

# Software Architecture Built from Behavior Models

Mikhail Auguston

Computer Science Department  
Naval Postgraduate School  
Monterey, CA, 93943, USA  
email: [maugusto@nps.edu](mailto:maugusto@nps.edu)

## Abstract

This paper suggests an approach to formal software system architecture specification based on behavior models. The behavior of the system is defined as a set of events (event trace) with two basic relations: precedence and inclusion. The structure of event trace is specified using event grammars and other constraints organized into schemas. The schema framework is amenable to stepwise architecture refinement, reuse, composition, visualization, and application of automated tools for consistency checks. The concept of event attribute supports a continuous architecture refinement up to executable design and implementation models.

## Keywords:

Software Architecture Description Language, behavior models

## 1 Introduction

Architecture development is done very early in the software design process and is concerned with the high-level structure and properties of the system [33]. For example, [16] provides the following definition: “The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.”

“When designers discuss or present a software architecture for a specific system, they typically treat the system as a collection of interacting components. Components define the primary computations of the application. The interactions or connections between components define the ways in which the components communicate or otherwise interact with each other. In practice a large variety of component and connector types are used to represent different forms of computation or interaction” [1].

The following aspects have emerged as characteristic for software architecture descriptions [16][42].

- An architecture description belongs to a high level of abstraction, ignoring many of the implementation details, such as algorithms and data structures.

- Software architecture plays a role as a bridge between requirements and implementation.
- An architecture specification should be supportive for the refinement process, and needs to be checked carefully at each refinement step (preferably with tools).
- The architecture specification should support the reuse of well known architectural styles and patterns. Practice has provided several architectural styles and referential architectures, as well established, reusable architectural solutions. There should be flexible and expressive composition operators supporting the refinement process.
- The software architect needs a number of different views of the software architecture for the various uses and users [35] [42] (including visual representations, like diagrams).

One of the major concerns in software architecture design is the question of the behavior of the system. The purpose of this paper is to demonstrate that behavior models may be used as a basis for software architecture description, and structural and some other properties may be extracted from the behavioral specifications.

Presented here is an approach for building system behavior models based on the concepts of event and event traces. This position paper presents a sketch of the architecture description language Monterey Phoenix based on behavior specifications, with an emphasis on the interaction and the coordination aspects of component behavior, although component behavior related to functionality may be captured as well. The approach provides abstractions for building architecture descriptions amenable to refinement toward more detailed design specifications, and for validation and verification with formal methods and automated tools, such as Alloy Analyzer [5][27].

The framework supports extracting multiple views of the system from the same architecture description, specification of behavior patterns at a high level of abstraction, reuse of architectures, and use as a basis for different automated tools. We find an inspiration in the statement by D.Jackson: “Exploiting tools to check arguments at the design and requirements level may be more important, and it is often

more feasible since artifacts at the higher level are much smaller” [29].

The main novelty of this work is in the framework for system behavior modeling based on event traces, which provides a high level of abstraction for systems architecture and its environment descriptions, supports stepwise refinement up to the detailed design models, and allows architecture reuse, composition, and tool use for sanity checks during the process.

This paper is structured as follows. Sec. 2 introduces the concept of behavior modeling based on events and event traces, gives a brief outline of related work on behavior modeling, and provides the first example of an event grammar. The Phoenix concept of schema as an architecture behavior model is described in Sec. 3, with introductory examples and the discussion of a schema’s semantics. Sec. 4 outlines the approach to a schema’s composition, refinement, and reuse. In Sec. 5 event attributes and special events for retrieving event attributes are discussed in the context of refinement towards detailed design and implementation models. Sec. 6 gives a preliminary discussion about the prospects for automation tools use in schema’s specification and refinement, and Sec. 7 suggests potential future work.

## 2 Behavior Models

In a certain sense, software development is a process aimed at the design of a compact description of a set of required behaviors. The source code written in any programming language - a finite object by itself - specifies a potentially infinite number of execution paths.

### 2.1 Software Behavior Models in Related Work

The following ideas of behavior modeling and formalization have provided inspiration and insights for this work.

Literate programming introduced by D.Knuth laid the first inroads in the hierarchical refinement of structure mapped into behavior, with the concept of pseudo-code and tools to support the refinement process [34].

Bruegge, Hibbard [18], and Campbell, Habermann [19] have demonstrated the application of path expressions for program monitoring and debugging. Path expressions in [42] have been used (semi-formally) as a part of software architecture description.

A.Hoare’s CSP (Communicating Sequential Processes) [25][43] is a framework for process modeling and formal reasoning about those models. This behavior modeling approach has been applied to software architecture descriptions to specify a connector’s protocol [3][4][41].

Rapide [37][38] by Luckham et al. uses events and posets of events to characterize component interaction.

D.Harel’s Statecharts became one of the most common behavior modeling frameworks, integrated in broader architecture specification systems UML [17], and AADL [21].

In [50], Wang and Parnas proposed to use trace assertions to formally specify the externally observable behavior of a software module and presented a trace simulator to symbolically interpret the trace assertions and simulate the externally observable behavior of the module specified. The approach is based on algebraic specifications and term rewriting.

The concept of behavior model based on events and event traces was introduced in [8][11][14] as a framework for debugging and testing automation tools.

### 2.2 The Event Concept

The approach here focuses on the notion of an *event*, which is an abstraction for any detectable action performed during the program execution, such as a statement execution, expression evaluation, procedure call, message transmission and reception, etc.

Actions (or events) are evolving in time, and the behavior model represents the temporal relationship between actions. This implies the necessity to introduce an ordering relation for events. Semantics of parallel programming languages and even some sequential languages (such as C) do not require the total ordering of actions, so partial event ordering is the most adequate method for this purpose [36].

Actions performed during the program execution are at different levels of granularity, some of them including other actions, e.g., a subroutine call event contains statement execution events. This consideration brings inclusion relation to the model. Under this relationship events can be hierarchical objects, and it becomes possible to consider behavior at the appropriate levels of granularity.

Two basic relations are defined for events: *precedence* (PRECEDES) and *inclusion* (IN). Any two events may be ordered in time, or one event may appear inside another event.

The behavior model of the system can be represented as a set of events with these two basic relations defined for them (*event trace*). Each of the basic relations defines a partial order of events. Two events are not necessarily ordered, that is, they may happen concurrently. Both relations are transitive, non-commutative, non-reflexive, and satisfy distributivity constraints. The following axioms should hold for any event trace. Let a, b, and c be events of any type.

#### Mutual Exclusion of Relations

*Axiom 1)*  $a \text{ PRECEDES } b \Rightarrow \neg(a \text{ IN } b)$

*Axiom 2)*  $a \text{ PRECEDES } b \Rightarrow \neg(b \text{ IN } a)$

*Axiom 3)*  $a \text{ IN } b \Rightarrow \neg(a \text{ PRECEDES } b)$

Axiom 4)  $a \text{ IN } b \Rightarrow \neg(b \text{ PRECEDES } a)$

**Non-commutativity**

Axiom 5)  $a \text{ PRECEDES } b \Rightarrow \neg(b \text{ PRECEDES } a)$

Axiom 6)  $a \text{ IN } b \Rightarrow \neg(b \text{ IN } a)$

Irreflexivity for PRECEDES and IN follows from non-commutativity.

**Transitivity**

Axiom 7)  $(a \text{ PRECEDES } b) \wedge (b \text{ PRECEDES } c) \Rightarrow (a \text{ PRECEDES } c)$

Axiom 8)  $(a \text{ IN } b) \wedge (b \text{ IN } c) \Rightarrow (a \text{ IN } c)$

**Distributivity**

Axiom 9)  $(a \text{ IN } b) \wedge (b \text{ PRECEDES } c) \Rightarrow (a \text{ PRECEDES } c)$

Axiom 10)  $(a \text{ PRECEDES } b) \wedge (c \text{ IN } b) \Rightarrow (a \text{ PRECEDES } c)$

Event trace is always a directed acyclic graph.

**2.3 Event Grammar**

The structure of possible event traces is specified by an *event grammar*. A grammar rule specifies structure for a particular event type (in terms of IN and PRECEDES relations). A grammar rule has form,

$$A:: \text{right-hand-part};$$

where A is an event type name.

Event types that do not appear in the left hand part of rules are considered atomic and may be refined later by adding corresponding rules.

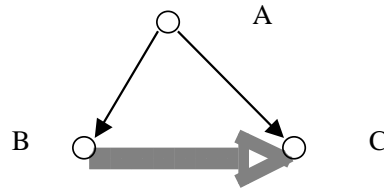
There are the following *event patterns* for use in the grammar rule's right hand part. Here B, C, D stand for event type names or event patterns.

Events may be visualized by small circles, and basic relations - by arrows, like

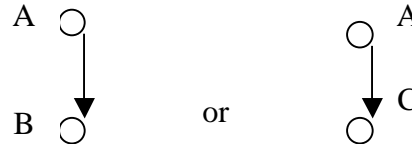


1) Sequence denotes ordering of events under the PRECEDES relation. The rule  $A:: B C$ ; means that an event a of the type A contains ordered events b and c matching B and C, correspondingly ( $b \text{ IN } a$ ,  $c \text{ IN } a$ , and  $b \text{ PRECEDES } c$ ). A grammar rule may contain a sequence of more than two events, like  $A:: B C D$ ;

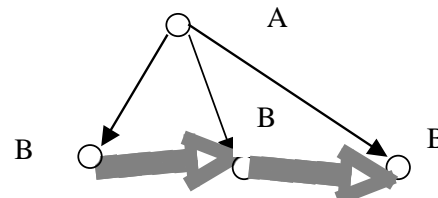
The rule  $A:: B C$ ; specifies the following event trace.



2)  $A:: (B | C)$ ; denotes an alternative - event B or event C.

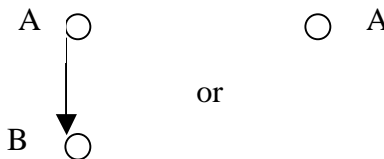


3)  $A:: (* B *)$ ; means an ordered sequence of zero or more events of the type B. Here is an example of an event trace satisfying this pattern:

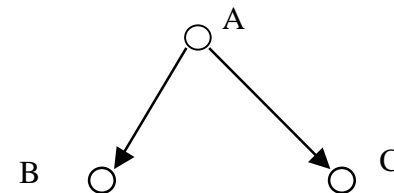


The relations induced by the transitivity and the distributivity axioms are not explicitly shown in this and following pictures.

4)  $A:: [B]$ ; denotes an optional event B.



5)  $A:: \{ B, C \}$ ; denotes a set of events B and C without an ordering relation between them.



6)  $A:: \{ * B * \}$ ; denotes a set of zero or more events B without an ordering relation between them.

Extension (+ B +) may be used to denote a sequence of one or more events B, and {+ B +} as a set of one or more events B. Together with an (\* ... \*) event pattern those may be useful for specifying dynamic architectures. In both cases it is assumed that all iterated event instances are unique.

#### 2.4 Example of an Event Grammar

The Shooting\_competition event contains a number of independent (i.e., potentially concurrent) Shooting events.

```
Shooting_competition:: { * Shooting * };
```

The Shooting event contains a sequence of zero or more Shoot events.

```
Shooting:: ( * Shoot * );
```

Each Shoot event starts with Fire event and may have one of two possible outcomes: Hit or Miss.

```
Shoot:: Fire ( Hit | Miss );
```

Together these event grammar rules specify a set of possible event traces representing different scenarios for the Shooting\_competition.

### 3 Schema as a Behavior Specification for an Abstract Machine

An abstract machine is a model of a software system. The behavior of a particular abstract machine is specified as a set of all possible event traces using a *schema*. The concept of the schema in Monterey Phoenix has been inspired by Z schema [48]. The schema is similar to the fundamental architectural concept of *configuration*, which is a collection of interacting components and connectors, as introduced in [1]. The purpose of the Phoenix schema is to define the structure of possible event traces (in terms of IN and PRECEDES relations) using event grammar rules and other constraints.

The schema may define the behavior of the abstract machine on different levels of abstraction/granularity. Any event trace specified by a schema should also satisfy Axioms 1) – 10), i.e. all basic relations induced by the Axioms are included in the event trace structure by default. A schema may define both finite and infinite traces, but most analysis tools for reasoning about a system's behavior assume that a trace is finite.

Specifying the PRECEDES relation for a pair of events in the schema is usually a substantial design decision, manifesting the presence of a cause/effect in the model or other essential dependency condition for these events.

Some events appearing in the schema's rule section at the left-hand side of the grammar rule are marked as *root events*. A root event type should not appear in the grammar rule's right hand part. There is precisely one instance of each root event in

any trace defined by the schema. The schema also may contain auxiliary grammar rules defining event types used in the right-hand part of other grammar rules or providing additional structure constraints. Usually root events correspond to the components and connectors in traditional architecture descriptions, while other event types are used to specify event structure and interactions.

#### Example 1. Simple transaction.

A very simple architectural model contains two components TaskA and TaskB with a connector between them. The presence of a connector usually means that components can interact, for example send and receive a message, call each other and pass a parameter, or use a shared memory to deliver a data item. It is common to represent this kind of a system in a diagram like in Figure 1.



Figure 1. Two components and a connector

The schema called Simple\_transaction specifies the behavior of components involved in a single transaction.

```
Simple_transaction
```

---

```
root TaskA:: Send;
```

```
root TaskB:: Receive;
```

```
root Transaction:: Send Receive;
```

---

```
TaskA, Transaction share all Send;
```

```
TaskB, Transaction share all Receive;
```

The rule section introduces root events TaskA, TaskB, and Transaction, while Send and Receive events are needed to specify the root event's structure. The grammar rules specify the structure of the event trace in terms of relations IN and PRECEDES. There is a single event of type TaskA containing a single event Send (IN relation). Similarly for TaskB and Receive. The single event Transaction contains two events – Send and Receive ordered w.r.t. PRECEDES relation. The use of PRECEDES represents the intention to model a cause/effect relationship.

The event type stands for a set of event traces satisfying the event structure defined for that type. The constraints section uses the predicate **share all**, which is defined as following (here X, Y are root events, and Z is an event type).

$$X, Y \text{ share all } Z \equiv \{ v: Z \mid v \text{ IN } X \} = \{ w: Z \mid w \text{ IN } Y \}$$

This equality holds for each trace that contains at least one Z, and there exists at least one event trace satisfying the schema,

such that the X and Y in this trace each have at least one event of the type Z.

Event sharing in Phoenix plays the role of a coordination mechanism similar to the communication events in CSP [25][43].

### 3.1 Schema Interpretation

Similar to context-free grammars, event grammars can be used as production grammars to generate instances of event traces. The grammar rules in a schema  $S$  can be used for the *basic trace* generation, with the additional schema constraints and Axioms filtering the generated traces to a set of traces called  $\text{Basic}(S)$ . This set contains only traces, which satisfy all schema's constraints, and have only events and relations imposed by the schema's grammar rules and Axioms. The process of generating traces from  $\text{Basic}(S)$  defines the semantics of the schema  $S$  and could be specified in the form of an abstract machine, for example, similar to the Abstract Chemical Machine [15] [26]. If such an abstract machine can be designed for a particular version of Phoenix, it becomes possible to claim that schemas *are executable*.

The schema represents instances of behavior (event traces), in the same sense as a Java source code represents instances of program execution. Just as a particular program execution path can be extracted from a Java program by running it on the JVM, similarly a particular event trace from the  $\text{Basic}(S)$  can be extracted by running  $S$  on the Phoenix abstract machine, i.e. by generating a trace instance.

Traces from  $\text{Basic}(S)$  can be refined by introducing additional events, event types, and basic relations between them, while maintaining the consistency with the original trace's constraints and Axioms. The set of all refined traces for  $S$  is called  $\text{Refined}(S)$ . The schema  $S'$  is a *refinement* of  $S$  if

$$\text{Basic}(S') \subseteq \text{Refined}(S)$$

Checking this property during the schema's refinement process may be one of the main applications for formal methods and tools supporting the Phoenix framework.

Figure 2 renders the only event trace from the  $\text{Basic}(\text{Simple\_transaction})$ . There may be other traces consistent with the structure imposed by the schema, for example, a trace from Figure 2 with an additional relation  $\text{TaskA PRECEDES TaskB}$ , but those traces with redundant relations (or redundant events) not imposed by the schema are not accepted as members of the basic trace set defined by the schema.

Alloy Analyzer [5][28] is a good candidate for implementing the Phoenix abstract machine. Appendix 1 contains the Alloy model, which may produce basic traces for the  $\text{Simple\_transaction}$  schema (like on Figure 2).

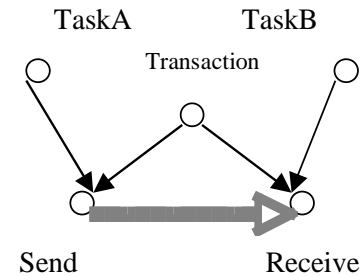


Figure 2. Example of event trace for  $\text{Simple\_transaction}$  schema

This example demonstrates that both a component and a connector within a model are uniformly characterized by the patterns of behavior; each of them is modeled as a certain activity using an event trace. Synchronization patterns may be specified in the additional schema's section, using **share all** constraints. This behavior composition operation may be considered as a certain simplification and unification of the "role" and "glue" concepts in [3][4].

The schemas may be viewed as a way to specify sets of event traces with basic binary relations or graphs, i.e. as a kind of graph grammars [23] in a textual form. Behaviors may be specified at an appropriate level of abstraction, without imposing unnecessary constraints and details of the behavior.

#### Example 2. Multiple strictly synchronized transactions (simple pipe/filter).

Yet another interpretation of the architecture in Figure 1 may assume that components are involved in a strictly synchronized stream of transactions, i.e., the next Send may appear only when the previous Receive has been accomplished.

#### Multiple\_synchronized\_transactions

```

root TaskA:: (* Send *);
root TaskB:: (* Receive *);
root Connector:: (* Send Receive *);

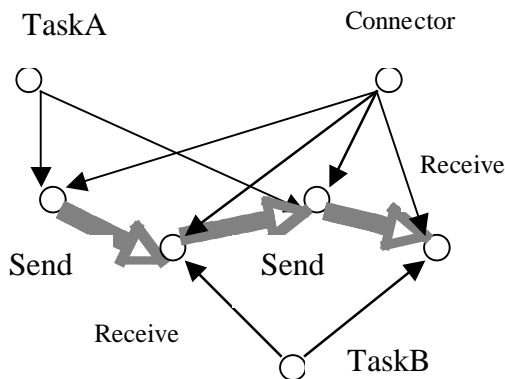
```

---

```

TaskA, Connector share all Send;
TaskB, Connector share all Receive;

```



**Figure 3.** Example of event trace for multiple synchronized transactions

An event trace specified by a schema always satisfies Axioms hence the transitivity of PRECEDES between consecutive Receive and Send (PRECEDES relations enforced by the transitivity are not shown).

The Connector event represents the communication activity, and may be refined further to provide details of the synchronization protocol.

### Example 3. Client/Server architecture [4][46]

#### Client\_server

```

root Client::  { * Request * };
root Server::  { * Provide * };
root Connector:: Initialize
                { * ( Request Provide ) * }
                Close;

```

Client, Connector **share all** Request;

Server, Connector **share all** Provide;

In this behavior model requests may arrive in arbitrary order, each Request should be met by a corresponding Provide event (cause/effect relation is imposed), and the whole connector should be initialized before the first transaction and closed after the end of work.

### 3.2 Architecture Visualization and Views

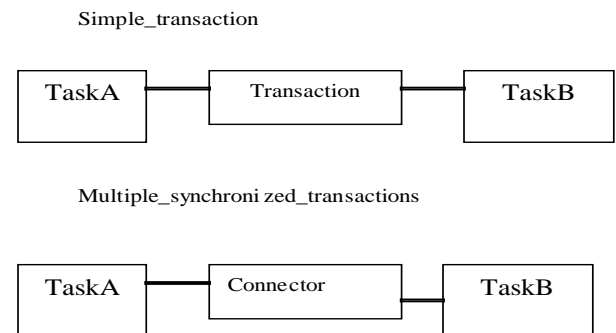
Different abstract views could be extracted from the Phoenix schemas. For example, let X and Y be the root events in the schema S. If there is at least one event trace in Basic(S), such that the predicate

$$\text{CONNECTED}(X, Y) \equiv \exists a ((a \text{ IN } X) \text{ and } (a \text{ IN } Y))$$

is true on that trace, then a dependency exists between events X and Y in terms of sharing an event. This may be a data item sharing event, or method's call, or any other synchronized

activity between X and Y captured by the schema model. The root events correspond to components and connectors, and usually are rendered as boxes in architecture diagrams. The dependency between them could be visualized by connecting corresponding boxes in the diagram.

The architecture diagrams in Figure 4 (view of the static structure with respect to the root events) are extracted from behavior models in schemas Simple\_transaction and Multiple\_synchronized\_transactions and are based on the CONNECTED predicate.



**Figure 4.** Diagrams extracted from the schemas

Those models seem to be more abstract than corresponding schemas, and are missing some details about the component's interaction, but still may be of interest. This kind of abstraction is similar to a number of architecture description techniques currently in use. For example, [33] points out basic architecture elements being components, connectors, and behavior.

Defining appropriate predicates on the events in the schema and mapping those events and relations into different kinds of diagrams may yield multiple architecture views. The IN relation provides for choice of granularity in rendering the hierarchy of models. Event threads loosely coupled by synchronization events (event sharing) may be used to model the separation of concerns issues. It seems that this way of extracting views from the architecture descriptions could provide at least the logical, process, and scenario (or event trace) views as suggested in the 4+1 view model [35].

## 4 Composition and Reuse

The compiler's front-end model is inspired by [42][26] and the unforgettable picture of compiler architecture from the "Dragon Book" [2] (page 13).

### Example 4. Compiler front end in batch processing mode.

The simple model of lexical analyzer captures the behavior of the typical LEX machine.

Lexer


---

```

Input::      (* (Get_char | Unget_char) *);
Output::     (* Put_token *);
root Processing:: (* Token_recognition *);
include      Token_processing

```

---

```

Processing, Input share all Get_char, Unget_char;
Processing, Output share all Put_token;

```

The root event is Processing, whereas Input and Output are auxiliary events to a large degree similar to the *role* concept in [4]. Their role is to define sort of pre- and post- conditions for the Processing component, formalizing our assumptions about input and output streams of events. These constraints should be checked for consistency when added to the schema.

The structure of the Token\_recognition event is defined in the schema Token\_processing and is included (reused) in the Lexer schema. It refines the Lexer behavior toward the typical Unix/LEX semantics, when the regular expression in each LEX rule is applied independently, and hence no ordering is imposed. Each RegExpr\_Match consumes one or more Get\_char events until all finite automata involved in the token recognition enter the Error state, then the winner is selected, and all look-ahead characters beyond the recognized lexeme are returned back into the input stream by Unget\_char; the Fire\_rule event follows it. As a result of the **include** composition operation the root mark is deleted.

Token\_processing


---

```

root Token_recognition:: { * RegExpr_Match * }
                          (* Unget_char * )
                          Fire_rule;
RegExpr_Match::          (+ Get_char +);
Fire_rule::              Put_token;

```

---

```

all RegExpr_Match share all Get_char;
|{x: Get_char | x IN Token_recognition}| >
  |{ y: Unget_char | y IN Token_recognition}|;

```

The first constraint enables the synchronization between a sequence of one or more consecutive Get\_char and a single Put\_token, which follows this Get\_char group via the Fire\_rule. The second constraint ensures that at least one character will be consumed. All those constraints are imposed on the Lexer behavior when the schema is included.

The following schema provides a rough model of bottom-up parsing with a stack (represented by Push and Pop events).

Parser


---

```

Input::      (* Get_token *);
Output::     (* Put_node *);
root Parsing:: Push -- push the start symbol
                (* Get_token (* Reduce *) Shift *)
                [Syntax_error];

```

---

```

Shift::      Push ;
Reduce::     (+ Pop +) Push Put_node;
include     Stack;

```

---

```

Parsing, Input share all Get_token;
Parsing, Output share all Put_node;
Parsing, Stack share all Pop, Push;

```

Put\_node events represent the construction of a parse tree. The behavior of the stack can be encapsulated for reuse in a separate schema and included in the Parser schema when needed. Stack behavior constraint will be inherited from the **include** operation.

Stack


---

```

root Stack_operation:: (* ( Push | Pop ) *);

```

---

```

∀x: Pop ( |{ y: Push | y PRECEDES x}| >
          |{ z: Pop | z PRECEDES x}| );

```

The constraint reflects the absence of stack underflow.

To merge both Lexer and Parser schemas into a single schema we need to tell how those components will interact. The following schema specifies batch processing.

Batch\_processing


---

```

root Batch::      Produce_tokens Consume_tokens;
Produce_tokens::  (* Put_token *);
Consume_tokens::  (* Get_token *);

```

---

```

|{x: Put_token | x IN Batch }| >=
  |{y: Get_token | y IN Batch}|

```

The ordering of Produce\_tokens and Consume\_tokens events in this schema ensures that production of the whole set of tokens will precede the consumption. The constraint requires that the number of produced tokens is sufficient, although there is no specific requirement how the tokens are consumed (e.g. by storing them in the queue or on the stack).

The composition of the component and the connector architectures is described by the schema composition operation **merge**. The result is a new schema Batch\_parsing. Since the Phoenix schema represents a set of event traces defined by the rules within the schema, the schema's name may be used instead of X, Y in the **share all** predicate. In this case, sharing is extended to events Z defined in the schema. In fact, the schema S introduces an event type S, such that for all root events A within S relation A IN S holds. This provides a rationale for the fact that schemas may be operands for the **share all**.

Batch\_parsing

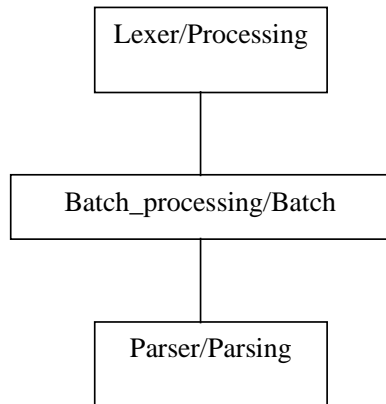

---

**merge** Lexer, Parser, Batch\_processing ;
 

---

 Lexer, Batch\_processing **share all** Put\_token;  
 Parser, Batch\_processing **share all** Get\_token;

The diagram on Figure 5 represents a simplified component/connector view of this architecture.



**Figure 5. Diagram extracted from the Batch\_parsing schema**

**Example 5. Compiler's front end in incremental mode.**

Yet another possible interaction is a mode in which the Parser requests the next token and triggers an event inside the Lexer, generating a token (the traditional LEX/YACC operation pattern). The schema *Incremental* represents this operation mode. The IN relation imposed here reflects the cause/effect dependency or synchronization between events from Lexer and Parser schemas involved in the token request/delivery. In fact, the *Get\_token* event is now refined with the *Token\_recognition* event.

Incremental


---

*Get\_token*:: *Token\_recognition*;
 

---

The composition of components with another connector schema is done in the same fashion.

Incremental\_parsing


---

**merge** Lexer, Parser, Incremental;
 

---

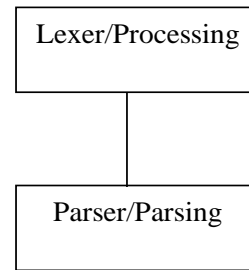
 Lexer, Parser **share all** *Token\_recognition*;
 

---

The merged architecture defines a set of event traces where all structuring is inherited from Lexer, Parser, and Incremental schemas with the additional constraints for sharing the token processing events. The basic sanity checks for consistency of merged event sets (traces) may be reduced to standard regular

expression equivalence and inclusion problems, and can be done by automated tools.

The root events are inherited from all schemas. With the help of a *CONNECTED* predicate the root events may be rendered in Figure 6 diagram. This time both Lexer and Parser components are interacting directly via the shared *Token\_recognition* event, and there is no connector box.



**Figure 6. Diagram extracted from the Incremental\_parsing schema**

As [4] points out "... [there] is the need for a simple but powerful form of composition. Architecture is inherently about putting parts together to make large systems." The composition of the *Incremental\_parsing* schema bears an analogy with the Aspect Oriented Programming approach [32].

The examples above demonstrate the architecture reuse. In practice the merge operation may yield a schema, that does not have any event traces at all because of inconsistencies resulting from imposing the union of constraints inherited from the participating schemas. Tools like Alloy Analyzer [5] can be used for sanity checks to verify whether the merged schema still has trace instances.

**4.1 Architectural Styles and Patterns**

Here is yet another common example of a small architectural style.

**Example 6. Buffered pipe/filter.**

Suppose that the communication between the components is implemented via a buffer of size *N*, and not necessarily all sent messages should be consumed, i.e. some of them could stay in the buffer indefinitely. Each message may be consumed no more than once, and the ordering of receiving does not necessarily correspond to the ordering of sending.

Buffered\_transaction


---

**root** TaskA:: (\* Send \*);

**root** TaskB:: (\* Receive \*);

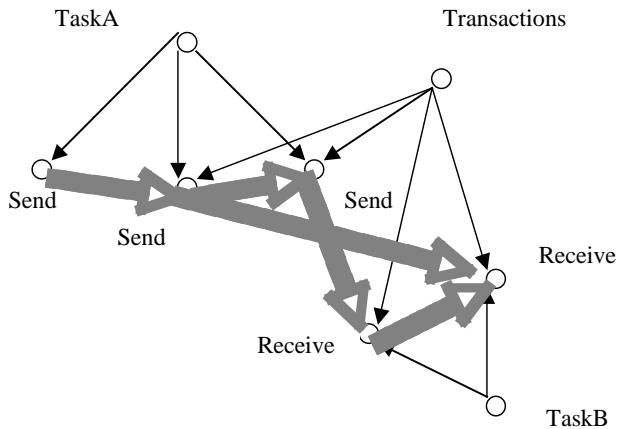
---

**root** Transactions:: { \* (Send Receive) \* };
 

---


$$\{ x: \text{Send} \mid x \text{ IN Transactions} \} \subseteq \{ x: \text{Send} \mid x \text{ IN TaskA} \};$$

TaskB, Transactions **share all** Receive;

$$\forall x: \text{Send} ( \{ \{ y: \text{Send} \mid y \text{ PRECEDES } x \} \mid \{ z: \text{Receive} \mid z \text{ PRECEDES } x \} \} < N);$$


**Figure 7.** Example of Buffered\_transaction trace

## 5 Event Attributes and Refinement

At the top levels of architecture description schemas usually are focused on capturing event trace structure in terms of basic relations and event sharing (synchronization). With the progress of refinement the need to introduce more detailed view on data flow starts to appear. As [20] puts it: “The best architecture is worthless if the code does not follow it”. In order to specify meaningful system behavior properties events are enriched with *attributes* such as type, event begin time, end time, duration, program state associated with the event (i.e. variable values at the beginning and at the end of event), etc.

To manage event attribute values the concept of *special event* is introduced. Typically a special event represents some operation with event attributes and is enclosed in a pair of symbols / and /.

/ action changing or retrieving attribute values /

In addition there are special events that may influence structure of event trace for alternatives depending on conditions of certain event attributes, like

IF (condition involving attributes) THEN E1 ELSE E2

This special event refines on the event alternative (E1 | E2) by making the choice depending on the value of the condition.

In a similar way the number of event iterations may be constrained by conditions involving attributes, like

WHILE(condition involving attributes) (\* E \*)

or

(\* E \*) UNTIL(condition involving attributes)

The number of iterations for (\* ... \*) and { \* ... \*} may be indicated explicitly as well, like (\* A \*) (50).

Special event may be subjected to the basic relations IN and PRECEDES like any other event. The additional constraint is that the semantics of special events requires them to be executed in accordance to the PRECEDES. Thus, if for special events S1 and S2 holds

S1 PRECEDES S2

then S1 should be evaluated before S2. If there is no PRECEDES relation the order of evaluation is non-deterministic.

Special events make it possible to refine schemas (i.e. sets of event traces) close to the detailed design or even implementation level. This notation provides a support for pseudo code development inspired by [34]. The structure of event trace may depend on attribute values.

### Example 7. Implementation model

Phoenix emphasizes top-down design. Using special events and event attributes it becomes possible to refine schemas to the level when mapping into executable program becomes straightforward.

Factorial\_calculation

```

root Main::    /enter (Factorial.input);/
               Factorial
               /print (Factorial.output);/ ;

Factorial:: IF ( Factorial[1].input <= 1) THEN
              /Factorial[1].output = 1;/
            ELSE
              (/Factorial[2].input = Factorial[1].input - 1;/
              Factorial
              /Factorial[1].output =
                Factorial[2].output * Factorial[1].input;/) ;

```

The Factorial[1] denotes the first instance of the event Factorial in the left hand part of the rule, and Factorial[2] denotes the second instance of the event Factorial within the rule. This event grammar describes event traces (sequences of special events in this case) representing calculation of the factorial and depending on the attribute Factorial.input. It is obvious that this model can be transformed into implementation in some common programming language.

### Example 8. Refinement toward design

As another example, let's refine the Shooting\_competition model introduced in Section 2.3 into a more detailed simulation, which may be converted into an executable program, e.g. in Java with threads.

```
Shooting_Competition ::
  { * / Shooting.ammo =10; /
    Shooting          * } (100);
```

The attribute Shooting.ammo is bound with the corresponding Shooting event. The (100) constraint sets the number of Shooting events within the Shooting\_competition.

```
Shooting:: Load / Shooting.points = 0; /
  WHILE (Shooting.ammo > 0)
    (* Shoot
      /Shooting.ammo -=1;/ *);
```

The WHILE constraint causes the number of Shoot events to be controlled as in a traditional While loop statement, and effectively implies that the number of Shoot events is equal to the initial value of Shooting.ammo attribute.

```
Shoot:: Fire
  ( Hit | Miss)
  / ENCLOSING Shooting .points +=
    Shoot.points; /;
```

```
Hit:: /ENCLOSING Shoot. points = Rand[1..10];/;
```

```
Miss:: /ENCLOSING Shoot. points = 0;/;
```

These grammar rules do not yet provide a mechanism for making a decision regarding choice of Hit or Miss event, but such additional constraint can be added if needed.

Nevertheless, the event grammar above defines a set of event traces approximating the expected behaviors pretty closely. ENCLOSING scope is determined by the IN event hierarchy, and is a technical convenience eliminating attribute copying chains, like in traditional attribute grammars [39]. This model is not yet completely executable in traditional sense, but could be refined into one by adding necessary design details. In general, each schema refinement may add new events and additional constraints to event traces and therefore narrows the set of event traces.

### 5.1 Models of the environment

#### Example 9. A model of a system interacting with the environment.

The User schema represents the environment behavior in which the Calculator operates.

#### User

---

```
Use_calculator:: (* Perform_calculation *);
```

```
Perform_calculation::
  Enter_number
  Enter_operator
  Enter_number
  Request_result;
```

```
Enter_number:: (+ Press_digit_button +);
```

The model of Calculator.

#### Calculator

---

```
Calculator_in_action:: (* Single_calculation *);
```

```
Single_calculation::
  Get_number Get_operator Get_number
  IF (Get_operator.operation == '+') THEN
    / Single_calculation.result =
      Get_number[1].value +
      Get_number[2].value; /
  ELSE
    / Single_calculation.result =
      Get_number[1].value -
      Get_number[2].value; /
  Show_result ;
```

```
Get_number:: / Get_number.value= 0; /
  (* Get_digit
    / Get_number.value =
      Get_number.value * 10 +
      Get_digit.value;/ *);
```

```
Show_result:: /show_result(ENCLOSING
  Single_calculation.result);/;
```

The following schema defines the interaction between the User and the Calculator by establishing a connection between events in the environment and in the system.

#### Connection

---

```
Press_digit_button::
  /Get_digit.value = Press_digit_button.value;/
  Get_digit ;
```

```
Enter_operator::
  / Get_operator.operation = Enter_operator.operation;/
  Get_operator;
```

```
Request_result:: Show_result;
```

The model of a calculator interacting with the environment:

User\_and\_Calculator

---

**merge** User, Calculator, Connection;

---

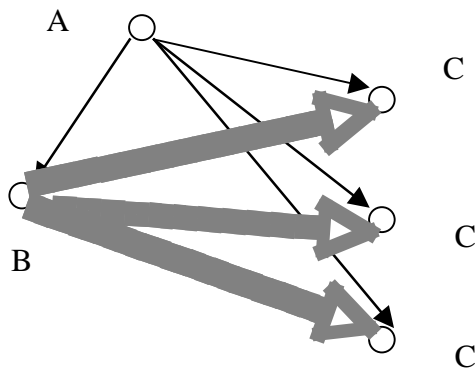
User, Calculator **share all** Get\_digit,  
Get\_operator, Show\_result;

## 5.2 More Grammar Rules

Sometimes the multiplicity of event pairs with the PRECEDES should be specified explicitly. The multiplicity notation may be used within grammar rules.

A:: ( B 1 - 1..\* C);

One B and one or more C under PRECEDES.



A:: B C; is the default for A:: (B 1 - 1 C);

A:: (B 1 - \* C); means one B and zero or more C.

A:: B [C]; is the same as A:: (B 1 - 0..1 C);

### Example 10. Blackboard architecture.

Messages are announced on the blackboard and consumed in arbitrary order, each message may be consumed zero or more times [44]. This generic schema may be merged with component schemas, which impose more constraints on Announce/Receive events, for instance, some ordering.

---

Blackboard\_transaction

Input:: { \* Announce \*};

Output:: { \* Receive \*};

---

**root** Transactions:: { \* (Announce 1 - \* Receive) \*};

Input, Transactions **share all** Announce;

Output, Transactions **share all** Receive;

## 6 Model Building, Analysis, and Tools

The following discussion inevitably is a very preliminary and has the purpose to outline the potential of the Phoenix formalism to support “lightweight” automated formal

methods. Each of the suggested directions requires time and effort to build and evaluate corresponding tools, some pretty straightforward, some more demanding.

It has been acknowledged within the Software Architecture community that there is a relationship between architectural design and quality attributes [45]. This implies the importance of tools for architecture properties evaluation, for checking the conformance between architecture and code, and for software testing on the basis of software architecture. It seems that the Phoenix framework may be responsive to these needs.

First of all, since the approach is based on the concept of event trace as a set of events with two basic relations and additional constraints imposed by schemas, like event sharing, the full might and glory of tools like Alloy Analyzer [5][28] may be deployed to generate instances of models (i.e. instances of event traces), and to check properties expressed with set-theoretical operations and first order predicate logic. Design of a model transformation tool from Phoenix to Alloy is feasible. Appendix 1 provides an example of the Alloy model for generating Basic(Simple\_transaction) event traces.

Another path is represented by the experience of using CSP notation [25][43][41] to specify architecture connector protocols [3][4] and to apply existing model checking tools like FDR [22] or PAT [40] to verify these protocols. It is plausible that a subset of Phoenix could be converted into models verifiable with such tools.

Yet another way to check properties of event traces is exemplified by the FORMAN language [6][7][8][9] [10][11] [12][30][31] for computations over event traces, which supports generic trace property checks. Event trace generation may be implemented directly based on the event grammars.

Similarly to Alloy, the same formalism used to specify schemas could be used to specify assertions about event traces (in a way as schema constraints do), and is amenable to applying tools to verify or refute those assertions with tools like Alloy Analyzer. The spectrum of properties covers a broad range from purely structural properties (e.g. to assert that selected subset of events within trace is totally ordered) to more detailed assertions involving event’s timing attributes. Of special interest may be properties involving event’s timing attributes. Event duration and frequency estimates obtained from the model may be used to figure out throughput and latency estimates, in particular, when combined with the environment behavior models.

The above considerations lead to the conjecture that model refinement may be supported by reasonable automated sanity checks. It has been noted that architecture description framework needs “capabilities to validate, and measure the performance of architecture before the system is built, and to test the conformance of a system to a predefined standard architecture” [37].

Schemas visualization is of interest as well, and may be supported by tools to define/extract/render customized diagrams from the Phoenix models.

## 7 Future work

First of all, schema formalism needs precise and rigorous syntax and semantics definition to make the structure of event traces defined by the schema unambiguous. An abstract machine able to generate basic traces could provide an acceptable solution to Phoenix semantics definition.

Second, a methodology, guidelines, and a representative collection of reusable templates for the Phoenix framework are essential to help users to develop intuition and skills to use it, in particular, the skill of choosing the right abstraction level for refinement and reuse.

Software architecture modeling touches on the very fundamental issues in software design process and has substantial consequences for the next phases in software system design. This paper is just a very preliminary sketch of some approaches to the problem. There are many threads of future research stemming from the ideas described above. Each of those will require significant investment into a rigorous design and experimentation. The following outlines several prospective directions to pursue.

### *Schemas composition and reuse.*

A good catalog of typical architecture styles and templates (e.g. different specializations of broadcasting, pipe/filter, client/server, layered architectures) could provide the basis for automated code synthesis tools. This also requires an assortment of schema composition operators. Operations with schemas will need tools for simple sanity checks, trace instance generation, and refinement checking.

### *Visualization and architecture views.*

Libraries of predefined predicates, functions, and tools to extract and visualize views are needed.

### *Assertion checking.*

Since schemas are executable via event trace generation on the Phoenix abstract machine, it becomes possible to do some model testing with respect to the formal properties specified in Phoenix formalism.

### *Throughput/latency estimate.*

Given duration and frequency estimates for events within components and connectors it becomes possible to estimate throughput and latency.

### *Environment models and Business Process Models for System Engineering.*

Behavior of the environment may be modeled by event grammars [13] [14] and merged with the system models. The result is amenable to the same kind of analysis and verification

as a stand-alone architecture long before the detailed design and implementation of the system are available [47]. The model of interaction between the system and its environment may be of special value for testing of reactive and real-time systems.

The process of deploying the software system as a component also could be modeled as an appropriate set of events, where timing and effort attributes are of special interest. This provides yet another aspect for the environment modeling as a part of systems engineering, supported by the integration of systems behavior models. Instances of event traces generated from such integrated schemas are known as use cases, and may be of interest by itself, especially if this activity is supported by appropriate visualization and analysis tools.

### *Model checking*

It would not be surprising if rich enough Phoenix schemas could be mapped into adequate Kripke structures and properties of schemas behavior could be specified with temporal logic formulas, similarly as connector protocols could be specified in CSP and model-checked [3][4][41]. The potential of applying model checking power to a reasonable subset of Phoenix schemas is awaiting for an enthusiastic researcher.

### *Use cases and architecture synthesis from examples.*

An architecture model can be tested with tools like Alloy Analyzer [5], to verify, and to validate the system model at the early stages of the design on instances of the traces. But the reverse process may be of interest as well. A representative set of event trace instances, or use cases, may be used to synthesize a whole architecture scheme of which those are representatives. When the level of abstraction is coarse enough this may be feasible, in a way similar to synthesizing Finite Automata from a representative sets of acceptable input strings [49]. D.Harel's work [24] provides a direction based on UML sequence diagrams.

### *Dynamic and evolving architecture*

Some of the presented event grammar patterns, like iterators, may be useful for modeling dynamics of component/connector creation at the run time.

More specific event constructors like (SPAWN A) that inserts in the event trace an event A with the only basic relation

(SPAWN A) PRECEDES A

may be useful to model independent process threads.

The (KILL A) special event may be added to capture dynamic architectures for exception handling and other situations when an event A and all its offsprings (events under IN relation) should be cancelled/destroyed (i.e. deleted from the emerging trace) even before they are completed.

*Meta-architecture, software product lines, domain-specific architectures*

Lex/Yacc behavior example in Sec. 4 gives a flavor of the potential to specify reusable domain-specific architectures. The Phoenix formalism could be extended to the next abstraction level to specify meta-architectures.

## References

- [1] Gregory Abowd, Robert Allen, and David Garlan, Formalizing Style to Understand Descriptions of Software Architecture, *ACM Transactions on Software Engineering and Methodology* 4(4):319-364, October 1995.
- [2] A.Aho, R.Sethi, J.Ullman, *Compilers, Principles, Techniques, and Tools*, 1986, Addison-Wesley
- [3] Robert J. Allen, *A Formal Approach to Software Architecture*, Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, May 1997
- [4] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. In *ACM Transactions on Software Engineering and Methodology*, Vol. 6(3): 213-249, July 1997.
- [5] "Alloy Analyzer 4.1.10" MIT, Accessed May 8, 2009 <http://alloy.mit.edu/community/software>
- [6] M. Auguston, "FORMAN -- A Program Formal Annotation Language", *Proceedings of the 5th Israel Conference on Computer Systems and Software Engineering*, Gerclia, May 1991, IEEE Computer Society Press, 149-154.
- [7] M. Auguston, "A language for debugging automation", *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering, SEKE'94*, Jurmala, Latvia, June 1994, Knowledge Systems Institute, pp. 108-115.
- [8] M. Auguston, "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation", in *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, May 1995.
- [9] M. Auguston, P. Fritzson, "PARFORMAN -- an Assertion Language for Specifying Behavior when Debugging Parallel Applications", *International Journal of Software Engineering and Knowledge Engineering*, vol.6, No 4, 1996, pp. 609-640
- [10] M. Auguston, A. Gates, M. Lujan, "Defining a program Behavior Model for Dynamic Analyzers", *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*, Madrid, Spain, June 1997, pp. 257-26
- [11] M.Auguston, C.Jeffery, S.Underwood, A Framework for Automatic Debugging, in *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, September 23-27, 2002, Edinburgh, UK, IEEE Computer Society Press, pp.217-222.
- [12] Mikhail Auguston, Clinton Jeffery, Scott Underwood, A Monitoring Language for Run Time and Post-Mortem Behavior Analysis and Visualization, in the *Proceedings of 5<sup>th</sup> International Workshop on Algorithmic and Automatic Debugging AADEBUG 2003*, Ghent, Belgium, September 8-10, 2003, pp. 41-54 (also on the CoRR web site at <http://arxiv.org/abs/cs/0310025>).
- [13] Mikhail Auguston, James Bret Michael, Man-Tak Shing, Environment behavior models for scenario generation and testing automation, *International Conference on Software Engineering, Proceedings of the 1st International Workshop on Advances in Model-Based Testing A-MOST'5*, ACM SIGSOFT Software Engineering Notes, v.30 n.4, July 2005, in the ACM Digital Library.
- [14] M.Auguston, B.Michael, M.Shing, Environment Behavior Models for Automation of Testing and Assessment of System Safety, *Information and Software Technology*, Elsevier, Vol. 48, Issue 10, October 2006, pp. 971-980
- [15] J.-P. Banâtre and D. Le Métayer, "Programming by Multiset Transformation," *Comm. ACM*, vol. 36, no. 1, pp. 98-111, Jan. 1993.
- [16] Bass, Len; Paul Clements, Rick Kazman, *Software Architecture In Practice*, Second Edition, Boston: Addison-Wesley, 2003
- [17] Grady Booch, Ivar Jacobson & Jim Rumbaugh (2000) *OMG Unified Modeling Language Specification*, <http://www.omg.org/docs/formal/00-03-01.pdf>
- [18] B. Bruegge, P. Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism", *The Journal of Systems and Software* 3, 1983, pp. 265-276.
- [19] R.H. Campbell and A.N. Habermann, The Specification of Process Synchronization by Path Expressions, *Lecture Notes in Computer Science*, No. 16, Apr. 1974, pp. 89-102. Vol. 14, No. 3, May 1989, pp. 147-150.
- [20] P.Clements, M.Shaw, "The Golden Age of Software Architecture" Revisited, *IEEE Software*, Vol. 26, No 4, 2009, pp. 70-72
- [21] Peter H. Feiler, David P. Gluch, John J. Hudak, *The Architecture Analysis & Design Language (AADL): An Introduction*, Technical Note CMU/SEI-2006-TN-011, [http://www.sei.cmu.edu/publications/documents/06\\_reports/06tn011.html](http://www.sei.cmu.edu/publications/documents/06_reports/06tn011.html) (accessed June 2009)
- [22] FDR2 user manual. <http://www.formal.demon.co.uk/fdr2manual/index.htm>, 1999.
- [23] *Handbook of Graph Grammars and Computing by Graph Transformation*, (edited by G.Rosenberg), World Scientific Publishing Co, 1997
- [24] D.Harel, R.Marely, Come, Let's Play, *Scenario-Based Programming Using LSCs and the Play-Engine*, Springer, 2003
- [25] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [26] Inverardi, P.; Wolf, A.L., Formal specification and analysis of software architectures using the chemical abstract machine model, *IEEE Transactions on Software Engineering*, Vol. 21, Issue 4, April 1995, pp.373 - 386
- [27] Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich, Static Checking of System Behaviors Using Derived Component Assumptions, *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 3, July 2000, pp. 239-272.

- [28] Jackson, Daniel. 2006. *Software Abstractions: Logic, Language, and Analysis*. Cambridge, Massachusetts: The MIT Press.
- [29] Jackson, Daniel, "A Direct Path to Dependable Software: Who could fault an approach that offers greater credibility at reduced cost? ", *Communications of the ACM*, Vol. 52 No. 4, 2009, Pages 78-88, A version of this article with a fuller list of references is available at <http://sdg.csail.mit.edu/publications.html>
- [30] C.Jeffery, M.Auguston, S.Underwood, Towards Fully Automatic Execution Monitoring, in Proceedings of the Monterey Workshop 2002 "Radical Innovations of Software and Systems Engineering in the Future", sponsored by US Army Research Office and NSF, Venice, Italy, October 7-11, 2002, pp.232-243.
- [31] C.Jeffery, M.Auguston, "Some axioms and issues in the UFO dynamic analysis framework", in the Proceedings of Workshop on Dynamic Analysis, ICSE'03, 25th International Conference on Software Engineering, Portland, Oregon, May 3-11, 2003, pp.45-48.
- [32] Kiczales, Gregor; John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin (1997). "Aspect-Oriented Programming". Proceedings of the European Conference on Object-Oriented Programming, vol.1241. pp. 220-242.
- [33] Jung Soo Kim, David Garlan, Analyzing Architectural Styles, January 2007, <http://acme.able.cs.cmu.edu/pubs/uploads/pdf/jss2006.pdf> last accessed May 2009
- [34] Donald E. Knuth. "Literate Programming". *The Computer Journal*, 27(2):97-111, May 1984
- [35] Philippe Kruchten: Architectural Blueprints - the 4+1 View Model of Software Architecture. In: *IEEE Software*. 12 (6) November 1995, pp. 42-5
- [36] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, No. 7, July 1978, pp. 558-565.
- [37] David C Luckham, Lary M. Augustin, John J. Kenney, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, 21(4):336-355, April 1995.
- [38] David C. Luckham, James Vera, An Event-Based Architecture Definition Language, *IEEE Transactions on Software Engineering*, 21(9):717-734, September 1995.
- [39] Paakki, J. Attribute grammar paradigms - A high-level methodology in language implementation, *ACM Computing Surveys* 27, 2 (June 1995), pp. 196-255.
- [40] PAT: Process Analysis Toolkit An Enhanced Simulator, Model Checker and Refinement Checker, <http://www.comp.nus.edu.sg/~pat/>
- [41] Pelliccione, Patrizio; Inverardi, Paola; Muccini, Henry, CHARMY: A Framework for Designing and Verifying Architectural Specifications, *IEEE Transactions on Software Engineering*, Vol. 35, No 3, 2009, pp.325-346
- [42] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, 17:4 (1992), pp. 40-52.
- [43] Bill Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall International Series in Computer Science (580pp), ISBN 0-13-674409-5
- [44] Shaw, M. and Clements, P., A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems, COMPSAC97, 21st Int'l Computer Software and Applications Conference, 1997, pp. 6-13.
- [45] Shaw, M. and Clements, P., The Golden Age of Software Architecture, *IEEE Software*, Vol. 23, No 2, 2006, pp.31-39
- [46] Shaw, M. and Garlan, D. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, New Jersey.
- [47] Joao Pedro Sousa, Bradley Schmerl, Peter Steenkiste and David Garlan, Activity Oriented Computing, In Soraya Kouadri Mostefaoui, Zakaria Maamar and George Giaglis editors, *Advances in Ubiquitous Computing: Future Paradigms and Directions*, IGI Publishing, Hershey, PA, 2008. Book link: <http://www.igi-pub.com/books/details.asp?ID=7314>.
- [48] J.M.Spivey, *The Z Notation: A reference manual*, Prentice Hall International Series in Computer Science, 1989. (2nd ed., 1992)
- [49] B. A. Trakhtenbrot and Ya. M. Barzdin, *Finite automata: behavior and synthesis*, Translated from the Russian by D. Louvish, Amsterdam, North-Holland Pub. Co.; New York, American Elsevier, 1973.
- [50] Wang, Y. and Parnas, D. Simulating the behavior of software modules by trace rewriting, *IEEE Trans. Software Eng.* 20, 10 (Oct. 1994), pp. 750-759.

### Appendix 1.

This is a complete Alloy model for generating the Basic(Simple\_transaction) event traces. Alloy Analyzer can produce and visualize instances of event traces satisfying this model (in this case there is just a single trace). Note that additional constraints are needed to eliminate redundant traces (to suppress PRECEDES for the root events). Removing it yields refined traces satisfying the model, but containing some redundant relations. Increasing the scope of Alloy model (upper number of objects in this instance) also provides examples of traces satisfying the schema's constraints, but containing redundant events.

-- a model for the simple\_transaction schema  
module simple\_transaction

```
abstract sig Event {
    included_in: set Event,
    contains: set Event,
```

```

    followed_by:    set Event
  }

fact { included_in = ~ contains}

one sig trace {
  IN:              Event -> Event,
  PRECEDES:       Event -> Event
}

fact { trace.IN          = ^included_in }
fact { trace.PRECEDES   = ^followed_by }

-- axioms -----

-- mutual exclusion
fact {no trace.PRECEDES & trace.IN}
fact {no trace.PRECEDES & ~(trace.IN)}

-- non-commutativity
fact {no trace.IN & ~(trace.IN)}
fact {no trace.PRECEDES & ~( trace.PRECEDES)}

-- transitivity
fact {trace.PRECEDES = ^(trace.PRECEDES)}
fact {trace.IN = ^(trace.IN)}

-- distributivity
fact {all a, b, c: Event | (a -> b) in trace.IN and
      (b ->c) in trace.PRECEDES =>
      (a ->c) in trace.PRECEDES }

fact {all a, b, c: Event | (a -> b) in trace.PRECEDES and
      (c ->b) in trace.IN =>
      (a ->c) in trace.PRECEDES }

-- Simple_transaction schema's model
one sig TaskA extends Event{}
  { one a: Send | a = contains
    no followed_by
    no trace.PRECEDES.this }

one sig TaskB extends Event{}
  { one a: Receive | a = contains
    no followed_by
    no trace.PRECEDES.this }

one sig Transaction extends Event{}
  { no followed_by
    no trace.PRECEDES.this
    one a: Send, b: Receive | contains = (a+b) and
      ( a -> b) in trace.PRECEDES }

sig Send extends Event{}
  { no contains }

sig Receive extends Event{}
  { no contains }

-- shared constraints
fact {TaskA.contains & Send =
      Transaction.contains & Send}

fact {TaskB.contains & Receive =
      Transaction.contains & Receive}
-----
run {} for exactly 5 Event

```