Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 114 (2005) 103–117
www.elsevier.com/locate/entcs

Mathematical Services Composition

Yannis Chicha[1] Marc Gaëtano[2]

I3S/Université de Nice-Sophia Antipolis
ESSI, 930 route des Colles, BP 145
06903 Sophia Antipolis Cedex, France

Abstract

This paper describes the definition and the use of a plan language in the context of mathematical web services. A plan is a document intended to describe how to use different mathematical web services to solve a particular problem. A plan is like a program in which most of the function calls have to be handled by web services. A plan is a multiple-state choreography document which could be either abstract, unresolved or resolved, depending on how much of the web services involved in the choreography is known. Such a plan can be instantiated into a composition language such as BPEL or to a mathematical routine (like a Maple routine) for execution.

Keywords: Mathematical Web services, Service Composition, Mathematical Computation.

1 Introduction

Web services may offer an interesting solution to the use of heterogeneous mathematical packages in a common environment. MONET [20], a European Union funded project aims at demonstrating the applicability of the latest ideas for creating a semantic web to mathematical software. In the framework defined by this project, a mathematical package becomes a web service providing various functions through a common description scheme. These routines implement simple to sophisticated algorithms corresponding to highly specific mathematical functions, which need to be combined in order to solve a problem. This paper presents a mechanism for composing mathematical web services based on multiple-state choreography documents called "plan"s. An

[1] Email: chicha@essi.fr
[2] Email: gaetano@essi.fr

doi:10.1016/j.entcs.2004.02.069

abstract plan is an algorithm involving mathematical functions described by ontologies or classifications such as GAMS [23]. *Unresolved* plans are abstract plans for which some but not all mathematical functions have been replaced by actual invocations of mathematical web services. Resolved plans are abstract plans for which all mathematical functions have been replaced by actual invocations of corresponding mathematical web services.

In Section 2, we explain why the web service technology may provide the computational mathematics community with a powerful, flexible and standard framework to solve mathematical problems over the Internet. This framework is centered on the concept of brokering services. A broker is a dedicated web service that can be contacted to locate appropriate mathematical servers to solve a given problem. We describe the general architecture designed in the MONET project and we discuss the role of each component of the broker. In Section 3, we explain the importance of composition in the context of mathematical computation and introduce the notion of abstract, unresolved and resolved plan. We give some insight on the possible translation of a plan into a suitable document (like a BPEL document or a Maple routine) in order to be actually computed. Section 4 describes related work. Finally, after concluding remarks, we describe possible future directions of this work.

2 Mathematical Web Services

Mathematical computing has been a major branch of computer science since the early ages. Both numerical and symbolic computation brought up many highly specialized and efficient pieces of software. This section first describes motivations for this work and then proposes a possible architecture for a web service-based framework to discover and invoke mathematical packages.

2.1 Motivation

Because mathematics are so fundamental, numerous packages have been developed for both numerical and symbolic computation. Although created in an academic context, most of them are fully documented and regularly updated and some became well-known, commercial products, like Maple and Mathematica for symbolic computation, and the NAG libraries for numerical computation. On hot topics, one can find many different packages solving the same problem. For example, Macaulay [13], Gb [14], and Singular [15] offer very efficient routines to compute Groebner bases.

Unfortunately, most of these mathematical packages remain unknown from a significant part of their potential users: too difficult to use for non-specialists, not available on the user's platform or environment, or simply not advertised

enough, mathematical packages rarely evolve beyond the stage of prototypes. A natural way to improve the accessibility of these packages is to turn them into mathematical web services, providing:

- automated service discovery based on semantic matching.
- uniform access to routines.
- composition of services to solve complex problems.

The primary application for web services in the world of mathematical software is to simplify point-to-point integration between systems, thereby allowing the use of one package from within another. Typical examples include the use of a highly specialized package from a general purpose computer algebra system like Maple [19] or Mathematica [29]. For example, [6] describes how to use the functions of a package called Bernina from Maple. This application, however, only scratches the surface of the true potential of mathematical web services. Service oriented computing can enable users to access agile mathematical processes that can adapt and respond to high level queries, through the use of loosely coupled, standards-based mathematical services. Unlike general web services, mathematical services can be organized according to the problem they are able to solve. These problems themselves can be organized as a tree structure reflecting the natural inheritance between mathematical problems.

Another motivation for the use of web services is the commercial exploitation of mathematical packages. A natural exploitation plan for a company would be to sell access to mathematical routines or libraries. In this context, there would be concerns such as billing and transaction (see [27] on this topic) and security (see WS-Security [16], for example). However, this topic is outside the scope of this paper and we will mainly focus on the composition of services.

2.2 General architecture

The MONET framework is centered on the concept of brokering services. A broker is a dedicated web service that can be contacted to locate appropriate mathematical servers to solve a given problem. Unlike discovery of services in other areas of application, precise semantic information is essential to localize mathematical packages. For example, many mathematical algorithms may be called *solve* because they compute a solution for a given type of equation or set of equations. However, these equations may be of very different types and may require completely different services to be solved. Alone, the name of the operation implementing an algorithm can not be trusted. Complementary data on the problem solved is required for automated service discovery. That

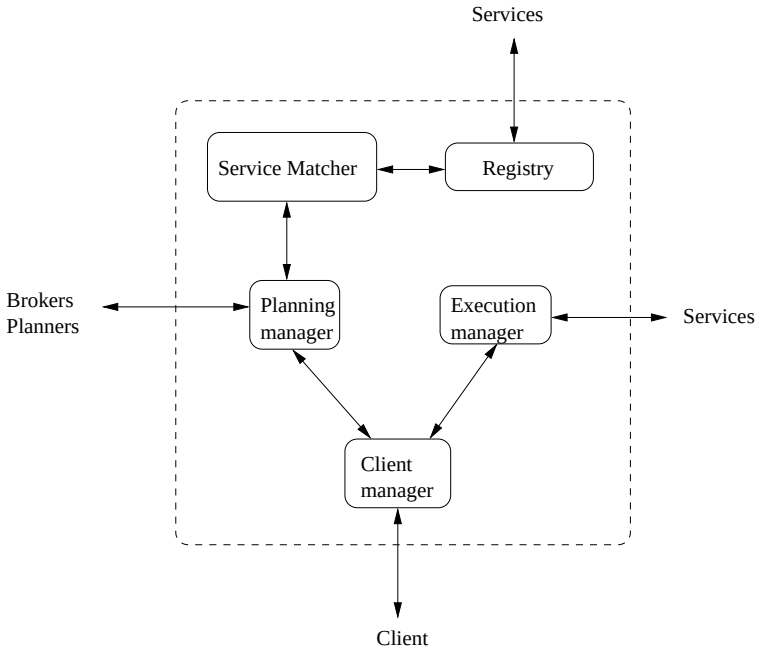

Fig. 1. Broker architecture

is why semantics is crucial in the description of a service – it is not realistic to simply "assume" that a mathematical package is appropriate for a given problem. Semantic matching for mathematical operations and the architecture of the framework are discussed in more details in the publications of the MONET project (see [21]). For example, [26] describes semantic matching in our architecture and [11] explains why UDDI does not provide sufficient semantic support to advertise and discover mathematical services. There is current work in the project to understand whether DAML-S/OWL-S [10] would be suitable for our context.

We provide here an overview of the architecture we are using. The MONET project refines the notion of *broker* in the semantic web and makes it the central component in the process of discovering and calling a mathematical service. The broker itself is viewed as a special service which typically assists clients in identifying suitable computational services for the problem they wish to solve. Figure 1 presents the architecture of this special service.

The Client Manager handles communication with clients and hides the details of the broker. Its job is to forward clients' queries to the appropriate component. The Service Matcher (SM) uses a Registry to store information about the functionalities registered by services. In particular, it stores information about how to contact the service. The SM also has a role of reasoning

about functionalities, exploring mathematical classifications. Reasoning is performed by the Instance Store [18], which is based on the DIG [12] language developed at the University of Manchester. The Planning Manager allows the composition of services to solve mathematical problems. The idea is that a query received by the broker from a client may not be matched by the SM. The Execution Manager (EM) executes plans created by the Planning Manager. Even though plans are typically sent back to clients for approval and execution, our architecture allows clients to request a plan to be executed by the broker. The entity responsible for this execution (and for transaction management related to this execution) is the EM.

2.3 Example

In the context of the MONET project, we are experimenting with a few symbolic mathematical packages, including Bernina, an interactive interface to the `Sum^it` [7] library. This library provides some efficient computations revolving around differential operators in $Q[x, d/dx]$ or $Q(x)[d/dx]$. A detailed presentation of this experiment can be found at [9]. Certain functions of Bernina would better benefit to the community if exposed as web services. Such functions – efficiently implementing non-trivial algorithms related to differential operators – solve very specific problems. Simple access to them (i.e. in a standard way and without installing the software) will increase the visibility and reachability of Bernina.

As for many existing applications, one obstacle to providing web service access to Bernina's functions is that they are implemented using a language (here Aldor [2]) that does not provide web services functionalities like Java or a .NET language do. Consequently, adding a web service capability to Bernina reveals non-natural. In order to make this happen, it is necessary to either develop a web services library within the environment in which Bernina was written, or, more simply, to wrap the operations of Bernina in a web services compliant platform such as Java (Figure 2 illustrates such a wrapping). In this case, there is no need to modify the code of Bernina (even re-compilation is not necessary), because we are wrapping its functionalities into a Java program. As can be seen on Figure 2, the service exposing Bernina's operations, through an MSDL (Mathematical Service Description Language, see [22]) document, is composed of the Bernina executable, a Java front-end (the actual service), and an adapter that makes the link. The Java front-end allows us to "import" Bernina into the world of web services. In the OpenMath [25] terminology, such a wrapper is called a *phrasebook*. Also, the formats (Maple or Lisp) of the objects manipulated by Bernina are known but not standard. Although the MONET framework allows the use of such languages, it is recommended to

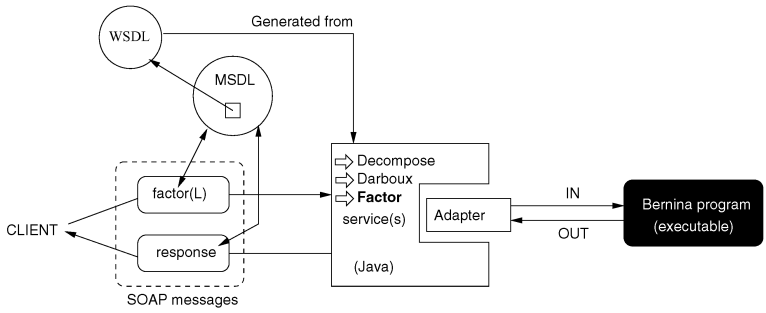

Fig. 2. Exposing Bernina operations as web services

rely on standard languages such OpenMath or MathML [28]. In our current experiments, our solution is to use MONET languages to expose Bernina's functions as mathematical web services and OpenMath to represent objects.

3 Composition

We call composition of services a mechanism to combine such services in order to achieve a given goal. We call "plan" such a composition, a plan in the MONET framework is different from plans in AI planning (use of such techniques for semantic web services composition is being studied by several people, but no conclusion has been reached yet, see [8] for example). In our case, a plan is rather a "script" similar to Maple procedures. Typical users do not wish to simply invoke one routine performing one computation. A more realistic use case would be to allow users to express a sequence of computation combining several services. In our framework, composition is an alternative or rather a complementary tool to service discovery: whenever a service is not available, a composition of services may solve the submitted problem. Composition is thus essential to using web services in the context of mathematics. This section discusses such planning and presents our solution.

3.1 Motivation

In our architecture (see Section 2.2), finding an appropriate service for a given query is the job of the Service Matcher. This entity queries the Registry and returns information on a service solving the submitted problem. Obviously, such a registry is not universal and, thus, there is no guarantee that a match will be found. In this case, a multi-step strategy may be known to solve the problem. Such a strategy takes the form of a document that we call "plan" and which contains a series of algorithmic instructions to reach a solution to the problem. For each step, a service invocation should be made. The Service

Matcher is thus queried again but for different, probably simpler, problems. Once again, matches may or may not be found: mathematical plans do not guarantee a solution, but propose a different technique to answer a query.

Another aspect of the issue is similar to solving problems in the context of specialized libraries or interactive mathematical systems. There exists a number of routines available, but users almost never make just one call to such routines. They rather use a language provided by the systems to combine the calls in more complete algorithms suited to their needs. Mathematical web services only offer interfaces to invoke one routine or another. We propose to get back the flexibility and usability of known interactive systems by using plans and the planning environment we setup in our architecture. A planning language would allow end-users to build programs combining routine calls. Such programs would be analyzed for optimization by the Planning Manager and executed by the Execution Manager.

Plans can be obtained from two sources: human and program. We notice that, even in a specific area of mathematics, being able to automatically analyze a problem and produce a plan "on-the-fly" is very difficult. Obviously, we can not expect the planning manager to solve any general problem. Most of the time, it cannot even decide by simply "looking" at a problem what domain is involved in a given query. This will depend on the classification chosen by the client to describe this problem. We foresee that a more common scenario would be to use human-made plans. People manually produce plans using a "mathematical plan language" and submit them to the Planning Manager. Also, note that automatically-generated plans may be submitted to the clients for approval and/or modification. Human-made plans are most likely to be widespread because they will cover any type of operation and may contain "tricks" that can not be inferred by a program. For convenience and reusability, human-made or automatically-generated plans can be gathered into repositories that the Planning Manager can query.

3.2 Plan language

In order to compose services, a "plan language" proves necessary. The reason is that services can not just be invoked in sequence transmitting values along the way. More subtle operations should be supported such as conditional expressions, loops, and so on. There are currently several composition languages to choose from (BPEL [4], WSCI [5], ...). Unfortunately, it is difficult to know what language will actually become a standard. Consequently, one can not rely on one formalism or another and we decided, for the time being, to produce our own language inspired from BPEL. The goals for this language are:

- **independence**: if the specification of a given composition language changes, we should not have to reflect it in our tools.
- **simplicity**: the translation with BPEL should be straightforward, thus allowing the use of existing tools (e.g. IBM's BPWS4J [17]).
- **flexibility**: our language should be flexible enough to allow the translation into a number of target languages. As we are going to see in Section 3.3, one possibility would be to use a computer algebra system such as Maple [19] to execute plans.

3.2.1 Abstract plan

The plans constructed by the language we propose are two-fold. There is an abstract part in which no notion of web service exists, and a resolved part which adds the necessary information to invoke the services. Abstract plans generated by our Mathematical Services Planning Language are similar to actual mathematical algorithm. The important point is that we allow users to write their own composition using a language sufficiently expressive to combine operations (which may then be mapped onto services). The use case is like Maple's or Mathematica's use case. Both systems provide efficient mathematical routines to solve a number of problems. Fortunately, they also provide an environment with a small language able to provide sufficient glue to combine routine calls. That allows users to create on-the-fly specific algorithms to solve their needs. In the architecture, one can see that the broker plays a central role. In a sense, a planning/composition language shows the broker as a computing system similar to Mathematica or Maple, with a difference: routines are available on various nodes of the Internet. That is the reason why abstract plans can be created independently from any notion of web service.

The actual language is still a work-in-progress. We present here a first draft, which is a subset of both the BPEL and Maple languages. There is nothing surprising here, as it corresponds to an algorithmic language. The idea was to start from one or two known languages to benefit from the experience of the conceptors and the tools developed for it. We will use this plan language for our experiments and tune it to adapt to the mathematical computing world.

- **Control structures**: loops, conditionals, sequencing, invocation. We also add the notion of parallel execution (available in BPEL). Parallel execution can typically be required when a client asks to run the same computation with several services and obtain the first available result. Also, traditional parallel computation can be of benefit when there are independent branches of the plan for example.

- **Assignment and local variables**: as for most languages, temporary variables prove very useful for expressiveness. Our case is no different and we use this feature.
- **Types and objects**. Simple data structures (lists and arrays) should suffice for most operations. We do not create a programming language, but a kind of scripting language. Too sophisticated features would be confusing and may help defeat the purpose of web services (by using the language to program computations rather than invoke them). Mathematical objects manipulated by the plan language should not be of a fixed formalism. Ideally, a standard format (OpenMath or MathML) should be used. However, it is difficult to impose such a choice. Consequently, our language includes a parameter to specify the format of objects manipulated in a document. This format is identified by a namespace as we evolve in an XML world.

The elements described above should be sufficient to function reasonably well with most mathematical services. As will be described in Section 5, a (near) future work is to experiment with several users and services to understand more specific needs. A current issue is to decide whether an abstract plan may contain indications stating that such or such step should be carried out by a service with given characteristics. The advantage would be to further specify a plan, the drawback is that this would introduce the notion of web services inside an abstract document.

3.2.2 Resolved and unresolved plans

The *resolved* component of a plan contains instructions to associate Service Description Language documents to each invocation statement. In the plan language, we simply add the possibility to annotate each invocation statement with the necessary contact information of services. In our architecture, this would mean that invocations are associated with Service Description documents (or at least with URIs referencing such documents). Unresolved plans are plans for which not all invocation statements are annotated (an abstract plan is a special kind of unresolved plan in which *no* invocation statement has been annotated). A specificity of resolved plans is that one should be able to translate them into an execution language *without* any more manipulation. This means that a plan is also deemed unresolved when an invocation statement is associated with several possible services. The only exception would be when the author of the plan wishes to execute the same step in parallel by several services.

3.2.3 Example: Finding real poles

We provide here an example of planning to solve a mathematical problem that would not typically be available as one operation exported by a package:

Problem: "Find the real poles of"
Input: $F(X) = \frac{X^2-X-2}{X^4+X^3-X^2+X-2}$

No package usually provides such operation, so the Service Matcher would not find any service returning the real poles of a fraction. Fortunately, it is likely that a plan exists to solve this problem. Here is an example of such a plan:

Problem: "Find the real poles of a rational fraction $F(X) = \frac{N(X)}{D(X)}$"
Steps:
(i) extract $D(X)$ the denominator of $F(X)$
(ii) compute $S = \{X_1, X_2, ..., X_n\}$, the set of solutions of the equation $D(X) = 0$
(iii) extract the subset $R = \{X_{\alpha_1}, X_{\alpha_2}, ..., X_{\alpha_k}\}$ of the real values of S

Plans can thus be considered as an orchestration of various mathematical routines to create "on-the-fly" an otherwise unavailable service. In our example, an instantiation of the steps would be:

(i) extract the denominator: $D(X) = X^4 + X^3 - X^2 + X - 2$
(ii) compute the solutions of $D(X) = 0$: $S = \{i, -i, 1, -2\}$
(iii) extract the subset R of the real values of S

This would lead to the result: $R = \{1, -2\}$.

We now show a version of this plan using our language. Note that we show here the *abstract* version of the plan. We will see that this is quite close to the informal description presented above.

```
RealPoles(f: fraction)
   sequence
        assign(d, invoke("denom",f))
        assign(equation, invoke("apply","=",d,0))
        assign(s, invoke("solve",equation)
        assign(result, invoke("extractrv", s))
        return(result)
```

The only surprising element in this example is the second invocation: `invoke("apply","=",d,0)`. This instruction illustrates the fact that we might need to create local objects during the execution of a plan. In this example, we create an equation (`d = 0`) that should then be solved by invoking a "solving service" in the next step. The invocation of an "apply" operation should prob-

ably be handled locally (within the execution engine). Section 3.3 provides further details about local invocations. Also, the creation of a new object is typically an abstract operation that should be instantiated depending on the formalism we choose for manipulating objects.

3.3 Execution and local computation

The MONET architecture contains an Execution Manager (EM). This entity is responsible for supervising the invocation of mathematical web services. The sophistication of the EM is implementation-dependent and can range from a simple service invocation module to a secured, transaction-compliant, and plan-compliant system. The EM itself is a web service similar to the way Planning Manager and Service Matcher are web services. In our prototype, the Execution Manager receives a resolved plan as an input and should take all necessary actions to execute it.

We distinguish two possibilities to achieve this result: either the Execution Manager directly executes a resolved plan handling, as needed, concerns such as transactions, security, and so on, or it relies on a third-party composition language such as BPEL used to transform the resolved plan. An orchestration engine (e.g. BPWS4J) can then proceed with the execution. Once again, we see here the advantage of choosing an independent language: we are not tied to one standard, and the actual execution mechanism is a simple back-end that can be changed easily. One can also imagine to use execution environments that are not typical web services orchestration engines. In the context of mathematical services, a possibility would be to use Maple (or Mathematica, for example) as an execution engine: the resolved plan can be translated into the Maple language, which is a kind of planning language, and submit it to a Maple system. With a few added functions Maple can call web services, and obtain and manipulate the results.

We provide here the Maple version of the Real Poles example presented in the previous section. Note that we assume a special Maple routine called "invokeService", which is responsible for invoking web services. This version typically corresponds to a composition document translated from a resolved plan. `DenomWSDLURI`, `SolveWSDLURI`, and `ExtractrvWSDLURI` are Maple variables that contain a pointer to the contact information for the given services, the values for these variables are taken from the annotations of the resolved plan.

```
DenomWSDLURI := ###URI for the Denom service###
SolveWSDLURI := ###URI for the Solving service###
ExtractrvWSDLURI := ###URI for the Extractrv service###
```

```
RealPoles := proc(f) local d, equation, s, result;
   d := invokeService("denom", f, DenomWSDLURI);
   equation := d = 0;
   s := invokeService("solve", equation, SolveWSDLURI);
   result := invokeService("extractrv", s, ExtractrvWSDLURI);
   return result;
end proc;
```

It is easy to see that even abstract plans can be expressed using Maple, however there is an advantage in using a Maple system as an orchestration engine (of course, this would require a web service library, which is reasonable to assume). In the "Real poles" example, we find three steps: extract denominator, solve equation, and extract real values. Two of these steps (extraction of denominator and real values) are very simple and it would be a waste of resources to invoke operations to perform these steps (if a service even decides to export such operations). This is a real problem, because, without those operations, the plan can not be carried out. In this case, a local computation engine such as Maple could help and perform these steps locally. The operation "solve equation" can then be done by any given service.

Local computation for trivial operations appears as a very useful feature of mathematical web services. Whenever an operation can be done locally (what these operations are can be decided at implementation, deployment or even run-time), a local computation system can be invoked by the execution engine. The advantage may also be to reduce the number of lookups done by the Service Matcher (in our example, only a "solve equation" operation has to be searched for). Using Maple makes local computation very simple to setup, although we believe it should be possible to link computation engines to the execution manager or even a BPEL engine through a system of plug-ins.

4 Related work

In our approach to service composition, we distinguish two levels when looking at related work: abstract and concrete ("resolved" in our framework). Abstract composition through plans is similar to scripting. Piccola [1] is an example of a similar language in the world of web services, while the Maple language [19] is an example of a scripting language in the world of symbolic computation. The purpose is to express simple combinations of operations using an algorithmic language. In the context of MONET, this allows to consider the broker as a kind of "distributed computer algebra system" in which routines are implemented by various services. Rather than using an existing

language, we chose to create our own in order to have complete control over our experiments. The idea is to discover whether the language requires specific elements when applied to a mathematical context. Later on, we might want to switch to a more standard language (e.g. the abstract side of BPEL [4]).

While the goal of composition in the MONET framework is to allow users to express strategies to solve a given problem, we do not forget that plans should ultimately be executed and that services corresponding to each step of a plan should be invoked. That is why we need to "resolve" abstract plans (see section 3). From this point of view, our approach to composition also takes into account the ideas of choreography. Indeed, a query sent by a client to the broker may contain logistical constraints requiring, for example, transaction management, billing information, and so on. Such features can be found in choreography languages such as BPEL, WSCI/BPML [5], and so on. The mathematical context leads us to consider a feature that does not seem to be taken into account in the frameworks we studied: the need for "local" computation (see section 3). BPEL offers such local computation through the use of XPath. However, XPath proves limited and is not suitable in the context of computational mathematics. This is why we also envision the translation of our plans into the Maple scripting language and the use of Maple as a possible orchestration engine.

5 Conclusion and future work

Composition of web services appears to be essential in the context of mathematical computation. Mathematical web services only offer specific and highly specialized functions to solve atomic problems (i.e. a problem whose intermediate solving steps are not relevant to the user). The user needs to be able to compose these functions into her or his particular algorithm to solve her or his particular problem. This can be achieved using a plan language to produce plan documents. The plan language we described in this paper allows to compose web services into plan documents whose state ranges from abstract to totally resolved. Abstract plans are just like programs with parameters and can be instantiated by choosing the actual mathematical web services to perform the function calls. Resolved plans associate each function with a web service call. Execution of resolved plans can be achieved by translating a plan into a choreography document in a well-known language such as BPEL. On this topic, we also introduce the idea of using a mathematical language such as the Maple language as a target for the translation. Mathematical computation systems are indeed ideal plan execution environments. They ease the possibility of "local service invocation" with which invocation of certain

steps may be made at the location of the execution engine. This idea provides an interesting optimization that can be ported to other domains than mathematics.

To conclude this paper, we now expose a few directions we wish to explore. The completion of the language and development of appropriate tools are an obvious next step. This will allow us to pursue experiments with selected users and then at a larger scale by publishing the tools. The purpose of these experiments is to establish the exact needs of the various mathematical communities and should help to adapt the plan language. Furthermore, we believe that the idea of using computational environments for executing plans requires further investigation. In particular, we wish to conduct comparative studies about the suitability of systems like for instance Maple, Mathematica, and AXIOM [3] to the use of mathematical web services composition. This may help to determine deployment strategies for these services and the context of use for the MONET broker.

References

[1] Achermann F., Oscar Nierstrasz, *Applications = Components + Scripts - A Tour of Piccola*, Software Architectures and Component Technology, Mehmet Aksit (Ed.), pp. 261-292, Kluwer, 2001.

[2] Aldor.org, Aldor, http://www.aldor.org.

[3] Axiom Community, Axiom: The Computer Algebra System, http://www.nongnu.org/axiom

[4] BEA Systems, IBM, Microsoft, SAP AG, Siebel Systems, *Business Process Execution Language for Web Services version 1.1*, Available from http://www-106.ibm.com/developerworks/library/ws-bpel.

[5] BEA Systems, Intalio, SAP AG, Sun Microsystems, *Web Service Choreography Interface (WSCI) 1.0 Specification*, Available from http://wwws.sun.com/software/xml/developers/wsci.

[6] Bronstein M., "The BERNINA User Guide", http://www-sop.inria.fr/cafe/Manuel.Bronstein/sumit/berninadoc.

[7] Bronstein M., *SUM-IT: A strongly-typed embeddable computer algebra library*, in Proceedings of DISCO'96, Springer LNCS 1128, 22-33.

[8] Carman M., Luciano Serafini, Paolo Traverso, *Web Service Composition as Planning*, ICAPS 2003, Workshop on Planning for Web Services. Available from http://www.isi.edu/info-agents/workshops/icaps2003-p4ws/program.html

[9] Chicha Y., Marc Gaëtano, *Putting Bernina on the Web*, Mathematics on the Semantic Web Workshop, Eindhoven, 2003.

[10] DAML, *DAML services*, http://www.daml.org/services.

[11] Dewar M., David Carlisle, Olga Caprotti, *Description Schemes For Mathematical Web Services*, Electronic Workshops in Computing, Oxford, 2002.

[12] DL Implementors Group, *Description Logic Implementors Group*, http://potato.cs.man.ac.uk/dig.

[13] Eisenbud D., Daniel R. Grayson, Michael E. Stillman, Bernd Sturmfels, *Computations in algebraic geometry with Macaulay 2*, Springer-Verlag, September, 2001, n.8 in "Algorithms and Computations in Mathematics", ISBN 3-540-42230-7.

[14] Faugère J.-C., Gb, http://www-calfor.lip6.fr/ jcf/Software/Gb

[15] Greuel G.-M., G. Pfister, and H. Schönemann, "SINGULAR 2.0. A Computer Algebra System for Polynomial Computations." Centre for Computer Algebra, University of Kaiserslautern (2001). http://www.singular.uni-kl.de

[16] IBM, Microsoft, Verisign, *Specification: Web Services Security (WS-Security) Version 1.0*, Available from http://www-106.ibm.com/developerworks/library/ws-secure.

[17] IBM alphaWorks, BPWS4J, Available from http://www.alphaworks.ibm.com/tech/bpws4j.

[18] The Information Management Group, *Instance Store Database Support for Reasoning over Individuals*, http://instancestore.man.ac.uk.

[19] Maplesoft, Maple, http://www.maplesoft.com.

[20] The MONET Consortium, MONET, http://monet.nag.co.uk.

[21] The MONET Consortium, MONET documents, http://monet.nag.co.uk/cocoon/monet/publicdocs.

[22] The MONET Consortium, *Mathematical Service Description Language*, http://monet.nag.co.uk/cocoon/monet/publicdocs/monet-msdl-final.pdf.

[23] National Institute of Standards and Technology, *GAMS: Guide to Available Mathematical Software*, http://gams.nist.gov.

[24] OASIS, *Universal Description, Discovery and Integration of Web Services*, http://www.uddi.org/.

[25] The OpenMath Society, OpenMath, http://www.openmath.org.

[26] Padget J., *MONET: Mathematical service discovery and composition*, http://monet.nag.co.uk/cocoon/monet/publicdocs/slides.pdf.

[27] Verheecke, B., M.A. Cibran, *AOP for Dynamic Configuration and Management of Web services in Client-Applications*, Published in the Proceedings of 2003 International Conference on Web Services - Europe (ICWS'03-Europe), Erfurt (Germany), September 2003

[28] W3C, *W3C Math Home*, http://www.w3.org/Math.

[29] Wolfram Research, Mathematica, http://www.wolfram.com/products/mathematica.