Connecting Software Components with Declarative Glue

Brian W. Beach
Software and Systems Laboratory
HPL-91-152
October, 1991

database, distributed systems architecture, module interconnection language, software bus, software glue, software reuse

We describe a software bus, called Bart, that addresses the problem of maintaining flexibility in software systems by supporting *component independence.* Software components can be built independently of the context in which they are used, allowing them to be reused in many different situations.

The connections between software components are described using a novel *Software Glue* language that declaratively defines the connections between the data models in heterogeneous components. This glue language is compiled into an efficient procedural form and, to reduce communication overhead, executed on the machine where the data resides.

Bart is a software bus that handles message transport, data sharing, and connections using Software Glue. It operates in a distributed environment and can connect components written in different programming languages. We illustrate the use of Bart in supporting a hypertext system.

Internal Accession Date Only
Submitted to the *14th International Conference on Software Engineering,* May, 1992

1 Introduction

Software systems, as they become larger and/or older, tend to become less flexible; the components that comprise a system develop more interconnections and interdependencies. This is a problem, especially if there is a need to reuse the components as part of a new system. A lack of *component independence* is also a problem for the builders of reuse libraries. It is difficult to build a software component independent of the context in which it is used. Bart[1], the software bus presented in this paper, is designed to support component independence, which will in turn allow software systems to be more flexible and components more reusable.

Bart addresses both data and control integration of components in a distributed environment. By doing so, it hides much of the inherent complexity of a distributed application from the programmer. Components can send messages to one another, share data, and, through the use of a *glue language*, share data represented in different data models. By remaining independent of any specific programming language, Bart allows components written in different languages to be easily connected.

Connections between components are written in a declarative (non-procedural) glue language. Because the connections are specified outside of the components, the components remain largely independent of one another. To reduce the overhead of using a software bus, the glue language is compiled into efficient procedural code that is run inside the address space of each component.

Data is shared between components using a publish/subscribe metaphor. A component can make some data available (publish it), and any interested parties can use it (subscribe to it), without the knowledge of the publisher. Because the publisher does not need to know how its data is used, it can remain independent of context.

Databases provide a means for cooperating components to work with common data. The database schema is the middleman that shields one component from another and allows one to be changed without affecting others. For components that require a schema different from the one used in the database, views allow *data independence* by showing a specially tailored view to each component. Data independence means that the schema of the database can be changed, and as long as the same views are provided to the components running on top of it, the components can continue to operate unchanged.

Using a database to mediate between components is too inefficient for many applications, especially those that are composed of many small components that interact a great deal. The idea of data independence can still be used, though, to insulate components from changes in other components. Bart uses its glue language to provide data independence, defining the relationships between components, and bridging the gap when components have different data models. The glue language is compiled for efficient execution.

The next section gives an overview of Bart as a software bus, as a precurser to describing each of its three layers. The message transport layer is described briefly, followed by a discussion of the mapping between objects in software components and relations on the bus. The most important section is the one on Software Glue, which is the key to Bart, and gives four examples of different ways to use Software Glue. We close with an overview of related work and a summary of Bart's current status and future directions.

---

[1]Residents of the San Francisco bay area will recognize BART as the acronym for Bay Area Rapid Transit.

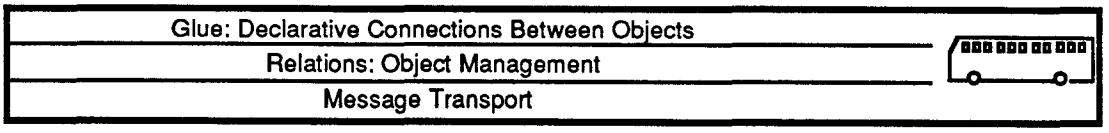

Figure 1: The three layers of the software bus.

2 The Software Bus

A *software bus* is a mechanism for connecting software components [Purt91a, Purt91b]. Software buses have been used successfully to build large applications. Bart is novel in allowing components to share object-oriented data directly, and in providing Software Glue to connect components with different data models. The bus is composed of three separate layers, each providing a higher level of component integration (see Figure 1).

The message transport layer delivers messages between components and allows for control integration between components, with multi-cast messaging in both synchronous and asynchronous styles.

The relational layer allows sharing of data between components. It translates between an object-oriented view inside components and a relational view on the bus. Objects inside a component correspond to tuples in a relation on the bus. A class of objects in one component can be connected to a relation on the bus, which can in turn be connected to a class of objects in another component. This allows the establishment of a one-to-one correspondence between objects in one component and objects in another component. This is a useful facility, but is restricted to connecting object classes with the same set of attributes and the same set of objects.

The glue layer provides the real power in Bart, permitting components with different data models to be connected. While the relational layer provides a one-to-one correspondence between objects, allowing pairs of objects to be connected, the glue layer allows more complex relationships, with any number of objects connected together. Also, an object in one component can be connected to several objects in another component. The data required for each object in a component is specified declaratively, and can come from other objects in other components, via direct or indirect links. This means that the data required by a component can be delivered directly to where it is needed, without complex access procedures.

Bart assumes that components have an event-based architecture, with a top-level loop waiting (or polling) for inputs from a number of sources. This is the architecture of applications running under X windows. When input arrives, it is dispatched to the appropriate handler routine in the application. Bart takes care of fielding input from the bus and routing it to the correct place.

3 Message Transport

The message transport layer uses a multi-cast approach, where each component indicates which messages it is interested in; when a message is sent it is delivered to all interested parties. This approach was pioneered in Field [Reiss90], and is used today in HP's Broadcast Message Server [Cagan90] (a key component of HP Softbench). A key feature of this approach is the fact that messages are sent anonymously; the sender of a message does not name the recipient. We have extended the model to include both synchronous and asynchronous messaging. The sender of a message can choose whether or not to wait for an answer.

```
relation node is
        key id: integer,
        string name,
        string text;
```

```
relation link is
        key id: integer,
        from: node,
        from_offset: integer,
        to: node,
        to_offset: integer,
        string label;
```

Figure 2: Node and link relations.

Callbacks are used to service messages. When a component registers interest in a message, it provides a callback function that is invoked whenever the message is received. For synchronous messages, the answer returned by the callback function is sent back to the originator of the message. Messages can have parameters that have one of a number of primitive types, or can be object references (see section 5 below).

4 Relations

A relation is a table. Each row of the table is called a *tuple* and each column is called a *column*. Relations form the structure used to communicate data between components connected to the software bus. The shape and properties of a relation are defined by a *schema*. The schema defines how many columns a relation has, which columns are keys, and which columns are required to have values.

The key columns of a relation are guaranteed to be unique for each tuple in a relation and are used to identify it and its corresponding object. The column types can be primitive types (such as integer or string), or tuple references. A tuple reference identifies a tuple in some relation and is stored as the key for that tuple.

Relations are the fundamental mechanism for communicating data between components connected to the software bus. Each relation is exported by one component and imported by zero, one, or many others. The component exporting the relation is responsible for letting the bus know when tuples are added, updated, and removed. It is then the bus's job to forward the information to all interested parties.

In some cases, as will be seen in later sections, it is useful to split the ownership of a relation across several components. Bart provides a very simple mechanism for distributing a relation; a special value type that identifies components, called component, is used as one of the keys for a relation. This means that the component identified by the key is the one that owns a tuple. Since we do not allow the modification of keys, tuples are not allowed to move from one components to another.

We have used two test cases to drive the work on Bart. The first is the Physician's Workstation (PWS) [Tang91], an information management system for physicians. The PWS was designed from the beginning as a distributed application, so porting it to Bart did not involve any major changes to the design. The second test case is Kiosk [CFG91], a hypertext system used to support searching software libraries. Kiosk was originally developed as a single-process, single-user system, and changing it into a distributed application will involve some major design changes.

Throughout this paper we will be using Kiosk as the driving example to illustrate the use of Bart. In Kiosk, a hypertext document is a set of nodes connected by links. Each node contains text, and each link connects some point in the text of one node to a point in the text of another node. Each

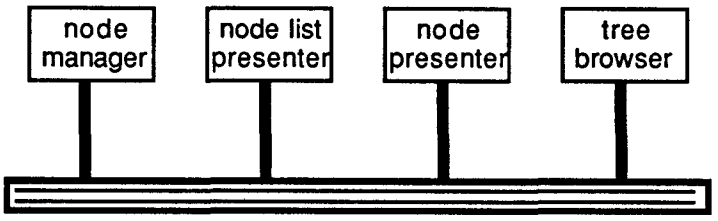

Figure 3: Hypertext components.

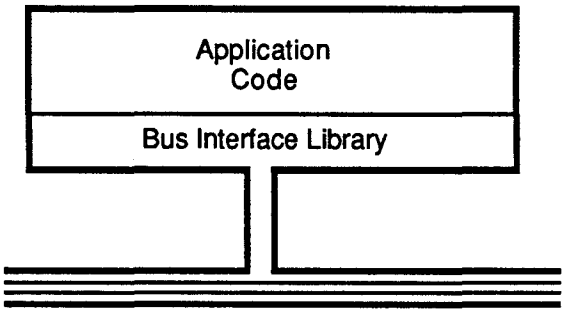

Figure 4: Bus Interface

endpoint of a link is a reference to a node and an offset in the text of that node. These structures have a natural representation as relations, shown in figure 2. Note than the from and to field of links are references to nodes.

The hypertext system, Kiosk, is divided into four components, as shown in Figure 3. The node manager maintains the repository of links and nodes. The other three components are presenters; they present information to the user and allow it to be edited. The node manager exports the node and link relations for use by the presenters.

The first presenter that we will look at is the node list presenter. It simply shows a list of all nodes to the user, for which it needs the contents of the node relation. When the node manager and the node list presenter are connected to the bus, the bus gives the node list presenter the contents of the node relation, and keeps the presenter up to date as the relation changes. The job of the presenter, then, is to present the information it is given.

4.1 Object Management

The messages regarding changes to relations, and the relations themselves are invisible to the application programmer, who sees only objects inside the component that s/he constructs. A bus interface layer is provided to insulate components from the details of interacting with the bus (see Figure 4).

Though relations are an excellent way to model the data shared by a number of components, they are not the way most components are built internally. Objects are a more convenient way to represent data inside a component, so some way is needed to connect relations and views to objects. Barsalou and Wiederhold have explored the connection between objects in a running program and tuples in a relational database [BW86]. We use a similar approach on both ends of a connection between two components. Objects in one component are mapped into tuples on the bus and then back into objects in another component.

```
class node { ... };

node::node() {
  // initialize the object
  port_register_object( node_port, this, my_id, my_name, my_text );
};

void node::set_text( ... ) {
  // update the text in the node
  port_register_update( this, "TEXT", my_text );
};

node::~node() {
  port_unregister_object( this );
  // cleanup
};
```

Figure 5: Publishing data

To do this, the bus interface layer establishes a one-to-one relationship between the tuples in a relation and objects inside a component. There is not room to go into the details here, but a brief outline is given below.

4.2 Local Objects

Components that export information to the outside world have what are called *local objects*. These objects are owned by the component and exported for use by other components.

Components with local objects are responsible for telling the bus when they create new objects, modify objects, or destroy objects so that the bus can transform them into relational form and transmit them to other components. They are also responsible for creating, updating, and destroying objects when requested to do so by the bus.

In our hypertext example, the node manager component is responsible for maintaining the entire collection of nodes and links, which are stored internally as C++ objects. The bus interface library maintains the mapping between the C++ objects and the tuples in the relation on the bus.

A *data port* takes care of the connection between one object class and one relation on the bus. The connection to the bus for nodes is illustrated in the code shown in Figure 5. When a new object is created, it is registered with the data port, which sends out an announcement on the bus that a tuple has been added. When a node is modified, the change is registered with the data port. And, when a node is destroyed, the port is informed so that it can broadcast an announcement.

The external appearance of an object as a tuple in a relation need not match its internal data representation. While some of the values in the tuple may be represented by the object directly as instance variables, others may be computed from instance variables or attributes of related objects. All that is required of an object is that it notify the bus of its existence, and keep the bus apprised of any changes to its externally visible attributes, however they might be obtained.

```
main()
{
  /// ...
  node_port = create_remote_data_port( "NODE", ..., create_shadow_node, ... );
}

create_shadow_node( int id, char* name, char* text )
{
  /// build the shadow object.
}
```

Figure 6: Subscribing to data

4.3 Remote Objects

Components that import information have *remote* objects. These objects are managed by the bus interface, and are *shadows* of objects exported by some other component.

The node list presenter from our hypertext example needs to import information about nodes. To do this it creates a data port connected to the node port in the node manager, and provides callback functions that manipulate shadow objects. When nodes are created in the node manager, the bus interface will invoke the callback function to create shadow nodes in the node list presenter. Figure 6 shows an outline of the code.

The bus maintains the relationships between the nodes, tuples, and shadow nodes so that when a node is updated, the corresponding shadow object is notified.

5 Glue

Software Glue is the key to Bart's ability to maintain component independence. The most sensible way to organize data may be different in each component. It is the job of the Software Glue to bridge the gap between these different data models and allow the components to communicate effectively with each other. The relational layer of the bus can form a simple connection between two object classes, represented on the bus by two relations, that have the same schema. The glue layer provides much more power by supporting much more complex connections, involving any number of objects.

Our glue language is a logical language similar to Prolog. It expresses *derived* relations in terms of *base* relations. For example, if there are two relations, `a` and `b`, exported by two of the components connected to the bus, a third relation `c` can be defined to include those things that are in both `a` and `b` using the following glue statement:

```
c(X) <- a(X) and b(X);
```

Glue can be used to do many different things. The following sections describe some of its more common uses: renaming relations, filtering data, summarizing data and collecting data.

5.1 Renaming

In its simplest form, a glue statement can rename a relation, as in:

```
foo(X) <- bar(X);
```

Renaming is useful when two components agree on the structure of data but not on what to call it.

```
relation np-interest is
        component   pres      key,
        integer     node-id;

relation np-text is
        component   pres      key,
        string      text;

relation np-inlink is
        component   pres      key,
        integer     link-id   key,
        string      label;

relation np-outlink is
        component   pres      key,
        integer     link-id   key,
        string      label;
```

Figure 7: Schema for the node presenter.

```
np-text( Presenter, Text ) <-
        np-interest( Presenter, Node ) and node( Node, ?, Text );

np-outlink( Presenter, Link, Label ) <-
        np-interest( Presenter, Node ) and link( Link, Node, ?, ?, Label );

np-inlink( Presenter, Link, Label ) <-
        np-interest( Presenter, Node ) and link( Link, ?, Node, ?, Label );
```

Figure 8: Glue for the node presenter.

5.2 Filtering

In Kiosk, the *node presenter* puts a window on the screen showing the text contained in one node and a list of the links coming into and going out of the node. It exports one piece of data, the identifier of the node that it is interested in, and imports the data concerning that node. The schema that the node presenter sees is shown in figure 7. Note that the presenter interested in the information (pres) is explicitly named in the schema. This means that there can be several different node presenters active at one time, and that each can have its own view of the data.

The np-interest relation is exported by the node presenter and names the node it wants to see. Software Glue is used to define the contents of the other three relations, based on np-interest and on the node and link relations exported by the node manager. The glue acts as a *filter* to extract only the node and link information required by the node presenter and ignore the rest; it is shown in Figure 8.

The first glue statement takes the identifier of the node to be viewed from the node presenter, which it exports via the np-interest relation, and uses it to look up the text for that node in the node relation. The resulting text is then placed in the np-text relation to be imported by the node presenter. The second and third statements select the links entering and leaving the node and extract their labels.

```
relation tb-interest is
        component   pres      key,
        node        node-id   key;

relation tb-node is
        component   pres      key,
        node        node-id   key,
        string      name;

relation tb-link is
        component   pres      key,
        node        from      required,
        node        to        required;

tb-node( Presenter, NodeID, Name ) <-
        tb-interest( Presenter, NodeID ) and node( NodeID, Name, ? );

tb-link( Presenter, From, To ) <-
        unique( tb-interest( Presenter, From ) and
                link( ?, From, To, ?, ? ) and
                tb-interest( Presenter, To ) );
```

Figure 9: Schema and glue for tree browser

5.3 Summarizing

The last presenter we will look at is the tree browser, which shows a graphical representation of a collection of nodes and their interrelationships. This presentation is a summarization of the connections between the nodes shown. Even if there is more than one link between a pair of nodes shown on the screen, a single line is drawn between them.

As with the node presenter, the tree browser exports a relation containing the set of nodes it is interested in, in this case using the tb-interest relation. While the node presenter was interested in only one, the tree browser is interested in an arbitrary number. The tree browser needs to show node names and their interconnections, so it imports this information via the tb-node and tb-link relations. The schema and glue for the tree browser are shown in Figure 9.

The first glue statement selects the nodes of interest and finds their names. The second glue statement is the interesting one. It finds all of the links between the nodes of interest and then *summarizes* the results, finding all of the unique combinations for (Presenter,From,To). Because the glue expresses this summarization, the bus manager can assign this task to be done in the node manager process, greatly reducing the amount of information that must be transmitted across the bus. This is an example where the performance benefits of a customization to the node manager can be had without making any changes to it.

5.4 Collecting

A use of glue which is not illustrated by the hypertext example, because all of the information about nodes and links is stored in the same place, is collecting information stored in various places. As a hypothetical example, suppose that the node manager were split into two pieces, one of which stores the names of the nodes, while the other stores the text for each node. The first would export a relation node-name and the second would export a relation node-text. The information in these two relations can be collected together into the one node relation using this glue statement:

```
node( ID, Name, Text ) <- node-name( ID, Name ) and node-text( ID, Text ).
```

This is another example of component independence in action. If we happen to have the presenters discussed above that expect nodes to have both a name and text, we can combine these two separate sources of information and use the presenters as-is.

5.5 Updating

The views described by the Software Glue can be modified as well as inspected. The techniques used for updating views in relational databases [Furt84,Kell85] have been used in Bart to allow updates to derived relations. In the Kiosk example, the various presenters that show information to the user can let the user modify the information, and then store the changes as updates to their views. Bart takes care of translating these updates into updates on the underlying relations stored by the node manager.

6 Related Work

Most proposals and existing systems for connecting components in a distributed environment are based on control integration rather than data integration. Components communicate by sending messages, with interfaces defined to say which messages and parameter types are allowed. In distributed systems, these interfaces are explicitly stated in an interface description language [OMG91a, OMG91b].

Some systems have taken a step toward component independence by introducing a mediator between the two components being connected [Sull90]. Contracts have been proposed for single-address-space systems [Helm90]. ANSA [ANSA91] uses a *trader* to provide anonymity between components. A request is sent to the trader, who knows which components can do what; the trader then forwards the request to the component that knows how to handle it. The "agent-based software engineering" [Sho90, Gen90] paradigm places an *agent* between the components to mediate between them.

By contrast, Bart is unique in its ability to support data integration. Components need not send messages to request needed information; Software Glue is used to describe what information is needed, and provide it when needed.

Database systems have used rules for a variety of purposes [Stone90], including constraint maintenance, views, security, and referential integrity. The rules in Bart's Software Glue are used to define views. Some database systems have used object-oriented interface specifications to connect heterogeneous databases [Bert89].

Cox has proposed [Cox91] a hierarchy of interconnection mechanisms for software, ranging from variables and assignment statements up through processes and streams. We are proposing a more declarative mechanism for interconnection, where the components interact through shared data rather than streams of values. This facilitates reuse by leaving the interaction style open until two components are connected together, rather than building it into the components.

AP5 [Cohen85] is the language used in the FSD system to glue things together. As with our glue language, it expresses the relationships in data in a declarative way. In AP5, though, the relationships are represented as constraints, and the programmer must write down explicit instructions on what to do when the constraints are violated.

The notion of a *policy*, proposed by Garlan [Garl90], is a form of Software Glue for control integration. The procedural interfaces to components are defined, and then the glue is used to connect them. This is similar to our approach for data integration.

7 Status and Future Work

Bart is written in C++ and can connect components written in C, C++, and Xlisp. It is currently being used to support a Physician's Workstation [Tang91], and work has begun in changing Kiosk from a single-process application to a multi-process distributed application running on Bart. We will be closely examining the performance characteristics of Bart, and how it supports these two applications[2].

The original version of the Physician's Workstation (PWS) was built on top of HP's Broadcast Message Server; we ported it to run on Bart. Initial indications are that the performance of the two platforms is comparable. Once the work on Kiosk is well under way we will start investigating one or two other systems, looking for different ways to stress Software Glue.

We have encountered some rather mundane problems in connecting different applications to the software bus. One of the more persistent is the problem of "who is in charge". The X11 Motif toolkit insists that applications use its top-level loop (`XtMainLoop`), and InterViews, used by Kiosk, has a similar requirement. This means that Bart must be adaptable enough to exist with many different top-level loops. The fact that Bart is multi-lingual exacerbates the problem; many Lisp implementations also insist on using their own top-level loop.

These problems with top-level loops are what might be called *micro-engineering* issues: how to design components so that they fit into the bus architecture. These contrast with the *macro-engineering* issues of building a complete application from large-scale components. We plan to study exactly how much change is required to take an existing software component and package it so that it works with Bart.

The current version of Software Glue does not support negation or recursion. We plan to correct this deficiency, although it will still be necessary to have stratification requirements on the rules, disallowing recursion through negation.

As Bart evolves we want to incorporate it into a complete module interconnection language (MIL) [Prie86]. Bart provides data independence between components, but does not have any high-level mechanism for describing control structure. CDL [OMG91a] is an interface specification language that provides enough information to automatically generate C++ stubs for accessing remote facilities. Bart could benefit from this approach, allowing C++ classes to be connected directly to the bus without programmer intervention.

The relational approach may make it simpler to connect relational databases into applications. The fact that all information is transmitted in a relational form means that a client would not need to know whether the information is being provided by a database or by another component. Thus, the database can be just another component; it needs no special connection mechanism.

Our approach so far has been inspired by a relational view of the world. Object-oriented database technology may prove to be even better at providing data independence between object-oriented software components [Lor84].

One of the initial goals of Bart was to support a variety of different implementation strategies for implementing Software Glue. So far, only one has been built, based on sending messages across socket connections where each component is connected via a socket to the bus manager. In the future it would be nice to extend Bart to include more communication media, including direct

---

[2] The results of this analysis will be available in the final version of this paper.

socket connections between components that communicate a lot, and shared memory for components executing on the same machine.

8 Summary

We have described the software bus Bart, and the three layers composing it: message transport, relations, and Software Glue. Bart allows components in a distributed environment, written in different languages, to be easily connected using high-level Software Glue that bridges the gap between different data models. The two key ideas are: (1) the mapping between an object-oriented view inside software components and a relational view on the bus, and (2) the use of a high-level declarative glue language to define the sometimes complex connections between components. Bart is being tested on two medium-scale software projects at HP labs.

Even the early versions of Bart, with very simple Software Glue, have proven to be very effective at connecting software components. We expect that the full implementation will be useful in a wide range of applications.

Acknowledgements

Martin Griss has been my mentor for many years, providing innumerable insights. Many thanks to Mark McAuliffe, for putting up with early implementations and for pointing out better ways to use the bus. Kevin Wentzel, Jon Gustafson, and Mark McAuliffe have participated in many lengthy discussions mapping out the territory of software buses. Finally, Mike Creech and Mark Gisi shared their knowledge of Kiosk and provided many useful ideas.

References

[ANSA91] "ANSAware 3.0 Implementation Manual." Document RM.097.00, Architecture Projects Management Limited, Cambridge, 1991.

[Bert89] Bertino, Negri, Relagatti and Sbatella. "Integration of Heterogeneous Database Applications Through and Object-Oriented Interface." *Information Systems*, 14(5), 1989, pp. 407–420.

[BW86] Barsalou, T. and G. Wiederhold. "Complex Objects for Relational Databases." *Computer Aided Design*, 22(8) October 1990.

[Cagan90] Cagan, M. "The HP Softbench Environment: An Architecture for a New Generation of Software Tools." *Hewlett-Packard Journal*, June 1990, pp 36–47.

[CFG91] Creech, M. and D. Freeze and M. Griss. "Using Hypertext In Selecting Reusable Software Components." Accepted for publication in the *Third ACM Conference on Hypertext.*

[Cohen85] Cohen, D. "AP5 Manual". ISI technical report, 1985.

[Cox91] Cox, B. "TaskMaster." Stepstone technical report, 1991.

[Furt84] Furtado, A. "Updating Relational Views." In *Query Processing in Database Systems*, Springer-Verlag, 1984.

[Garl90] Garlan, D. and E. Ilias. "Low-cost, Adaptable Tool Integration Policies for Integrated Environments." In *Proceedings of ACM SIGSOFT90: Fourth Symposium on Software Development Environments*, pp. 1-10, December 1990.

[Gen91] Genesereth, M. *Knowledge Interchange Format Version 2.1 Reference Manual.* Stanford University Computer Science Department report Logic-90-4.

[Helm90] Helm, R. and I. Holland and D. Gangopadhyay. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", SIGPLAN Notices 25(10), 1990, pp. 169-180 (OOPSLA/ECOOP '90).

[Kell85] Keller, A. *Updating Relational Databases Through Views.* Ph. D. thesis, Stanford University, 1985.

[Lor84] Lorie, R. "Supporting Complex Objects in a Relational System for Engineering Databases." In *Query Processing in Database System*, Springer-Verlag, 1984, pp. 145–155.

[OMG91a] Hewlett–Packard Company and Sun Microsystems, Inc. "Class Declaration Language Specification." OMG report 91.1.4.9.

[OMG91b] Hewlett–Packard Company and Sun Microsystems, Inc. "Distributed Object Management Facility Core Specification." OMG report 91.1.4.10.

[Prie86] Prieto-Diaz, R. "Module Interconnection Languages." *Journal of Systems and Software*, 6(4) 1986, pp. 307–334.

[Purt91a] Purtilo, J. "Software Bus Organization: Reference Model and Comparison of Existing Systems." Draft, April 1991.

[Purt91b] Purtilo, J. "The Polylith Software Bus." To appear in *acm Transactions on Programming Languages and Systems.*

[Reiss90] Reiss, S. "Connecting Tools Using Message Passing in the Field Environment." IEEE Software 7(4), July 1990, pp. 57-66.

[Sho90] Shoham, Y. "Agent-Oriented Programming." Stanford University Computer Science Department report STAN-CS-90-1335.

[Stone90] Stonebraker, M. "The POSTGRES Rule Manager." *IEEE Transactions on Knowledge and Data Engineering*, 2(1) March 1990, pp. 125–142.

[Sull90] Sullivan, K. and D. Notkin. "Reconciling Environment Integration and Component Independence." In *Proceedings of ACM SIGSOFT90: Fourth Symposium on Software Development Environments*, pp. 22-33, December 1990.

[Tang91] Tang, P., J. Annevelink, D. Fafchamps, W. Stanton, and C. Young. "Physician Workstations: Integrated Information Management for Clinicians." Accepted for the Symposium on Computer Applications in Medical Care, 1991.