An Activity Language for the ADL Toolkit

David Garlan and Andrew Kompanek
(with John Kenney, David Luckham, Bradley Schmerl and Dave Wile)
September 2000

1 Motivation and Background

Over the past decade numerous architecture description languages (ADLs) and tools have been developed [2]. Each has certain strengths, and each tends to work in isolation. A desirable goal is to find ways to combine the capabilities so that new architecture development environments can be easily developed by combining existing building blocks for architectural description, analysis, code generation, simulation, testing, etc. Such a collection of capabilities would then serve as an ADL Toolkit for building architecture-based design environments.

A stumbling block to achieving this goal is that different tools use different representations for architecture-related information. Some of these representational variations arise from deep semantic differences in the underlying models. However, others are accidents of history. And even for those that *do* differ significantly, typically there is *some* information that is shared between different languages and tools.

One approach to help ameliorate the situation is to identify areas of common semantic concern, and then propose representations that can capture the commonality. With suitable translators, tools associated with different ADLs could then communicate via these common representations.

A first step towards that vision was taken with Acme, a language for representing architectural structure [3]. Acme leverages the common semantic ground underlying most ADLs, which describe architectural structure as a graph of components and connectors. Acme also permits additional annotations of that structure. These annotations capture extra-structural information specific to certain ADLs or analyses.

Beyond Acme, a natural candidate for such "standardization" is architectural behavior. Many architectural descriptions have ways of describing and analyzing architectural behavior in terms of the architecturally significant *events* that can be exhibited at run time. We call a collection of such events an *activity*. For example, Wright describes behavior in terms of event patterns defined in a subset of CSP [1]. Rapide also describes behavior in terms of event patterns. In addition, Rapide and several tools developed by others allow one to monitor architectural behavior and then analyze the results, or "play it back" as a form of architectural animation [5,6].

Given this area of commonality and the general need for event-based architectural analysis, a next obvious step is to find ways to represent events in such a way that different event-based architectural tools can work together. Ideally, this would lead to common models for representing and analyzing event behavior.

A first step towards sharing event-based behavior is to find a standard way to represent events and collections of events. With an eye toward this goal, a number of researchers at ISI (Wile), CMU (Garlan & Kompanek) and Stanford (Luckham & Kenney) started a dialogue to converge on an event standard. The results are summarized in this working report. The main idea of the proposal is to adopt an Acme-like approach: a simple base-level event representation would capture the minimal, core aspects of events. Additionally, other more tool-specific information could be added to those event representations in the form of annotations.

In the remainder of this report we outline the proposed scheme. We begin by enumerating some of the requirements that we identified early in our discussions. Next we discuss the basic ontology of events, activities and their types. Then we introduce two proposals for concrete representation: one based on XML, the other on an Acme-like syntax. Finally, we illustrate how collections of events derived from existing ADLs can be mapped into the activity language. Appendices (yet to be written) contain additional examples of mapping into other ADLs.

2 Requirements

The activity language will provide an *ADL-neutral* representation of behavior in terms of collections of events. It is intended as a standard representation format to be used in an ADL Toolkit for capturing both the observed behavior and the specification of the intended behavior of a system.

Specific requirements for the event language are grouped into five categories:

a. Event Identifiers
b. Extensibility and Reuse
c. Ordering and Timing
d. Aggregation and Abstraction, and
e. Event Context

Note that the following requirements are expressed informally: words like "occurrence", "event" and "activity" should be interpreted using their usual English meanings.

a. Event Identifiers

In order for tools to process events, it must be possible to distinguish one event from another.

Events should be uniquely identifiable. Event instances should be distinguishable from one another. A solution could be based on unique tags (possibly generated by tools) or implicit identifiers (such as the n^{th} event in some uniquely named sequence of events).

b. Extensibility and Reuse

While some aspects of event enumeration and specification will be fixed, other aspects will need to vary from one usage to another. It should therefore be possible to define specialized event vocabularies and to associate with those events properties specific to a particular ADL or set of tools.

The event model should be extensible with new event concepts. The event language should support the definition of specialized event vocabularies, tailored to a particular domain or tool. This allows the reuse of a common event model (e.g., based on a particular ADL) for description.

Events may have domain-specific properties. An event may have properties associated with it that add semantic detail to the description of the event. For example, a communication event might include a property that describes the data transmitted or the format in which the data is encoded. The particular properties of interest will depend on the application domain.

c. Event Ordering and Timing

Depending on the application, different sorts of information about the timing and ordering of events will be relevant. For example, the behavior of a single process over some period of time might be described as a sequence of events, each of which has a timestamp associated with it. In contrast, a simulation based on a simple state machine model of a system might produce a sequence of events without any timing information. As a final example, consider an event log generated by a distributed simulation with independent distributed clocks: in this case it may possible to assign only a *partial* order to the collection of observed events.

All events are instantaneous with regard to their peer events (e.g., events in the same activity). At a lower level, an event might be seen as a composition of other events that occur in a time span.

***Events may include timestamps*.** An instantaneous event may have a timestamp. The timestamp(s) will include a value as well as a description of the scale/units (e.g., milliseconds) associated with the value. A timestamp may also be associated with a particular clock.

Event traces may be ordered, partially ordered or have no explicit order. Depending on what is being described and how the trace was obtained, a given trace may have an explicit total order, explicit partial order or no explicit ordering at all.

d. Aggregation

Activities should be viewable at different levels of abstraction. At a high level of abstraction a single event may itself represent some activity (collection of events).

Events should be decomposable into sub-events. A particular event may be described in more detail by a collection of sub-events. This can be seen as analogous to representations in Acme. An event could have more than one representation depicting different aspects of the event. Ideally, we would like to be able to specify a mapping between an event and its sub-events. We could use that relationship to perform consistency checks or to infer information about an event from its sub-events (e.g., determine the start and end of an event based on the timestamps of its sub-events). At this point, the constraint mechanism seems to suffice for this mapping purpose. We would investigate other means of mapping if needed, which could include the mechanism for architectural mapping.

e. Context

An observed event may be associated with some part(s) of an architecture – a component or a connector that generated the event, for example. There should be some standard way of referring to the architectural context of the event. In some cases, however, an event may refer to a meta-operation on the architecture – i.e., one that changes the architecture.

Events may occur in the context of some system. There should be a way to indicate *where* in some system an event occurred or what part of the system *caused* the event. For example, the context may be a system, some component within a system, a specific interface of a component, or some communication channel between components. In some cases, this context may refer to an element of an explicit architectural representation of a system – for example, a component in an Acme description could be referenced with its fully qualified name.

A certain class of events represents run-time transformations of a system's architecture. Some events occur in the context of an architecture while others may represent changes to the architecture (e.g., dynamic creation of new components and connections).

3 The Proposal

We distinguish between two basic kinds of entities: *events* and *activities*. Events are individual behavioral occurrences. Events are uniquely identified. Further, one can define *event types*, which prescribe additional constraints on the form of the event.

Activities are collections of events. As the name indicates, an activity is intended to capture some aggregate behavior over time. Activities may have different ways of relating their constituent events. Some activities will be simple linear sequences of events. Other activities may have more complex ordering relationships. It is possible to define *activity types* that identify a set of constraints that some family of activities observes.

Both activities and events may be annotated with *properties* that allow one to associate detailed information with those activities and events. For example, a property might indicate data parameters associated with an event or the context in which that event occurred. A property of an activity might indicate the system context in which the set of activities occurred, the duration of the activity, etc.

Activity Language Concepts

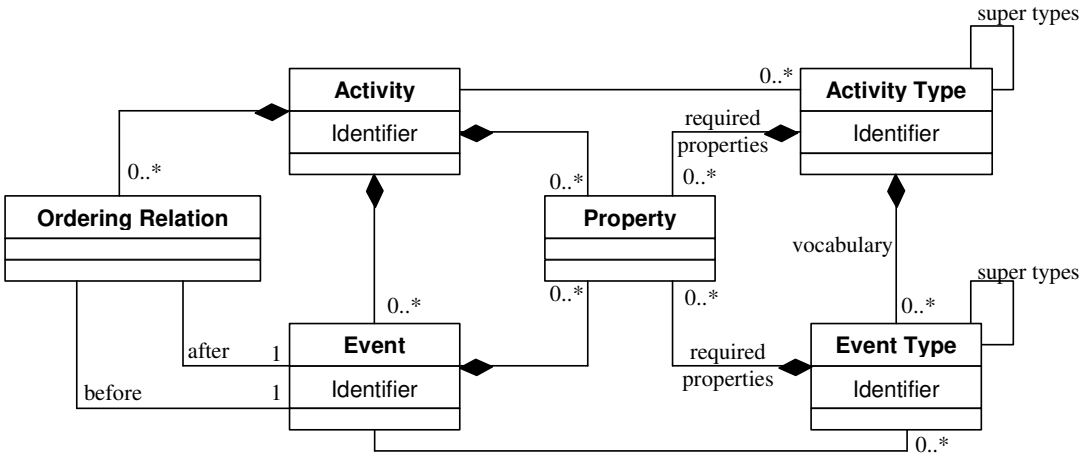

Figure 1. Activity language concepts.

Figure 1 illustrates the UML diagram indicating the main conceptual elements of the proposed activity language and their relationships. The concepts are described below:

Activity — An activity is a description of a behavior consisting of a set of *events*, a set of *ordering relations* over those events, and a set of *properties* that describe auxiliary information associated with the activity. An activity may also be associated with zero or more *activity types* (see below).

Event — An event is a uniquely identified behavioral occurrence that takes place within some system. An event has a set of properties that describe the nature of the event. An event may be associated with zero or more *event types* (see below). An event could have one or more representations, each representation being an activity.

Property — A property is a typed attribute-value pair used to encode semantic information about an event or activity.

Ordering Relation — A set of ordering relations are used to define the temporal (or causal) structure of events within an activity. In general, an ordering relation will determine a partially-ordered set (poset) in which certain events are related to others. In posets, event order is described by a set of explicit ordering relationships which specify pairwise ordering relationships between events in an activity. An important special case of a poset is a sequence (often called a trace).

Activity Type — An activity type defines a family of activities. The members of this family are determined by a predicate associated with that type, which defines a set of constraints that each activity (instance) must satisfy. Constraints include such things as required properties of the activity, the event types that can be included in an activity (i.e., the event vocabulary), and a specification of how events are ordered within an activity of this type. This proposal does not provide a means to declare such an ordering pattern, and it is not its intention to provide one. Given that there are so many notations for declaring the pattern of events (state charts, regular expressions, CSP, just to name a few), and various communities accept one or more of these notations, we decided not to choose any one as our "standard" notation. Instead, any notation could be used to specify an ordering pattern as a property of the activity type. An activity type may also describe

other constraints that must be satisfied by an activity (not yet included in this proposal). An activity type can be defined as an extension/subtype of an existing set of activity types, meaning that it satisfies the constraints of all of those types, in addition to any others that it defines.

Event Type An event type defines a family (or vocabulary) of events. The members of the family are determined by a predicate associated with that type, which defines a set of constraints that each event (instance) must satisfy. Constraints include such things as a set of properties that all instances of this type must have, relationships between those properties, etc. An event type may be defined by extending an existing set of event types.

4 Notations for the Activity Language

Having listed abstractly the main conceptual elements, the question of concrete representation now arises. In this proposal we outline two alternatives. The first is one that adopts an Acme-like notation. The second uses XML. In both cases, we extend the conceptual model by adding an explicit construct for representing the common case of event sequences – a more convenient syntax for representing a linear trace of events.

In the remainder of this section we present both notations. In the following section we give an example of using them.

A. Acme-like Representation

A notation with an Acme-like syntax has the advantage of being easily integrated into the Acme language itself as a standard syntax for describing behavior. In this way, support for behavioral description can be easily integrated into existing Acme-based tools.

```
ACTIVITY ::=
        "Activity" [ACTIVITY-ID] [":" TYPE-ID-LIST] "="
        (ACTIVITY-SEQUENCE | ACTIVITY-POSET) ["with" PROP-LIST] [";"]
ACTIVITY-SEQUENCE ::=
        "<" (EVENT)* ">"
ACTIVITY-POSET ::=
        "{" (EVENT )* "}" "ordered by" "{" (ORD-RELATION ";")* "}"
ORD-RELATION ::=
        EVENT-ID "precedes" EVENT-ID
EVENT ::=
        "Event" [EVENT-ID] ":" [TYPE-ID-LIST] ["in" CONTEXT-DESC] "=" "{"
        [PROP-LIST] | [REPRESENTATION] "}" [";"]
REPRESENTATION ::=
        "representation" [IDENTIFIER] [":" TYPE-ID-LIST]
        "=" (ACTIVITY-SEQUENCE | ACTIVITY-POSET) ["with" PROP-LIST]  [";"]
ACTIVITY_TYPE ::=
        "Activity" "Type" TYPE-ID "=" ("Sequence" | "Poset") "of" TYPE-ID-LIST
        ["with" "{" PROP-LIST "}"] [";"]
EVENT_TYPE ::=
        "Event" "Type" TYPE-ID (("extends" TYPE-ID-LIST "with") | "=")
        "{" PROP-LIST"}" [";"]
TYPE-ID-LIST ::=
        TYPE-ID ("," TYPE-ID)*
CONTEX-DISC =
        ELEMENT-ID ("." ELEMENT-ID)*
PROP-LIST ::=
        (PROPERTY|DESIGNRULE) (";" (PROPERTY | DESIGN-RULE))*
PROPERTY ::= <see Acme>
DESIGN-RULE ::= <see Armani>
```

Type checking these descriptions would be performed by a tool that is a straightforward extension of current Acme/Armani tools.

B. DTD for XML Representation[1]

An XML-based representation has the advantage that behavioral descriptions can be parsed, navigated and displayed using existing XML tools.

```
<!ELEMENT act:ActivityDescription (act:EventType | act:ActivityType |
                                   act:Activity)*>
<!ELEMENT act:Activity (act:EventSequence|act:EventPoset)>
<!ATTLIST act:Activity identifier ID #REQUIRED
                       name NMTOKEN #IMPLIED>
<!ELEMENT act:EventSequence (act:Event*)>
<!ELEMENT act:EventPoset (act:Event*)>
<!ELEMENT act:Event (act:Extends*, act:Context, arch-core:Property*,
                     arch-constraints:DesignRule, act:Activity*)>
<!ATTLIST act:Event identifier ID #REQUIRED
                    name NMTOKEN #IMPLIED>
<!ELEMENT act:Extends EMPTY>
<!ENTITY % xlink "xlink:type (simple) #FIXED "simple"
                  xlink:href CDATA #IMPLIED
                  xlink:role CDATA #IMPLIED
                  xlink:arcrole CDATA #IMPLIED
                  xlink:title CDATA #IMPLIED
                  xlink:show (new|replace|embed|other|none) #IMPLIED
                  xlink:actuate (onLoad|onRequest|other|none) #IMPLIED">
<!ATTLIST act:Extends &xlink;>
<!ELEMENT act:Context EMPTY>
<!ATTLIST act:Context &xlink;>
<!ELEMENT act:Order (Before, After)>
<!ELEMENT act:Before EMPTY>
<!ATTLIST act:Before %xlink;>
<!ELEMENT act:After EMPTY>
<!ATTLIST act:After &xlink;>
<!ELEMENT act:ActivityType (act:Extends,
                            (act:SequenceDesc|act:PosetDesc),
                            arch-core:Property*, arch-constraints:DesignRule)>
<!ATTLIST act:ActivityType identifier ID #REQUIRED
                           name NMTOKEN #IMPLIED>
<!ELEMENT act:EventType (act:Extends*, arch-core:Property*
                         arch-constraints:DesignRule*)>
<!ATTLIST act:EventType identifier ID #REQUIRED
                        name NMTOKEN #REQUIRED
                        supertypes IDREFS #REQUIRED>
<!ELEMENT act:SequenceDesc (EventRef)*>
<!ELEMENT act:PosetDesch (EventRef)*>
<!ELEMENT act:EventRef EMPTY>
<!ATTLIST act:EventRef &xlink;>
```

In this approach, we represent types as we did instances, directly in XML. The Activity Language DTD defines both type elements and instance elements. In this case, type checking of an activity description (i.e., that an activity satisfies the constraints specified by one or more types) must be done using a special tool written for this purpose. The tool would resolve the type references and compare instances to type definitions. Linking between documents, and to contexts and types is achieved using XLink [7], a W3C candidate recommendation for linking between parts of XML documents. We also take advantage of XML namespaces to use portions of other proposed software architecture DTDs. In particular, we use `arch-constraints` to define design rules, and `arch-core` to define properties.

An alternative approach to describing documents would be to represent types using the "element type" mechanism provided by XSchema [8]. Standard XML schema tools could then be leveraged for validation.[2]

---

[1] Note that this DTD specification is subject to change in the light of recent development within the DASADA community of the xArch XML description. It is anticipated that Activities and Events will be an extension of xArch,

[2] Note that the DTD mechanism itself is not rich enough for representing activity types since there it does not include support for sub-typing DTDs.

5 Example: Wright

To illustrate how one can use the activity language to describe the behavior generated or observed of some ADL we consider how one might go about encoding events derived from a Wright specification.

A. Overview

We begin by summarizing the approach:

Name: WrightTrace

Summary of Approach:

Wright behavioral descriptions are derived from CSP. Each such description is defined by a set of "processes" that describe patterns of events using a set of operators for sequencing, choice and parallel composition. Events are either simple names (e.g., e1, e2, read, write), or structured names that indicate a communication channel and some data (e.g., chanA.5, chanB."Hi-world"). We distinguish input events from output events using the convention of "?" for input and "!" for output. We will treat an activity in Wright as one of the possible behaviors that can be realized by a system satisfying a Wright description. For purposes of illustration we will represent a system behavior as an event trace, i.e., a finite sequence of events.[3] (For details see [1, 4])

Event Vocabulary:

In this encoding, we define three types of events that may be instantiated within a Wright activity:

WriteEventT	An event that writes data
ReadEventT	An event that reads data
DoEventT	An event that does not involve I/O

Ordering Constraints:

Events are linearly ordered in an activity (i.e., they form a trace).

Restrictions/Assumptions:

This encoding does not capture failures and divergences, or initiated/observed distinctions

B. Sample system

The following is a description of a simple client-server system in Wright. The system is composed of a single client connected to a single server with one connector. The connector translates requests from the client into requests of the server, and responses from the server into responses received by the client. For simplicity, the example ignores issues of process termination.

```
Connector CSconnector
    Role Client = (request!x -> result?y -> Client)
    Role Server = (invoke?x -> return!y -> Server)
    Glue = (Client.request?x -> Server.invoke!x -> Server.return?y ->
                Client.result!y -> Glue)
Component Client
    Port Send = (request!x -> result?y -> Send)
    Computation = (Send.request!x -> Send.result?y -> displayResult
                                -> Computation)
Component Server
    Port Receive = (invoke?x -> return!y -> Receive)
    Computation = (Receive.invoke?x -> processRequest -> Receive.return!y
                                -> Computation)
Instances
```

[3] This is a special case of the semantics of CSP. A more general form is needed to capture the full meaning of a CSP process.

```
    client : Client
    server : Server
    conn : CSconnector

Attachments
    client.Send as conn.Client
    server.Receive as conn.Server
```

C. Events

For simplicity, we assume that data associated with an event is always represented as a string. A behavior describing a request from the client followed by response from the server would look like:

```
client.Send.request!"question1"      Client sends request
server.Receive.invoke?"question1"    Server is invoked
server.processRequest                Server processes request
server.Receive.return!"answer1"      Server returns result
client.Send.return?"answer1"         Client receives result
client.displayResult                 Client displays result
```

This trace was obtained by placing the client and server computation processes in parallel with the glue, and filtering out connector events.[4]

D. Event Types for Wright Traces

We define four event types for Wright traces using our Acme-like syntax. The first type captures the notion of a generic event. It is defined by a name and an indication of whether the event involves input, output or no I/O. From this abstract event type, we define three sub-types. The input and output event types include a property for encoding the data associated with an event.

```
// Abstract base type
Property Type WrightEventType : enum {do, read, write};

Event Type SimpleEventT = {
       Property wrightEventName : String;
};

// Three event types: no data, output data, input data
Event Type DoEventT extends SimpleEventT with {
       Property eventType : WrightEventType = do;
};

Event Type ReadEventT extends SimpleEventT with {
       Property eventType : WrightEventType = read;
       Property data : String;
};

Event Type WriteEventT extends SimpleEventT with {
       Property eventType : WrightEventType = write;
       Property data: String;
};
```

The following XML defines the types according to our DTD:

```
<arch-typed:PropertyType identifier='WrightEventType'>
  <arch-core:ComplexPropertyType>
    <arch-core:PropertyEnum values='do read write'/>
  </arch-core:ComplexPropertyType>
</arch-typed:PropertyType>
<act:EventType identifier='SimpleEventT'>
  <arch-core:Property name='wrightEventName' type='string'/>
</act:EventType>
<act:EventType identifier='DoEventT'>
```

[4] In this example, we don't represent the behavior of the connector observing the client request and invoking the server. This approach may not always be the appropriate interpretation, especially if the connection is more complex.

```
  <act:Extends xlink:href='SimpleEventT'/>
  <arch-core:Property name='eventType' xlink:href='#WrightEventType' value='do'/>
</act:EventType>
<act:EventType identifier='ReadEventT'>
  <act:Extends xlink:href='SimpleEventT'/>
  <arch-core:Property name='eventType' xlink:href='#WrightEventType'
                      value='read'/>
  <arch-core:Property name='data' type='string'/>
</act:EventType>
<act:EventType identifier='WriteEventT'>
  <act:Extends xlink:href='SimpleEventT'/>
  <arch-core:Property name='eventType' xlink:href='#WrightEventType'
                      value='write'/>
  <arch-core:Property name='data' type='string'/>
</act:EventType>
```

Note the use of xlinks to refer to supertypes. For example, the attribute `xlink:href='#WrightEventType'` is a reference to the XML element in this document that has an identifier `'WrightEventType'`.

E. Activity Type for Wright Traces

Since we are modeling Wright behavior as traces (sequences) composed of the three basic types, we have:

```
Activity Type WrightTraceT =
        Sequence of ReadEventT, WriteEventT, DoEventT
        with {
                // properties and constraints would go here - TBD
        }
```

In XML, this would be represented by the following fragment:

```
<act:ActivityType identifier='WrightTraceT'>
  <act:SequenceDesc>
    <act:EventRef xlink:href='#ReadEventT'/>
    <act:EventRef xlink:href='#WriteEventT'/>
    <act:EventRef xlink:href='#DoEventT'/>
  </act:SequenceDesc>
</act:ActivityType>
```

F. Activity Instance for the Example

To show how an instance of an activity would be represented, we encode the example in three forms: Acme-like syntax using the sequence construct, Acme-like syntax using the poset construct and XML.

First the sequence syntax:

```
Activity csSimulation : WrightTraceT = <
        Event e1 : WriteEventT in client.Send =
        { Property wrightEventName : String = "request"
          Property data : String = "question1"};
        Event e2 : ReadEventT in server.Receive =
        { Property wrightEventName : String = "request"
          Property data : String = "question1"; };
        Event e3 : BasicEventT in server =
        { Property wrightEventName : String = "processRequest"; // no data };
>
```

Using a syntax appropriate for partially ordered sets we have:

```
Activity csSimulation : SimpleWrightTraceT = {
        Event e1 : WrightEventT in client.Send =
        { Property wrightEventName : String = "request";
          Property data : String = "question1"};
        Event e2 : ReadEventT in server.Receive =
        { Property wrightEventName : String = "request";
          Property data : String = "question1"; };
        Event e3 : BasicEventT in server =
        { Property wrightEventName : String = "processRequest"; // no data };
} ordered by {
        e1 precedes e2;
        e2 precedes e3;
```

```
}
```

Using the XML syntax for partially ordered sets:

```
<act:Activity identifier='csSimulation' name='csSimulation'>
  <act:EventPoset>
    <act:Event identifier='csSimulation.e1' name='e1'>
        <act:Extends xlink:href='#WriteEventT'/>
       <act:Context xlink:href='sample.xml#client.Send'/>
       <arch-core:Property name='wrigthEventName' type='string'
                           value='request'/>
       <arch-core:Property name='data' type='string' value='question1'/>
    </act:Event>
    <act:Event identifier='csSimulation.e2' name='e2'>
      <act:Extends xlink:href='#ReadEventT'/>
      <act:Context xlink:href='sample.xml#server.Receive'/>
      <arch-core:Property name='wrightEventName' type='string'
                          value='request'/>
      <arch-core:Property name='data' type='string' value='question1'/>
    </act:Event>
    <act:Event identifier='csSimulation.e3' name='e3'>
      <act:Extends xlink:href='#DoEventT'/>
      <act:Context xlink:href='sample.xml#server' />
      <arch-core:Property name='wrightEventName' type='string'
                          value=processRequest/>
    </act:Event>
    <act:Order>
      <act:Before xlink:href='#csSimulation.e1'/>
      <act:After xlink:href='#csSimulation.e2'/>
    </act:Order>
     <act:Order>
      <act:Before xlink:href='#csSimulation.e2'/>
      <act:After xlink:href='#csSimulation.e3'/>
    </act:Order>
  </act:EventPoset>
</act:Activity>
```

6 References

[1] R. Allen and D. Garlan. *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology. July 1997.

[2] D. Garlan, R. Monroe, and D. Wile. *ACME: An Architecture Interchange Language*. Technical Report, CMU-CS-95-219. Carnegie Mellon University, November 1995.

[3] D. Garlan, R. Monroe, and D. Wile. *Acme: Architectural Description of Component-Based Systems*. In Foundations of Component-Based Systems, edited by G. Leavens and M. Sitaraman. Cambridge University Press, 2000.

[4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J. 1985.

[5] N. Medvidovic and R. Taylor. *A Framework for Classifying and Comparing Architecture Description Languages*. In Proceedings of European Software Engineering Conference 1997, September 1997.

[6] Rapide Design Team. *Rapide 1.0 Language Reference Manual*. Program Analysis and Verification Group, Computer Systems Lab, Stanford University, January 1996.

[7] W3C Consortium. *XML Linking Language (XLink) Version 1.0: W3C Candidate Recommendation 3 July 2000*. Available at: http://www.w3.org/TR/2000/CR-xlink-20000703/

[8] W3C Consortium. *XML Schema Part 0: Primer W3C Working Draft, 7 April 2000*. Available at: http://www.w3.org/TR/2000/WD-xmlschema-0-20000407/