 1 + 2
o5 = 3
```

We then create two matrices in Macaulay2 (suppressing the output of the creation of the second one), and create a remote object G in GAP representing the group they generate.

```
i6 : m1 = id_(QQ^10)^{1,6,2,7,3,8,4,9,5,0}
o6 = | 0 1 0 0 0 0 0 0 0 0 |
     | 0 0 0 0 0 0 1 0 0 0 |
     | 0 0 1 0 0 0 0 0 0 0 |
     | 0 0 0 0 0 0 0 1 0 0 |
     | 0 0 0 1 0 0 0 0 0 0 |
     | 0 0 0 0 0 0 0 0 1 0 |
     | 0 0 0 0 1 0 0 0 0 0 |
     | 0 0 0 0 0 0 0 0 0 1 |
     | 0 0 0 0 0 1 0 0 0 0 |
     | 1 0 0 0 0 0 0 0 0 0 |
              10        10
o6 : Matrix QQ   <--- QQ

i7 : m2 = id_(QQ^10)^{1,0,2,3,4,5,6,7,8,9};

i8 : G = GAP <=== matrixGroup({m1,m2})
```

```
o8 = << Remote GAP object >>
o8 : RemoteObject
```

When we ask for the size of the group, Macaulay2 simply creates a new object representing $|G|$. Finally, evaluating this object in GAP gives the number of elements in the group generated by those matrices.

```
i9 : size G
o9 = << Remote GAP object >>
o9 : RemoteObject

i10 : GAP <== size G
o10 = 10080
```

One of the most important features of this example is that despite the fact that Macaulay2 has no support for groups at all, by using OpenMath and SCSCP, we can still create an object that represents a group, and obtain useful information about it.

4.3 MuPAD

To show how the OpenMath MuPAD package is used, we first demonstrate some features of the OpenMath package:

```
>> package("OpenMath"):
>> 1+a*sin(x)
a*sin(x) + 1
>> om := OpenMath(%)
arith1.plus(1, arith1.times(transc1.sin($x), $a))
>> OpenMath::toXml(om)
<OMA>
  <OMS cd='arith1' name='plus'/>
  <OMI>1</OMI>
  <OMA>
    <OMS cd='arith1' name='times'/>
    <OMA>
      <OMS cd='transc1' name='sin'/>
      <OMV name='x'/>
    </OMA>
    <OMV name='a'/>
  </OMA>
</OMA>
```

Now we use it to establish an SCSCP connection to a machine 400km away that is running KANT [12]. We use the KANT server to factor the product of shifted Swinnerton-Dyer polynomials. Of course, we could do it locally in MuPAD, but that would take 38 seconds:

```
>> swindyer := proc(plist) ... :
>> R := Dom::UnivariatePolynomial(x,Dom::Rational):
>> p1 := R(swindyer([2,3,5,7,11])):
>> p2 := R(subs(swindyer([2,3,5,7,13,17])),x=3*x-2):
>> p := p1 * p2:
>> degree(p), nterms(p)
96, 49
>> st := time(): F1 := factor(p): time()-st
38431
```

Now let us use KANT remotely:

```
>> package("OpenMath"):
>> kant := SCSCP("scscp.math.tu-berlin.de",26133):
>> st:=rtime():
   F2:=kant::compute(hold(factor)(p)):
   rtime()-st
1221
```

Establishing the connection, marshalling and unmarshalling the objects, sending them over the network, and the actual KANT computation took only 1.2 seconds in total. This demonstrates the flexibility of the SCSCP approach: the most appropriate system may be used for the task at hand. Users are no longer restricted to performing all aspects of a required computation in a single system that may not provide good support for all required operations.

5. INFRASTRUCTURE FOR PARALLEL COMPUTATIONS

In contrast to notations for numerical computations, which have an emphasis on floating point arithmetic, monolithic arrays, and programmer-controlled memory allocation, symbolic computing has an emphasis on functional notations, greater interactivity, very high level programming abstractions, complex data structures, automatic memory management, etc. With this different evolutionary path, it is not surprising that symbolic computation has parallelisation requirements that differ significantly from those for traditional *numerical* high-performance computing. In particular, parallel symbolic computations are often highly irregular, need to exploit more complex data structures than their numerical counterparts, and exhibit sophisticated computational patterns that are only just being identified.

We have developed a number of tools that exploit the capabilities of SCSCP for marshaling/unmarshaling symbolic data as part of a parallel computation, outlined below: SPSD (a middleware written in Java using the SCSCP API); the master-worker skeleton implemented directly in GAP; and a general programmable framework for parallelism, **SymGrid-Par**. These provide increasing levels of capability and scalability.

5.1 WUPSI/SPSD

The Java framework outlined above [28] has been used to construct "WUPSI", an integrating software component that is a universal Popcorn SCSCP Interface providing several different technologies for interacting with SCSCP clients and servers. One of these is the Simple Parallel SCSCP Dispatcher (SPSD), which allows very simple patterns like parallel map or zip to be used on different SCSCP servers simultaneously. The parallelization functionality is offered as an SCSCP service itself, so it can be invoked not only from the WUPSI command line, but also by any other SCSCP client. Since WUPSI and all parts of it are open source and freely available, they can be exploited to build whatever infrastructure seems necessary for a specific use case.

5.2 GAP Master-Worker Skeleton

Using the SCSCP package for GAP, it is possible to send requests to multiple services to execute them in parallel, or to wait until the fastest result is available, and implement various scenarios on top of the provided functionality. One of these is the master-worker skeleton, included in the package and implemented purely in GAP. The client (i.e. master, which orchestrates the computation) works in any system that is able to run GAP, and it may even orchestrate both GAP based and non-GAP based SCSCP servers, exploiting such SCSCP mechanisms as transient content dictionaries to define OpenMath symbols for a particular operation that exists on a specific SCSCP server, and remote objects to keep references to objects that may be supported only on the other CAS. It is quite robust, especially for stateless services: if a server (i.e. worker) is lost, it will resubmit the request to another available server. Furthermore, it allows new workers (from a previously declared pool of potential workers) to be added during the computation. It has flexible configuration options and produces parallel trace files that can be visualised using EdenTV [5]. The master-worker skeleton shows almost linear (e.g. 7.5 on 8-core machine) speedup on irregular applications with low task granularity and no nested parallelism. The SCSCP package manual [30] contains further details and examples. See also [13, 31] for two examples of using the package to deal with concrete research problems.

5.3 SymGrid-Par

SymGrid [25] provides a new framework for executing symbolic computations on *computational Grids*: distributed parallel systems built from geographically-dispersed parallel clusters of possibly heterogeneous machines. It builds on and extends standard Globus toolkit [23] capabilities, offering support for discovering and accessing Web and Grid-based symbolic computing services (**SymGrid-Services** [9]) and for orchestrating symbolic components into Grid-enabled applications (**SymGrid-Par** [2]). Both of these components build on SCSCP in an essential way. Below, we will focus on **SymGrid-Par**, which aims to orchestrate multiple sequential symbolic computing engines into a single coherent parallel system.

5.3.1 Implementation details

SymGrid-Par (Figure 1) extends our implementation of the Gum system [4, 41], a message-based portable parallel implementation of the widely used purely functional language Haskell [38] for both shared and distributed memory architectures. **SymGrid-Par** comprises two generic interfaces: the "*Computational Algebra system to Grid middleware*" (**CAG**) interface links a CAS to Gum; and the "*Grid middleware to Computational Algebra system*" (**GCA**) interface conversely links Gum to a CAS. The CAG interface is used by computational algebra systems to interact with Gum. Gum then uses the GCA interface to invoke remote computational algebra system functions, to communicate with the CAS etc. In this way, we achieve a clear separation of concerns: Gum deals with issues of thread creation/coordination and orchestrates the CAS engines to work on the application as a whole; while each instance of the CAS engine deals solely with execution of individual algebraic computations.

The GCA interface interfaces our middleware with a CAS, connecting to a small interpreter that allows the invocation of arbitrary computational algebra system functions, marshaling/unmarshaling data as required. The interface comprises both C and Haskell components. The C component is mainly used to invoke operating system services that are needed to initiate the computational algebra process, to establish communication channels, and to send and receive commands/results from the computational algebra system process. It also provides support for static memory that can be used to maintain state between calls. The Haskell component provides interface functions to the user program and implements the communication protocol with the computational algebra process.

The CAG interface comprises an API for each symbolic system that provides access to a set of common (and po-

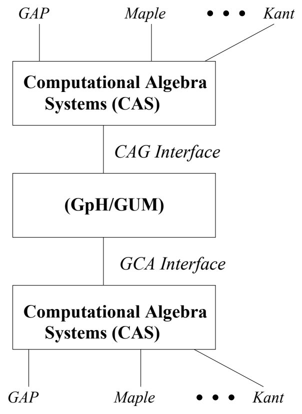
GAP
Maple
• • •
Kant
Computational Algebra Systems (CAS)
CAG Interface
(GpH/GUM)
GCA Interface
Computational Algebra Systems (CAS)
GAP
Maple
• • •
Kant

Figure 1: SymGrid-Par Design Overview

tentially parallel) patterns of symbolic computation. These patterns form a set of dynamic *algorithmic skeletons* (see [10]), which may be called directly from within the computational algebra system, and which may be used to orchestrate a set of sequential components into a parallel computation. In general (and unlike most skeleton approaches), these patterns will be nested and can be dynamically composed to form the required parallel computation. Also, in general, they may mix components taken from several different computational algebra systems.

5.3.2 Standard Parallel Patterns

The standard patterns we have identified are listed below. The patterns are based on commonly-used sequential higher-order functions that can be found in functional languages such as Haskell. Similar patterns are often defined as algorithmic skeletons. Here, each argument to the pattern is separated by an arrow (`->`), and may operate over lists of values (`[..]`), or pairs of values (`(..,..)`). All of the patterns are *polymorphic*: i.e. `a`, `b` etc. stand for (possibly different) concrete types. The first argument in each case is a function of either one or two arguments that is to be applied in parallel.

```
parMap::       (a->b) -> [a] -> [b]
parZipWith::   (a->b->c) ->[a] -> [b] -> [c]
parReduce::    (a->b->b) -> b -> [a] -> b
parMapReduce::(d->[a]->b) -> (c->[(d,a)]) -> c -> [(d,b)]
masterSlaves::((a->a)->(a->a->b)) -> [(a,a)] -> [(a,a,b)]
```

So, for example, `parMap` is a pattern taking two arguments and returning one result. Its first argument (of type `a->b`) is a function from some type `a` to some other type `b`, and its second argument (of type `[a]`) is a list of values of type `a`. It returns a list of values each of type `b`. Operationally, `parMap` applies a function argument to each element of a list, in parallel, returning the list of results, e.g.

```
parMap double [1,4,9,16]  ==  [2,8,18,32]
  where double x = x + x
```

It thus implements a parallel version of the common `map` function, which applies a function to each element of list. The `parZipWith` pattern similarly applies a function, but in this case to two arguments, one taken from each of its list arguments. Each application is performed in parallel, e.g.

```
parZipWith add [1,4,9,16] [3,5,7,9]  ==  [4,9,16,25]
  where add x y = x + y
```

Again, this implements a parallel version of the `zipWith` function that is found in functional languages such as Haskell. Finally, `parReduce` reduces its third argument (a list of type `[a]`) by applying a function (of type `a->b->b`) between pairs of its elements, ending with the value of the same type `b` as its second argument; `parMapReduce` pattern combines features of both `parMap` and `parReduce`, first generating a list of key-value pairs from every input item (in parallel), before reducing each set of values for one key across these intermediate results; `masterSlaves` is used to introduce a set of tasks and generate a set of worker processes to apply the given function parameter in parallel to these tasks under the control of a coordinating master task. The `parReduce` and `parMapReduce` patterns are often used to construct parallel pipelines, where the elements of the list will themselves be lists, perhaps constructed using other parallel patterns. In this way, we can achieve nested parallelism. [3] contains further details on **SymGrid-Par**, including the description of several experiments and a detailed analysis of their parallel performance.

6. CONCLUSIONS

We have presented a framework for combining computer algebra systems using a newly-developed remote procedure call protocol SCSCP (Symbolic Computation Software Composability Protocol). By defining common data and task interfaces for all systems, we allow complex computations to be constructed by orchestrating heterogeneous distributed components into a single symbolic application. Any system supporting SCSCP can immediately connect to all other SCSCP-compliant systems, thus avoiding the need for special cases and minimizing repeated efforts. Furthermore, if some CAS changes its internal format then it only needs to update one interface, namely that to the SCSCP protocol (instead of as many interfaces as there are programs it connects to). Moreover, this change can take place completely transparently to the other CAS connecting to it.

We have demonstrated several examples of setting up communication between different CAS, thus exhibiting SCSCP benefits and features including its flexible design, the ability to solve problems that can not be solved in the "home" system, and the possibility to speed up computations by sending request to a faster CAS. Finally, we have shown how sequential systems can be combined into heterogeneous parallel systems that can deliver good parallel performance.

SCSCP uses an OpenMath representation to encode both transmitted data and protocol instructions, and may be supported not only by a CAS, but by any other software as well. To achieve this, it is necessary only to support SCSCP messages accordingly to the protocol specification, while the support of particular OpenMath constructions and objects is dictated only by the nature of the application. This support may consequently be limited to a few basic OpenMath data types and a small set of application-relevant symbols. For example, a Java applet to display the lattice of subgroups of a group may be able to draw diagrams for partially ordered sets without any support for the group-theoretical OpenMath CDs. Other possible applications may include a web or SCSCP interface to a mathematical database, or, as an extreme proof-of-concept, even a server providing access to a computer algebra system through an Internet Relay Chat bot.

Additionally, SCSCP-compliant middleware may look inside an SCSCP message, extracting all necessary technical information from its outer levels and taking the embedded OpenMath objects as a "black box". This approach is essentially used in the **SymGrid-Par** middleware (Section 5) which performs marshaling and unmarshaling of OpenMath-represented data between CASes. By exploiting well-established adaptive middleware (GUM), we can manage complex irregular parallel computations on clusters and shared-memory parallel machines. This allows us to harness a number of advanced GUM features that are important to symbolic computations, including: automatic control of task granularity, dynamic task creation, implicit asynchronous communication, automatic sharing-preserving data marshaling/unmarshaling, ultra-lightweight work stealing and task migration, virtual shared memory, and distributed garbage collection.

We have already seen examples of SCSCP-compliant software that were created outside the SCIEnce project and we hope that we will have more of them in the future. We anticipate that existing and emerging SCSCP APIs will be useful here as templates for new APIs. In conclusion, SCSCP is a powerful and flexible framework for combining CAS, and we encourage developers to cooperate with us in adding SCSCP support to their software.

7. ACKNOWLEDGMENTS

"SCIEnce – Symbolic Computation Infrastructure in Europe" (`www.symbolic-computation.org`) is supported by EU FP6 grant RII3-CT-2005-026133. We would like to acknowledge the fruitful collaborative work of all partners and CAS developers involved in the project.

8. REFERENCES

[1] A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer Algebra meets Automated Theorem Proving: Integrating Maple and PVS. In *Proc. TPHOLs 2001: Intl. Conf. on Theorem Proving in Higher Order Logics, Springer LNCS 2152*, pages 27–42, 2001.

[2] A. Al Zain, K. Hammond, P. Trinder, S. Linton, H.-W. Loidl, and M. Costantini. SymGrid-Par: Designing a Framework for Executing Computational Algebra Systems on Computational Grids. In *Proc. ICCS '07: 7th Intl. Conf. on Computational Science, Springer LNCS 4488*, pages 617–624, 2007.

[3] A. Al Zain, P. Trinder, K. Hammond, A. Konovalov, S. Linton, and J. Berthold. Parallelism without pain: Orchestrating computational algebra components into a high-performance parallel system. In *Proc. IEEE Intl. Symp. on Parallel and Distributed Processing with Applications (ISPA 2008), Sydney, Australia*, pages 99–112, 2008.

[4] A. Al Zain, P. Trinder, G. Michaelson, and H.-W. Loidl. Evaluating a High-Level Parallel Language (GpH) for Computational GRIDs. *IEEE Trans. Parallel Distrib. Syst.*, 19(2):219–233, 2008.

[5] J. Berthold and R. Loogen. Visualizing parallel functional program runs: Case studies with the eden trace viewer. In *Proc. PARCO 2007: Intl. Conf. on Parallel Computing: Architectures, Algorithms and Applications*, volume 15 of *Advances in Parallel Computing*, pages 121–128. IOS Press, 2007.

[6] H. Besche and B. Eick. Construction of finite groups. *J. Symbolic Comput.*, 27(4):387–404, 1999.

[7] H. Besche, B. Eick, and E. O'Brien. *The Small Groups Library.* `http://www-public.tu-bs.de:8080/~beick/soft/small/small.html`.

[8] J. Cannon and W. Bosma (Eds.). *Handbook of Magma Functions, Edition 2.15*, 2008. School of Mathematics and Statistics, University of Sydney `http://magma.maths.usyd.edu.au/`.

[9] A. Cârstea, M. Frîncu, G. Macariu, D. Petcu, and K. Hammond. Generic Access to Web and Grid-based Symbolic Computing Services: the SymGrid-Services Framework. In *Proc. ISPDC 07: Intl. Symp. on Parallel and Distributed Computing, Castle Hagenberg, Austria, IEEE Press*, pages 143–150, 2007.

[10] M. Cole. Algorithmic Skeletons. In *Research Directions in Parallel Functional Programming*, chapter 13, pages 289–304. Springer-Verlag, 1999.

[11] M. Costantini, A. Konovalov, and A. Solomon. *OpenMath – OpenMath functionality in GAP, Version 10.1*, 2010. GAP package. `http://www.cs.st-andrews.ac.uk/~alexk/openmath.htm`.

[12] M. Daberkow, C. Fieker, J. Klüners, M. Pohst, K. Roegner, M. Schörnig, and K. Wildanger. KANT V4. *J. Symbolic Comput.*, 24(3-4):267–283, 1997. Computational Algebra and Number Theory, 1993.

[13] B. Eick and A. Konovalov. The modular isomorphism problem for the groups of order 512. In *Groups St. Andrews 2009*, London Math. Soc. Lecture Note Ser. (Accepted).

[14] S. Freundt, P. Horn, A. Konovalov, S. Lesseni, S. Linton, and D. Roozemond. OpenMath in SCIEnce: Evolving of symbolic computation interaction. In proceedings of OpenMath Workshop 2009 (to appear).

[15] S. Freundt, P. Horn, A. Konovalov, S. Linton, and D. Roozemond. Symbolic Computation Software Composability Protocol (SCSCP) specification. `http://www.symbolic-computation.org/scscp`, Version 1.3, 2009.

[16] S. Freundt, P. Horn, A. Konovalov, S. Linton, and D. Roozemond. Symbolic computation software composability. In *AISC/MKM/Calculemus, Springer LNCS 5144*, pages 285–295, 2008.

[17] S. Freundt and S. Lesseni. *KANT 4 SCSCP Package.* `http://www.math.tu-berlin.de/~kant/kantscscp.html`.

[18] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, 2008. `http://www.gap-system.org`.

[19] M. Gastineau. *SCSCP C Library – A C/C++ library for Symbolic Computation Software Composibility Protocol, Version 0.6.0.* IMCCE, 2009. `http://www.imcce.fr/Equipes/ASD/trip/scscp/`.

[20] M. Gastineau. Interaction between the specialized and general computer algebra systems using the SCSCP protocol. Submitted.

[21] M. Geck, G. Hiss, F. Lübeck, G. Malle, and G. Pfeiffer. CHEVIE – A system for computing and processing generic character tables for finite groups of Lie type, Weyl groups and Hecke algebras. *Appl. Algebra Engrg. Comm. Comput.*, 7:175–210, 1996.

[22] I. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic SBDD Using Computational Group Theory. In *Proc. CP 2003: Intl. Conf. on Principles and Practice of Constraint Programming, Kinsale, Ireland*, pages 333–347, 2003.

[23] Globus toolkit. `http://www.globus.org/toolkit/`.

[24] D. Grayson and M. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at `http://www.math.uiuc.edu/Macaulay2/`.

[25] K. Hammond, A. Al Zain, G. Cooperman, D. Petcu, and P. Trinder. SymGrid: a Framework for Symbolic Computation on the Grid. In *Proc. EuroPar'07*, LNCS, Rennes, France, August 2007.

[26] P. Horn. *MuPAD OpenMath Package*, 2009. `http://mupad.symcomp.org/`.

[27] P. Horn and D. Roozemond. *java.symcomp.org – Java Library for SCSCP and OpenMath.* `http://java.symcomp.org/`.

[28] P. Horn and D. Roozemond. *WUPSI –Universal Popcorn SCSCP Interface.* `http://java.symcomp.org/wupsi.html`.

[29] P. Horn and D. Roozemond. OpenMath in SCIEnce: SCSCP and POPCORN. In *Intelligent Computer Mathematics – MKM 2009*, volume 5625 of *Lecture Notes in Artificial intelligence*, pages 474–479. Springer, 2009.

[30] A. Konovalov and S. Linton. *SCSCP – Symbolic Computation Software Composability Protocol, Version 1.2*, 2010. GAP package. `http://www.cs.st-andrews.ac.uk/~alexk/scscp.htm`.

[31] A. Konovalov and S. Linton. Parallel computations in modular group algebras. (Submitted,2010).

[32] F. Lübeck and M. Neunhöffer. *GAPDoc – A Meta Package for GAP Documentation*, 2008. `http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc`.

[33] Maple. `http://www.maplesoft.com/`.

[34] MuPAD. `http://www.sciface.com/`.

[35] M. Neunhöffer. *IO – Bindings for low level C library IO*, 2009. `http://www-groups.mcs.st-and.ac.uk/~neunhoef/Computer/Software/Gap/io.html`.

[36] E. O'Brien, W. Nickel, and G. Gamble. *ANUPQ – ANU p-Quotient, Version 3.0*, 2006. `http://www.math.rwth-aachen.de/~Greg.Gamble/ANUPQ/`.

[37] OpenMath. `http://www.openmath.org/`.

[38] S. Peyton Jones (ed.), J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge University Press, April 2003.

[39] Sage. `http://www.sagemath.org/`.

[40] L. Soicher. Computing with graphs and groups. In *Topics in Algebraic Graph Theory*, pages 250–266. Cambridge University Press, 2004.

[41] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *Proc. PLDI '96: Intl. Conf. on Programming Language Design and Implementation, Philadelphia, PA, USA*, pages 79–88, May 1996.