2014-11

Behavior models for software architecture

Auguston, Mikhail

Monterey, California. Naval Postgraduate School

NPS-CS-14-003

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Behavior Models for Software Architecture

by

Mikhail Auguston

November 2014

Approved for public release; distribution is unlimited

Prepared for:
Consortium for Robotics and Unmanned Systems Education and Research (CRUSER)

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE	**2. REPORT TYPE unclassified**	**3. DATES COVERED**
11-06-2014	Technical Report	09-01-2014 -11-01-2014

4. TITLE AND SUBTITLE

Behavior Models for Software Architecture

5a. CONTRACT NUMBER

5b. GRANT NUMBER
RWGY8

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Mikhail Auguston

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NPS Computer Science Department

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-CS-14-003

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Consortium for Robotics and Unmanned Systems Education and Research (CRUSER; http://cruser.nps.edu)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Monterey Phoenix (MP) is an approach to formal software system architecture specification based on behavior models. Architecture modeling focuses not only on the activities and interactions within the system, but also on the interactions between the system and its environment, providing an abstraction for interaction specification. The behavior of the system is defined as a set of events (event trace) with two basic relations: precedence and inclusion. The structure of possible event traces is specified using event grammars and other constraints organized into schemas. The separation of the interaction description from the components behavior is an essential MP feature. The schema framework is amenable to stepwise architecture refinement, reuse, composition, visualization, and multiple view extraction. The approach yields a basis for executable architecture specification supporting early testing and verification, systematic use case generation, and performance estimates with automated tools.

15. SUBJECT TERMS
software and system architecture models, executable architecture models, behavior models, architecture description languages

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Mikhail Auguston
Unclassified	Unclassified	Unclassified	UU	36	**19b. TELEPHONE NUMBER** ***(include area code)***

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Ronald A. Route
President

Douglas A. Hensler
Provost

The report entitled *"Behavior Models for Software Architecture"* was prepared for and funded by Consortium for Robotics and Unmanned Systems Education and Research (CRUSER)

Further distribution of all or part of this report is authorized.

This report was prepared by:

Mikhail Auguston

Reviewed by:

Peter Denning, Chairman
Computer Science Department

Released by:

Jeffrey D. Paduan
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Monterey Phoenix (MP) is an approach to formal software system architecture specification based on behavior models. Architecture modeling focuses not only on the activities and interactions within the system, but also on the interactions between the system and its environment, providing an abstraction for interaction specification. The behavior of the system is defined as a set of events (event trace) with two basic relations: precedence and inclusion. The structure of possible event traces is specified using event grammars and other constraints organized into schemas. The separation of the interaction description from the components behavior is an essential MP feature. The schema framework is amenable to stepwise architecture refinement, reuse, composition, visualization, and multiple view extraction. The approach yields a basis for executable architecture specification supporting early testing and verification, systematic use case generation, and performance estimates with automated tools.

THIS PAGE INTENTIONALLY LEFT BLANK

1. INTRODUCTION

The software system development process involves several layers of models, starting with the abstractions capturing the requirements, and proceeding to the detailed design represented by models expressed in some programming language notation, refining the original requirements into a form that is executable on the target computational platform.

In the last decades the concept of architecture has emerged as one such abstraction layer. Architecture plays a role as a bridge between requirements and implementation of a system, and represents a stepwise refinement in the design process with specific objectives and stakeholders. The following principles have emerged as characteristic for software architecture descriptions [Perry, Wolf 1992], [Shaw, Garlan 1996], [Bass et al. 2003].

- An architecture description belongs to a high level of abstraction, ignoring many of the implementation details, such as algorithms and data structures.
- There should be composition operators for the hierarchical design involving components and subsystems.
- 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.
- An architecture of a system should be considered in the context of the environment in which it operates, as suggested in the international standard ISO/IEC 42010 "Systems and Software Engineering – Architecture Description" [ISO 2011]
- The software architect needs a number of different views of the software architecture for various uses and stakeholders [Kruchten 1995], including visual representations, like diagrams.
- Errors in early system design are the most expensive to fix when detected later in the development lifecycle. Software architecture descriptions may be used for the early assessment of design, and for stakeholder communication [Bosch 2000].
- The use of scenarios for evaluating software architectures is recommended as one of the best industrial practices [Abowd et al., 1997], [Dobrica, Niemela 2002]. Scenario-Based Architecture Analysis Method (SAAM) [Bass, Clements, Kazman 2003] is one of the most mature and commonly used architecture analysis methods.

A conclusion in [Rozanski and Woods 2012] states: "Every system has an architecture, whether or not it is documented and understood."

Software architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions that are necessary to satisfy the requirements and serve as a basis for the design [Perry, Wolf 1992]. The practice of architecture description has converged on the concepts of architectural elements, such as component, connector, and the relationships between them.

[Oreizy et al. 1998], [Taylor et al. 2010] have emphasized the role of connectors in the architecture description as first-class entities. From this point of view a connector becomes yet another behavior interacting with the behaviors of components.

Software design starts with finding an algorithm and mapping it on the appropriate computational platform. An algorithm commonly is specified as a behavior applying a step-by-step procedure to solve the problem at hand. The design process usually proceeds from the high level behavior and involves several layers of refinement. This rationale is behinds the use of pseudo-code for software design.

One of the central tenets in architecture is the principle of reuse. This requires common abstractions on which the reuse framework can be based. For both system and software architectures one of the basic commonalities is the concept of behavior. When designing a system the main concern is to ensure the proper behavior in the context of environment in which the system is supposed to operate.

These considerations imply the importance of behavior models and executable architecture models, and the need to test and verify the system architecture early in the design phase.

We suggest a framework for software architecture modeling called Monterey Phoenix (or MP). Behavior modeling is at the core of this approach. In traditional architecture models the main elements are components (representing the functionality), and connectors (representing the information flow between components). In MP the main concepts are activities and coordination between activities, based on the following principles.

- A view of the architecture as a high level description of possible system behaviors, emphasizing the behavior of subsystems and interactions between subsystems. The MP behavior model is based on the concept of an *event* as an abstraction of activity.
- Separation of the component behavior description from the interaction description provides for a high level of abstraction and supports the reuse of architectural models. Interactions between activities are modeled using event coordination constructs.
- The environment's behavior is an integral part of the system architecture model. MP provides a uniform method for modeling behaviors of the software, hardware, business processes, and other aspects of the system. This facilitates the role of architecture as a bridge between the system requirements and design.
- The event grammar provides a view of the behavior as a set of activities (event trace) with two basic relations, where the PRECEDES relation captures the dependency abstraction, and the IN relation represents the hierarchical relationship. Since the event trace is a set of events, additional constraints can be specified using set-theoretical operations and predicate logic.
- The MP architecture description is amenable to deriving multiple views, and provides a uniform basis for specifying structural and behavioral aspects of a software system.
- MP supports automated and exhaustive (for a given scope) scenario generation for early system architecture verification. The Small Scope Hypothesis [Jackson 2006] states that most flaws in models could be demonstrated on relatively small counterexamples.
- MP framework can assists in unifying UML activity and sequence diagrams, statechart notations, and the recent Executable UML Alf language [OMG 2010] for behavior specification. MP can be used as an addition to the existing tools and methodologies.

The MP framework is intended for the use of lightweight Formal Methods in software and system architecture design and maintenance. It provides an ecosystem for sanity checking tools, reusable architecture patterns, reusable assertions, queries, and tools for extracting architecture views.

2. BRIEF RELATED WORK SURVEY

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

Literate programming introduced by D.Knuth set the directions for hierarchical refinement of structure mapped into behavior, with the concept of pseudo-code and tools to support the refinement process [Knuth 1984].

[Campbell, Habermann 1974] and [Bruegge, Hibbard 1983] have demonstrated the application of path expressions as appropriate abstraction for program monitoring and debugging. In [Perry, Wolf 1992] path expressions have been used (semi-formally) as a part of software architecture description.

CSP (Communicating Sequential Processes) and other process algebras [Hoare 1985], [Milner 1989], [Roscoe 1997] provided a framework for process behavior modeling and formal reasoning about those models, including the ideas of individual events, composite events, and event sharing. This behavior modeling approach has been applied to software architecture descriptions for connector protocol specification [Allen 1997], [Allen, Garlan 1997], [Pelliccione et al. 2009].

Rapide [Luckham et al. 1995a, 1995b] uses events and partially ordered sets of events (*posets*) to characterize component interaction.

"The backbone of the system model should be a hierarchy of activities, … that capture the functional capabilities of the system - suitably decomposed to a level with which the designer is happy." [Harel 1992]. Statecharts [Harel 1987] is an example of labeled transition system approach to the behavior modeling. It became one of the most common behavior modeling frameworks, integrated in the broader modeling and specification systems UML [Booch et al. 2000], and AADL [Feiler et al. 2006].

Coordination models and languages advocate separation of the interactional and the computational aspects of software components. Configuration and architectural description languages share these principles with coordination languages [Papadopolous, Arbab 1998], [Carriero, Gelernter 1992]. Separation of the component behavior from the coordination between behaviors in MP follows this principle and extends on it.

[Wang, Parnas 1994] proposed to use trace assertions to formalize 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. The approach is based on algebraic specifications and term rewriting.

The Alloy modeling framework [Jackson 2006] has strongly influenced MP through ideas of integration of sets and first order predicate logic within the relational logic framework, inheritance structure, emphasis on lightweight Formal Methods as opposed to the full-scale theorem proving, with the fundamental concept of Small Scope Hypothesis, and the principles of immediate feedback and visualization during model design.

The concept of software behavior models based on event grammars and event traces was also introduced in [Auguston 1991, 1995], [Auguston, Jeffery, Underwood 2002], [Auguston, Michael, Shing 2006] as an approach to software debugging and testing automation. The early draft of Monterey Phoenix has appeared in [Auguston 2009 a, 2009 b]. This paper is a substantial extension of a conference paper [Auguston, Whitcomb 2012].

3. BEHAVIOR MODELS

In a certain sense, the source code of a program is a compact description for 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. The behavior of the system is usually the main concern for the developer, and the presence of unintended behaviors manifests errors in the design. A system is operating in a certain environment, which has its own behavior and interacts with the system. The objective of the MP approach is to provide a framework for specifying behaviors of the system, its parts and its environment, and interactions between them.

3.1 Event concept

An implemented software system usually represents an algorithm, i.e. a step-by-step description of activities tailored towards achieving a certain goal. The MP behavior model is based on the concept of an *event* as an abstraction of activity. The event has a beginning and an end, and may have duration (a time interval during which the action is accomplished).

The behavior of a system is modeled as a set of events with two binary relations defined for them: precedence (PRECEDES) and inclusion (IN) – the *event trace*. One action is required to precede another if there is a dependency between them, e.g. the Send event should precede the Receive event. Events may be nested, when a complex activity contains a set of other activities. Imposing one of these basic relations on a

pair of activities represents an important design decision. Usually system behavior does not require a total ordering of events. Both PRECEDES and IN are partial ordering relations. If two events are not ordered, they may occur concurrently. Appendix 1 provides axioms specifying the properties of basic relations.

3.2 Event grammar

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

A: pattern_list;

where A is an event type name and pattern_list is composed from event patterns. 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.

An instance of an event trace satisfying the grammar rule can be visualized as a directed graph with two types of edges (one for each of the basic relations). Events are visualized as boxes, and basic relations as arrows. Fig. 1 outlines the event patterns for use in the grammar rule's right hand part. Here B, C, D stand for event type names or event patterns.

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 several events, like A: B C D;

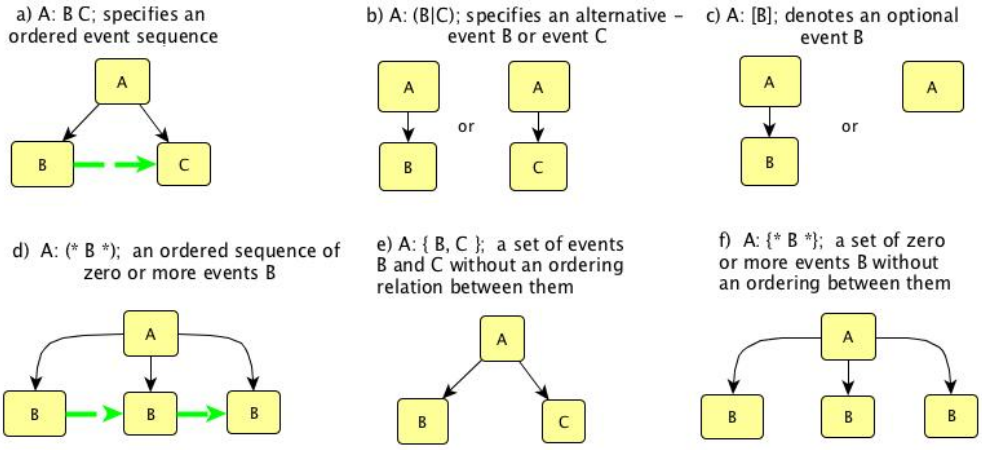

Fig. 1. Event patterns and examples of event traces.

Pattern (+ B +) may be used to denote a sequence of one or more events B, and {+ B +} denotes a set of one or more events B. In all cases it is assumed that iterated event instances are unique. Event patterns may use recursion or iteration to describe repeated behavior patterns.

An event grammar is a graph grammar for directed acyclic graphs of vertices (events) with edges representing relations IN and PRECEDES.

Example 1. An event grammar for car race scenarios.

car_race: {+ driving_a_car +};

driving_a_car: go_straight (* (go_straight | turn_left | turn_right) *) stop;

go_straight: (accelerate | decelerate | cruise);

Similar to context-free grammars, event grammars can be used as production grammars to derive instances of event traces.

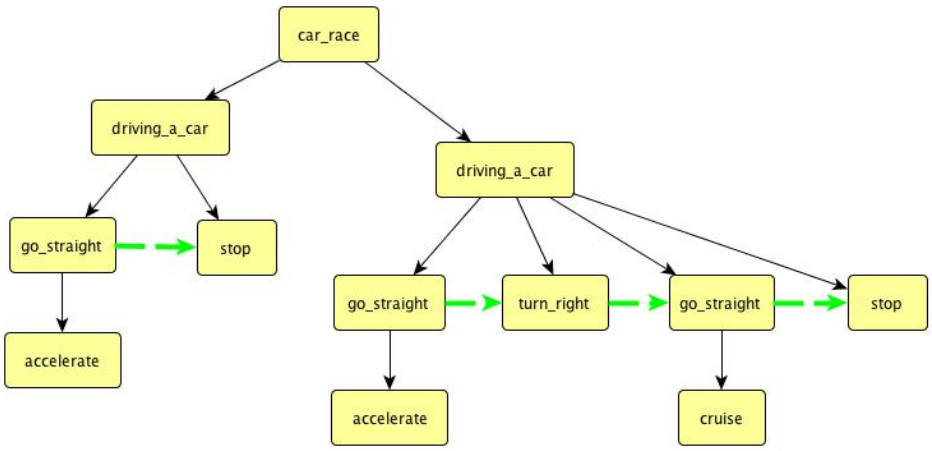

Fig. 2. An instance of event trace derived from the event grammar in Example 1.

4. BEHAVIOR COMPOSITION AND ARCHITECTURE VIEWS

The behavior of a particular system is specified as a set of possible event traces using a schema. The concept of the MP schema is inspired by the Z schema [Spivey 1992]. The purpose is to define the structure of event traces (in terms of IN and PRECEDES relations) using event grammar rules and other constraints. A schema usually contains a collection of events called *roots* representing the behaviors of parts of the system (components and connectors in common architecture descriptions), composition operations specifying interactions between these behaviors, and additional constraints on behaviors.

There is precisely one instance of each root event in a trace. A schema can contain auxiliary grammar rules defining composite event types used in other rules, and may be reused by including it in another schema. 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.

The schema represents instances of behavior (event traces) in the same sense as Java source code represents instances of program execution. Just as a particular execution path can be extracted from a Java program's source code by running it on a JVM, a particular event trace specified by a MP schema can be derived from the event grammar rules by applying behavior composition operations and constraints.

Example 2. Simple pipe/filter architecture pattern.

```
SCHEMA simple_message_flow
ROOT   Task_A:        (* send *);
ROOT   Task_B:        (* receive *);

COORDINATE   $x: send        FROM Task_A,
             $y: receive     FROM Task_B
       DO
             ADD $x PRECEDES $y
       OD;
```

The composition operation **COORDINATE** coordinates behaviors of two root events sending and receiving messages. This trace transformation operation takes two root event traces and produces a modified event trace (merging behaviors of **Task_A** and **Task_B**) by adding the **PRECEDES** relation for the selected **send** and **receive** pairs. Essentially it is a loop performed over the structure of coordinated events, hence the **DO - OD** notation for the loop body.

This synchronized **COORDINATE** composition uses event *selection patterns* to specify subsets of root traces that should be coordinated. The **send** pattern identifies the set of events selected from **Task_A**. Synchronized composition requires that events selected in each coordinated root trace are totally ordered (with respect to the transitive closure of **PRECEDES**), both selected event sets should have the same number of elements (**send** events from the first trace and **receive** events from the second), and the pair coordination follows this ordering (*synchronous coordination*), i.e. first **send** is paired with first **receive**, second with the second, and so on. Labels **$x** and **$y** provide access to the pair of events matching the selection pattern within each iteration. The **ADD** composition completes the behavior adjustment, specifying additional **PRECEDES** relation for each pair of selected events. Behavior specified by this schema is a set of matching event traces for **Task_A** and **Task_B** with the modifications imposed by the composition. If any of selected event sets is not totally ordered, the synchronized coordination operation fails to produce a resulting trace.

The selection pattern may be either an event type name (atomic or composite), or an alternative pattern composed of event type names (Example 3). Sometimes it is desirable to coordinate groups of events. This can be done with the selection pattern **(+ Pattern +)** with optional iteration scope (Example 8).

Fig. 3 (a) gives a sample of event trace satisfying the schema simple_message_flow. It resembles a UML sequence diagram's "swim lanes".

Different views for different stakeholders can be extracted from MP schemas. For example, each root may be visualized as a box. If there is a composition operation specifying an interaction (or coordination) between root behaviors, the boxes are connected by an arrow marked by the interaction type as illustrated in Fig. 3 (b). The root behavior by itself may be visualized with UML Activity Diagram [Booch et al. 2000]. The environment for MP development may have a library of predefined views providing different visualizations for schemas.

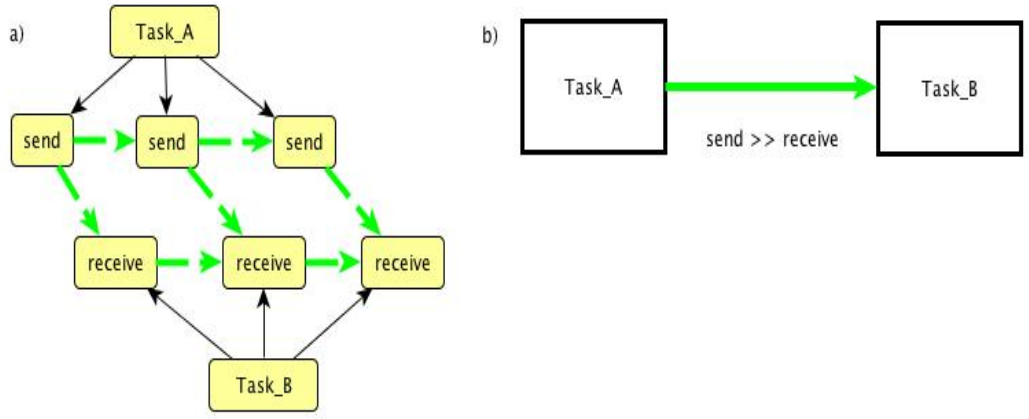

Fig. 3. a) Example of a composed event trace for the simple_message_flow schema. b) An architecture view for the simple_message_flow schema.

Another case of selecting event pairs is called *asynchronous coordination.* In this case the coordination operation should be marked with a symbol **<!>**. For example,

```
COORDINATE <!>      $x: E1   FROM A,
                    $y: E2   FROM B
      DO            ADD $x PRECEDES $y    OD;
```

Matching event sets from **A** and **B** should contain an equal number of selected events **E1** and **E2**, correspondingly. Selected sets of **E1** and **E2** may be totally ordered or not. But now the resulting merged traces will include all permutations of events **E2** from **B** paired with events **E1** from **A**, with the **PRECEDES** relation imposed on each selected pair. This assumes that other constraints, like the partial ordering axioms from Appendix 1, are satisfied. Each permutation yields one potential instance of a resulting trace for the schema deploying this composition. Use of **<!>** may significantly increase the number of composed traces. In order to reduce the exponential explosion, optimizations similar to symmetry reduction in model checking tools may be considered for trace generation.

The composition operation may be considered as an abstract interaction (interface) description for root behaviors. The separation of the interaction description from the components behavior is an essential MP feature. The same component behavior may be reused with different interaction descriptions – a useful composition/reuse aspect (see Sec. 7).

The **COORDINATE** operation supports a "cause-effect" refinement for the behavior of two components and it bears a certain similarity to Aspect-Oriented Programming (AOP) paradigm [Kiczales et al. 1997]. For example, the following AOP behavior could be modeled by MP schema where event coordination implements AOP join point and advice coordination.

Suppose that the main stream of execution contains calls to methods M1 and M2 as join points, and the aspect behavior requires a call to Prolog before, and to Epilog after each method call as an advice. The corresponding MP model may look like the following.

Example 3.

```
ROOT Main:          (* ( M1 | M2) *);
ROOT PreAdvice:     (* Prolog *);
ROOT PostAdvice:    (* Epilog *);

COORDINATE  $jp: ( M1 | M2)  FROM  Main,
            $a1: Prolog      FROM  PreAdvice,
            $a2: Epilog      FROM  PostAdvice
      DO    ADD  $a1 PRECEDES $jp,   $jp PRECEDES $a2   OD;
```

4.1 DATA ITEMS AS BEHAVIORS

Data items in MP are represented by actions (events) that may be performed on that data. This principle follows the Abstract Data Type (ADT) concept introduced in [Liskov, Zilles 1974].

Example 4. Data flow.

```
SCHEMA Data_flow
ROOT Process_1:  (*  work  write  *);
ROOT Process_2:  (* ( read | work ) *);
ROOT File:  (+ write +)  (* read *);

Process_1, File  SHARE ALL      write;
Process_2, File  SHARE ALL      read;
```

The behavior of the **File** requires **write** events to be completed before any **read** events, and there should be at least one **write** event. The **SHARE ALL** composition operation ensures that the schema admits only event traces where corresponding event sharing is implemented.

It is defined as following (here X, Y are root events, Z is an event type, and **IN*** is a transitive closure of **IN**).

X, Y SHARE ALL Z ≡ { v: Z | v IN* X } = { w: Z | w IN* Y }

Event sharing is yet another way of behavior coordination. It is assumed that shared events may appear in the root event at any level of nesting.

The architecture view of this schema in Fig.4 (b) renders root interaction with a line where the shared event name is attached as a label. At the architectural level data items are inputs or outputs of activities and are modeled as operations that may be performed on them. This is a simple and uniform concept. If bringing particular data values in the architecture model is needed, for instance in the assertions describing some constraints on even traces, this can be accomplished by using event attributes.

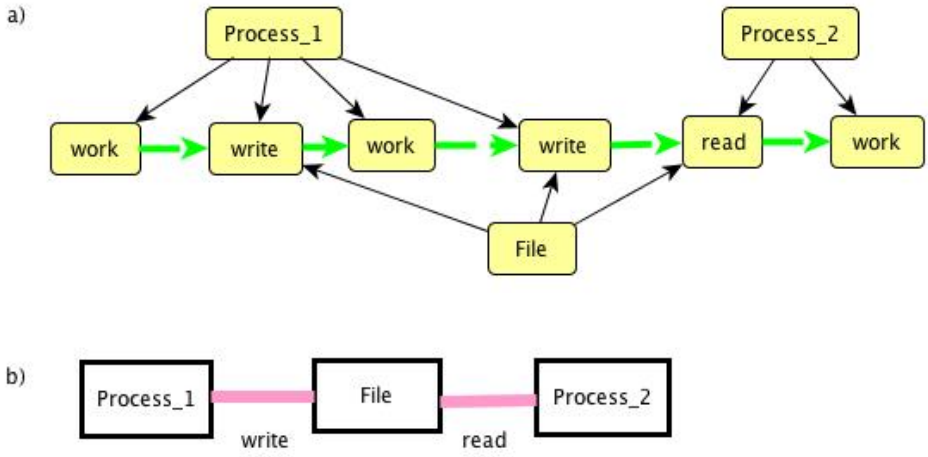

Fig. 4. a) An example of composed event trace for the Data_flow schema.
b) An architecture view for the Data_flow schema.

4.2 BEHAVIOR CONSTRAINTS

A set of behaviors (event traces) is defined by the root event rules, composition operations, and some additional constraints.

Example 5. Stack behavior.
SCHEMA Stack
ROOT Stack_operation: (* (push | pop) *);

ENSURE FOREACH $x: pop FROM Stack_operation
(Number_of (pop) before ($x) < Number_of (push) before ($x));

This schema specifies the behavior of a stack in terms of stack primitive operations. Let **IN*** denote the transitive closure of the **IN** relation (similarly, **PRECEDES*** is a transitive closure for **PRECEDES**). The **ENSURE** Boolean expression provides a condition that each acceptable trace should satisfy. The domain of the universal quantifier is the set of all events **e**, such that **(e IN* Stack_operation).** The function **Number_of (pop) before ($x)** yields the number of **pop** events **e** such that **(e PRECEDES* $x)**. The set of event traces specified by this schema contains only traces that satisfy the constraint. This example presents a filtering operation as yet another kind of behavior composition, and demonstrates an example of combining imperative (event grammar) and declarative (Boolean expressions) features for behavior specification.

4.3 COMPONENTS AND CONNECTORS

Connectors and components, which are the core elements in traditional architecture descriptions, can be uniformly modeled in MP as behaviors. The idea that connectors should be elevated to first-class-citizen status on par with components is often discussed in the literature, for example, in [Taylor et al. 2010].

Suppose that the communication between components is implemented via a buffer of size ***max_buffer_size***, and not necessarily all sent messages are consumed, i.e. some of them could stay in the buffer indefinitely. Each message may be consumed no more than once, and the order of receiving does not necessarily correspond to the order of sending. The root **Buffered_channel** simulates the behavior of a connector between **Task_A** and **Task_B**.

Example 6.

```
SCHEMA Buffered_transaction
ROOT Task_A:   (* Send *);
ROOT Task_B:   (* Receive *);
ROOT Buffered_channel:        {* (Send [ Receive ] ) *}  (Overflow | Normal);
        -- ordering between Send and possible Receive is specified here
Task_A, Buffered_channel      SHARE ALL      Send;
Task_B, Buffered_channel      SHARE ALL      Receive;

ENSURE                  FOREACH $x: Receive FROM Buffered_channel
    ( Number_of (Send) before ($x) - Number_of (Receive) before ($x) ) <= max_buffer_size;
ENSURE                  FOREACH $x: Overflow FROM Buffered_channel
    ( Number_of (Send) before ($x) - Number_of (Receive) before ($x) ) > max_buffer_size;
ENSURE                  FOREACH $x: Normal FROM Buffered_channel
    ( Number_of (Send) before ($x) - Number_of (Receive) before ($x) ) <= max_buffer_size;
```

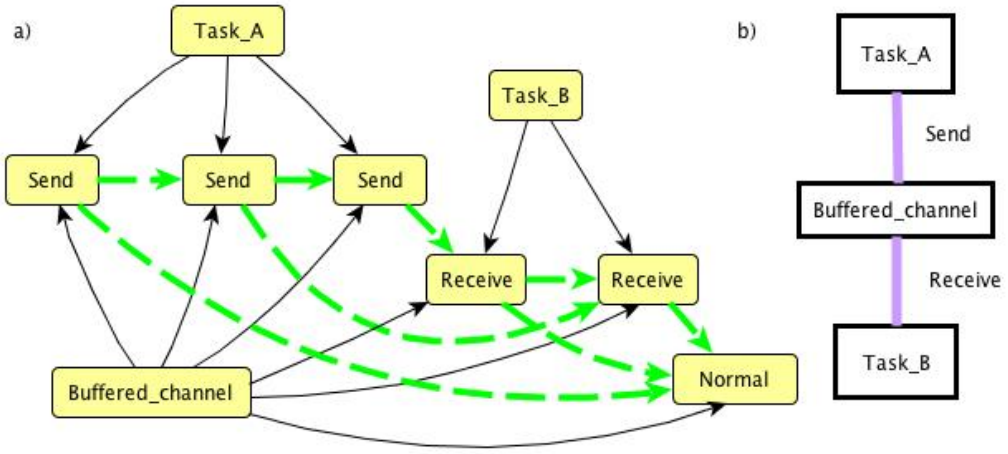

Fig. 5. a) An example of event trace without overflow for the Buffered_transaction schema with *max_buffer_size* = 3.
b) An architecture view for the Buffered_transaction schema

The view of both components and connectors as coordinated behaviors provides flexibility to define hierarchical protocols for interactions between parts of the system.

5. ENVIRONMENT'S BEHAVIOR

The following example demonstrates how to integrate the behavior model of an environment with the behavior model of a system. The **ATM_withdrawal** schema specifies a set of possible interactions between the **Customer**, **ATM_system**, and **Data_Base**.

Example 7. Withdraw money from ATM.

```
SCHEMA ATM_withdrawal
ROOT Customer:              (* insert_card
                         ( (  identification_succeeds
                              request_withdrawal
                             ( get_money | not_sufficient_funds ) )  |
                           identification_fails                       )    *);

ROOT ATM_system: (*  read_card        validate_id
                     ( id_successful  check_balance
                              (  ( sufficient_balance   dispense_money) |
                                 unsufficient_balance  )                  |
                      id_failed                                           )     *);

ROOT Data_Base:     (* ( validate_id | check_balance ) *);

Data_Base,  ATM_system SHARE ALL validate_id, check_balance ;

COORDINATE   $x: insert_card  FROM Customer,
             $y: read_card    FROM ATM_system
      DO     ADD $x PRECEDES $y    OD;
COORDINATE   $x: request_withdrawal          FROM Customer,
             $y: check_balance       FROM ATM_system
      DO     ADD $x PRECEDES $y    OD;
COORDINATE   $x: identification_succeeds  FROM Customer,
             $y: id_successful            FROM ATM_system
      DO     ADD $y PRECEDES $x    OD;
COORDINATE   $x: get_money            FROM Customer,
             $y: dispense_money       FROM ATM_system
      DO     ADD $y PRECEDES $x    OD;
COORDINATE   $x: not_sufficient_funds          FROM Customer,
             $y: unsufficient_balance          FROM ATM_system
      DO     ADD $y PRECEDES $x    OD;
COORDINATE   $x: identification_fails  FROM Customer,
             $y: id_failed             FROM ATM_system
      DO     ADD $y PRECEDES $x    OD;
```

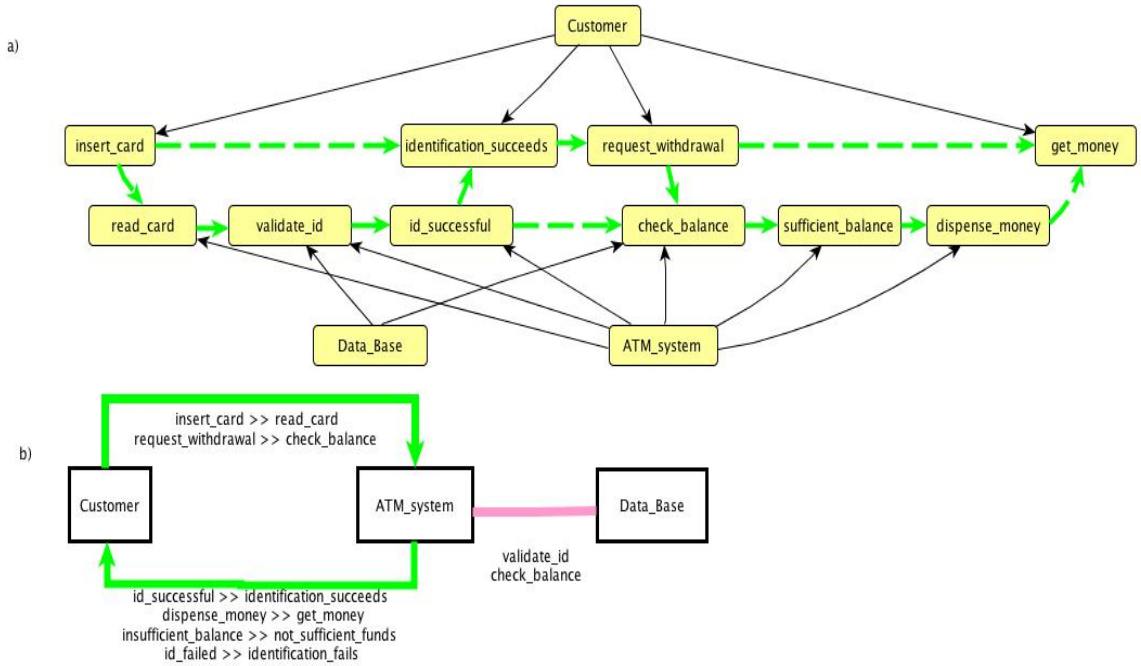

Fig. 6. a) An example of event trace for the ATM_withdrawal schema.
b) An architecture view for the ATM_withdrawal schema.

If the view of the whole system's behavior emphasizing the interaction between the parts (components) can be visualized as in Fig. 6, (b), then the view of the root's standalone behavior can be rendered as a UML Activity Diagram. Since event aggregate patterns (iterations, alternatives, sets) in MP are well structured, it is possible to use Nassi–Shneiderman diagrams [Nassi, Shneiderman 1973] as yet another kind of view.

An event trace generated from the schema can be considered as a use case example. The event trace on Fig. 6, (a) can be viewed also as an analog of UML sequence diagram's "swim lanes" for **Customer** and **ATM_system** interactions. MP models can be integrated into the standard frameworks, like UML, SysML, DoDAF, providing the level of abstraction convenient for architecture models, where MP focuses on the interaction aspects.

The concept of environment in an architecture model includes behavior of other systems, hardware, business processes, and any other behaviors, which are not part of the system under consideration, but may interact with it. In particular, this approach may be of use for analyzing emergent behaviors of System of Systems, when the architecture model of SoS is composed from the models of its components.

6. MULTI-TIER ARCHITECTURE MODELING

Choices of platform, operating system, middleware, database, and such are major architectural choices [Kruchten 2001]. The following example outlines an approach to multilayer architecture modeling. An event in the **Top_layer** deploys several events in **Bottom_layer**. This may involve 1..n multiplicity for event coordination.

Example 8.

SCHEMA two-tier-architecture
ROOT Top_layer: (* top_event anything_else *);
ROOT Bottom_layer: (* bottom_event anything_else *);

```
COORDINATE   $x: top_event              FROM Top_layer,
             $y: (+ bottom_event +)     FROM Bottom_layer
      DO     ADD $y IN $x    OD;
```

The coordination is based on **IN** relation. It models the case when one **top_event** contains one or more **bottom_event**. To limit the number of **bottom_event** repetitions, iteration multiplicity can be used, like:

```
$y: (* <1..3>    bottom_event *)
```

7. COMPONENT REUSE WITH THE JOIN OPERATION

The following compiler's front-end architecture model is inspired by [Perry, Wolf 1992], [Shaw, Garlan 1996] and the unforgettable picture of compiler architecture from the "Dragon Book" [Aho, Sethi, Ullman 1986](page 13). The following examples demonstrate component reuse with different interaction patterns and emphasize the advantages of separation between the specification of component behavior and the specification of interactions between components.

7.1 COMPILER FRONT END IN BATCH PROCESSING MODE

The **Lexer** schema models the behavior of a typical LEX machine.

```
SCHEMA Lexer
ROOT Text_Input:      (* ( String | Unget_char ) *);
      String:         (+ Get_char +);

ROOT Token_processing:      (* Token_recognition *);
      Token_recognition:    {+ RegExpr_Match +}
                            (+ Unget_char +)  Fire_rule;
      RegExpr_Match:              (+ Get_char +);
      Fire_rule:            [ Put_token ];

Text_Input, Token_processing  SHARE ALL     Unget_char;

COORDINATE   $t: Token_recognition   FROM   Token_processing,
             $s: String              FROM   Text_Input
      DO
             COORDINATE <!>   $r: RegExpr_Match    FROM        $t
                    DO     $r, $s   SHARE ALL   Get_char   OD
      OD;
```

The **Token_recognition** event defines Lexer's behavior according to the semantics when a regular expression in each LEX rule is applied independently, and hence no ordering on **RegExpr_Match** events is imposed. Each **RegExpr_Match** performs one or more **Get_char** until all finite automata involved in the token recognition enter the Error state. Then the winner is selected and look-ahead characters beyond the recognized lexeme are returned into the input stream by **Unget_char**. The **String** event contains all **Get_char** events involved in this cycle of recognizing a token, including characters returned back into the input stream. Some recognized tokens, like spaces or comments, don't trigger **Put_token.**

The second **COORDINATE** composition operation provides for a traversal of each instance of **RegExpr_Match** within the given **Token_recognition** (which are unordered, hence the use of <!>). This ensures that all instances of **RegExpr_Match** share all **Get_char** with the **String** coordinated with the **Token_recognition**.

The following **ENSURE** constraint requires that at least one input character will be consumed by **Token_recognition**.

```
ENSURE  FOREACH $t: Token_recognition          FROM Token_processing
                ( Number_of(Get_char) in ($t) > Number_of(Unget_char) in ($t) );
```

The join operation for schemas looks like:

```
SCHEMA A
INCLUDE B;
NEW B;
Roots for A
Additional constraints and composition operations involving roots from both A and B
```

The **INCLUDE** brings schema B into scope, and **NEW B** creates a new instance of schema B. The resulting schema A contains roots defined in A and roots defined in B, merges within its scope constraints and composition operations defined in B, and may have additional constraints and composition operations involving all roots.

Appendix 1 contains **Base** schema specifying properties for the basic relations **IN*** and **PRECEDES***. It is assumed that any MP schema joins **Base**. This operation on schemas is inspired by the Z schema expressions concept [Spivey 1992]. A typical use of such schema composition may be for assembling the architecture of a System-of-Systems from the architectures of its constituent systems.

The following schema provides a model for bottom-up parsing and reuses stack behavior (represented by **push** and **pop** events) defined in Example 5.

```
SCHEMA Parser
INCLUDE  Stack;
NEW Stack;

ROOT Parsing:  push              -- push the start symbol on the stack
               (*       Get_token (* Reduce *) Shift      *) [ Syntax_error ];
        Shift:  push ;
        Reduce:         (+ pop +)  push  Put_node;

ROOT Parse_Tree: (* Put_node *);
        --Put_node event represents the construction of a parse tree.

Parsing, Stack          SHARE ALL       pop, push;
Parsing, Parse_Tree     SHARE ALL       Put_node;
```

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

Example 9.

```
SCHEMA Batch_processing
INCLUDE Lexer, Parser;
NEW Lexer;
NEW Parser;
ROOT Batch:     Produce_tokens  Consume_tokens;
        Produce_tokens:         (*  Put_token  *);
        Consume_tokens:         (*  Get_token  *);

Batch, Lexer    SHARE ALL       Put_token;
Batch, Parser   SHARE ALL       Get_token;

ENSURE          Number_of(Put_token) in (Batch)  >=  Number_of(Get_token) in (Batch);
```

The ordering of **Produce_tokens** and **Consume_tokens** events ensures that production of the whole set of tokens will precede the consumption. The following diagram represents a view of the **Batch_processing** architecture. Black arrows stand for the IN relation, wider lines represent abstract interaction defined by the composition operation SHARE ALL, hexagons represent schemas, and rectangles represent root events.

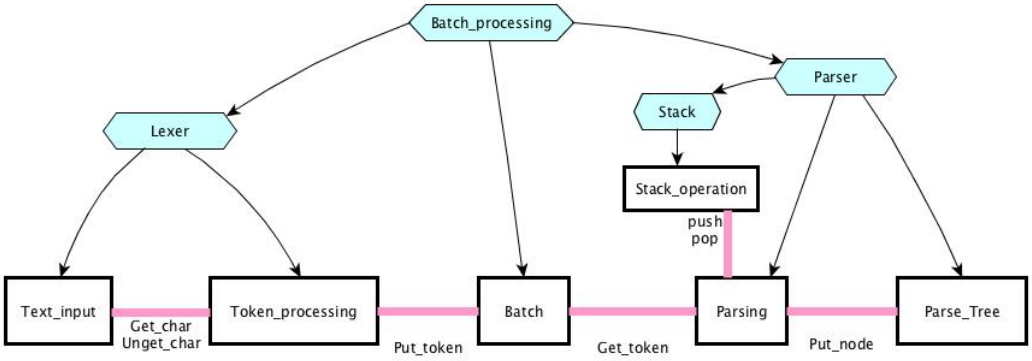

Fig. 7. An architecture view on the Compiler's front end in batch mode.

7.2 COMPILER'S FRONT END IN INCREMENTAL MODE

Yet another possible interaction is a case in which **Parser** requests the next token and triggers an event inside **Lexer**, delivering a token (the usual LEX/YACC operation pattern). The schema **Incremental_processing** represents such operation mode. The **IN** relation imposed between **Put_token** and **Get_token** events reflects the dependency or synchronization between **Lexer** and **Parser** behaviors involved in the token request/delivery. In fact, the **Get_token** event is now refined with the **Put_token** event. **Incremental_processing** schema reuses **Lexer** and **Parser** schemas, but defines different interaction pattern.

Example 10.

```
SCHEMA Incremental_processing
INCLUDE Lexer, Parser;
NEW Lexer;
NEW Parser;

COORDINATE   $x: Token_recognition  FROM Token_processing,
             $y: Get_token          FROM Parsing
      DO     ADD $x IN $y    OD;
```

The merged architecture defines a set of event traces with structure inherited from **Lexer** and **Parser**, into **Incremental_processing** schema with the additional constraints for sharing the token processing events. The following diagram represents a view of the **Incremental_processing** architecture.

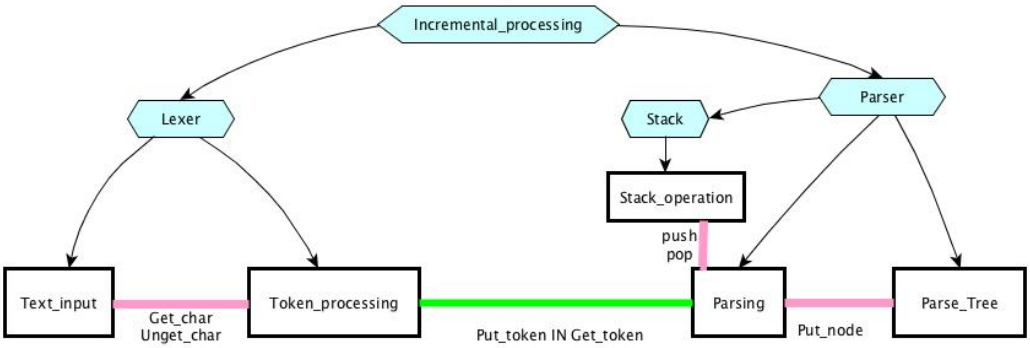

Fig. 8. An architecture view on the Compiler's front end in incremental mode.

8. SCHEMA REUSE WITH THE MAP COMPOSITION OPERATION

Reuse of a behavior defined by a schema may require mapping of some activities in one schema onto activities in another, thus defining a composite behavior. The **MAP** operation establishes event aliases (event mapping) between two instances of behaviors. Example 11 provides an architecture model for a user's login in order to receive access to a system's services. This solution is reused as another architecture's component in Example 12.

Example 11
The User requests access to a System's services by providing his general Id. The System requests that specific credentials be provided. If the supplied credentials are valid, the System authorizes the User to access the services, otherwise the System notifies the User that the received credentials are invalid and the User may re-attempt access up to two more times. At any time the User may decide to abandon the access request.

```
SCHEMA          Authentication_Scenario
ROOT User:      request_access
                (* creds_invalid          request_access  *)
                ( creds_valid     (run_services | abandon_access_request)                |
                  creds_invalid   (attempt_exhausted| abandon_access_request)           );
        request_access:                 provide_general_ID      provide_unique_ID;
```

The alternative for **abandon_access_request** always includes an interaction with the **System**. Absence of the interaction triggers **long_wait_for_User** in the **System**.

```
ROOT System:    request_unique_ID
                [ creds_invalid  request_unique_ID
                        [ creds_invalid  request_unique_ID
                                [ creds_invalid  attempt_exhausted
                                        invalid_creds_notice   cancel_access_request] ] ]
                [ ( creds_valid  (   authorize_access  run_services                    |
                                     long_wait_for_User  cancel_access_request )       |
                    creds_invalid  long_wait_for_User  cancel_access_request           )
                ] ;

User, System SHARE ALL          creds_valid,            creds_invalid,
                        attempt_exhausted,      run_services;

COORDINATE      $x: provide_general_ID          FROM User,
                $y: request_unique_ID   FROM System
        DO      ADD $x PRECEDES $y      OD;

COORDINATE      $x: request_unique_ID   FROM System,
                $y: provide_unique_ID   FROM User
        DO      ADD $x PRECEDES $y      OD;
```

Example 12
The task is to reuse the behavior from the **Authentication_Scenario** schema. The **INCLUDE** statement brings **Authentication_Scenario** schema into the context of **Processing** schema. This means that all constraints specified in the **Authentication_Scenario** also will be included.

```
SCHEMA Processing
INCLUDE Authentication_Scenario;
ROOT Customer:                  (*  work_session        *);
```

```
        work_session:  (+ log_in +) ( login_succeeds  work | login_fails  );
        work: (*  ( read | write )  *);

ROOT Login_service:     (* credentials_check *);
        credentials_check:      ( authorize_access | refuse_access );

ROOT DataBase:                  (*  ( read | write )  *);

Customer, DataBase      SHARE ALL       read,    write;

COORDINATE    $a: authorize_access      FROM   Login_service,
              $b: login_succeeds        FROM   Customer
        DO    ADD $a PRECEDES $b        OD;

COORDINATE    $a: refuse_access         FROM   Login_service,
              $b: login_fails           FROM   Customer
        DO    ADD $a PRECEDES $b        OD;
```

The following **COORDINATE** specifies the reuse of the **Authentication_Scenario** schema and imposes a coordination between each (synchronized) pair of events **work_session** and **credentials_check** from **Processing** schema and a new instance of the **Authentication_Scenario** behavior. Synchronized **COORDINATE** composition requires the number of **work_session** and **credentials_check** events in the trace to be the same, and each of them has to be totally ordered within the corresponding root.

Since **NEW Authentication_Scenario** appears in the **COORDINATE** loop body, a new instance of a complete schema's **Authentication_Scenario** trace is created at each **COORDINATE** iteration to match events selected for coordination. The nested **COORDINATE** operation follows the nesting of coordinated source and target events. The **MAP** is yet another event composition performed within the **COORDINATE** operation. Each **A AS B** pair establishes A and B as aliases for the same event instance.

```
COORDINATE    $w:  work_session      FROM   Customer,
              $ch: credentials_check FROM   Login_service,
        DO
              $a:     NEW    Authentication_Scenario;
              MAP     $w             AS     User            FROM   $a,
                      $ch            AS     System          FROM   $a,
                      work FROM $w    AS    run_services    FROM          $a,
                      authorize_access FROM Login_service
                                     AS     authorize_access FROM        $a,
                      refuse_access  FROM Login_service
                                     AS     cancel_access_request FROM   $a;
              COORDINATE  $log:      log_in          FROM   $w,
                          $req:      request_access  FROM   (User   FROM   $a)
                      DO
                          MAP        $log    AS      $req
                      OD
        OD;
```

Fig. 9 gives an example of event trace generated from the **Processing** schema. Only IN and PRECEDES relations directly specified in the grammar rules or in composition operations are shown. For instance, the PRECEDES* relation between **provide_unique_ID** and corresponding **creds_valid** or **creds_invalid** is implied by Axiom 9 (Appendix 1).

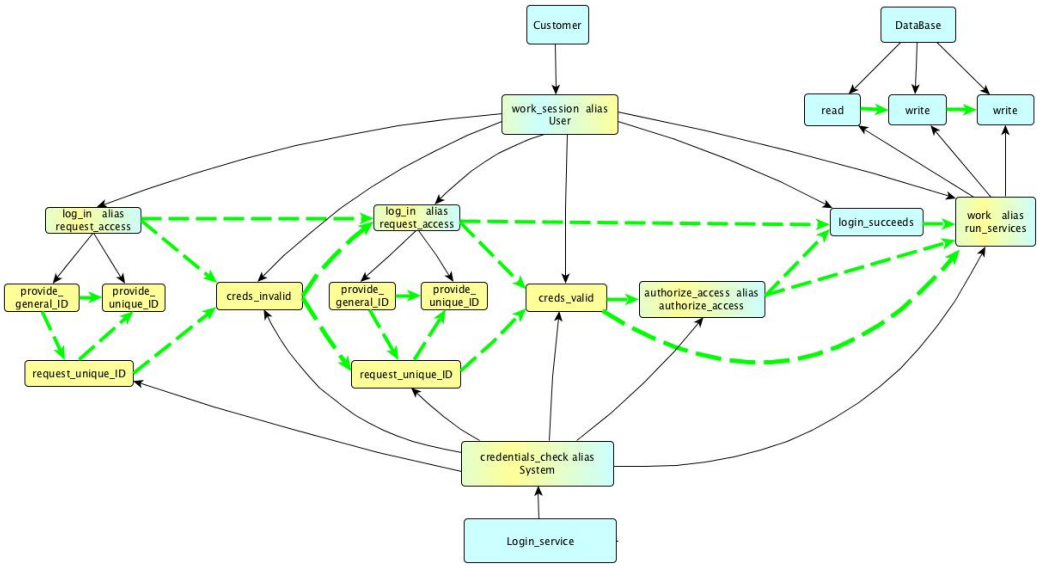

Fig. 9. Example of event trace when Customer gets access and runs services after one unsuccessful login attempt.

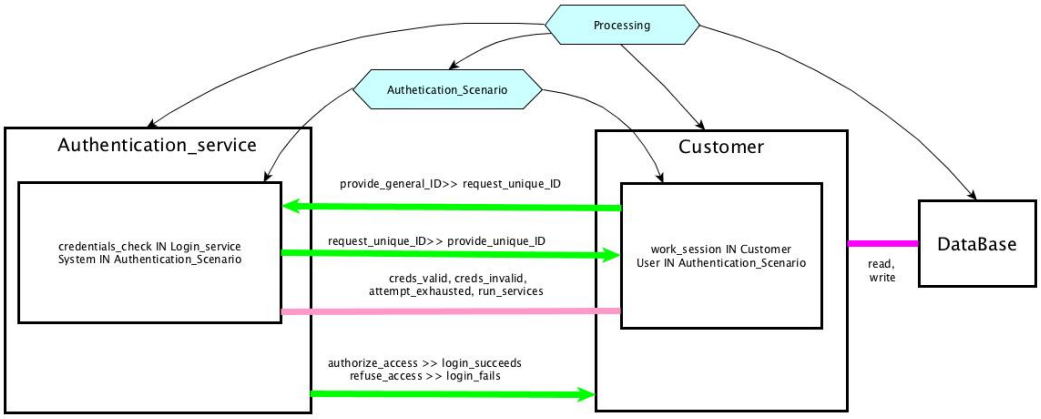

Fig. 10. Architecture view for the Processing schema with reused components from Authentication_Scenario.

MAP AS can be considered a generalization of the **SHARE ALL** composition. For atomic events it is equivalent to **SHARE ALL**. For example,

A, B SHARE ALL c;
is equivalent to
COORDINATE <!> $c1: c FROM A,
$c2: c FROM B
DO MAP $c1 AS $c2 OD;

For composite events **MAP AS** means merging the behaviors as independent threads, which can be coordinated further if needed. If original event structures are
A: pattern1;
B: pattern2;

then **MAP A AS B** establishes an event **AB** (for which both **A** and **B** are aliases).
AB: { pattern1, pattern2 };

9. EVENT PARTITIONING AND COORDINATION

Event partitioning may be needed to specify activities like competitive resource sharing at a high level of abstraction, without going into the implementation details. The following example demonstrates event partitioning rendered in the **SHARE ALL** composition.

Example 13. Communicating via unreliable channel.

```
SCHEMA AtoB
ROOT TaskA:   (*        A_sends_request_to_B
                        ( A_receives_data_from_B  | A_timeout_waiting_from_B )  *);
-- it is assumed that A is the leading actor

ROOT TaskB:   (* ( B_working | request_bounces_back )  *);
        B_working:      B_receives_request_from_A       B_sends_data_to_A;
-- request_bounces_back event simulates the connector's unsuccessful attempt to connect to B.

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;
```

Operator **|+|** is used in the context of **SHARE ALL** as an exclusive union (a partitioning of events, XOR-like operation) with respect to the sharing of a specific event. The event **A_timeout_waiting_from_B** cannot belong to both **Connector_A_to_B** and **Connector_B_to_A**, although sharing of other events between them may be permitted.

10.REACTIVE EVENT INTEGRATION

Reactive component integration, or implicit invocation [Shaw, Garlan 1996] has been introduced in architecture techniques to capture situations when an activity is triggered by an event arriving from the environment or from another component. The exception handling mechanism in programming languages and the interrupt operator in CSP [Roscoe 1997] provide examples of such control flow.

In MP, the behavior model event stream defined by the event grammar rule may be interrupted at any place by another event that triggers continuation of the event flow using another event pattern. This behavior pattern may be modeled with the WHEN clause. The <| and |> delimiters determine the scope of events, which could be interrupted.

<| event_pattern 'WHEN' when_unit , when_unit , ... |>

The ***when_unit*** contains the triggering event's name followed by the event pattern for continuation.

event_name ==> pattern_list

When_unit can be interrupted by another WHEN event. This option may be computationally expensive for event trace generation and should be used with care, to specify for instance an event that may occur in the

environment at any unpredictable moment.

The meaning of the WHEN event pattern for trace generation can be defined with the concept of **CUT(E)** as a pattern specifying possible initial segments of an event trace for the event pattern **E**, when **E** has been interrupted while generating its event trace.

1) CUT(A) = **[A]** if A is atomic event pattern
2) CUT(A) = **[A: CUT(Body)]** if A is a composite event **A: Body**
3) CUT(A B) = **(CUT(A) | A CUT(B))**
4) CUT((A | B)) = **(CUT(A) | CUT(B))**
5) CUT((* <n..m> E *)) = **(* <n..m-1> E *) CUT(E)**
6) CUT({A, B}) = **{ CUT(A), CUT(B) }**
7) CUT({* <n..m> E *}) = **{* <n..m> CUT(E) *}**
8) CUT((+ E +)) = **(* E *) CUT(E)**
9) CUT({+ E +}) = **{+ CUT(E) +}**

The CUT() is an abbreviation for an otherwise cumbersome composition of alternative event patterns inserting the triggering event in all possible positions. The WHEN clause event pattern can be defined as:

```
<| P WHEN A1 ==> A2, B1 ==> B2, ... |> =
      (  P                                                                  |
         CUT(P)  (* ( A1 CUT(A2) | B1 CUT(B2) | ... ) *)  ( A1 A2 |  B1 B2 | ... )        )
```

Example 14. A process that launches and terminates another process.
Process_A launches a single instance of Process_B, which proceeds concurrently and may either terminate normally, or be terminated by Process_A as an emergency. In both cases Process_B may be launched later again by Process_A.

```
SCHEMA Invoke_and_terminate
ROOT Process_A: (* Launch_B  do_something [ Abort_B ]  *);

ROOT Process_B: (* (   <| Idle    WHEN Abort_B  ==>  |>                          |
                       Launch_B  <| (* b1 b2 b3 *)  WHEN Abort_B  ==> wrap_up |>          )
                *);
Process_A, Process_B    SHARE ALL       Launch_B,  Abort_B  ;
```

When **Process_B** is **Idle**, there is no response to **Abort_B**. The whole WHEN clause around **Idle** can be omitted, since MP model defines only acceptable event traces.

11.EVENT ATTRIBUTES

Event attributes are immutable values associated with event instances. Timing attributes, like time of beginning, time of end, and duration may be associated with any event instance, with obvious constraints on their values implied by PRECEDES and IN relations. If event duration attributes are provided in the model, it becomes possible to perform different timing estimates for event traces, to search for critical paths (similar to the PERT charts [Fazar 1959]), and to obtain average timing estimates for the sets of event traces.

Assertions involving event attributes may impose additional constraints on a system's behavior. For example, events used as other event attributes may be useful to specify a system's topology, like networks and other graphs.

Example 15. Events used as event attributes.
Dining Philosophers problem (http://en.wikipedia.org/wiki/Dining_philosophers_problem) is often used to illustrate synchronization issues and the techniques for resolving them. The following MP model provides abstract specification of all correct behaviors, but does not provide implementation details.

The example demonstrates the use of events as attributes of other events. Attribute names are binary relations, and syntax and semantics of Alloy relational logic notation [Jackson 2006] are used in this example to describe constraints involving the attribute relations.

Some Alloy-specific syntax:

- the **disj** keyword implies that the attribute values (events in this example) are disjoint;
- **~r** is a transposition of binary relation **r**;
- **^r** stands for the transitive closure of binary relation **r;**
- the dot, like in **left_user.right** stands for the left-associative join operator in Alloy;
- **&&** denotes the set intersection operation in Alloy.

```
SCHEMA Dining_Philosophers
ROOT Philosophers:    {* <5>   Philosopher *};
        Philosopher:    (*  think   eat  *)
                ATTRIBUTES {  -- attributes of the Philosopher event
                disj    left, right:                Philosopher;
                disj    left_fork, right_fork:      Fork;          };
```

Using events as attributes of other events extends the standard relations PRECEDES and IN, for instance, to model interacting components' topology and other dependencies between events.

Event grammar rules appear to be yet another event attribute with a specific syntax and without attribute name. By assigning attribute names to the behavior rules, it may be possible to define several different behaviors as attributes and to make the selection of a behavior attribute dependent on other attributes. This may be used to specify dynamic and product line architectures.

```
        eat:            { use_left_fork,  use_right_fork };
        use_left_fork:  use_fork;
        use_right_fork: use_fork;
```

No ordering is required for **use_left_fork**, **use_right_fork** activities.

```
ROOT Forks: {* <5> Fork *};
        Fork:   (* ( idle | use_fork )  *)
                ATTRIBUTES {
                disj    left_user, right_user:      Philosopher; };
```

The following constraints provide topology specification as filters for trace generation.

Several attributes are mutually symmetric relations.

```
ENSURE          ( left = ~right and left_fork = ~right_user and right_fork = ~left_user );
```

Philosophers form a cycle. Standalone **Philosopher** event name stands here for the set of all events of this type in the event trace (following Alloy conventions).

```
ENSURE          FOREACH $p: Philosopher FROM Philosophers ($p.^left = Philosopher );
```

Neighbors share the same **Fork**.

```
ENSURE          FOREACH $f: Fork FROM Forks ( $f.left_user.right.left_fork = $f );
```

Philosophers behavior is synchronized via event sharing. **A |+| B** is an exclusive union of events w.r.t. **SHARE ALL**, which prevents **A** and **B** from sharing the same instance of the event **E** (partition for sharing) in **(A |+| B), C SHARE ALL E**.

```
COORDINATE <!>          $p: Philosopher          FROM   Philosophers
```

```
DO
        ( use_left_fork && ^IN.$p)  |+|  ( use_right_fork && ^IN.($p.left) ),
        $p.left_fork                                SHARE ALL       use_fork
OD;
```

12.A FEW WORDS ON ASSERTIONS AND QUERIES

An event trace represents an example of particular execution of the system or a use case, especially if the behavior of the environment is included. Event traces can be effectively derived from the event grammar rules and then adjusted and filtered according to the composition operations and constraints in the schema. This justifies the term *executable architecture model*. For a given MP schema it is possible to obtain all valid event traces up to a certain limit. Usually such a limit (*scope*) may be set by the maximum total number of events within the trace, or by the upper limit on the number of iterations in grammar rules (recursion can be limited in a similar way). For many purposes a modest limit of 3 iterations will be sufficient. This process of generating and inspecting event traces for the schema is similar to the traditional software testing process.

In the case of MP models it is possible to automatically generate all event traces within the given scope (exhaustive testing). Careful inspection of generated traces (scenarios/use cases) may help developers identify undesired behaviors. Usually it is easier to evaluate an example of behavior (particular event trace) than the generic description of all behaviors (the schema). The *Small Scope Hypothesis* [Jackson 2006] states that most errors can be demonstrated on relatively small counterexamples.

The assertion language for MP requires a rigorous and precise discussion, and the use of assertion checking for MP model testing and debugging is not subject of this paper. Here are some considerations outlining the possible approach.

The event trace is a set of events and the assertion formalism can embrace the traditional predicate calculus notation. Properties of behavior can be formalized as assertions about traces (similar to the **ENSURE** constraint), and verified exhaustively for all event traces within the scope, yielding the counterexamples when the assertion is violated. For example, hazard states can be specified as a result of certain interactions between the system and its environment, and the traces within scope can be searched for a trace that matches the hazard scenario. Since assertion checking is performed on a complete event trace, it becomes possible to refer to events following a given event to specify fairness conditions. This brings the expressiveness of MP assertions closer to temporal logic [Pnueli 1981]. For instance, simple_message_flow schema (*Example 2*) can be annotated with the assertion

```
ASSERT          FOREACH $x: send FROM Task_A          Has_following(receive) ($x);
```

where **Has_following(receive)** predicate is based on the **PRECEDES*** relation.

In a similar fashion queries can be performed on the traces, providing different kinds of statistics. For example, collecting a representative amount of event traces and calculating durations for event sequences of interest can provide system performance estimates.

Another example of an add-on in MP model may be the probability of an event in alternatives, like **([0.3] A | [0.7] B)** establishing that **A** happens with the probability 0.3 and **B** with probability 0.7. Now it becomes possible to estimate probabilities of certain event traces within a given scope, e.g. probability for the system to get into a hazard state. This opens a direction for system simulation and statistical experiments based on executable systems

architecture models and their environment models. [Songzheng Song et al. 2014] has an example of such experiment with MP model using PAT model checker [Jun Sun et al. 2009].

Assertions and queries may be generic for a class of architecture models and could be reused, as Appendix 1 suggests.

13.IMPLEMENTATION

An online demo of the early MP version Eagle6 (on-line editor, event trace generation and visualization, simple query processor, and rudimentary event trace probability estimation) is available at http://eagle6modeling.riverainc.com/. This web site also provides several complete examples.

The MP prototype [Auguston, Whitcomb 2010] has been implemented as a compiler generating an Alloy model [Jackson 2006] from the MP schema and then running the Alloy Analyzer to obtain event traces and to perform assertion checks. It has benefited from Alloy's relational logic formalism and visualization tools. Performance depends on the performance of SAT solver used by Alloy Analyzer.

Direct trace generation from the event grammar can be accomplished quite efficiently, and the process of generating all traces for the given schema within a given scope can be roughly described by the following procedure.

1. For each root in the schema derive a collection of all possible event traces within a given scope.
2. Select one trace from each root's collection, assembling the resulting trace from root traces with matching numbers of shared/coordinated events.
3. Apply schema's composition operations and filters. If the resulting composed trace is consistent with the schema's filters (including Axioms from Appendix 1) and composition operations, it is included into the schema's trace collection. Otherwise, proceed with the next selection (step 2).

This process has potential for optimization by applying early pruning whenever possible. The main optimization ideas stem from the considerations that composition operations (**COORDINATE, SHARE ALL,** and **MAP**) usually require an equal number of selected events in the matching traces. Root traces can be sorted according to the number of matching events to avoid selection of inconsistent root traces in Step 2. Rearrangement of composition operations and filters may also provide a significant speed up in the trace assembly.

A prototype trace generator has been built by converting MP schemas into a C++ code and then compiling and running the generated C++ code to obtain all event traces within a given scope. This architecture solution is similar to the one implemented in the SPIN/PROMELA model checker (using C as a target language) [Holzmann 2004].

Several optimizations mentioned above have been implemented. A sample run on an iMac with 2.8 GHz/4 GB yields the following performance for a schema example with approximately 60 lines of MP source text, including 9 roots, 10 composite event types, 12 atomic event types, 12 **SHARE ALL** compositions, and for a maximum scope of 3 for iterations. Actually, it is an architecture model for the MP -> C++ prototype itself, the complete MP code for this model is available as Example 9 in the MP Crash Course on http://wiki.nps.edu/display/MP and can be executed on Eagle6 prototype as well.

- Total 1328 traces generated, with total 79836 events, average 60.1175 events/trace, max trace length 69;

- Initial search space (number of all root traces before filtering) 35100;
- Selection ratio 3.78348%, generation speed 18021.8 events/sec;
- Elapsed time (including compilation of the generated C++ code) 4.42997 sec.

14.CONCLUSIONS

MP executable architecture models provide a high level of abstraction for testing, verifying, and documenting system architecture early in the design phase. The main advantages may be summarized as follows.

- MP focuses a developer's attention on the behavior of the system early in the process.
- The ability to separate behavior models of components from the models of interactions between them simplifies the modeling process and facilitates the reuse of architecture models.
- The schema framework is amenable to stepwise architecture refinement, reuse, composition, visualization, and application of automated tools for sanity checks.
- Executable system architecture models integrated with environment behavior models can be helpful for identifying emergent behaviors.
- The ability to generate an exhaustive set of use cases (within a given scope) for requirements specification and for testing the system's implementation. This emphasizes the role of architecture models as a bridge between the requirements and design.
- The pseudo-code style of use cases (event traces) derived from MP schema provides a uniform communication vehicle for different stakeholders. It may be easier for humans, especially not skilled in formal specification notation, to understand and inspect examples of system behavior, neither to deal with the complete and formal MP schema, from which these use cases have been derived. Assertion checking can automatize search for counterexamples violating expected behavior properties.
- The ability to extract different architecture views from the same MP architecture model.
- The ability to develop performance estimates based on statistics obtained from the generated event traces.

14.1 WHAT IS NEXT?

Architecture modeling has substantial consequences for the next phases in software design process. Here are some threads of research stemming from the ideas described above.

- Testing automation. Monitoring the behavior of an implemented system using the MP executable architecture model as an oracle. If the source code of implementation can be instrumented to mark which segments of code start and end corresponding MP events, it becomes possible to log actual execution traces (using [Lamport 1978] timestamps for partial ordering monitoring), and to check the actual traces for consistence with expected behaviors. Test cases could be constructed from the MP event traces using the environment's behavior as the source for inputs and deriving the oracle from the corresponding MP behavior event trace.
- Developing methods and techniques for static analysis of the architecture model, for example, by verifying MP models with a model checking tool. The first prototypes are presented in [Jiexin Zhang et al. 2012] and [Songzheng Song et al. 2014].
- Introducing metrics for MP models for system cost estimates based on the architecture models. Function Points [Albrecht 1979] can be identified as abstract interactions between components and between the system and its environment in the MP model.

Consequently FP analysis and COCOMO II [Boehm et al. 2000] techniques could be applied for the early system cost estimates.

- Statistical simulation and analysis of system's behavior models [Songzheng Song et al. 2014].
- Development of reusable architecture patterns and architecture views libraries.
- Development of business process models in MP. Existing process modeling frameworks (BPEL, BPMN [Grosskopf et al. 2009], IDEF) usually follow the *"single flowchart"* paradigm. MP separates component behaviors from the component interaction, and thus provides a multidimensional picture of concurrent behaviors, with overlapping threads of process phases, participating actors, and other environment behaviors. Exhaustive scenario generation, assertion checking, and queries supported by automated tools (like critical path calculation on PERT charts [Fazar 1959]) could be useful for business process model verification and validation within the MP framework.

ACKNOWLEDGMENTS

Joey Rivera [Rivera 2010] and Alex Gociu were instrumental in the implementation of the MP online demo Eagle6. The diagrams in Fig. 1 - 10 have been designed with the yEd Graph Editor (http://www.yworks.com/en/products_yed_about.html).

The author would like to thank Kristin Giammarco and Monica Farah-Stapleton, for valuable feedback. Kristin's Giammarco public web site for Monterey Phoenix project at the NPS Department of Systems Engineering (http://wiki.nps.edu/display/MP) contains an MP Crash Course for Eagle6 with additional collection of executable examples.

This research was funded in part by the Consortium for Robotics and Unmanned Systems Education and Research (CRUSER http://cruser.nps.edu). Views and conclusions in this article are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government, of the U.S. Department of Defense, or of the U.S. Department of Navy.

APPENDIX 1

Base schema specifies a filter for every MP event trace and ensures that it satisfies partial order axioms for **IN*** and **PRECEDES*** relations. It uses predefined generic event type **Event**. The special variable **$Trace** stands for the whole trace specified by a schema. We use symbols ¬, ∧, ⇒ for Boolean operations here.

SCHEMA Base

There are no root events, this schema is used only to bring the following filter into derived schema.

ENSURE FOREACH $a, $b, $c: Event FROM $Trace

-- Mutual Exclusion of Relations

($a PRECEDES* $b ⇒ ¬($a IN* $b)) ∧ *--Axiom 1)*
($a PRECEDES* $b ⇒ ¬($b IN* $a)) ∧ *--Axiom 2)*
($a IN* $b ⇒ ¬($a PRECEDES* $b)) ∧ *--Axiom 3)*
($a IN* $b ⇒ ¬($b PRECEDES* $a)) ∧ *--Axiom 4)*

-- Non-commutativity

($a PRECEDES* $b ⇒ ¬($b PRECEDES* $a)) ∧ *-- Axiom 5)*
($a IN* $b ⇒ ¬($b IN* $a)) ∧ *-- Axiom 6)*

-- Irreflexivity for PRECEDES and IN* follows from non-commutativity.*

-- Transitivity

(($a PRECEDES* $b) ∧ ($b PRECEDES* $c) ⇒ ($a PRECEDES* $c)) ∧ *-- Axiom 7)*
(($a IN* $b) ∧ ($b IN* $c) ⇒ ($a IN* $c)) ∧ *-- Axiom 8)*

-- Distributivity

(($a IN* $b) ∧ ($b PRECEDES* $c) ⇒ ($a PRECEDES* $c)) ∧ *-- Axiom 9)*

(($a PRECEDES* $b) ∧ ($c IN* $b) ⇒ ($a PRECEDES* $c)); *-- Axiom 10)*

Each MP schema uses **Base** as a default extension. As a result, all event traces will be filtered for compliance with the Axioms. For example, the following schema has an empty set of traces, because it violates Axiom 5 for partial ordering.

SCHEMA Wrong
INCLUDE Base;
NEW Base;
ROOT A: a b;
ROOT B: b a;
A, B SHARE ALL a, b;

REFERENCES

ABOWD, G., ALLEN, R., GARLAN, D., 1995, Formalizing Style to Understand Descriptions of Software Architecture, ACM Transactions on Software Engineering and Methodology 4(4): 319-364

ABOWD, G., BASS, L., CLEMENTS, P., KAZMAN, R., NORTHROP, L., ZAREMSKI, A., 1997, Recommended Best Industrial Practice for Software Architecture Evaluation, Technical Report, CMU/SEI-96-TR-025.

AHO, A., SETHI, R., ULLMAN, J., 1986, Compilers, Principles, Techniques, and Tools, Addison-Wesley

ALBRECHT, A. J., Measuring Application Development Productivity, Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, Monterey, California, October 14–17, IBM Corporation (1979), pp. 83–92.

ALLEN, R., 1997, A Formal Approach to Software Architecture, Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, May 1997

ALLEN, R., GARLAN, D., 1997, A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, Vol. 6(3): 213-249, July 1997.

AUGUSTON, M., 1991, FORMAN - Program Formal Annotation Language, in Proceedings of 5th Israel Conference on Computer Systems and Software Engineering, Herclia, May 27-28, IEEE Computer Society Press, 1991, pp.149-154.

AUGUSTON, M., 1995, 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.

AUGUSTON, M., JEFFERY, C., UNDERWOOD, S., 2002, 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.

AUGUSTON, M., MICHAEL, B., SHING, M., 2006, 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

AUGUSTON, M., 2009, Software Architecture Built from Behavior Models, ACM SIGSOFT Software Engineering Notes, 34:5.

AUGUSTON, M., 2009, Monterey Phoenix, or How to Make Software Architecture Executable, OOPSLA'09/Onward conference, Orlando, Florida, OOPSLA Companion, October 2009, pp.1031-1038

AUGUSTON, M., WHITCOMB, C., 2010, System Architecture Specification Based on Behavior Models, in Proceedings of the 15th ICCRTS Conference (International Command and Control Research and Technology Symposium), Santa Monica, CA, June 22-24, 2010

AUGUSTON, M., WHITCOMB, C., 2012, Behavior Models and Composition for Software and Systems Architecture, ICSSEA 2012, 24th International Conference on Software & Systems Engineering and their Applications, Telecom ParisTech, Paris, October 23-25, 2012

BASS, L., CLEMENTS, P., KAZMAN, R., 2003, *Software Architecture In Practice*, 2nd Edition, Boston, Addison-Wesley.

BOEHM, B., ABTS, C., BROWN, A. W., CHULANI, S., CLARK, B.K., HOROWITZ, E., MADACHY, R., REIFER, D.J., and STEECE, B. *Software Cost Estimation with COCOMO II* (with CD-ROM). Englewood Cliffs, NJ:Prentice-Hall, 2000.

BOOCH, G., JACOBSON, I., RUMBAUGH, J., 2000, OMG Unified Modeling Language Specification, http://www.omg.org/docs/formal/00-03-01.pdf

BOSCH, J., 2000, Design and Use of Software Architectures, ACM Press and Addison-Wesley.

BRUEGGE, B., HIBBARD, P., 1983, Generalized Path Expressions: A High-Level Debugging Mechanism, The Journal of Systems and Software 3, 1983, pp. 265-276.

CARRIERO, N., GELERNTER, D., 1992, "Coordination Languages and their Significance", *Communications of the ACM* **35 (2)**, pp. 97-107.

CAMPBELL, R.H., HABERMANN, A.N., 1974, The Specification of Process Synchronization by Path Expressions, Lecture Notes in Computer Science, No. 16, Apr. 1974, pp. 89-102.

DOBRICA, L., NIEMELA, E., A Survey on Software Architecture Analysis Methods, 2002, IEEE Transactions on Software Engineering, Vol.28, No 7, pp.638-653.

FAZAR, W., "Program Evaluation and Review Technique", *The American Statistician*, Vol. 13, No. 2, (April 1959), p.10.

FEILER, P., GLUCH, D., HUDAK, J., 2006, 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)

GROSSKOPF, DECKER and WESKE. (Feb 28, 2009). The Process: Business Process Modeling using BPMN, Meghan Kiffer Press.

HAREL, D., 1987, A Visual Formalism for Complex Systems. Science of Computer Programming 8(3), pp.231–274

HAREL, D., 1992, Biting the silver bullet: toward a brighter future for system development, IEEE Computer, Vol. 25(1), pp.8-20

HOARE, C. A. R., Communicating Sequential Processes. Prentice-Hall, 1985.

HOLZMANN, G., 2004, The SPIN Model Checker, Boston, Addison-Wesley 2004

ISO 2011, International Organization for Standardization, ISO Standard ISO/IEC 42010:2007, "Systems and Software Engineering – Recommended Practice for Architectural Description of Software-Intensive Systems."

JACKSON, D., 2006, Software Abstractions: Logic, Language, and Analysis, Cambridge, Massachusetts: The MIT Press.

JUN SUN, YANG LIU, JIN SONG DONG and JUN PANG, PAT: Towards Flexible Verification under Fairness. The 21th International Conference on Computer Aided Verification (CAV 2009), pages 709-714, Grenoble, France, June, 2009.

KICZALES, G., LAMPING, J., MEHDBEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J., IRWIN, J., 1997, "Aspect-Oriented Programming", Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241. June 1997

KNUTH, D., 1984, Literate Programming, The Computer Journal, 27(2): 97-111, May 1984

KRUCHTEN, P., 1995, Architectural Blueprints - the 4+1 View Model of Software Architecture. IEEE Software. 12 (6), pp. 42-45

KRUCHTEN, P., 2001, "Common misconceptions about Software Architecture", The Rational Edge 01/1998

LAMPORT, L., 1978, "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, vol. 21, No. 7, July 1978, pp. 558-565.

LISKOV, B., ZILLES, S., 1974, Programming with abstract data types, ACM SIGPLAN Notices, Vol 9 Issue 4, pp. 50 – 59

LUCKHAM, D., AUGUSTIN, L., KENNEY, J., VERA, J., BRYAN, D., MANN, W. 1995, Specification and analysis of system architecture using Rapide. IEEE Transactions on Software Engineering, Special Issue on Software Architecture, 21(4):336–355, April 1995.

LUCKHAM, D., J., VERA, J., 1995, An Event-Based Architecture Definition Language, IEEE Transactions on Software Engineering, 21(9): 717–734, September 1995.

MEDVIDOVIC, N., ROSENBLUM, D., REDMILES, D., 2002, Modeling Software Architectures in the Unified Modeling Language, ACM Transactions on Software Engineering and Methodology, Vol.11, No. 1, January 2002, pp.2-57.

MILNER, R., 1989, "Communication and Concurrency", Prentice Hall

NASSI, I.; SHNEIDERMAN, B., 1973, Flowchart techniques for structured programming, ACM SIGPLAN Notices XII, August 1973, pp.12 – 26

OMG, 2010, Concrete Syntax for UML Action Language (Action Language for Foundational UML), version Beta 1 (2010), www.omg.org/spec/ALF

OREIZY, P., ROSENBLUM, D., TAYLOR, R., 1998, On the Role of Connectors in Modeling and Implementing Software Architectures, Department of Information and Computer Science, University of California, Irvine, Technical Report UCI-ICS-98-04, Feb. 1998.

PAPADOPOLOUS, G.A., ARBAB, F.,1998, Coordination Models and Languages, Advances in Computers, 48

PELLICCIONE, P., INVERARDI, P., MUCCINI, H., 2009, CHARMY: A Framework for Designing and Verifying Architectural Specifications, IEEE Transactions on Software Engineering, Vol. 35, No 3, 2009, pp.325-346

PERRY, D., WOLF, A., 1992, Foundations for the Study of Software Architecture, ACM SIGSOFT Software Engineering Notes, 17:4, pp. 40-52.

PNUELI, A., 1981, A temporal logic of programs, *Theoretical Computer Science*, 13: pp.45-60.

RIVERA, J., 2010, System Architecture Modeling Methodology for Naval Gunship Software, PhD dissertation, CS Department, Naval Postgraduate School, Monterey, CA, USA, December 2010.

ROSCOE, B., 1997, The Theory and Practice of Concurrency, Prentice Hall International Series in Computer Science (580pp), ISBN 0-13-674409-5

ROZANSKI, N., WOODS, E., 2012. *Software Systems Architecture*, 2nd Edition, Addison-Wesley.

SHAW, M., GARLAN, D. 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs, New Jersey.

SONGZHEN SONG, JIEXIN ZHANG, YANG LIU, MIKHAIL AUGUSTON, JUN SUN, JIN SONG DONG, TIEMING CHEN, Formalizing and verifying stochastic system architectures using Monterey Phoenix, Software & Systems Modeling, Springer Berlin Heidelberg, April 2014, pp.1-19.

SPIVEY, J.M., The Z Notation: A reference manual, Prentice Hall International Series in Computer Science, 1989. (2nd Ed., 1992)

TAYLOR, R., MEDVIDOVIC, N., DASHOFY, E., 2010, *Software Architecture, Foundations, Theory, and Practice,* John Wiley & Sons, Inc.

WANG, J., PARNAS, D., 1994, Simulating the behavior of software modules by trace rewriting**,** IEEE Trans. Software Eng. 20, 10 (Oct. 1994), pp. 750-759.

ZHANG JIEXIN, YANG LIU, AUGUSTON, M., JUN SUN, JIN SONG DONG, 2012, Using Monterey Phoenix to Formalize and Verify System Architectures, 19th Asia-Pacific Software Engineering Conference APSEC 2012, December 4 – 7, 2012, Hong Kong.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943

4. Consortium for Robotics and Unmanned Systems Education and Research, Naval Postgraduate School (CRUSER; http://cruser.nps.edu)
   Monterey, CA 93943