Martlet: A Scientific Work-Flow Language for Abstracted Parallisation

Daniel Goodman
Oxford University Computing Laboratory, Parks Road, Oxford, OX1 3QD, UK
Daniel.Goodman@comlab.ox.ac.uk

27th July 2006

Abstract

This paper describes a work-flow language 'Martlet' for the analysis of large quantities of distributed data. This work-flow language is fundamentally different to other languages as it implements a new programming model. Inspired by inductive constructs of functional programming this programming model allows it to abstract the complexities of data and processing distribution. This means the user is not required to have any knowledge of the underlying architecture or how to write distributed programs.

As well as making distributed resources available to more people, this abstraction also reduces the potential for errors when writing distributed programs. While this abstraction places some restrictions on the user, it is descriptive enough to describe a large class of problems, including algorithms for solving Singular Value Decompositions and Least Squares problems. Currently this language runs on a stand-alone middleware. This middleware can however be adapted to run on top of a wide range of existing work-flow engines through the use of JIT compilers capable of producing other work-flow languages at run time. This makes this work applicable to a huge range of computing projects.

1 Introduction

The work-flow language Martlet described in this paper implements a new programming model that allows users to write parallel programs and analyse distributed data without having to be aware of the details of the parallelisation. It abstracts the parallelisation of the computation and the splitting of the data through the inclusion of constructs inspired by functional programming. These allow programs to be written as an abstract description that can be adjusted to match the data set and available resources automatically at runtime. While this programming model adds some restriction to the way programs can be written, it is possible to perform complex calculations across a distributed data set such as Singular Value Decomposition or Least Squares problems, and it creates a much more intuitive way of working with distributed systems. This allows inexperienced users to take advantage of the power of distributed computing resources, and reduces the work load on experienced distributed programmers.

While applicable to a wide range of projects, this was originally created in response to some of the problems faced in the distributed analysis of data generated by the Climate*Prediction*.net[1][9, 12] project. Climate*Prediction*.net is a distributed computing project inspired by the success of the SETI@home[2][1] project. Users download a model of the earth's climate and run it for approximately fifty model years with a range of perturbed control parameters before returning results read from their model to one of the many upload servers.

The output of these models creates a data set that is distributed across many servers in a well-defined fashion. This data set is too big to transport to a single location for analysis, so it must be worked on in a distributed manner if a user wants to analyse more than a small subset of the data. In order to derive results, it is intended that users will submit analysis functions to the servers holding the data set. As this data set provides a resource for many people, it would be unwise to allow users to submit arbitrary source code to be executed. In addition users are unable to ascertain how many servers a given subset of this data that they want to analyse spans, and nor should they care. Their interest is in the information they can derive from the data, not how it is stored. These requirements mean a trusted work-flow lan-

[1] http://www.climateprediction.net
[2] http://setiathome.ssl.berkeley.edu

guage is required as an intermediate step, allowing the construction of analysis functions from existing components, and abstracting the distribution of the data from the user.

2 Related Work

Existing work-flow languages such as BPEL[2], Pegasus [7] and Taverna [11] allow the chaining together of computational functions to provide additional functions. They have a variety of supporting tools and are compatible with a wide range of different middlewares, databases and scientific equipment. They all implement the same programming model where a known number of data inputs are mapped to computational resources and executed, taking advantage of the potential for parallelisation where possible and supporting *if* and *while* statements *etc*. As they only take a known number of inputs, none of them are able to describe a generic work-flow in which the number of inputs is unknown, which the middleware can then adapt to perform the described function at runtime once the number of inputs is known.

Independently Google have developed a programming model called Map-Reduce [6] to perform distributed calculations. This is similar to, but not as general, or as loosely coupled as Martlet. The implementing library works with the Google File System [8] to allow parallel calculations on data, while abstracting the complexity of the data storage and processing. Though similar, as it is aimed at the internal work of Google programmers working with large clusters. As such it is a set of objects that are dependant on the Google infrastructure and extended by the user. These require that the user to provide information about the environment such as the number of computers to be involved in the calculation, and functions to partition the data. All of these make it not suited to the more public heterogeneous domain that this project is aimed at.

3 Example Problem

The average temperature of a given set of returned models is an example of a situation where the level of abstraction described in this paper is required. If this data spans a servers, this calculation can be described in way that could be used for distributed computing as:

$$y_0 = \sum_{i=0}^{n_1-1} x_i$$
$$z_0 = n_1$$

$$y_1 = \sum_{i=n_1}^{n_2-1} x_i$$
$$z_1 = n_2 - n_1$$

$$\vdots$$

$$y_{a-1} = \sum_{i=n_{a-1}}^{n_a-1} x_i$$
$$z_{a-1} = n_a - n_{a-1}$$

$$\overline{x} = \frac{\sum_{i=0}^{a-1} y_i}{\sum_{i=0}^{a-1} z_i}$$

where each subset of the data set has a computation performed on it, with the results used by a final computation to produce the over all average. Each of these computations could occur on a different computing resource.

To write this in an existing work-flow language in such a way that it is properly executed in parallel, the user must first find out how many servers their required subset of data spans. Only once this value is known can the work-flow be written, and if the value of a changes the work-flow must be rewritten. The only alternative is that the user himself must write the code to deal with the segregated data. It is not a good idea to ask this of the user since it adds complexity to the system that the user does not want and may not be able to deal with, as well as adding a much greater potential for the insertion of errors into the process. In addition, work-flow languages are not usually sufficiently descriptive for a user to be able to describe what to do with an unknown number of inputs, so it is not possible just to produce a library for most languages This problem is removed with Martlet, by making such abstractions a fundamental part of the language.

4 Introducing Martlet

Our work-flow language *Martlet* supports most of the common constructs of the existing work-flow languages. In addition to these constructs, it also has constructs inspired by inductive constructs of functional programming languages [5]. These are used to implement a new programming model where functions are submitted in an abstract form and are only converted into a concrete function that can be executed when provided with concrete data structures at runtime. This hides from the user the parallel nature of the execution and the distribution of the data they wish to analyse.

We chose to design a new language rather than extending an existing one because the widely used languages are already sufficiently complex that an extension for our purposes would quickly obfuscate the features we are aiming to explore. Moreover, at the time the decision was taken, there were no suitable open-source work-flow language implementations to adapt. It is hoped that in due course the ideas developed in this language will be added into other languages.

The inspiration for this programming model came from functional programming languages where it is possible to write extremely concise powerful functions based on recursion. The reverse of a list of elements for instance can be defined in Haskell [5] as;

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

This simply states that if the list is empty, the function will return an empty list, otherwise it will take the first element from the list and turn it into a singleton list. Then it will recursively call reverse on the rest of the list and concatenate the two lists back together. The importance of this example is the explicit separation between the base case and the inductive case. Using these ideas it has been possible to construct a programming model and language that abstracts the level of parallelisation of the data away from the user, leaving the user to define algorithms in terms of a base case and an inductive case.

Along with the use of functional programming constructs, two classes of data structure, *local* and *distributed*, were created. Local data structures are stored in a single piece; distributed data structures are stored in an unknown number of pieces spanning an unknown number of machines. Distributed data structures can be considered as a list of references to local data structures. These data structures allow the functional constructs to take a set of distributed and local data structures, and functions that perform operations on local data structures. These are then used as a base case and an inductive case to construct a work-fbw where the base function gets applied to all the local data structures referenced in the distributed data structures, before the inductive function is used to reduce these partial results to a single result. So, for example, the distributed average problem looked at in Section 3, taking the distributed matrix `A` and returning the average in a column vector `B`, could be written in Martlet as the program in Figure 1.

Due to this language being developed for large scale distributed computing on huge data sets, the data is passed by reference. In addition to data, functions are also passed by reference. This means that functions are first class values that can be passed into and used in other functions, allowing the workfbws to be more generic.

5 Syntax and Semantics

To allow the global referencing of data and functions, both are referenced by URIs. The inclusion of these in scripts would make them very hard to read and would increase the potential for user errors. These problems are overcome using two techniques. First, local names for variables in the procedure are used, so the URIs for data only need to be entered when the procedure is invoked. This means that in the procedure itself all variable names are short, and can be made relevant to the data they represent. Second, a define block is included at the top of each procedure where the programmer can add abbreviations for parts of the URI. This works because the URIs have a logical pattern set by whom the function or data belongs to and the server it exists on. As a result the URIs in a given process are likely to have much in common.

The description of the process itself starts with the keyword "`proc`", then there is a list of arguments that are passed to the procedure, of which there must be at least one due to the stateless nature of processes. While additional syntax describing the read, write nature of the arguments could improve readability, it is not included as it would also prevent certain patterns of use. This may change in future variants of the language. Finally there is a list of statements in between a pair of curly braces, much like C. These statements are executed sequentially when the program is run.

There are two types of statement: normal statements and expandable statements. The difference between the two types of statements is the way they behave when the process is executed. At runtime an *expand* call is made to the data structure representing the abstract syntax tree. This call makes it adjust its shape to suit the set of concrete data references it has been passed. Normal statements only propagate the *expand* call through to any children they have, whereas expandable statements adjust the structure of the tree to match the specific data set it is required to operate on.

5.1 Normal Statements

As the language currently stands, there are six different types of normal statement. These are if-else, sequential composition, asynchronous composition, while, temporary variable creation, and process calls. Their syntax is as follows:

Sequential Composition is marked by the keyword `seq` signalling the start of a list of statements that need to be called sequentially. Although the `seq` keyword can be used at any point where a statement would be expected, in most places sequential composition is implicit. The only location that this construct really is required is when one wants to create a function in which a set of sequential lists

```
// Declare URI abbreviations in order to improve the script readability
define
{
   uri1 = baseFunction:system:http://cpdn.net:8080/Martlet;
}

proc(A,B)
{
// Declare the required local variables for the computation. Y and Z
// are used to represent the two sets of values Yi and Zi in the
// example equations. ZTotal will hold the sum of all the Zi's.
   Y = new dismatrix(A);
   Z = new disinteger(A);
   ZTotal = new integer(B);

// The base case where each Yi and Zi is calculated, and recorded in
// Y and Z respectively. The map construct results in each Zi and Yi
// being calculated independently and in parallel.
   map
   {
      matrixSum:uri1(A,Y);
      matrixCardinality:uri1(A,Z);
   }

// The inductive case, where we sum together the distributed Yi's
// and Zi's into B and ZTotal respectively.
   tree((YL,YR)\Y -> B, (ZL,ZR)\Z -> ZTotal)
   {
      matrixSumToVector:uri1(YL,YR,B);
      IntegerSum:uri1(ZL,ZR,ZTotal);
   }
// Finally we divide through B with ZTotal to finish computing the
// average of A storing the result in B.
   matrixDivide:uri1(B,ZTotal,B);
}
```

Figure 1: Function for computing the average of a matrix *A* split across an unknown number of servers. The syntax and semantics of this function is explained in Section 5.

of statements were run concurrently by an asynchronous composition. An example of this is shown in Figure 2

Asynchronous Composition is marked by the keyword `async` and encompasses a set of statements. When this is executed each statement in the set is started concurrently. The asynchronous statement only terminates when all the sub-statements have returned.

In order to prevent race conditions it is necessary that no process uses a variable concurrently with a process that writes to the variable. This is enforced by the middleware at runtime.

if-else & while are represented and behave the same as they would in any other procedural language. There is a test and then a list of statements.

Temporary Variables can be created by statements that look like

```
identifier =
        new type(identifier);
```

The identifier on the left hand side of the equality is the name of the new variable. The type on the right is the type of the variable, and the identifier on the right is a currently existing data structure used to determine the level of parallelisation required for the new variable. For example if the statement was

```
A = new DisMatrix(B);
```

this will create a distributed matrix `A` that is split into the same number of pieces as `B`. The type field is required as there is no constraint that the type of `A` is the same as the type of `B`. This freedom is required as there is no guarantee that a distributed data structure of the right type is going to appear at this stage in the procedure, as was the case in the average calculation example in Figure 1.

Process calls fall into one of two categories. They can either be statically named in the function or passed in as a reference at runtime. Both appear as an identifier and a list of arguments.

5.2 Expandable Statements

The are four expandable statements, `map`, `foldr`, `foldl` and `tree`. Each of these has a functional programming equivalent. Expandable statements don't propagate the call to expand to their children and must have been expanded before the function can be computed. This means that on any given path between the root and a leaf there must be at most one expandable statement.

```
async{
      seq{
          function1(A,B,C);
          function2(A,B);
          function3(B,C);
      }

      seq{
          function4(D,E);
          function1(D,E,F);
          function5(E,F);
      }
}
```

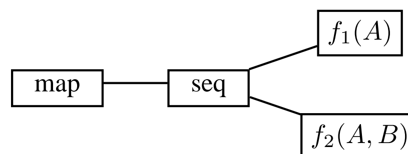
Figure 2: `seq` used to run two sequential sets of operations asynchronously.

map is equivalent to `map` in functional programming where it takes a function `f` and a list, and applies this function to every element in the list. This is shown below in Haskell:

```
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

Map in Martlet encompasses a list of statements as shown in the example below. Here function calls `f1` and `f2` are implicitly joined in a sequential composition to create the function represented by `f` in the Haskell definition. The list is created by distributed values `A` and `B`. While in its unexpanded abstract form, this example maps onto the abstract syntax tree also shown below.

```
map
{
  f1(A);
  f2(A,B);
}
```



When this is expanded, it looks at the distributed data structures it has been passed and creates a copy of these statements to run independently on each piece of the distributed data structure as shown in Figure 3.

Due to the use of an asynchronous statement in this transformation, no local value that is passed into the map statement can be written to. However local values created within the map node can be written to.

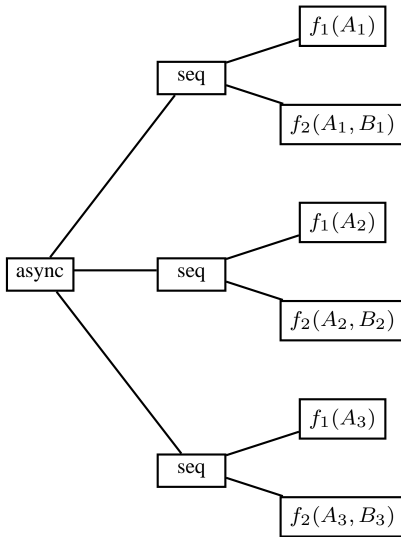


Figure 3: The abstract syntax tree for the example map statement after expand has been called setting $A = [A_1, A_2, A_3]$ and $B = [B_1, B_2, B_3]$.

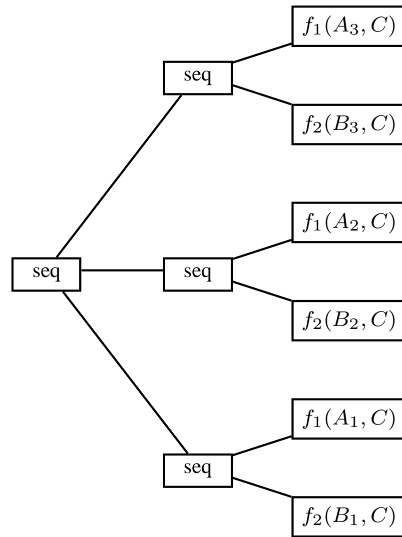


Figure 4: The abstract syntax tree for the example foldr statement after expand has been called setting $A = [A_1, A_2, A_3]$ and $B = [B_1, B_2, B_3]$.

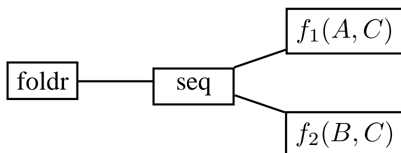
foldr is a way of applying a function and an accumulator value to each element of a list. This is defined in Haskell as:

```
foldr f e [] = e
foldr f e (x:xs) = f x
             (foldr f e xs)
```

This means that the elements of a list `xs = [1,2,3,4,5]` can be summed by the statement; `foldr (+) 0 xs` which evaluates to `1+(2+(3+(4+(5+0))))`

Foldr statements are constructed from the `foldr` keyword followed by a list of one or more statements which represent `f`. An example is shown below with its corresponding abstract syntax tree.

```
foldr
{
   f1(A,C);
   f2(B,C);
}
```



When this function is expanded this is replaced by a sequential statement that keeps any non-distributed arguments constant and calls `f` repeatedly on each piece of the distributed arguments as shown in Figure 4.

foldl is the mirror image of foldr so the Haskell example would now evaluate to `((((0+1)+2)+3)+4)+5`

The syntax tree in Martlet is expanded in almost exactly the same way as foldr. The only difference is the function calls from the sequential statement are in reverse order. The only time that there is any reason to choose between foldl and foldr is when `f` is not commutative.

tree is a more complex statement type. It constructs a binary tree with a part of the distributed data structure at each leaf, and the function `f` at each node. When executed this is able to take advantage of the potential for parallel computation. A Haskell equivalent is:

```
tree f [x] = x
tree f (x:y:ys) =
     f (tree f xs') (tree f ys')
     where (xs',ys') =
                    split (x:y:ys)
```

`split` is not defined here since the shape of the tree is not part of the specification. It will however always split the list so that neither is empty.

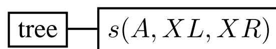
Unlike the other expandable statements, each node in a tree takes $2n$ inputs from n distributed data structures, and produce n outputs. As there is insufficient information in the structure to construct the mappings of values between nodes within the tree, the syntax requires the arguments that the statements use to be declared in brackets above the function in

such a way that the additional information is provided.

Non-distributed constants and processes used in `f` are simply denoted as a variable name. The relationship between distributed inputs and the outputs of `f` are encoded as `(ALeft,ARight)\A->B`, where `ALeft` and `ARight` are two arguments drawn from the distributed input `A` that `f` will use as input. The output will then be placed in `B` and can be used as an input from `A` at the next level in the tree.

Lets consider a function that uses a method sum passed into the statement as `s`, a distributed argument `X` as input and outputs the result to the non-distributed argument `A`. This could be written as:

```
tree((XL,XR)\X -> A)
{
   s(A,XL,XR);
}
```



When this is expanded, it uses sequential, asynchronous and temporary variables in order to construct the tree as shown in Figure 5. Because of the use of asynchronous statements any value that is written to must be passed in as either an input or an output.

5.3 Example

If the Martlet program to calculate averages from the example in Figure 1 where submitted it would produce the abstract syntax tree shown in Figure 6. This could then be expanded using the techniques show here to produce a concrete functions for different concrete datasets.

6 Conclusions

In this paper we have introduced a language and programming model that use functional constructs and two classes of data structure. Using these constructs it is able to abstract from users the complexity of creating parallel processes over distributed data and computing resources. This allows the user simply to think about the functions they want to perform and does not require them to worry about the implementation details.

Using this language, it has been possible to describe a wide range of algorithms, including algorithms for performing Singular Value Decomposition, North Atlantic Oscillation and Least Squares.

To allow the evaluation of this language and programming model, a supporting middleware has been constructed [10] using web services supported by Apache Axis [3] and Jakarta Tomcat [4]. As we have found no projects with a similar approach aimed at a similar style of environment, a direct comparison with other projects has not been possible. This work is, however, currently being tested with data from the Climate*Prediction*.net project with favorable results and will hopefully be deployed on all our servers over the course of the next year allowing testing on a huge data set.

At runtime, when concrete values have been provided, it is possible to convert abstract functions into concrete functions. The concrete functions then contain no operations that are not supported by a range of other languages. As such, it is envisaged that the middleware will in time be cut back to a layer that can sit on top of existing work-flow engines, providing extended functionality to a wide range of distributed computing applications. This capability will be provided through the construction of a set of JIT compilers for different work-flow languages. Such compliers need only take a standard XML output produced at runtime and performing a transformation to produce the language of choice. This would then allow a layer supporting the construction of distributed data structures and the submission of abstract functions to be placed on top of a wide range of existing resources with minimal effort, extending their use without affecting their existing functionality. Such a middleware would dramatically increase the number of projects that Martlet is applicable to. Hopefully the ideas in Martlet will then be absorbed into the next generation of work-flow languages. This will allow both existing and future languages to deal with a type of problem that thus far has not been addressed, but will become ever more common as we generate ever-larger data sets.

References

[1] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[2] Tony Andrews, Francisco Curbera, Hitesh Doholakia, Yaron Goland, Johannes Kiein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smitth, Satish Thatte, Ivana Trickovic, and Sanjiva Weerwarana. BPEL4WS. Technical report, BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, 2003.

[3] Apache Software Foundation. *Apache Axis*, 2005. URL: http://ws.apache.org/axis/.

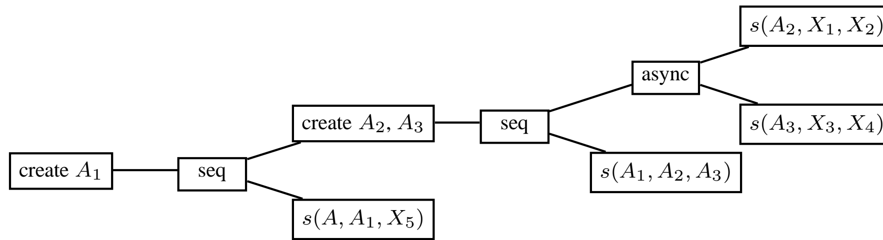


Figure 5: When the tree function on page 7 is expanded with $X = [X_1, X_2, X_3, X_4, X_5]$, this is one of the possible trees that could be generated.

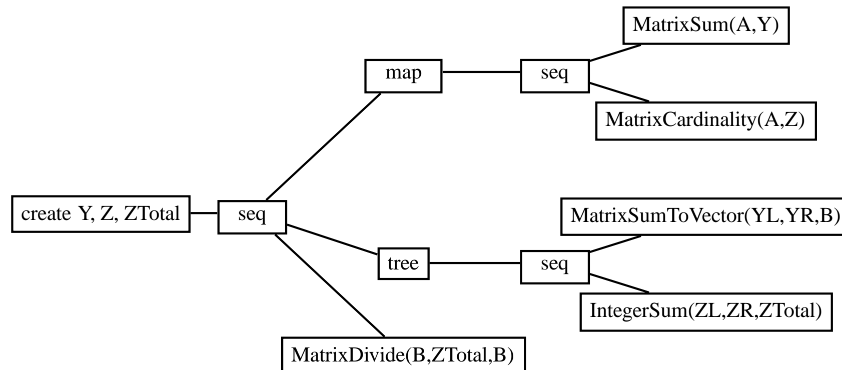


Figure 6: The abstract syntax tree representing the generic work-flow to compute the an average introduced in Figure 1.

[4] Apache Software Foundation. *The Apache Jakarta Project*, 2005. URL: http://jakarta.apache.org/tomcat/.

[5] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition, 1998.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. Technical report, Google Inc, December 2004.

[7] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Pegasus: Planning for execution in grids. Technical report, Information Sciences Institute, 2002.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[9] Daniel Goodman and Andrew Martin. Grid style web services for climate*prediction*.net. In Steven Newhouse and Savas Parastatidis, editors, *GGF workshop on building Service-Based Grids*. Global Grid Forum, 2004.

[10] Daniel Goodman and Andrew Martin. Scientific middleware for abstracted parallelisation. Technical Report RR-05-07, Oxford University Computing Lab, November 2005.

[11] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[12] David Stainforth, Jamie Kettleborough, Andrew Martin, Andrew Simpson, Richard Gillis, Ali Akkas, Richard Gault, Mat Collins, David Gavaghan, and Myles Allen. Climate*prediction*.net: Design principles for public-resource modeling research. In *14th IASTED International Conference Parallel and Distributed Computing and Systems*, Nov 2002.