and easily tailor it for use within the multicast system. After expressing an initial application design in terms of MIL specifications, the application code and specifications may be compiled and executed. The second level of support we have presented here is demonstrated by the *PTM* tool, discussed in the previous section. Using *PTM*, prototypers can concern themselves with the way information should flow through the application and not with the mechanics of the integration of the application. Together, these technologies enable programmers to spend less of their time crafting interfaces, and more of their time studying the prototyping apparatus behavior — a balance which leads to improved requirements and a higher quality software product.

References

[1] B. Bershad, D. Ching, E. Lazowska, J. Sanislo and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering*, SE-13, 8 (August 1987), 880-894.

[2] J. Bloch. The Camelot library: C Language Extensions for Programming General Purpose Distributed Transaction Systems. *Proceedings of the 9th Conference on Distributed Computing Systems*, (June 1989) 172-180.

[3] J. Callahan, J. Purtilo. A Packaging System for Heterogeneous Execution Environments. *IEEE Transactions on Software Engineering*, (June 1991), 626-635.

[4] E. Cooper. Programming Language Support for Multicast Communication in Distributed System. *IEEE Transactions on Computers*, (July 1990), 450-457.

[5] S. Feldman. Make: A Program for Maintaining Computer Programs. *UNIX Programmer's Manual*, USENIX, (1984).

[6] J. Jones, R. Rashid, and M. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, (January 1985), 225-235.

[7] J. Magee, J. Kramer and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15,6 (June 1989), 663-675.

[8] J. Purtilo. The Polylith Software Bus. To appear, *Transactions on Programming Language and Systems*.

[9] J. Purtilo, C. Hofmeister. Dynamic Reconfiguration of Distributed Programs. *Proceedings of the 11th International Conference on Distributed Computing Systems*, (1991), 560-571.

[10] C. Hofmeister, J. Atlee and J. Purtilo. Writing Distributed Programs in Polylith. *Dept of Computer Science, University of Maryland, CS-TR-2575* (December 1990).

[11] S. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Transaction on Computers*, (July 1990), 57-66.

[12] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, Vol. 2, No. 4 (December 1990).

[13] Sun Microsystems. *Remote Procedure Call Protocol Specification.* Sun Microsystems, (January 1985).

its mode is *sql*. *PTM* currently recognizes modes *in, out* and *sql*, but can easily be expanded and generate more types of stubs if more stub generating routines are written.

Given the spreadsheet defined by users, information about modules is analyzed and stored in file *application_file* by pressing button **save**. When button **run** is chosen, *PTM* generates the appropriate stubs and creates an executable for each module on its destination machine. The generated source code for modules **input1**, **db** and **output1** are shown in Figure 9. After the installation, *Polycast* uses the preprocessed specification given by *PTM* to invoke processes, coerce data representation, synchronization, and marshaling of data during communications between modules.

One feature of this tool is that it is very easy to change the structure of the application simply by changing the attributes on the spreadsheet. For example, if a user wants module **output2** to receive the messages from module **input2**, as shown in Figure 10, this can be achieved by simply changing row *output*2 column *in* from *db* to *db input*2 in the original spreadsheet specification. Further work on this tool will involve facilitating this type of structure change dynamically, or while the application is executing, by utilizing the reconfiguration techniques described in [9].

PTM was constructed by combining *Polycast*, an environment that supports multicast communication, with technology from software packaging. However, we have not fully exploited either of these ideas. Just as the Polygen packaging technology allows programmers to transparently use point-to-point communication or RPC interactions between application modules, we would like to further facilitate development of applications that use multicast communication for modules interactions. To accomplish this, we need to allow designers to use a module interconnection language to specify the events in the system and the set of events each modules will generate and will be interested in. Future work on this includes extending current packaging technology to handle this situation, allowing designers to reason about these types of applications at a high level.

6 Conclusion

In this paper we have shown how to provide various levels of support for multicast interactions between the modules of an application. The first level is provided by *Polycast*, a multicast execution environment built on top of a software interconnection system that supports applications executing in a heterogeneous distributed environment. Programmers are able to take a component's source program, written in a high-level language,

```
input1()
  /* declare message type name */
  mh_msgtype("S", "input1_msgtype");
  q = query;
  while ( q is not empty) do
     /* multicast query */;
     mh_multicast("input1_msgtype,
                  "S", q);
     q = next query;
  end while
end input1

output1
  /* register its interest in multicast
      messages */
  mh_rsgmulticast("db_msgypte");
  /* declare standard null message */
  mh_nomsg("NO MESSAGE.");
  while (true) do
     /* get a multicast message into r */;
     mh_getmsg("S", r, msgtypebuf);
     write r to output;
  end while
end output1


db()
  /* declare standard null message */
  mh_nomsg("NO MESSAGE.");
  /* declare message type name */
  mh_msgtype("S", name);
  /* register its interest in multicast
     messages */
  mh_rgsmulticast("input1_msgtype");
  mh_rgsmulticast("input2_msgtype");
  while (true) do
    /* get a multicast message in q*/
     mh_getmsg("S", q, msgtypebuf);
     if q is not "NO MESSAGE."
       r = result of executing query q;
       /* multicast the result of the query */;
       mh_multicast(name, "S", r);
     end if
  end while
end db
```

Figure 9: modules of mode **in**, **out** and **sql**

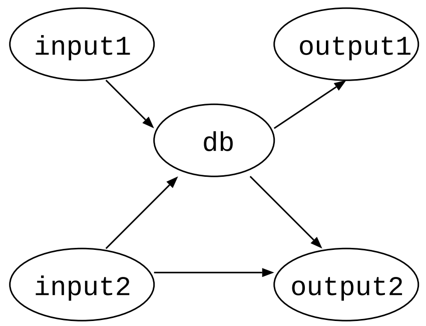

Figure 10: the application structure for a new example

ident	name	binary	machine	in	mode
1	input1	/jteam/input1	calloo	-	out
2	input2	/jteam/input2	calvin	-	out
3	db	/jteam/db	callay	input1 input2	sql
4	outut1	/jteam/output1	home	db	in
5	output2	/jteam/output2	grandwazoo	db	in

save quit run

Figure 8: a snapshot of the spreadsheet

interconnection system.

5 PTM and Example Revisited

When building a software prototype, developers need to experiment with the system repeatedly in order to make decisions to reduce risk of failure in a software product. So we need a software environment that allows experimental activities to be carried out inexpensively. *PTM* is a multicast environment designed to help developers to build prototypes for software components to communicate with each other in heterogeneous distributed systems. It is made possible by combining results from both multicast and software packaging. It helps developers to generate multicast software executables automatically given abstract descriptions of the components and of the desired application geometry. In such a way, developers can experiment with their prototype quickly and easily without the overhead of having to prepare the software manually, and it is flexible enough to allow applications to be reconfigured dynamically.

PTM provides a spreadsheet-like user interface for developers that allows them to define the structure of the applications. *PTM* uses these specifications to decide what interfaces a module must have, to select interconnection options between interfaces, and to build the interface software and integrate the modules. Currently, *PTM* packages applications to run in the *Polycast* multicasting environment described in Section 4.1. This tool allows designers to prototype applications where:

- An *event* is the output of some module. For example, in the database application, the events are the output of information from module **input1**, the output of information from module **input2**, and the output of information from module **db**.
- More than one module may be interested in the output of a given module.
- Each module may receive information from more than one module.
- Modules are restricted to the set of types for which there are implemented stub generators.

Figure 8 shows what the spreadsheet for the database application described in Section 2 would look like. For each application module, the name of the module, the location of the binary, the machine where the module should execute, the list of modules that the current module is interested in and the mode of the module are specified. Users do not need to edit source files and or a Makefile to build the application; all they need to do is to provide attributes associated with each module.

Looking back at Figure 1, we see that module **db** takes input from **input1** and **input2** and sends its results to **output1** and **output2**. Using the spreadsheet interface, this interaction is described by the column *in*. The *mode* of a module indicates what stub generator will be used to construct the module. For example, in the first row of the spreadsheet, we have module **input1** with its executable */jteam/input1* on machine *calloo*. Since **input1** does not take input from other modules but sends output to other modules, its mode is *out*. On the other hand, **output1** takes input from some module but does not send output to other modules, its mode is *in*. Module **db** queries the database,

```
input1()
  /* declare message type name */
  mh_msgtype("S", "input1_msgtype");
  q = query;
  while ( q is not empty) do
     /* multicast query */;
     mh_multicast("input1_msgtype,
                    "S", q);
     q = next query;
  end while
end input1

output1
  /* register its interest in multicast
      messages */
  mh_rsgmulticast("db_msgypte");
  /* declare standard null message */
  mh_nomsg("NO MESSAGE.");
  while (true) do
     /* get a multicast message into r */;
     mh_getmsg("S", r, msgtypebuf);
  end while
  write r to output;
end output1

db()
  /* declare standard null message */
  mh_nomsg("NO MESSAGE.");
  /* declare message type name */
  mh_msgtype("S", "db_msgtype");
  /* register its interest in multicast
     messages */
  mh_rgsmulticast("input1_msgtype");
  mh_rgsmulticast("input2_msgtype");
  while (true) do
    /* get a multicast message in q*/
     mh_getmsg("S", q, msgtypebuf);
     if q is not "NO MESSAGE."
       r = result of executing query q;
       /* multicast the result of the query */;
       mh_multicast("db_msgtype", "S", r);
     end if
  end while
end db
```

Figure 7: Modules for the database example using multicast

ticast communications between modules in *Polycast*, programmers take some ordinary software modules, insert multicast primitives into the modules, write specifications for the application in terms of MIL, and then compile it. It is easy to build multicast applications using *Polycast*; the extra cost of building applications is to insert multicast primitives and write the specification by programmers.

4.2 Software Packaging

One of the goals of this work was to provide multicasting facilities at a minimum cost to programmers. The approach we take is to use programmer-supplied module and application specifications to direct the automatic packaging of the application. This packaging includes the automatic generation of interface software (or *stubs*) and the automatic enumeration of commands required to transform (compile and link, for example) the components of the application into executable objects.

The way application modules are adapted for integration depends in part on the languages in which the modules are implemented, the hardware on which the modules are to execute, and the interprocess communication (IPC facilities) available in the environment. Given this kind of information plus a specification of the modules' interfaces, stubs can be generated customized to the environment and to some type of module interaction, such as remote procedure call (RPC). Once compiled with or otherwise included in the implementation of a module, a stub acts as an intermediary between the module and the rest of the application. RPC stub generation tools such as Matchmaker [6] and SunRPC [13] have existed for many years.

However, programmers need assistance with more than just the generation of stubs themselves. Each desired configuration requires different communication mechanisms and integration strategies. Once stubs are generated and the commands needed to integrate the new files have been enumerated, tools such as MAKEFILES [5] can help programmer obtain executables reliably — the problem is identifying the program units and generating the appropriate command. This is a tedious task that no programmer is interested in performing manually.

The packaging work described in this paper is based on the software packaging technology of Polygen[3]. In Polygen, a *primitive module specification* is an abstract description of a software module describing the interfaces of a module and other attributes (the location of the source code for example) of that module. The initial configuration of an application is given in an *composite specification* that lists the modules of the application and describes the bindings between the module interfaces. Polygen uses these specifications analyze constraints affecting compatibility, select interconnection options between interfaces and enumerate the configuration commands needed to build the interface software and integrate the modules. The generated configuration commands include calls to Polygen's stub generators to produce interface software customized for the modules and the environment. In this manner, modules can be composed and reused in different applications without being modified explicitly by the software developer. Currently, Polygen packages applications that use point-to-point communication channels or RPC interactions between modules for the Polylith software

environments. The software toolbus already manages data marshaling; encoding data into a stream. When an encoded stream is transmitted to another module, Polylith decodes it into the corresponding data structure. In addition to the data marshaling, the software toolbus coerces the low level representation of primitive data types on different underlying architecture.

Now we present details concerning *Polycast*, the environment we have constructed for experimenting with multicast in heterogeneous distributed systems. To build *Polycast*, we have added a set of primitives to Polylith to support multicast communications between modules:

- $mh\_msgtype(data\_type, msgtype\_name)$

  A module uses this service to declare a message type $msgtype\_name$ of data type $data\_type$. For example, $mh\_msgtype(\text{“}S\text{”}, \text{“}input1\_msgtype\text{”})$ declares a multicast message type “$input1\_msgtype$” of data type string.

- $mh\_rgsmulticast(msgtype\_name)$

  A module uses this service to register its interest in message type $msgtype\_name$.

- $mh\_multicast(msgtype\_name, data\_type, msg)$

  A module uses this service to multicast a message.

- $mh\_signal\_multicast(msgtype\_name, data\_type, message)$

  A module uses this service to multicast a message and send signals to all the modules interested in this type of message.

- $mh\_bgetmsg(data\_type, msgtype\_name, message)$

  A module uses this service to receive a multicast message. This operation is blocking, returning a message from the message queue if one is available, otherwise, it waits until one is available.

- $mh\_getmsg(data\_type, msgtype\_name, message)$

  A module uses this service to receive a multicast message. This operation is non-blocking, returning a message from the message queue if one is available, otherwise, a standard null message will be returned.

- $mh\_nomsg(message)$

  A module uses this service to declare the standard null message that will be returned when there are no waiting multicast message.

- $mh\_query\_msgtype(msgtype\_buffer)$

  A module uses this service to list all message types in which it has registered an interest.

```
module input1{
   binary = "/jteam/input1.exe"::
   machine = "calloo"          ::
}
module input2{
   binary = "/jteam/input2.exe"::
   machine = "calvin"          ::
}
module db{
   binary = "/jteam/db.exe"    ::
   machine = "callay"          ::
}
module output1{
   binary = "/jteam/output1.exe"::
   machine = "home"             ::
}
module output2{
   binary = "/jteam/output2.exe"::
   machine = "grandwazoo"       ::
}
module example{
  instance input1
  instance input2
  instance db
  instance output1
  instance output2
}
```

Figure 6: MIL for the database example using multicast

Using *Polycast*, programmers can take ordinary software modules and add multicast primitives into the modules to execute in multicast-based environment. Figure 6 and Figure 7 show what the MIL and the pseudo code looks like for the database example in Section 2 using our *Polycast* multicast primitives when done manually. For example, **db** declares a standard null message to be “NO MESSAGE” and then declares a message type $db\_msgtype$. By using $mh\_rgsmulticast$, it registers its interest in message type $input1\_msgtype$ and $input2\_msgtype$. Then **db** calls mh_getmsg to receive a message. If the message queue for the multicast interface of module **db** is not empty, then a message is dequeued and sent back to **db**, otherwise it will get “NO MESSAGE”. When a message arrives, module **db** consults the database and multicasts the result of the query as message type $db\_msgtype$. Since module **output1** and **output2** are interested in $db\_msgtype$ type of message, a copy of the message is enqueued on interfaces of both **output1** and **output2**.

Cooper discusses essential and desirable properties of a programming language support for multicast communication[4]; Reiss's Field environment does not allow tools across a heterogeneous network [11], while *Polycast* is our implementation of an abstract multicast formulation for software to execute in heterogeneous distributed environments. In order to allow mul-

with Polylith *read* and *write* operations is shown in Figure 4. At runtime, Polylith starts the five modules and handles message passing. Figure 5 shows what the application would look like at runtime.

Unfortunately, this approach to application integration has several problems. Because the modules are connected via point-to-point communication channels, each module must know exactly how many other modules are interested in the messages it sends, making it difficult for the application to evolve and grow. Also, in order to send the message to the N modules interested in its message, the module must make N Polylith *write* calls. This duplicate work increases the cost of the run time performance. These problems would be solved with a facility to support multicast message passing in distributed systems application integration.

Another problem with this approach is that the programmer is responsible for defining the structure of the application. In a multicast framework, this means deciding what message types are to be sent by each module and what message types are of interest to each module. Once this is done, the module code must be augmented manually with Polylith read and write calls on interfaces and Polylith MIL specifications must be written that define the module interfaces and bindings. It is the programmer's responsibility to ensure that these capture the intended structure of the application. We believe that programmers might find it useful to be able to reason about an application's configuration in terms on a module interconnection language for multicast interconnection.

In the rest of this paper, we focus on answering a number of questions. Can we provide multicasting in the presence of system architecture heterogeneity? Moreover, how can we provide this capability at the minimum cost to programmers? Can the preparation for execution in a multicast environment be automated? Our answer to these questions are embodied in a new research tool called *PTM* that helps programmers generate software executables tailored to a multicasting environment.

4 Contributing Technologies

The work described in this paper is based on two different technologies: multicast communications and software packaging. Section 4.1 discusses our requirements for multicast communications in distributed systems and presents *Polycast*, our implementation of a multicast environment in terms of a software bus organization. Section 4.2 introduces the software packaging technology that we use to assist the development of distributed software applications using multicast.

4.1 Polycast: Multicast in the Polylith Environment

In the previous sections, we talked about the problems we have in a traditional point-to-point communication system. Multicast is a mode of module interaction where messages or events produced by one module can be sent to multiple modules at the same time. In order to let modules interact in a multicasting environment, application modules need to specify what types of messages they are interested in receiving. When a message is multicast from some module, all modules that are interested in this type of message can receive it. Modules that want to multicast messages do not have to know which modules are interested in this type of message. Modules that want to get multicast messages do not have to know where the messages come from.

Our objective is to provide a framework for multicast in distributed applications. An environment to support multicast must meet the following requirements:

- Users need an easy way to configure and invoke an application.
- Multicast communication between modules should function normally in the presence of system architecture heterogeneity.
- The mechanism must not compromise the data type system of the programming language. The parameters must be marshaled correctly. The low level representations of primitive data types on diverse underlying architectures should match.
- A module should be able to declare any type of message.
- A module should be able to register its interest in any type of message.
- A module should be able to multicast any message.
- A module should be able to get a multicast message it is interested in it.
- Multicast should be provided at minimum cost to programmers and without loss of run-time performance.

Our approach to meeting the above requirements is to build a multicast execution environment *Polycast* upon the existing Polylith interconnection system mentioned in Section 2. We made this decision for several reasons. Polylith already provides users with an environment that facilitates construction of applications. Using Polylith, the modules interface directly with a software toolbus for execution in heterogeneous

```
module input1{
  binary = "/jteam/input1.exe"::
  machine = "calloo"          ::
  source "send_input1_db"
      pattern = {string}      ::
}
module input2{
  binary = "/jteam/input2.exe"::
  machine = "calvin"          ::
  source "send_input2_db"
      pattern = {string}      ::
}
module db{
  binary = "/jteam/db.exe"    ::
  machine = "callay"          ::
  source "send_db_output1"    ::
      pattern  =  {string}
  source "send_db_output2"
      pattern  =  {string}    ::
  sink "receive_input_db"
      pattern  =  {string}    ::
}
module output1{
  binary = "/jteam/output1.exe"::
  machine = "home"             ::
  sink "receive_db_output1"
      pattern =  {string}      ::
}
module output2{
  binary = "/jteam/output2.exe"::
  machine = "grandwazoo"       ::
  sink "receive_db_output2"
       pattern = {string}      ::
}

module example{
  instance input1
  instance input2
  instance db
  instance output1
  instance output2
  bind "input1 send_input1_db"
            "db receive_input_db"
  bind "input2 send_input2_db"
            "db receive_input_db"
  bind "db send_db_output1"
            "output1 receive_db_output1"
  bind "db send_db_output2"
            "output2 receive_db_output2"
}
```

Figure 3: MIL program for the database example

to establish communication channels. For example, *bind* "*input*1 *send_input*1*_db*" "*db receive_input_db*" establishes a communication channel from module **input1** to module **db**. When **input1** sends messages using interface *send_input*1*_db*, **db** can receive these messages from interface *receive_input_db*.

In order to use Polylith to allow communication between the modules of an application, the user manually inserts into the modules' code Polylith *read* (*mh_read*) and *write* (*mh_write*) calls using the interfaces defined in the MIL. The pseudo code for the database example

```
input1()
  q = query;
  while ( q is not empty) do
    /* send q on the "send_input1_db"
       interface */;
    mh_write("send_input1_db",
              "S", NULL, NULL, q);
    q = next query;
  end while
end input1

output1()
  while (true) do
    /* r is a result message from the
     "receive_db_output1" interface */;
    mh_read("receive_db_output1",
             "S", NULL, NULL, r);
    write r to output;
  end while
end output1

db()
  while (true) do
    /* q is a query message from the
       "receive_input_db" interface; */
    mh_read("receive_input_db", "S",
                NULL, NULL, q);
    r = result of executing query q;
    /* send r on the "send_db_output1"
       interface */;
    mh_write("send_db_output1", "S",
                NULL, NULL, r);
    /* send r on the "send_db_output2"
       interface */;
    mh_write("send_db_output2", "S",
                NULL, NULL, r);
  end while
end db
```

Figure 4: Modules for the example with Polylith *read* and *write* operations

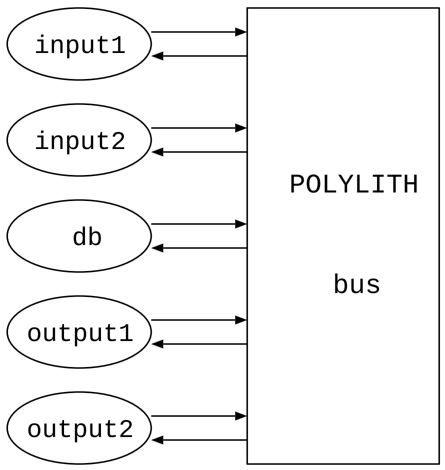

Figure 5: application structure using the Polylith interconnection system

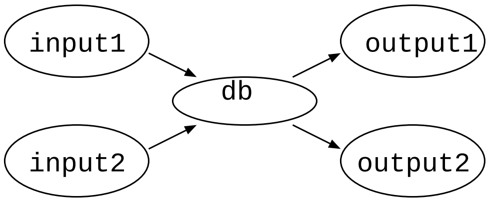

Figure 1: application structure

```
input1()
  q = query;
  while ( q is not empty) do
    send q on "send_input1_db" interface;
    q = next query;
  end while
end input1

output1()
  while ( true ) do
    r is a result message from
    "receive_db_output1" interface;
    write r to output;
  end while
end output1

db()
  while ( true ) do
    q is a query message from the
    "receive_input_db" interface;
    r = result of executing query q;
    send r on "send_db_output1" interface;
    send r on "send_db_output2" interface;
  end while
end db
```

Figure 2: Modules for the database problem

multicast environment and proposes *Polycast*, a set of primitives and underlying system that fulfills these requirements. Packaging technology is discussed in Section 3.2. Finally, in Section 4 we explain how multicast facilities and packaging technology are combined to produce *PTM*.

3 Motivating Problem

A distributed software application is a group of interoperating components distributed across a network of possibly heterogeneous processors. Each component is implemented by a module, i.e. a collection of individual data and program units, and has interfaces that are used to send and receive messages. These module interfaces may be bound to one another, providing communication channels between the modules.

Figure 1 shows an application consisting of five modules **input1**, **input2**, **db**, **output1** and **output2**, that are distributed across different host machines in order to balance the load of the system. A relational database is resident on the same machine where **db** will execute. Modules **input1** and **input2** are simply input windows where a user of the application can type in sql queries. Module **db** will receive the queries sent from **input1** and **input2** and will send the results of the queries to **output1** and **output2**, the modules responsible for displaying the results of the queries. Pseudo code for module **input1**, **output1** and **db** in this database application is shown in Figure 2, while **input2** and **output2** are similar to **input1** and **output1** respectively.

Developing a distributed software application such as this one can be a difficult task. One approach to building or *integrating* distributed software applications is to manually transform the modules so that the code (also called interface software) that uses the underlying environment to send and receive messages is part of the implementation. Unfortunately, interface software is difficult and time consuming to create since it requires an understanding of how the modules of an application are to interact and requires knowledge of the target machine's architecture and communication mechanisms.

Another approach to developing a distributed software application is to use a system, such as Polylith [8], PVM [12], and Camelot [2], that provides a standard interface for message passing between modules. We will first describe how to implement this application in terms of the software interconnection system Polylith. Polylith allows programmers to configure applications from mixed-language software components (modules), and then execute those applications in diverse environments. Users provide abstract descriptions of the application modules, including information about the modules' interfaces, a list of the modules that make up the application and the bindings between these modules' interfaces. This information is processed and at runtime, Polylith's software tool bus is responsible for invoking processes, and for coercing data representation, synchronization, and marshaling of data during communication.

A Polylith module interconnection language (MIL) specification program for the database application is shown in Figure 3. In the MIL program, we specify information for each module: the location of the executable code, the name of host machine where it should run, and information about its interfaces such as the data type of the messages or whether it it is a *source* interface (outgoing), a *sink* interface (incoming) or bidirectional. For the application as a whole we must specify what modules make up the application (using *tool*) and we also have to specify bindings between the interfaces

PTM, which helps programmers view and manipulate the connectivity of their program structures within this run time environment.

2 Background

Writing distributed programs can be difficult for programmers, who must deal with network interfacing considerations at the same time they write their application. Moreover, once the programs are completed, they are typically difficult to alter for reuse in other applications. To combat this problem, programmers use systems and tools that allow them to encapsulate interfacing decisions separately from their application code. This makes the network transparent and gives programmers the illusion of simplicity in coding, as the support tools prepare interfacing code automatically while generating executables. Many systems now provide this functionality, such as HRPC [1] and Conic [7].

This approach has proven effective for applications whose interfaces are established by programmers using explicit, point-to-point communication semantics. But recently the community has shown increasing interest in multicast communication mechanisms, where output from a tool interface may be delivered to a set of other processes that are not only unknown to the sender, but a set that is dynamically changing. Up to now, this situation has defeated automatic packaging tools that bind interfaces using static conventions. The network is once again visible to programmers, who must manually introduce interface code to their applications. This reverses the trend towards software that is less costly to build and more easily reused.

The problem for programmers is in managing the events that characterize communication in multicast. Hiding a raw network from programmers is easy enough, and most multicast systems that we are aware of would support introduction of simple interface stubs. The difficulty is how to fill in those stubs with enough information so that an initial configuration involving those components can be established in the network reliably and repeatably. 'Bindings' in a multicast environment are manifest very differently than in a traditional point-to-point system — in a sense, tools produce certain kinds of messages or events, and similarly consume certain others. Each tool is either statically prepared to produce and consume these events, or the user of the program enables the mapping of these events dynamically. In previous systems, if programmers wished to have a particular set of programs configured as a system, then they have the following obligations:

1. They need to determine the kinds of events that constitute discourse between the processes.
2. They must tailor each tool to produce or consume such events (or know the kinds of events traded by tools already packaged by others).
3. At execution time, each tool must be initiated manually.

By itself, this multicast configuration would have little advantage over point-to-point communication semantics, so it is only in observing how easily multicast systems can evolve without needing to alter the tools that we see multicast excel.

Our hypothesis in this work was that programmers might find it more useful on occasion to reason about an application's initial configuration in terms of a module interconnection language, much like how they operate on networks providing point-to-point communication semantics already [10]. If this structural description of the system could be used to automatically prepare executables suitable for multicast environments, then we could have the flexibility and evolvability of the multicast paradigm, along with the software engineering benefits afforded by modern packager technology. In short, programmers would not need to think up 'kinds' of events for tools to produce and consume in order for them to be 'bound' via an interface.

We are now able to demonstrate these benefits via our experimental system called *PTM*. Our approach has been to formulate a multicast execution environment in terms of software bus organization [8], and then take advantage of existing packager technology [3] (based upon the software bus) to show how large software configurations can be automatically prepared for execution on the new bus — without requiring manual adaptation of the code. The multicast bus implementation by itself gives no more functionality than other (and perhaps more mature) systems that came first, but the structural requirements needed to be characterized as a 'bus' mean that a packager will have enough information to prepare interfacing software in order to achieve a valid implementation of the programmer's configuration. The programmer can manipulate ordinary software as if it is to execute using traditional RPC or message passing semantics; he can automatically prepare it for execution in the multicast-based environments; and, once initiated, he can operate on the software transparently and flexibility, just as with any multicast system.

In the next section, we give some background on various approaches to application integration and describe a distributed software application. We then use this application to demonstrate the need for tools that allow multicast facilities to be easily used by designers. Section 3 gives background on the two ideas that we have combined. Section 3.1 lists our requirements for a

A PACKAGER FOR MULTICAST SOFTWARE IN DISTRIBUTED SYSTEMS

Chen Chen, Elizabeth L. White and James M. Purtilo
Computer Science Department
University of Maryland at College Park

Abstract

Prototyping is an important part of modern software development projects, and the ease with which prototyping apparatus can be assembled for experimentation is a major factor in the effectiveness of a prototyping operation. If interconnecting existing components for experimentation is too expensive, then the benefits promised from prototyping will remain unrealized. Multicast is one run-time technology which delivers on the promise of low-cost interconnection — but only once the components are prepared to execute within a multicast system. Moreover, finding ways for prototypers to view their apparatus (from what, in multicast, can be a very tangled logical structure) can be difficult. In support of a software prototyping initiative, we have produced a new packaging technology which automates the task of adapting software components for execution within multicast execution environments. This paper describes our new packager, along with our experiences in use of the system to date. In addition, we describe our visualization tool which, in the run-time environment, allows prototypers to view and operate upon their application's structure at run-time, as enabled by the multicast control paradigm.

With oversight by Office of Naval Research, this research is supported by DARPA/ISTO in conjunction with the Common Prototyping Language project, contract number N00014-90-C-0015.

1 Introduction

Prototyping — an experimental activity intended to expose properties or design alternatives to developers before they make critical decisions — is an increasingly important step within modern software development processes. Among the decisions which a developer might wish to address by prototyping are those having to do with the configuration of a large application out of reusable components. The desired choice of components, their connectivity and their modularization may in some cases only be determined by experiments in a prototyping environment. In order to facilitate rapid configuration of the prototyping apparatus, some developers are turning to a multicast system for integrating the components. Multicast (also known as selective broadcast) is a communication mechanism where the messages sent by some component may be received by a set of components. The components in this set are not only unknown to the sender, but the set may be dynamically changing during execution. Since the control paradigm for multicast is based upon events rather than named interfaces, developers find reduced coupling between their components, and are hence free to vary the structural design easily (even dynamically, in some cases). This greatly facilitates rapid experimentation with prototyping alternatives, which in turn leads to higher quality software products.

A multicast approach alone, however, does not reduce cost of experimentation sufficiently. Our experience with this interconnection mechanism indicates that the cost of tailoring software components in order to operate within a multicast run-time environment can outweigh the cost of simply building the application program statically for each experiment; when this is true, then developers loose some of the incentive to choose prototyping.

Our research shows how emerging *packaging technology* can be applied to the task of automatically preparing software components for use in multicast execution environments. 'Packaging' refers to the task of reasoning about compatibility of software components in order to determine valid means for integrating and interconnecting them. It is made possible by the abstract software bus organization developed here previously, and in turn facilitates our research in the area of prototyping technology. In the work reported here, we have build a packager called *Polycast* which lets us name and package ordinary program units for use in our own experimental multicast system. We describe the multicast environment (built out of our Polylith software bus system), and also a high level prototyping tool called