14.0 A List of Possible Software Architecture Styles

- layered
- main program and subroutines
- pipe and filter
- communicating/distributed processes
- event-based (implicit invocation) systems
- object-oriented (client/server)
- software bus
- multithreaded systems
- rule-based (expert) systems
- transactional database systems
- blackboard systems
- logic programming
- real-time systems
- feedback systems
- (domain specific styles)
% there is still room in the car
passengers' = passengers union {newp?}
% the postoperation set of passengers is the preoperation union with the
set of (a single) passenger
end schema

In general, inputs end with a ‘?’ , outputs with a ‘!’ , and postoperation state attributes with an apostrophe. Preoperation attributes do not end with anything special.
axiom
  population > 100
  population < 6000000000
end axiom

The list of axioms is an implicit conjunction.

Schemas

The basic unit for capturing state in Z is the schema. Schemas have two parts: a signature which captures attributes, and a property which captures constraints about the attributes in the signature.

```z
schema Car
  type : CarType
  passengers : P Name
  wheels : N1
where
  # passengers < 6
  % the cardinality of the set passengers is less than 6
  wheels <= 4
end schema
```

The list of constraints in the property are implicitly part of a conjunction.

Schema Operations

Schemas can be used to describe operations which affect the state of the attributes, and which can also take inputs and outputs.

```z
schema AddPassenger
  Delta Car
  % the Car schema is changing
newp? : Passenger
  % an input passenger
currentpassengers! : N1
  % output number of current passengers after the operation
where
  newp? notin passengers
  % new passenger not in the set of current passengers
  # passengers < 5
```

138
Types

We can use given sets in declarations of variables to act as the type of the variable. Types can also be drawn from other Z constructs (e.g. schemas).

```plaintext
nameofsystem : Name
population : N
```

Building Sets

We sometimes want to build sets out of other types.

```plaintext
names : P Name
% a set of names

nameandages : P (Name cross Z)
% a set of binary tuples - name with integers

nameandages2 : Name <-> Z
% the same as the previous: it is a relation

humantoage : Name --> Z
% a relation which is also a function
```

Predicates (Constraints)

First order predicates are supported in Z.

```plaintext
forall n : N1 | n in setofbignumbers @
  n > 1000
% some set of big numbers all have numbers greater than 1000

exists min : N1 | min in setofbignumbers @
  forall n : setofbignumbers @ min <= n
% the set of big numbers has a minimum
```

Declaring Variables

When we want to declare a variable, we must specify its type and some optional constraints (also known as axioms in this case).

```plaintext
global
  population : Z
```
In this appendix we present a very brief introduction to Z and ZSL. Z is a formal notation for specifying software systems, but it is general enough to model many other types of systems. It is based on first order logic and set theory. It uses certain graphical constructs as part of the notation. For this reason, ZSL was invented: it is a purely textual form of Z. ZSL does not add anything to Z beyond an ASCII interface.

For a deeper introduction to Z, there are many good references: [SPIV92, WORD92, POTT91].

Given Sets

Sometimes it is not necessary to give a detailed specification of items that we are trying to model. In that case, the items can be captured by given sets.

[Name, BasicUnit, Book, Human]

Z comes with some built-in given sets, namely the integers, natural numbers, and whole numbers.

[Z, N, N1]
% integers, natural numbers, whole numbers respectively

To represent set membership:

3 in N
-3 notin N1


[SPIV92] J. Spivey. The Z Notation, Prentice Hall International, 1992


12.0 References


Part IV: References and Appendices

A list of references used for this thesis is given, as well as appendices describing the formal notation used earlier and a list of possible software architectural styles.
• *Generate code.* Another long-term extension would be to generate skeleton code from the architectural specifications that are based on this model. Similar to UNAS, the specification could be used to generate an executable system that could be operated and experimented with.

### 11.3 Extensions to the tool and general approach

• *Evaluate more case studies.* An effective way to improve the tool and general approach is to apply them to existing large software systems, and evaluate how well they are able to predict actual mismatches that occurred.

• *Develop an ADL.* The specification of our model as it stands acts as the semantics for an architecture description language. This ADL would go beyond the simple notation currently used for the Automated Architect’s Assistant. Notably, it would support the expression of user-defined constraints.

• *Extend AAA.* There are many ways to extend the tool: provide a graphical front end, allow user-defined styles, allow on-the-fly editing of entity attributes, provide saving and loading capabilities, port it to different platforms, etc.
example, an architect may be interested in the performance or reliability of a composition.

- *Allow additional interactions/connectors.* The model currently supports only a limited set of possible interactions/connectors between subsystems during composition (the group operation). Additional interactions are extremely important to consider, e.g. dynamic declaration of data, import/export dependencies, etc. The nature of the interactions will greatly depend on the supported views.

- *Identify additional mismatches.* If the underlying model of styles is extended to cover other styles (and consequently introducing one or more conceptual features), the composition operations will need to be reexamined for the possibility of more mismatches. Another source of new mismatches will be any new allowed interactions that are introduced to the model. Yet another source will come from the addition of new supported views (e.g. a system’s poor reliability may cause a composition to fail due to imposed high reliability constraints).

- *Allow removal during composition.* Currently the model only supports the addition of components and connectors during composition. There are cases when an architect may wish to specify that certain components or connectors need to be removed before a composition may succeed.

- *Propose solutions.* A long-term extension would be to associate a solution with each mismatch generated during composition (e.g. protocol translators, synchronization mechanisms, resource sharing schemes, etc.).
• *Reflect additional views.* The model currently is focused to varying degrees on the structural, topological, behavioral, and environmental views. However, there are other views of a software architecture that are important to capture as well, e.g. performance, reliability, cost, etc. An interesting subproblem here will be how well the choice of formalism (Z) holds up to the new views.

• *Extend the existing views.* Our model is essentially a static one, and does not fully represent behavioral attributes and constraints (e.g. ordering of operations). Adding additional constructs for behavior is an important extension to the model. There are many existing models of behavior; it is a matter of applying them to this domain of architectures and architectural styles. One immediate subproblem will be to identify what are the primary ‘architectural actions’ of the entities in the styles. For example, a main/subroutine procedure may engage in procedure calls and returns, while a filter can engage in sending and receiving streams of data.

• *Remove artificial constraints.* The model currently disallows spawning of object methods. This constraint will need to be removed eventually - unless it is justifiable to ignore this behavior as atypical and therefore keep the constraint.

### 11.2 Extensions to the model of architectural composition

• *Support additional views.* The group operation needs to be extended to support any additional views which are added to the underlying model of styles. For
11.0 Future Extensions

There are many remaining extensions to our models that would be very useful. They can be broken down into three categories: those that are related to the underlying model of styles (section 6.0), those that are related to the model of architectural composition (section 7.0), and those that are related to the tool (section 8.0) and approach in general.

11.1 Extensions to the underlying model of styles

- *Capture more styles.* Our model (and the set of conceptual features in particular) can adequately distinguish between a good number of styles - but not all. It can distinguish systems that are in the following styles: layered, main/subroutine, pipe & filter, distributed processes, event-based (implicit invocation), object-oriented, software bus, and multithreaded systems. However, there are other styles which cannot be completely distinguished, and these include: database, blackboard, logic programming, rule-based, real-time, and feedback systems. The model must be extended to encompass these styles.

- *Determine other conceptual features.* This extension is closely related to the extension of capturing more styles. If the model is extended to distinguish more and more styles, it will necessarily introduce at least one more conceptual feature that will be the main determining constraint for identifying the new styles.
• the model identifies architecture mismatches of heterogeneous systems, system-subsystem and system-system

• a general framework for rapidly incorporating new styles into the composition operations is identified

• The two models are the basis for an CASE tool which allows architects to analyze specifications of heterogeneous architectures for architecture mismatches. In particular,

• the tool supports many different styles, and their composition into heterogeneous systems

• the tool is extensible to include additional style and COTS descriptions by someone familiar with the tool program internals

• the tool has successfully been used to analyze the architecture of another CASE tool which suffered from certain architecture mismatches
10.0 Summary of Key Contributions

- A model of several styles has been developed, and it has enabled us to draw some insights on how styles are similar and how they differ. In particular,
  - the formal specification of the model provides an extensible framework for supporting different views of architectures
  - the difference between static and dynamic structures is highlighted
  - the model identifies a core set of conceptual entities that are the basis for a uniform representation of many different styles
  - commercial-off-the-shelf systems are captured as very specific styles by the model
  - a space of architectural styles is described which rests on a set of critical attributes and constraints, the conceptual features, and which eliminates much of the complexity associated with comparing and composing different styles
- A model of composition based on architectural styles has been developed, allowing architects to systematically specify and analyze heterogeneous architectures. In particular,
  - the wide scope of composition is related to the problem of enumerating different interaction types between systems
  - a disciplined approach to composing heterogeneous architectures is identified, and it is based on a single operation (group)
4. System dependencies for building Aesop out of four COTS packages were tangled. This required manually aiding a supposedly automated build process. The underlying model of AAA only looks at a subset of the possible interactions between systems, and one pair of interactions that is not currently modeled is the notion of imports/exports. Build dependencies add to the complexity of the composition operation.
grow rapidly (in this case, 3 control components result in 3.6 Mb of binary code).

There are mismatches that the developers reported that AAA is unable to capture. For each one, we can try to identify why AAA does not catch it:

1. *Softbench does not provide blocking on request-reply event pairs.* MIG was partially introduced to provide this procedure call semantics. This type of mismatch requires a stronger semantic model of events which assigns meanings to different events. Currently, the underlying model behind AAA does not support this.

2. *Softbench and InterViews both assume they are the ‘main controllers’ in the systems which are built out of them.* Hence, both expect to control the flow of events through Aesop. The developers had to integrate the two event loops together to achieve a single integrated tool. This mismatch slips by AAA because of a flaw in the model of event-based systems: it abstracts the event manager away entirely. A future model may explicitly highlight it.

3. *InterViews does not allow independent manipulation of window children.* While InterViews does provide hierarchical data structures for windows and their children, it does not allow independent manipulation of the children. The Aesop developers introduced duplicate data structures to achieve this functionality. AAA does not catch this mismatch because it is extremely specific, almost not an *architecture* mismatch.
that Softbench does (X Window Xlib versus XIntrinsic events), and a similar change was planned for MIG but time ran out. AAA is able to catch this mismatch of event vocabularies.

3. **MIG event data is expressed in C structs/arrays, and Softbench data is expressed in ASCII strings.** To allow communication between these two tools, the Aesop developers had to build a real-time translator between the two. The basic set of possible data formats is captured in AAA, so this mismatch is found by simply checking for compatibility.

4. **OBST requires a large library of standard object classes to be installed even if not needed.** This mismatch is harder to catch because of the ambiguity of determining how large is too large. However, since it is possible to know beforehand how large the OBST repository might be on average, an architect can hardwire that information into the ‘OBST style’ description in AAA, and future uses of OBST will report the expected size. This is precisely what is captured here (400K of shared data - regardless of the size of the project, this is a hypothetical minimum bound).

5. **Softbench requires all attached components to link in X Window library. This mismatch leads to inordinately large binaries.** An X Window application typically takes more than 1 Mb of space due to the size of the libraries. This information can be hardwired into the AAA description of Softbench, and an architect can quickly see that the size of an application built in Softbench will
9.2.2 Analysis of Aesop specification with AAA

We passed the specification through the Architect’s Automated Assistant, and it detected several potential problems with this architecture (figure 23).

![Aesop Architecture Mismatches detected by AAA](image)

The five mismatches that it detected are all reported by the developers:

1. **OBST is unable to provide concurrent access to the same piece of data.** This forced the Aesop developers to implement their own transaction manager because their tool demanded collaborating sub-tools which could conceivably share a piece of the database concurrently. AAA captures this mismatch by noting that at least two concurrent threads may access the data component representing OBST.

2. **Softbench, InterViews, and MIG have incompatible event sets.** The developers report that they had to modify InterViews to recognize the same set of events
system aesop_softbench_subsystem
  initialcontrolcomponents : aesop_process1, aesop_process2;
  controlcomponents : aesop_process1, aesop_process2, aesop_process3;
  triggers : start_aesop_process3;
end;
end;

style interviews
  controlcomponent aesop_window1
  end;

  controlcomponent aesop_window2
  end;

system aesop_interviews_subsystem
  initialcontrolcomponents : aesop_window1, aesop_window2;
  controlcomponents : aesop_window1, aesop_window2;
end;
end;

style base
  group aesop
    subsystems : aesop_mach_subsystem, aesop_softbench_subsystem, aesop_interviews_subsystem;
    extendedshareddata : [aesop_database, [aesop_database_interface1, aesop_database_interface2], []];
  end;
end;
end;

style mach
    event machmsgtype_rpc
    end;

    event machmsgtype_encrypted
    end;

    event machmsgtype_normal
    end;

controlcomponent aesop_database_interface1
end;

controlcomponent aesop_database_interface2
end;

system aesop_mach_subsystem
    initialcontrolcomponents : aesop_database_interface1,
aesop_database_interface2;
    controlcomponents : aesop_database_interface1,
aesop_database_interface2;
end;
end;

style softbench
    event xevent_keypress
    end;

    event xevent_keyrelease
    end;

    event xevent_buttonpress
    end;

controlcomponent aesop_process1
end;

controlcomponent aesop_process2
end;

controlcomponent aesop_process3
end;

trigger start_aesop_process3
    controlcomponent : aesop_process2;
inmessage : xevent_buttonpress;
outcalls : aesop_process3;
end;
AAA only by someone who understands Prolog and Z, however this step need be done only once for each new additional style or COTS package.

9.2.1 Simple Specification of Aesop

The precise design of Aesop is not documented in the two reports by the developers ([GARL94], [GARL95]). However, there is enough detail to reconstruct an oversimplified specification of Aesop which still captures the critical architectural decision of composing four different tools. Considering that architectural composition is more concerned with the interfaces than with the subsystems behind the interfaces ([RECH91]), our specification is a fair depiction of Aesop’s gross design. The one major weakness in the specification stems from our model’s inability to clearly represent certain styles: OBST, the central repository, is specified by a single data component (which is shared by the other systems). This obviously excludes major architectural details associated with databases, but this style is not well-supported currently by our model.

The specification is given below (see section 8.1 for more information about the notation). It declares three systems (based on Softbench, MIG, and InterViews) and a data component for capturing OBST. These four entities are then passed to a ‘group’ operation to arrive at Aesop. This specification, which may seem extremely high-level, contains enough information to analyze it for mismatches.

```prolog
style obst
  datacomponent aesop_database
end;
```
- Softbench, an event-based tool integrator from Hewlett-Packard
- Mach RPC interface generator (MIG), an event-based remote procedure call package from Carnegie Mellon University

InterViews was used to develop the graphical front end of Aesop, Softbench to integrate the tools with each other (including possible future tools), OBST to act as the repository, and MIG to provide blocking semantics into the repository.

It was more difficult to build Aesop than anticipated. The developers anticipated six months and one person year, but the actual results were two years and five person years. Six general problem areas were identified:

- excessive code size
- poor performance
- need to modify COTS packages
- need to reinvent existing function
- unnecessarily complicated tools
- error-prone construction process

Specific architecture mismatches were provided by the developers as instances of the general problem areas. Our model (and tool) was used to identify some of these mismatches when a simple specification of Aesop (which we generated) was fed into it. In order to do this, we entered partial descriptions of the four COTS packages described above into the repository of the Architect’s Automated Assistant. As mentioned earlier (section 8.2), this information can be added to
9.2 Identifying Architecture Mismatches: Aesop Case Study

Our model (and tool) can be used to identify architecture mismatches that may occur when the outputs of different COTS packages are composed with each other to produce an application. Each COTS package can be modeled as an instance of a very specific style (section 6.2.6). We present here an example of such an application: Aesop. Aesop is a CASE tool developed at Carnegie Mellon University for building architectural design environments ([GARL94]). After undergoing a harder and longer than expected process of building Aesop, the developers published an insightful report on the problems they encountered, with an analysis that traced many of the problems to architecture mismatches ([GARL95]).

Aesop was built out of four different COTS packages (figure 22):

- OBST, an object-oriented database from Forschungszentrum Informatik
- InterViews, an event-based GUI builder from Stanford
pared four styles with each other in terms of the conceptual features, but with the addition of a column for UNAS-generated systems. The table clearly indicates part of the reason behind the success of UNAS: *the systems it can generate contain a mix of capabilities found in several styles*. With respect to pipe and filter systems, UNAS systems appear to be their superset (however we note that one cannot specify streaming data connectors in UNAS). Main/subroutine systems are hard to model in UNAS because there are no control connectors beyond the callbacks, and shared data is not supported. Distributed processes systems are very close to UNAS systems, but they add spawns and remove triggering. Finally, UNAS systems have the advantage over event-based systems of allowing concurrency, but event-based systems have the advantage in terms of the supported data transfer mechanisms and encapsulation.

UNAS only provides support for the group composition operation. Wrapping entire systems with tasks or procedures is not supported. This limits the tool to creating monolithic systems whose tasks cannot be hidden or abstracted away.

<table>
<thead>
<tr>
<th></th>
<th>Pipe &amp; Filter</th>
<th>Main/Subroutine</th>
<th>Distributed Processes</th>
<th>Event-Based</th>
<th>UNAS systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 9: Four Instances in an Architectural Style Space
does not change over time (see section 4.2.2.1). This is in effect a simplifying assumption for UNAS systems, with several benefits:

- reduces the number of connectors an architect needs to worry about: the system revolves around data connectors primarily
- eases analysis against resource usage constraints
- allows a simple failure model (“if this task fails, bring up another copy”)
- simplifies the graphical user interface and the underlying code necessary to describe and simulate systems

Even with this simplifying assumption, however, UNAS still manages to support more than one style - though to differing extents.

We can use the conceptual features (section 6.3) to evaluate how well UNAS supports multiple styles. In table 9 below, we duplicate table 6 which com-

<table>
<thead>
<tr>
<th>Pipe &amp; Filter</th>
<th>Main/ Subroutine</th>
<th>Distributed Processes</th>
<th>Event-Based</th>
<th>UNAS systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamism</td>
<td>static</td>
<td>dynamic</td>
<td>static</td>
<td>static</td>
</tr>
<tr>
<td>Supported data transfers</td>
<td>explicit data connectors</td>
<td>shared data variables</td>
<td>explicit data connectors</td>
<td>implicit network, shared data variables</td>
</tr>
<tr>
<td>Triggering capability</td>
<td>no</td>
<td>N/A</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Concurrency</td>
<td>multi-threaded</td>
<td>single-threaded</td>
<td>multi-threaded</td>
<td>single-threaded</td>
</tr>
<tr>
<td>Distribution</td>
<td>unconstrained</td>
<td>single node</td>
<td>multiple nodes</td>
<td>unconstrained</td>
</tr>
<tr>
<td>Layering</td>
<td>unconstrained</td>
<td>unconstrained</td>
<td>unconstrained</td>
<td>unconstrained</td>
</tr>
</tbody>
</table>

Table 9: Four Instances in an Architectural Style Space
t.inmessage in UNAS_Message and
t.outmessageset subseteq UNAS_Message and
t.outcalls subseteq UNAS_Procedure and
t.subtype = ctype and
  % limit reception of messages to tasks (control components)
t.outspawns = {};
  % no spawns allowed to be triggered
forall s : UNAS_System @
  s.dataconnectors subseteq UNAS_Circuit and
  s.controlcomponents subseteq UNAS_Task setunion UNAS_Procedure and
  % control components are either tasks or procedures (callbacks)
s.recognizedmessages subseteq UNAS_Message and
s.triggers subseteq UNAS_Trigger and
s.initialcontrolcomponents =
  {c : UNAS_Task | c in s.controlcomponents} and
  % all tasks are running at startup, no callbacks
s.initialdataconnectors = s.dataconnectors and
  % all circuits are running at startup
(forall c : s.dataconnectors @
  exists t1, t2 : s.controlcomponents @
    c.p1 in t1.ports and c.p2 in t2.ports) and
  % all circuit sockets connect system task sockets
(forall c1, c2 : s.dataconnectors @
  c1 /= c2 =>
    (not (c1.p1 = c2.p1 and c1.p2 = c2.p2)) and
    (not (c1.p1 = c2.p2 and c1.p2 = c2.p1)) and
  % no two circuits connect the same pair of sockets
s.globalobjects = {} and
s.classes = {} and
s.calls = {} and
  % SALE only allows specification of callbacks *not* general calls
s.spawns = {} and
s.shareddata = {} and
s.call_layers = {} and
s.spawn_layers = {} and
s.dataconnector_layers = {} and
s.triggers /= {};
  % UNAS allows callbacks off of message receipt
end axiom

Beyond the triggered callbacks, there are no other control connectors in
UNAS systems. The set of tasks in the system are assumed to execute concurrently
for the duration of the system: from startup to shutdown. The advantage of this
approach is that at this level of abstraction, the system is very static: the topology
limited ourselves to modeling systems built using the graphical frontend of UNAS, hence we are also limited to describing the presence of callbacks but not any of their internal details (e.g. we cannot specify if a callback makes any calls itself).

% A description of what the UNAS case tool directly supports/enforces (as opposed to what is possible) (e.g. layering is not supported/enforced although it can be done)

global
UNAS_Message : P DataComponent;
UNAS_Socket : P Port;
UNAS_Procedure : P ControlComponent;
UNAS_Task : P ControlComponent;
UNAS_Callback : P ControlConnector;
UNAS_Circuit : P DataConnector;
UNAS_Trigger : P Trigger;
UNAS_System : P System;

axiom
forall s : UNAS_Socket @
  s.streaming = FALSE and % communication is in discrete messages
  s.blocking = FALSE; % data transfers do not block
forall pc : UNAS_Callback @
  pc.c1 in UNAS_Task and % the caller is a task
  pc.c2 in UNAS_Procedure and % the callee is a procedure
  pc.blocking = TRUE and % the task blocks until the called procedure returns
  pc.queuesize = 0; % procedure calls cannot be queued
forall t : UNAS_Task @
  t.ports subseteq UNAS_Socket;
forall p : UNAS_Procedure @
  p.ports = {};
forall c : UNAS_Circuit @
  {c.p1, c.p2} subseteq UNAS_Socket and % ports are UNAS sockets
  {c.c1, c.c2} subseteq UNAS_Task and % circuits connect UNAS tasks
  c.cc_or_obj = ctype and % circuits always connect tasks and not objects
  c.reliable = TRUE and % UNAS circuits use TCP/IP as underlying protocol
  c.directionality = twoway; % circuits are bidirectional
forall t : UNAS_Trigger @
  t.controlcomponent in UNAS_Task and
9.0 Applications of Approach

9.1 UNAS as an architectural style

TRW’s Universal Network Architecture Services (UNAS) is a middleware COTS package for building large, distributed network Ada (and later C++) systems ([ROYC91]). It facilitates rapid construction and prototyping of applications that can best be described as heterogeneous, or having characteristics of more than one ‘pure’ style. UNAS applications are most similar to distributed processes systems, but with event-based implicit invocation capabilities.

There are two ways to use UNAS to build applications: either manually access the provided libraries in one’s own code, or use the supplied graphical interface for laying out an architecture (a ‘skeleton’) and subsequently fill in the user code. We only discuss the latter approach here (and only for the Ada version 3.2 UNAS). Much of our experience with UNAS is derived from using it to design the software architecture of a hypothetical satellite ground station used as an example in the United States Air Force Academy ([LARS92]).

Though our model does not capture all the details of UNAS, the systems generated by the tool can be modeled as sets of tasks, procedures, triggers, sockets, and circuits. Tasks are the primary control components of each system, and they interact with one another by sending and receiving messages over attached sockets which are connected by circuits. The reception of messages can trigger additional messages and/or the invocation of callback procedures. We have
Figure 21: Architecture mismatches detected in an entity by AAA
After displaying an entity which the architect may be interested in (figure 20), he or she may choose to ask for an analysis of that entity for architecture mismatches (figure 21). The analysis will indicate if any mismatches are found by AAA. For systems, AAA will also report certain resource usage statistics such as shared data and code sizes.

**Figure 20: The attributes and values of an entity stored in AAA**
Figure 18: Results of loading a specification into AAA

Working Style: main subroutine
datastructure(main subroutine): d1
datastructure(main subroutine): d2
procedure(main subroutine): main
procedure(main subroutine): sub1
procedure(main subroutine): sub2
procedurecall(main subroutine): main sub1
  c1: [main]
c2: [sub1]
procedurecall(main subroutine): sub1 sub2
  c1: [sub1]
c2: [sub2]
system(main subroutine): sys8
  initialcontrolcomponent:: [main]
  controlcomponents: [main sub1 sub2]
  wheredefined: [d1, d2, main, sub1, sub1, sub2, sub1, sub2, sub1, sub2, sub1, sub2, sub1, sub2]
  calls: [main sub1]
end Working Style: main subroutine

--> Specification successfully parsed.

Figure 19: Listing of entities stored in AAA after loading a project

Choose Type To Browse: All

Entities ("Style : Entity : Name")
Main/Subroutine : datastructure : d1
Main/Subroutine : datastructure : d2
Main/Subroutine : procedure : main
Main/Subroutine : procedure : sub1
Main/Subroutine : procedure : sub2
Main/Subroutine : procedurecall : main sub1
Main/Subroutine : procedurecall : sub1 sub2
Main/Subroutine : system : sys8
8.3 Sample Operation of AAA

In this section, we briefly describe usage of the tool. AAA is still a prototype application, and therefore is missing certain basic functions that users would typically expect (e.g. saving and loading of projects). However, the essence of AAA is implemented: architectures can be analyzed for mismatches. The main screen is shown in figure 17. A user may click on ‘Project’ to bring up a menu which allows a project (a specification of an architecture) to be loaded. A sample specification of a small main/subroutine system has been successfully loaded in figure 18.

After loading a project, a user can view the entities that are described in the specification via the ‘Browse’ menu. A window listing all the currently loaded entities will be displayed (figure 19), and the user may click on each entity to view its attributes and their values. If the attributes are fixed-value attributes for that style, the default values are filled in by the tool. Otherwise, either the user-specified value is displayed, or the value is listed as ‘unspecified’. AAA will operate with partially specified architectures, however the mismatches will correspondingly fail to be flushed out.
validentity(object(base), Name) :-
  indatabase([Name]),
  getvalues(object, _, Name,
  [methods, nodes, platform],
  [Methods, Nodes, Platform]),

  (Methods == [unspecified] ->
    true;
    (findall(M, (member(M,Methods),
      getvalue(controlcomponent,_,M,method,T),
      (T == [unspecified] ; T == [true])),
       TrueUnspecifiedMethods),
    findall(N, (member(M,Methods),
      getvalue(controlcomponent,_,M,nodes,N),
      N \== [unspecified]), AllMethodNodes),
    findall(P, (member(M,Methods),
      getvalue(controlcomponent,_,M,platform,P),
      P \== [unspecified]), MethodPlatforms),

    (seteq(TrueUnspecifiedMethods, Methods) ->
      true;
      zz('Nonmethod declared as method', Name,
      (TrueUnspecifiedMethods,Methods))
    ),
  ),
  (Nodes == [unspecified] ->
    true;
    (mapvartofix(subset,AllMethodNodes,Nodes) ->
      true;
      zz('Method nodes not subset of object nodes', Name,
      (AllMethodNodes, Nodes))
    ),
  ),
  (Platform == [unspecified] ->
    true;
    (mapvartofix(platformcompatible, MethodPlatforms,Platform) ->
      true;
      zz('Incompatible method platform with object platform', Name,
      (MethodPlatforms, Platform))
    ),
  )
).
are recognized by AAA (as instances of specific styles) includes: TRW’s Universal Network Architecture Services (UNAS), and to a lesser degree HP’s Softbench, Stanford’s InterViews, Forschungszentrum Informatik’s OBST, and CMU’s Mach RPC Interface Generator. Additional styles and COTS packages can be added to AAA only by someone who understands Prolog and Z, however this step need be done only once for each new addition.

An example of the Prolog code which checks for the correctness of a particular entity (the base entity ‘object’ in this case) is given below, preceded by that entity’s Z specification (section 6.2.4).

```prolog
% schema for high-level object
schema Object
  name : Name;
class : Class;
  % class the object belongs to
ports : P Port;
  % object ports
data : P DataComponent;
  % private data of the object
methods : P ControlComponent;
  % control components which act as the object’s methods
nodes : P Node;
  % nodes it resides on
resources : Resources;
  % resource usage
platform : Platform;
  % expected platform
where
  forall c : methods @
    c.method = TRUE and
c.nodes subseteq nodes and
c.platform PlatformCompatible platform;
  % all the methods reside on object nodes, and are compatible with the object’s platform
addResources {r : Resources | exists m : methods @ r = m.resources}
  ResourceCompatible resources;
  % combined method resources are compatible with object resources
end schema
```
AAA is implemented in Prolog (Quintus Prolog Release 3.2), and currently runs on SunOS 4.1.3. Prolog is a relational programming language that has strong support for first-order logic and set theory. It is therefore fairly straightforward to implement the Z specification of our model in Prolog. AAA currently is implemented in approximately 4750 lines of Prolog code. The code is mostly executable descriptions of the Z schemas for the entities that make up each style. There is an X Window front end to the tool, but the graphical entry of architectures is currently not supported.

The styles which AAA recognizes include: base, distributed, multi-threaded, layered, software bus, main/subroutine, pipe & filter, distributed processes, and event-based (implicit invocation). The list of COTS packages which

```
style base
    controlcomponent m1
        method: false;
        ports: p1;
        nodes: node1, node2, node4;
        platform: plat1;
    end;
end;

style distributedprocesses
    system clientserver
        interfacesockets: sktserver;
        initialprocesses: pclient, pdist, pserver;
        allowedprocesses: pclient, pdist, pserver;
        initialcircuits: cktc1todist, cktdisttoserver;
        allowedcircuits: cktc1todist, cktc2todist, cktdisttoserver, redundant;
        threadspawns: clientspawn;
    end;
end;
```

Figure 16: Sample AAA specification fragments

8.2 Technical Details of AAA

AAA is implemented in Prolog (Quintus Prolog Release 3.2), and currently runs on SunOS 4.1.3. Prolog is a relational programming language that has strong support for first-order logic and set theory. It is therefore fairly straightforward to implement the Z specification of our model in Prolog. AAA currently is implemented in approximately 4750 lines of Prolog code. The code is mostly executable descriptions of the Z schemas for the entities that make up each style. There is an X Window front end to the tool, but the graphical entry of architectures is currently not supported.

The styles which AAA recognizes include: base, distributed, multi-threaded, layered, software bus, main/subroutine, pipe & filter, distributed processes, and event-based (implicit invocation). The list of COTS packages which
matches. AAA internally supports constraints on different architectural views to varying degrees, including structure, topology, behavior, and interoperability - i.e. the same views which the model supports.

### 8.1 Architecture Specifications for AAA

The users of AAA do not describe their architectures in Z. Instead, they use a rudimentary notation we have developed for declaring entities and their attributes. This notation is a far cry from a true language, however it is sufficient for specifying basic architectures. All specifications follow a relatively standard format:

- declaration of all base and style-specific systems (followed by)
- declaration of all groupings and wrappings

Each entity (e.g. port, socket, control component, filter, system, etc.) is specified in the context of a particular style, and the specification is in terms of its attributes. The attributes are taken directly from the Z specification of each entity (section 6.2). The user enters in values for the attributes, which AAA checks for proper type and range. Two examples are given in figure 16: a ‘style-less’ or base control component and a distributed processes system.

Constraints are not currently supported in user-entered specifications. A major future extension to AAA would be to allow users to describe constraints on entities, similar perhaps to the constraints given in the property part of the Z schemas used in the model.
8.0 The Architect’s Automated Assistant (AAA)

The model of architectural styles and composition developed thus far is the basis for an architectural analysis CASE tool we have implemented called the Architect’s Automated Assistant (AAA). Software system architects can use this tool to specify large system architectures such as for a satellite ground station or a complex software engineering environment. These specifications can be subsequently analyzed by the tool for architectural mismatches that can be prohibitively expensive to correct if left till later stages in the lifecycle.

As with the model, AAA revolves around the concept of architectural styles. Each system specified in AAA is expressed as an instantiation of a particular architectural style. Examples of architectural styles include main/subroutine, pipe & filter, distributed processes, and event-based. The notion of style can also be used to capture COTS packages, where each package presents its own particular style.

Architects can also specify how subsystems are composed into larger systems using AAA. The tool will check the composition and the underlying subsystems for consistency with each other, reporting any constraint conflicts or mis-
c1 /= c2 and l1 /= l2 and c1 in l1 and c2 in l2 @
s1 in newsystem.shareddata) and
% 2 threads may not share a piece of data

end axiom
or
(forall layer : Layer | layer in newsystem.spawn_layers @
    {spawn.c1, spawn.c2} setint second layer = {}))) and
% spawns constrained to between layers or not constrained at all
(forall d : DataConnector | d in newdataconnectors @
    ((exists layer : Layer | layer in newsystem.dataconnector_layers @
        (d.c1 in first layer and d.c2 in second layer) or
        (d.c2 in first layer and d.c1 in second layer))
    or
    (forall layer : Layer | layer in newsystem.dataconnector_layers @
        {d.c1, d.c2} setint second layer = {}))) and
% dataconnectors constrained to between layers or not constrained at all
(forall s1, s2 : System | s1 in subsystems and s2 in subsystems @
    s1.recognizedmessages = s2.recognizedmessages) and
% all subsystems have identical sets of recognized messages
newsystem.recognizedmessages =
    {x : DataComponent |
        exists s : System | s in subsystems
        @ x in s.recognizedmessages} and
newsystem.triggers =
    {x : Trigger |
        exists s : System | s in subsystems
        @ x in s.triggers} and
newsystem.nodes =
    {x : Node |
        exists s : System | s in subsystems
        @ x in s.nodes} and
not (exists threads : P Thread;
        l1, l2, l3 : Thread;
        c1, c2a, c2b, c3 : ControlComponent;
        d1, d2 : DataConnector |
        threads = {thread : Thread | thread in
            getmaxthreads(newsystem)} and
        l1 in threads and l2 in threads and l3 in threads and
        l1 /= l2 and l2 /= l3 and l3 /= l1 and c2a /= c2b and
        c1 in l1 and c2a in l2 and c2b in l2 and c3 in l3 and
        ((d1.c1 = c1 and d1.c2 = c2a) or (d1.c1 = c2a and d1.c2 = c1)) and
        ((d2.c1 = c3 and d2.c2 = c2b) or (d2.c1 = c2b and d2.c2 = c3)) @
        d1 in newsystem.dataconnectors and
d2 in newsystem.dataconnectors) and
% 2 threads may not have dataconnectors going to 2 different components in a thread
not (exists threads : P Thread;
        l1, l2 : Thread;
        c1, c2 : ControlComponent;
        s1 : DataComponent & (P ControlComponent & P Object) |
        c1 in (first (second s1)) and c2 in (first (second s1)) and
        threads = {thread : Thread |
            thread in getmaxthreads(newsystem)} and
        l1 in threads and l2 in threads and
\( \forall x \in s.\text{calls} \) setunion newcalls and
\{x : \text{ControlConnector} \mid \exists s : \text{System} \mid s \in \text{subsystems} \@ x \in s.\text{spawns} \} setint newspawns = \{\} and
\text{newsystem.spawns} =
\{x : \text{ControlConnector} \mid \\
\exists s : \text{System} \mid s \in \text{subsystems} \\
\@ x \in s.\text{spawns} \} setunion newspawns and
\{x : \text{DataConnector} \mid \exists s : \text{System} \mid \\
s \in \text{subsystems} \@ x \in s.\text{dataconnectors} \} setint newdataconnectors = \{\} and
\text{newsystem.dataconnectors} =
\{x : \text{DataConnector} \mid \\
\exists s : \text{System} \mid s \in \text{subsystems} \\
\@ x \in s.\text{dataconnectors} \} setunion newdataconnectors and
\text{newsystem.shareddata} =
\{x : \text{DataComponent} \& (\text{P ControlComponent} \& \text{P Object}) \mid \\
\exists s : \text{System} \mid s \in \text{subsystems} \\
\@ x \in s.\text{shareddata} \} += \text{extendedshareddata} and

% extended shared data overrides subsystem’s shared data

(forall layer : \text{Layer} \mid \\
layer in \text{newcall_layers} or \\
layer in \text{newspawn_layers} or \\
layer in \text{newdataconnector_layers} @ \\
first layer setunion second layer subseteq \\
\text{getallcontrolcomponents}(\text{newsystem.controlcomponents}, \\
\text{newsystem.globalobjects}) \} and

% all layers drawn from the control components
\text{newsystem.call_layers} =
\{x : \text{Layer} \mid \\
\exists s : \text{System} \mid s \in \text{subsystems} \\
\@ x \in s.\text{call_layers} \} setunion newcall_layers and
\text{newsystem.spawn_layers} =
\{x : \text{Layer} \mid \\
\exists s : \text{System} \mid s \in \text{subsystems} \\
\@ x \in s.\text{spawn_layers} \} setunion newspawn_layers and
\text{newsystem.dataconnector_layers} =
\{x : \text{Layer} \mid \\
\exists s : \text{System} \mid s \in \text{subsystems} \\
\@ x \in s.\text{dataconnector_layers} \} setunion newdataconnector_layers and
(forall call : \text{ControlConnector} \mid \\
call in \text{newcalls} @ \\
(forall layer : \text{Layer} \mid \\
layer in \text{newsystem.call_layers} @ \\
(call.c1 in first layer and call.c2 in second layer) or \\
(call.c2 in first layer and call.c1 in second layer)) \\
or \\
(forall layer : \text{Layer} \mid \\
layer in \text{newsystem.call_layers} @ \\
\{call.c1, call.c2\} setint second layer = \{\}) \}) and

% calls constrained to between layers or not constrained at all
(forall spawn : \text{ControlConnector} \mid \text{spawn in newspawns} @
(forall layer : \text{Layer} \mid \\
layer in \text{newsystem.spawn_layers} @ \\
(spawn.c1 in first layer and spawn.c2 in second layer) or \\
(spawn.c2 in first layer and spawn.c1 in second layer))

99
exists s : System | s in subsystems @ x in s.controlcomponents} and
newsystem.globalobjects =
{x : Object |
exists s : System | s in subsystems @ x in s.globalobjects} and
(forall c : ControlConnector | c in newcalls or c in newspawns @
(exists s1, s2 : subsystems @
c.c1 in getallcontrolcomponents(s1.controlcomponents, s1.globalobjects) and
c.c2 in getallcontrolcomponents(s2.controlcomponents, s2.globalobjects) and
s1 /= s2)) and
(forall d : DataConnector | d in newdataconnectors @
(exists s1, s2 : subsystems @
d.c1 in getallcontrolcomponents(s1.controlcomponents, s1.globalobjects) and
d.c2 in getallcontrolcomponents(s2.controlcomponents, s2.globalobjects) and
s1 /= s2)) and
% all new connectors span the subsystems
{c : ControlComponent |
(exists call : ControlConnector | call in newcalls @
c = call.c1 or c = call.c2) or
(exists spawn : ControlConnector | spawn in newspawns @
c = spawn.c1 or c = spawn.c2) or
(exists dc : DataConnector | dc in newdataconnectors @
c = dc.c1 or c = dc.c2}) subseteq
newsystem.controlcomponents and
% all components addressed in the new calls, spawns, and dataconnectors are members of the new system control components
(this constraint is redundant)
(forall d : DataComponent; cset : P ControlComponent; oset : P Object |
(d, (cset, oset)) in extendedshareddata @
(exists oldcset : P ControlComponent; oldoset : P Object; s : System @
s in subsystems and (d, (oldcset, oldoset)) in s.shareddata and
oldcset subseteq cset and
oldoset subseteq oset and
getallcontrolcomponents(newsystem.controlcomponents, newsystem.globalobjects) and
oldcset subseteq cset and
oset subseteq getallobjects(newsystem.controlcomponents, newsystem.globalobjects))) and
% extended shareddata uses existing data but with larger sets of sharing components/objects
{x : ControlConnector | exists s : System |
s in subsystems @ x in s.calls} setint newcalls = {} and
newsystem.calls =
{x : ControlConnector |
exists s : System | s in subsystems
global
  getallcontrolcomponents :
    (P ControlComponent & P Object) +-> P ControlComponent;
axiom
  forall inputccs, outputccs : P ControlComponent; inputobs : P Object @
  getallcontrolcomponents(inputccs, inputobs) = outputccs <=>
    outputccs = inputccs setunion r_getallmethods(inputobs) setunion
    r_getallmethods(r_getallobjects({o : Object |
      exists c : inputccs; n : Name |
      n in c.localobjects @ n = o.name}));
end axiom

% Helper function for recursively retrieving all the objects associated with a set
% of control components and objects. Retrieves all local objects found.

global
  getallobjects : (P ControlComponent & P Object) +-> P Object;
axiom
  forall inputobs, outputobs : P Object; inputccs : P ControlComponent @
  getallobjects(inputccs, inputobs) = outputobs <=>
    outputobs = inputobs setunion
    r_getallobjects({o : Object | exists c : inputccs; n : Name |
      n in c.localobjects @ n = o.name});
end axiom

% group operation for composition

global
  group : (P System & Name & P ControlConnector & P ControlConnector &
  (DataComponent +-> (P ControlComponent & P Object)) &
  P DataConnector & Layers & Layers & Layers) +-> System;
  % inputs include a set of (sub)systems and new bridging connectors. out-
  put is the resulting system.
axiom
  forall subsystems : P System; newname : Name;
  newcalls, newspawns : P ControlConnector;
  extendedshareddata : DataComponent +->
    (P ControlComponent & P Object);
  newdataconnectors : P DataConnector;
  newcall_layers, newspawn_layers, newdataconnector_layers : Layers;
  newsystem : System @
  group(subsystems, newname, newcalls, newspawns, extendedshareddata,
  newdataconnectors, newcall_layers, newspawn_layers, newdataconnector_layers) =
  newsystem <=>
    newsystem.name = newname and
    newsystem.initialcontrolcomponents =
    {x : ControlComponent |
      exists s : System | s in subsystems
        @ x in s.initialcontrolcomponents} and
    newsystem.controlcomponents =
    {x : ControlComponent |
13. A spawn is made into a subsystem which is not concurrent.

14. A triggered spawn is made into a subsystem which is not concurrent.

15. A remote connector is extended into or out of a non-distributed subsystem (i.e. a subsystem originally confined to a single node).

16. A node resource is overused (this is actually checked by summing across the subsystems’ usage of that particular resource).

Along with these mismatches, we can also identify a general interoperability mismatch where an attribute incompatibility arises across a connector between two components (e.g. different subsystems may assume different platforms).

The conceptual feature of encapsulation mostly results in scope/access structural mismatches, and these are covered under the general system constraints (see section 6.2.4.2 for the specification of base systems and their constraints).

Note that the group operation produces a system as its output, and therefore there is actually another set of possible mismatches to consider. That particular set corresponds to the constraints of the system schema.

Each interaction type which is allowed by the group operation adds a new row of potential mismatches to table 8. This is indicative of part of the difficulty of composition: new interactions rapidly escalate the complexity of searching for mismatches.

### 7.4.2 Specification of Group Operation

% Helper function for recursively retrieving all the control components associated with a set of control components and objects. Retrieves methods of all local objects found.
2. Two threads have data connectors to 2 different control components in a third thread (it is impossible for the third thread to execute in the two components simultaneously - see figure 14).

3. Two control components in the same thread share a blocking data connector, creating a possibility of deadlock.

4. A layering constraint is violated.

5. Different sets of recognized messages are used by two subsystems that permit triggers.

6. A spawn is made into a subsystem which originally forbade them.

7. An unrecognized trigger message is used.

8. A triggered spawn is made into a subsystem which originally forbade spawns.

9. A trigger refers to a subsystem which originally forbade triggering.

10. A data connector is made into a subsystem which originally forbade them.

11. A shared data relationship refers to a subsystem which originally forbade them.

12. A trigger refers to a subsystem which forbids explicit or implicit data connectors, hence the trigger may never occur.
some interaction/connector type $T$ (the row) and some conceptual feature $F$ (the column). Each mismatch is caused by one of three cases where we join two sub-systems via a set of connectors of type $T$:

- subsystem $S_1$ having property $F$ with subsystem $S_2$ also having property $F$
- subsystem $S_1$ having property $F$ with subsystem $S_2$ not having property $F$
- subsystem $S_1$ not having property $F$ with subsystem $S_2$ also not having property $F$

An explanation of each mismatch is given below.

1. Two concurrent threads share data, with potential synchronization problems.
nectors that are introduceable by a group are only a subset of the possible types (section 7.1). They include new sets of bridging:

- blocking control connectors (calls)
- non-blocking control connectors (spawns)
- data connectors
- shared data relationships that add control components and objects to existing shared data
- triggered calls
- triggered spawns
- triggered data transfers
- shared machine relationships

These new connectors are not specific to any particular style. This allows the group operation to connect subsystems possessing different styles (thereby achieving heterogeneity).

When a collection of subsystems are grouped, architecture mismatches may be generated. For every new interaction between two subsystems, there is a possibility of subsystem-subsystem constraint inconsistencies. There is also the possibility that the resulting system may impose a constraint which conflicts with the subsystems’ constraints (see section 7.3). While every constraint is a potential participant in a mismatch, we noted earlier that the conceptual features are especially important since they affect more styles. We highlight the mismatches due to the conceptual features in table 8. Each mismatch in the table’s cells is related to
involving a style’s ability to engage in blocking or non-blocking control transfers (i.e. the *dynamism* conceptual feature) will affect multiple styles (main/subroutine, distributed processes, event-based, etc.), but a constraint such as the “main” procedure not being reentrant will only affect the main/subroutine style.

A closely related benefit of the conceptual features is the quick extensibility of the mismatches to *new styles* that can be distinguished by this model. If the new style can be identified in terms of certain conceptual features, then it will suffer from all the mismatches that relate to its choices on those features during composition.

### 7.4 The Group Operation

#### 7.4.1 Overview of Group Operation

The group operation joins two or more subsystems via a set of bridging interactions/connectors to form a single, larger system. The types of bridging con-
• during the group operation, a subsystem constraint conflicts with another subsystem constraint

For example, in the first case, an architect may find that a library of components does not meet certain imposed performance requirements. In the second case, two libraries may use different network protocols for data transfers.

Identifying architecture mismatches only demonstrates potential failures of actually implementing the composition. Success is not guaranteed if the composition indicates no mismatches. This is due to the number of limiting assumptions we have made for the composition operation, especially our focus on only a limited number of all the possible interactions between systems (section 7.1). We also do not go beyond identifying the mismatches; i.e. our model does not provide solutions for resolving the mismatches.

7.3.1 Utility of the Conceptual Features for finding Mismatches

Earlier we identified a set of attributes and constraints that are useful for distinguishing many different idealized architectural styles (section 6.3). This set of conceptual features is also useful for identifying important mismatches during composition. While a mismatch can arise due to an inconsistency between any two constraints, it is the mismatches that involve the conceptual features which are especially useful to identify. The set of mismatches which involve the conceptual features are likely to occur in more than one style, whereas mismatches involving other constraints are not as likely to occur (figure 13). For example, mismatches
system between the two original systems, and grouping the three together. For example, we may wish to combine two systems which utilize variants of the pipe & filter style: one is streaming and the other is not. To compose them using pipes, we could wrap one system to satisfy the requirement of the other, or we could simply insert a third, mediating system in between the two original systems. This mediator would translate from streaming to non-streaming data transfers and vice-versa.

- The semantics of systems is not modeled at the component or connector level if we look at our eight base entities (section 6.2.4). For example, system properties that are reflected by the conceptual features (section 6.3), like concurrency or dynamism, are not found in control components or data connectors the way we have defined them. This makes it very difficult to identify architecture mismatches that stem from component-system or connector-system constraint inconsistencies.

7.3 Architecture Mismatches under Composition

A major obstacle to architectural composition is identifying architecture mismatches ([GARL94]). A mismatch is an inconsistency between two or more constraints belonging to different architectures that are being composed. Mismatches typically arise in the following general cases:

- during the group operation, a subsystem constraint conflicts with an imposed system-wide constraint
tant consequence of the uniform representation used in our model of architectures and architectural styles. Any new style which is expressible in our model can also be addressed by the composition operation. In the next two sections, we explain this result in more detail by describing how we determine architectural mismatches, and also by the formal specification of the operation.

7.2.1 Other Composition Operations

It is possible to envision other composition operations which may seem to add functionality beyond the group operation. The most notable example would be a wrap operation which encapsulates a complete system as a component or a connector for use in another system. We do not model this operation for three reasons:

- The group operation does not presuppose any architecture mismatches (see the next section). It may, on the other hand, result in architecture mismatches. A wrap operation is typically done to resolve an architecture mismatch which prevented two or more existing systems from being grouped previously. The wrap operation is therefore a solution to mismatches, one of many solutions in fact. However, as we noted earlier in the introduction (section 1.2), we are only identifying the potential failures without enumerating the solutions.

- A group operation can be substituted for a wrap operation in an architectural specification. The wrapper provides some mediation between two ‘ungroup-able’ architectures. However, we can achieve the same effect by placing a third
7.2 Overview of Operations Involved in Composition

It is useful to establish a disciplined approach to composition; that is, an approach which is a sequence of simple composition operations. Given the model of architectures and architectural styles specified earlier (section 6.2), and given the discussion in the previous section on the different interpretations of composition, we can model heterogeneous architectural composition using essentially one basic operation. This operation, called **group**, joins two or more systems together by adding a set of bridging interactions (connectors) between the systems.

A software system architecture can be composed using the group operation by applying it to a set of subsystems (figure 12). This process of applying grouping operations is not new in software of course, having widespread use at other stages of software design. What is new here is our **formal specification of the systematic use of a composition operation on systems described in terms of their architectural styles.**

The group operation is not tailored to one architectural style or another; i.e. it is independent of the styles of the systems being composed. This is an impor-
will confine ourselves to the composition of systems. We will not consider the
details of how other base entities are constructed as composition.

The second aspect is concerned with the choice of allowed interaction
types between systems (and only systems as the previous paragraph established).
The number of possible types of interactions between two or more systems is large
(table 7). By interaction, we mean any type of relationship that can exist between

<table>
<thead>
<tr>
<th>call</th>
<th>spawn</th>
</tr>
</thead>
<tbody>
<tr>
<td>data connector</td>
<td>shared data</td>
</tr>
<tr>
<td>triggered call</td>
<td>triggered spawn</td>
</tr>
<tr>
<td>triggered data transfer</td>
<td>share a machine</td>
</tr>
<tr>
<td>statically declare</td>
<td>dynamically declare</td>
</tr>
<tr>
<td>import</td>
<td>export</td>
</tr>
</tbody>
</table>

Table 7: Partial list of interactions between system components

the systems or their parts (typically their objects and control components). For
example, a control component in system A can call a control component in system
B.

Naturally, the more views considered (see table 4 in section 4.2), the
more types of interactions are possible. In this report we only consider a subset of
these interaction types during composition. Each possible interaction type expo-
nentially complicates the problem of composition (see section 7.3 and section 7.4).
7.0 Composition of Architectural Styles

There are few large software systems that adhere purely to a single architectural style. Most systems are composed out of a mixture of more than one style. Garlan and Shaw have described these systems as having heterogeneous architectures ([GARL93]). For example, the X Window System graphical user interface which served as the basis for our description of the event-based style is actually object-oriented as well. Another example is a hypothetical satellite ground station which relies on the pipe and filter style for some parts of the system, and the event-based style for other parts (figure 1 on page 4).

7.1 Interpreting “Composition”

Even after restricting its scope to the level of architectures, software composition remains a hard problem. There are at least two different aspects of composition that make it complicated:

- different granularities of components can be considered
- many types of interactions can exist between components

The first aspect is concerned with the choice of operands during an architectural composition (if we view it as an operation). We have identified eight base entities (section 6.2.4), and each one could potentially be the focus of composition. For example, we could consider the process of constructing a control component out of ports, local variables, etc. as composition. In this report, however, we
system may engage in control transfers when the state of the central blackboard’s
data changes (i.e. the knowledge sources respond). Though this is a form of trig-
ger, our model is unable to capture it since our triggers as defined (section 6.2.4)
only respond to events on data connectors. A more serious shortcoming is the
inability of the conceptual features to address styles which are especially charac-
terized by their behavior, e.g. real-time and feedback control systems.
Together, these seven features can be used to represent the essence of several styles. In table 6, we use this architectural space to describe four different styles: pipe & filter, main/subroutine, distributed processes, and event-based.

<table>
<thead>
<tr>
<th></th>
<th>Pipe &amp; Filter</th>
<th>Main/Subroutine</th>
<th>Distributed Processes</th>
<th>Event-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamism</td>
<td>static</td>
<td>static</td>
<td>dynamic</td>
<td>static</td>
</tr>
<tr>
<td>Supported data transfers</td>
<td>explicit data connectors</td>
<td>shared data variables</td>
<td>explicit data connectors</td>
<td>implicit network, shared data variables</td>
</tr>
<tr>
<td>Triggering capability</td>
<td>no</td>
<td>N/A</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Concurrency</td>
<td>multi-threaded</td>
<td>single-threaded</td>
<td>multi-threaded</td>
<td>single-threaded</td>
</tr>
<tr>
<td>Distribution</td>
<td>unconstrained</td>
<td>single node</td>
<td>multiple nodes</td>
<td>unconstrained</td>
</tr>
<tr>
<td>Layering</td>
<td>unconstrained</td>
<td>unconstrained</td>
<td>unconstrained</td>
<td>unconstrained</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Table 6: Four Instances in an Architectural Style Space**

Naturally, the question arises whether these features are sufficient to adequately differentiate each possible architectural style from all other styles. The list of styles which can be distinguished includes pipe & filter, main/subroutine, distributed processes, event-based, software bus, object-oriented, and layered. A reminder however that certain minor attributes and/or constraints of these styles are not reflected in the conceptual features. For example, often in a main/subroutine system, there is a constraint stating that the main routine is not reentrant.

Styles that cannot be fully distinguished by the conceptual features alone include blackboards, databases, rule-based systems, logic programming systems, real-time systems, and feedback control systems. For example, a blackboard
2. **Supported data transfers.** Of the styles we studied, all of them achieved data transfers through one (or more) of three mechanisms: explicit data connectors, an implicit global network of data connectors, or shared data variables.

3. **Triggering capability.** Hardware has interrupts, and software has triggers. Some styles allow the transfer of data (events) along explicit data connectors or a global network to cause certain actions, e.g. control transfers or additional data transfers.

4. **Concurrency.** Styles often constrain the number of concurrent threads that may execute within a system. A single-threaded system is limited to only one thread of control components linked by a chain of calls (blocking control connectors), while multi-threaded systems allow more than one thread to execute concurrently. Note that concurrency is not the same as dynamism.

5. **Distribution.** A style may or may not constrain the mapping of system entities to nodes. If the mapping is to more than one node, then the style’s systems are naturally distributed.

6. **Layering.** Styles may or may not impose system layering constraints on its control components. The layers must be specified with respect to a connector, e.g. a set of layers linked by control connectors.

7. **Encapsulation.** As shown earlier (section 6.2.4.1), objects are fundamentally different from data control components. A style may choose to be object-oriented or not.
to the composition of different architectural styles with each other. Hence, it is useful to model COTS as very specific styles. We give an extended example of one particular COTS package in section 9.1.

6.3 A Conceptual Feature Space underlying Architectural Styles

The eight base entities have many attributes which are constrained in different ways by each style to produce style-specific, refined entities. Of these attributes and constraints, we can identify a core subset which is especially useful for differentiating one style from another. This particular set of attributes and constraints, which we call conceptual features, represents the underlying dimensions for a space of idealized architectural styles. Different styles are specified by choosing different constraints for each conceptual feature. The importance of these features will become clear when we discuss the operations involved in heterogeneous composition (section 7.3.1).

There are seven conceptual features which can be used to differentiate the styles we formally specified:

1. Dynamism. Earlier we discussed how the topology of software can be dynamic, adding and removing concurrent threads as it executes (section 4.2.2.1). Some styles constrain the topology to be static (i.e. the number of concurrent threads remains constant), while other styles do not. A style is dynamic if and only if it allows non-blocking control connectors (spawns).
axiom

forall p : EB_Procedure @
    p.locals subseteq EB_DataStructure and
    p.ports = {};

forall pc : EB_ProcedureCall @
    {pc.c1, pc.c2} subseteq EB_Procedure and
    % call connects two procedures
    pc.queuesize = 0 and
    % procedure calls cannot be queued
    pc.blocking = TRUE;
    % the calling procedure blocks until the called procedure returns

forall o : EB_Object @
    o.data subseteq EB_DataStructure and
    o.methods subseteq EB_Procedure;

forall t : EB_Trigger @
    t.object in EB_Object and
    t.inmessage in EB_Event and
    t.outmessageset subseteq EB_Event and
    t.outcalls subseteq EB_Procedure and
    t.subtype = otype and
    % limit reception of events to objects
    t.outspawns = {};
    % no spawns allowed to be triggered

forall s : EB_System @
    s.controlcomponents subseteq EB_Procedure and
    s.globalobjects subseteq EB_Object and
    s.calls subseteq EB_ProcedureCall and
    s.recognizedmessages subseteq EB_Event and
    s.triggers subseteq EB_Trigger and
    s.spawns = {} and
    s.shareddata = {} and
    s.triggers /= {};

end axiom

6.2.6 COTS-specific extensions to base entities

The base ‘style-less’ entities can be refined not only to specific styles as described in the previous section, but also to high-level models of commercial-off-the-shelf (COTS) packages. Many COTS packages for building software systems inherently embody very specific styles, each with its own set of capabilities and idiosyncrasies. If an application is built by integrating several different COTS packages together, often there will be architectural mismatches that can be traced
An event-based system is made up of a collection of event-based objects and an event manager. The event manager is responsible for forwarding and receiving events to and from objects. Different managers will recognize different sets of events (e.g. in the X Window System, there are Xlib, XtIntrinsic, and other sets). The objects must register an interest in a particular event type if they want the manager to forward them events of that type.

Unlike the distributed processes and pipe/filter styles, there are no explicit data connectors (e.g. pipes or circuits) that are required to transfer events between objects and the manager. This is due to the way that event-based systems are expected to behave. Instead of forwarding events across static paths all the time, events are sent to objects that have registered an interest for that particular event type. Thus, there is implicitly one global data connector between all the objects (including the event manager which is responsible for distributing events). This is the case with the X Window System, the global connector being the ‘network’. CORBA provides an interesting variant to this style by making event channels explicit objects in themselves. This is a more complex approach, but also more scalable since it allows multicasting of events instead of perpetual broadcasting.

%% Event-Based (implicit invocation) Style
\begin{verbatim}
% global
EB_Event : P DataComponent;
EB_DataStructure : P DataComponent;
EB_Procedure : P ControlComponent;
EB_ProcedureCall : P ControlConnector;
EB_Object : P Object;
EB_Trigger : P Trigger;
EB_System : P System;
\end{verbatim}
Like most systems, however, CORBA is best understood as a heterogeneous architecture since it draws upon at least two other styles (object-oriented and distributed database).

An older example of the event-based style is the class of applications built for the X Window System, a popular graphical interface for Unix based workstations [SCHE86, NYE92]. It is the basis for our model of this style. One important thing to note about X and our semantics for this style is that it is also object-oriented. Thus, it is not as subjectively ‘pure’ as the others. Informally, this style is characterized by a set of event-based objects producing events which are sent to a centralized event manager which in turn forwards the events back to the appropriate objects. The receiving objects may invoke certain procedures (known as callbacks) in response to the input events - and thereby produce a new set of events. Data structures are used to store the data within each event-based object.

Events are modeled as simple data components, however, the architect is responsible for listing all the possible types (i.e. enumerating the members of the set of events - e.g. `exposewindowevent`, `resizewindowevent`, etc.).

Event-based objects are specified mainly by the events they can produce and consume, as well as by the procedures used for callbacks. In the X Window System, objects are sometimes called widgets. Objects are assumed to process events sequentially, i.e. each callback invocation must return before any further callbacks are made.
\{(c.c1, c.c2) \subseteq DP\_Process \land
\quad c.cc\_or\_obj = ctype \land
\% \ \text{circuits always connect processes and not objects}
\quad c.directionality = twoway; \%
\quad \text{bidirectional}
\}
\forall pc : DP\_ProcessCall @
\{(pc.c1, pc.c2) \subseteq DP\_Process \land
\quad \text{call connects two processes}
\quad pc.queuesize = 0 \land
\% \ \text{process calls cannot be queued}
\quad pc.blocking = \text{TRUE}; \%
\quad \text{the calling process blocks until the called process returns}
\}
\forall ps : DP\_ProcessSpawn @
\{(ps.c1, ps.c2) \subseteq DP\_Process \land
\quad \text{spawn connects two processes}
\quad ps.queuesize = 0 \land
\% \ \text{process spawn cannot be queued}
\quad ps.blocking = \text{FALSE}; \%
\quad \text{the spawning process does not block}
\}
\forall s : DP\_System @
\quad s \in \text{DistributedSystems and}
\quad s.controlcomponents \subseteq DP\_Process \land
\quad s.dataconnectors \subseteq DP\_Circuit \land
\quad s.calls \subseteq DP\_ProcessCall \land
\quad s.spawns \subseteq DP\_ProcessSpawn \land
\quad s.globalobjects = {} \land
\quad s.classes = {} \land
\quad s.shareddata = {} \land
\quad s.triggers = {} \land
\quad (\forall c : s.dataconnectors @
\quad \quad \exists p1, p2 : s.controlcomponents @
\quad \quad \quad c.p1 \in p1.ports \land c.p2 \in p2.ports) \land
\% \ \text{all circuit sockets connect process sockets}
\quad (\forall c1, c2 : s.dataconnectors @
\quad \quad \quad \text{c1} \neq \text{c2} \Rightarrow
\quad \quad \quad \quad \quad (\text{not} \ (\text{c1.p1} = \text{c2.p1} \land \text{c1.p2} = \text{c2.p2})) \land
\quad \quad \quad \quad \quad (\text{not} \ (\text{c1.p1} = \text{c2.p2} \land \text{c1.p2} = \text{c2.p1})); \%
\quad \quad \quad \text{no two circuits connect the same pair of sockets}
\end{axiom}

6.2.5.5 Event-Based (Implicit Invocation)

Event-based systems are traditionally associated with graphical user interfaces, but this style is becoming more popular for other domains. For example, the Common Object Request Broker Architecture (CORBA) is an approach to client/server computing which has elements of the event-based style in it
data will eventually be transferred. Sockets in this style actually contain two buffers: one to send with, and another to receive with. However, the ‘buffersize’ attribute in ‘DP_Socket’ s specification is assumed to apply to both buffers. Circuits are bidirectional, so messages can be transferred in both directions.

The topology of a distributed processes system can be extremely dynamic. Similar to the main/subroutine style, both threads and processes can make ‘calls’, and in addition, they can also ‘spawn’ one another. When a call is made, the calling entity blocks and is suspended from computing until the called entity returns. When a spawn is made however, the calling entity does not block, and the result is that the caller and callee now proceed concurrently.

Systems built in this style are essentially modeled as a set of communicating processes. We also specify the sets of allowed control connectors, blocking and non-blocking. However, we have disallowed shared memory in our model of this style, preferring to leave it as a property of a variant on this style.

% Distributed Processes Style
global
   DP_DataStructure : P DataComponent;
   DP_Socket : P Port;
   DP_Process : P ControlComponent;
   DP_Circuit : P DataConnector;
   DP_ProcessCall : P ControlConnector;
   DP_ProcessSpawn : P ControlConnector;
   DP_System : P System;
axiom
   forall s : DP_Socket @
      s.iotype = io_inout;
   forall p : DP_Process @
      p.ports subseteq DP_Socket and
      p.locals subseteq DP_DataStructure and
      p.localobjects = {};
   forall c : DP_Circuit @
      {c.p1, c.p2} subseteq DP_Socket and
forall s : MS_System @
s.controlcomponents subseteq MS_Procedure and
dom s.shareddata subseteq MS_DataStructure and
s.calls subseteq MS_ProcedureCall and
s.globalobjects = {} and
s.classes = {} and
s.spawns = {} and
s.dataconnectors = {} and
s.spawn_layers = {} and
s.dataconnector_layers = {} and
s.triggers = {} and
# s.initialcontrolcomponents = 1 and
# (getmaxthreads(s)) = 1 and
% all M/S systems have a single thread
(exists main : MS_Procedure; subroutines : P MS_Procedure @
s.initialcontrolcomponents = {main} and
{main} setunion subroutines = s.controlcomponents and
main notin subroutines and
(forall call : s.calls @ call.c2 /= main) and
cdgraph (main, subroutines) = {c1, c2 : MS_Procedure |
exists call : s.calls @ call.c1 = c2 and call.c2 = c2}) and
% system forms a connected, directed graph of calls starting from
main
# (n : Node | (exists c : ControlComponent |
c in s.controlcomponents @ n in c.nodes) or
(exists call : ControlConnector | call in s.calls @
n in call.nodes)) = 1;
% the system rests on one node
end axiom

6.2.5.4 Distributed Processes

As networked computing has exploded, so has the prevalence of this
style. Its strong support for concurrency, data transfers, and control transfers made
it the key to solving integration problems for one researcher [POWE90]. The
semantics of our model of this style is based on traditional Unix programming.

Processes and data structures are this style’s only control and data com-
ponents respectively. Processes communicate with one another (i.e. transfer data
between one another) through sockets and circuits. A process may send data to a
socket, and if there is a circuit attached to it connecting it to another socket, the
6.2.5.3 Main/Subroutine

This style was one of the earliest to develop since it is well suited to uniprocessor platforms. The semantics of our model of this style is based on a simple C program [KERN78]. The only data and control components are the data structure and procedure respectively. There are no explicit data connectors, but interprocedure communication can be achieved by utilizing shared, global variables. There is a single control connector in the main/subroutine style: the ubiquitous procedure call. Interprocedure communication can also be achieved by procedure call arguments.

Main/subroutine systems are single-threaded collections of procedures and data structures, with the procedures interconnected by procedure calls and shared variables. A special procedure, ‘main’, is distinguished from the other procedures: it cannot be called, and it must be at the root of an activation directed graph encompassing all the procedures.

```plaintext
% Main/Subroutine Style
global
  MS_DataStructure : P DataComponent;
  MS_Procedure : P ControlComponent;
  MS_ProcedureCall : P ControlConnector;
  MS_System : P System;
axiom
  forall p : MS_Procedure @
    p.ports = {} and
    p.locals subseteq MS_DataStructure and
    p.localobjects = {};
  forall pc : MS_ProcedureCall @
    {pc.c1, pc.c2} subseteq MS_Procedure and
    % call connects two procedures
    pc.queuesize = 0 and
    % procedure calls cannot be queued
    pc.blocking = TRUE;
    % the calling procedure blocks until the called procedure returns
```
PF_Socket : P Port;
PF_Pipe : P DataConnector;
PF_Filter : P ControlComponent;
PF_System : P System;

axiom

forall s : PF_Socket @
  s.iotype /= io_inout and
  % no socket is used for both input and output
  s.streaming = TRUE;
  % all data transfers are streamed
forall p : PF_Pipe @
  p.cc_or_obj = ctype and
  % pipes always connect filters and not objects
  p.directionality = forward and
  % unidirectional
  {p.c1, p.c2} subseteq PF_Filter and
  % pipe connects two filters
  {p.p1, p.p2} subseteq PF_Socket and
  % ports belong to the P/F style
  p.p1.iotype = io_out and p.p2.iotype = io_in and
  % p1 acts as sender, p2 acts as receiver
  p.p1 /= p.p2;
  % the sender socket is different than the receiver
forall f : PF_Filter @
  f.ports subseteq PF_Socket and
  % ports belong to the P/F style
  f.localobjects = {};
forall s : PF_System @
  s.controlcomponents subseteq PF_Filter and
  s.dataconnectors subseteq PF_Pipe and
  s.globalobjects = {} and
  s.classes = {} and
  s.calls = {} and
  s.spawns = {} and
  s.shareddata = {} and
  s.call_layers = {} and
  s.spawn_layers = {} and
  s.triggers = {} and
  s.initialcontrolcomponents = s.controlcomponents and
  s.initialdataconnectors = s.dataconnectors and
  (forall p : s.dataconnectors @
   exists f1, f2 : s.controlcomponents @
   p.p1 in f1.ports and p.p2 in f2.ports) and
  % all pipe sockets connect filter sockets
  (forall p1, p2 : s.dataconnectors @
   (p1.p1 = p2.p1 and p1.p2 = p2.p2) => p1 = p2));
  % no two pipes connect the same sender and receiver
end axiom
([VENU90], [KUO90]). Our description of this style differs from those previous efforts in two ways:

- the internal attributes of sockets are explicitly modeled.
- filters are not constrained to process data in lockstep fashion where every instance of consumed input is accompanied by some production of output.

The semantics adopted here is based on Unix programming level pipes, but there is one important difference in the way that systems may behave (described below).

Filters are the only control components in this style. They possess sockets which act as message buffers. Filters can place data on sockets which are connected to other sockets via pipes. Pipes transfer the data from one socket to another. Among other things, we constrain pipes to be unidirectional and streaming.

Unlike most other styles, pipe and filter systems are unusual in that the dynamic topology is fairly static. We have purposely constrained the style so that the topology is invariant over time: no filters or pipes are added or removed during the execution of the system. Thus, pipe and filter systems begin execution with all filters and pipes running simultaneously. Note that this is not the case in Unix based systems where filters and pipes can be created dynamically. We have constructed our description in this way to emphasize the data-oriented nature of this style.
(exists t : system.triggers @ topcc in t.outspawns or topcc in t.outcalls) or
(exists c1 : ControlComponent; spawn : ControlConnector | spawn in system.spawns @
spawn.c1 = c1 and spawn.c2 = topcc)) and
thread = {x : ControlComponent |
exists chain : P ControlConnector |
chain subseteq system.calls @
dconnected (topcc, x) = {a,b : ControlComponent |
exists call : ControlConnector |
call in chain @ a = call.c1 and b = call.c2})

end axiom

% three weakly constrained styles

global

SoftwareBusSystems : P System;
DistributedSystems : P System;
MultithreadedSystems : P System;

axiom

forall s : System | s in SoftwareBusSystems @
(ustar s.controlcomponents = {x, y : ControlComponent |
exists d : DataConnector @ d in s.dataconnectors and
d.c1 = x and d.c2 = y}) and
(# s.dataconnectors = # s.controlcomponents - 1);
% dataconnectors form a star (center/bus distributes messages)
forall s : System | s in DistributedSystems @
# {n : Node |
(exists c : ControlComponent | c in s.controlcomponents @
n in c.nodes) or
(exists call : ControlConnector | call in s.calls @ n in call.nodes) or
(exists spawn : ControlConnector | spawn in s.spawns @
n in spawn.nodes) or
(exists dc : DataConnector | dc in s.dataconnectors @
n in dc.nodes}) > 1;
% the system rests on more than one node
forall s : System | s in MultithreadedSystems @
# (getmaxthreads(s)) > 1;
% the maximum number of possible threads is greater than one

end axiom

6.2.5.2 Pipes and Filters

Pipe and filter systems were some of the first to receive formal attention as examples of an architectural style ([ALLE92], [ABOW93]). Earlier research in systems integration used the simplicity of this style to facilitate composition
be modeled very simply by a base system with non-empty sets of layer for one of
the connector types (call, spawn, etc.). We do not show this example here since it
is relatively simple, but instead we show the specification for three other weakly
constrained styles: *software buses*, *distributed systems*, and *multithreaded systems*.

In order to describe multithreaded systems in particular, we introduce
the function ‘getmaxthreads’ which extracts the maximum number of unique
threads in a system (note that at runtime, a system may have more threads but we
are simply looking for unique threads here, and this can be determined statically). Given a system which has a set of control components and a set of calls (blocking
control connectors) over the components, we can find a thread by applying the fol-
lowing algorithm:

1. Find a control component in the system which is either an initial control com-
   ponent or which is spawned.

2. Using a sequence of calls starting from that control component, find the largest
   reachable set of control components which are in the system. This set is repre-
   sents a unique thread in the system.

We repeat this as necessary until all possible unique threads are found.

```plaintext
global
    getmaxthreads : System +→ P Thread;
    % retrieve maximum number of different threads in a system
    % Assumes that objects do not start out executing any method concur-
    % currently

axiom
    forall system : System; threads : P Thread @
    getmaxthreads(system) = threads ↔
    (forall thread : Thread | thread in threads @
     (exists topcc : ControlComponent @
      (topcc in system.initialcontrolcomponents or
```

71
6.2.5 Style-Specific Extensions to Base Entities

The eight base entities described in the previous section can be further constrained to produce style-specific entities (figure 11). A control component, for example, can be specialized to a filter (pipe and filter style) or a procedure (main/subroutine style). In the following sections, we give examples of extensions to different styles.

6.2.5.1 Weakly Constrained Styles

Although it is impossible to draw a definitive line (see [PERR92]), some styles are more constrained than others. A layered system, for example, can
(c2 in first layer and t.controlcomponent in second layer))
or
(forall layer : Layer | layer in spawn_layers @
    {t.controlcomponent, c2} setint second layer = {}));
%  outsprawns constrained to between layers or not constrained at all
(foreall t : Trigger | t in triggers and t.subtype = otype @
    t.object in r_getallobjects(globalobjects) setunion
        r_getallobjects({o : Object | exists c : controlcomponents;
            n : Name | n in c.localobjects @ n = o.name}) and
    % trigger object part of the system
t.outcalls subseteq controlcomponents and
    % trigger outcalls part of the system
t.outspawns subseteq controlcomponents and
    % trigger outcalls part of the system
t.inmessage in recognizedmessages and
    % incoming message is a recognized message
t.outmessageset subseteq recognizedmessages);
% outgoing messages are recognized messages
{n : Node |
    (exists c : ControlComponent | c in controlcomponents setunion
        r_getallmethods(globalobjects) setunion
        r_getallmethods(r_getallobjects({o : Object | exists c : controlcomponents;
            n : Name | n in c.localobjects @ n = o.name})) @ n in c.nodes) or
    (exists call : ControlConnector | call in calls @ n in call.nodes) or
    (exists spawn : ControlConnector | spawn in spawns @
        n in spawn.nodes) or
    (exists dc : DataConnector | dc in dataconnectors @ n in dc.nodes})
    subseteq nodes;
% available nodes form a superset of the connector nodes
end schema

The final Z fragment in this section imposes a unique name on every base entity
(and consequently any style-specific entity derived from a base entity).

%  unique name constraints
axiom
    forall p1, p2 : Port @ p1.name = p2.name => p1 = p2;
    forall d1, d2 : DataComponent @ d1.name = d2.name => d1 = d2;
    forall c1, c2 : ControlComponent @ c1.name = c2.name => c1 = c2;
    forall d1, d2 : DataConnector @ d1.name = d2.name => d1 = d2;
    forall c1, c2 : ControlConnector @ c1.name = c2.name => c1 = c2;
    forall o1, o2 : Object @ o1.name = o2.name => o1 = o2;
    forall t1, t2 : Trigger @ t1.name = t2.name => t1 = t2;
    forall s1, s2 : System @ s1.name = s2.name => s1 = s2;
end axiom
(exists thread : Thread | thread in threads @ c in thread) and
  % every control component and every method is either initial, [trigger] called, or [trigger] spawned
  (forall c : ControlConnector | c in calls or c in spawns @ ((c.c1.method = TRUE and c.c2.method = TRUE) =>
    (c.o2 in globalobjects) or
    (exists cc : ControlComponent |
      cc in controlcomponents setunion
      r_getallmethods(globalobjects) setunion
      r_getallmethods(r_getallobjects({o : Object | exists c : controlcomponents; n : Name |
        n in c.localobjects @ n = o.name})) @
      (c.o1.name, c.o2.name) subseteq cc.localobjects) or
    (exists cc1, cc2 : controlcomponents; t1, t2 : threads |
      cc1 /= cc2 and t1 /= t2 and cc1 in t1 and cc2 in t2 @
      c.o1.name in cc1.localobjects and
c.o2.name in cc2.localobjects)) and
  % method-method calls/spawns allowed only to global objects OR
  between local objects in the same controlcomponent OR
  between objects in different threads
  ((c.c1.method = FALSE and c.c2.method = TRUE) =>
    (c.o2 in globalobjects) or
    (c.o2.name in c.c1.localobjects));
  % nonmethod-method calls/spawns allowed only to global objects or
  from a controlcomponent to a local object
  (forall t : Trigger | t in triggers and t.subtype = ctype @
    t.controlcomponent in controlcomponents setunion
    r_getallmethods(globalobjects) setunion
    r_getallmethods(r_getallobjects({o : Object | exists c : controlcomponents; n : Name |
      n in c.localobjects @ n = o.name})) and
    % trigger controlcomponent part of the system
    t.outcalls subseteq controlcomponents and
    % trigger outcalls part of the system
    t.outspawns subseteq controlcomponents and
    % trigger outcalls part of the system
    t.inmessage in recognizedmessages and
    % incoming message is a recognized message
    t.outmessageset subseteq recognizedmessages and
    % outgoing messages are recognized messages
    (forall c2 : ControlComponent | c2 in t.outcalls @
      ((exists layer : Layer | layer in call_layers @
        (t.controlcomponent in first layer and c2 in second layer) or
        (c2 in first layer and t.controlcomponent in second layer))
      or
      (forall layer : Layer | layer in call_layers @
        {t.controlcomponent, c2} setint second layer = {}))) and
    % outcalls constrained to between layers or not constrained at all
    (forall c2 : ControlComponent | c2 in t.outspawns @
      ((exists layer : Layer | layer in spawn_layers @
        (t.controlcomponent in first layer and c2 in second layer) or
        (c2 in first layer and t.controlcomponent in second layer) or

layer in dataconnector_layers @
first layer setunion second layer subseteq
controlcomponents setunion
  r_getallmethods(globalobjects) setunion
  r_getallmethods(r_getallobjects({o : Object |
    exists c : controlcomponents; n : Name |
    n in c.localobjects @ n = o.name})));
% all layers drawn from the controlcomponents & methods
forall call : ControlConnector | call in calls @
  ((exists layer : Layer | layer in call_layers @
    (call.c1 in first layer and call.c2 in second layer) or
    (call.c2 in first layer and call.c1 in second layer))
  or
  (forall layer : Layer | layer in call_layers @
    {call.c1, call.c2} setint second layer = {}));
% calls constrained to between layers or not constrained at all
forall spawn : ControlConnector | spawn in spawns @
  ((exists layer : Layer | layer in spawn_layers @
    (spawn.c1 in first layer and spawn.c2 in second layer) or
    (spawn.c2 in first layer and spawn.c1 in second layer))
  or
  (forall layer : Layer | layer in spawn_layers @
    {spawn.c1, spawn.c2} setint second layer = {}));
% spawns constrained to between layers or not constrained at all
forall d : DataConnector | d in dataconnectors @
  ((exists layer : Layer | layer in dataconnector_layers @
    (d.c1 in first layer and d.c2 in second layer) or
    (d.c2 in first layer and d.c1 in second layer))
  or
  (forall layer : Layer | layer in dataconnector_layers @
    {d.c1, d.c2} setint second layer = {}));
% dataconnectors constrained to between layers or not constrained at all
forall c : ControlComponent; threads : P Thread |
  c in controlcomponents setunion
r_getallmethods(globalobjects) setunion
r_getallmethods(r_getallobjects({o : Object |
  exists c : controlcomponents; n : Name |
  n in c.localobjects @ n = o.name})) and
threads = {thread : Thread |
  (exists topcc : ControlComponent @
    (topcc in initialcontrolcomponents or
     (exists t : triggers @ topcc in t.outspawns
      or topcc in t.outcalls) or
     (exists c1 : ControlComponent; spawn : ControlConnector |
      spawn in spawns @ spawn.c1 = c1
      and spawn.c2 = topcc)) and
    thread = {x : ControlComponent |
      exists chain : P ControlConnector | chain subseteq calls @
      dconnected (topcc, x) = {a,b : ControlComponent |
      exists call : ControlConnector |
      call in chain @ a = call.c1 and b = call.c2})})} @
o.class in classes;
% all global/local objects belong to a system class

initdataconnectors subseteq dataconnectors;
% all initially running dataconnectors belong to declared set of dataconnectors

{c : ControlComponent |
  (exists call : calls @ c = call.c1 or c = call.c2) or
  (exists spawn : spawns @ c = spawn.c1 or c = spawn.c2) or
  (exists dc : dataconnectors @ c = dc.c1 or c = dc.c2)}

subseteq
  controlcomponents setunion
  r_getallmethods(globalobjects) setunion
  r_getallmethods(r_getallobjects({o : Object |
    (exists dc : dataconnectors | dc.cc_or_obj = otype @
    o = dc.o1 or o = dc.o2)}))

subseteq
  r_getallmethods(globalobjects) setunion
  r_getallobjects({o : Object |
    exists c : controlcomponents; n : Name |
    n in c.localobjects @ n = o.name})));
% all components connected by calls, spawns, and/or dataconnectors are members of the system control components or of methods of global/local objects

{o : Object |
  (exists dc : dataconnectors | dc.cc_or_obj = otype @
  o = dc.o1 or o = dc.o2)}

subseteq
  r_getallobjects(globalobjects) setunion
  r_getallobjects({o : Object |
    exists c : controlcomponents; n : Name |
    n in c.localobjects @ n = o.name}))
% all objects connected by dataconnectors are global or local objects

{s : Port | (exists dc : dataconnectors @ s = dc.p1 or s = dc.p2)) subseteq
  {s : Port | (exists c : ControlComponent | c in controlcomponents setunion
    r_getallmethods(globalobjects) setunion
    r_getallmethods(r_getallobjects({o : Object |
      exists c : controlcomponents; n : Name |
      n in c.localobjects @ n = o.name})) @ s in c.ports) or
    (exists o : Object | o in r_getallobjects(globalobjects) or
    o in r_getallobjects({o : Object |
      exists c : controlcomponents; n : Name |
      n in c.localobjects @ n = o.name}) @ s in o.ports))
% dataconnector ports are subset of controlcomponent, method, and (global/local) object ports

forall cset : P ControlComponent; oset : P Object |
  (exists d : DataComponent @ (d, (cset, oset)) in sharreddata)
  @ cset subseteq controlcomponents setunion
  r_getallmethods(globalobjects) setunion
  r_getallmethods(r_getallobjects({o : Object |
    exists c : controlcomponents; n : Name |
    n in c.localobjects @ n = o.name})) and
  oset subseteq r_getallobjects(globalobjects) setunion
  r_getallobjects({o : Object |
    exists c : controlcomponents; n : Name |
    n in c.localobjects @ n = o.name})
% components sharing data are found in controlcomponents/methods and objects sharing data are found in global or local objects

forall layer : Layer |
  layer in call_layers or layer in spawn_layers or
schema System
  name : Name;
  initialcontrolcomponents : P ControlComponent;
    % initially running control components
  initialdataconnectors : P DataConnector;
    % initially running data connectors
  globalobjects : P Object;
    % global objects
  controlcomponents : P ControlComponent;
    % all system control components, excluding object methods
  classes : P Class;
    % allowed system classes
  shareddata : DataComponent +-> (P ControlComponent & P Object);
    % all system shared data
  dataconnectors : P DataConnector;
    % all system data connectors
  calls : P Call;
    % all system blocking control connectors
  spawns : P Spawn;
    % all system non-blocking control connectors
  recognizedmessages : P DataComponent;
    % system recognized messages (events)
  triggers : P Trigger;
    % all system triggers
  call_layers : Layers;
    % layers with respect to calls (top, bottom)
  spawn_layers : Layers;
    % layers with respect to spawns (top, bottom)
  dataconnector_layers : Layers;
    % layers with respect to dataconnectors (top, bottom)
  dataconnectorattributes : DataConnector;
    % the global attributes of the data connectors in the system
  controlconnectorattributes : ControlConnector;
    % the global attributes of the control connectors in the system
  nodes : P Node;
    % available nodes
  resources : Resources;
    % resource usage
  platform : Platform;
    % expected platform

where
  # initialcontrolcomponents + # globalobjects >= 1;
    % there is at least one initially running control component or global object
    in the system
  initialcontrolcomponents subseteq controlcomponents;
    % all initially running control components belong to declared set of control
    components
  forall o : Object |
    o in r_getallobjects(globalobjects) or
    o in r_getallobjects({o : Object | exists c : controlcomponents;n : Name |
      n in c.localobjects @ n = o.name}) @
• **Structural and behavioral completeness and consistency.** For example, all control connectors must connect system control components, objects or object methods.

• **Layering.** These constraints are given with respect to a particular type of connector, i.e. there are call layers, spawn layers, etc. which are not necessarily isomorphic with each other. Our model currently applies layering constraints to control components but not objects.

• **Reachability.** Every control component and object method is constrained to run on system startup, be called, be spawned, or be triggered. We also force all systems to have at least one initial control component or global object on system startup.

• **Environmental consistency.** The system’s entities (control components, objects, etc.) may have environment requirements that conflict with the overall system’s environment.

Later sections will show how systems in other styles can be specified by adding further constraints to the base system.

\[ \text{Layer} \equiv \text{P ControlComponent} \& \text{P ControlComponent} \]
\[ \% \text{Layer is a Z abbreviation definition for two sets of control components (top half and bottom half of layer)} \]

\[ \text{Layers} \equiv \text{P ControlComponent} \leftrightarrow \text{P ControlComponent} \]
\[ \% \text{An abbreviation definition for a set of layers} \]

\[ \text{Thread} \equiv \text{P ControlComponent} \]
\[ \% \text{A thread is a set of control components that are presumably connected by calls (not spawns). “Thread” should only be used in the proper context otherwise use “P ControlComponent”} \]

\[ \% \text{schema for high-level system} \]
% the set of triggered outgoing messages
outcalls : P ControlComponent;
% the set of triggered calls
outspawns : P ControlComponent;
% the set of triggered spawns

where
forall s : Port | s in inports @
  s.iotype = io_in or s.iotype = io_inout;
% in ports accept input
forall s : Port | s in outports @
  s.iotype = io_out or s.iotype = io_inout;
% out ports allow output
subtype = ctype =>
inports setunion outports subseteq controlcomponent.ports and
controlcomponent.method = FALSE;
% if the trigger is for a control component, the ports are indeed found in the
specified control component, and it is not a method
subtype = otype =>
inports setunion outports subseteq object.ports and
outcalls subseteq object.methods and
outspawns = {};
% if the trigger is for an object, the ports are indeed found in the specified
object. Calls are to object methods, and no spawns are allowed of
object methods (artificial constraint).
end schema

The last base entity we introduce is the **system**. Systems are built out of the previously defined base entities. They contain control components, objects, and different connectors (shared data, calls, spawns, triggers). When a system is started, a set of initial control components, initial dataconnectors, and global objects are activated. The control connectors and triggers determine what other control components or objects are activated. Systems also include resource, platform, and node commitments which can be checked for compatibility with the entities which are used to construct them.

*Base systems* contain a number of constraints that are common to systems of different styles. These constraints fall into four categories:
resources : Resources;
% resource usage
platform : Platform;
% expected platform

where
{n : Node | n in c1.nodes or n in c2.nodes} subseteq nodes;
% the connector nodes form a superset of the component nodes
c1.method = TRUE <=> c1 in o1.methods and
c2.method = TRUE <=> c2 in o2.methods;
% if either component is a method, it belongs to the respective object
arguments = c2.arguments;
returnvalues = c2.returnvalues;
end schema

Call == \{c : ControlConnector | c.blocking = TRUE\}

Spawn == \{c : ControlConnector | c.blocking = FALSE\}

axiom
forall c : Spawn @ c.c1.method = FALSE and c.c2.method = FALSE;
% impose artificial constraint: no spawns to/from object methods
end axiom

Another connector or form of interaction is the trigger. This models the
association between receiving an event or message by a control component or an
object, and its consequent engagement in a control or data transfer. As above for
control connectors, we constrain triggered spawns to not include object methods.
In this way, objects remain single-threaded entities.

% schema for high-level trigger
schema Trigger
name : Name;
subtype : COsubtype;
% indicate if an object or a control component is being connected
controlcomponent : ControlComponent;
% the control component which receives the data message
object : Object;
% the object which receives the data message
inmessage : DataComponent;
% the input message
inports : P Port;
% the input port which the message come in over
outports : P Port;
% the output ports for any outgoing messages
outmessageset : P DataComponent;
Another base connector found in many styles is the **control connector**. It comes in two ‘flavors’: blocking or non-blocking. The blocking variant represents a traditional ‘call’ and the non-blocking variant represents a ‘spawn’ (‘fork’ in UNIX parlance). Control connectors are laid between two control components (which may also be methods of objects). Our model currently imposes the artificial constraint that spawns are not allowed to or from methods. The reason for this is that we do not yet have a strong model of the effects of spawning a method for an object which may go out of scope (and therefore be destroyed).

```plaintext
% schema for high-level control connector
schema ControlConnector
  name : Name;
  c1, c2 : ControlComponent;
  % connected control components
  o1, o2 : Object;
  % connected objects (if the control components are methods)
  arguments : seq DataType;
  % arguments passed in during a call or spawn (if any)
  returnvalues : seq DataType;
  % values when returning from a call or spawn (if any)
  rate : N1;
  % maximum transfer rate
  queuesize : N;
  % queuesize - 0 if connector does not queue requests
  blocking : Boolean;
  % true if the transfer blocks waiting for a response
  reliable : Boolean;
  % true if each transfer is reliably sent
  nodes : P Node;
  % nodes it resides on
```

The next base entity is the **data connector**. It may be used to connect two control components or two objects.

**COsubtype ::= ctype | otype**  
% a useful type for indicating whether a property applies to control components or objects

**Directionality ::= forward | reverse | twoway**  
% a useful type for data connectors

% schema for high-level data connector

```
schema DataConnector
  name : Name;
  p1, p2 : Port;
    % connected ports
  c1, c2 : ControlComponent;
    % connected control components
  o1, o2 : Object;
    % connected objects (if the control components are methods or if the particular spec hasn't been refined to include specific methods)
  cc_or_obj : COsubtype;
    % indicate if an object or a control component is being connected
  dataformat : seq DataType;
    % format of the data expected to be transferred
  rate : N1;
    % maximum transfer rate
  directionality : Directionality;
    % indicate direction of communications
  buffersize : N;
    % buffersize - 0 if not buffered
  reliable : Boolean;
    % true if each transfer is reliably sent
  nodes : P Node;
    % nodes it resides on
  resources : Resources;
    % resource usage
  platform : Platform;
    % expected platform
```

**where**

cc_or_obj = ctype => p1 in c1.ports and p2 in c2.ports;
cc_or_obj = otype => p1 in o1.ports and p2 in o2.ports;
% the connected ports belong to the specified components or objects

\[\text{c1.method = TRUE} \Rightarrow \text{c1 in o1.methods};\]
\[\text{c2.method = TRUE} \Rightarrow \text{c2 in o2.methods};\]
% if either component is a method, it belongs to the respective object
forall c : methods @
  c.method = TRUE and
  c.nodes subseteq nodes and
  c.platform PlatformCompatible platform;
%  all the methods reside on object nodes, and are compatible with the
  object’s platform
addResources {r : Resources | exists m : methods @ r = m.resources}
  ResourceCompatible resources;
%  combined method resources are compatible with object resources
end schema

%  ensure no methods exist without a parent object
axiom
  forall c : ControlComponent @
    c.method = TRUE <=> (exists o : Object @ c in o.methods);
end axiom

Note that objects impose resource and platform constraints on the methods they
contain.

The next Z fragment specifies two functions that will be useful in many
places later in the specification. The first function recursively returns all the objects
that are declared in the methods of a given set of objects. The second function
recursively returns all the methods of a set of objects and any locally declared
object’s methods.

%  two functions for retrieving objects and methods
global
  r_getallobjects : P Object +-> P Object;
  %  recursively gets all objects accessible via input objects’ methods
  r_getallmethods : P Object +-> P ControlComponent;
  %  recursively gets all methods accessible via input objects’ methods
axiom
  forall inobjects, outobjects : P Object @
    r_getallobjects(inobjects) = outobjects <=>
      outobjects = inobjects setunion r_getallobjects({o : Object |
        (exists inob : inobjects; m : ControlComponent |
        m in inob.methods @ o.name in m.localobjects)});
  forall inobjects : P Object; methods : P ControlComponent @
    r_getallmethods(inobjects) = methods <=>
      methods = {m : ControlComponent |
        (exists inob : inobjects @ m in inob.methods)}
      setunion r_getallmethods({o : Object |
Note that the locally declared objects (‘localobjects’) of a control component are only referred to by name, as opposed to the locally declared data variables (‘locals’) which are referred to directly. This is due to Z’s inability to make a forward reference to an entity which has not been declared yet - in this case objects which are defined next. This has forced us to introduce the constraint of unique names for objects, and for that matter all entities (see below).

The definitions for objects and classes are given below:

[Class]
% given set of classes
schema Object
  name : Name;
  class : Class;
  % class the object belongs to
  ports : P Port;
  % object ports
  data : P DataComponent;
  % private data of the object
  methods : P ControlComponent;
  % control components which act as the object’s methods
  nodes : P Node;
  % nodes it resides on
  resources : Resources;
  % resource usage
  platform : Platform;
  % expected platform
where
We highlight two ports in the fragment below, ‘standard input’ and ‘standard output’, that are often present in most systems, but which are not associated with any single control component or object. The ‘standard input’ port models the capability of a system to accept data entered interactively by a user, and the ‘standard output’ port models the capability to print out data to a printer or screen for example.

```
% declare two common ports that are not attached to any particular control component or object

global
    stdin, stdout : Port;
axiom
    stdin.iotype = io_in;
    stdout.iotype = io_out;
end axiom
```

Next, we define the attributes for data components and control components.

```
% schema for high-level data component

schema DataComponent
    name : Name;
    type : seq DataType;
        % type of the data determined by internal structure
    nodes : P Node;
        % nodes it resides on
    resources : Resources;
        % resource usage
end schema
```

Though a data component has a ‘resources’ attribute, only its (the ‘resources’ schema) ‘datasize’ attribute is relevant in this context.

```
% schema for high-level control component

schema ControlComponent
    name : Name;
    ports : P Port;
    arguments : seq DataType;
        % expected arguments if called or spawned
    returnvalues : seq DataType;
        % type of returned values when returning from a call or spawn
    locals : P DataComponent;
```
tors that are missing in our model. For example, a source code view might include a ‘depends-on’ connector between two or more control components, akin to the “#include” C programming language construct.

6.2.4.2 Specification of Base Entities

We can now present the formal definition of each of the eight base entities in our model. We begin with ports (preceded by a few primitive declarations):

```
[Name]
% given set of names for components, connectors, etc.

Boolean ::= TRUE | FALSE
% boolean type

[DataType]
% given set for primitive data types
%
% declare a few common primitive data types (can be extended)
global
character, integer, float, array, structure, string : DataType;
end global

IoType ::= io_in | io_out | io_inout
% for indicating the I/O type of a port
%
% schema for high-level (data) port
schema Port
  name : Name;
  iotype : IoType;
  % handles input, output, or both
  dataformat : seq DataType;
  % format of the expected data
  rate : N1;
  % maximum input and/or output rate
  buffersize : N;
  % buffersize - 0 if not buffered
  streaming : Boolean;
  % true if the transfer streams data incrementally
  blocking : Boolean;
  % true if the transfer blocks on output (not applicable to io_in ports)
  maximumconnections : N1;
  % maximum connections allowed
end schema
```
A triggered call (connector number 7 in the figure) can only be made from an object to a control component, or from a control component to another control component.

Our model currently does not support a small set of possible connectors (these are indicated in the figure by placing a set of parentheses around the number of the connector). The missing connectors mostly revolve around the possible dynamic creation of data components or objects at run-time. However, keeping in mind that the set of connectors we have focused on in the figure are drawn from the structural and behavioral views, there are actually many other types of connec-
• an object can include complex interconnections within itself, forming its own single-threaded minisystem (see the definition of system below)

• an object can be instantiated many times, each time introducing a new set of different data and control component copies

5. A **data connector** models the potential for two or more control components or objects to engage in data transfers amongst themselves.

6. A **control connector** models the potential for two or more control components to engage in control transfers (possibly with data) amongst themselves.

7. A **trigger** associates an action (a data or control transfer) with the reception of a data component by a control component or object.

8. A **system** is a non-empty set of interconnected control components or objects satisfying some unique purpose.

While it was relatively easy to determine the components of the base entities (i.e. data component, control component, object, and system), it was not so easy to determine the connectors. One of the weaknesses of our model, in fact, is that it does not reflect *all* the possible fundamental types of connectors. Even if we concentrate purely on the structural and behavioral views, we can identify several types of possible connectors (a better word might be *interactions*) between data components, control components, and objects (see figure 10). For example, a shared data connector (connector number 3 in the figure) can be extended between an object with another object, a control component with another control component, and an object with a control component (and vice-versa). However, a trig-
1. A **port** is typically associated with a control component, and is the latter’s entry and exit points for data during data transfers.

2. A **data component** models data that is used to store state or is transferred across data connectors.

3. A **control component** models data that is executed by the underlying machine and which can initiate (and respond to) data and control transfers. It is assumed to have a single thread of control (from a hardware point of view, a control component has a single program counter).

4. An **object** is an encapsulation of a set of data components with a set of control components. Not only from a formal standpoint is an object fundamentally different from a data or control component (i.e. \{X\} is not the same type as X), but there are other reasons for this distinction,

### Table 5: Generalized components and connectors

<table>
<thead>
<tr>
<th>Main/Sub.</th>
<th>Dist. Processes</th>
<th>Pipe &amp; Filter</th>
<th>Event-Based</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td>Data Structure</td>
<td>Data</td>
<td>Event</td>
<td>Data Component</td>
</tr>
<tr>
<td>Procedure</td>
<td>Process</td>
<td>Filter</td>
<td>Procedure</td>
<td>Control Component</td>
</tr>
<tr>
<td>Call</td>
<td>Call</td>
<td></td>
<td>Call</td>
<td>Control Connector</td>
</tr>
<tr>
<td>M/S System</td>
<td>D/P System</td>
<td>P/F System</td>
<td>E/B System</td>
<td>System</td>
</tr>
<tr>
<td>Circuit</td>
<td>Pipe</td>
<td></td>
<td></td>
<td>Data Connector</td>
</tr>
<tr>
<td>Call</td>
<td>Spawn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Callback Trigger</td>
<td></td>
<td></td>
<td></td>
<td>Trigger</td>
</tr>
</tbody>
</table>

1. **port**
2. **data component**
3. **control component**
4. **object**
NoResources.mainmemory = 0;
NoResources.networkbandwidth = 0;
forall inputset : P Resources; r1, r2, r3 : Resources @
  (inputset = {} => addResources inputset = NoResources) and
  (inputset = {r1} => addResources inputset = r1) and
  ({(r1, r2) subseteq inputset and r3 notin inputset} =>
     addResources(inputset) = addResources((inputset \ {r1, r2})
     setunion {r3}) and
     r3.sloc = r1.sloc + r2.sloc and
     r3.datasize = r1.datasize + r2.datasize and
     r3.codesize = r1.codesize + r2.codesize and
     r3.cpuload = r1.cpuload + r2.cpuload and
     r3.mainmemory = r1.mainmemory + r2.mainmemory and
     r3.networkbandwidth = r1.networkbandwidth +
     r2.networkbandwidth);
forall r_actual, r_limits : Resources @
  r_actual ResourceCompatible r_limits <=>
  r_actual.sloc <= r_limits.sloc and
  r_actual.datasize <= r_limits.datasize and
  r_actual.codesize <= r_limits.codesize and
  r_actual.cpuload <= r_limits.cpuload and
  r_actual.mainmemory <= r_limits.mainmemory and
  r_actual.networkbandwidth <= r_limits.networkbandwidth;
end axiom

6.2.4 Base (‘style-less’) Entities

In our initial approach to modeling architectural styles (section 5.0 and see [ABDA95] for more details), we described the components and connectors of four different styles: pipe/filter, main/subroutine, distributed processes, and event-based (implicit invocation). Although this approach suffered from problems described earlier, it did provide us with insights for a common underlying set of entities (components and connectors) that can be specialized into each style’s particular set of entities.

6.2.4.1 Overview of Base Entities

We have identified eight base entities based on the four styles (table 5), informally defined here:
in a single monolithic schema (‘Resources’). This schema can be included in other schemas (e.g. components or connectors), with some or all of the attributes being relevant in the particular context. The context will also determine whether the an attribute means ‘supplied resource’ or ‘demanded resource’.

\[
\text{Percent} \equiv \{n : \mathbb{N} \mid n \leq 100\}
\]
% percent type

\[
\text{schema Resources}
\]
\[
\begin{align*}
\text{cpuload} & : \text{Percent}; \\
\text{sloc} & : \mathbb{N}; \\
\text{datasize} & : \mathbb{N}; \\
\text{codesize} & : \mathbb{N}; \\
\text{mainmemory} & : \mathbb{N}; \\
\text{networkbandwidth} & : \mathbb{N};
\end{align*}
\]
% cpu load (percent of total load)
% source lines of code (lines)
% data size (bytes)
% code size (bytes)
% main memory (bytes)
% network bandwidth (bytes per second)
\[
\text{end schema}
\]

Simple resource compatibility is formally defined below. A pair of resource sets are compatible with each other if the first set’s resources are all less than or equal to the second’s. In this case, the first set of resources represents the actual set of resources that are demanded, and the second set represents the available limits.

\[
\text{%% inrel ResourceCompatible}
\]

\[
\text{global}
\]
\[
\text{NoResources} : \text{Resources};
\]
% ‘zero’ for resources schema
\[
\text{addResources} : (\text{P Resources}) \leftrightarrow \text{Resources};
\]
% adds a set of resources together
\[
\text{ResourceCompatible} \_ : \text{Resources} \leftrightarrow \text{Resources};
\]
% determine if one set of resources is compatible with another
\[
\text{axiom}
\]
\[
\text{NoResources.sloc} = 0; \\
\text{NoResources.datasize} = 0; \\
\text{NoResources.codesize} = 0; \\
\text{NoResources.cpuload} = 0;
\]
port the former. Platform and platform compatibility (captured by a simple equality relation) are formally specified below.

\[
\text{% schema for platform options}\n\text{schema Platform}\n\quad \text{cpumodel} : \text{PlatformOption[CpuModelType]};\n\quad \text{discommmech} : \text{PlatformOption[DistributedCommunicationMechanismType]};\n\quad \text{language} : \text{PlatformOption[ImplementationLanguageType]};\n\quad \text{os} : \text{PlatformOption[OperatingSystemType]};\n\quad \text{gui} : \text{PlatformOption[GraphicalUserInterfaceType]};\n\text{end schema}\n\text{%% inrel PlatformCompatible}\n\text{global}\n\quad _\text{PlatformCompatible}_ : \text{Platform} \leftrightarrow \text{Platform};\n\quad \text{% determine if two platforms are compatible with each other}\n\text{axiom}\n\quad \forall p1, p2 : \text{Platform} @ p1 \text{ PlatformCompatible } p2 \iff\n\quad \quad p1.\text{cpumodel} = p2.\text{cpumodel} \text{ and}\n\quad \quad p1.\text{discommmech} = p2.\text{discommmech} \text{ and}\n\quad \quad p1.\text{language} = p2.\text{language} \text{ and}\n\quad \quad p1.\text{os} = p2.\text{os} \text{ and}\n\quad \quad p1.\text{gui} = p2.\text{gui};\n\text{end axiom}\n\]

We also introduce the notion of a node, a piece of hardware which will execute part or all of the software system.

\[
\text{schema Node}\n\quad \text{platform} : \text{Platform};\n\text{end schema}\n\]

6.2.3 Useful resource attributes and constraints

In addition to the platform, a software system will consume certain resources in its environment. An architect can estimate the expected consumption to determine if a particular hardware configuration will satisfy the system’s needs. The resources modeled below include cpu load, source lines of code, data size, code size, main memory, and network bandwidth. We place all resources together
required operating system, and the required user interface (if any). Each option is not specified any more than its name essentially. Extending the model to include the intricacies of each platform option is a future goal (e.g. the type of cpu might be expanded to indicate the size of its data bus, maximum address space, clock speed, etc.).

\[
[CpuType, DistributedCommunicationMechanismType, OperatingSystemType, GraphicalUserInterfaceType, ImplementationLanguageType]
\%
sets of platform options

global
  sparc, i80x86, m680x0, powerpc : CpuType;
  rpc, messages, sharedmemory : DistributedCommunication-MechanismType;
  sunos, solaris, aix, windows3, windows95,
   dos, os2, system7, vms : OperatingSystemType;
  xwindowsgui, windows3gui, windows95gui,
  system7gui, os2gui : GraphicalUserInterfaceType;
  c, cplusplus, ada, pascal, prolog, lisp, fortran,
  cobol, smalltalk, basic : ImplementationLanguageType;
end global
\%
partial listing of the possible platform options

For each one of these options, we add the vendor and version - these two pieces of information are often vital. We use a generic schema to capture all the options since they are almost identical to one another.

\[
[VendorType, VersionType]
\%
schema PlatformOption [X]
  type : X;
  vendor : VendorType;
  version : VersionType;
end schema
\%
partial listing of the possible platform options

We can use the above declarations to specify not only what the software demands, but also what the platform supplies. The constraints of the software can be checked against the platform to determine if the latter can adequately sup-
generic [X]
    uring : P X <-> (X <-> X)
    % true iff an undirected graph is a ring
where
    forall nodes : P X; edges : X <-> X @
    uring nodes = edges <=>
        ((#nodes > 2 and dom edges = nodes and ran edges = nodes
        and #edges = #nodes and
        (forall a : nodes @
            #{x, y : X | (x,y) in edges and (x=a or y=a)} = 2))
        % the number of edges is equal to the number of nodes, and each
        % node is found in 2 edges
        or
        (#nodes = 0 or #nodes = 1 or #nodes = 2))
    % trivial case: a ring of zero, one, or two nodes
end generic

generic [X]
    ufullyconnectedgraph : P X <-> (X <-> X)
    % true iff an undirected graph is fully connected
where
    forall nodes : P X; edges : X <-> X @
    ufullyconnectedgraph nodes = edges <=>
        ((#nodes > 1 and dom edges = nodes and ran edges = nodes
        and #edges = (#nodes * (#nodes - 1)) div 2 and
        (forall a,b : nodes @
            (a,b) in edges or (b,a) in edges))
        % the number of edges is equal to the number of nodes multiplied by
        % the same minus one, all divided by two, and an edge exists
        % between all pairs of nodes
        or
        (#nodes = 0 or #nodes = 1))
    % trivial case: a fully connected graph of zero or one node
end generic

6.2.2 Useful platform attributes and constraints

The platform for software systems can vary tremendously from one
system to another. To cover the large variation, we introduce partially defined sets
of platform options that can be further elaborated per architecture or per style. The
sets we model address the following platform options: the cpu model, the distrib-
uted communication mechanism used (if any), the implementation language, the
and \( \#\text{edges} = \#\text{nodes} - 1 \)
and (forall a, b : nodes @ u\text{connected}(a, b) = \text{edges})
% the number of edges is one less than the number of nodes, and all
nodes are connected to each other

or
\( \#\text{nodes} = 0 \) or \( \#\text{nodes} = 1 \))
% trivial case: a tree of zero or one node
end generic

generic [X]
dtree : (X & P X) <-> (X <-> X)
% true iff a directed graph is a tree with respect to a given `root' node
(where the graph is represented as a set of nodes and a set of
edges)
where
forall root : X; nodes : P X; edges : X <-> X | root in nodes @
dtree(root, nodes) = edges <=>
((\#nodes > 1 and dom edges = nodes and ran edges = nodes
and \#edges = \#nodes - 1
and (forall a : nodes @ d\text{connected}(root, a) = \text{edges}))
% the number of edges is one less than the number of nodes, and all
nodes are connected to the root
or
\( \#\text{nodes} = 0 \) or \( \#\text{nodes} = 1 \))
% trivial case: a tree of zero or one node
end generic

generic [X]
ustar : P X <-> (X <-> X)
% true iff an undirected graph is a star
where
forall nodes : P X; edges : X <-> X @
ustar(nodes = edges <=>
((\text{utree}(nodes = edges)) and
(exists center : X @
\#nodes > 1 =>
(forrall a, b : X | (a, b) in edges @ a = center or b = center)))
% the graph is a tree, and one node is found in all the edges
end generic

generic [X]
dstar : (X & P X) <-> (X <-> X)
% true iff a directed graph is a star
where
forall center : X; nodes : P X; edges : X <-> X | center in nodes @
dstar(center, nodes) = edges <=>
((\text{dtree}(center, nodes) = edges) and
\#nodes > 1 =>
(forrall a, b : X | (a, b) in edges @ a = center))
% the graph is a tree, and one node is found in all the edges (the root)
end generic
architecture can be described as a directed graph - not necessarily a tree - with all
nodes connected to the main routine). Although there are an infinite number of
topological patterns, we only describe eight useful ones below: connected undi-
rected graph, connected directed graph, undirected tree, directed tree, undirected
star, directed star, undirected ring, and undirected fully connected graph.

generic [X]
cugraph : P X <-> (X <-> X)
% true iff an undirected graph is connected (where the graph is repre-
sented as a set of nodes and a set of edges)

where
forall nodes : P X; edges : X <-> X @
cugraph nodes = edges <=>
((#nodes > 1 and dom edges = nodes and ran edges = nodes
and (forall a, b : nodes @ uconnected (a,b) = edges))
% all nodes are connected to each other
or
(#nodes = 0 or #nodes = 1))
% trivial case: a graph of zero or one node
end generic

generic [X]
cdgraph : (X & P X) <-> (X <-> X)
% true iff a directed graph is connected with respect to a given `root' node
(where the graph is represented as a set of nodes and a set of
edges)

where
forall root : X; nodes : P X; edges : X <-> X | root in nodes @
cdgraph (root, nodes) = edges <=>
((#nodes > 1 and dom edges = nodes and ran edges = nodes
and (forall a : nodes @ dconnected (root, a) = edges))
% all nodes are connected to the root
or
(#nodes = 0 or #nodes = 1))
% trivial case: a graph of zero or one node
end generic

generic [X]
utree : P X <-> (X <-> X)
% true iff an undirected graph is a tree (where the graph is represented as
a set of nodes and a set of edges)

where
forall nodes : P X; edges : X <-> X @
utree nodes = edges <=>
((#nodes > 1 and dom edges = nodes and ran edges = nodes
6.2 Model Specification

6.2.1 Useful topological constraints

Many of the constraints introduced by the topological view are naturally modeled by concepts from elementary graph theory. We first introduce the notion of a connected pair of nodes in an undirected graph, and in a directed graph.

```
generic [X]
  uconnected : (X & X) <-> (X <-> X)
  % true iff a pair of nodes are connected by a set of edges in an undirected
  % graph (where the edges are given as a set of binary tuples of nodes)
where
  forall a, b : X; edges : X <-> X @
    uconnected (a,b) = edges <=>
      ((exists c : X @
        ((a,c) in edges or (c,a) in edges) and
        uconnected (c, b) = edges)
      % an edge exists from node a to some other node c, and nodes b and
      % c are connected (recursive definition)
      or
      (exists a : X @ a=b))
    % trivial case: each node is connected to itself
end generic

generic [X]
  dconnected : (X & X) <-> (X <-> X)
  % true iff a pair of nodes are connected by a set of edges in a directed
  % graph (where the edges are given as a set of binary tuples of nodes)
where
  forall a, b : X; edges : X <-> X @
    dconnected (a,b) = edges <=>
      ((exists c : X @
        (a,c) in edges and dconnected (c, b) = edges)
      % an edge exists from node a to some other node c, and nodes b and
      % c are connected (recursive definition)
      or
      (exists a : X @ a=b))
    % trivial case: each node is connected to itself
end generic
```

Building on these notions of connectivity, we can define functions which determine if a given graph (undirected or directed) meets certain properties. These functions will be useful for different styles (e.g. the topology of a main/subroutine
6.0 Another Approach: A Uniform Representation

The problem of formally modeling heterogeneous architectural composition is greatly simplified by having a *uniform representation* of different styles. Such a model easily and naturally supports the description of composing a pipe/filter system with a distributed-processes system for example. Another benefit of a uniform representation is that common attributes and constraints are easily shared amongst different styles. We can construct a ‘style-less’ or ‘base’ set of entities (components and connectors) that embody attributes and constraints common to many styles. Style-specific entities can inherit attributes from the base entities, and add their own particular constraints.

6.1 Outline of Model Specification

Our model of architectural composition can be divided into several parts. In the following section (section 6.2), we will present those parts of the model which deal with the representation of different styles, namely:

- Useful topological constraints (section 6.2.1)
- Useful platform attributes and constraints (section 6.2.2)
- Useful resource attributes and constraints (section 6.2.3)
- Base entities (section 6.2.4)
- Style-specific extensions to base entities (section 6.2.5 and section 6.2.6)

In later sections, we will describe the composition operations.
situation much more difficult. Composition can be modeled as an operation on several systems, but if the systems are of different formal types, then the formal specification of the operation becomes needlessly complex. Each pair of styles requires a different composition operation (figure 9).

Another disadvantage to this approach of using different types to build different styles is that it masks common attributes and constraints that may occur across styles. Many styles, for example, contain variants of computational components with common attributes (e.g. procedures, filters, and processes may all possess local data). These variants use different names for what is essentially the same structural entity.

In light of these problems, we decided to search for another approach which would simplify the specification of composition, and which would highlight common attributes and constraints that are shared by different styles. This led to an approach that is essentially based on a uniform representation.
schema MainSubroutineSystem
   name : Name
   % name of the system
   globaldata : P DataStructure
   % a set of global variables
   procedures : P Procedure
   % a set of procedures
   procedurecalls : Procedure <-> Procedure
   % the set of allowed procedure calls
   inputstream, outputstream : Socket
   % data input and output streams
where
   main in procedures
   % the main routine is part of the set of procedures
   forall d : DataStructure | d in globaldata @
      d.validaccessors = procedures
   % global variables are accessible to all procedures
end schema

schema EventBasedSystem
   name : Name
   % the name of the system
   allowedobjects : P EventBasedObject
   % a set of event based objects
   initialobjects : P EventBasedObject
   % the initial set of running objects
   recognizedevents : P EventType
   % the set of events the manager can respond to
   registry : EventType <-> EventBasedObject
   % a relation showing which events are interesting to which objects
where
   initialobjects subeq allowedobjects
   % the initial set of objects is a subset of the allowed set
   dom registry subeq recognizedevents
   % the events of interest to objects are events which the manager can recognize
   ran registry subeq allowedobjects
   % registered objects are in the set of allowed objects
end schema

5.1 Weaknesses of this Approach

The most important characteristic to note about these three schemas is that they are not the same type (Z has an extremely strong name-based typing system). Consequently, operations that take pipe/filter systems as input, for example, will not work on main/subroutine systems. This makes the specification of compo-
5.0 Initial Approach: Style-Specific Models

Our initial approach to modeling architectural styles was to construct a representation of each style independently from the others (see the qualifying proposal for more details on this approach [ABDA95]). Four styles were formally specified:

- pipe and filter
- main/subroutine
- distributed processes and threads
- event-based

Each style was described in terms of its components and connectors, as well as the operations they could engage in. The entire specification is not given here, but instead we simply show three fragments, namely: the schemas for pipe/filter-style, main/subroutine, and event-based systems.

```
schema PipeFilterSystem
    name : Name
    % name of the system
        filters : P Filter
    % the set of running filters
        pipes : P Pipe
    % the set of running pipes
where
    forall p : Pipe | p in pipes @
        exists f1, f2 : Filter | f1 in filters and f2 in filters @
            p.sender in f1.outsockets and
            p.receiver in f2.insockets
    % all pipe sockets connect filter sockets
    forall p1, p2 : Pipe | {p1, p2} subeq pipes @
        (p1.sender = p2.sender and p1.receiver = p2.receiver =>
            p1 = p2)
    % no two pipes connect the same sender and receiver
end schema
```
tion mechanism (if any), the implementation language, etc. A list of the choices available for the underlying hardware cpu type, for example, might include “sparc”, “i80x86”, “powerpc”, etc. In addition to the platform, a software system will consume certain resources in its environment. An architect can estimate the expected consumption to determine if a particular hardware configuration will satisfy the system’s needs. The resources we model include cpu load, memory usage, disk space, and network bandwidth.

4.2.5 A Note on Representing Different Views Together

We have described how architectures and architectural styles can be formally specified in terms of many different views. The specification will not, however, cleanly separate the views from each other. The attributes and constraints for each view are actually interleaved with each other in the component and connector schemas of the specification (figure 8).

![Component/connector schema](image)

<table>
<thead>
<tr>
<th>component/connector schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>structure attributes</td>
</tr>
<tr>
<td>topology attributes</td>
</tr>
<tr>
<td>behavior attributes</td>
</tr>
<tr>
<td>environment attributes</td>
</tr>
<tr>
<td>structure constraints</td>
</tr>
<tr>
<td>topology constraints</td>
</tr>
<tr>
<td>behavior constraints</td>
</tr>
<tr>
<td>environment constraints</td>
</tr>
</tbody>
</table>

...more components/connectors...

Figure 8: An Architectural Specification Consolidating Multiple Views
grams, traces, etc.). We model only some important attributes here that are relevant to connectors in particular at the architectural level. The following Z fragment is an example of the attributes that will be added to the components and connectors of various styles.

```
rate : N1;
% maximum input and/or output rate of a socket or connector
buffersize : N;
% buffersize - 0 if not buffered
bidirectional : Boolean;
% one-way or two-way transfers supported
synchronous : Boolean;
% true if each transfer requires an acknowledgement
blocking : Boolean;
% true if the transfer blocks waiting for a response
reliable : Boolean;
% true if each transfer is reliably sent
streaming : Boolean;
% true if the data is streamed across (no message boundaries)
```

### 4.2.4 Environment

The original description of software architectural styles by Garlan and Shaw did not address issues such as integration with other systems, platform requirements, and resource consumption. These issues are the focus of another architectural view, namely the environment view. A classic example of some of these issues can be found in a recent paper describing the experience of integrating four complex software systems together [GARL94]. Researchers in systems integration have developed many different techniques for addressing these issues [NG90].

Platform requirements are essentially captured by fixed sets of choices for options such as the underlying hardware cpu type, the distributed communica-
The notion of a dynamic topology is important for many reasons, but for our purposes we emphasize it for the composition problem. Architectural composition needs to be examined from as many views as are being considered. For example, if the architectural specifications of two different systems include constraints from the environmental view, the composition of these two systems must be examined with respect to the environmental view; in other words, we must look for constraint mismatches in that view. However, a dynamic topology makes this problem more difficult since it introduces nondeterminism regarding how many components and/or connectors are actually executing - and consuming environmental resources. Thus, a dynamic topology basically makes architectural analysis more difficult. This is especially true when an architecture contains recursive calls or cycles of control connectors.

As will be shown in later sections, a dynamic topology will often require the architect to specify the initial set of components and connectors, as well as the (total) allowed set. The initial set is assumed to begin running immediately when the system is activated. The allowed set are extra components and connectors that can be invoked during execution.

4.2.3 Behavior

The behavioral view of an architecture or style places constraints on the operations which the components and connectors may engage in. This view has been extensively modeled in many different ways in the past (statecharts, flow dia-
To reflect all the possible topologies that may arise dynamically, we specify the topology of an architecture by showing all potential components and connectors. This ‘total’ topology constrains the communications and invocations of components. For example, the topology of a simple architecture in the distributed processes style is given in figure 6. Three processes will comprise the system.

![Figure 6: Architectural topology for a simple system](image)

There are data connectors between processes 1 and 2, and processes 3 and 2. Process 1 may spawn process 3. Assuming that processes 1 and 2 are running at system startup, snapshots of the dynamic topology at different times may reveal different sets of components and connectors as in figure 7 - however, both sets are allowed by the specified topology.

![Figure 7: Two dynamic topologies satisfying the same architecture](image)
Control signals, the reality is that they are actually data signals between already running components.

In software, the situation is different. In addition to data transfers, there are truly legitimate control transfers: the topology can dynamically change. Components and connectors which are executing concurrently may be added and removed while the system is running. For example, in the distributed processes style, a process spawn adds a new instance of some process to the current set of running processes. In a main/subroutine style system, a recursive call can generate many copies of a single procedure, each consuming its own share of resources.
4.2.2 Topology

The topological view of an architecture or style describes the allowed interconnections between components. All the possible data and control transfers are explicitly identified. We use this view to specify any expected or imposed patterns in the interconnected set of components - including any layering constraints. Besides these topological patterns, however, there is a fundamental difference between software and hardware architectures that complicates the composition problem greatly. It is related to both the topology and the behavior of a software system.

4.2.2.1 The Dynamic Topology of Software

One major difference between hardware and software architectures is that the topology of software architectures typically changes over time. The topology of a hardware architecture is assumed to stay constant during the period of its execution. This assumption applies at all the different levels of abstraction that can be used to model hardware (figure 4). For example, the topology of a full-adder expressed at the register transfer level can be described using two half-adders and an or-gate (figure 4). At any point in time during the regular use of the full-adder, we are virtually assured that no additional components will be added to the system regardless of the behavior. Another way of expressing is this is to note that all connectors in hardware are data connectors. Though hardware designers speak of con-
(with all its attributes and constraints), and then add its own specific attributes or
constraints (in either the same or different views addressed by the inherited style).

4.2.1 Structure

The structural view of an architecture (and similarly of a style) describes the possible components and connectors out of which the system will be built. In a main/subroutine-style architecture, for example, the structure includes data variables, procedures, and procedure calls. A pipe/filter-style architecture includes only pipes and filters. Systems are instantiated or built out of the components and connectors supplied by its inherited style.

Each type of component and connector which is supplied by a style is modeled in our specification by a schema which details its structural attributes and constraints. For example, the specification of a procedure in the main/subroutine style might read as follows (ignoring the type of the attributes for the moment):

```plaintext
schema Procedure
    name : Name;
    ports : P Port;
    % possible data ports for circuits
    arguments : seq DataType;
    % expected arguments if called or spawned
    returnvalues : seq DataType;
    % type of returned values when returning from a call or spawn
    locals : P DataVariable
    % local variables
where
    ports = {};
    % a procedure is restricted to communicating via calls or shared data
end schema
```
Whereas a software architecture can include many different views in its specification (figure 3), the software architecture *styles* originally listed by Garlan and Shaw ([GARL93]) are more limited. Most of the styles mentioned in that report are characterized by their attributes and constraints within the structural, topological, and behavioral views. For example, the main/subroutine style says nothing about dynamic resource usage such as cpu load or memory. In this report, however, we expand the notion of style slightly to include other views. For our purposes, an architectural style is formally modeled as a collection of attributes and constraints *primarily* within the structure, topology, and behavior views, but also within other architectural views. A particular architecture can inherit a style
and some views are very domain-specific as mentioned above. We have chosen to support only a small subset of views in our model (each to varying degrees): **structure, topology, behavior, and environment** - although we do touch on some factors of other views. For each view, we model its essential attributes and constraints, and also the way in which constraint mismatches or conflicts may occur within that view.
ture view which describes the data and control components and connectors out of which the system is built. Another view, closely related to the structural, is the topology view which describes any expected patterns in the interconnected set of components (including layering constraints). A third important view is the behavior, which is concerned with the allowed sequences of actions of the architecture.

These three views are fairly broad, and are applicable to all software architectures, but often there are other views which are sensible only within a particular domain. For example, the software architecture of a satellite may include a weight view which is concerned with the amount of cpu and battery weight that is necessary to support a particular software feature.

Before we can address the question of heterogeneous architectural composition, we have to first enumerate which views are being considered in the composition. The number of potential views is large (see table 4 for only a partial

<table>
<thead>
<tr>
<th>View</th>
<th>Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>components and connectors attributes component construction approaches (COTS, manual, project-specific reuse, etc.)</td>
</tr>
<tr>
<td>Topology</td>
<td>components and connectors topology</td>
</tr>
<tr>
<td>Environment</td>
<td>dynamic resource usage (e.g. cpu, memory, bandwidth) static resource usage (e.g. code size) computing platform interoperability interfaces</td>
</tr>
<tr>
<td>Behavior</td>
<td>dynamic state actions, state transitions timing</td>
</tr>
</tbody>
</table>

Table 4: Different views of a system software architecture
Since Z has direct support for state-based attributes and constraints, and since it can also be extended to support behavioral attributes and constraints, it is a sufficiently expressive language for representing several different architectural views. Views like structure, topology, and interoperability can be modeled directly with the state-based constructs in Z, whereas a view like behavior can be represented by traces and constraints on the traces. Z does not appear to exclude the representation of one particular view or another, and this will facilitate extending the our architectural model in the future.

4.1.1 ZSL

In this report, we use a plain text form of Z called ZSL [JIA94]. ZSL has the advantage of coming with an automated type checker (ZTC), and we have used this tool to verify the syntactic correctness of our model’s specification. In the spirit of typical Z specifications, we interleave the formal ZSL text with informal, explanatory English text. Different fonts are used to improve readability:

\texttt{this is formal ZSL text}

\% \texttt{this is an embedded comment}

Please see appendix 13.0 for more information on Z and ZSL.

4.2 Architectures and Architectural Styles Have Many Views

There are many views to any software architecture. A view is a convenient grouping of architectural attributes and constraints that deal with a common issue. For example, perhaps the most popular view of an architecture is the struc-
ture in Rapide, and UniCon does not support behavioral specification. In order to model architectures and architectural styles, we want to avoid choosing a formal notation which commits too soon to a particular class of attributes and constraints at the expense of others.

There are also secondary but nonetheless important pragmatic concerns involved in choosing a particular notation. Readily available literature and experts (e.g. books, papers, newsgroups, etc.) are obvious desirable properties for any formal language. A related benefit is that published models and techniques are likelier to exist if the language is widely accepted, and we can build upon those already existing models and techniques.

With these concerns in mind, we chose to build our model of architectures and architectural styles in the Z notation ([SPIV92, WORD92, POTT91]). Z was originally developed for specifying and designing software. It is based on set theory and first order logic, and it provides a *schema* construct as the main building block for writing formal specifications. Z is ideally suited to model state and incremental changes of state (i.e. operations), but it does not directly support the representation of sequences of operations or of object-oriented hierarchies of schemas. Both of these weaknesses can be worked around, however, as several researchers have shown ([HALL94, STEP92, SUFR90, SPIV90]). Their work is a tiny fraction of the total amount of literature available on Z - it is a very widely accepted formal notation.
4.0 Laying a Foundation for Architectural Models

4.1 Choosing a Formal Notation for Modeling: Z

Our models of architectures and architectural styles must be expressed in a formal notation to facilitate automated analysis of composition. The question automatically arises then “Which formal notation should we choose?” A better question to try to answer first, however, is “What exactly will we be modeling?” While we grapple with this latter question essentially throughout this entire report, we can briefly present an informal answer here. We know that certain styles are characterized largely by the restricted types and patterns of connections (e.g. main/subroutine, layered). Other styles are characterized by the patterns of data exchange behavior between components (e.g. software bus, event-based). By reviewing these styles, and others, we can identify certain classes of attributes and constraints (views - see section 4.2) which seem to be common across styles. They include structure, topology, behavior, and others (see table 4).

There are a large number of existing architecture description languages that are designed for formal specification and analysis of software systems. Many of these languages focus on particular classes of attributes and constraints. Rapide, for example, focuses on system behavior (section 3.1.1), whereas UniCon is well-suited for structural attributes (section 3.2.1). By committing to a particular class of attributes and constraints, however, ADL’s often lose the flexibility to express other classes. Certain interoperability constraints, for example, are difficult to cap-
Part III: Contribution

The following sections describe our answer to the central thesis question. We first describe a general framework for specifying architectures and architectural styles formally. This is followed by a uniform representation of several styles, as well as the introduction of an architectural style space. A model of composition is subsequently built on the general model of styles, as well as a specification of how architecture mismatches are generated. Finally, a tool which sits on these models is presented, with its application to a case study.
tecture mappings, but there is less attention paid to describing formal mechanisms for composition. The semantics of components is flatly ignored, an important omission. Only one example of a style mapping is given, and as will be shown later on, it is necessary to examine several style mappings to see how they are in many cases difficult to create - and in some cases, impossible. Another shortcoming of their approach is that they use three different formalisms. While this may be an unfortunate consequence of the inherent size and ambiguity of software architectural issues, it may be possible to use a single formalism (this is a problem we are struggling with as well). Finally, the authors assume that, during composition, the abstract architecture is unchanged regardless of the concrete architecture. This is not necessarily correct: composition of architectures can sometimes only be achieved by modifying the systems to be composed.
3.3 Theoretic Approaches to Composition

Though there are a large amount of researchers working on software composition at different levels of design, we only present one example that is closely related to our topic.

3.3.1 Moriconi and Qian

A recent paper by Moriconi and Qian discusses correctness and composition of software architectures [MORI94]. Given a pair of abstract and concrete architectures, the authors discuss how to establish the correctness of the concrete architecture with respect to an interpretation mapping. The interpretation mapping is subdivided into two other mappings: a name mapping for translating from names in the abstract architecture to names in the concrete architecture, and a style mapping for translating one style’s theory into the other’s (a theory being a collection of constants, functions, and predicates).

Proving that the style mapping is correct requires defining the semantics of the two involved styles. The authors present one lengthy example, data-flow style to shared-memory style, using temporal logic to capture the semantics of the two styles. This proof does not need to be repeated for subsequent instances of this particular combination. A brief discussion of composition by joining two architectures is also given.

Though this approach is close to ours, its focus is slightly different. The authors are more concerned with showing how to prove the correctness of archi-
large library of code, UNAS allows designers to ‘draw’ an architecture and automatically generate its Ada equivalent. There is no textual ADL, however the tool does a large amount of completeness and consistency checks on the drawn architectures.

UNAS targets the specific architectural problem of communication for large distributed applications across a variety of hardware platforms. A simple set of architectural primitives are provided to the designer, including tasks, processes, sockets, circuits, and messages. A UNAS architecture is essentially a set of tasks interconnected by circuits that carry messages back and forth. The interface to a task is determined by its sockets and the messages sent and received across these sockets.

UNAS provides very strong (but not complete) support for the distributed processes and threads (tasks) software architectural style. Composition of larger systems from subsystems is allowed using simple but powerful mechanisms (placing tasks within processes, and connecting the sockets of tasks to each other). The CASE tool that comes with UNAS does not allow the modeling of control transfers (neither calls nor spawns of tasks or processes). Other styles are not supported directly.
the user with a design environment built around those elements (i.e. specialized to that style). Styles are specified based on the principles developed in earlier work by members of the same group at CMU [ABOW93]. Each style is characterized by its design elements, configuration rules, semantic interpretation, and analyses.

The implementation of Aesop is based on a generic object model comprised of components, connectors, configurations, ports, roles, representations, and bindings. These objects can be subtyped for specific styles. Based on the objects, a graphical interface can be built per style to allow architects to draw architectures for systems in that style. The tool can check for topological errors by using the configuration rules for the style.

Composition in Aesop is limited to structural composition. There is no behavioral view of styles, leaving out a broad class of constraints that may be useful at the architectural level. More importantly, Aesop is currently targeted at homogeneous architectures. As the developers of Aesop point out in their own conclusions for future directions, “heterogeneity of styles is critical.” Finally, there is no formal approach suggested for heterogeneous composition.

### 3.2.3 UNAS

The Universal Network Architecture Services (UNAS) product from TRW is a commercial entry in the field of ADL’s [ROYC91]. UNAS was developed partly in response to the needs of the Ada Process Model [ROYC89] and megaprogramming [BOEH92]. Relying on powerful graphical entry tools and a
their architectures using a mixture of graphical and textual input. If all components in an architecture are decomposed sufficiently into primitives (i.e. executable pieces of code), then the tool can generate an executable system from the architectural specification. One of the original goals of the Shaw team was to allow external tool analysis of architectures, and as a step in this direction, UniCon architectures have been successfully analyzed for scheduling problems by an external tool that supports rate monotonic analysis.

Although at first glance it may seem that the UniCon approach has strong support for architectural styles composition, closer examination yields some significant shortcomings. The most important deficiency is the lack of support for styles as distinct entities. Instead, UniCon provides a prospective architect with a ‘grab-bag’ of components and connectors with a set of complex, implicit rules on how they may be composed. Another important deficiency is the absence of a way to describe global constraints on systems constructed or specified in UniCon. Finally, no formal approach to the process of composition is presented; the emphasis is squarely on providing a tool.

### 3.2.2 Aesop

Continuing with their efforts on architectures and architectural styles, David Garlan’s team at Carnegie Mellon University has developed a tool called Aesop that exploits styles for design environments [GARL94]. Aesop allows the user to specify the elements of a particular architectural style, and then provides
complex behavior. By modelling behavior in Hoare’s CSP [HOAR85], architectures expressed in Wright are rigorously checked for consistency of interconnections and correctness of system behavior - in a manner very similar to Rapide. Wright’s focus on behavior leaves out much structural information that is useful for architectural styles.

3.2 Composition Using Architecting CASE Tools

Researchers and commercial developers have also tackled the problem of architectural composition using CASE tools.

3.2.1 UniCon

UniCon, a language for Universal Connector support, is an ADL developed at Carnegie Mellon University by a team led by Mary Shaw [SHAW94]. Compared to other ADL’s, UniCon has a rich set of built-in components and connectors to give designers an immediate base to work from. Special attention is paid to the ‘packaging’ of a component or connector; UniCon differentiates between a filter that sorts and a procedure that sorts. Components and connectors are described essentially by their interfaces, and systems are therefore hierarchically composed by binding the interfaces of groups of components and connectors. To allow some degree of openness and extensibility, UniCon uses property lists that can be used to specify attributes not directly provided in the language.

UniCon architectures can be specified in a textual form and, to a lesser degree, graphically. A CASE tool has been built which allows designers to enter in
messages. ArTek also supports distinct assignment of functionality (capabilities) to components (elements), providing a link to the requirements phase of the lifecycle.

Though architectural composition can be described in ArTek, it is at a relatively high level. Components may be connected to other components, and architectures can be decomposed into sets of components and connectors. There is no explicit support for styles however.

### 3.1.3 Other ADL’s

There are many other languages that are in use today for describing architectures. Two very domain-specific examples are ControlH and MetaH, both developed at Honeywell [BINN]. Not only do both languages concentrate on a specific problem (ControlH on guidance, control, and navigation applications; MetaH on scheduling of embedded applications), but both also draw on a rich body of established mathematical theory (linear systems theory and rate monotonic analysis respectively). These advantages yield potent architectures that can be used to automatically generate code - but only within the domain. Neither language supports heterogeneous style compositions.

Another language is Wright, developed at Carnegie Mellon University by Robert Allen and David Garlan [ALLE94]. This particular ADL highlights the problems of getting the proper overall system behavior when different components and connectors are joined to one another - each component having its own possibly
also hard to model clearly in Rapide. One possible reason for this is that Rapide has a background in hardware, and as will be shown later (section 4.2.2.1), there are fundamental differences between hardware and software architectures.

3.1.2 ArTek

ArTek was developed by Teknowledge Federal Systems in collaboration with the US Army Armament Research Development and Engineering Center (ARDEC) [TERR93]. The purpose of this particular ADL is not so much formal analysis as it is rapid development and documentation of a system architecture. There are three levels of design to consider when using ArTek: the architecture (generic elements and connectors, among other things), the application software (modules, operations, etc.), and the delivery platform (processors, files, etc.). Several application softwares may map to the same architecture, e.g. an architectural element can be realized by a variety of different modules. ArTek supports hierarchical composition of components and connectors at the architecture and application software levels of design.

At all three levels of design, the basic unit of representation is a set of free-form text fields grouped under a single name (description schema). The free-form nature of ArTek detracts from its strength as a formalism but allows for greater flexibility. ArTek does demand a certain amount of precision with respect to message transferral between components. The order of messages sent and received must correspond to the order expected by the connectors carrying the
occur and are passed around between system entities. Partially ordered sets (posets) of events are used to describe system behavior instead of the more traditional linear traces that are descriptively weaker.

There are actually five independent sublanguages within Rapide shown below (table 3). Tools exist which can take Rapide specifications as input, and which output traces of events to model the behavior of the system. Automated analysis for behavior or timing problems such as deadlock or improper event orders has also been done. TRW has recently developed a tool which allows entry of Rapide specifications graphically [BELZ94].

Like other ADL’s, Rapide has a strong notion of interfaces for composing elements together, however no built-in components and connectors are provided. The strength of Rapide is also its weakness: it is a general-purpose language for systems. It does not explicitly support software architectural styles, and in fact has a bias towards event-based systems. Two systems may be joined easily in Rapide using data connections, but it is not so easy to model control connections (i.e. connections for transfers of control). Other software constructs are

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types Language</td>
<td>To describe the interfaces of components</td>
</tr>
<tr>
<td>Architecture Language</td>
<td>To describe the flow of events between components</td>
</tr>
<tr>
<td>Specification Language</td>
<td>To write abstract specifications of component behavior</td>
</tr>
<tr>
<td>Executable Language</td>
<td>To write executable bodies for components</td>
</tr>
<tr>
<td>Pattern Language</td>
<td>To describe patterns of events</td>
</tr>
</tbody>
</table>

Table 3: The sublanguages of Rapide [LUCK94]
In spite of modelling module composition, most MIL’s were restricted to supporting a single type of interconnection (e.g. procedure call), and also had a bias towards implementation issues [PRIE86]. ADL’s, on the other hand, give connectors the same importance as components, and allow different kinds of connectors to be modelled (e.g. pipe, procedure call, message-passing). A partial list of ADL’s is given in table 2 along with their primary purposes. An overview of some of these languages follows.

<table>
<thead>
<tr>
<th>Name</th>
<th>Primary Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlH</td>
<td>Automated production of guidance, navigation, and control applications</td>
</tr>
<tr>
<td>MetaH</td>
<td>Automated scheduling analysis and production of embedded software applications</td>
</tr>
<tr>
<td>LEAP</td>
<td>Ada code synthesis from architectural specification</td>
</tr>
<tr>
<td>WRIGHT</td>
<td>Analysis of interconnection behaviors</td>
</tr>
<tr>
<td>UNAS</td>
<td>Rapid prototyping and development based on architectural specification</td>
</tr>
<tr>
<td>Rapide</td>
<td>Analysis and simulation of general systems</td>
</tr>
<tr>
<td>UniCon</td>
<td>Analysis and prototyping of software systems</td>
</tr>
<tr>
<td>ArTek</td>
<td>Document architecture of general systems</td>
</tr>
</tbody>
</table>

Table 2: Some Architecture Description Languages

3.1.1 Rapide

Rapide is one of the oldest and most mature ADL’s in use today. Developed at Stanford University by David Luckham et. al. [LUCK94], and having its roots in such diverse languages as VHDL, ML, C++, and TSL, Rapide’s focus is not purely on software. Instead, its purpose is to support simulation of systems in general before they are implemented. Rapide is event-based and object-oriented; that is, Rapide architectures of systems are described in terms of the events which
3.0 Architectural Composition

There have been previous research and commercial efforts that have gone beyond just defining software architectures, and touched on architectural composition explicitly. We present three groups of results in the following sections: work related to architecture description languages (ADL’s), work related to CASE tools for architecting, and work related to theoretic approaches. Some of the efforts described below may actually span more than one group (e.g. UniCon is an ADL but is also supported with a CASE tool).

3.1 Composition Using Architecture Description Languages

One of the most active areas of research in software architectures is concerned with producing notations for describing architectures and, to a lesser extent, styles. These architecture description languages (ADL’s) not only support representation of components and connectors, but they also try to support other possible architecting activities such as prototyping and megaprogramming [BOEH92]. A number of ADL’s were born in very narrow domains, so they also support domain specific analysis. The success of these domain specific ADL’s has illustrated an important tradeoff between the generality of a language and its support of formal analysis: the ability to analyze architectures for specific properties gets easier if the scope of the ADL is limited to a particular domain.

Historically, ADL’s were preceded by Module Interconnection Languages (MIL’s) which tackled the problem of ‘programming-in-the-large’
2.4 Gacek, Abd-Allah, Clark, and Boehm

A more recent entry into the field of defining software architectures has been generated by a group of which this author is a member of [GACE95]. We have expanded the notion of software architectures into “system software architectures” with a set of criteria for identifying them. The criteria are based on making the architectural rationale a first-class citizen, and on requiring the rationale to ensure that the architecture’s components, connectors, and constraints define a system that will satisfy a set of defined stakeholder needs (table 1). These needs are fairly diverse, and make it extremely unlikely that a single notation or method for supporting them exists. No mention of composition of architectural styles is made in our paper.

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Concern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>• Schedule and budget estimation</td>
</tr>
<tr>
<td></td>
<td>• Feasibility and risk assessment</td>
</tr>
<tr>
<td></td>
<td>• Requirements traceability</td>
</tr>
<tr>
<td></td>
<td>• Progress tracking</td>
</tr>
<tr>
<td>User</td>
<td>• Consistency with requirements and usage scenarios</td>
</tr>
<tr>
<td></td>
<td>• Future requirement growth accommodation</td>
</tr>
<tr>
<td></td>
<td>• Performance, reliability, interoperability, etc.</td>
</tr>
<tr>
<td>Architect and System Engineer</td>
<td>• Requirements traceability</td>
</tr>
<tr>
<td></td>
<td>• Support of tradeoff analyses</td>
</tr>
<tr>
<td></td>
<td>• Completeness, consistency of architecture</td>
</tr>
<tr>
<td>Developer</td>
<td>• Sufficient detail for design</td>
</tr>
<tr>
<td></td>
<td>• Reference for selecting / assembling components</td>
</tr>
<tr>
<td></td>
<td>• Maintain interoperability with existing systems</td>
</tr>
<tr>
<td>Maintainer</td>
<td>• Guidance on software modification</td>
</tr>
<tr>
<td></td>
<td>• Guidance on architecture evolution</td>
</tr>
<tr>
<td></td>
<td>• Maintain interoperability with existing systems</td>
</tr>
</tbody>
</table>

Table 1: Stakeholder Concerns [GACE95]
technologies which will be available for implementing applications consistent with the architecture. [HAYE94]

A *domain-specific* software architecture, on the other hand, consists of a software architecture as well as a domain model and a set of standardized requirements. The DSSA definitions seem to combine the best of Garlan/Shaw and Perry/Wolf plus a little extra in the case of a domain-specific architecture. Garlan and Shaw’s definition of architectural style has been more or less adopted [TRAC94].

In spite of their common definition, the different DSSA projects have yielded architectures whose representations focus on different aspects of systems. For example, the TRW/Stanford project has produced architectures expressed in Rapide, an “event-based concurrent, object-oriented language specifically designed for prototyping system architectures.” [LUCK94] The Honeywell/Maryland project, on the other hand, has focused on and produced architectures for intelligent guidance, navigation, and control (GNC) applications. Consequently, their architectures are expressed in ControlH and MetaH, two languages that focus on GNC and scheduling issues respectively [BINN]. A third project from Teknowledge/Stanford has generated architectures for autonomous vehicles expressed in ArTek, yet another distinct language [TERR93].

Within all the DSSA projects mentioned here, there has been some attention paid to composition of architectures. None of these projects, however, refer to architectural styles during composition. We will examine one of the projects (Rapide) in greater detail later (section 3.1.1).
a single component may use a mixture of connectors of other styles

The authors claim that most systems are composed from different styles.

A key element of software architectures is to precisely identify the constraints of the architecture. Garlan and Shaw only mention that there are constraints, but aside from a couple of informal examples, no formal or disciplined approach is given towards identifying a general class or classes of constraints. Another shortcoming of this paper (left as future work by the authors) is the lack of a formal classification of styles. Finally, though combining different styles is discussed, there is no claim made as to the completeness of the composition methods that are identified. There is no detailed attention given to a general approach or mechanism for architectural composition.

2.3 The DSSA Program

Since its inception in 1991, the ARPA Domain Specific Software Architecture (DSSA) program has contributed valuable insights into the field of software architecture. While a general definition of what is an architecture has been laid down, the exact form varies from project to project within the DSSA program. In general, a software architecture is defined as

...An abstract system specification consisting primarily of functional components described in terms of their behaviors and interfaces and component-component interconnections. The interconnections provide means by which components interact. Architectures are usually associated with a rationale that documents and justifies constraints on component and interconnections or explains assumptions about the
2.2 Garlan and Shaw

A very popular definition of software architecture has been advanced by Garlan and Shaw [GARL93] which is more restrictive than the definition of Perry and Wolf. Garlan and Shaw proposed that a software architecture for a specific system be represented as “a collection of computational components - or simply components - together with a description of the interactions between these components - the connectors.” Based on this definition, the authors used the term architectural style to denote a family of systems (architectures) that share a common vocabulary of components and connectors, and which meet a set of constraints for that style. The constraints can be on a variety of things, including on the topology of connectors and components, or on the execution semantics.

To instantiate the concepts of architecture and style, Garlan and Shaw presented a valuable partial taxonomy of known architectural styles (see section 1.0 for a list). For each style, they asked questions designed to bring out its unique characteristics such as “What is the structural pattern of components and connectors?” and “What are some common examples of its use?” Quality attributes are missing from Garlan and Shaw’s view of software architecture - in fact, the rationale found in Perry and Wolf’s definition is entirely missing.

Architectures that are composed of different styles are discussed by Garlan and Shaw. The authors identify two examples of composition:

- hierarchical elaboration of a component or system in one style into another system of a different style
three classes: processing elements, data elements, and connecting elements. Whereas the processing and data elements have been studied extensively in the past (e.g. functions and objects), it is the connecting elements that especially distinguish one architecture (or style) from another. The form of the architecture is given by enumerating the properties of the different elements and the relationships between the elements. Another essential part of the architecture is its rationale which includes quality attribute aspects among other things.

Perry and Wolf also define an architectural style as “that which abstracts elements and formal aspects from various specific architectures.” In short, a style is a less constrained, less complete architecture - there is no hard dividing line. The utility of a particular style comes from its addressing of important design decisions up front, isolating and highlighting certain aspects. A style or a specific architecture can be viewed in three different ways based on the elements: a processing view, a data view, and a connections view. All three views are necessary for the understanding of the style or architecture.

Perry and Wolf mention only a single style in their paper (“multi-phase compiler”). Consequently, they do not address the issue of composing different styles together within a single architecture. The representation of the style they do describe is only semiformal.
2.0 Defining Software Architectures and Styles

The words “software architecture” are two of the most overused and overloaded words in software engineering. At least two reasons exist to explain this. The first is that “architecture” by itself brings with it a number of implied meanings and assumptions from its association with the construction of physical buildings, as in the architecture of the Sears Tower. The second reason for confusion is that software architectures are relatively new as a recognized separate product of the software lifecycle. While it has been known at least since the mid-70’s that structuring a collection of modules together into a system is fundamentally different from traditional statement-level programming [DERE76], it was not until the late 80’s that the term “software architecture” emerged [SHAW89]. Since then, the use of the term has skyrocketed, often with very different meanings. For example, a software architecture has been variously defined as an executable prototype [ROYC91], a static description of the source topology [GARL93], and a document [KRUC94].

2.1 Perry and Wolf

One of the earliest formal definitions of software architectures, by Perry and Wolf [PERR92], has remained one of the most insightful. After examining the architectures of other disciplines (hardware, networks, and buildings), Perry and Wolf describe a software architecture as “a set of architectural (or, if you will, design) elements that have a particular form.” The elements are divided into
Part II: Related Work

Other researchers have been working on software architectures and their composition. The following sections describe some of their results. First, we present several similar definitions of architectures and styles developed over the last few years. Next, we show different approaches to software architectural composition, and discuss whether the approach utilizes architectural styles (and to what extent). Finally, we summarize the deficiencies of these past efforts with respect to addressing the central question of this thesis.
Our choice of a representation language for our models was driven by which views of architectures we wanted to capture (section 4.0). Though there are many potential views to an architecture, we have only considered a subset of those views in our models. The formal notation we chose has the important property of not being exceptionally biased against one view or another (i.e. it does not appear to exclude any particular view). It also has the property of strongly supporting certain views.

Finally, the subproblem of identifying the precise meaning of composition has been resolved in our approach. The first step is to recognize that the composition of systems is achieved by introducing a set of interactions between them. Understanding what those potential interactions are is half the battle, and the other half is deciding which ones to consider in our specification of a composition model. Choosing all interactions is not an option: there are simply too many.

1.3 Key Contributions

Please see section 10.0 for a high-level discussion of our key contributions.
1.2 Approach

Our approach to the central problem described above can be summarized in two steps:

- create a formal model of a few pure architectural styles to gain greater insight, and to generate some ‘working material’
- create a formal model of composition based on the styles generated in the prior step

Our models of styles and composition are currently not intended to generate ‘architectural theorems’, but rather they will be used to show the conditions under which the composition of a heterogeneous architecture may fail. The models are also the basis for a CASE tool which the architect can use to specify and analyze heterogeneous architectures.

Our approach addresses all of the subproblems mentioned previously. One of the thorniest issues to deal with is the different focus of different styles (see section 5.0 and section 6.0 for more details). There are two possible ways to address this: either look for underlying commonalities between the styles, or reduce the number of styles and how to compose them with a suitable rationale. Initially we took the latter, easier way but as we gained understanding into the essence of architectural styles, we were able to ultimately take the former, harder way. By identifying certain key properties that define a space of styles, we reduced the complexity associated with modeling them and their composition.
1.1 Problem Statement

The aim of this thesis is to formally address software composition at the architectural style level. A concise statement of the problem I propose to examine reads as follows,

What is a formal model for the composition of different architectural styles within software systems?

In spite of scoping the central problem of composition by relying on styles, there are several formidable subproblems,

- architectural styles often focus on different properties
- finding appropriate representation languages for architectural attributes and constraints
- composition can be achieved in many different ways

Each of these subproblems needs to be addressed by any successful approach.

There are several benefits to be gained by studying this problem of architectural composition using styles. An architectural style is a partial starting point for organizing the design of a system. It provides a baseline set of entities and constraints that the architect can leverage to solve the problem at hand. By studying the problem of composition, we are forced to examine individual styles in a precise manner, and this helps to clear some of the prevailing confusion on the definition of styles and architectures. If our approach is ultimately successful, another benefit will be a tool which provides analysis of high-level compositions of styles within heterogeneous systems.
An example of a heterogeneous system which uses multiple styles as a result of the choice of subparts or subsystems is the Aesop CASE (Computer-Aided Software Engineering) tool ([GARL94]). More details about this system can be found in section 9.2, but we present its overall design in figure 2. Four different commercial-off-the-shelf packages are composed with each other, and two distinct styles are being used.

These high level descriptions sound very attractive on paper, yet there are a great deal of important details that are hidden behind the ‘+’ signs and the lines in the figures. The hidden details are implicit attributes and constraints within each style. Typically, architects may rely on an ad hoc method to compose different styles together, trusting their personal experience to adequately address the implicit constraints. In order to provide architects with a systematic approach for composing these styles together, it is necessary to construct a model and/or a tool to aid the architect during composition.
Consider, for example, the problem of designing a satellite ground station for a representative satellite. The ground station is responsible for receiving satellite telemetry, processing the telemetry, interacting with the users, and sending satellite commands (figure 1). One possible design for the system might partition the ground station into three distinct parts. For receiving telemetry, the architect may choose a streaming pipe and filter style to handle the more or less constant flow of repetitive data. The part of the station responsible for sending commands up to the satellite might be designed as another pipe and filter subsystem. The remainder of the system which is responsible for interacting with the users and processing the telemetry might be based on the event-based style to achieve high flexibility. This system is an example of the problem domain driving the choice of multiple styles.
• object-oriented
• main program/subroutines
• repositories
• event-based (implicit invocation)
• rule-based
• state transition based
• process control (feedback)
• domain-specific
• heterogeneous

Some styles focus on certain attributes or constraints that are not addressed by other styles. For example, the layered style is concerned with constraining the topology of connections, whereas the repository style merely says that at the heart of the system is a central repository. One interesting question to consider is if there is a reasonable, finite set of attributes or constraints that we can describe all or most styles with.

Architectural styles are a powerful means for grasping the design of large software systems. Though software designers may not precisely articulate the architecture of the systems they design, almost all successful systems will rely on one or more architectural styles. It is likely that more than one style is used, especially if the system is large. The need for multiple styles can come from either the problem domain or the subparts used to construct the system.
1.0 Introduction

A persistent problem in computer science is how to put software systems together out of smaller subsystems, the problem of software composition. There are many levels of granularity at which this problem can be tackled. For example, some of the earliest software engineers dealt with systems at the machine language or assembly language level of granularity. Succeeding engineers addressed the composition problem at a coarser granularity using higher level programming languages. The emergence of software architectures and architectural styles has introduced a still coarser level of granularity (and higher level of abstraction) at which we can create and compose software systems.

One of the earliest discussions of software composition at the architectural level was first mentioned by Garlan and Shaw in their seminal paper describing some architectural styles ([GARL93]). In that paper, the authors state that “most systems typically involve some combination of several styles.” These types of systems are termed to have heterogeneous architectures. However, the focus of their paper is more on introducing pure architectural styles rather than on addressing the problem of composing heterogeneous architectures. There are twelve styles explicitly mentioned by the authors:

- layered
- distributed processes and threads
- pipes and filters
Part I: Introduction

In the following sections, we present a high-level introduction to the field of software architectures and style composition, and a concise description of the problem which we examined for the dissertation. We also summarize the steps in our approach towards solving the problem.
Abstract

A persistent problem in software engineering is how to put complex software systems together out of smaller subsystems, the problem of software composition. The emergence of software architectures and architectural styles has introduced a higher level of abstraction at which we can create and compose software systems. We examine the problem of providing formal semantics to the composition of different architectural styles within software systems, i.e. the problem of composing heterogeneous architectures. We describe a model of pure styles, and a model of their composition.

Our model of pure styles is highlighted by a uniform representation for describing many different styles. An architectural style space of major conceptual features is introduced which allows new styles to be rapidly incorporated into the model, including commercial-off-the-shelf packages which embody a specific style(s). We show a disciplined approach to the process of architectural composition, and show how architecture mismatches can be generated during composition. Finally, we describe a prototype tool which is built on top of the models.
List of Tables

Table 1: Stakeholder Concerns [GACE95] ............................................................15
Table 2: Some Architecture Description Languages .............................................17
Table 3: The sublanguages of Rapide [LUCK94] ................................................18
Table 4: Different views of a system software architecture ..............................31
Table 5: Generalized components and connectors ...........................................53
Table 6: Four Instances in an Architectural Style Space .................................84
Table 7: Partial list of interactions between system components .....................87
Table 8: Mismatches related to the conceptual Features .................................94
Table 9: Four Instances in an Architectural Style Space .................................114
List of Figures

Figure 1: A Satellite Ground Station Using Multiple Architectural Styles......4
Figure 2: A CASE tool Using Multiple Architectural Styles: Aesop..............5
Figure 3: Views within a Software Architecture........................................33
Figure 4: Hardware levels of description ([MILN94])................................36
Figure 5: Full-Adder Structure ([MILN94])...............................................36
Figure 6: Architectural topology for a simple system...............................37
Figure 7: Two dynamic topologies satisfying the same architecture ..........37
Figure 8: An Architectural Specification Consolidating Multiple Views ......40
Figure 9: Combinations of Style-Style Compositions...............................43
Figure 10: Possible Architectural Connectors/Interactions.........................55
Figure 11: Extending the Base entities to Style-specific entities ..............70
Figure 12: Composition by using the Group Operation...............................88
Figure 13: Conceptual features mismatches affect more styles ...............92
Figure 14: Mismatch #2 of Group Operation............................................95
Figure 15: The Architect’s Automated Assistant .......................................102
Figure 16: Sample AAA specification fragments.................................104
Figure 17: Main Screen for AAA.................................................................107
Figure 18: Results of loading a specification into AAA............................108
Figure 19: Listing of entities stored in AAA after loading a project...........108
Figure 20: The attributes and values of an entity stored in AAA...............109
Figure 21: Architecture mismatches detected in an entity by AAA..........110
Figure 22: Aesop .......................................................................................116
Figure 23: Aesop Architecture Mismatches detected by AAA.................121
### Table of Contents

6.2.3 Useful resource attributes and constraints ............................................ 50
6.2.4 Base (‘style-less’) Entities .................................................................... 52  
  6.2.4.1 Overview of Base Entities ............................................................ 52  
  6.2.4.2 Specification of Base Entities ....................................................... 56  
6.2.5 Style-Specific Extensions to Base Entities ........................................... 70  
  6.2.5.1 Weakly Constrained Styles......................................................... 70  
  6.2.5.2 Pipes and Filters ....................................................................... 72  
  6.2.5.3 Main/Subroutine ..................................................................... 75  
  6.2.5.4 Distributed Processes ............................................................... 76  
  6.2.5.5 Event-Based (Implicit Invocation) ........................................... 78  
6.2.6 COTS-specific extensions to base entities ............................................ 81

6.3 A Conceptual Feature Space underlying Architectural Styles ................. 82

7.0 Composition of Architectural Styles ..........................................................86
  7.1 Interpreting “Composition” .................................................................... 86
  7.2 Other Composition Operations .............................................................. 89

7.3 Architecture Mismatches under Composition .......................................... 90
  7.3.1 Utility of the Conceptual Features for finding Mismatches .............. 91

7.4 The Group Operation .............................................................................. 92
  7.4.1 Overview of Group Operation ........................................................... 92
  7.4.2 Specification of Group Operation ...................................................... 96

8.0 The Architect’s Automated Assistant (AAA) ........................................102
  8.1 Architecture Specifications for AAA ..................................................... 103
  8.2 Technical Details of AAA ................................................................. 104
  8.3 Sample Operation of AAA ................................................................. 107

9.0 Applications of Approach ........................................................................111
  9.1 UNAS as an architectural style ............................................................. 111
  9.2 Identifying Architecture Mismatches: Aesop Case Study .................... 116
    9.2.1 Simple Specification of Aesop ....................................................... 118
    9.2.2 Analysis of Aesop specification with AAA ................................... 121

10.0 Summary of Key Contributions ..............................................................125

11.0 Future Extensions ............................................................................... 127
  11.1 Extensions to the underlying model of styles ..................................... 127
  11.2 Extensions to the model of architectural composition ...................... 128
  11.3 Extensions to the tool and general approach .................................... 130

**Part IV: References and Appendices** ..........................................................131

12.0 References ..........................................................................................132

13.0 The Z Notation and ZSL ......................................................................136

14.0 A List of Possible Software Architecture Styles ....................................140
Composing Heterogeneous Software Architectures

by

Ahmed Abd-el-Shafy Abd-Allah

--------

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Science)

August 1996

Copyright 1996 Ahmed Abd-Allah