A New Approach to System and Software Architecture Specification Based on Behavior Models

Mikhail Auguston, Clifford Whitcomb, and Kristin Giammarco

Department of Computer Science, Department of Systems Engineering
Naval Postgraduate School
Monterey, CA 93943, USA
Email: maugusto@nps.edu, cawhitco@nps.edu, kmgiamma@nps.edu

Abstract. This paper suggests a new approach to formal system and software 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 an event trace is specified using event grammars and other constraints organized into schemas. Graphical and predicate calculus expressions are used to present the grammar and illustrate some simple examples. The framework provides high level abstractions for analyzing system behavior properties expressed as computations over event traces. The automated tools can support extracting of different views from the model, and verification of behavior properties within a given scope. Advantages of this approach compared with those used by the common simulation tools are as follows.

- Provides a means to write assertions about the system behavior and tools to verify those assertions.
- Performs exhaustive search through all possible scenarios (up to the scope limit). The small scope hypothesis states that most errors can be demonstrated on small examples.
- Provides support for verifiable refinement of the architecture model, up to design and implementation models.
- Allows integration of the architecture models with environment models for defining typical scenarios (use cases) and verifying the system's behavior for those scenarios.

Keywords: system and software architecture models, architecture verification

1. Introduction

One of the major concerns in the design of complex engineered systems from a holistic perspective is the question of the behavior of the system. The development of executable architecture models allows for the study of emergent behaviors through computational modeling and simulation during the earliest stages of conceptual design. Architecture development is done very early in the system design process and is concerned with the high-level structure and properties of the system. Software architecture can be viewed as a level of design and modeling that forms a bridge between requirements and code[1]. System architecture can be seen as an initial level of design and system modeling that forms the fundamental basis for defining elements, including software, hardware, and humans, and their relationships to one another and their operating environment considering stakeholder concerns.. The following aspects that define the characteristics for architecture descriptions [1][7][14][9][13][20] are provided to summarize the key tenets under which this research was conducted.

- Architecture descriptions belong to a high level of abstraction, ignoring many of the implementation details, such as algorithms and data structures.
- Architecture plays a role as a bridge between requirements and implementation.
- System architecture models become the earliest form of the system structure, and are created so that stakeholders and designers can begin to reason about various proposed "to-be" alternatives.
- System architecture models must allow for simulation in order to study behaviors and determine possible emergent properties.
- Architecture specifications should be supportive of the refinement process, and need to be checked carefully at each refinement step (preferably with tools).

- Architecture specifications 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.
- Software and system architects, as well as stakeholders, need a number of different views of the architecture for the various uses and users (including visual representations, like diagrams).
- Formal methods enable architects to decompose and express architecture schemas that are unambiguous, abstract, and independent of tools.
- Architecture tools are what Fred Brooks would describe as "accidents" [22], providing the technological means by which architects develop their work products. The "essence" of architecting is tool-agnostic: how can the state of the art and practice of system and software architecting be improved?
- Architecture descriptions should specify fundamental rules in the universal language of logic and be independent of methodology, tool, and programming language, and map those rules into higher level processes, tools, and languages (which may be tailored to need) for the broadest use.

Many commercial tools [21] have been designed for use for architecture development and analysis efforts. Many languages and tools exist for describing and studying system behavior in particular, e.g., [16][17][18][19]). However, each tool uses standard or unique notation or code that contains a finite set of logical constructs available to the architect for describing system behavior. In this paper we suggest a new framework for system and software architecture specification based on behavior models called Monterey Phoenix (MP for short). This framework constitutes an improvement to the set of underlying logical constructs used to model a system's behavior, from the perspective of what fundamental rules are involved in specifying an event trace. A new grammar is presented that can be used to formally define common, reusable statements related to behavior. Our purpose is to demonstrate that behavior models expressed in the MP framework may be used as a basis for architecture description, that structural and some other properties may be extracted from the behavioral specifications, and that the MP framework can be supported by automated tools for the validation and verification of the developed architecture models.

Our approach addresses the following issues in systems architecture modeling.

- The MP approach provides a high level of abstraction for architecture models based on behavior modeling. Architecture models are executable, so that it becomes possible early in the system development phase to do testing and verification at the top level of system design.
- The MP approach provides formalism for specifying a system's environment models, based on the behavior modeling, so that the system architecture can be tested and verified in the interaction with its environment. Models of a system's emerging behaviors can be obtained and analyzed as a result of the interaction between the system and the environment behavior models.
- The MP framework supports the ability to specify constraints on the behavior and the assertions about behavioral properties, which can be used by automated tools for architecture verification and validation.
- The MP framework provides a method for system stepwise refinement from the top level architecture models to the detailed design and implementation models, supported by tools for sanity checks and refinement consistency checks.
- The MP approach provides a new abstract view on the interfaces, composition, coordination, and synchronization for system modeling.
- The MP approach makes it possible to automate verification of certain non-functional requirements, e.g. performance estimates based on statistics extracted from the event traces.
- The MP approach makes it possible to extract different views from the models (different box/arrow diagrams), relevant to existing standards (like SysML, DoDAF, CORES, etc.) and to build corresponding visualization tools that can be customized to the user's needs.

The initial idea for this work was first published in [3] and [4]. These papers contain more examples of architecture reuse and models of system and environment interaction. A complete example of assertion checking for the architecture model in MP using Alloy Analyzer is available in paper [5].

2. Technical Approach

Formal methods [23][24][25] are employed in this research as a technical approach to define a new grammar for behavior models. By using formal methods as the basis for our approach, we increase the chances that unexpected states or behaviors will be uncovered early on during architectural analysis, and before system development, production, and deployment. Graphical and predicate calculus expressions are used to present the grammar and illustrate some simple examples. The main novelty of this work is illustrated in the method for system behavior modeling based on event traces, which provides a high level of abstraction for system architecture and its environment descriptions.

2.1 Behavior Models and the Event Concept

Our approach focuses on the notion of an *event,* which is an abstraction for any detectable action. An event has a beginning, an end, and duration, and usually corresponds to a time interval.

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. Actions performed during program execution are at different levels of granularity, some of them including other actions. This consideration brings an *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). 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. An event trace is always a directed acyclic graph. This concept of behavior modeling is one of the most distinctive novel features of our approach.

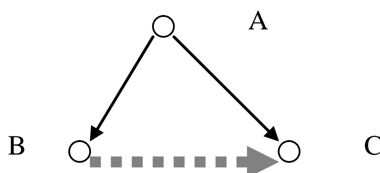
2.2 Event Grammar

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

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

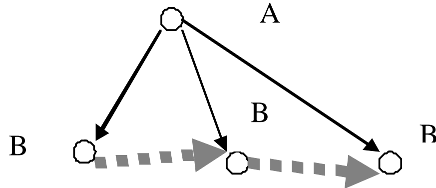
1) A sequence denotes ordering of events under the PRECEDES relation. The rule A:: B C; means that an event a of the type A contains ordered events b and c matching B and C, correspondingly (b IN a, c IN a, and b PRECEDES c). A grammar rule may contain a sequence of more than two events, like A:: B C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

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



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

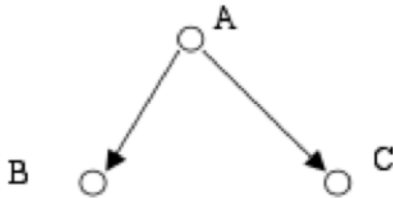
3) A:: (* B *); means an ordered sequence of zero or more events of the type B. Here is an example of an event trace satisfying this pattern:



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 (* … *) and {* … *} event patterns those may be useful for specifying dynamic architectures.

2.3 Schema as a Behavior Specification

The behavior of a system's model are specified as a set of all possible event traces using a *schema*. The concept of the MP schema is inspired by the Z schema [1]. The schema is similar to the fundamental architectural concept of *configuration*, which is a collection of interacting components and connectors, as introduced in [1]. 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*. 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 Simple_transaction specifies the behavior of components involved in a single transaction.

SCHEMA 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 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}

Event sharing in MP plays the role of a synchronization mechanism similar to the communication events in CSP [11].

2.4 Schema Interpretation

Similar to context-free grammars that 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 filtering the generated traces to a set of traces called Basic(S). 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 MP, 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 Java Virtual Machine (JVM), similarly a particular event trace from the Basic(S) can be extracted by running S on the MP abstract machine, i.e. by generating a trace instance.

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 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.

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 using **SHARE ALL** constraints.

The Alloy Analyzer[1] [2][12] is a good candidate for implementing the MP Abstract Machine. The following event trace for Example 1 was obtained by prototyping on the Alloy Analyzer. The complete Alloy model for this schema can be found in the Appendix 1 in the paper [4].

---

[1] Alloy Analyzer is a model building tool that helps humans reason about models and construct a more complete set of assertions, bringing any undocumented or implicit assumptions / unexpected states that the system may enter to the attention of the modeler.

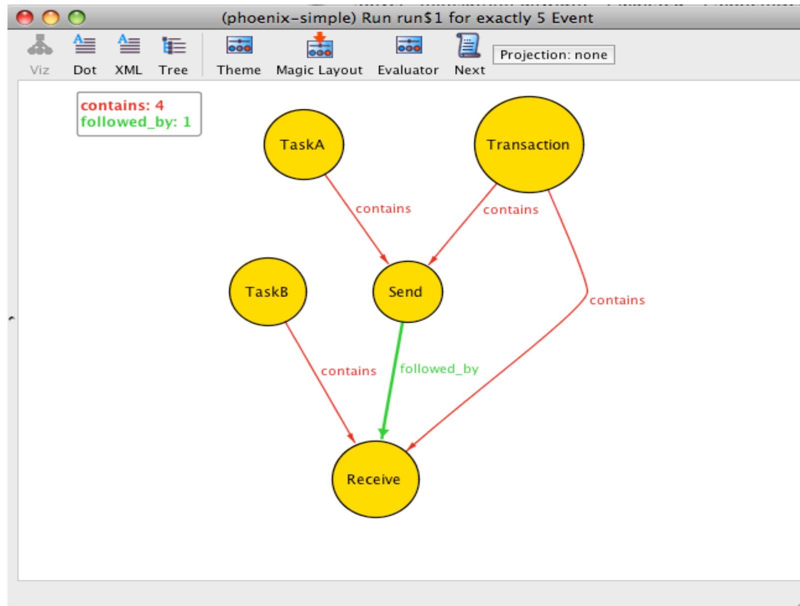


Figure 1. Example of an event trace for Simple_transaction schema obtained on Alloy Analyzer.

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

Yet another semantic 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.

SCHEMA Multiple_synchronized_transactions

ROOT TaskA:: (* Send *);

ROOT TaskB:: (* Receive *);

ROOT Connector:: (* Send Receive *);

TaskA, Connector **SHARE ALL** Send;

TaskB, Connector **SHARE ALL** Receive;

The following event trace has been generated for the AtoB schema by our MP -> Alloy compiler prototype using the Alloy Analyzer model builder. The white boxes represent auxiliary events ("scaffolds" needed to create the resulting graph), and have been included in the graph using the highly customizable Alloy Analyzer's graph rendering tool. The schema specifies a set of event traces, which can be generated in a systematic order. One of traces satisfying the schema is given in Figure 2.

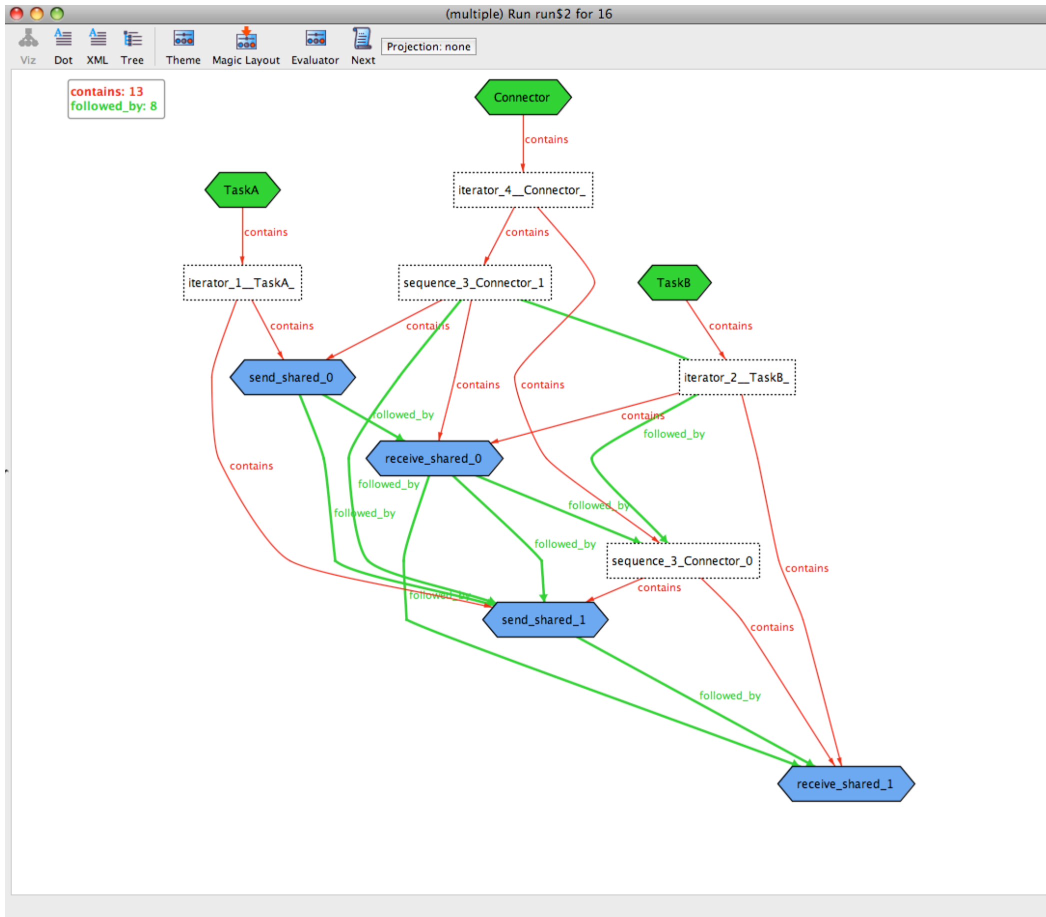


Figure 2. Example of an event trace for multiple synchronized transactions

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.

2.5 Architecture Visualization and Views

Different abstract views can be extracted from the MP 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

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

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 3 (views 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.

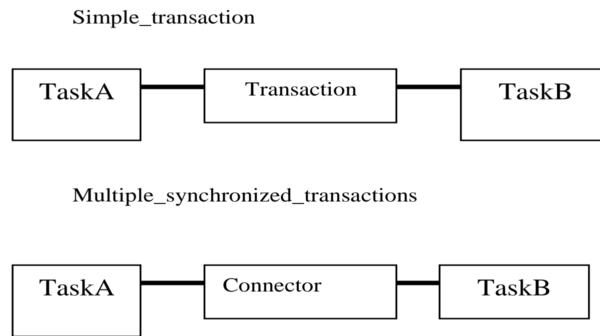


Figure 3. Diagrams extracted from the schemas

Those models seem to be more abstract than corresponding schemas, and 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, such as architecture products or views, currently in use.

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.

2.6 System Behavior Models and Assertions

Both behavior of the system and its environment can be modeled and merged together. This provides for verification and validation of the system's interaction with the environment.

Example 3. An example of two components communicating via an unreliable channel.

```
SCHEMA AtoB

ROOT TaskA: (* A_sends_request_to_B
                  (A_receives_data_from_B  |  A_timeout_waiting_from_B  )    *);

ROOT TaskB: (* (B_working | B_not_working ) *);
 B_working: (*    B_receives_request_from_A  B_sends_data_to_A  *);
 B_not_working: (* request_bounces_back *);

ROOT Connector_A_to_B: (*  A_sends_request_to_B
      (B_receives_request_from_A | [ request_bounces_back ]  A_timeout_waiting_from_B )   *);

        /* A_timeout_waiting_from_B may happen either because Connector_A_to_B just fails or
        because TaskB is not working */

ROOT Connector_B_to_A: (* B_sends_data_to_A
        ( A_receives_data_from_B | A_timeout_waiting_from_B )  *);
_____________________________________________________________________
TaskA, Connector_A_to_B SHARE ALL A_sends_request_to_B;
TaskB, Connector_A_to_B SHARE ALL B_receives_request_from_A, request_bounces_back;
TaskB, Connector_B_to_A SHARE ALL B_sends_data_to_A;
TaskA, Connector_B_to_A SHARE ALL A_receives_data_from_B;
TaskA, (Connector_A_to_B + Connector_B_to_A) SHARE ALL A_timeout_waiting_from_B;

ASSERTION A1: all a:A_sends_request_to_B | some b: B_sends_data_to_A | PRECEDES[a, b];
```

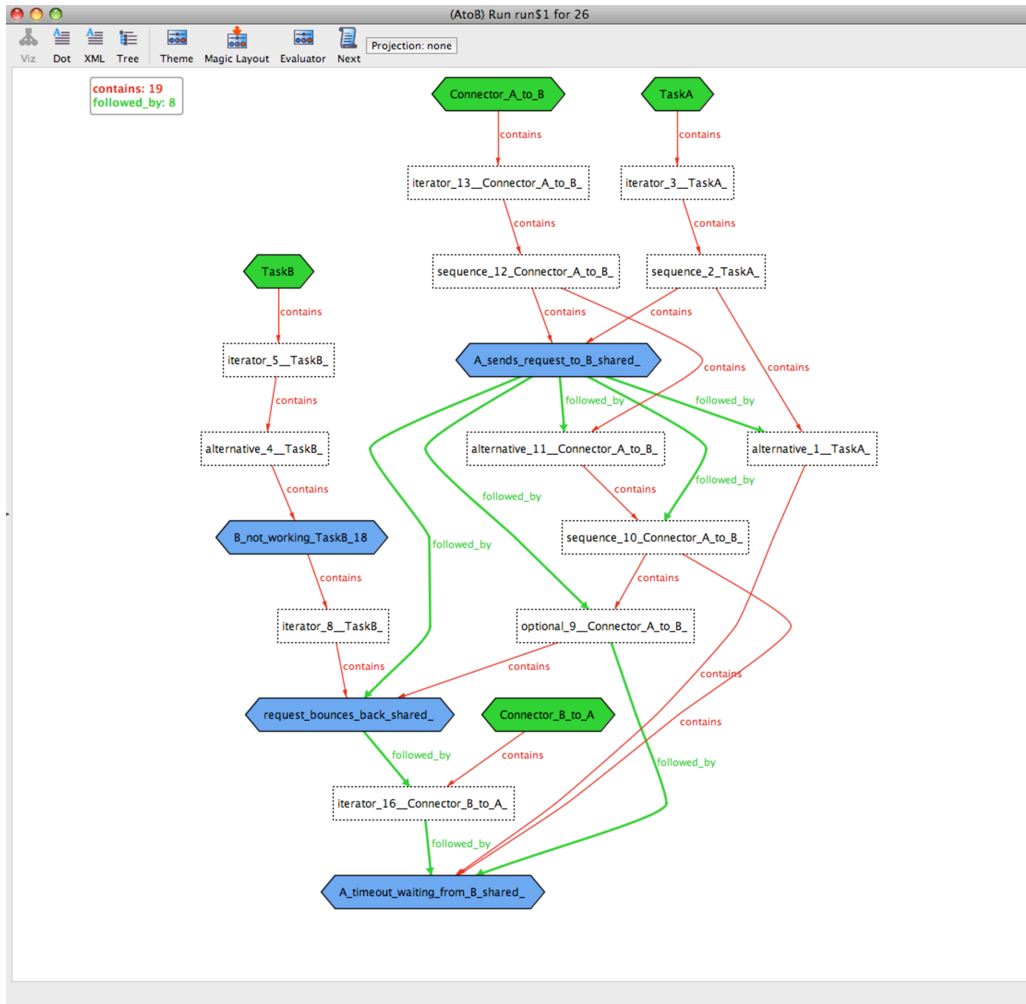


Figure 4. A counter-example for assertion A1 on the schema AtoB.

Figure 4 above demonstrates the result of assertion checking performed with Alloy Analyzer – an instance of event trace violating assertion A1. This counter-example was found within a scope of 23 (the upper limit on the total number of events in the trace) in approximately 35 seconds on iMac workstation with 2.8 GHz processor and 4 GB memory.

3. Conclusions and Future Work.

Architecture modeling touches on the very fundamental issues in system and software design processes and has substantial consequences for the next phases in 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 these will require significant investment in rigorous design and experimentation. The following outlines several prospective directions to pursue.

Schema 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, and can be implemented in the MP framework.

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

Throughput/Latency Estimate.
Given duration and frequency estimates for events within components and connectors it becomes possible to estimate throughput and latency for the whole event trace.

Environment Models and Business Process Models for Systems Engineering.
Behavior of the environment may be modeled by event grammars [6] 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. The model of interaction between the system and its environment may be of special value for testing of reactive and real-time systems. 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 themselves, especially if this activity is supported by appropriate visualization and analysis tools.

Dynamic and Evolving Architectures.
Some of the presented event grammar patterns, like iterators, are useful for modeling dynamics of component/connector creation at run time.

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] "Alloy Analyzer 4.1.10" MIT, Accessed May 8, 2009 http://alloy.mit.edu/community/software
[3] Mikhail Auguston, "Monterey Phoenix, or How to Make Software Architecture Executable", OOPSLA'09/Onward conference, OOPSLA Companion, October 2009, pp.1031-1038
[4] Mikhail Auguston, "Software Architecture Built from Behavior Models", ACM SIGSOFT Software Engineering Notes, Vol. 34, No 5, September 2009
[5] Mikhail Auguston, Clifford Whitcomb, "System Architecture Specification Based on Behavior Models", in the Proceedings of the 15th International Command and Control Research and Technology Symposium (ICCRTS), June 22–24, 2010, in Santa Monica, CA (USA). http://dodccrp.org/html4/events_15.html
[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] Bass, Len; Paul Clements, Rick Kazman, Software Architecture In Practice, Second Edition, Boston: Addison-Wesley, 2003
[8] Boardman, John, and Brian Sauser, *System of Systems – the meaning of "of"*, Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering, Los Angeles, CA, USA - April 2006.
[9] D.Buede, The Engineering Design of Systems, (2nd Edition), Wiley, 2009
[10] D.Garlan, Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events, in "Formal Methods for Software Architecture", Lecture Notes in Computer Science, Vol. 2804, 2003, Springer Verlag, pp.1-24.
[11] C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.

[12] Jackson, Daniel. 2006. Software Abstractions: Logic, Language, and Analysis. Cambridge, Massachusetts: The MIT Press.
[13] M.Maier, E.Rechtin, The Art of Systems Architecting, (3rd Edition), CRC Press, 2009
[14] 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.
[15] J.M.Spivey, The Z Notation: A reference manual, Prentice Hall International Series in Computer Science, 1989. (2nd ed., 1992)
[16] Lee W. Wagenhals, Insub Shin, Daesik Kim, and Alexander H. Levis, “C4ISR Architectures: II. A Structured Analysis Approach for Architecture Design”, Systems Engineering Journal, Vol. 3, No. 4, pp. 248-287
[17] Object Management Group™, “Unified Modeling Language™ (UML®)” version 2.3, May 2010. Retrieved August 20, 2010 from http://www.omg.org/spec/UML/2.3/
[18] Object Management Group™, “Systems Modeling Language (SysML™)” version 1.2, June 2010. Retrieved August 20, 2010 from http://www.omg.org/spec/SysML/1.2/
[19] Long, J., “Relationships between Common Graphical Representations in System Engineering”, Vitech Corporation Technical Paper, 2002. Retrieved August 20, 2010 from http://www.vitechcorp.com/whitepapers/files/200701031634430.CommonGraphicalRepresentations_2002.pdf
[20] Giammarco, K., “Formal Methods for Architecture Model Assessment in Systems Engineering”, Proceedings of the 8th Conference on Systems Engineering Research, March 17-19, 2010, Hoboken, NJ.
[21] INCOSE SE Tools Database, Retrieved on December 1, 2009 from http://www.incose.org/ProductsPubs/products/toolsdatabase.aspx
[22] Brooks, Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering," Computer, Vol. 20, No. 4 (April 1987) pp. 10-19.
[23] Wing, J., “A Specifier’s Introduction to Formal Methods”, IEEE Computer, 23(9), Sept. 1990, pp. 8-24.
[24] Luqi and Goguen J., “Formal Methods: Promises and Problems”, IEEE Software, Vol.14, Jan. 1997, pp. 73-85.
[25] Berry, D. “Formal Methods: The Very Idea, Some Thoughts About Why They Work When They Work”, Proceedings of the 1998 ARO/ONR/NSF/ARPA Monterey Workshop on Engineering Automation for Computer Based Systems, Monterey CA, Oct. 1998, pp. 9-18.