Chapter

Event-Based Execution Architectures for Dynamic Software Systems

James Vera, Louis Perrochon, David C. Luckham
Computer Systems Laboratory
Stanford University
Stanford,, CA 94305, USA
{vera,perrochon,dcl}@pavg.stanford.edu

Key words: Evolutionary software architectures, software artifacts, component engineering.

Abstract: Distributed systems' runtime behavior can be difficult to understand. Concurrent, distributed activity make notions of global state difficult to grasp. We focus on the runtime structure of a system, its *execution architecture*, and propose representing its evolution as a partially ordered set of predefined *architectural* event types. This representation allows a system's topology to be visualized, analyzed and constrained. The use of a predefined event types allows the execution architectures of different systems to be readily compared.

1. INTRODUCTION

Distributed software systems consist of computational components interacting over a communications infrastructure. The executions of these systems can be highly dynamic with components being created and destroyed and the communications infrastructure undergoing continual reconfiguration. We propose to represent the evolution of the structure of such a running system, termed the *execution architecture* of the system, as a set of events, partially ordered by time and causality. This partial order of *architectural* events enables the precise analysis of the topological evolution of a system, just as a partial order of behavioral events enables a precise analysis of the functional activity of a system [1].

The need for understanding execution architectures is driven by the main trends of software. *Component-oriented software engineering* has resulted in systems composed of components connected through middleware. *Distribution*, especially large scale, leads to asynchronous systems. The effect on execution architecture is dramatic: there may be no single depiction of the execution architecture of an asynchronous distributed system at a particular "point" in time. Instead, different observers can have a different views of what the architecture is.

We define a model for execution architectures and event types used to indicate changes in such a model. We show how systems such as distributed Java programs or systems communicating over commercial middleware can have their topological evolution projected onto our model. Using a predefined set of event types allows us to compare the execution architectures of systems implemented in different languages and which utilized different communications middleware.

Finally, we show how our representation of an execution architecture allows a system's topological evolution to be visualized, analyzed, and constrained.

2. PREVIOUS WORK

Our work is builds on two previously separate lines of research: software architecture and causal modeling.

2.1 Software Architecture

The term *architecture* has been widely discussed in the literature (e.g., [2] [3] [4] [5]). Soni et al. [6] discuss four categories of architecture: Conceptual, Module, Execution and Code. Conceptual architecture describes a system in terms of high level, abstract elements. Module architecture is the a more detailed functional decomposition. Execution architecture is the structure of the running system. Code architecture is the organizational structure of the source code of the system. Execution architecture is unique among the four in being a dynamic structure. We focus on execution architecture and argue that its appropriate representation is a partially ordered set of events.

Current research in software architectures has often focused on conceptual or module architectures (we will term architectures in either of these categories as *component* architectures). Architectures are described as entities possibly within other entities and interconnected somehow. Such descriptions are sometimes referred to as "boxes and arrows" representations. While being useful for many purposes, they have their shortcomings in describing a dynamic system. The representation of an execution architecture needs to be able to deal with change. In simple cases, execution architectures may be thought of as a series of static architectures, snapshots at different points in time. However, in many cases this is not enough.

The ACME system developed by Garlan et al. is designed as an language for exchanging architectural designs [7]. The ACME system is inherently static though there is a proposed extension to allow the specification of potential dynamism. Darwin [8] focuses on design specification and is not intended to be used in systems where new component types and the pathways between them are defined and added at runtime.

2.2 Causal Modeling

The use of partial orders of events to depict the behavior of distributed systems is well established [9, 10]. The relation of the partial order, typically called *causality*, enables true concurrency to be represented, information which is lost in a trace-based model.

Fidge and Mattern [11, 12] separately developed the notion of vector time which is an algorithmic way of representing and analyzing the causal relation. Subsequent work has been done in improving the performance of such algorithms in special cases (e.g., [13]. See [14] for an excellent survey). Other work has been done on applying causal modeling notions to existing programming languages [15].

Our framework for execution architectures is an extension of our previous work in event-based systems [16, 17]. There we created a programming language, `RAPIDE`, in which a causal record of a program's behavior was automatically deduced and recorded during the program's execution.

3. A THEORY OF EXECUTION ARCHITECTURE

3.1 Execution Architectures

Execution architecture is a runtime notion. It is the architecture of an executing system. Its building blocks are executable constructs (e.g., objects, processes, tasks) which we call *modules* and the mechanisms they use to communicate which we call *pathways*. Both of these building blocks may be created and deleted during the system's execution making execution architecture an inherently dynamic notion. It can best be thought of as the record of the evolution of the structure of a running system.

3.2 Modules and Pathways

Our framework for execution architectures is built on two basic constructs:

– *Modules* which are groupings of computational capabilities, and
– *Pathways* which are the means modules use to communicate amongst themselves.

Module: A module is a grouping of computational capabilities. Modules have an associated type. The type consists of a set of provided and required features of each module, called *declarations*. These declarations are used to communicate with other modules. In an event-based system, these declarations would denote what events a module can send and receive. In a system based on synchronous (remote) procedure calls, the declarations would describe the procedures provided and called by each module. The type of a module describes what the module requires from other modules as well as what the module provides to other modules. Some architecture description language type systems only describe what modules provide.

In addition, we define a parent-child containment relationship over modules. Each module has maximum one parent. The parent relationship forms a directed graph. Being dynamic, the parent of a module may change. While parent-child is the only module relation we predefine, additional relationship may be defined, such as a relation between the software modules and the hardware modules they currently run on, etc.

Pathway: A pathway represents potential communication among modules. A pathway has a name, a set of inputs and a set of outputs. The inputs may be thought of as those things which invoke or use the pathway and the outputs as those things which result from the invocation or observe the use of the pathway. The inputs and outputs of a pathway may change. Typically, one input or output identifies a pair (module, declaration). More generally, we allow the use of patterns to concisely specify sets of inputs or outputs. For example, a pattern could express "any module of type Airplane performing a RadioOut event." A pathway also has a scope over which it operates. The scope may be a particular module or the entire system.

A pathway can represent a mechanism or simply a state or condition. Possible examples of pathways are a UNIX pipe, a Java socket, a serial cable between two computers, or a dynamic scoping rule of a particular programming language.

What constitutes a module is a subjective determination. For example, in a producer-consumer example, the producer and consumer are likely to be modules while the data communicated between them is probably not. Thus what is defined as objects in the source language does not necessarily correspond to modules. Not all objects need be modules, not all modules need be objects. In a system of workstations and network links one modeler may choose to have the workstations be represented as modules and the network links to be pathways. However, for a modeler more concerned with the network protocols, the network links might be the modules and the workstations the pathways. The key point is that modules represent the building blocks of the architecture. The definition of the actual correspondence is determined by the system implementor though language/system defaults may be used.

3.3 Execution Architecture Events

An execution architecture changes over time. Modules are created and destroyed, pathways come into and go out of existence. Such occurrences may be serialized or may happen independently. We model such changes as **events**. For example, the creation of a module or the additional of an output to a pathway would each be denoted by events. In our framework, we have templates for nine architectural events to describe creation and deletion of modules and pathways, addition and deletion of inputs and outputs from pathways, and changing of the parent of a module.

Events have parameters containing additional information. A `CreateModule` e.g., has parameters denoting the type of the module that was created, the parent of that module and the name of the module. We give the simplified description of the templates below:

```
CreateModule(type : ModuleType, parent : Event,
             name : String);
DeleteModule(module : Event);
CreatePathway(inputs : Pattern, outputs : Pattern,
              name : String);
DeletePathway(pathway : Event);
ChangeParent(module : Event, parent : Event);
AddPathwayInputs(pathway : Event, inputs : Pattern);
AddPathwayOutputs(pathway : Event, outputs : Pattern);
DeletePathwayInputs(pathway : Event, inputs : Pattern);
DeletePathwayOutputs(pathway : Event, outputs : Pattern);
```

Some explanations may be necessary: first, our events do not directly refer to modules or pathways, as modules and pathways are transient objects in an execution architecture. In many cases, such as debugging post mortem, these objects no longer exist. Instead, we refer to the event that denotes the creation of the module or pathway. This can be seen in the parent parameter of `CreateModule` Instead of referring to the parent module, we refer to the `CreateModule`-event of the parent.

Second, we would like to be able to define the inputs and outputs of a pathway in a descriptive way, rather than as an enumeration of all possible inputs. `RAPIDE` allows us to easily describe the sets of input and output of a pathway using a *pattern*. For our purpose, the pattern in `CreatePathway` just specifies a set of declarations of certain modules. If a pattern language is not available, sets of pairs of a module (denoted by an event) and a declaration would work also.

Events are ordered temporally and causally. In the context of an event processing system such as `RAPIDE`, our architectural events can be treated like normal events. This allows us to use existing browsing tools and, more interestingly, pattern matching and constraint tools on architectural events. Using event constraint tools, we can write *topological* architecture constraints. Examples of such are presented in section 4.1.

3.4 Causal and Time Orders

When events are created they are (partially) ordered by cause and time. Two events are temporarily ordered if their temporal relation can be determined by any single clock in the system. The temporal order of two events in a distributed system without a common clock is not a priory known, but may be derived later.

Two events are causally ordered if one causes the other (transitively). The exact meaning of *cause* is configurable and is captured by the system architect in a causal model. A common definition is that the events produced by a thread are totally ordered, the receipt of an event causally follows its sending.

The partial ordered set (poset) of architectural events forms a record of the evolution of the architecture. Recording relations between events in distributed systems as partial orders (instead of just time-stamping them) reveals that "the execution architecture at a certain point in time" is not a well defined concept. [18] introduces the notion of **consistent cuts** as architectural observation points. A consistent cut partitions a partially ordered set into a *before* and *after* part. If an event is in the after part, then all events that follow it temporarily or causally are in the after part, and vice versa. Informally, an observer could have seen only and exactly the *before* part of the poset. When we speak of a "point" in the execution we mean "at a consistent cut".

3.5 Static Snapshots

At any consistent cut in the poset, a static representation or *snapshot* of the execution architecture similar to a component architecture may be derived from all of the events preceding the consistent cut. Such a snapshot is amenable to the types of analysis typically done on component architectures.

A *compatible sequence* of consistent cuts is graphically defined as a sequence of cuts which do not cross. Such a sequence may be viewed as an animated movie of the architecture's evolution. Since a poset may contain a set of such sequences, an execution architecture may contain a set of such animations. Each animation corresponds to a particular observers view of the architecture over time. The example below gives examples for such compatible and incompatible sequences of consistent cuts.

4. APPLICATIONS OF EXECUTION ARCHITECTURES

4.1 An Air Traffic Control System

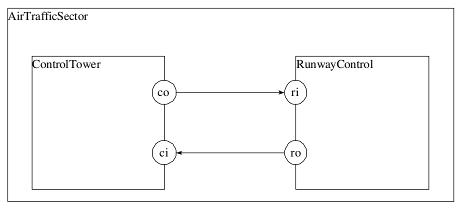

Figure 1. Initial air traffic architecture

Consider an air traffic control system as depicted in figure 1. Its architecture consists of AirTrafficSector which contains a ControlTower and a Runway-Control module. This initial architecture was created by the execution represented by the poset in figure 2. The arrows denote the causal relation. Note that the consistent cut C1 in figure 2 marks the "point" in the execution at which the architecture depicted in figure 1 holds.

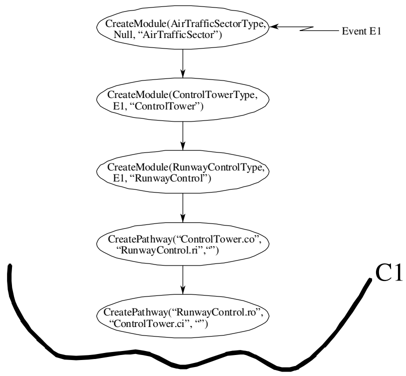

Figure 2. An initial execution of the air traffic system

Next imagine that two Flights (one called UA17, the other AA23) are created and that their creations are independent. A pathway from each Flight to the ControlTower is also created. This execution is represented by the poset in figure 3. At the point in that poset indicated by consistent cut C3 the architecture depicted in figure 4 holds.

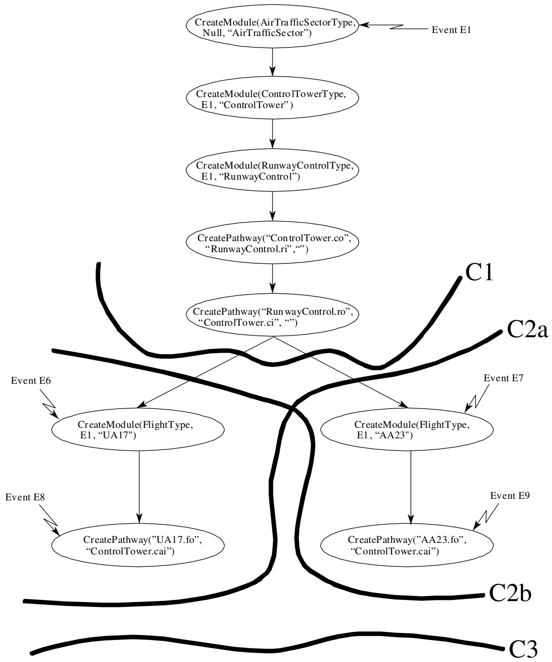

Figure 3. Continuation of execution of the air traffic system

In between consistent cut C1 and consistent cut C3 there are seven consistent cuts[1] two of which are shown in figure 3. Cuts C2a and C2b are inconsistent (graphically the cuts cross) so they would not both appear in the same architecture animation. One architecture animation A1 could consist of sequence of consistent cuts C1, C2a, C3 and another architecture animation A2 could consist of the sequence C1, C2b, C3.

[1]The consistent cut for which the maxima is (1) Event E6, (2) Event E7, (3) Event E8 (this cut is labeled C2a in figure 3), (4) Event E9 (this cut is labeled C2b in figure 3), (5) Events E6 and E7, (6) Events E6 and E9 and (7) Events E7 and E8

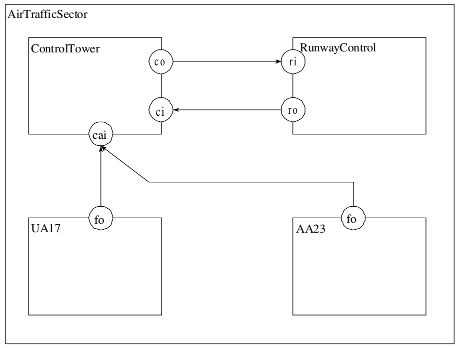

Figure 4. Air traffic architecture at consistent cut C3

In architecture animation A1, the initial snapshot shown in figure 1 would appear, then Flight UA17 and its connection to the ControlTower would appear and finally Flight AA23 and its connection to the ControlTower would appear. In architecture animation A2, the same initial architecture as in A1 would appear, followed by the appearance of Flight AA23 and its connection to the ControlTower followed by Flight UA17 and its connection to the ControlTower.

In a system which is merely time-stamping its architectural changes, or which observes them by break-pointing the system, only animation A1 or animation A2 would be seen (or potentially a third animation A3 in which at one "frame" neither flight is visible and in the next both are. This animation would result from an overly course time-stamping or breakpointing interval.) This is a specific instance of a more general case. Whenever there are concurrent changes to an architecture, a single trace of those changes (such as would result from time-stamping or breakpointing) will only capture one animation. They cannot capture the information contained in incompatible consistent cuts.

4.1.1 Use of Partially Ordered Architectural Events

The representation of execution architecture as partially ordered sets (posets) of events allows poset oriented tools and methods to be applied to execution architectures. In particular, the pattern and constraint languages developed in the `RAPIDE` project may be applied to specify topological (as opposed to purely functional) constraints on executing systems. The `RAPIDE` languages can be used to set up simple filters, constraints or maps. Some illustrative examples follow.

4.1.2 Filters

Filters are operators which takes as input a poset and outputs a subset of its input selected by a pattern. Filters allow a reduction of the space being examined. Suppose we are only interested in the module containment structure. The following filter could be used:

```
observe select CreateModule() or DeleteModule()
            or ChangeParent();
```

4.1.3 Constraints

The representation of an execution architecture as a poset allows us to write constraints about its evolution as well as static snapshots. For example, we could constrain that a Radar module must be created before a Depot module. Or that a particular communication topology (full connected, strongly connected) exists among a class of modules before some condition.

Given a poset constraint language such as that available in RAPIDE, the existence of architecture events allows these specifications of topological constraints. In an event generating system (where the behavior is also represented as events), mixed-mode functional/topological constraints can be expressed.

Suppose we want to require that the creation of Flight modules must be serialized. We might make this requirement because the creation of a new Flight module involves the manipulation of some global state (e.g., the number of Flights currently in the sector). We can express this constraint as requiring the events signifying the creation of Flight modules be totally ordered:

```
observe select CreateModule(type is FlightType)
 match [* rel -> ] CreateModule;
```

4.1.4 Maps

Maps are operators which transform a poset into a new poset. The new poset is generally at a higher level of abstraction. That allows the behavior of system to be understood in more abstract terms than those in which it was implemented.

As a simple example, suppose we wish to abstract ControlTower module and RunwayControl module pairs into single AirportControl module. To do this we would create a map that does this abstraction and adjusts the communication structure accordingly. If the input poset also contained the functional behavior of the system then behavior of a ControlTower or RunwayControl module would also need to be mapped into behavior by an AirportControl module.

A subset of such a map is given below:

```
map AirportAbstract is trans : array [Event] of Event;
    (?c,?r,?p, ?a : Event; ?s1,?s2 : String)
    ?c@CreateModule(ControlTowerType, ?p, ?s1) and
    ?r@CreateModule(RunwayControlType, ?p, ?s2)
=>  ?a@CreateModule(AirportControlType, ?p, ?s1+?s2);
    trans[?c] := ?a; trans[?r] := ?a; end map;
```

The above rule looks for pairs of CreateModule events, one denoting the creation of a ControlTower module, the other a RunwayControl module. If they both have the same parent then a CreateModule event is created in the new poset which denotes the creation of an AirportControl module. The association of the lower level events to the higher level event is stored in an associative array for subsequent use by other rules.

4.1.5 Conformance to Reference Architectures

By combining maps and constraints, the conformance of systems to reference architectures may be checked [16]. Architecture events allow topological conformance to be expressed. This can be useful for checking requirements such as duplicate communication channels.

4.1.6 Reverse Engineering

Reverse engineering of architectures is necessary when the original architecture has been lost (or never existed). Research has focused on extracting component architectures from source code [19]. By extracting architecture events from a running system via instrumentation (such as monitoring middleware) we can extract the execution architecture even when the original source code is unavailable. Perhaps more comparative work is the extraction of call trees by debugging software. These tools can be thought of as providing a maximal depiction of the use of the execution architecture. An execution architecture poset, in contrast, captures its evolution.

4.2 Applications to Other Domains

The mapping of concepts from event-based systems into our architectural constructs is flexible and in each case, different strategies are supported with emphasis on different attributes.

Whatever choice is made, the ability to map one poset into another allows such decisions to be changed ex post facto. In the above example, the choice of the assignment to modules and pathways could be inverted by a mapping.

In this subsection we present some example translations of distributed systems to our execution architecture constructs.

4.2.1 A System Implemented in Java

The Java notion of objects is easily mappable to our module concept. More interesting is the choice of constructs which map to pathways.

The ability of one object to name an other object (generally known as dynamic scoping) is one form of pathway. If an object A can name an object B then we can say that a pathway exists from A to B.

The Java socket construct is amenable to translation into a pathway. A Java socket is a bidirectional mechanism over which data may be sent from one object to another. It has two ends. Any object which can name an end may send or receive data along the socket. Therefore, a Java socket could be translated into two of our pathway constructs (pathways are one directional while sockets are bidirectional) where the sources of one of the pathways are the destinations of the other (and vice versa).

4.2.2 A System Hosted on Commercial Middleware

The Information Bus (TIB) [20] is a communication middleware which supports the subject-based publish-subscribe metaphor. Objects send out (publish) messages labeled with a particular textual field (subject). Other objects can request to receive (subscribe to) messages with a particular subject. Higher level protocols are built on top of the publish/subscribe mechanism such as point to point communication, synchronous communication, and automatic selection of one from several destinations.

In our application of execution architectures to the TIB [21], we map each TIB client into a module and map the basic publish/subscribe mechanism into pathways. In a component architecture description, for every subject a connection is needed from the modules which may publish that subject to the modules that may subscribe to the subject. Not surprisingly, pictures of such architectures show the TIB only as a bus. In an execution architecture, pathways are only maintained between modules that actually publish and modules that actually listen to a certain subject, e.g., only after a module subscribes to a subject it is added as a destination of the pathway which corresponds to that subject. This results in a point-to-point depiction of the communication network.

Other TIB protocols can be captured via their implementation on top of the publish/subscribe protocol. However, the semantics of the higher level protocols are more accurately captured by dealing with them individually.

5. SUMMARY AND CONCLUSIONS

We developed a technology to define, track and control execution architectures of dynamically changing software systems. Architectural changes are represented by causally and temporarily (partially) ordered events. Our framework has the following features:

- Architecture events provide a formal language to describe execution architectures.
- Filters and maps, together with the visualization tools such as those available in `RAPIDE` allow real time monitoring of execution architectures.
- The `RAPIDE` engine raises exception when the formal specification (i.e., constraints) of an execution architecture is violated. Maps allow corrective actions in non fatal error conditions.
- Static snapshots at consistent cuts provide backward compatibility with previous approaches.
- Posets of architecture events capture the execution architecture of an asynchronous, distributed system in cases where static architectures are not expressive enough.
- Posets can easily be stored and analyzed at a later time.

Our technology is applicable to systems that are distributed, asynchronous and have a high change rate. We believe understanding execution architectures is important because it fills the gap between the abstractness of conceptual architectures and what is actually implemented in systems. In particular, our partially-ordered event-based execution architectures is superior to simple, time-stamped traces of architectural changes.

6. REFERENCES

1. Peled, D. A., et al., *Partial Order Methods in Verification*Series in Discrete Mathematics and Theoretical Computer Science, ed. 29. 1996, American Mathematical Society.
2. Garlan, D. and Shaw, M., *An Introduction to Software Architecture*Advances in Software Engineering and Knowledge Engineering, ed. 1. 1993, World Scientific Publishing Company.
3. Moriconi, M. and Qian, X., *Correctness and Composition of Software Architectures*. In Proceedings of *SIGSOFT'94 Software Engineering Notes*. 1994. New Orleans, LA: ACM Symposium on Foundations of Software Engineering.
4. Perry, D. E. and Wolf, A. L., *Foundations for the Study of Software Architecture*. In Proceedings . 1992. SIGSOFT '92, Software Engineering Notes: ACM Symposium on Foundations of Software Engineering.
5. Thompson, C., ed. *Workshop on Compositional Software Architectures*. 1998, OMG, DARPA, MCC, OBJS: Monterey, California. <http://www.objs.com/workshops/ws9801/report.html>
6. Soni, D., et al., *Software Architecture in Industrial Applications*. In Proceedings of *17th International Conference on Software Engineering*. 1995: ACM.
7. Garlan, D., et al., *ACME - Software Architecture Interchange Language.* 1995.
8. Magee, J., et al., *Specifying Distributed Software Architectures*. In Proceedings of *5th European Software Engineering Conference (ESEC 95)*. 1995. Sitges, Spain. <http://www-dse.doc.ic.ac.uk/research/darwin/>
9. Lamport, L., *Time, Clocks, and the Ordering of Events in a Distributed System.* CACM, 1978. 21(7): p. 558-565.
10. Pratt, V. R., *Modeling concurrency with partial orders.* Int. J. of Parallel Programming, 1986. 15(1): p. 33-71.
11. Fidge, C. J., *Partial Orders for Parallel Debugging*. In Proceedings of *Workshop on Parallel and Distributed Debugging*. 1988. Madiscon, Wisconsin: ACM SIGPLAN/SIGOPS.
12. Mattern, F., *Virtual Time and Global States of Distributed Systems*. In Proceedings of *Parallel and Distributed Algorithms*. 1988: Elsevier Science Publishers.
13. Meldal, S., et al., *Exploiting Locality in Maintaining Potential Causality*. In Proceedings of *10th ACM Symposium on the Principles of Distributed Computing*. 1991. New York, New York: ACM Press.
14. Schwarz, R. and Mattern, F., *Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail.* Distributed Computing, 1994. 7(3): p. 149-174.
15. Santoro, A., et al., *eJava - Extending Java with Causality*. In Proceedings of *10th International Conference on Software Engineering and Knowledge Engineering (SEKE'98)*. 1998. Redwood City, CA, USA. <http://pavg.Stanford.EDU/cep/>
16. Luckham, D. C., et al., *Specification and Analysis of System Architectures using RAPIDE.* IEEE Transactions on Software Engineering, 1995. 21(4).
17. Luckham and Vera, *An Event-Based Architecture Definition Language.* IEEE Transactions on Software Engineering, 1996. 21(9): p. 717-734.
18. Vera, J. S., *Software Architecture Description Languages: Descriptive Constructs and Execution Algorithms.* PhD thesis (forthcoming). Stanford University, Stanford, 1998.

19. Harris, D. R., et al., *Reverse Engineering to the Architectural Level*. In Proceedings of *17th International Conference on Software Engineering*. 1995: ACM.
20. TIBCO, *TIBCO Web Site*, . 1998, TIBCO. <http://www.tibco.com/>
21. Luckham, D. C. and Frasca, B., *Complex Event Processing in Distributed Systems.* Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford, 1998. <http://pavg.Stanford.EDU/cep/>