2013-09-13

Monterey Phoenix, or How to Make Software Architecture Executable

Auguston, Mikhail

Auguston, Mikhail, “Monterey Phoenix, or How to Make Software Architecture Executable”, OOPSLA’09/Onward conference, OOPSLA Companion, October 2009, pp.1031-1038
http://hdl.handle.net/10945/36348

Monterey Phoenix, or How to Make Software Architecture Executable

Mikhail Auguston
Computer Science Department
Naval Postgraduate School
Monterey, CA, 93943, USA
email: 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 up to executable design and implementation models, reuse, composition, visualization, and application of automated tools for consistency checks.

ACM Category D.2.11 Software Architectures

General terms Design, Documentation, Verification

Keywords Software Architecture Description Language, behavior model

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. The following aspects have emerged as characteristic for software architecture descriptions [8][20].

- 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). There should be flexible and expressive composition operators supporting the refinement process.
- 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.
- The software architect needs a number of different views of the software architecture for the various uses and users [20] (including visual representations, like diagrams).

OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA.

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.

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 [3].

The framework supports extracting multiple views of the system from the same architecture description, specification of behavior patterns at high level of abstraction, reuse of architectures, and possible use of 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” [16].

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.

2 Behavior Models

The software development is aimed at the design of a compact description of a set of required behaviors. The source code 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 [17].

Bruegge, Hibbard [10], and Campbell, Habermann [11] have demonstrated the application of path expressions for program monitoring and debugging. Path expressions in [20] have been used (semi-formally) as a part of software architecture description.

A.Hoar's CSP (Communicating Sequential Processes) [14] 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 [2][19].

Rapide [18] 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 [9], and AADL [13].

The concept of behavior model based on events and event traces was introduced in [4][5][6] 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.

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 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 PRECEDES b $\Rightarrow \neg$(a IN b)
Axiom 2) a PRECEDES b $\Rightarrow \neg$(b IN a)
Axiom 3) a IN b $\Rightarrow \neg$(a PRECEDES b)
Axiom 4) a IN b $\Rightarrow \neg$(b PRECEDES a)

Non-commutativity

Axiom 5) a PRECEDES b $\Rightarrow \neg$(b PRECEDES a)
Axiom 6) a IN b $\Rightarrow \neg$(b IN a)

Irreflexivity for PRECEDES and IN follows from non-commutativity.

Transitivity

Axiom 7) (a PRECEDES b) $\wedge$ (b PRECEDES c) $\Rightarrow$ (a PRECEDES c)
Axiom 8) (a IN b) $\wedge$ (b IN c) $\Rightarrow$ (a IN c)

Distributivity

Axiom 9) (a IN b) $\wedge$ (b PRECEDES c) $\Rightarrow$ (a PRECEDES c)
Axiom 10) (a PRECEDES b) $\wedge$ (c IN b) $\Rightarrow$ (a PRECEDES c)

Event trace is always a directed acyclic graph.

2.3 Event Grammar

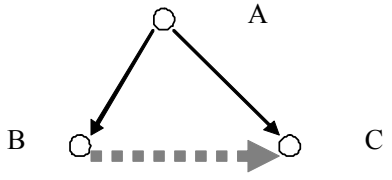
The structure of possible event traces is specified by an *event grammar*. A grammar rule specifies structure for a particular event type (in terms of IN and PRECEDES relations). Event types that do not appear in the left hand part of rules are considered atomic and may be refined later by adding corresponding 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.

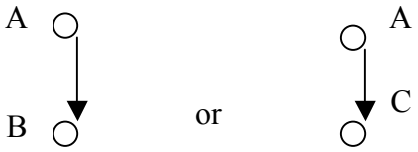
Events may be visualized by small circles, and basic relations - by arrows, like

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 IN a, c IN a, and b PRECEDES c).

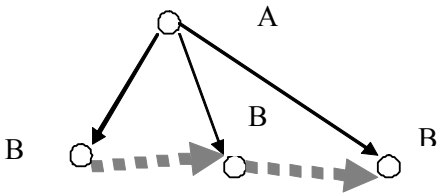
The rule A:: B C; specifies the following event trace.



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



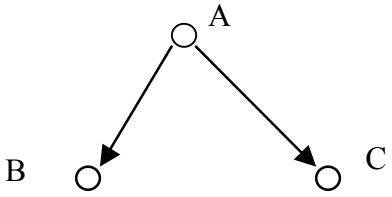
3) A:: (* B *); means an ordered sequence of zero or more events of the type B. Here is an example of an event trace satisfying this pattern:



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.

2.4 Example of an Event Grammar

The Shooting_competiton 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 behaviors of a particular abstract machine are specified as a set of all possible event traces using a *schema*. The concept of the Phoenix schema has been inspired by Z schema [24]. 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 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 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 may contain also 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. 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 **share all** Z ≡ { v: Z | v IN X} = {w: Z | w 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 synchronization mechanism similar to the communication events in CSP [14].

3.1 Schema Interpretation

Similarly as context-free grammars could be used to generate strings, event grammars could be used as production grammars to generate instances of event traces (or graphs). 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 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 Basic(S) defines the semantics of the schema S and could be specified in the form of an abstract machine. 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 Basic(S) can be extracted by running S on the Phoenix abstract machine, i.e. by generating a trace instance.

Traces from Basic(S) can be refined by introducing additional events, event types, and basic relations between them, while maintaining the consistency with original trace's constraints and Axioms. The set of all refined traces for S is called 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 1 renders the only event trace from the Basic(Simple_transaction). There may be other traces consistent with the structure imposed by the schema, for example, a trace from Figure 1 with an additional relation 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 [3][15] is a good candidate for implementing the Phoenix abstract machine.

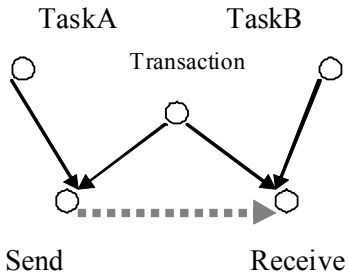


Figure 1. Example of event trace for Simple_transaction schema

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 [2].

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

Yet another semantics of the connector 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;

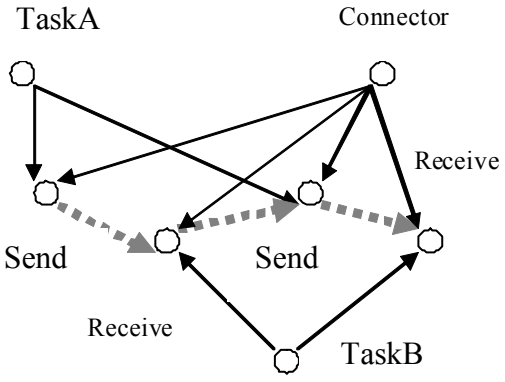


Figure 2. Example of event trace for multiple synchronized transactions

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 [2][23]

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.

4 Composition and Reuse

The compiler's front-end model is inspired by [20].

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 [2]. 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 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's behavior when the Token_processing schema is included.

The following schema provides a rough model of a 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;

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 the Lexer and Parser schemas involved in token request/delivery. In fact, the Get_token event is now refined with the Token_recogniton 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.

As [2] 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 examples above demonstrate the architecture reuse. Tools like Alloy Analyzer [3] can be used for sanity checks to verify whether the merged schema still has trace instances.

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 or coordination). With the progress of refinement the need to introduce more detailed view on data flow starts to appear. As [12] 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*. An event may have a type and other attributes, such as 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.

Example 6. 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.

6 Future work

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.

It becomes acknowledged within the Software Architecture community that there is a relationship between architectural design and quality attributes [22]. 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 [3][15] 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 Monterey Phoenix to Alloy is pretty feasible. Small scope hypothesis behinds Alloy use still makes simple sanity checks on Phoenix schemas meaningful.

Yet another way to check properties of event traces is exemplified by the FORMAN language [4][5] for computations over event traces, which supports generic trace property checks. Event trace generation may be implemented directly based on the event grammars.

Behavior of the environment in which the system is embedded could be specified using event grammars in a similar fashion [6]. Merging schemas of the environment model and the system model could be a meaningful basis for further system analysis.

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 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.

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] 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.

[3] "Alloy Analyzer 4.1.10" MIT, Accessed May 8, 2009 http://alloy.mit.edu/community/software

[4] 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.

[5] 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.

[6] 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

[7] Robert Allen and David Garlan. A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology, July 1997.

[8] Bass, Len; Paul Clements, Rick Kazman, Software Architecture In Practice, Second Edition, Boston: Addison-Wesley, 2003

[9] Grady Booch, Ivar Jacobson & Jim Rumbaugh (2000) OMG Unified Modeling Language Specification, http://www.omg.org/docs/formal/00-03-01.pdf

[10] B. Bruegge, P. Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism", The Journal of Systems and Software 3, 1983, pp. 265-276.

[11] 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.

[12] P.Clements, M.Shaw, "The Golden Age of Software Architecture" Revisited, IEEE Software, Vol. 26, No 4, 2009, pp. 70-72

[13] 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 (accessed June 2009)

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[15] Jackson, Daniel. 2006. *Software Abstractions: Logic, Language, and Analysis*. Cambridge, Massachusetts: The MIT Press.

[16] 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,

[17] Donald E. Knuth. "Literate Programming". The Computer Journal, 27(2):97-111, May 1984

[18] 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.

[19] 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

[20] 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.

[21] 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.

[22] Shaw, M. and Clements, P., The Golden Age of Software Architecture, IEEE Software, Vol. 23, No 2, 2006, pp.31-39

[23] Shaw, M. and Garlan, D. 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs, New Jersey.

[24] J.M.Spivey, The Z Notation: A reference manual, Prentice Hall International Series in Computer Science, 1989. (2nd ed., 1992)