Configuration-Level Programming of Distributed Applications Using Implicit Invocation

Chen Chen

Computer Science Department
University of Maryland
College Park, MD 20742

Abstract

An event-based distributed application is a group of software components interacting with each other by producing events that in turn trigger the invocation of procedures. In this work, we are concerned with the technology and methods for integrating an event-based application, whether that application is being constructed from scratch or synthesized from existing systems. Developing an event-based application is a complex task for programmers, who must address several issues not found in traditional systems and, currently, must do so without much assistance. These issues include event declaration, structure, binding, and naming. Our objective is to provide the same software engineering benefits to programmers of event-based applications as are currently provided to programmers of applications using traditional RPC or message-passing mechanisms.

In this work, we broaden the technology for integration to encompass event-based programming. A method is described for separating event interaction properties from the implementation of the application modules so that system integration can be performed using only the abstractions. Then based upon the abstract aggregate, all interface software needed to validly implement the system can be generated automatically. A software bus model has been enhanced to accommodate the models which drive event-based distributed applications. In this way, designers may define complex event-based interactions abstractly, thus making it easier to integrate and experiment with event-based distributed applications.

This paper is the author's PhD dissertation, as supervised by James M. Purtilo.

Chapter 1

Introduction

An event-based distributed application is composed of a group of components interacting to accomplish some logical end. These components interact with each other by producing events that in turn trigger the invocation of procedures. These application modules could be written in different programming languages and distributed across a network of possibly heterogeneous processors. In this work, we are interested in coarse-grained, or configuration-level interaction and integration, and we focus on technology and methods for integrating an event-based application, whether that application is being constructed from scratch or is a synthesis of existing event-based systems. Our objective is to provide the same software engineering benefits to programmers of event-based applications as are currently provided to programmers of applications using traditional RPC or message passing mechanisms.

Distributed applications have been traditionally constructed out of modules that interact with each other by explicitly invoking procedures on named interfaces. Recently, however, software developers have shown increasing interest in an alternative integration technique, variously referred to as implicit invocation [16] [18], multicast [9], or selective broadcast [38] [19]. The idea behind this integration technique is that instead of invoking a procedure directly on a named interface, events multicast by modules trigger the invocation of procedures. Multicast [1] is a mode of communication where an event produced by some module can be sent to a set of modules at the same time. In order for modules to interact in an event-based application, modules must specify what events they are interested in receiving. A module registers its interest in an event by associating a procedure with that event. When an event is multicast, all the modules that are interested in the event receive it and all procedures that have been associated with the event are invoked by the system.

There are many advantages to this integration approach. First, it provides strong support for software reuse. Since modules are not required to explicitly name other modules, it is possible to integrate modules by registering their interest in events of the system. In addition, the integration

[1] Multicast refers to one-to-many communication, while broadcast refers to one-to-all communication. In this sense, broadcast is a special case of multicast.

approach eases system evolution. A module may be replaced by another without affecting the interfaces of modules that implicitly depend on it. In contrast, in a system based on explicit invocation, changes made to interfaces of one module will affect other modules that interact with it on those interfaces. Since the control paradigm for event-based application is based upon events rather than named interfaces, developers find reduced coupling between their modules, and are hence free to vary the structural design easily.

Developing an event-based distributed application of any kind can be difficult for programmers, who must deal with several issues not found in traditional systems and, currently, must do so without much assistance. These issues include event declaration, structure, binding, and also complexity in naming events used in communication. Several languages have been developed to enable programmers to build a single event-based program, intended to execute on one or a very small number of hosts [11] [18] [13] [27] [33] [39]. But a language-based approach does not scale up to systems of event-based components; where interaction among components is complex, application modules could be written in different programming languages and distributed across a network of possibly heterogeneous processors.

When the complexity of building a large program became too great, our research field quickly shifted to building systems of smaller components instead. This was enabled by the use of module interconnection languages and by technology for integrating those components [3] [5] [12] [28] [35] [41]. In particular, software packaging [4] — that is, reasoning about compatibility of software modules in order to determine valid means for integrating and interconnecting them — has been successful in simplifying system design and implementation, with all the benefits widely known about modular programs. To date, however, this technology has not encompassed applications that are event-based; programmers must resort to manual techniques for developing and integrating components, mixing code having to do with communication with code having to do with the application's functionality. This reverses the trend towards software that is less costly to build and more easily reused.

In the remainder of this chapter, we introduce two examples to illustrate some of the problems driving this research and discuss possible approaches to integrating event-based distributed applications. We then describe our integration approach and the support provided by our work. We discuss other related research efforts. Finally, we give an outline of the thesis.

1.1 Motivating Problems

To illustrate the problems of integrating event-based applications, we will first introduce two examples, a 2D and a 3D virtual environment applications. We will then discuss different approaches to building the applications from scratch and to supporting interactions between them.

Figure 1.1 shows an application that simulates a 3D virtual environment (VE3D) where a user can walk through a building. This application consists of four modules, **Controller**, **Display**, **Simulator** and **Log**, which are distributed on different host machines. Module **Controller** is used to enter the current user's position. Module **Simulator** simulates the position of users of other walkthroughs. Information on the current user and the users of other walkthroughs is fed to module **Display**, which displays users of other walkthroughs as objects in the current world. Module **Log** records both the current user's position and other users' positions into a log file.

The execution of VE3D can be modeled as an event-based application where modules interact with each other by multicasting events that in turn trigger the invocation of procedures. Figure 1.2 shows the pseudo code of VE3D. Module **Controller** multicasts event *obj* describing the current user's position. Module **Simulator** multicasts event *ext_obj* describing other users' positions. Modules **Display** and **Log** register their interest in events *obj* and *ext_obj*. Figure 1.3 shows modules' interest in events in VE3D and procedures associated with them. When module **Display** receives event *obj*, it invokes procedure *update_viewpoint*, and event *ext_obj* triggers invocation of procedure *update_ext_obj*. Similarly, events *obj* and *ext_obj* received by module **Log** trigger the invocation of procedures *log_obj* and *log_ext_obj*, respectively.

Figure 1.4 shows an application that simulates a 2D virtual environment (VE2D) where a user can walk through a building. This application consists of six modules, **Controller**, **Display_XY**, **Display_YZ**, **Display_XZ**, **Simulator** and **Log**, which are distributed on different host machines. Like VE3D, module **Controller** is used to enter the current user's position. Module **Simulator** simulates the position of users of other walkthroughs. Module **Log** records both the current user's position and other users' positions in a log file. Information on positions of the current user and the users of other walkthroughs is fed to display modules, which display users of other walkthroughs as objects in the current world. Unlike VE3D, modules in VE2D display objects in the *2D* world instead of the 3D world. **Display_XY**, **Display_YZ**, and **Display_XZ** display objects in the XY plane, YZ plane, and XZ plane, respectively.

Similarly, the execution of VE2D can be modeled as an event-based application. Figure 1.5 shows the pseudo code of VE2D. Module **Controller** multicasts events *obj*, *obj_xy*, *obj_yz*, and *obj_xz* describing the current user's position. Module **Simulator** multicasts events *ext_obj*, *ext_obj_xy*, *ext_obj_yz*, and *ext_obj_xz* describing other users' positions. Figure 1.6 shows modules' interest in events in VE2D and procedures associated with them. For example, module **Display_XY** registers its interest in events *obj_xy* and *ext_obj_xy*. When event *obj_xy* is received, it triggers invocation of procedure *update_viewpoint_xy*, and event *ext_obj_xy* triggers invocation of procedure *update_ext_obj_xy*.

First, let us consider building an event-based application from scratch. One approach to building or integrating event-based distributed software applications is to transform the modules manually, mixing the code having to do with event-based communication with code having to do with the application's functionality. While the multicast primitives must still be explained, Figure 1.7 shows what the source code for modules **Controller** and **Display** in VE3D would look like using POLYLITH. For example, module **Controller** of VE3D application would include the code to declare event *obj* and define the event structure as three integers before it multicasts the event. Module **Display** of VE3D would include the code to define procedures *update_viewpoint* and *update_ext_obj*, would register its interest in events *obj* and *ext_obj*, would get the next event available for processing, would decode the events according to the event structure, and would

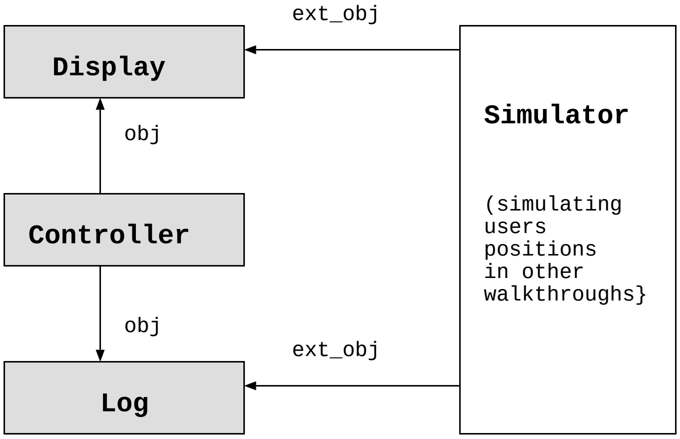

Figure 1.1: Application structure for VE3D

```
Display()
begin
  while ( true) {
     e = next event;
     if event_name is obj
       update_viewpoint;
     else if event_name is ext_obj
       update_ext_obj;
  }
end Display
```

```
Log()
begin
  while ( true) {
      e = next event;
      if event_name is obj
          log_obj;
      else if event_name is ext_obj
          log_ext_obj;
  }
end Log
```

```
Controller()
begin
  while ( true ) {
    read current user's position
    from input;
    multicast event obj;
  }
end Controller
```

```
Simulator()
begin
  while ( true ) {
    read users' positions in other
    walkthroughs from input;
    multicast event ext_obj;
  }
end Simulator
```

Figure 1.2: Pseudo code for VE3D

Module	Events of Interests	Procedures
Display	*obj* *ext_obj*	*update_viewpoint* *update_ext_obj*
Log	*obj* *ext_obj*	*log_obj* *log_ext_obj*

Figure 1.3: Module event interests in VE3D

invoke the corresponding procedures. When application modules are tailored to event interaction properties of the application, both system calls for event interaction in the underlying run time environment (also called interface software) and the application's functionality become part of the module implementation. Creation of interface software is difficult and time consuming, because it demands the comprehension of interaction among modules as well as an understanding of the architecture and communication mechanism of the target machine. Programmers have to modify the source code for the modules if the event-based application must be ported to a new environment. It is also difficult to reuse a module in a new application where event interactions are different.

There has been increasing interest in combining existing pieces of software to produce new applications as many realize that future breakthroughs in software productivity will depend on our ability to reuse components. By constructing new applications out of reusable software components, developers can build large, high-quality software applications using less time than building similar applications from scratch. These benefits led to a number of efforts to standardize component interaction [31] and to promote component-based software reuse [15] [44]. When programmers try to reuse software components, however, they cannot find exactly what they want; often such components need to be adapted before they can be reused. This is also the case when constructing event-based applications out of existing applications. We next discuss how to integrate separately developed but related event-based distributed applications. Looking back at VE3D and VE2D applications, the interoperation between them is clear. Instead of using a module to simulate users of other walkthroughs, we want to build an application out of the existing VE3D and VE2D in order to let two users (one in VE3D, the other in VE2D) control their own positions and display the other user as an object in their respective (3D or 2D) worlds. The question is how to integrate two related but separately developed applications without the cost of developing an entire third application (as shown in Figure 1.8). In order to build the composite application, the two original applications need to be adapted in the following areas:

- **Application Structure**

  The composite application will consist of eight modules as shown in Figure 1.8. The **Simulator** modules from both VE3D and VE2D are not needed in the composite application. Module **Simulator** in VE2D should be replaced by modules **Display**, **Controller** and **Log** from VE3D. Similarly, module **Simulator** in VE3D should be replaced by modules **Display_XY**, **Display_YZ**, **Display_XZ**, **Controller** and **Log** from VE3D.

- **Module Names**

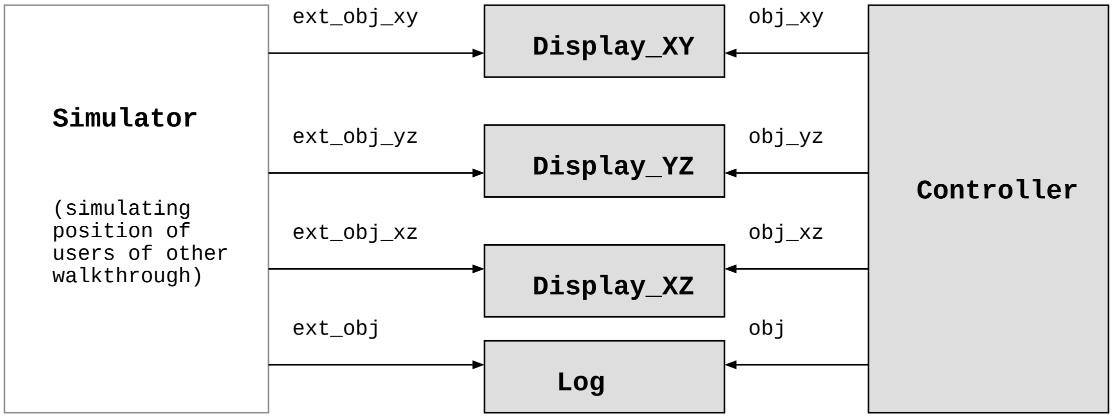

Figure 1.4: Application structure for VE2D

```
Display_XY()
begin
  while ( true) {
      e = next event;
      if event_name is obj_xy
          update_viewpoint_xy;
      else if event_name is ext_obj_xy
          update_ext_obj_xy;
  }
end Display_XY
```

```
Display_XZ()
begin
  while ( true) {
    e = next event;
    if event_name is obj_xz
       update_viewpoint_xz;
    else if event_name is ext_obj_xz
       update_ext_obj_xz;
  }
end Display_XZ
```

```
Display_YZ()
begin
  while ( true) {
      e = next event;
      if event_name is obj_yz
          update_viewpoint_yz;
      else if event_name is ext_obj_yz
          update_ext_obj_yz;
  }
end Display_YZ
```

```
Simulator()
  while ( true ) {
    read users' positions in other
    walkthroughs from input;
    multicast event ext_obj;
    multicast event ext_obj_xy;
    multicast event ext_obj_yz;
    multicast event ext_obj_xz;
  }
end Simulator
```

```
Log()
  while ( true) {
      e = next event;
      if event_name is obj
          log_obj;
      else if event_name is ext_obj
          log_ext_obj;
  }
end Log
```

```
Controller()
  while ( true ) {
    read current user's position
    from input;
    multicast event obj;
    multicast event obj_xy;
    multicast event obj_yz;
    multicast event obj_xz;
  }
end Controller
```

Figure 1.5: Pseudo code for VE2D

Module	Events of Interests	Procedures
Display_XY	*obj_xy* *ext_obj_xy*	*update_viewpoint_xy* *update_ext_obj_xy*
Display_YZ	*obj_yz* *ext_obj_yz*	*update_viewpoint_yz* *update_ext_obj_yz*
Display_XY	*obj_xz* *ext_obj_xz*	*update_viewpoint_xz* *update_ext_obj_xz*
Log	*obj* *ext_obj*	*log_obj* *log_ext_obj*

Figure 1.6: Module event interests in VE2D

In both applications, we have modules **Controller** and **Log**. When there is a conflict between module names, we need to differentiate one module from the other.

- **Event Names**

  Since the two applications have been fabricated separately, they may have different event naming systems. We need to match up events of interest. For example, both applications have events *obj* and *ext_obj*. In VE3D, event *obj* represents the current user's position. It corresponds to *ext_obj* (not *obj*) in VE2D, where it represents the other user's position. So event *obj* in VE3D should be matched up with event *ext_obj* in VE2D. Similarly, event *obj* in the VE2D should be matched up with event *ext_obj* in VE3D.

- **Event Structures**

  The event structures of two matching events may not match. For example, event *obj* in VE3D is defined to be three integers:

  ```
  obj (x:integer; y:integer; z:integer)
  ```

  while in VE2D, its matching event *ext_obj* is defined to be a string followed by a record structure consisting of three integers:

  ```
  ext_obj (name:string; {x:integer; y:integer; z:integer})
  ```

  We need to transform the event structure from one application to the other.

- **Event Generation**

  When some modules from the original applications are removed from the composite application, such a change may affect events generated or multicast by those modules. For example, in VE2D, events *ext_obj_xy*, *ext_obj_yz* and *ext_obj_xz* are only generated by module **Simulator**. When module **Simulator** is removed from the composite application, we need to decide whether or not we still need the missing events. If the missing events are still needed, we need to determine which module should generate them and how they should be generated.

One approach to adapting the existing applications for reuse in the new application is to modify the applications by hand, tailoring them to the way we want them to interact. Unfortunately, this

```
/* Controller.c */

#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   char s[256];
   int x, y, z;

   mh_init(&argc, &argv, NULL, NULL);
   mh_declare_event("obj", "III");
   while (gets(s, sizeof(s))!=NULL) {
      sscanf(s, "%d %d %d",
                &x, &y, &z);
      mh_multicast("obj", "III",
                     x, y, z);
   }
}
```

```
/* Display.c */

#include <stdio.h>

void update_viewpoint(x, y, z)
int x, y, z;
{
    ...
}

void update_ext_obj(name, x, y, z)
char *name;
int x, y, z;
{
   ...
}

main(argc, argv)
int argc;
char *argv[];
{
   char *event_name, *event_buf;
   int x, y, z;
   char *name;

   mh_init(&argc, &argv, NULL, NULL);
   mh_rgsmulticast("obj");
   mh_rgsmulticast("ext_obj");
   while (TRUE) {
      mh_next_event(event_name,
                    event_buf);
      if (strcmp(event_name,
                 "obj")==0) {
        mh_decode_event(event_buf,
                        &x,&y,&z);
        update_viewpoint(x, y, z);
      }
      else if (strcmp(event_name,
                      "ext_obj"==0) {
        mh_decode_event(event_buf,
                        name,&x,&y,&z);
        update_ext_obj(name, x, y, z);
      }
   }
}
```

Figure 1.7: Source code for **Controller** (left) and **Display** (right) in VE3D

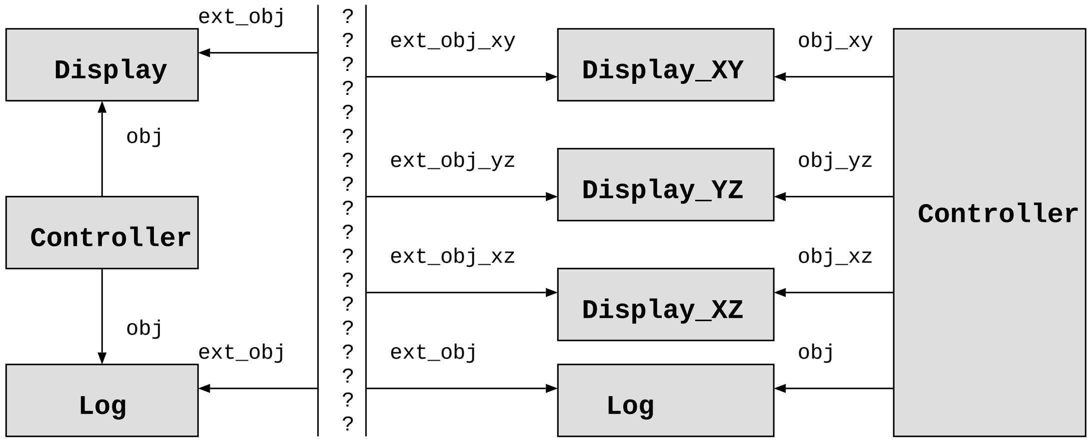

Figure 1.8: Application structure for the integrated VE3D/VE2D

is a difficult task for programmers, who must change the application structure, rename modules, match up events of interest, transform the event structures, and generate the missing events. Suppose programmers rename events *obj* and *ext_obj* in VE3D as *3D_obj* and *3D_ext_obj* so that the event names are unique. They must modify the source code to change the old event names to the new ones. For those missing events, programmers must modify the source code for module **Controller** in VE3D. Besides multicasting *3D_obj*, extra code must be added to project the right fields from the input data and to multicast events *ext_obj_xy*, *ext_obj_yz* and *ext_obj_xz*. In addition to event renaming and event generation, programmers also must decide which event structure to choose. If programmers choose to use the event structure in VE2D to represent points, i.e., if they choose to use a record structure consisting of three integers for a 3D point, and a record structure consisting of two integers for a 2D point, then they must change modules' source code in VE3D in order to multicast/receive events of the appropriate structure and to match procedures' formal parameters. For example, module **Controller** must aggregate the integers into record structures consisting of integers in order to multicast events of the appropriate event structures. Figure 1.9 illustrates how modules **Controller** and **Display** in VE3D from Figure 1.7 would grow in complexity merely in order to map the event names and event structures when adapted manually.

The above approach requires programmers to manually edit software modules' sources for reuse in a new application where the event interactions are different. If programmers introduce extra interfacing code into the existing software modules to have the event names and event structures correspond, it would make the application modules more complex, more costly to maintain, would increase the chance that programmers will introduce errors to the applications. The cost of adapting modules for reuse is high and the economic benefit of reusing modules resulting from this effort remains unrealized.

```
/* Controller.c */

#include <stdio.h>
struct {
    int a, b;
} point_2d;
struct{
    int a, b, c;
} point_3d;
main(argc, argv)
int argc;
char *argv[];
{
   char s[256];
   int x, y, z;

   mh_init(&argc, &argv, NULL, NULL);
   mh_declare_event("3D_obj", "{III}");
   mh_declare_event("ext_obj_xy", "{II}");
   mh_declare_event("ext_obj_yz", "{II}");
   mh_declare_event("ext_obj_xz", "{II}");

   while (gets(s, sizeof(s))!=NULL) {
      sscanf(s, "%d %d %d", &x, &y, &z);
      point_3d.a = x;
      point_3d.b = y;
      point_3d.c = z;
      mh_multicast("3D_obj", "{III}",
             "chen", point_3d);
      point_2d.a = x;
      point_2d.b = y;
      mh_multicast("ext_obj_xy", "{II}",
             "chen", point_2d);
      point_2d.a = y;
      point_2d.b = z;
      mh_multicast("ext_obj_yz", "{II}",
             "chen", point_2d);
      point_2d.a = x;
      point_2d.b = y;
      mh_multicast("ext_obj_xz", "{II}",
             "chen",  point_2d);
   }
}
```

```
    /* Display.c */

#include <stdio.h>
struct {
    int a, b;
} point_2d;
struct {
    int a, b, c;
} point_3d;

void update_viewpoint(x, y, z)
int x, y, z;
{
    ...
]

void update_ext_obj(name, x, y, z)
char *name;
int x, y, z;
{
    ...
}

main(argc, argv)
int argc;
char *argv[];
{
   char *event_name, *event_buf;
   char *name;

   mh_init(&argc, &argv,NULL,NULL);
    mh_rgsmulticast("3D_obj");
    mh_rgsmulticast("2D_obj");

    While (TRUE) {
       mh_next_event(event_name,
                     event_buf);
        if (strcmp(event_name,
                   "3D_obj")==0) {
           mh_decode_event(event_buf,
                           &point_3d);
           update_viewpoint(point_3d.a,
              point_3d.b, point_3d.c);
        }
        else if (strcmp(event_name,
                  "2D_obj")==0) {
           mh_decode_event(event_buf,
                  name, &point_2d);
           update_ext_obj(name,
                          point_2d.a,
                          point_2d.b);
        }
    }
}
```

Figure 1.9: Hand-coded source code for **Controller** (left) and **Display** (right) in VE3D. The adapted code is shown in bold type.

1.2 Our Integration Approach

In this work, we are concerned with the technology and methods for integrating an event-based application, whether that application is being constructed from scratch or is a synthesis of existing systems. In this section, we first describe our approach to integrating event-based distributed applications from scratch, we then describe our approach to building event-based distributed applications from existing applications. Finally, we give a brief description of the three layers of support we have provided for integrating event-based distributed applications.

Our approach to integrating event-based distributed applications from scratch is to separate the event interactions of an application from the implementation of the application modules, so that system integration can be performed using only the abstraction. Then, based on the abstract aggregate, customized interface software is generated automatically. Using our approach, programmers provide both the source code for the modules and an abstract description of them. The abstract description includes the module interaction properties of an application in a module interconnection language. Module interaction properties are the event-based behavior and communication of individual modules in an application. Given module interaction specifications for an application, the integration tool extracts and preprocesses the information and writes the information to a file, which is the input to both the software bus and an integration tool. The integration tool then creates a Makefile containing the configuration commands needed to build the application and the customized interface software that maps the specification to the execution environment.

The obligations for a user of this approach to building an event-based application from scratch are as follows:

1. Provide a specification of the event interaction properties of individual modules in the application.
2. Provide the source code for the application modules.
3. Execute the stub generator to generate interface software tailored to the environment.
4. Execute the generated configuration commands to create executables for the application.

The ability to reuse components is an economic necessity with software development projects. However, often such components must be adapted in some manner before they can be used. As the amount of adaptation increases, the economic benefit of reusing the component is decreased. In order to reduce the cost of manually adapting software for reuse when integrating existing applications, we turn to automatic techniques for transforming the abstract specifications into valid implementations.

We approach the problem of interoperation between existing applications by providing a specification language that allows programmers to express desired interactions between existing applications, and by then generating the valid implementation automatically. The generated software modules (also called coercion modules) implement programmer-defined interactions between existing applications. Given module interaction specifications for the existing applications and a composite interaction specification, our integration tools will analyze these interaction specifications and output a bus file and some event transformation files. The bus file contains a list of modules that make up the composite application as well as attributes associated with those modules. This preprocessed information is written in a format understood by the software bus, which will use the information written in the bus file to start up the composite application. The event transformation files contain a set of event transformation rules for mapping events from one application to another. Our integration tools can take the event transformation files as input and automatically generate the coercion modules and Makefiles for building the coercion modules. In this way, the interoperation can be achieved without the cost of developing a third entire system or the need for any changes to module implementations.

The obligations for a user of this approach to building an event-based application from existing applications are as follows:

1. Provide a specification of the event interaction properties for each existing application that will be part of the new application. [2]
2. Provide a specification of the event interaction properties for how the existing applications are to interact in the new application.
3. Provide the executables for the existing applications that will be part of the new application. [3]
4. Execute the stub generator to generate coercion modules tailored both to the interaction between existing applications and to the environment.
5. Execute the generated configuration commands to create executables for the coercion modules.

In using our integration approach programmers do not have to introduce extra interfacing code in the application modules. This allows the application source to remain simple and easy to maintain. Our approach reduces the opportunity of introducing errors if changes are made manually. The generated interface software is the valid implementation of the interaction. Since interactions are specified separately from the application modules, programmers can build applications of different interactions using the same application modules, allowing the software modules to be reused in a broad range of applications. However, the real value of this work is realized when programmers seek to change the event interactions, because they can reason about the interaction at the abstract level and then have the valid interface software generated automatically. In this way, programmers can define complex event interactions abstractly without having to edit the modules' sources; thus making it easier to integrate and experiment with event-based distributed applications, and facilitating reuse in a broader range of applications.

[2]The specification for each existing application is already created when the application is built from scratch.
[3]The executable for each existing application is already created when the application is built from scratch.

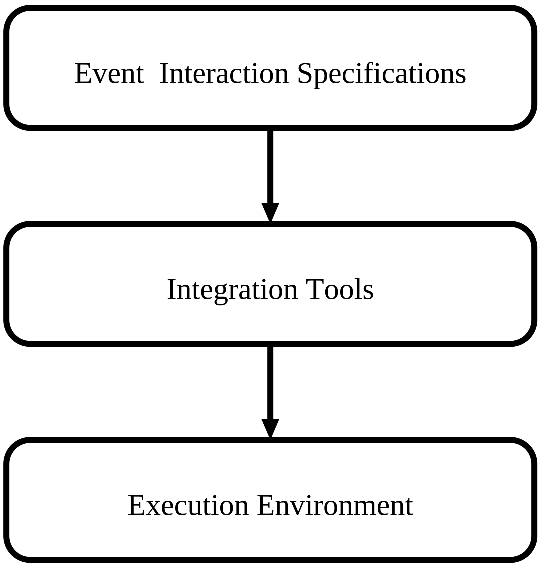

Figure 1.10: Three layers of support for integrating event-based distributed applications

In this thesis, we provide three layers of support for integrating event-based distributed applications as shown in Figure 1.10. The top layer presents the specification languages used to describe event interaction properties. The middle layer is a set of integration tools that analyze specification languages and generate interface software that maps the specifications to the underlying environment. The bottom layer provides an execution environment built on a software bus model to accommodate event-based distributed applications.

At the top layer, we focus on specifying *event interaction properties* of distributed applications, which properties include not only the means to support the module interactions in an event-based application, but also the means to facilitate the interoperation between separately developed but related event-based distributed applications. We have defined the specification languages used to describe event interaction properties, which can be divided into two types of requirements: module interaction requirements and composite interaction requirements.

- **Module interaction requirements** are the event-based behavior and communication of individual modules in an application. Events that are sent to or received by modules should be declared along with their event structures. In order to let modules interact in an event-based application, modules need to register their interest in a set of events by associating a procedure with each event name. These are called *event-triggered rules* or *event bindings*. More than one module may be interested in the same event, but the modules may react differently upon receiving them. A module can generate or multicast multiple events, and an event can be generated by different modules.
- **Composite interaction requirements** define a set of relations among the existing event-

based distributed applications and describe the way we want them to interact. They help us relate the *abstractions* among the existing applications, such as the applications from which the composite application is built, the modules from the original applications which are no longer needed in the composite application, the means to rename modules so that module names are unique, and the means of mapping an event from one application to another, including event names and event structures.

At the middle layer, we concentrate on analyzing and using event interaction specifications in the integration process. We have developed a set of integration tools for building an event-based application both from scratch and from the existing applications. The integration tools will help programmers analyze specification languages, generate interface software that maps the specifications to the underlying environment, and preprocess information for the execution environment to support interoperation among existing applications.

At the bottom layer, we have built an execution environment to support event-based distributed applications. An environment to support event-based distributed applications in the presence of heterogeneity should meet the following requirements. While the first two requirements apply to building event-based applications both from scratch and from existing applications, the last requirement is solely for supporting interoperation among the existing applications:

1. **Communication across heterogeneous hosts:** Communication between modules should function normally in the presence of system architecture heterogeneity. The environment must not compromise the data type system of the programming languages in which the modules are written. The parameters must be marshaled correctly. The low-level representations of primitive data types on diverse underlying architectures should match.

2. **Event multicast:** Modules should be able to declare events, define event structures, register interest in events, multicast events, and receive events of interest.

3. **Event mapping:** The environment should be able to transform events from one application to another at run time; such transformations should include event names and event structures. The transformation should be transparent to the application modules.

The main contributions of this work are simplifying the task of building event-based distributed applications by helping programmers define and analyze the event interaction properties, and providing support for integrating an application from scratch and for enabling interoperation among existing event-based applications. We have defined an approach to integrating event-based applications at the configuration-level by defining, relating, and using the abstractions of event interactions.

1.3 Related Work

There are many systems which use implicit invocation to compose their components. These systems can be divided into three categories. The first category is based on special-purpose languages. Systems in this category use specialized notations and run-time support to access implicit invocation. The specialized notations are designed into the language from the start. Examples are given in [18].

The second category is adding implicit invocation to traditional programming languages, such as Ada [18], C++ [40], and Common Lisp [20]. The objective of work in this category is to make implicit invocation available in traditional programming languages without defining special-purpose mechanisms. Notkin *et al.* [30] discuss design issues that arise when embedding implicit invocation in traditional programming languages. Although the basic mechanisms have substantial similarities, the details differ significantly because of the natures of the underlying programming languages.

The third category comprises systems which can be classified as integration frameworks. In this category, a system is usually composed of a group of components running as separate processes. These components interact with each other by multicasting events or messages. An agent implemented as a separate process is responsible for routing events through communication channels to the components. Examples are Field [38], Forest [16], HP SoftBench [19], and Sun ToolTalk [26] [43]. Our work falls into the third category.

Field [38] is a principal programming environment for teaching undergraduates. It is also a research programming environment and a testbed for developing new tools. Field uses message passing to integrate a set of tools, including an annotation editor, a cross-referencer, a debugger, a viewer, etc. Field shows how its selective broadcast message facility Msg connects Unix tools. In Field, tools register patterns with Msg to describe the messages that interest them. When a message is received by Msg, it routes the message to the appropriate tools. Interfaces are added to the existing Unix tools to allow them send/receive messages and take appropriate actions. Since the interface software is part of the implementation, policies of when and how the tools are invoked are hand-coded in the program. Different tool interaction would require a change in the interface software.

Forest is an extension of Field [16]. As in Field, Forest's tool integration framework is based on selective broadcast. Unlike Field, Forest separates policies of when and how the tools are invoked from what tools do when they are invoked, with the goal of allowing dynamic configurable policies of tool interaction. In Forest, policies are implemented by a set of condition-action pairs, where conditions are boolean expressions of numeric expressions, functions, and state variables. Programmers can change the values of state variables dynamically. Although policies of tool interaction are specified separately, programmers must still manually adapt the existing Unix tools to send and receive messages.

The HP SoftBench product is an integrated software development environment designed to facilitate rapid, interactive program construction, testing, and maintenance in a distributed computing environment. The environment provides an architecture for integrating a set of CASE tools, including a program editor, a static analyzer, a program debugger, a program builder, and mail. As in Field, the SoftBench integration framework is also based on selective broadcast.

The HP Encapsulator can build message-based application program interfaces to the tools [14]. To encapsulate a tool means to integrate it into the HP SoftBench architecture. The HP Encapsulator does not require changes to the source code of the tools being encapsulated. It allows users to integrate tools for which no source code is available. However, the tools to be encapsulated must have command-line interfaces, as in the case with Unix tools, script files, etc. Highly interactive tools are not good candidates for encapsulation, since the HP Encapsulator does not comprehend the tool's activity. In order to achieve a better level of event granularity, each operation needs a unique command-line interface.

There are also efforts to formalize design spaces of implicit invocation mechanisms and event-based software integration frameworks [17] [1]. The goal of such research is to identify common properties of these systems so that researchers can reason about the properties of certain systems as well as the relationships among different systems. A potential research area for future research is to explore the interoperability issues among different mechanisms.

1.4 Outline of the Thesis

The rest of the thesis is organized as follows:

- Chapter 2 discusses top-layer support and introduces the specification languages used to describe module interaction requirements and composite interaction requirements.
- Chapter 3 discusses bottom-layer support and describes our underlying execution environment based on event multicasting.
- Chapter 4 describes middle-layer support and presents our integration method as well as the means of generating interface software that maps the given specification to our underlying event-based execution environment.
- Chapter 5 presents the use of our approach in some more complex examples.
- The final chapter summarizes this work.

Chapter 2

Event Interaction Specification

In this chapter, we discuss how to specify event interaction properties in order to generate the interface software automatically and to support interoperation among existing applications. Section 2.1 first describes the use of enhanced POLYLITH MIL to specify module interaction requirements and then gives specifications for modules in VE3D and VE2D applications. In Section 2.2 we discuss how to use a specification language ENIMBLE to specify composite interaction requirements and to illustrate its use in the integrated VE3D/VE2D example.

2.1 Module Interaction Requirements

Module interaction requirements are the abstract descriptions of the event-based behavior and communication of individual modules in an application. We use the POLYLITH Module Interconnection Language (MIL) [23] [21] to specify interfaces and attributes for each module; these specifications include the location of the executable code, the name of the host machine where it should run, and the interfaces of a module. In addition to information about interfaces and attributes, we also need to specify information about events for each module, including the events in which the module is interested, their event structures, the procedures that are associated with events of interest. Information about module interaction requirements is made part of the specifications of a module. Figure 2.1 and Figure 2.2 show how an enhanced POLYLITH MIL could be used to specify module interaction requirements for VE3D and VE2D applications, respectively.

In the enhanced POLYLITH MIL, the `declare` clauses specify events that a module will use. An event consists of an event name and an event structure. An event name is a string of characters. An event structure could be any legal data type of a programming language in which the module is written. In Figure 2.1, module **Controller** declares an event: the event name is *obj*; the event structure is three integers x, y, and z. In Figure 2.2, module **Simulator** declares four events: *ext_obj*, *ext_obj_xy*, *ext_obj_yz*, and *ext_obj_xz*. The event structure for *ext_obj* is a string

```
module "Display":{
  binary = "/thumper/chenchen/display";
  machine = "thumper.cs.umd.edu";
  file = "display_user.c";
  when obj => update_viewpoint;
  when ext_obj => update_ext_obj;
}

module "Simulator":{
  binary = "/bugs/chenchen/simulator";
  machine = "bugs.cs.umd.edu";
  file = "simulator_user.c";
  declare  ext_obj(name: string; x: integer; y: integer; z: integer);
}


module "Controller":{
  binary = "/harvey/chenchen/controller";
  machine = "harvey.cs.umd.edu";
  file = "controller_user.c";
  declare obj (x:integer; y:integer; z:integer);
}


module "Log":{
  binary = "/xring/chenchen/log";
  machine = "xring.cs.umd.edu";
  file = "log_user.c";
  when obj => log_obj;
  when ext_obj => log_ext_obj;
}

orchestrate "VE3D":{
  tool "Display";
  tool "Simulator";
  tool "Controller";
  tool "Log";
}
```

Figure 2.1: Module interaction requirements for VE3D using an enhanced POLYLITH MIL

```
module "Display_XY":{
  binary = "/thumper/chenchen/display_xy";
  machine = "thumper.cs.umd.edu";
  file = "display_xy_user.c";
  when obj_xy => update_viewpoint_xy;
  when ext_obj_xy => update_ext_obj_xy;
}
module "Display_YZ":{
  binary = "/thumper/chenchen/display_yz";
  machine = "thumper.cs.umd.edu";
  file = "display_yz_user.c";
  when obj_yz => update_viewpoint_yz;
  when ext_obj_yz => update_ext_obj_yz;
}
module "Display_XZ":{
  binary = "/thumper/chenchen/display_xz";
  machine = "thumper.cs.umd.edu";
  file = "display_xz_user.c";
  when obj_xz => update_viewpoint_xz;
  when ext_obj_xz => update_ext_obj_xz;
}
module "Simulator":{
  binary = "/bugs/chenchen/simulator";
  machine = "bugs.cs.umd.edu";
  file = "simulator_user.c";
  declare  ext_obj(name: string; {x: integer; y: integer; z: integer});
  declare  ext_obj_xy(name: string; {x: integer; y: integer});
  declare  ext_obj_yz(name: string; {y: integer; z: integer});
  declare  ext_obj_xz(name: string; {x: integer; z: integer});
}
module "Controller":{
  binary = "/harvey/chenchen/controller";
  machine = "harvey.cs.umd.edu";
  file = "controller_user.c";
  declare obj ({x:integer; y:integer; z:integer});
  declare obj_xy ({x:integer; y:integer});
  declare obj_yz ({y:integer; z:integer});
  declare obj_xz ({x:integer; z:integer});
}
module "Log":{
  binary = "/xring/chenchen/log";
  machine = "xring.cs.umd.edu";
  file = "log_user.c";
  when obj => log_obj;
  when ext_obj => log_ext_obj;
}
orchestrate "VE2D":{
  tool "Display_XY";
  tool "Display_YZ";
  tool "Display_XZ";
  tool "Simulator";
  tool "Controller";
  tool "Log"
}
```

Figure 2.2: Module interaction requirements for VE2D using an enhanced POLYLITH MIL

name, followed by a record consisting of three integers *x*, *y*, and z. The brackets represent record structures.

The `when` clauses define which procedures will be invoked when events of interest are received by the module. These are called *event-triggered rules* or *event bindings*. An event-triggered rule consists of two parts: an event-name trigger and a procedure body. The semantics of event-triggered rules is outlined as follows. When an event of interest is received by a module, the module checks the event name with the trigger of each rule. When a match is found, the corresponding procedure is invoked. The event structure is passed to the procedure. In Figure 2.1, module **Display** uses `when` clauses to register its interest in two events, *obj* and *ext_obj*, and to define two event-triggered rules:

> *when obj* ⇒ *update_viewpoint*;
> *when ext_obj* ⇒ *update_ext_obj*;

Each module has its own set of event bindings, which are only visible to the module and are not affected by event bindings of other modules in the application. Thus, modules can associate different procedures with the same event in different modules. As in **Display**, module **Log** in Figure 2.1 is interested in events *obj* and *ext_obj* but has different event bindings:

> *when obj* ⇒ *log_obj*;
> *when ext_obj* ⇒ *log_ext_obj*;

Within an individual module, the mappings between events and procedures can be one-to-one or many-to-one.

Rather than creating a new language, we have enhanced the POLYLITH MIL in order to describe module interaction requirements. We made this decision for two reasons:

- Since module interaction requirements are the abstract description of the event-based behavior of an individual module, and the module specification of the original POLYLITH MIL already describes attributes and interfaces associated with an individual module, it is only natural to include the module interaction in the specifications of an individual module.
- For those programmers who are already familiar with POLYLITH MIL, it should not be difficult to learn the new event-related statements.

The declarative nature of events fits well with the concept of implicit invocation. For example, the `declare` clauses define event names and event structures; the `when` clauses associate procedures with event names. There should be no conceptual problems for programmers to transfer their understanding of the abstract model of implicit invocation into specifications in the enhanced POLYLITH MIL.

```
composite "VE3D/VE2D":{
  application VE3D;
  application VE2D;

  exclude Simulator from VE3D;
  exclude Simulator from VE2D;

  name Controller from VE3D Controller_3D;
  name Log from VE3D Log_3D;

  map VE3D obj(x: integer; y: integer; z: integer)
  to VE2D ext_obj(string('chen'); {x; y; z})
  to VE2D ext_obj_xy(string('chen'); {x; y})
  to VE2D ext_obj_yz(string('chen'); {y; z})
  to VE2D ext_obj_xz(string('chen'); {x; z});

  map VE2D obj({x:integer; y: integer; z: integer})
  to VE3D ext_obj(string('jim'); a.x; a.y; a.z);
}
```

Figure 2.3: Composite interaction specification for the integrated VE3D/VE2D application using ENimble

Module interaction specifications are used to construct event-based distributed applications from scratch. In the integration process, the specification will be the input of a stub generator for preparing the interface software for execution. The details of the integration process are described later in Chapter 4.

2.2 Composite Interaction Requirements

Composite interaction requirements define a set of relations among existing event-based distributed applications in order to support the interoperation between them. We have defined a composite interaction specification language called ENimble, which allows programmers to describe the way they want the existing applications to interact [8]. In this language, we specify applications from which the the composite application is built. Modules that no longer need to be in the composite application are identified. If there is a conflict between module names, we can rename the software modules. When there are different event naming systems and different event structures, we can map an event from one application to another. An earlier version of this language can only map events with different event names but the same event structures [7], ENimble has advanced features for mapping events of different event structures. Figure 2.3 shows the composite interaction specification for the integrated VE3D/VE2D example.

The `application` clauses enumerate the applications from which the composite application is built. In Figure 2.3, we specify that the composite application is built from VE3D and VE2D, thus corresponding to identifiers following `orchestrate` in the module interaction specifications in Figure 2.1 and Figure 2.2.

The `exclude` clauses identify modules from the original applications that are not needed in the

maplist	::=	maplist `map` application event (event-structure) tolist | ϵ
tolist	::=	`to` application event (enimble) tolist | ϵ
event-structure	::=	arg | arg; event-structure | ϵ
arg	::=	unlabeled-arg | labeled-arg
labeled-arg	::=	label : unlabeled-arg
enimble	::=	primitive(value) | label | primitive(label) |
		primitive(label); enimble |
		primitive(value); enimble | label; enimble | {enimble} | ϵ
unlabeled-arg	::=	primitive | structured | pointer | array
structured	::=	{event-structure}
array	::=	primitive[indexlist]
indexlist	::=	number | number, indexlist
pointer	::=	$\uparrow$ unlabeled-arg
primitive	::=	`boolean` | `integer` | `float` | `string` ...

Figure 2.4: ENimble grammar

composite application. In Figure 2.3, we specify that module **Simulator** from VE3D application and module **Simulator** from VE2D application are excluded from the composite application. While we could enumerate all the modules we want to keep instead of those we delete from the two original applications, we chose not to do so.

The `name` clauses can be used to rename modules when there is a conflict between module names. Since there are two modules named **Controller**, we rename **Controller** from VE3D as **Controller_3D** so that when it is integrated with other modules, it will not be confused with module **Controller** from VE2D.

The `map` clauses map events from one application to another when they have different event names and/or different event structures. ENimble allows programmers to rearrange the order of the parameters, to initialize data, to mask out data, to coerce data, to flatten and aggregate data. The event mapping notation in the ENimble language is similar to the interface mapping notation in Nimble, a language for declaring how the actual parameters in a procedure call are to be transformed at run time [36]. The BNF for the `map` clauses in ENimble is shown in Figure 2.4.

In ENimble, each parameter in an event structure is assigned a unique name, either provided either by programmers or by the system. The labeling allows programmers to refer to parameters explicitly when declaring an event map from one application to another. Maps, are a list of labels separated by semicolons; they contain optional curly brackets to indicate record structures, optional square brackets to indicate arrays, and optional initial values. A map is created in terms of the labels of the event structure.

Now let us look at some examples to illustrate six basic coercion scenarios that can be handled by our event mapping notation. In the following examples, we map event *event1* in application

A to event *event2* in application B:

1. Rearrange the order of the parameters

   `Example 1`: Event *event1* from application A consists of an integer x and and a string of characters y. Event *event2* from application B consists of a string and an integer. We want to have the parameters reordered after the map.

   ```
   map A event1(x: integer; y: string)
   to B event2(y;x)
   ```

2. Initialize data

   `Example 2`: Event *event1* from application A consists of two integers: x and y. Event *event2* from application B consists of two integers followed by a string. We want to keep the parameters in the same order and have the string initialized to '`hello`' after the map.

   ```
   map A event1(x: integer; y: integer)
   to B event2(x; y; string('hello'))
   ```

3. Mask out data

   `Example 3`: Event *event1* from application A consists of two integers: x and y. Event *event2* from application B is an integer. We want to keep only the second parameter of *event1* after the map.

   ```
   map A event1(x: integer; y: integer)
   to B event2(y)
   ```

4. Coerce data

   `Example 4`: Event *event1* from application A consists of two integers: x and y. Event *event2* from application B consists of an integer and a string. We want to keep the parameters in the same order and cast y to a string after the map.

   ```
   map A event1(x: integer; y: integer)
   to B event2(x; string(y))
   ```

5. Flatten data

   `Example 5`: Event *event1* from application A is a record consisting of two integers: x and y. Event *event2* from application B consists of two integers. We want to keep the parameters in the same order after the map.

   ```
   map A event1({x: integer; y: integer})
   to B event2(a.x; a.y)
   ```

   where `a` is a system-provided label. Programmers could also provide their own descriptive labels. For example,

```
map A event1(point:{x: integer; y: integer})
to B event2(point.x; point.y)
```

6. Aggregate data

   `Example 6`: Event *event1* from application A consists of two integers: x and y. Event *event2* from application B is a record consisting of two integers. We want to keep the parameters in the same order after the map.

   ```
   map A event1(x: integer; y: integer)
   to B event2({x; y})
   ```

The above list only addresses the basic coercion scenarios. Programmers can combine any of them when creating a map. The mapping between events of two different naming systems could be one-to-one, one-to-many or many-to-one as shown in Figure 2.5. The many-to-one mapping is not shown in this particular example. In Figure 2.3, the clause

```
map VE3D obj(x: integer; y: integer; z: integer)
to VE2D ext_obj(string('chen'); {x; y; z})
to VE2D ext_obj_xy(string('chen'); {x; y})
to VE2D ext_obj_yz(string('chen'); {y; z})
to VE2D ext_obj_xz(string('chen'); {x; z})
```

maps event *obj* in VE3D to four events in VE2D:

1. event *ext_obj* consists of a string initialized to `'chen'`, followed by a record consisting of all three integers of event *obj*. It aggregates the three integers;
2. event *ext_obj_xy* consists of a string initialized to `'chen'`, followed by a record consisting of the first two integers of event *obj*. The parameter `z` is masked out;
3. event *ext_obj_yz* consists of a string initialized to `'chen'`, followed by a record consisting of the last two integers of event *obj*. The parameter `x` is masked out;
4. event *ext_obj_xz* consists of a string initialized to `'chen'`, followed by a record consisting of the first and the third integers of event *obj*. The parameter `y` is masked out;

The clause

```
map VE2D obj({x: integer; y: integer; z: integer})
to VE3D ext_obj(string('jim'); a.x; a.y; a.z)
```

flattens the record of *obj* and adds a string `'jim'` before the three integers.

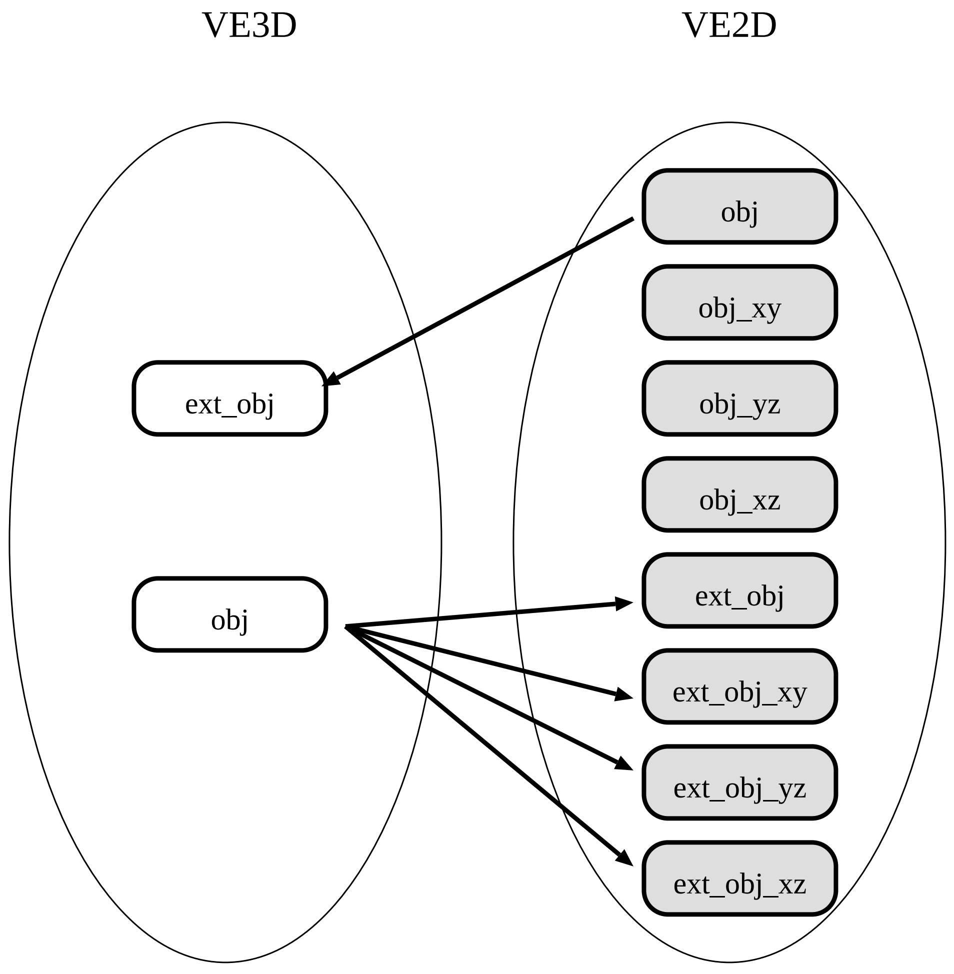

Figure 2.5: Event mappings between VE3D and VE2D

Notice that not all events in VE3D and the VE2D appear in the ENIMBLE specification given in Figure 2.3. We only map events that the other application is interested in receiving, i.e., events that have matching events in the other application. Events in which the other application is not interested do not appear in the ENIMBLE specification because they are irrelevant to the application's interoperation. For example, VE3D is only interested in receiving event *obj* from VE2D, and VE2D is only interested in receiving event *obj* from VE3D. Neither is interested in the rest of the events from the other application. Therefore, we map only events *obj* in VE3D and VE2D in the ENIMBLE specification.

There are several advantages of using the ENIMBLE specification language for composite interaction requirements. First, ENIMBLE provides a place for a user to describe composite interaction requirements separately from module interaction requirements. In addition, ENIMBLE is a declarative language similar to the POLYLITH MIL. For programmers who are familiar with the POLYLITH MIL, it should not be difficult to learn ENimble. Finally, ENIMBLE can be used to express complex composite interaction requirements among existing applications. Although the example in this section shows only the composite interaction between two applications, ENimble can be used to relate abstractions of more than two applications. When more applications are involved, the interactions among them become more complex, the amount of manual adaptation requires more programming effort, and the ability to define the interactions at a high level and then automatically to generate valid coercion modules becomes more important and more desirable.

There are also some limitations of ENIMBLE which should be understood before an ENIMBLE program is attempted. The current version of ENIMBLE does not allow for any computation (other than previously stated coercions) to be done when events are mapped. For example, with two events from different coordinate systems, we cannot specify an arbitrary event mapping because there are many ways that an event could be mapped. ENIMBLE is not intended to solve all the problems of event mapping. However, the current version could be extended to deal with such situations. We could provide extensions for code/expressions to be evaluated by allowing programmers to provide the name of a customized event mapping routine. We could easily generate the skeleton for the event mapping routine in the corresponding coercion module and let programmers fill in what they need.

The cost of creating composite interaction specifications is the cost of understanding the interactions among applications in addition to the cost of translating such interactions into ENIMBLE. The most challenging aspect of creating the composite interaction specifications is to understand the way the existing applications interact. This task requires programmers to understand the configuration of the applications as well as to be familiar with the event names and structures involved. Once the interaction is clear at the conceptual level, it should not be difficult for programmers to translate it into specification in ENIMBLE.

The composite interaction specification given by programmers, along with module interaction specifications, will be used as the input of an integration tool for integrating existing event-based distributed applications. The details of how to use these abstractions to enable the interoperation among existing systems are given in Chapter 4.

Chapter 3

Execution Environment

In this chapter, we describe our underlying event-based execution environment. In the Introduction, we listed the requirements of the execution environment, including the ability to communicate across heterogeneous hosts, to multicast events, and to transform events. Our approach to meeting the above requirements is to build an execution environment upon the existing POLYLITH Software Interconnection System. The POLYLITH already provides programmers with an environment that facilitates construction of distributed applications. Using POLYLITH, the modules interface directly with a software toolbus for execution in heterogeneous environments. POLYLITH currently helps users accommodate heterogeneity in their choices of programming languages, host operating systems, and communication mechanisms. The software toolbus already manages data marshaling by 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 data marshaling, the software toolbus coerces the low-level representation of primitive data types on different underlying architectures. Thus, the POLYLITH Software Interconnection System satisfies our first requirement.

The remaining requirements for an event-based execution environment were not supported by the original POLYLITH system. Our approach is to enhance the POLYLITH software toolbus by adding multicast primitives and transformation facilities to support the remaining requirements. Section 3.1 describes a set of multicast primitives. Section 3.2 describes our event mapping facilities.

3.1 Multicast

Multicast is a mode of module interaction where an event produced by some module can be sent to multiple modules at the same time. In order to enable modules to interact in a multicasting environment, application modules need to specify what events they are interested in receiving. When an event is multicast from some module, all modules that are interested in the event can receive it. Modules that want to multicast (or publish) an event are not required to know which modules are interested in the event, and modules that want to receive (or subscribe) events are not required to know where the events come from.

In Section 3.1.1, we discuss the design issues related to adding multicast primitives to a software bus model. These multicast primitives allow modules to declare events, to define their event structures, to register events of interest, to multicast events, and to receive events of interest. We illustrate the use of these primitives in a database example in Section 3.1.2.

3.1.1 Design Issues

There are several issues to consider when adding multicast primitives to a software interconnection system. These issues are event declaration, event structure, event bindings, multicast events, receive events, and concurrency. Garlan *et al.* discussed design considerations when adding implicit invocation to three traditional programming languages: Ada, C++, and Common Lisp [30]. Different design decisions were made because of the nature of the different languages. Our discussion focuses on how the nature of a software bus influenced our design decisions.

In the following discussion of each of the design issues, we first briefly introduce possible approaches as summarized by Garlan and Scott [18]; we then describe our approach and explain why this approach was chosen.

- **Event Declaration** — `fixed event vocabulary`, `static event declaration`, `dynamic event declaration`, and `no event declaration`.

  A `fixed event vocabulary` allows programmers to use only a fixed set of events determined before the application is created. `Static event declaration` requires that all events be declared at compile time. These two approaches do not fit well with a software bus model, which supports dynamic reconfiguration activities [37]. For example, modules that are added later to an application during dynamic reconfiguration could declare new events. Obviously, the first two approaches do not support this situation. Field and HP SoftBench took the fourth approach and chose not to declare events in their systems. Components in these systems can multicast events named by arbitrary strings, but events used in the system are typically described in external documents.

  We have chosen `dynamic event declaration`, which fits well with the software bus model. In order for this method to run with a software bus, a module is usually written in a traditional programming language (such as C) with some library calls or primitives to interface and synchronize with the software bus. An event declaration primitive can appear anywhere in a program before the event is used. In other words, modules can declare any event dynamically at run time.

  One thing to notice is that all events can be declared within a single module or the declaration could be distributed across modules. In either situation, every module in the application can use the declared event. Usually a module declares the events that it will multicast.

- **Event Structure** — `simple event names`, `fixed event structures`, `structures determined by event name`, and `structures determined by announcement`.

  `Simple event names` imply that an event has only an event name and no event structure. This is clearly insufficient for a software bus model, where modules of a distributed application usually need to exchange data. `Fixed event structures` imply that all events in the system have the same event structure. One can easily imagine a situation where modules need to exchange a variety of events of different event structures. `Structures determined by announcement` means that a module specifies the event structure only when it multicasts the event; the event structure can vary. If an event does not have a fixed event structure, it maybe difficult for modules to decode the event as well as to interpret the meaning of the data.

  We have chosen the third approach, `structures determined by event name`. Each event has a fixed event structure (or list of parameters). There are several advantages of this approach. First, it is easy for the software bus to perform type checking. In addition, it is easy for modules to decode events. Finally, it allows modules to send/receive events of different structures. Two or more events can have the same physical structure but different logical meanings.

- **Event Bindings** — `static event bindings` and `dynamic event bindings`.

  `Static event bindings` require that procedures be associated with events at compile time. In this approach, modules cannot change their interest in events at run time. This approach prevents modules from re-associating procedures with events. As such, it is not flexible enough for distributed applications.

  We have chosen the `dynamic event binding` approach. Modules can register and unregister their interest in events dynamically at run time. Using our approach, programmers have total control of what to do when events are received by modules. A primitive is designed to decode an event into appropriate parameters according to its event structure. Programmers can pass either some or all of the parameters to a procedure, or they may do some computation with the event.

- **Multicast Events** — `single multicast procedure`, `multiple multicast procedure`, `language extension`, and `implicit multicast`.

  `Language extension` refers to the addition of implicit invocation to traditional programming languages. It does not fit in our situation. `Single multicast procedure` implies a single procedure or primitive for multicasting all events in the system. In contrast, `multiple multicast procedure` uses one procedure for each event name so that the procedure to multicast events can be designed to take exactly the same parameters of the event. For example, the number of the parameters is exactly the same, and the type and order of the parameters match exactly. This approach does not fit with a software bus model because the software bus is designed for general distributed applications and not for a particular application. It is impossible to provide a multicast primitive for each event. The `implicit multicast` approach allows events to be multicast as a side effect of executing some procedures.

`mh_declare_event (event_name, event_structure)`
Declare an event
`mh_rgsmulticast (event_name)`
Register its interest in an event
`mh_unrgsmulticast (event_name)`
Unregister its interest in an event
`mh_multicast (event_name, event_structure,` $var_1, \ldots, var_n$`)`
Multicast an event
`mh_signal_multicast (event_name, event_structure,` $var_1, \ldots, var_n$`)`
Multicast an event and send signals at the same time
`mh_bgetmsg (event_structure, event_name,` $var_1, \ldots, var_n$`)`
Receive an event (blocking)
`mh_getmsg (event_structure, event_name,` $var_1, \ldots, var_n$`)`
Receive an event (non-blocking)
`mh_next_event (event_structure, event_name, event)`
Receive an event (blocking)
`mh_decode_event (event_structure, event,` $var_1, \ldots, var_n$`)`
Decode an event according to its event structure
`mh_nomsg (event)`
Declare a standard null event
`mh_query_msgtype (event)`
List all declared events
`mh_query_rgsmsgtype (event)`
List all event names in which which it has registered an interest

Figure 3.1: Polylith Multicast Primitives

We have taken the first approach of providing a single multicast primitive for all events. Our multicast primitive takes an event name, an event structure, and actual parameters. It is flexible enough to take a variable number of parameters. Modules can multicast any declared events. A module can multicast events declared either by itself or by other modules. We have also implemented a variation of the multicast primitive. When a module multicasts an event, it also has the capability of sending signals to modules that are in interested in the event.

- **Receive Events** — `blocking` and `non-blocking`.

  The first approach allows a module to block its execution when waiting for a multicast event. The second approach allows a module to receive events in a non-blocking fashion. If there is are no events available, a module can continue its execution. Both situations are quite common in distributed applications. We have implemented both blocking and non-blocking primitives for receiving events.

- **Concurrency** — `concurrency` and `no concurrency`.

  Because of the nature of distributed computing, different modules need the capability of executing different callback functions concurrently upon receiving the same event.

We have added a set of multicast primitives to the POLYLITH software interconnection system as shown in Figure 3.1. A detailed description of the multicast primitives is given in Appendix A and in the POLYLITH programming manual [21].

3.1.2 The Database Example

In this section, we introduce a database application in order to demonstrate the use of the multicast primitives. Figure 3.2 shows an application consisting of five modules **input1, input2, db, output1** and **output2**, which are distributed across different host machines in order to balance the load on the system. Module **db** is a front-end to a relational database. Modules **input1** and **input2** are simply input windows where a user of the application can enter SQL queries. Module **db** will receive the queries sent from **input1** and **input2** and will multicast the results of the queries to **output1** and **output2**, the modules responsible for displaying the results of the queries. Pseudo code for module **input1**(**input2**), **output1**(**output2**) and **db** in this database application is shown in Figure 3.3.

Figure 3.4 is the application specification for the database example using POLYLITH's Module Interconnection Language (MIL). In the MIL program, we specify information for each module, including the location of the executable code and the name of the host machine where it should run. For the application as a whole, we specify what modules make up the application (using *tool*).

Programmers can take ordinary software modules and add multicast primitives to the modules' source code to execute in a multicast-based environment. Figure 3.5 shows what the pseudo code looks like in POLYLITH for the database example using our multicast primitives. Module **db** calls *mh_declare_event* to declare event *db_event*. By using *mh_rgsmulticast*, **db** registers its interest in events *input1_event* and *input2_event*. Then **db** calls *mh_bgetmsg* to receive an event. If the event queue for the multicast interface of module **db** is not empty, then an event is dequeued and sent back to **db**; otherwise it waits for events. When an event arrives, module **db** consults the database and multicasts the result of the query as event *db_event*. Since module **output1** and **output2** are interested in event *db_event*, a copy of the event is enqueued on interfaces of both **output1** and **output2**.

The source code needs to be compiled and linked with the POLYLITH library to create the executable objects for the application modules. Once these steps are taken, the executables need to be installed in the appropriate paths on their destination machines. The information described in the MIL program is processed, and at run time POLYLITH's software toolbus is responsible for invoking processes and for coercing data representation, synchronization, and communication among application modules.

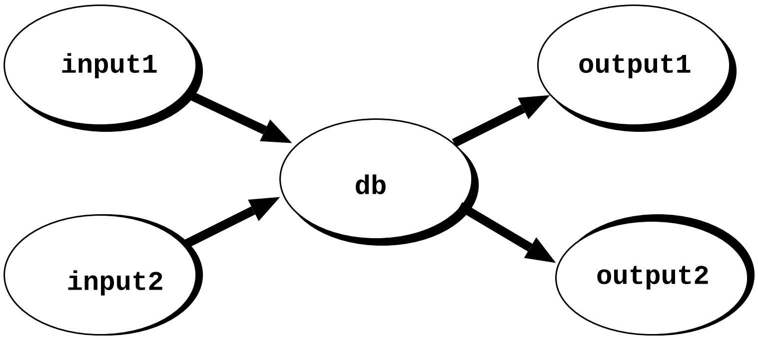

Figure 3.2: Application structure for the database example

```
input1()
begin
  q = query;
  while ( q is not empty) do
    send q to database;
    q = next query;
  end while
end input1

output1()
begin
  while ( true ) do
    r is the results received from database;
    write r to output;
  end while
end output1

db()
begin
  while ( true ) do
    q is a received query
    r = result of executing query q;
    send r to output modules;
  end while
end db
```

Figure 3.3: Modules for the database example

```
module "input1":{
   binary = "/jteam/input1.exe";
   machine = "bugs";
}
module "input2":{
   binary = "/jteam/input2.exe";
   machine = "thumper";
}
module "db":{
   binary = "/jteam/db.exe";
   machine = "xring";
}
module "output1":{
   binary = "/jteam/output1.exe";
   machine = "harvey";
}

module "output2":{
   binary = "/jteam/output2.exe";
   machine = "highpower";
}

orchestrate "example":{
  tool "input1";
  tool "input2";
  tool "db";
  tool "output1";
  tool "output2"
}
```

Figure 3.4: MIL for the database example using multicast

```
input1()
begin
  /* declare event */
  mh_declare_event("input1_event", "S");

  q = query;
  while ( q is not empty) do
     /* multicast query */;
     mh_multicast("input1_event", "S", q);
     q = next query;
  end while
end input1


output1
begin
  /* register its interest in event */
  mh_rsgmulticast("db_event");

  while (true) do
     /* receive an event into r */;
     mh_bgetmsg("S", event_name, r);
  end while
  write r to output;
end output1


db()
begin
  /* declare event */
  mh_declare_event("db_event", "S");

  /* register its interest in events */
  mh_rgsmulticast("input1_event");
  mh_rgsmulticast("input2_event");

  while (true) do
    /* receive an event  in q */
     mh_bgetmsg("S", event_name, q);

     r = result of executing query q;
     /* multicast the result of the query */;
     mh_multicast("db_event", "S", r);
  end while
end db
```

Figure 3.5: Modules for the database example using multicast

```
main()
begin
   event-list = events generated or multicast by modules in application A
                that application B is interested in receiving;
   register interest in event-list;
   while true
      get events of interest from application A;
      transform_event(event_name)
   end while
end

procedure transform_event(event_name)
begin
         decode the event according to the old event structure;
         coerce the event to the new event structure;
         multicast event with its new event name and new event structure to
         modules in application B;
end
```

Figure 3.6: Algorithm for the coercion module

3.2 Event Mapping

In this section, we describe our facilities for mapping events from one application to another at run time when there are different event names and/or different event structures. Our goal is to be able to transform events transparently so that no changes need to be made to the application modules. First, we describe our techniques for dealing with the basic coercion scenarios in Section 3.2.1. In Section 3.2.2, we demonstrate how to apply our techniques to the integrated VE3D/VE2D example.

To allow transparent event mapping, we create*coercion modules* to implement the interaction between existing applications. Each application has a coercion module that acts as an adaptor. The task of a coercion module is to intercept an event of interest from another application and to transform it to the matching events of the corresponding event name and event structure for its associated application. The coercion modules are implemented in such a way that the transformation is transparent to the application modules.

Coercion modules are generated *automatically* in our approach. Programmers do not have to create coercion modules manually. In this section, we concentrate on explaining what the coercion modules look like and how they transform events. Details of how the coercion modules are generated will be described later in Section 4.2.

Suppose we want to build a composite application from existing applications A and B, and a coercion module named **Coercion_AB** is responsible for transforming events from application A to application B. The algorithm for the coercion module is given in Figure 3.6.

```
#include <stdio.h>
#include <strings.h>
extern transform_A_event1_B_event2();

main(argc, argv)
int argc;
char *argv[];
{
    char event_name[256], tape[256];
    char *event;

    mh_init(&argc, &argv, NULL, NULL);
    mh_rgsmulticast("event1");
    while(1) {
       mh_next_event(tape, event_name, &event);
       if (!strcmp(event_name, "event1"))
          transform_A_event1_B_event2(tape, event);
    }
}
```

Figure 3.7: The `main` routine for the coercion module **Coercion_AB**

In a multicasting environment, modules receive events without knowing where the events come from. Therefore, modules receive and react to events multicast by a coercion module as if they were multicast by an ordinary application module; they are unaware of the existence of a coercion module. In this way, we can enable the interoperation of existing applications while keeping the application modules intact.

3.2.1 Basic Coercion Scenarios

When we map an event from one application to another, it can undergo two possible changes: changes in event names and event structures. Changing an event name is simple, since the name is a string of characters. Changing an event structure is more difficult, since the structure could be any legal data type of the programming language in which the module is written. There are several possible changes which can be made to the event structure: parameters may appear in different order; data types may not exactly match; some data may need to be initialized; some data may be masked out; some data may need to be flattened; and some data may need to be aggregated. In Section 2.2, we discussed how to use the event mapping notation ENimble to express those changes. In this section we describe what the coercion modules would look like when we implement the event mapping. We use the same examples from Section 2.2 to map event *event1* in application A to event *event2* in application B.

A coercion module consists of two parts, a main routine and a transformation routine. Figure 3.7 shows the `main()` routine for a coercion module, which is the same for all six examples. The transformation routine is different for each example. The source code for the transforma-

```
struct A_event1 {
       int      x;
       char     *y;
};
struct B_event2 {
       char      *a;
       int       b;
};
struct A_event1  From_A_event1;
struct B_event2  *To_B_event2;
struct B_event2 *Coerce_A_event1_B_event2(From)
struct A_event1 *From;
{
    struct B_event2 To;
    To.a = From->y;
    To.b = From->x;
    return(&To);
}
void transform_A_event1_B_event2(tape, event)
char *tape;
char *event;
{
   mh_decode_event(tape, event, &(From_A_event1.x),&(From_A_event1.y));
   To_B_event2 = Coerce_A_event1_B_event2(&From_A_event1);
   mh_multicast("event2", "SI",To_B_event2->a,To_B_event2->b);
}
```

Figure 3.8: Transformation routine for `Example 1`

tion routine is straightforward. Let us study the transformation routines for the basic coercion scenarios:

1. Rearrange the order of the parameters

   `Example 1:`

   ```
   map A event1(x:  integer; y:  string)
   to B event2(y;x)
   ```

   The transformation routine is given in Figure 3.8.

2. Initialize data

   `Example 2:`

   ```
   map A event1(x:  integer; y:  integer)
   to B event2(x; y; string('hello'))
   ```

   The transformation routine is given in Figure 3.9.

```
struct A_event1 {
       int       x;
       int       y;
};
struct B_event2 {
       int       a;
       int       b;
       char     *c;
};
struct A_event1  From_A_event1;
struct B_event2  *To_B_event2;
struct B_event2 *Coerce_A_event1_B_event2(From)
struct A_event1 *From;
{
    struct B_event2 To;
    To.a = From->x;
    To.b = From->y;
    To.c = "hello";
    return(&To);
}
void transform_A_event1_B_event2(tape, event)
char *tape;
char *event;
{
   mh_decode_event(tape, event, &(From_A_event1.x),&(From_A_event1.y));
   To_B_event2 = Coerce_A_event1_B_event2(&From_A_event1);
   mh_multicast("event2", "IIS",To_B_event2->a,To_B_event2->b,To_B_event2->c);
}
```

Figure 3.9: Transformation routine for `Example 2`

```
struct A_event1 {
       int       x;
       int       y;
};
struct B_event2 {
       int       a;
};
struct A_event1  From_A_event1;
struct B_event2  *To_B_event2;
struct B_event2 *Coerce_A_event1_B_event2(From)
struct A_event1 *From;
{
    struct B_event2 To;
    To.a = From->y;
    return(&To);
}
void transform_A_event1_B_event2(tape, event)
char *tape;
char *event;
{
   mh_decode_event(tape, event, &(From_A_event1.x),&(From_A_event1.y));
   To_B_event2 = Coerce_A_event1_B_event2(&From_A_event1);
   mh_multicast("event2", "I",To_B_event2->a);
}
```

Figure 3.10: Transformation routine for `Example 3`

3. Mask out data

   `Example 3:`

   ```
   map A event1(x:  integer; y:  integer)
   to B event2(y)
   ```

   The transformation routine is given in Figure 3.10.

4. Coerce data

   `Example 4:`

   ```
   map A event1(x:  integer; y:  integer)
   to B event2(x:  string(y))
   ```

   The transformation routine is given in Figure 3.11.

5. Flatten data

   `Example 5:`

   ```
   map A event1(point:{x:  integer; y:  integer})
   to B event2(point.x; point.y)
   ```

```
struct A_event1 {
       int       x;
       int       y;
};
struct B_event2 {
       int       a;
       char      *b;
};
struct A_event1  From_A_event1;
struct B_event2  *To_B_event2;
struct B_event2 *Coerce_A_event1_B_event2(From)
struct A_event1 *From;
{
    struct B_event2 To;
    To.a = From->x;
    To.b = Int2Str(From->y);
    return(&To);
}
void transform_A_event1_B_event2(tape, event)
char *tape;
char *event;
{
   mh_decode_event(tape, event, &(From_A_event1.x),&(From_A_event1.y));
   To_B_event2 = Coerce_A_event1_B_event2(&From_A_event1);
   mh_multicast("event2", "IS",To_B_event2->a,To_B_event2->b);
}
```

Figure 3.11: Transformation routine for `Example 4`

```
struct Aa {
       int       x;
       int       y;
};
struct A_event1 {
       struct Aa   a;
};
struct B_event2 {
       int       a;
       int       b;
};
struct A_event1  From_A_event1;
struct B_event2  *To_B_event2;
struct B_event2 *Coerce_A_event1_B_event2(From)
struct A_event1 *From;
{
    struct B_event2 To;
    To.a = From->a.x;
    To.b = From->a.y;
     return(&To);
}
void transform_A_event1_B_event2(tape, event)
char *tape;
char *event;
{
    mh_decode_event(tape, event, &(From_A_event1.a));
    To_B_event2 = Coerce_A_event1_B_event2(&From_A_event1);
    mh_multicast("event2", "II",To_B_event2->a,To_B_event2->b);
}
```

Figure 3.12: Transformation routine for `Example 5`

The transformation routine is given in Figure 3.12.

6. Aggregate data

   `Example 6:`

   ```
   map A event1(x:  integer; y:  integer)
   to B event2({x; y})
   ```

   The transformation routine is given in Figure 3.13.

The above examples illustrate the basic coercion scenarios. Coercion modules may deal with combinations of the basic coercion scenarios.

3.2.2 Coercion Modules for VE3D/VE2D

```
struct Fa {
       int       a;
       int       b;
};
struct A_event1 {
       int       x;
       int       y;
};
struct B_event2 {
       struct Fa   a;
};
struct A_event1  From_A_event1;
struct B_event2  *To_B_event2;
struct B_event2 *Coerce_A_event1_B_event2(From)
struct A_event1 *From;
{
    struct B_event2 To;
    To.a.a = From->x;
    To.a.b = From->y;
    return(&To);
}
void transform_A_event1_B_event2(tape, event)
char *tape;
char *event;
{
    mh_decode_event(tape, event, &(From_A_event1.x),&(From_A_event1.y));
    To_B_event2 = Coerce_A_event1_B_event2(&From_A_event1);
    mh_multicast("event2", "{II}",To_B_event2->a);
}
```

Figure 3.13: Transformation routine for `Example 6`

```
/* VE2DVE3D.c */

#include <stdio.h>
extern transform_VE2D_obj_VE3D_ext_obj();
main(argc, argv)
int argc;
char *argv[];
{
   char event_name[256], tape[256];
   char *event;

   mh_init(&argc, &argv, NULL, NULL);
   mh_rgsmulticast("obj");
   while (1) {
      mh_next_event(tape, event_name, &event);
      if (!strcmp(event_name, "obj"))
       transform_VE2D_obj_VE3D_ext_obj(tape, event);
   }
}

  /* VE2D_obj_VE3D_ext_obj.c */

#include <stdio.h>
struct Aa {
   int x;
   int y;
   int z;
};
struct VE2D_obj {
   struct Aa a;
};
struct VE3D_ext_obj {
   char *name;
   int x;
   int y;
   int z;
};
struct VE2D_obj From_VE2D_obj;
struct VE3D_ext_obj *To_VE3D_ext_obj;

struct VE3D_ext_obj *Coerce_VE2D_obj_VE3D_ext_obj(From)
struct VE2D_obj *From;
{
   struct VE3D_ext_obj To;

   To.name = "jim";
   To.x = From->a.x;
   To.y = From->a.y;
   To.z = From->a.z;
   return(&To);
}

void transform_VE2D_obj_VE3D_ext_obj(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VE2D_obj.a));
      To_VE3D_ext_obj = Coerce_VE2D_obj_VE3D_ext_obj(&From_VE2D_obj);
      mh_multicast("ext_obj", "SIII", To_VE3D_ext_obj->name, To_VE3D_ext_obj->x,
                     To_VE3D_ext_obj->y, To_VE3D_ext_obj->z);
}
```

Figure 3.14: Source code for coercion module **Coercion_VE2DVE3D**

```
/* VE3DVE2D.c */
#include <stdio.h>
extern transform_VE3D_obj_VE2D_ext_obj();
extern transform_VE3D_obj_VE2D_ext_obj_xy();
extern transform_VE3D_obj_VE2D_ext_obj_yz();
extern transfrom_VE3D_obj_VE2D_ext_obj_xz();
main(argc, argv)
int argc;
char *argv[];
{
   char event_name[256], tape[256];
   char *event;

   mh_init(&argc, &argv, NULL, NULL);
   mh_rgsmulticast("obj");
   while (1) {
      mh_next_event(tape, event_name, &event);
      if (!strcmp(event_name, "obj"))
       transform_VE3D_obj_VE2D_ext_obj(tape, event);
      if (!strcmp(event_name, "obj"))
       transform_VE3D_obj_VE2D_ext_obj_xy(tape, event);
   }
}

/* VE3D_obj_VE2D_ext_obj.c */

#include <stdio.h>
struct Fa {
   int x;
   int y;
   int z;
};
struct VE3D_obj {
   int x;
   int y;
   int z;
}
struct VE2D_ext_obj {
   char *name;
   struct Fa b;
};
struct VE3D_obj From_VE3D_obj;
struct VE2D_ext_obj *To_VE2D_ext_obj;

struct VE2D_ext_obj *Coerce_VE3D_obj_VE2D_ext_obj(From)
struct VE3D_obj *From;
{
   struct VE2D_ext_obj To;

   To.name = "chen";
   To.b.x = From->x;
   To.b.y = From->y;
   To.b.z = From->z;
   return(&To);
}

void transform_VE3D_obj_VE2D_ext_obj(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VE3D_obj.x),
                     &(From_VE3D_obj.y), &(From_VE3D_obj.z);
      To_VE2D_ext_obj = Coerce_VE3D_obj_VE2D_ext_obj(&From_VE3D_obj);
      mh_multicast("ext_obj", "S{III}", To_VE2D_ext_obj->name,
                     To_VE2D_ext_obj->b);
}
```

Figure 3.15: Source code for coercion module **Coercion_VE3DVE2D** (part 1 of 4)

```
/* VE3D_obj_VE2D_ext_obj_xy.c */

#include <stdio.h>
struct Fa {
   int x;
   int y;
};
struct VE3D_obj {
   int x;
   int y;
   int z;
}
struct VE2D_ext_obj_xy {
   char *name;
   struct Fa b;
};
struct VE3D_obj From_VE3D_obj;
struct VE2D_ext_obj_xy *To_VE2D_ext_obj_xy;

struct VE2D_ext_obj_xy *Coerce_VE3D_obj_VE2D_ext_obj_xy(From)
struct VE3D_obj *From;
{
   struct VE2D_ext_obj_xy To;

   To.name = "chen";
   To.b.x = From->x;
   To.b.y = From->y;
   return(&To);
}

void transform_VE3D_obj_VE2D_ext_obj_xy(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VE3D_obj.x),
                      &(From_VE3D_obj.y), &(From_VE3D_obj.z);
      To_VE2D_ext_obj_xy = Coerce_VE3D_obj_VE2D_ext_obj_xy(&From_VE3D_obj);
      mh_multicast("ext_obj_xy", "S{II}", To_VE2D_ext_obj->name,
                      To_VE2D_ext_obj->b);
}
```

Figure 3.16: Source code for coercion module **Coercion_VE3DVE2D** (part 2 of 4)

```
/* VE3D_obj_VE2D_ext_obj_yz.c */

#include <stdio.h>
struct Fa {
   int y;
   int z;
};
struct VE3D_obj {
   int x;
   int y;
   int z;
}
struct VE2D_ext_obj_yz {
   char *name;
   struct Fa b;
};
struct VE3D_obj From_VE3D_obj;
struct VE2D_ext_obj_yz *To_VE2D_ext_obj_yz;

struct VE2D_ext_obj_yz *Coerce_VE3D_obj_VE2D_ext_obj_yz(From)
struct VE3D_obj *From;
{
   struct VE2D_ext_obj_xy To;

   To.name = "chen";
   To.b.y = From->y;
   To.b.z = From->z;
   return(&To);
}

void transform_VE3D_obj_VE2D_ext_obj_yz(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VE3D_obj.x), &(From_VE3D_obj.y),
                      &(From_VE3D_obj.z);
      To_VE2D_ext_obj_yz = Coerce_VE3D_obj_VE2D_ext_obj_yz(&From_VE3D_obj);
      mh_multicast("ext_obj_yz", "S{II}", To_VE2D_ext_obj->name,
                      To_VE2D_ext_obj->b);
}
```

Figure 3.17: Source code for coercion module **Coercion_VE3DVE2D** (part 3 of 4)

```
/* VE3D_obj_VE2D_ext_obj_xz.c */

#include <stdio.h>
struct Fa {
   int x;
   int z;
};
struct VE3D_obj {
   int x;
   int y;
   int z;
}
struct VE2D_ext_obj_xz {
   char *name;
   struct Fa b;
};
struct VE3D_obj From_VE3D_obj;
struct VE2D_ext_obj_yz *To_VE2D_ext_obj_xz;

struct VE2D_ext_obj_xz *Coerce_VE3D_obj_VE2D_ext_obj_xz(From)
struct VE3D_obj *From;
{
   struct VE2D_ext_obj_xy To;

   To.name = "chen";
   To.b.y = From->y;
   To.b.z = From->z;
   return(&To);
}

void transform_VE3D_obj_VE2D_ext_obj_xz(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VE3D_obj.x), &(From_VE3D_obj.y),
                      &(From_VE3D_obj.z);
      To_VE2D_ext_obj_xz = Coerce_VE3D_obj_VE2D_ext_obj_xz(&From_VE3D_obj);
      mh_multicast("ext_obj_xz", "S{II}", To_VE2D_ext_obj->name,
                      To_VE2D_ext_obj->b);
}
```

Figure 3.18: Source code for coercion module **Coercion_VE3DVE2D** (part 4 of 4)

We use the integrated VE3D/VE2D example to illustrate how the coercion modules work. In VE3D/VE2D, we need two coercion modules, **Coercion_VE3DVE2D** for transforming events from VE3D to VE2D, and **Coercion_VE2DVE3D** for transforming events from VE2D to VE3D. **Coercion_VE3DVE2D** takes event *obj* from VE3D and transforms it to four events *ext_obj*, *ext_obj_xy*, *ext_obj_yz*, and *ext_obj_xz* in VE2D. **Coercion_VE2DVE3D** takes event *obj* from VE2D and transforms it to event *ext_obj* in VE3D.

Figure 3.14 gives the source code for the coercion module **Coercion_VE2DVE3D**, which is generated automatically. The coercion module registers its interest in event *obj*. Since it is a coercion module, it receives only events of interest multicast from VE2D and not from VE3D. After receiving an event, it calls procedure

```
transform_VE2D_obj_VE3D_ext_obj(tape,event);
```

to transform the old event structure to the new one. During the transformation, it first decodes the event according to its event structure

```
mh_decode_event(tape, event, &(Actuals_VE2D_obj.a));
```

it then transforms the old event structure to a new one

```
Formals_VE3D_ext_obj= Coerce_VE2D_obj_VE3D_ext_obj(&Actuals_VE2D_obj);
```

and finally it re-multicasts the event with the new event name and new event structure to application modules in VE3D by using

```
mh_multicast("ext_obj", "SIII", ...);
```

Similarly, the other coercion module **Coercion_VE3DVE2D** is responsible for transforming events from VE3D to VE2D. Its automatically generated source code is given in Figure 3.15, Figure 3.16, Figure 3.17, and Figure 3.18.

3.3 Summary

Our execution environment supports event-based distributed applications, whether they are being constructed from scratch or from the synthesis of existing applications. In our execution environment, communications between modules can function normally across heterogeneous hosts. Using multicast primitives, modules can declare events, define event structures, register interest in events, multicast events, and receive events of interest. For interoperation between existing applications, our execution environment uses coercion modules to transform events at run time. The event transformation is transparent to the application modules.

Chapter 4

Integration

In this chapter, we discuss how to integrate event-based distributed applications given event interaction properties. Section 4.1 focuses on how to construct an event-based application from scratch. We describe the technique of automatic generation of interface software or stubs for use in an event-based application given module interaction specifications for application modules. In Section 4.2, we concentrate on how to use composite interaction specifications to integrate related but separately developed event-based distributed applications. In Section 4.3, we discuss advantages and limitations of our integration approach.

4.1 Interface Software Generation

As in other stub generation approaches, programmers using our approach provide the source code for the modules and an abstract description of them. The abstract description includes the desired geometry and module interaction properties of an application specified in the POLYLITH MIL. Given module interaction specifications for an application, we generate interface software or stubs for use in module interactions of an application. These stubs are customized to both the module interaction requirements and the POLYLITH execution environment.

Figure 4.1 shows the integration process of building an event-based distributed application from scratch. Given module interaction specifications for an application (`desc.cl`), the `preprocessor` tool extracts and preprocesses information from `desc.cl` and writes the information to the `desc.bus` file, which is the input to both the software bus and the stub generator POLYSTUB. POLYSTUB then generates a Makefile containing the configuration commands needed to build both the application and the customized interface software that maps the specification to the execution environment.

When compiled with or otherwise included in the module implementations, the stubs act as an intermediary between the module and the rest of the application. The stubs include appropriate

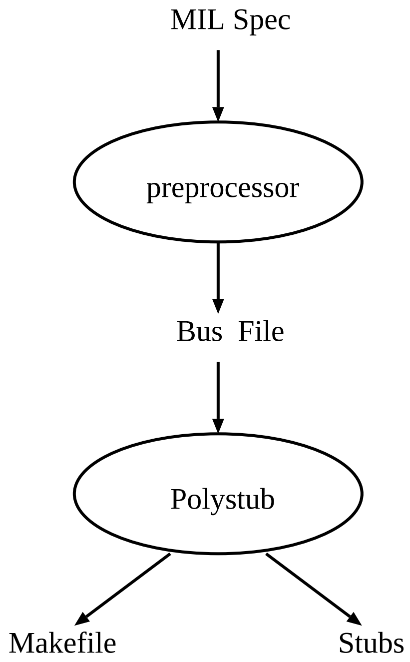

Figure 4.1: Integration process for building an application from scratch

```
LITH = /bugs/chenchen/SuperBus

all:    Display Simulator Controller Log

Display:  display_stubs.c display_user.c
         cc display_stubs.c display_user.c -L${LITH} -lith -o display

Simulator:  simulator_stubs.c simulator_user.c
         cc simulator_stubs.c simulator_user.c -L${LITH} -lith -o simulator

Controller:  controller_stubs.c controller_user.c
         cc controller_stubs.c controller_user.c -L${LITH} -lith -o controller

Log:  log_stubs.c log_user.c
         cc log_stubs.c log_user.c -L${LITH} -lith -o log

install:
        rcp display thumper:/thumper/chenchen/.
        rcp controller harvey:/harvey/chenchen/.
        rcp log xring:/xring/chenchen/.
```

Figure 4.2: Makefile for the VE3D application

calls for the underlying system to initialize modules for execution, to declare events, to define their event structures, to register interest in events, to receive events of interest, to decode events, and to invoke the corresponding procedures.

The interface software and user-supplied modules containing procedures associated with events are compiled to create an executable for each module on its destination machine. After the installation, the enhanced POLYLITH uses the preprocessed module interaction specifications to invoke processes and is responsible for coercing data representation, synchronization, marshaling of data, and communication between modules at run time.

We use VE3D example described in Chapter 1 to illustrate the integration process of building an application from scratch. The module interaction specification for VE3D given in Figure 2.1 (`desc.cl`) is preprocessed using command

`preprocess desc.cl`

This causes the information in `desc.cl` to be extracted, preprocessed and rewritten in file `desc.bus`. This file contains detailed information about each application module, including the interfaces of the module and the attributes associated with the module, as well as information concerning the event-based interactions. The `desc.bus` file is the input for the software bus and the stub generator. Thus, based on `desc.bus`, the POLYSTUB can generate both the Makefile and the customized interface software via command

`polystub desc.bus` .

Figure 4.2 gives the Makefile for VE3D application. The generated Makefile contains configuration commands for compiling the interface software and user-supplied source code and for linking

```
/* Display.c */

#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
   char *event_name, *event_buf;
   int x, y, z;
   char *name;

   mh_init(&argc, &argv, NULL, NULL);
   mh_rgsmulticast("obj");
   mh_rgsmulticast("ext_obj");

   while (TRUE) {
      mh_next_event(event_name, event_buf);
      if (strcmp(event_name, "obj")==0) {
          mh_decode_event(event_buf,&x,&y,&z);
          update_viewpoint(x, y, z);
      }
      else if (strcmp(event_name, "ext_obj"==0) {
          mh_decode_event(event_buf, name,&x, &y, &z);
          update_ext_obj(name, x, y, z);
   }
}
```

Figure 4.3: Interface software for **Display** of VE3D (`display_stubs.c`)

to POLYLITH's library to create an executable for each module. For example, the interface software for module **Display** produced by POLYSTUB is in file `display_stubs.c`. This file and the user-supplied source code (`display_user.c` containing procedures *update_viewpoint* and *update_ext_obj*) are input to the C compiler and linker to produce an executable `display`. Once created, the executable is installed at the desired location (`/thumper/chenchen/display`) on its destination machine.

Figure 4.3 shows the interface software for module **Display** of VE3D application. It is tailored to both module interaction requirements and the POLYLITH execution environment. After initialization (*mh_init*), **Display** calls *mh_rgsmulticast* to register its interest in events *obj* and *ext_obj* and calls *mh_next_event* to receive an event. It decodes the event (*mh_decode_event*) according to the event structure and then invokes the corresponding procedure.

Using our integration approach, users are not required to transform the modules manually, to mix the code for event-based communication with code for the application's functionality. Instead, users give the module interaction specifications for an application and provide the basic module source code. Based on the abstract aggregate, customized interface software will be generated automatically. The interface software is tailored to the event-based interaction properties of the application and contains system calls for event-based interaction in the underlying run time environment. In this way, designers can define complex event-based interactions abstractly, thus integrating and experimenting with them more easily.

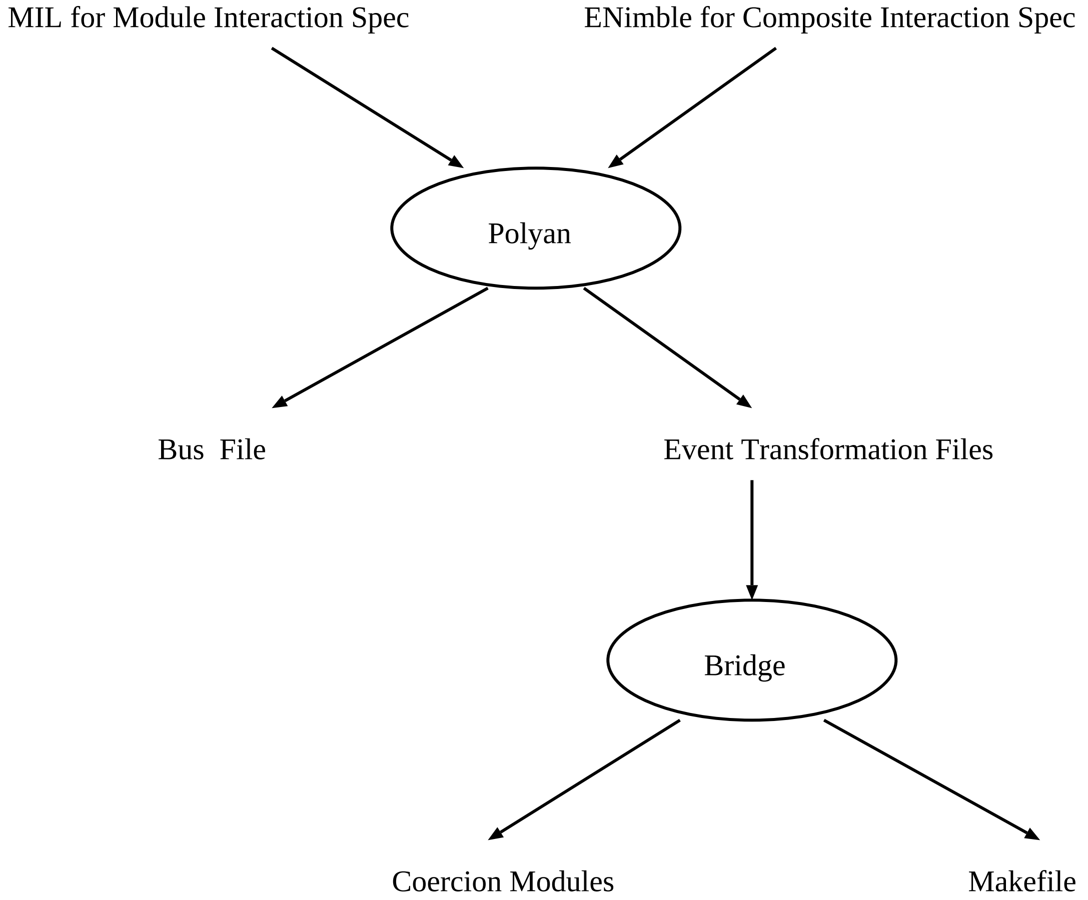

Figure 4.4: Composite integration process

4.2 Composite Interaction Integration

In this section, we discuss how to use composite interaction specifications to guide the composite application integration process (Figure 4.4). Given module interaction specifications for both the original applications and a composite interaction specification, we have developed a tool called POLYAN to analyze these interaction specifications. The output of this tool is a bus file and some event transformation files.

The bus file contains both a list of modules that make up the composite application and also attributes associated with those modules. This preprocessed information is written in a format understood by the software bus, which will use the information written in the bus file to start up the correct application modules and coercion modules of the composite application.

An event transformation file contains a set of event transformation rules for transforming events from one application to another. We have developed a tool called BRIDGE which takes an event transformation file as input and automatically generates the coercion module and the Makefile for building the coercion module. The BRIDGE tool is essential to the composite integration process. The execution of BRIDGE can be broken down into several steps:

1. Split event transformation files (`.enimble`) into separate files (`.data`), each containing only one event transformation rule;

2. Invoke `enimble` to perform an equivalence checking between the target event structure and the range of the event map;

3. Create the operational specification (`.p` files) for the event transformation rule;

4. Invoke `eventgen` to translate the operational specification to the source code for the coercion module in the target programming language (including the main routine and the transformation routine);

5. Create the Makefile (`.makefile`) for building the coercion module.

We use the VE3D/VE2D example described in Chapter 1 to illustrate the integration process of building an application from two related but separately developed applications. Figure 4.5 shows the production graph for the VE3D/VE2D example. The module interaction specifications for the VE3D (`ve3d.cl` shown in Figure 2.1) and VE2D (`ve2d.cl` shown in Figure 2.2), along with the composite interaction specification for the VE3D/VE2D (`comp` shown in Figure 2.3), are analyzed after issuing the command

```
polyan ve3d.cl ve2d.cl comp desc.bus
```

This command causes the information in these files to be analyzed, extracted, and rewritten in the bus file `desc.bus` in a format understood by the bus. The bus file contains information concerning eight application modules and two coercion modules, **Coercion_VE3DVE2D** and **Coercion_VE2DVE3D**. The command also outputs event transformation files `VE3DVE2D.enimble` and `VE2DVE3D.enimble`. We then use commands

```
bridge VE3DVE2D.enimble
bridge VE2DVE3D.enimble
```

to generate the source code for two coercion modules, **Coercion_VE3DVE2D** (given in Figure 3.15 and Figure 3.16) and **Coercion_VE2DVE3D** (given in Figure 3.14), and their makefiles, `VE3DVE2D.makefile` (given in Figure 4.6) and `VE2DVE3D.makefile` (given in Figure 4.7). The Makefile contains configuration commands needed to build the coercion module. After making the coercion modules, programmers invoke the software bus to start up the composite application using the command

```
sbus -b desc.bus
```

The run-time option `-b` tells the bus that this is a composite application built from existing applications. When all processes for modules are invoked, the software bus will handle event mapping automatically at run time.

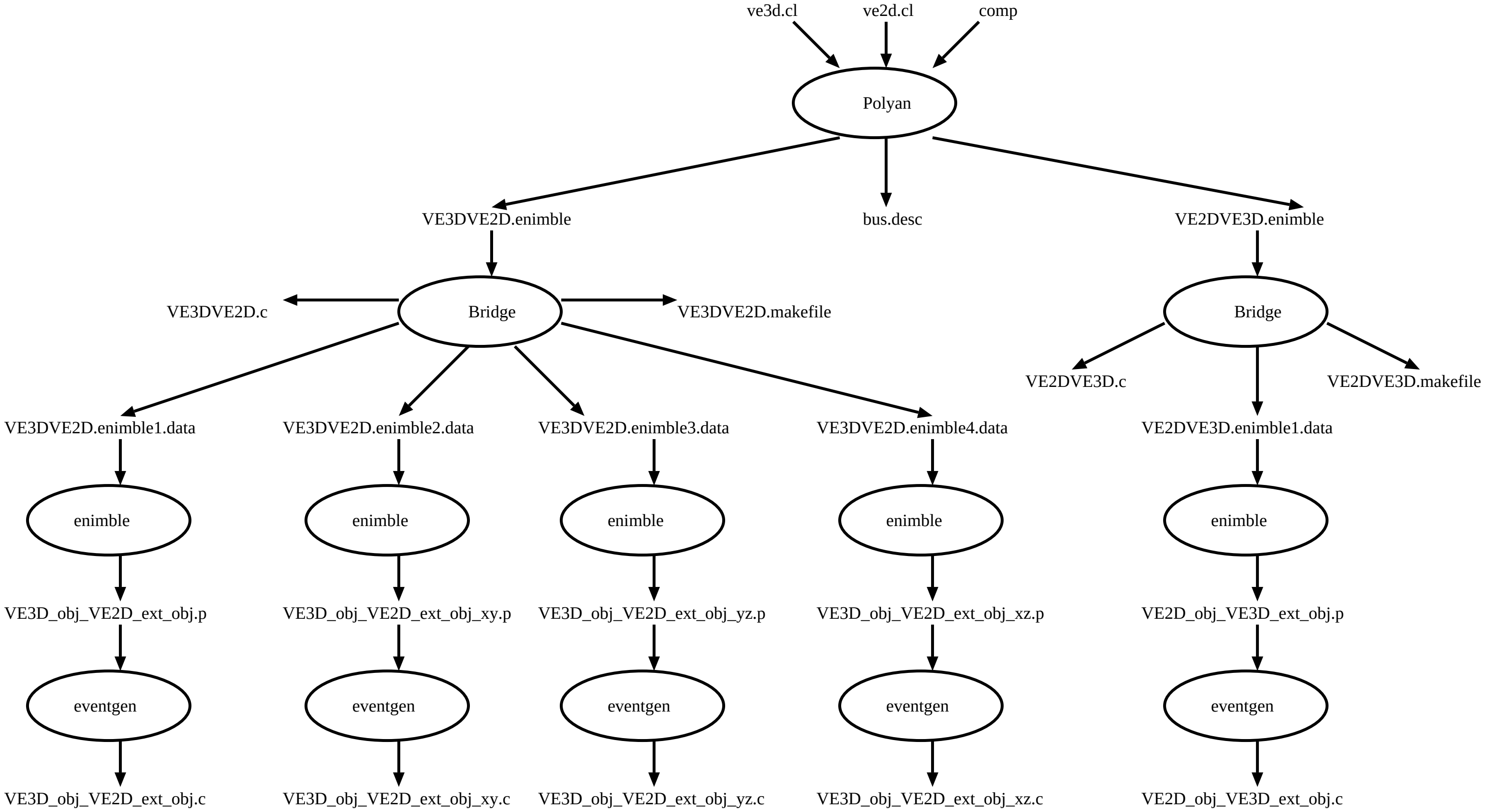

Figure 4.5: Production graph for the integrated virtual environment example

```
LITH=/bugs/chenchen/SuperBus
all: Coercion_VE3DVE2D

Coercion_VE3DVE2D: VE3DVE2D.o VE3D_obj_VE2D_ext_obj.o
   cc -o coercion_VE3DVE2D VE3DVE2D.o VE3D_obj_VE2D_ext_obj.o \
         VE3D_obj_VE2D_ext_obj_xy.o VE3D_obj_VE2D_ext_obj_yz.o \
         VE3D_obj_VE2D_ext_obj_xz.o -L${LITH} -lith

VE3DVE2D.o: VE3DVE2D.c
   cc -c -L${LITH} -lith VE3DVE2D.c

VE3D_obj_VE2D_ext_obj.o: VE3D_obj_VE2D_ext_obj.c
   cc -c -L${LITH} -lith VE3D_obj_VE2D_ext_obj.c

VE3D_obj_VE2D_ext_obj_xy.o: VE3D_obj_VE2D_ext_obj_xy.c
   cc -c -L${LITH} -lith VE3D_obj_VE2D_ext_obj_xy.c

VE3D_obj_VE2D_ext_obj_yz.o: VE3D_obj_VE2D_ext_obj_yz.c
   cc -c -L${LITH} -lith VE3D_obj_VE2D_ext_obj_yz.c

VE3D_obj_VE2D_ext_obj_xz.o: VE3D_obj_VE2D_ext_obj_xz.c
   cc -c -L${LITH} -lith VE3D_obj_VE2D_ext_obj_xz.c
```

Figure 4.6: Makefile for coercion module **Coercion_VE3DVE2D**

```
LITH=/bugs/chenchen/SuperBus
all: Coercion_VE2DVE3D

Coercion_VE2DVE3D: VE2DVE3D.o VE2D_obj_VE3D_ext_obj.o
   cc -o coercion_VE2DVE3D VE2DVE3D.o VE2D_obj_VE3D_ext_obj.o -L${LITH} -lith

VE2DVE3D.o: VE2DVE3D.c
   cc -c -L${LITH} -lith VE2DVE3D.c

VE2D_obj_VE3D_ext_obj.o: VE2D_obj_VE3D_ext_obj.c
   cc -c -L${LITH} -lith VE2D_obj_VE3D_ext_obj.c
```

Figure 4.7: Makefile for coercion module **Coercion_VE2DVE3D**

Using our integration approach, users are not required to manually modify the application structure or change the module implementation. Given the composite interaction specification, interoperation between two related but separately developed applications is achieved not only without the cost of developing a third entire system, but also without making any changes to module implementations.

4.3 Discussion

There are numerous advantages of our integration approach. Programmers are not required to manually adapt the existing applications for reuse in a new application. If programmers were to introduce extra interfacing code into the existing application modules in order to have the event names and event structures correspond, it would make the application modules more complex and more costly to maintain. In our approach, such adaptations are done by the coercion modules generated automatically, and the application modules remain intact. This approach allows the application source to remain simpler and easier to maintain.

If programmers were to adapt the application modules manually, they would have to modify the application structure, the modules names, the event names, and the event structures. There is a good chance that they would introduce errors in the applications. Our approach allows programmers to define the interactions at an abstract level and to automatically generate the valid software modules that implement the programmer-defined interactions. Thus, our approach reduces the opportunity for introducing errors and makes it easier to construct an application from existing systems.

Since interactions are specified separately from the application modules, programmers can build applications with different interactions from the same existing applications. For example, in the integrated VE3D/VE2D, we can remove some modules, like **Log** and **Display_XY**, from the composite application by simply changing the interaction specification. If we want to display the ZX plane instead of the XZ plane, we can rearrange the order of the parameters in the specification and have the coercion module generated automatically. Our approach allows the software modules to be reused in a broader range of applications.

Finally, our approach greatly facilitates software prototyping. Prototyping is an experimental activity intended to expose properties or design alternatives to developers before they make critical decisions. When prototyping, developers need to quickly build a prototype apparatus at a relatively low cost. If the cost of building a prototype is high, then the benefits of prototyping remain unrealized. Our approach not only reuses software modules to keep the cost down, but also automates their adaptation. Using our approach, developers can easily construct a prototype from existing applications and experiment with prototypes of different interactions. Thus developers spend less time adapting software modules and more time studying the prototyping apparatus behavior.

Although our current specification languages and integration tools limit us to interconnecting existing applications built in POLYLITH, our integration approach can be applied within any software bus environment. Indeed, by leveraging the power of the software bus model, it is possible to interconnect components from several different integration frameworks; but whether this is possible to do across all frameworks remains an open question. For example, we can not automatically support the interoperation between two existing applications, where one is built in POLYLITH and the other is built in Field.

It is difficult to facilitate interoperation among different integration mechanisms, because as yet there is no consistent model for describing existing event-based integration framework. Barrett *et al.* at University of Massachusetts attempt to address this problem by specifying a generic, event-based integration framework [1]. They provide an abstract data type based model for discussing and comparing event-based integration approaches. They intend to explore issues of interoperability and examine the extent to which the model can be used directly to support interoperability.

Although so far the examples used in this thesis show only how to integrate two existing applications, our approach can be applied to the integration of more than two applications. An example of interoperating among three applications will be shown in Chapter 5. When more applications are involved, the interactions among them become more complex, the amount of manual adaptation requires more effort from programmers, and the ability to define the interactions at a high level and then automatically to generate valid coercion modules becomes more important and desirable.

Chapter 5

Examples

In this chapter, we demonstrate the use of our approach with more complex examples. We add another related but separately developed virtual environment application, called VEAR, to the two existing virtual environment applications and experiment with interactions among them. The experiment demonstrates that we can achieve different interoperations by defining the interactions at the abstract level. It also gives us a rough idea of the amount of code generated in our approach. In this chapter, we first describe VEAR and explain how it is related to VE3D and VE2D. After describing different possible interactions between VE3D, VE2D, and VEAR, we perform a case study on one particular interaction and discuss how to achieve different interactions by changing the specifications.

Figure 5.1 shows the VEAR application that simulates a 3D virtual environment where a user can walk through a building. This application consists of four modules, **Controller**, **Display**, **Simulator**, and **Log**, which are distributed on different host machines. Module **Controller** is used to enter the current user's position. Module **Simulator** simulates the position of users of other walkthroughs. Information on the current user and the users of other walkthroughs is fed to module **Display**, which displays users of other walkthroughs as objects in the current world. Module **Log** records both the current user's position and other users' positions into a log file.

The event structures of *obj* and *ext_obj* in the VEAR are different from those of VE3D and VE2D. For example, event *obj* in VEAR is defined to be an array of three integers:

```
obj (point:integer[3])
```

and event *ext_obj* is defined to be a string followed by an array of three integers:

```
ext_obj (name:string; point:integer[3])
```

There are numerous ways in which the three applications can interact. We study the possible interactions among them and discuss how to capture the interaction in the specifications. First, we study the possible interactions at the configuration level or application level. If an application desires interaction with other applications, it is apparently interested in events generated from

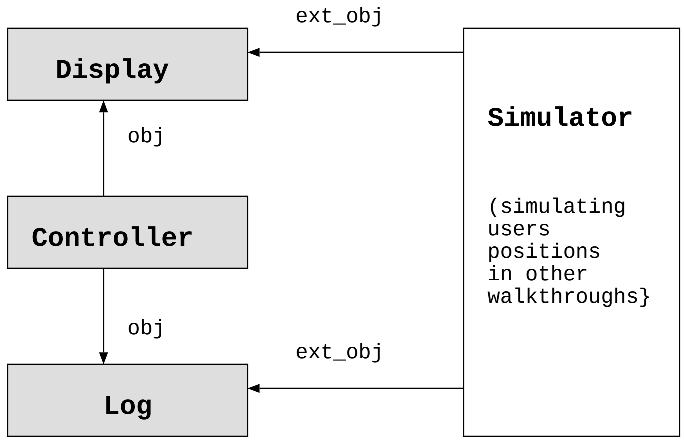

Figure 5.1: Application structure for VEAR

```
module "Display":{
  binary = "/thumper/chenchen/display";
  machine = "thumper.cs.umd.edu";
  file = "display_user.c";
  when obj => update_viewpoint;
  when ext_obj => update_ext_obj;
}

module "Simulator":{
  binary = "/bugs/chenchen/simulator";
  machine = "bugs.cs.umd.edu";
  file = "simulator_user.c";
  declare  ext_obj(name: string; point:integer[3]);
}


module "Controller":{
  binary = "/harvey/chenchen/controller";
  machine = "harvey.cs.umd.edu";
  file = "controller_user.c";
  declare obj (point:integer[3]);
}


module "Log":{
  binary = "/xring/chenchen/log";
  machine = "xring.cs.umd.edu";
  file = "log_user.c";
  when obj => log_obj;
  when ext_obj => log_ext_obj;
}

orchestrate "VEAR":{
  tool "Display";
  tool "Simulator";
  tool "Controller";
  tool "Log";
}
```

Figure 5.2: Module interaction requirements for the VEAR using an enhanced POLYLITH MIL

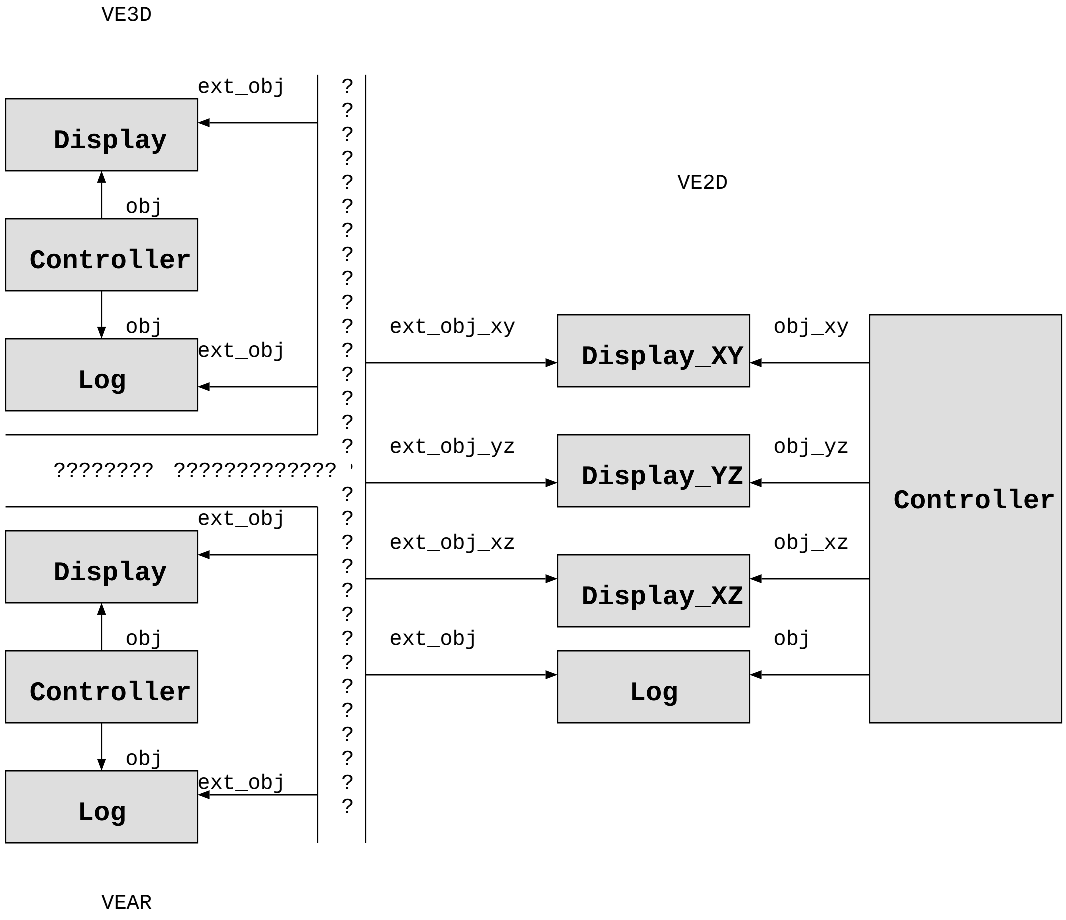

Figure 5.3: Application structure for the VE3D/VE2D/VEAR

other applications. When considering interoperation at the configuration-level, we can study each application's interest in events. In the virtual environment examples, if an application is interested in events from other applications, it means the application wants to display users of other walkthroughs as objects in its world. An application can select which users of other walkthroughs to display. For example, if VE3D is interested in VE2D and VEAR, it will display users of VE2D and VEAR as objects. VE2D may be only interested in VEAR but not VE3D, so it will only display users of VEAR as objects in VE2D. VEAR may be interested in neither of them. When integrating three applications, there are 57 possible interactions at the configuration level. [1]

Next, we go down one level and study the possible interactions at the module level. Because we can tailor an application by removing some redundant modules, we can achieve a greater variety of possible interactions. For example, we can delete the simulation modules, log modules, and/or some display modules in some of the virtual environment applications.

At this point, given a fixed configuration at both the application level and module level, we can still achieve more interactions by varying the event transformation rules. A different event mapping implies a different interaction. For example, if we want to display the ZX plane instead of XZ plane, we can rearrange the order of the coordinates in the specification.

In the remainder of this chapter, we will perform a case study of one particular interaction among the three applications. We first give an informal description of how we want the applications to interact in Section 5.1. Then we describe how we capture the intended interaction using the specification language in Section 5.2. After explaining the integration process in Section 5.3, we evaluate in Section 5.4 the amount of code generated by our approach and discuss how to change the specification to achieve different interactions.

5.1 Case Study: VE3D/VE2D/VEAR

Version 1

In this case study, we want to integrate VE3D, VE2D, and VEAR. We let three users control their own positions and display all the other users as objects in their respective worlds. There are several issues we need to consider before the specification is created:

- **Application Structure**

  The composite application will consist of eleven modules as shown in Figure 5.3. Modules **Simulator** from VE3D, VE2D, and VEAR are no longer needed in the composite application.

[1] Consider the configuration of the three applications as a directed graph $G = (V, E)$, where V consists of three vertices (representing applications) and E is a binary relation on V. E consists of up to six possible edges. By varing the edges in E, we have 57 ($C_6^2 + C_6^3 + C_6^4 + C_6^5 + C_6^6$) combinations.

- **Module Names**

  In all three applications, we have modules **Controller** and **Log**. In both VE3D and VEAR, we have module **Display**. When there is a conflict between module names, we need to differentiate one module from the other.

- **Event Names**

  Since the three applications have been fabricated separately, they have different event naming systems. We need to associate events of interest. For example, all applications have events *obj* and *ext_obj*. In VE3D, event *obj* represents the current user's position. It corresponds to *ext_obj* (not *obj*) in VE2D and VEAR and represents the other user's position. So event *obj* in VE3D should be associated with with event *ext_obj* in VE2D and VEAR. Similarly, event *obj* in VEAR should be associated with event *ext_obj* in VE3D and VE2D; event *obj* in VE2D should be associated with event *ext_obj* in VE3D and VEAR.

- **Event Structures**

  The event structures of associated events do not exactly match. For example, event *obj* in VE3D is defined to be three integers:

  `obj (x:integer; y:integer; z:integer)`

  while in VEAR, its associated event *ext_obj* is defined to be a string followed by an array of three integers:

  `ext_obj (name:string; point:integer[3])`

  and its matching event *ext_obj* in VE2D is defined to be a string followed by a record consisting of three integers:

  `ext_obj (name:string; {x:integer; y:integer; z:integer})`

  Not only are the applications' event structures totally different, but the meaning of each coordinate is also different. The x, y, z coordinates in VE3D correspond to the last, second, and first element of the array in VEAR, respectively. We need to transform the event structure from one application to another with projection of the right parameters.

- **Event Generation**

  When some modules from the original applications are removed from the composite application, there is an effect on events generated or multicast by those modules. For example, in VE2D, events *ext_obj_xy*, *ext_obj_yz* and *ext_obj_xz* are only generated by module **Simulator**. When module **Simulator** is removed from the composite application, we need to figure out whether we still need those missing events or not. If there are needed, we must decide which module should generate the missing events and how they should be generated.

5.2 Specification

Figure 5.4 gives the composite interaction specification for the integrated VE3D/VE2D/VEAR.

```
composite "VE3D/VE2D/VEAR":{
  application VE3D;
  application VE2D;
  application VEAR;

  exclude Simulator from VE3D;
  exclude Simulator from VE2D;
  exclude Simulator from VEAR;

  name Controller from VE3D Controller_3D;
  name Log from VE3D Log_3D;
  name Controller from VE2D Controller_2D;
  name Log from VE2D Log_2D;
  name Display from VEAR Display_AR;
  name Controller from VEAR Controller_AR;
  name Log from VEAR Log_AR;

  map VE3D obj(x: integer; y: integer; z: integer)
  to VE2D ext_obj(string('chen'); {x; y; z})
  to VE2D ext_obj_xy(string('chen'); {x; y})
  to VE2D ext_obj_yz(string('chen'); {y; z})
  to VE2D ext_obj_xz(string('chen'); {x; z})
  to VEAR ext_obj(string('chen'); [z, y, x]);

  map VE2D obj({x:integer; y: integer; z: integer})
  to VE3D ext_obj(string('jim'); a.x; a.y; a.z)
  to VEAR ext_obj(string('jim'); [a.z; a.y; a.x]);

  map VEAR obj(point: integer[3])
  to VE3D ext_obj(string('liz'); point[2]; point[1]; point[0])
  to VE2D ext_obj(string('liz'); {point[2]; point[1]; point[0]})
  to VE2D ext_obj_xy(string('liz'); {point[2]; point[1]})
  to VE2D ext_obj_yz(string('liz'); {point[1]; point[0]})
  to VE2D ext_obj_xz(string('liz'); {point[2]; point[0]})
}
```

Figure 5.4: Composite interaction specification for the VE3D/VE2D/VEAR application using ENimble (version 1)

The clause

```
application VE3D;
application VE2D;
application VEAR;
```

enumerates three applications from which the composite application is built.

The clause

```
exclude Simulator from VE3D;
exclude Simulator from VE2D;
exclude Simulator from VEAR;
```

excludes three **Simulator** modules from the composite application.

The clause

```
name Controller from VE3D Controller_3D;
name Log from VE3D Log_3D;
name Controller from VE2D Controller_2D;
name Log from VE2D Log_2D;
name Display from VEAR Display_AR;
name Controller from VEAR Controller_AR;
name Log from VEAR Log_AR
```

solves name conflicts among modules from different applications.

The clause

```
map VE3D obj(x: integer; y: integer; z: integer)
to VE2D ext_obj(string('chen'); {x; y; z})
to VE2D ext_obj_xy(string('chen'); {x; y})
to VE2D ext_obj_yz(string('chen'); {y; z})
to VE2D ext_obj_xz(string('chen'); {x; z})
to VEAR ext_obj(string('chen'); [z; y; x])
```

maps event *obj* in VE3D to five events:

1. Event *ext_obj* in VE2D consists of a string initialized to `'chen'`, followed by a record consisting of all three integers of event *obj*. It aggregates the three integers.

2. Event *ext_obj_xy* in VE2D consists of a string initialized to `'chen'`, followed by a record consisting of the first two integers of event *obj*. The parameter `z` is masked out.
3. Event *ext_obj_yz* in VE2D consists of a string initialized to `'chen'`, followed by a record consisting of the last two integers of event *obj*. The parameter `x` is masked out.
4. Event *ext_obj_xz* in VE2D consists of a string initialized to `'chen'`, followed by a record consisting of the first and the third integers of event *obj*. The parameter `y` is masked out.
5. Event *ext_obj* in VEAR consists of a string initialized to `'chen'`, followed by an array of three integers of event *obj*. It constructs an array and rearranges the order of the three integers.

The clause

```
map VE2D obj({x:  integer; y:  integer; z:  integer})
to VE3D ext_obj(string('jim'); a.x; a.y; a.z)
to VEAR ext_obj(string('jim'); [a.z; a.y; a.x])
```

maps event *obj* in VE2D to two events:

1. Event *ext_obj* in VE3D consists of a string initialized to `'jim'`, followed by three integers of event *obj*. It flattens the record structure.
2. Event *ext_obj* in VEAR consists of a string initialized to `'jim'`, followed by an array of three integers of event *obj*. It flattens the record structure, constructs an array, and rearranges the order of the coordinates.

The clause

```
map VEAR obj(point:  integer[3])
to VE3D ext_obj(string('liz'); point[2]; point[1]; point[0])
to VE2D ext_obj(string('liz'); {point[2]; point[1]; point[0]})
to VE2D ext_obj_xy(string('liz'); {point[2]; point[1]})
to VE2D ext_obj_yz(string('liz'); {point[1]; point[0]})
to VE2D ext_obj_xz(string('liz'); {point[2]; point[0]})
```

maps event *obj* in VEAR to five events:

1. Event *ext_obj* in VE3D consists of a string initialized to `'liz'`, followed by three integers of event *obj*. It flattens the array structure and rearranges the order of the three integers.

2. Event *ext_obj* in VE2D consists of a string initialized to `'liz'`, followed by a record consisting of three integers of event *obj*. It flattens the array structure, constructs a record structure, and rearranges the order of the three integers.

3. Event *ext_obj_xy* in VE2D consists of a string initialized to `'liz'`, followed by a record consisting of the third and second elements of event *obj*. It flattens the array structure, constructs a record structure, and rearranges the order of the two integers. The first element of the array `point[0]` is masked out.

4. Event *ext_obj_yz* in VE2D consists of a string initialized to `'liz'`, followed by a record consisting of the second and the first elements of event *obj*. It flattens the array structure, constructs a record structure, and rearranges the order of the two integers. The third element of the array `point[2]` is masked out.

5. Event *ext_obj_xz* in VE2D consists of a string initialized to `'liz'`, followed by a record consisting of the third and first elements of event *obj*. It flattens the array structure, constructs a record structure, and rearranges the order of the two integers. The second element of the array `point[1]` is masked out.

5.3 Integration

Figures C.1 through C.7 show the production graph for the VE3D/VE2D/VEAR example. The module interaction specifications for VE3D (`ve3d.cl` in Figure 2.1), VE2D (`ve2d.cl` in Figure 2.2), and VEAR (`vear.cl` in Figure 5.2), along with the composite interaction specifications for the VE3D/VE2D/VEAR (`comp` in Figure 5.4) are analyzed via command

```
polyan ve3d.cl ve2d.cl vear.cl comp desc.bus
```

This command causes the information in these files to be analyzed, extracted, and rewritten in the bus file `desc.bus` in a format understood by the bus. The bus file contains information about eleven application modules and six coercion modules. The command also produces six event transformation files:

```
VE3DVE2D.enimble,
VE3DVEAR.enimble,
VE2DVE3D.enimble,
VE2DVEAR.enimble,
VEARVE3D.enimble,
VEARVE2D.enimble.
```

We then use commands

```
bridge VE3DVE2D.enimble
bridge VE3DVEAR.enimble
bridge VE2DVE3D.enimble
bridge VE2DVEAR.enimble
bridge VEARVE3D.enimble
bridge VEARVE2D.enimble
```

to generate the source code for six coercion modules:

Coercion_VE3DVE2D (Figures 3.15 through 3.18),
Coercion_VE3DVEAR (Figure C.8),
Coercion_VE2DVE3D (Figure 3.14),
Coercion_VE2DVEAR (Figure C.9),
Coercion_VEARVE3D (Figure C.10),
Coercion_VEARVE2D (Figures C.11 through C.14)

and their Makefiles:

`VE3DVE2D.makefile` (Figure 4.6),
`VE3DVEAR.makefile` (Figure C.15),
`VE2DVE3D.makefile` (Figure 4.7),
`VE2DVEAR.makefile` (Figure C.16),
`VEARVE3D.makefile` (Figure C.17),
`VEARVE2D.makefile` (Figure C.18).

The Makefiles contain configuration commands needed to build the coercion modules. After making the coercion modules, programmers invoke the software bus to start up the composite application via command

```
sbus -b desc.bus
```

The run-time option `-b` tells the bus this is a composite application built from existing applications. When all processes for modules are invoked, the software bus will handle event mapping automatically at run time.

5.4 Discussion

The virtual environment applications give us a rough idea of the amount of code automatically generated using our integration approach. We have measured four virtual environment examples: VE3D, VE2D, VEAR, and integrated VE3D/VE2D/VEAR.

Figure 5.5 summarizes the amount of code generated in the VE3D application. The module interaction specification created by users has 32 lines of code. Based on this specification, we automatically generated a Makefile specification (15 lines), a bus file (12 lines) and 95 lines of interface software in C.

Figure 5.6 lists the amount of code generated in the VE2D application. The module interaction specification created by users has 54 lines of code. Based on this specification, we automatically generated a Makefile specification (19 lines), a bus file (18 lines) and 221 lines of interface software in C.

Figure 5.7 shows the amount of code generated in the VEAR application. The module interaction specification created by users has 32 lines of code. Based on this specification, we automatically generated a Makefile specification (15 lines), a bus file (12 lines) and 95 lines of interface software in C.

User-written Code	`Module Interaction Spec`	32
Generated Code	`Makefile`	15
	`Interface Software in C`	95
	`Bus File`	12

Figure 5.5: Amount of code generated for VE3D

User-written Code	`Module Interaction Spec`	54
Generated Code	`Makefile`	19
	`Interface Software in C`	221
	`Bus File`	18

Figure 5.6: Amount of code generated for VE2D

User-written Code	`Module Interaction Spec`	32
Generated Code	`Makefile`	15
	`Interface Software in C`	95
	`Bus File`	12

Figure 5.7: Amount of code generated for VEAR

User-written Code	`Module Interaction Spec for VE3D`	32
	`Module Interaction Spec for VE2D`	54
	`Module Interaction Spec for VEAR`	32
	`Composite Interaction Spec`	30
Generated Code	`Bus File`	68
	`Event Transformation Files`	36
	`Source Code for Coercion Modules in C`	634
	`Makefiles for Coercion Modules`	90

Figure 5.8: Amount of code generated in the VE3D/VE2D/VEAR (version 1)

Figure 5.8 summarizes the amount of code generated in the first version of the VE3D/VE2D/VEAR example. The interaction specifications created by users have 148 lines of code, including 30 lines for the composite interaction specification, 32 lines for `ve3d.cl`, 54 lines for `ve2d.cl`, and 32 lines for `vear.cl`. Notice that users must only create the composite interaction specifications at this point because the module interaction specifications were already created when building the applications from scratch. Based on these specifications, POLYAN automatically generated a bus file (68 lines) and six event transformation files (36 lines). From the event transformation files, BRIDGE automatically generated the source code for the coercion modules in C (634 lines) and six Makefiles for the coercion modules (90 lines). The Makefiles contain 22 configuration commands needed to build the coercion modules.

The above application shows how to integrate three existing applications by defining the interactions at a high level and having the interface software generated automatically. Since interactions are specified separately from the application modules, programmers can build applications of different interactions out of the same existing applications by changing the specifications. The above example illustrates one of many possible interactions of the three applications. Next, we will study three more versions, each with different interactions among the modules. We will use version 1 as our baseline and discuss what programmers need to do to achieve different interactions.

Version 2

Version 2 has a different interaction at the configuration level. In version 2, VE3D is still interested in both VE2D and VEAR, but VE2D is only interested in VE3D, and VEAR is only interested in VE2D. To build the second version of the composite application, programmers can simply delete five lines of code in the original composite specification (Figure 5.4). For example, event *obj* from VE3D will not be mapped to event *ext_obj* in VEAR; event *obj* from VEAR will not be mapped to events *ext_obj*, *ext_obj_xy*, *ext_obj_yz* and *ext_obj_xz* in VE2D. The new composite specification is shown in Figure 5.9. Changes made in specifications will result in changes to the application structure, which is reflected in the bus file. Two coercion modules, **Coercion_VE3DVEAR** and **Coercion_VEARVE2D**, from version 1 are not needed. The integration tools will handle all the low-level changes automatically.

Version 3

Version 3 has a different interaction at the module level. In version 3, we will remove all **Log** modules and **Display_XY** from version 1. This can be done by changing nine lines of code in the original composite specification, that is, by adding four `exclude` clauses, by deleting three `name` clauses, and by changing two `map` clauses (i.e., event *obj* from VE3D and VEAR will not be mapped to event *ext_obj_xy* in VE2D. The new composite specification is shown in Figure 5.10. It will result in changes to the application structure reflected in the bus file, to the event transformation rules reflected in the event transformation files, to the source code for two coercion modules, and to the Makefiles for building those two coercion modules.

Version 4

```
composite "VE3D/VE2D/VEAR":{
  application VE3D;
  application VE2D;
  application VEAR;

  exclude Simulator from VE3D;
  exclude Simulator from VE2D;
  exclude Simulator from VEAR;

  name Controller from VE3D Controller_3D;
  name Log from VE3D Log_3D;
  name Controller from VE2D Controller_2D;
  name Log from VE2D Log_2D;
  name Display from VEAR Display_AR;
  name Controller from VEAR Controller_AR;
  name Log from VEAR Log_AR;

  map VE3D obj(x: integer; y: integer; z: integer)
  to VE2D ext_obj(string('chen'); {x; y; z})
  to VE2D ext_obj_xy(string('chen'); {x; y})
  to VE2D ext_obj_yz(string('chen'); {y; z})
  to VE2D ext_obj_xz(string('chen'); {x; z})

  map VE2D obj({x:integer; y: integer; z: integer})
  to VE3D ext_obj(string('jim'); a.x; a.y; a.z)
  to VEAR ext_obj(string('jim'); [a.z; a.y; a.x]);

  map VEAR obj(point: integer[3])
  to VE3D ext_obj(string('liz'); point[2]; point[1]; point[0])
}
```

Figure 5.9: Composite interaction specification for the VE3D/VE2D/VEAR application using ENimble (version 2)

Version 4 keeps the same configuration as in version 1 at both the configuration level and module level. It differs in that it has a different set of event transformation rules. Suppose the first, second, and third elements of the array in VEAR correspond to x, y, z coordinates in VE3D and VE2D (instead of z, y, x). This transformation can be made by changing seven lines of code in the original composite specification, i.e., changing seven event transformation rules in the two `map` clauses. The new composite specification, shown in Figure 5.11, is the result of changes to four event transformation files and to the source code for four coercion modules scattered through seven different `.c` files.

The four versions of the virtual environment example demonstrate that we can build composite applications of different interactions out of the same existing applications. Given version 1 as the baseline, programmers need only change several lines of code in the specifications; the integration tools are responsible for the low-level changes. Our work greatly facilitates prototyping. When prototyping, our approach allows developers to easily construct a prototype from existing applications and to experiment with prototypes of different interactions. The real value of this work is realized when programmers seek to change the interactions among existing applications because programmers can define event interactions at a high level and have the valid implementations generated automatically.

```
composite "VE3D/VE2D/VEAR":{
  application VE3D;
  application VE2D;
  application VEAR;

  exclude Simulator from VE3D;
  exclude Simulator from VE2D;
  exclude Simulator from VEAR;
  exclude Log from VE3D;
  exclude Log from VE2D;
  exclude Log from VEAR;
  exclude Display_XY from VE2D;

  name Controller from VE3D Controller_3D;
  name Controller from VE2D Controller_2D;
  name Display from VEAR Display_AR;
  name Controller from VEAR Controller_AR;

  map VE3D obj(x: integer; y: integer; z: integer)
  to VE2D ext_obj(string('chen'); {x; y; z})
  to VE2D ext_obj_yz(string('chen'); {y; z})
  to VE2D ext_obj_xz(string('chen'); {x; z})
  to VEAR ext_obj(string('chen'); [z, y, x]);

  map VE2D obj({x:integer; y: integer; z: integer})
  to VE3D ext_obj(string('jim'); a.x; a.y; a.z)
  to VEAR ext_obj(string('jim'); [a.z; a.y; a.x]);

  map VEAR obj(point: integer[3])
  to VE3D ext_obj(string('liz'); point[2]; point[1]; point[0])
  to VE2D ext_obj(string('liz'); {point[2]; point[1]; point[0]})
  to VE2D ext_obj_yz(string('liz'); {point[1]; point[0]})
  to VE2D ext_obj_xz(string('liz'); {point[2]; point[0]})
}
```

Figure 5.10: Composite interaction specification for the VE3D/VE2D/VEAR application using ENimble (version 3)

```
composite "VE3D/VE2D/VEAR":{
  application VE3D;
  application VE2D;
  application VEAR;

  exclude Simulator from VE3D;
  exclude Simulator from VE2D;
  exclude Simulator from VEAR;

  name Controller from VE3D Controller_3D;
  name Log from VE3D Log_3D;
  name Controller from VE2D Controller_2D;
  name Log from VE2D Log_2D;
  name Display from VEAR Display_AR;
  name Controller from VEAR Controller_AR;
  name Log from VEAR Log_AR;

  map VE3D obj(x: integer; y: integer; z: integer)
  to VE2D ext_obj(string('chen'); {x; y; z})
  to VE2D ext_obj_xy(string('chen'); {x; y})
  to VE2D ext_obj_yz(string('chen'); {y; z})
  to VE2D ext_obj_xz(string('chen'); {x; z})
  to VEAR ext_obj(string('chen'); [x, y, z]);

  map VE2D obj({x:integer; y: integer; z: integer})
  to VE3D ext_obj(string('jim'); a.x; a.y; a.z)
  to VEAR ext_obj(string('jim'); [a.x; a.y; a.z]);

  map VEAR obj(point: integer[3])
  to VE3D ext_obj(string('liz'); point[0]; point[1]; point[2])
  to VE2D ext_obj(string('liz'); {point[1]; point[1]; point[2]})
  to VE2D ext_obj_xy(string('liz'); {point[0]; point[1]})
  to VE2D ext_obj_yz(string('liz'); {point[1]; point[2]})
  to VE2D ext_obj_xz(string('liz'); {point[0]; point[2]})
}
```

Figure 5.11: Composite interaction specification for the VE3D/VE2D/VEAR application using ENimble (version 4)

Chapter 6

Summary

An event-based distributed application is a group of software components interacting with each other by producing events which in turn trigger the invocation of procedures. In this work, we are concerned with technologies and methods for integrating an event-based application, whether that application is being constructed from scratch or synthesized from existing systems. Developing an event-based application is a complex task for programmers, who must address several issues not found in traditional systems and, currently, must do so without much assistance. These issues include event declaration, structure, binding, and naming.

While much work has been done on automatic generation of interface software for distributed applications using explicit invocation or point-to-point communication, there is little support for event-based distributed applications using implicit invocation or multicast communication, especially for facilitating interoperation among existing event-based applications. Our objective has been to provide the same software engineering benefits to programmers of event-based applications as are currently provided to programmers of applications using traditional RPC or message-passing mechanisms.

In this work, we have broadened the technology for integration to encompass event-based programming. This thesis describes an approach for integrating event-based distributed applications at the configuration-level by defining, relating, and using the abstractions of event interactions. Our approach separates the event interaction properties from the implementation of the application modules, so that system integration can be performed using only the abstractions. The event interaction properties can be divided into two types of requirements: module interaction requirements and composite interaction requirements. Module interaction requirements are the event-based behavior and communication of the individual modules in an application. Composite interaction requirements define the relationships among the existing event-based applications and describe the way we want them to interact. Based upon the abstract aggregate, our integration tools can automatically generate the interface software that implements programmer-defined interactions.

The obligations for a user of this approach to building an event-based application from scratch are listed below:

1. Provide a specification of the event interaction properties of individual modules in the application.
2. Provide the source code for the application modules.
3. Execute the stub generator to generate interface software tailored to the environment.
4. Execute the generated configuration commands to create executables for the application.

The obligations for a user of this approach to building an event-based application from existing applications are listed below:

1. Provide a specification of the event interaction properties for each existing application that will be part of the new application.
2. Provide a specification of the event interaction properties for how the existing applications are to interact in the new application.
3. Provide the executables for the existing applications that will be part of the new application.
4. Execute the stub generator to generate coercion modules tailored to the interaction among existing applications and to the environment.
5. Execute the generated configuration commands to create executables for the coercion modules.

The main contributions of this work are simplifying the task of building event-based distributed applications by helping programmers to define and analyze the event interaction properties, and providing support for integrating an application from scratch and for enabling interoperation among existing event-based applications.

This work demonstrates that event interaction properties of event-based distributed applications can be specified separately from the implementation of the application modules. These specifications can be used to generate interface software automatically. The generated interface software implements programmer-defined interactions. Using our approach, programmers do not have to introduce extra interfacing code to the application modules. This allows the application source to remain simpler and easier to maintain. Our approach reduces the opportunity of introducing errors if changes are made manually. The generated interface software is the valid implementation of the interaction. Since interactions are specified separately from the application modules, programmers can build applications with different interactions using the same application modules. Our approach allows the software modules to be reused in a broader range of applications. The real value of this work is realized when programmers seek to change the event interactions because they can reason about the interactions at the abstract level, and then have the valid interface software generated automatically. In this way, programmers can define complex event interactions abstractly without having to edit modules' sources; programmers can more easily integrate and experiment with event-based distributed applications and can more easily reuse the existing software in a broader range of applications.

Finally, our integration approach can be applied to other software interconnection systems. Although we chose the POLYLITH software bus as our execution environment, this software bus selection does not constrain the applicability of our integration approach; the specification languages and integration tools can be updated and extended to include other run-time systems as new standards and systems are developed.

Appendix A

Polylith Multicast Primitives

In order to let modules interact in a multicasting environment, application modules should be able to declare events, define their event structures, register events of interest, multicast events and receive events of interest. In this chapter, we give detailed description of Polylith multicast primitives.

Declare an event and define the event structure:

mh_declare_event (*EventName*, *EventStructure*)

`char*` *EventName*;
`char*` *EventStructure*;

The `mh_declare_event` call declares an event: the event name is *EventName*; the event structure is *EventStructure*. An event must be declared by some module before it is used. The call to `mh_declare_event` returns 0 if everything is ok, else it returns a negative value.

Register interest in an event:

mh_rgsmulticast (*EventName*)

`char*` *EventName*;

The `mh_rgsmulticast` call registers a module's interest in event *EventName*. A module must register its interest in an event before receiving it. When event *EventName* is multicast, a copy of the event will be sent to this module's predefined multicast interface, then the module could receive the event by calling `mh_next_event`, `mh_bgetmsg` or `mh_getmsg`. A module could register its interest in more than one event. The call to `mh_rgsmulticast` returns 0 if everything is ok, else it returns a negative value.

Unregister interest in an event:

mh_unrgsmultcast (*EventName*)

`char*` *EventName*;

The `mh_unrgsmulticast` call deregisters a module's interest in event *EventName*. When event *EventName* is multicast, a copy of the event will not be sent to this module's predefined multicast interface anymore, so the module can not receive event *EventName*. The call to `mh_unrgsmulticast` returns 0 if everything is ok, else it returns a negative value.

Multicast an event:

mh_multicast (*EventName*, *EventStructure*, var_1, ..., var_n)

`char*` *EventName*;
`char*` *EventStructure*;
$type_1$ var_1;
$\vdots$ $\vdots$
$type_n$ var_n;

The `mh_multicast` call multicasts event *EventName* with the parameters of the event in var_1, ..., var_n. The *EventStructure* is a string made up of `tape codes` from Table A.1 which must match the variables var_1, ..., var_n. The type of the i*th* variable must match the i*th* element of the *EventStructure*. Multicast is a mode of communication where events produced by some module can be sent to a set of modules at the same time (See TR for details). The bus will send a copy of the event to each module that has registered its interest in event *EventName*. The call to `mh_multicast` returns 0 if everything is ok, else it returns a negative value.

Multicast an event and send signals:

mh_signal_multicast (*EventName*, *EventStructure*, var_1, ..., var_n)

Tape Code	*Variable Type*	*Description*
S	`char*`	String
i	`int*`	Pointer to integer
b	`int*`	Pointer to boolean
f	`double*`	Pointer to double
{ *StructTape* }	`struct` *StructName*	Structure
{ *StructTape* }	`struct` *StructName*`*`	Pointer to structure

Table A.1: POLYLITH tape codes when multicasting events in C

`char*` *EventName*;
`char*` *EventStructure*;
$type_1$ var_1;
$\vdots$ $\vdots$
$type_n$ var_n;

The **`mh_signal_multicast`** call is similar to **`mh_multicast`**. It multicasts event *EventName* with the parameters of the event in variables $var_1, \ldots, var_n$. The *EventStructure* is a string made up of **`tape codes`** from table A.1 which must match the variables $var_1, \ldots, var_n$. The type of the i*th* variable must match the i*th* element of the *EventStructure*. In addition to multicasting event *EventName* with the event structure in *EventStructure*, it sends signals at the same time. The bus will send a copy of the event and signal SIGUSR1 (see <signal.h>) to each module that is interested in event *EventName*. It is useful when modules want to be informed of the arrival of event immediately. The call to **`mh_signal_multicast`** returns 0 if everything is ok, else it returns a negative value.

Receive an event (non-blocking):

mh_getmsg (*EventStructure*, *EventName*, $var_1, \ldots, var_n$)

`char*` *EventStructure*;
`char*` *EventName*;
$type_1$ var_1;
$\vdots$ $\vdots$
$type_n$ var_n;

The **`mh_getmsg`** call receives an event, assigns event name to variable *EventName*, event structure to variable *EventStructure*, and parameters of the event to variables $var_1, \ldots, var_n$. The *EventStructure* is a string made up of **tape codes** from table A.1 which must match the variables $var_1, \ldots, var_n$. The type of the i*th* variable must match the i*th* element of the *EventStructure*. This operation is non-blocking, returning an event from the event queue if one is available, otherwise a standard null event will be returned (see `mh_nomsg`). If `mh_signal_multicast` is called to multicast events, usually `mh_getmsg` or `mh_bgetmsg` is called in its signal handler The call to `mh_getmsg` returns 0 if everything is ok, else it returns a negative value.

Receive an event (blocking):

mh_bgetmsg (*EventStructure*, *EventName*, var_1, ..., var_n)

`char*` *EventStructure*;
`char*` *EventName*;
$type_1$ var_1;
⋮ ⋮
$type_n$ var_n;

The `mh_bgetmsg` call receives an event, assigns event name to variable *EventName*, event structure to variable *EventStructure*, and value of the event to variables var_1, ..., var_n. The *EventStructure* is a string made up of **tape codes** from table A.1 which must match the variables var_1, ..., var_n. The type of the i*th* variable must match the i*th* element of the *EventStructure*. This operation is blocking, returning an event from the event queue if one is available, otherwise it waits until one is available. The call to `mh_bgetmsg` returns 0 if everything is ok, else it returns a negative value.

Get next event:

mh_next_event (*EventStructure*, *EventName*, *Event*)

`char*` *EventStructure*;
`char*` *EventName*;
`char*` *Event*;

The `mh_next_event` call receives a multicast event, assigns value of event structure to the variable *EventStructure*, value of event name to the variable *EventName*, and the raw event to buffer *Event*. The event buffer will be decoded using `mh_decode_event`. This operation is blocking, returning an event from the event queue if one is available, otherwise it waits until one is available. The call to `mh_next_event` returns 0 if everything is ok, else it returns a negative value.

Decode an event according to the event structure:

mh_decode_event (*EventStructure*, *Event*, var_1, ..., var_n)

`char*` *EventStructure*;
`char*` *Event*;
$type_1$ var_1;
⋮ ⋮
$type_n$ var_n;

The `mh_decode_event` call decodes an event pointed to by *Event* according to the event structure given in the variable *EventStructure*, and assigns values to the variables var_1, ..., var_n. event name to the variable *EventName*. The *EventStructure* is a string made up of **tape codes** from table A.1 which must match the variables var_1, ..., var_n being decoded. The type of the i*th* variable must match the i*th* element of the *EventStructure*. The call to `mh_decode_event` returns 0 if everything is ok, else it returns a negative value.

Declare a standard null event:

mh_nomsg (*EventStructure*, var_1, ..., var_n)

`char*` *EventStructure*;
$type_1$ var_1;
⋮ ⋮
$type_n$ var_n;

The `mh_nomsg` call declares a standard null message. The *EventStructure* is a string made up of **tape codes** from table A.1 which must match the variables var_1, ..., var_n. The type of the i*th* variable must match the i*th* element of the *EventStructure*. When a module calls `mh_getmsg` and no one is available, a standard null event will be returned (see `mh_getmsg`). It should be called before `mh_getmsg` in case no events are available. The call to `mh_nomsg` returns 0 if everything is ok, else it returns a negative value.

Query declared events:

mh_query_msgtype (*Buffer*, *rsize*)

`char*` *MsgTypeBuffer*;
`int` *rsize*;

The `mh_query_msgtype` call queries events that have already declared. It returns event names of all the declared events in *Buffer*, including event names of events declared by other modules. Event names are separated by ",". The format of *Buffer* is "*event1,event2,...*" for example. The call to `mh_query_msgtype` returns 0 if everything is ok, else it returns a negative value.

Query event registration status:

mh_query_rgsmsgtype (*Buffer*, *rsize*)

`char*` *Buffer*;
`int` *rsize*;

The `mh_query_rgsmsgtype` call queries event registration status. It returns its registration status of all the declared events in *Buffer*. The format of *Buffer* is "*event1,yes,event2,no, ...*" for example. Each event is followed by "yes" if the module registered its interest in it, otherwise followed by "no". Once again, the events may be declared by other modules. The call to `mh_query_rgsmsgtype` returns 0 if everything is ok, else it returns a negative value.

Appendix B

Integration Tools

In this chapter, we summarize a set of tools we have built to support the integration process. These tools analyze event interaction specifications and generate customized interface software automatically.

Preprocessor

This tool takes enhanced Polylith module interaction specifications (`.cl`), extracts, preprocesses and rewrites them in a format understood by the Polylith software bus (`bus.desc`). The output of the tool will be fed to the bus to start up the application.

Example:

```
preprocessor ve3d.cl bus.desc
```

The input file `ve3d.cl` contains Polylith module interaction specifications. The output file `bus.desc` contains the operational specifications for the software bus.

Polystub

This tool takes the bus file as input, generates interface software for the application modules and makefiles containing configuration commands needed to build the modules (`.makefile`).

Example:

```
polystub bus.desc ve3d.makefile
```

The input file `bus.desc` contains the operational specifications for the software bus. The output file `ve3d.makefile` is the Makefile for building the application.

Polyan

This tool takes module interaction specifications and composite interaction specifications as input, analyzes them, and rewrites them in a format understood by the Polylith software bus. It also produces event transformation files (`.enimble`) containing event transformation rules.

Example:

```
polyan ve3d.cl ve2d.cl comp bus.desc
```

The input files `ve3d.cl` and `ve2d.cl` contain Polylith module interaction specifications; `comp` contains composite interaction specifications. The output file `bus.desc` contains the operational specifications for the software bus. Other output files contain event transformation rules.

Bridge

This tool takes an event transformation file as input, generates source code for coercion modules and makefiles containing configuration commands needed to build the coercion modules. The execution of `Bridge` could be broken down into several steps:

- Split event transformation files (`.enimble`) into separate files (`.data`), each containing only one event transformation rule;
- Invoke tool `enimble` to perform the equivalence checking between the target event structure and the range of the event map, and
- Create the operational specification (`.p` files) for the event transformation rule;
- Invoke tool `eventgen` to translate the operational specification to the source code for the coercion module in the target programming language (including the main routine and the transformation routine), and
- Create the Makefile (`.makefile`) for building the coercion module.

Example:

```
bridge VE2DVE3D.enimble
```

The input file `VE2DVE3D.enimble` contains a set of event transformation rules. The output files are a source file `VE2DVE3D.c` for the main routine of the coercion module, and a Makefile `VE2DVE3D.makefile` for building the coercion module.

Enimble

This tool takes a separate event transformation file containing only one event transformation rule as input, performs the equivalence checking between the target event structure and the range of the event map, and creates the operational specification (`.p` files) for the event transformation rule. The `.p` files will be input for tool `eventgen`. This tool is invoked by `Bridge`, not by users.

Example:

```
enimble VE2DVE3D.enimble1.data
```

The input file `VE2DVE3D.enimble1.data` contains an event transformation rule. The output file `VE2D_obj_VE3D_ext_obj.p` contains the operational specification for the event transformation rule.

Eventgen

This tool takes the operational specification for the event transformation rule (`.p`) as input, translates the operational specification to the source code for the coercion module in the target programming language (including the main routine and the transformation routine), and creates the Makefile (`.makefile`) for building the coercion module. This tool is invoked by `Bridge`, not by users.

Example:

```
eventgen VE2D_obj_VE3D_ext_obj.p VE2D_obj_VE3D_ext_obj.c
```

The input file `VE2D_obj_VE3D_ext_obj.p` contains the operational specification for the event transformation rule. The output file `VE2D_obj_VE3D_ext_obj.c` contains the source code for the event transformation routine.

Appendix C

Some Figures from the Case Study

In this chapter, we list some figures from the case study described in Chapter 5.

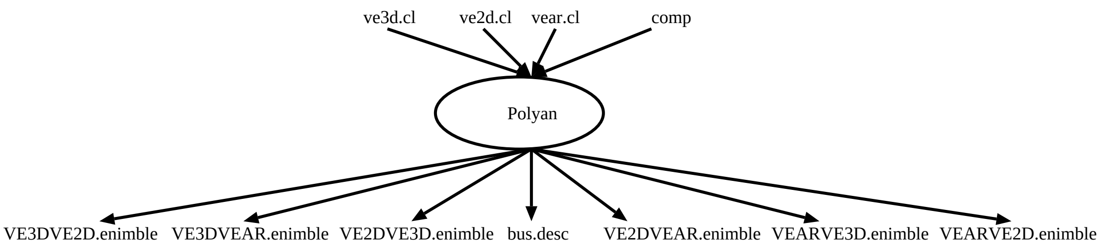

Figure C.1: Production graph for the integrated VE3D/VE2D/VEAR example (part 1 of 7)

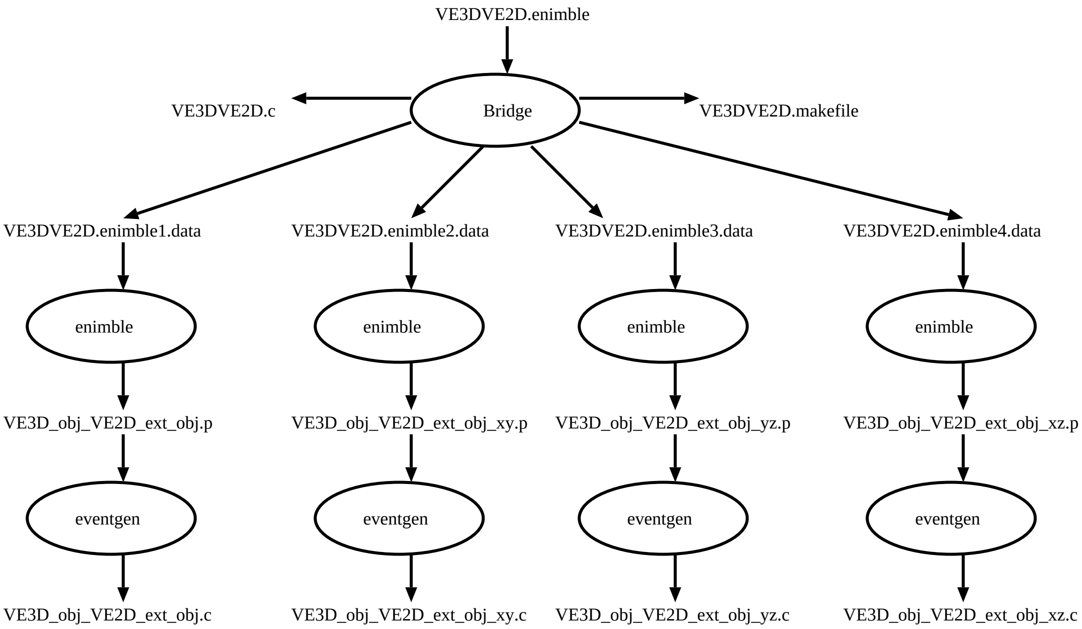

Figure C.2: Production graph for the integrated VE3D/VE2D/VEAR example (part 2 of 7)

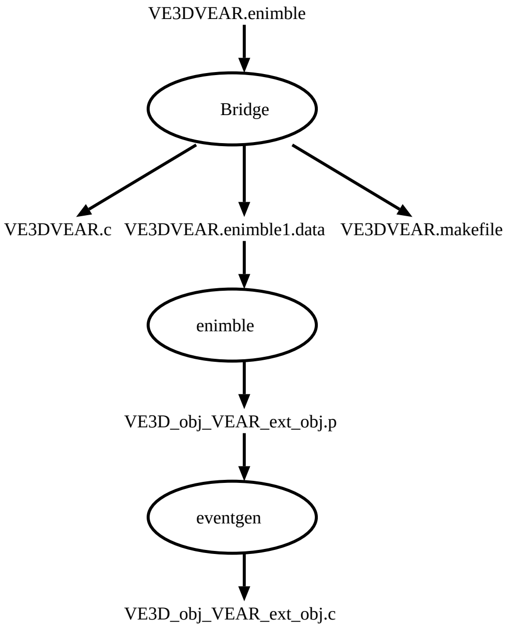

Figure C.3: Production graph for the integrated VE3D/VE2D/VEAR example (part 3 of 7)

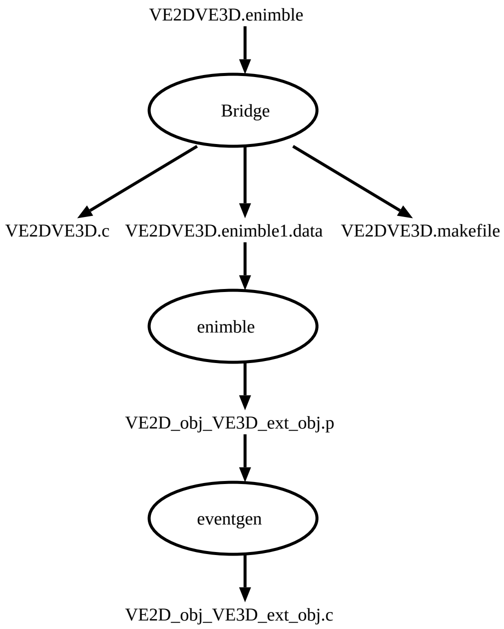

Figure C.4: Production graph for the integrated VE3D/VE2D/VEAR example (part 4 of 7)

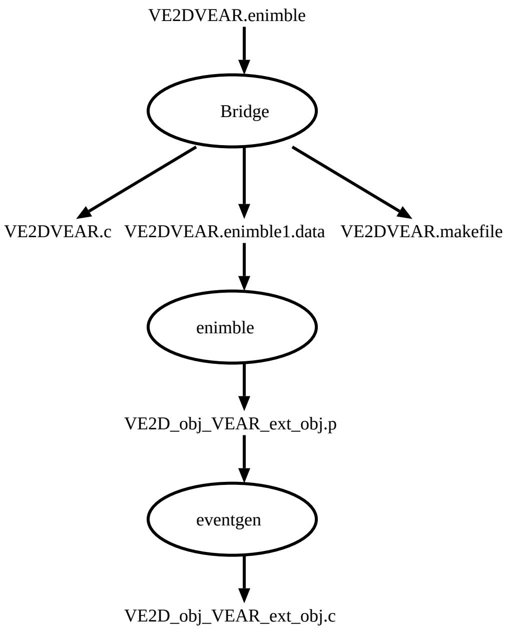

Figure C.5: Production graph for the integrated VE3D/VE2D/VEAR example (part 5 of 7)

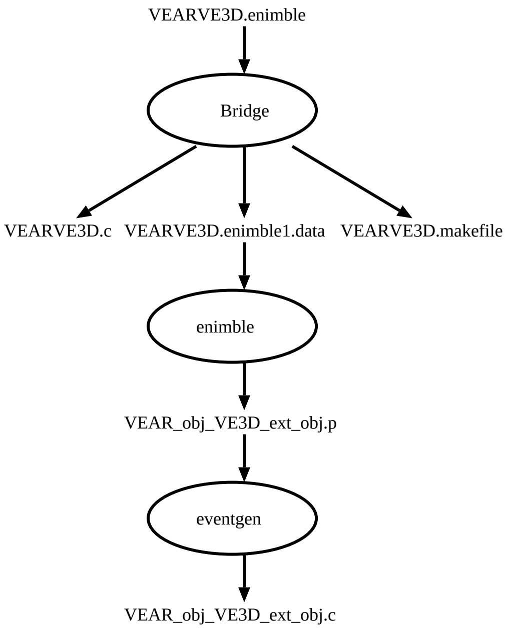

Figure C.6: Production graph for the integrated VE3D/VE2D/VEAR example (part 6 of 7)

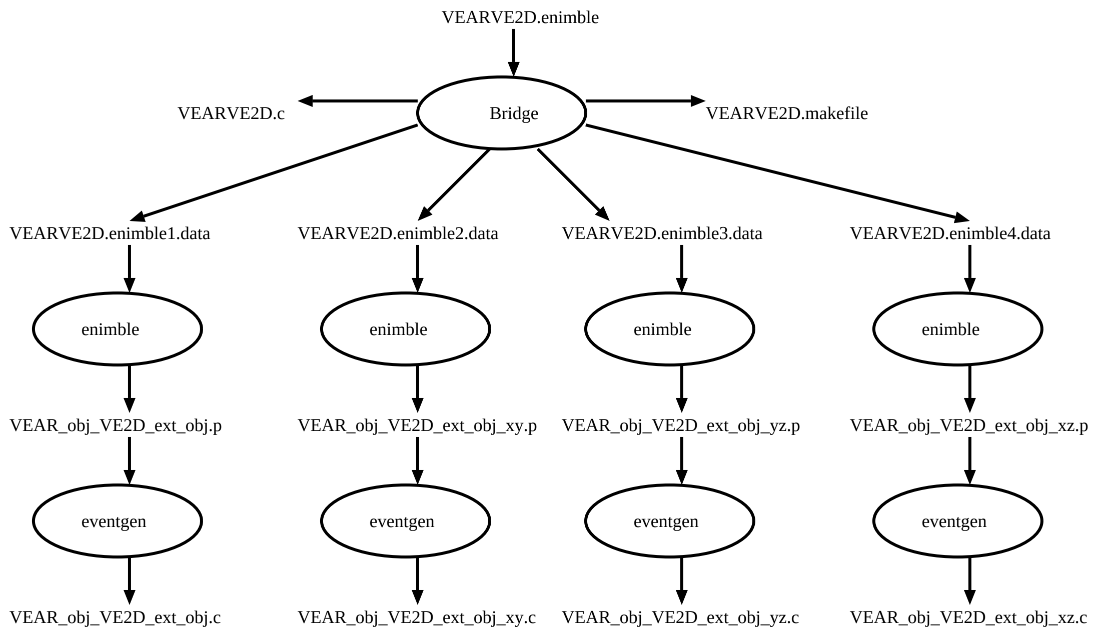

Figure C.7: Production graph for the integrated VE3D/VE2D/VEAR example (part 7 of 7)

```
/* VE3DVEAR.c */

#include <stdio.h>
extern transform_VE3D_obj_VEAR_ext_obj();
main(argc, argv)
int argc;
char *argv[];
{
   char event_name[256], tape[256];
   char *event;

   mh_init(&argc, &argv, NULL, NULL);
   mh_rgsmulticast("obj");
   while (1) {
      mh_next_event(tape, event_name, &event);
      if (!strcmp(event_name, "obj"))
       transform_VE3D_obj_VEAR_ext_obj(tape, event);
   }
}

  /* VE3D_obj_VEAR_ext_obj.c */

#include <stdio.h>
struct VE3D_obj {
   int x;
   int y;
   int z;
};
struct VEAR_ext_obj
   char *name;
   int point[3];
}
struct VE3D_obj From_VE3D_obj;
struct VEAR_ext_obj *To_VEAR_ext_obj;

struct VEAR_ext_obj *Coerce_VE3D_obj_VEAR_ext_obj(From)
struct VE3d_obj *From;
{
   struct VEAR_ext_obj To;

   To.name = "jim";
   To.point[0] = From->z;
   To.point[1] = From->y;
   To.point[2] = From->x;
   return(&To);
}

void transform_VE3D_obj_VEAR_ext_obj(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VE3D_obj.x),
                     &(From_VE3D_obj.y), &(From_VE3D_obj.z));
      To_VEAR_ext_obj = Coerce_VE3D_obj_VEAR_ext_obj(&From_VE3D_obj);
      mh_multicast("ext_obj", "SV3I", To_VEAR_ext_obj->name,
                     To_VE3D_ext_obj->point);

}
```

Figure C.8: Source code for coercion module **Coercion_VE3DVEAR**

```
/* VE2DVEAR.c */

#include <stdio.h>
extern transform_VE2D_obj_VEAR_ext_obj();
main(argc, argv)
int argc;
char *argv[];
{
   char event_name[256], tape[256];
   char *event;

   mh_init(&argc, &argv, NULL, NULL);
   mh_rgsmulticast("obj");
   while (1) {
      mh_next_event(tape, event_name, &event);
      if (!strcmp(event_name, "obj"))
       transform_VE2D_obj_VEAR_ext_obj(tape, event);
   }
}

  /* VE2D_obj_VEAR_ext_obj.c */

#include <stdio.h>
struct Aa {
   int x;
   int y;
   int z;
};
struct VE2D_obj {
   struct Aa a;
};
struct VEAR_ext_obj {
   char *name;
   int point[3];
};
struct VE2D_obj From_VE2D_obj;
struct VEAR_ext_obj *To_VEAR_ext_obj;

struct VEAR_ext_obj *Coerce_VE2D_obj_VEAR_ext_obj(From)
struct VE2D_obj *From;
{
   struct VEAR_ext_obj To;

   To.name = "jim";
   To.point[0] = From->a.z;
   To.point[1] = From->a.y;
   To.point[2] = From->a.x;
   return(&To);
}

void transform_VE2D_obj_VEAR_ext_obj(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VE2D_obj.a));
      To_VE3D_ext_obj = Coerce_VE2D_obj_VEAR_ext_obj(&From_VE2D_obj);
      mh_multicast("ext_obj", "SV3I", To_VEAR_ext_obj->name,
                     To_VEAR_ext_obj->point);
}
```

Figure C.9: Source code for coercion module **Coercion_VE2DVEAR**

```
/* VEARVE3D.c */

#include <stdio.h>
extern transform_VEAR_obj_VE3D_ext_obj();
main(argc, argv)
int argc;
char *argv[];
{
   char event_name[256], tape[256];
   char *event;

   mh_init(&argc, &argv, NULL, NULL);
   mh_rgsmulticast("obj");
   while (1) {
      mh_next_event(tape, event_name, &event);
      if (!strcmp(event_name, "obj"))
       transform_VEAR_obj_VE3D_ext_obj(tape, event);
   }
}

  /* VEAR_obj_VE3D_ext_obj.c */

#include <stdio.h>
struct VEAR_obj {
   int point[3];
};
struct VE3D_ext_obj {
   char *name;
   int x;
   int y;
   int z;
};
struct VEAR_obj From_VEAR_obj;
struct VE3D_ext_obj *To_VE3D_ext_obj;

struct VE3D_ext_obj *Coerce_VEAR_obj_VE3D_ext_obj(From)
struct VEAR_obj *From;
{
   struct VE3D_ext_obj To;

   To.name = "jim";
   To.x = From->point[2];
   To.y = From->point[1];
   To.z = From->point[0];
   return(&To);
}

void transform_VEAR_obj_VE3D_ext_obj(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VEAR_obj.point));
      To_VE3D_ext_obj = Coerce_VEAR_obj_VE3D_ext_obj(&From_VEAR_obj);
      mh_multicast("ext_obj", "SIII", To_VE3D_ext_obj->name, To_VE3D_ext_obj->x,
                   To_VE3D_ext_obj->y, To_VE3D_ext_obj->z);
}
```

Figure C.10: Source code for coercion module **Coercion_VEARVE3D**

```
/* VEARVE2D.c */
#include <stdio.h>
extern transform_VEAR_obj_VE2D_ext_obj();
extern transform_VEAR_obj_VE2D_ext_obj_xy();
extern transform_VEAR_obj_VE2D_ext_obj_yz();
extern transform_VEAR_obj_VE2D_ext_obj_xz();
main(argc, argv)
int argc;
char *argv[];
{
   char event_name[256], tape[256];
   char *event;

   mh_init(&argc, &argv, NULL, NULL);
   mh_rgsmulticast("obj");
   while (1) {
      mh_next_event(tape, event_name, &event);
      if (!strcmp(event_name, "obj"))
       transform_VEAR_obj_VE2D_ext_obj(tape, event);
      if (!strcmp(event_name, "obj"))
       transform_VEAR_obj_VE2D_ext_obj_xy(tape, event);
   }
}


/* VEAR_obj_VE2D_ext_obj.c */

#include <stdio.h>
struct Fa {
   int x;
   int y;
   int z;
};
struct VE3D_obj {
   int point[3];
}
struct VE2D_ext_obj {
   char *name;
   struct Fa b;
};
struct VEAR_obj From_VEAR_obj;
struct VE2D_ext_obj *To_VE2D_ext_obj;

struct VE2D_ext_obj *Coerce_VEAR_obj_VE2D_ext_obj(From)
struct VE3D_obj *From;
{
   struct VE2D_ext_obj To;

   To.name = "chen";
   To.b.x = From->point[2];
   To.b.y = From->point[1];
   To.b.z = From->point[0];
   return(&To);
}

void transform_VEAR_obj_VE2D_ext_obj(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VEAR_obj.point));
      To_VE2D_ext_obj = Coerce_VEAR_obj_VE2D_ext_obj(&From_VEAR_obj);
      mh_multicast("ext_obj", "S{III}", To_VE2D_ext_obj->name,
                     To_VE2D_ext_obj->b);
}
```

Figure C.11: Source code for coercion module **Coercion_VEARVE2D** (part 1 of 4)

```
/* VEAR_obj_VE2D_ext_obj_xy.c */

#include <stdio.h>
struct Fa {
   int x;
   int y;
};
struct VEAR_obj {
   int point[3];
}
struct VE2D_ext_obj_xy {
   char *name;
   struct Fa b;
};
struct VEAR_obj From_VEAR_obj;
struct VE2D_ext_obj_xy *To_VE2D_ext_obj_xy;

struct VE2D_ext_obj_xy *Coerce_VEAR_obj_VE2D_ext_obj_xy(From)
struct VEAR_obj *From;
{
   struct VE2D_ext_obj_xy To;

   To.name = "chen";
   To.b.x = From->point[2];
   To.b.y = From->point[1];
   return(&To);
}

void transform_VEAR_obj_VE2D_ext_obj_xy(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VEAR_obj.point));
      To_VE2D_ext_obj_xy = Coerce_VEAR_obj_VE2D_ext_obj_xy(&From_VEAR_obj);
      mh_multicast("ext_obj_xy", "S{II}", To_VE2D_ext_obj->name,
                   To_VE2D_ext_obj->b);
}
```

Figure C.12: Source code for coercion module **Coercion_VEARVE2D** (part 2 of 4)

```
/* VEAR_obj_VE2D_ext_obj_yz.c */

#include <stdio.h>
struct Fa {
   int y;
   int z;
};
struct VEAR_obj {
   int point[3];
}
struct VE2D_ext_obj_yz {
   char *name;
   struct Fa b;
};
struct VEAR_obj From_VEAR_obj;
struct VE2D_ext_obj_yz *To_VE2D_ext_obj_yz;

struct VE2D_ext_obj_yz *Coerce_VEAR_obj_VE2D_ext_obj_yz(From)
struct VEAR_obj *From;
{
   struct VE2D_ext_obj_yz To;

   To.name = "chen";
   To.b.y = From->point[1];
   To.b.z = From->point[0];
   return(&To);
}

void transform_VEAR_obj_VE2D_ext_obj_yz(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VEAR_obj.point));
      To_VE2D_ext_obj_yz = Coerce_VEAR_obj_VE2D_ext_obj_yz(&From_VEAR_obj);
      mh_multicast("ext_obj_yz", "S{II}", To_VE2D_ext_obj->name,
                    To_VE2D_ext_obj->b);
}
```

Figure C.13: Source code for coercion module **Coercion_VEARVE2D** (part 3 of 4)

```
/* VEAR_obj_VE2D_ext_obj_xz.c */

#include <stdio.h>
struct Fa {
   int x;
   int z;
};
struct VEAR_obj {
   int point[3];
}
struct VE2D_ext_obj_xz {
   char *name;
   struct Fa b;
};
struct VEAR_obj From_VEAR_obj;
struct VE2D_ext_obj_xz *To_VE2D_ext_obj_xz;

struct VE2D_ext_obj_xz *Coerce_VEAR_obj_VE2D_ext_obj_xz(From)
struct VEAR_obj *From;
{
   struct VE2D_ext_obj_xz To;

   To.name = "chen";
   To.b.x = From->point[2];
   To.b.z = From->point[0];
   return(&To);
}

void transform_VEAR_obj_VE2D_ext_obj_xz(tape, event)
char *tape;
char *event;
{
      mh_decode_event(tape, event, &(From_VEAR_obj.point));
      To_VE2D_ext_obj_xz = Coerce_VEAR_obj_VE2D_ext_obj_xz(&From_VEAR_obj);
      mh_multicast("ext_obj_xz", "S{II}", To_VE2D_ext_obj->name,
                     To_VE2D_ext_obj->b);
}
```

Figure C.14: Source code for coercion module **Coercion_VEARVE2D** (part 4 of 4)

```
LITH=/bugs/chenchen/SuperBus
all: Coercion_VE3DVEAR

Coercion_VE3DVEAR: VE3DVEAR.o VE3D_obj_VEAR_ext_obj.o
   cc -o coercion_VE3DVEAR VE3DVEAR.o VE3D_obj_VEAR_ext_obj.o \
         -L${LITH} -lith

VE3DVEAR.o: VE3DVEAR.c
   cc -c -L${LITH} -lith VE3DVEAR.c

VE3D_obj_VEAR_ext_obj.o: VE3D_obj_VEAR_ext_obj.c
   cc -c -L${LITH} -lith VE3D_obj_VEAR_ext_obj.c
```

Figure C.15: Makefile for coercion module **Coercion_VE3DVEAR**

```
LITH=/bugs/chenchen/SuperBus
all: Coercion_VE2DVEAR

Coercion_VE2DVEAR: VE2DVEAR.o VE2D_obj_VEAR_ext_obj.o
   cc -o coercion_VE2DVEAR VE2DVEAR.o VE2D_obj_VEAR_ext_obj.o -L${LITH} -lith

VE2DVEAR.o: VE2DVEAR.c
   cc -c -L${LITH} -lith VE2DVEAR.c

VE2D_obj_VEAR_ext_obj.o: VE2D_obj_VEAR_ext_obj.c
   cc -c -L${LITH} -lith VE2D_obj_VEAR_ext_obj.c
```

Figure C.16: Makefile for coercion module **Coercion_VE2DVEAR**

```
LITH=/bugs/chenchen/SuperBus
all: Coercion_VEARVE3D

Coercion_VEARVE3D: VEARVE3D.o VEAR_obj_VE3D_ext_obj.o
   cc -o coercion_VEARVE3D VEARVE3D.o VEAR_obj_VE3D_ext_obj.o \
         -L${LITH} -lith

VEARVE3D.o: VEARVE3D.c
   cc -c -L${LITH} -lith VEARVE3D.c

VEAR_obj_VE3D_ext_obj.o: VEAR_obj_VE3D_ext_obj.c
   cc -c -L${LITH} -lith VEAR_obj_VE3D_ext_obj.c
```

Figure C.17: Makefile for coercion module **Coercion_VEARVE3D**

```
LITH=/bugs/chenchen/SuperBus
all: Coercion_VEARVE2D

Coercion_VEARVE2D: VEARVE2D.o VEAR_obj_VE2D_ext_obj.o
   cc -o coercion_VEARVE2D VEARVE2D.o VEAR_obj_VE2D_ext_obj.o \
         VEAR_obj_VE2D_ext_obj_xy.o VEAR_obj_VE2D_ext_obj_yz.o \
         VEAR_obj_VE2D_ext_obj_xz.o -L${LITH} -lith

VEARVE2D.o: VEARVE2D.c
   cc -c -L${LITH} -lith VEARVE2D.c

VEAR_obj_VE2D_ext_obj.o: VEAR_obj_VE2D_ext_obj.c
   cc -c -L${LITH} -lith VEAR_obj_VE2D_ext_obj.c

VEAR_obj_VE2D_ext_obj_xy.o: VEAR_obj_VE2D_ext_obj_xy.c
   cc -c -L${LITH} -lith VEAR_obj_VE2D_ext_obj_xy.c

VEAR_obj_VE2D_ext_obj_yz.o: VEAR_obj_VE2D_ext_obj_yz.c
   cc -c -L${LITH} -lith VEAR_obj_VE2D_ext_obj_yz.c

VEAR_obj_VE2D_ext_obj_xz.o: VEAR_obj_VE2D_ext_obj_xz.c
   cc -c -L${LITH} -lith VEAR_obj_VE2D_ext_obj_xz.c
```

Figure C.18: Makefile for coercion module **Coercion_VEARVE2D**

Bibliography

[1] D. Barrett, L. Clarke, P. Tarr. "An Event-Based Software Integration Framework," *Technical Report UM-94-47, Computer Science Department, University of Massachusetts*, September 1994.

[2] 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*, vol. 13, pp. 880-894, August 1987.

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

[4] J. Callahan, J. Purtilo. "A Packaging System for Heterogeneous Execution Environments," *IEEE Transactions on Software Engineering*, vol. 17, pp. 626-635, June 1991.

[5] N. Carriero, D. Gelernter. "Linda in Context," *Communications of the ACM*, vol. 32, pp. 444-458, April 1989.

[6] C. Chen, A. Porter, J. Purtilo. "Tool Support for Tailored Software Prototyping," *Proceedings of the 3rd Symposium on Assessment of Quality Software Development Tools*, pp. 171-181, June 1994.

[7] C. Chen, J. Purtilo. "Configuration-level Programming of Distributed Applications Using Implicit Invocation," *Proceedings of IEEE Region 10's Annual International Conference*, pp. 43-49, August 1994.

[8] C. Chen, J. Purtilo. "Event Adaptation for Integrating Distributed Applications," *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, June 1995.

[9] C. Chen, E. White, J. Purtilo. "A Packager for Multicast Software in Distributed Systems," *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering*, pp. 612-621, June 1993.

[10] E. Cooper. "Programming Language Support for Multicast Communication in Distributed System," *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 450-457, July 1990.

[11] O. Dahl, K. Nygaard. "Simula - An ALGOL Based Simulation Language," *Communications of ACM*, Vol. 9, pp. 671-678, 1966.

[12] F. DeRemer, H. Kron. "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, vol. 2, pp. 80-86, June 1976.

[13] P. Diviat, R. Villanueva, H. Markowitz. "SIMSCRIPT II.5 Programming Language," *CACI*, 1973.

[14] B. Fromme. "HP Encapsulator: Bridging the Generation Gap," *Technical Report SESD-89-26, Hewlett-Packard Software Engineering Systems Division*, November 1989.

[15] D. Garlan, R. Allen, J. Ockerbloom. "Architectural Mismatch or Why it's Hard to Build Systems out of Existing Parts," *Proceedings of the 17th International Conference on Software Engineering*, pp. 179-185, May 1995.

[16] D. Garlan, E. Ilias. "Low-cost, Adaptable Tool Integration Policies for Integrated Environments," *Proceedings of SIGSOFT 4th Symposium on Software Development Environments*, pp. 1-10, December 1990.

[17] D. Garlan, D. Notkin. "Formalizing Design Spaces: Implicit Invocation Mechanisms," *Proceedings of VDM'91: Formal Software Development Methods*, October, 1991.

[18] D. Garlan, C. Scott. "Adding Implicit Invocation to Traditional Programming Languages," *Proceedings of the 15th International Conference on Software Engineering*, pp. 447-455, May 1993.

[19] C. Gerety. "A new Generation of Software Development Tools," *Technical Report SESD-89-25, Hewlett-Packard Software Engineering Systems Division*, November 1989.

[20] W. Griswold, D. Notkin. "Automated Assistance for Program Restructuring," *ACM Transactions on Software Engineering and Methodology*, July 1993.

[21] C. Falkenberg, C. Hofmeister, C. Chen, E. White, J. Atlee, P. Hagger, and J. Purtilo. "The Polylith Interconnection System: Programming Manual for the Network Bus," University of Maryland, College Park, 3.0 edition, September 1993.

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

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

[24] C. Hofmeister, E. White and J. Purtilo. "Surgeon: A Packager for Dynamically Reconfigurable Distributed Applications," *IEE Software Engineering Journal*, pp. 95-101, March 1993.

[25] 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*, pp. 225-235, January 1985.

[26] A. Julienne, L. Russell. "Why You Need ToolTalk," *SunEXPERT Magazine*, pp. 51-58, March 1993.

[27] D. Luckham, J. Vera. Event-based Concepts and Language for System Architecture.

[28] J. Magee, J. Kramer and M. Sloman. "Constructing Distributed Systems in Conic," *IEEE Transactions on Software Engineering*, vol. 15, pp. 663-675, June 1989.

[29] R. Nance. "Model Development Revisited," *Proceedings of 1984 Winter Simulation Conference*, pp. 75-80, 1984.

[30] D. Notkin, D. Garlan, W. Griswold, K. Sullivan. "Adding Implicit Invocation to Languages: Three Approaches", *Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software*, November, 1993.

[31] Object Management Group. "The Common Object Request Broker: Architecture and Specification," *OMG Document Number 91.12.1*, December 1991.

[32] V. Paxson, C. Saltmarsh. "Glish: A User-Level Software Bus for Loosely Coupled Distributed Systems," *Proceedings of the Winter 1993 USENIX Conference*, pp. 141-156, January 1993.

[33] A. Pritsker. **The GASP IV Simulation Language**, Wiley, 1974.

[34] L. Pollacia. "A Survey of Discrete Event Simulation and State-of-the-Art Discrete Event Languages," *Simulation Digest*, Vol. 20, 1989.

[35] J. Purtilo. "The Polylith Software Bus," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 663-675, January 1994.

[36] J. Purtilo, J. Atlee. Module Reuse by Interface Adaptation. *Software: Practice & Experience*, vol. 21, no. 6, pp. 539-556, 1991.

[37] J. Purtilo, C. Hofmeister. "Dynamic Reconfiguration of Distributed Programs," *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 560-571, May 1991.

[38] S. Reiss. "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, vol. 7, pp. 57-66, July 1990.

[39] T. Schriber. **Simulation Using GPSS**, Wiley, 1974.

[40] S. Sullivan, D. Notkin. " Reconciling Environment Integration and Software Evolution," *ACM Transaction on Software Engineering and Methodology*, vol. 1, no. 3, July 1992.

[41] V. Sunderam. "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice & Experience*, Vol. 2, December 1990.

[42] Sun Microsystems. **Remote Procedure Call Protocol Specification**. Sun Microsystems, January 1985.

[43] SunSoft Inc. **ToolTalk 1.0.2 Programmer's Guide**. Sun Microsystems, 1992.

[44] D. Yellin, R. Strom. "Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors," *Proceedings of Object-Oriented Programming Systems and Languages*, pp. 176-190, October 1994.

[45] B. Zeigler, A. Louri. "A Simulation Environment for Intelligent Machine Architectures," *Journal of Parallel and Distributed Computing*, Vol. 18, pp. 77-88, May 1993.