Monterey Phoenix

System and Software Architecture Modeling Language v.4 (Draft)

Examples of architecture models

Mikhail Auguston
Naval Postgraduate School
Monterey, CA, USA
maugusto@nps.edu

Example 1. Car race scenarios.

This example introduces the event grammar notation.

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 generate instances of event traces. An instance of event trace satisfying the grammar can be visualized as an acyclic directed graph with two types of edges (one for each of the basic relations).

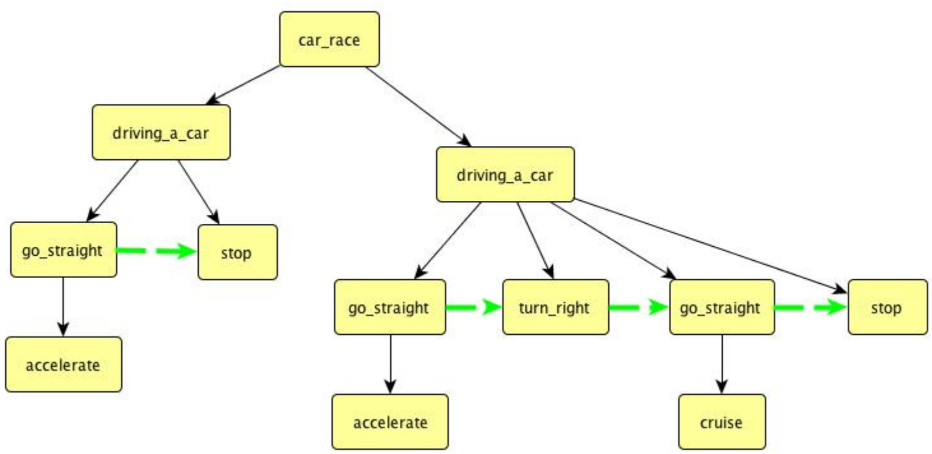


Fig. 1. An event trace derived from the event grammar in Example 1.

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
ADD $x PRECEDES $y;

In order to establish coordination between sending and receiving messages, we use the behavior composition operation **COORDINATE**. In this example the composition operation takes two traces and defines a modified event trace (merges behaviors of Task_A and Task_B) by adding the **PRECEDES** relation between selected **send** and **receive**.

The first part of composition operation (the source) uses event patterns to specify segments of root traces that should be selected. The **(* $x: send)** pattern identifies the sequence of totally ordered **send** events (with respect to the transitive closure of **PRECEDES** relation – **PRECEDES***). Use of the **(* P *)** pattern for selection means that all events P in the source root should be ordered, both iterations should have the same number of selected elements (**send** events from the first trace and **receive** events from the second), both should be totally ordered, and pair selection follows this ordering (*synchronous coordination*). Labels **$x** and **$y** provide access to the events selected within each iteration. The **ADD** keyword completes the behavior adjustment, specifying that ordering relation will be imposed on 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.
The composition operation may be considered as an abstract interface description for root behaviors. When *asynchronous coordination* is needed, an iterative set pattern can be used. For example,

COORDINATE {* $x: E1 *} FROM A, {* $y: E2 *} FROM B ADD $x PRECEDES $y;

In this case matching root traces for A and B still should contain an equal number of selected events of types E1 and E2, correspondingly. But now the resulting merged traces will include all permutations of events E2 from B matching 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 resulting trace for the schema deploying this composition. In order to reduce the exponential explosion, optimizations similar to symmetry reduction in model checking tools should be considered. Changing **(* … *)** for **{*… *}** in Example 2 may increase the number of composed traces in the schema.

Different views can be extracted from MP schemas. For example, each root may be visualized as a box, and if there is a composition operation specifying an interaction between root behaviors, the boxes are connected by arrow marked by the interacting event types. The root behavior may be visualized with UML Activity Diagrams [Booch et al. 2000]. The MP developer's environment may have a library of predefined views providing different visualizations for schemas.

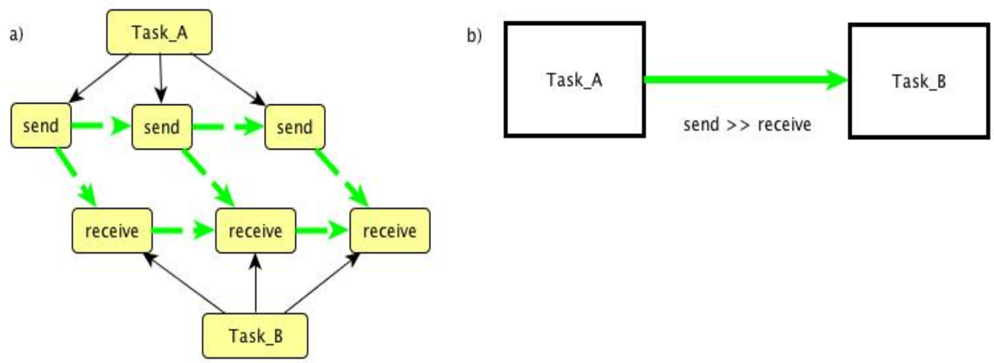


Fig. 2. a) Example of composed event trace for the simple_message_flow schema. b) An architecture view for the simple_message_flow schema.

1.1 Data items as behaviors

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

Example 3. 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;

Behavior of the File requires that all write events should be completed before the read events. The **SHARE ALL** composition operation ensures that the schema admits only event traces where corresponding event sharing is implemented. Event sharing is in fact yet another way of behavior coordination (similar to the *rendezvous* in Ada). It is assumed that shared events may appear in the root event at any level of nesting. The view of this schema in Fig.3 b) renders root interaction with a line where shared event name is attached as a label.

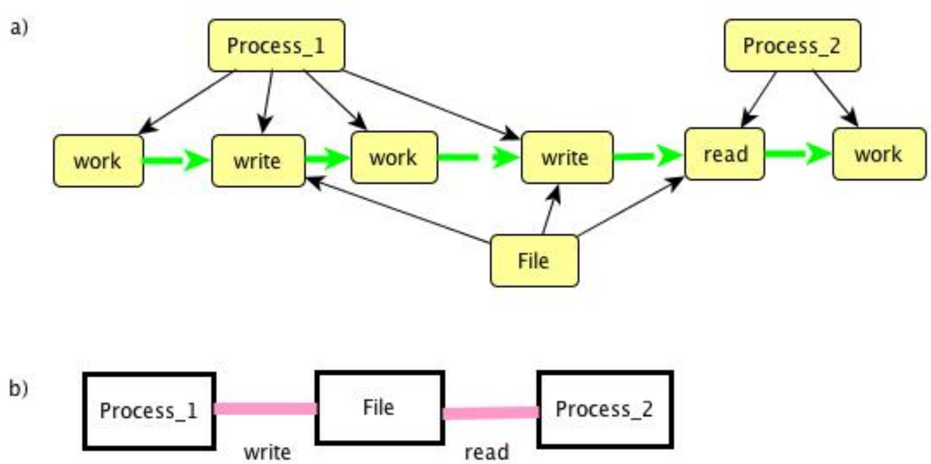


Fig. 3. a) an example of composed event trace for the Data_flow schema. b) an architecture view for the Data_flow schema.

Example 4. Stack behavior.

```
SCHEMA Stack
ROOT Stack_operation: (* ( push | pop ) *);
SATISFIES FOREACH $x: pop FROM Stack_operation
        ( Number_of (pop) before ($x) < Number_of (push) before ($x) );
```

This schema specifies behavior of a stack in terms of stack primitive operations. Let **IN*** denote the transitive closure of **IN** relation (similarly **PRECEDES*** is a transitive closure for **PRECEDES**). The domain of universal quantifier is the set of all **pop** events **e**, such that **(e IN* Stack_operation).** The operation **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.

Example 5. Reuse of a schema.

```
SCHEMA Two_stacks_in_use
INCLUDE Stack;
ROOT Main: {* (do_something | use_S1 | use_S2) *};
  use_S1:  (push | pop) ;
  use_S2:  (push | pop) ;
ROOT S1: Stack;
ROOT S2: Stack;
S1, Main SHARE ALL $x: (pop | push) SUCH THAT Has_enclosing (use_S1)($x) WITHIN Main;
S2, Main SHARE ALL $x: (pop | push) SUCH THAT Has_enclosing (use_S2)($x) WITHIN Main;
```

The **INCLUDE** statement brings the schema Stack into the scope. This means that all constraints specified in the Stack also will be included. The rule for Main is intentionally left very lax without imposing any specific ordering on embedded activities. Roots S1 and S2 represent the presence of two independent stacks as data items. The ordering of pop and push events inside use_S1 and use_S2 in each stack behavior is ensured and will be brought into the resulting trace by the included Stack behaviors as a result of sharing these events with Stack behavior. The **SHARE ALL** composition operation uses event patterns and context conditions to accomplish the necessary event trace construction. The predicate **Has_enclosing(T)(e1)** is true iff there exists an event e2 of the type T in the trace specified by the **WITHIN** clause, such that **e1 IN* e2**.

Predicates and functions like **Has_enclosing(T)(e**), and **Number_of (T) before (e)** are used for convenient navigation in the event graphs.

Example 6. Components and connectors.

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

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

Buffered_channel simulates the behavior of a connector between **Task_A** and **Task_B**. This behavior model does not provide details about what happens after the buffer overflow event.

SCHEMA Buffered_transaction
ROOT Task_A:: (* Send *);
ROOT Task_B:: (* Receive *);
ROOT Buffered_channel: {* (Send [Receive]) *} (Overflow | Normal);

Task_A, Buffered_channel SHARE ALL Send;
Task_B, Buffered_channel SHARE ALL Receive;

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

If the schema should satisfy only behaviors without buffer overflow, the three **SATISFIES** conditions above can be replaced by the following constraint (and the **Overflow** event can be removed from the schema):

SATISFIES FOREACH $x: Send FROM Buffered_channel
 Number_of ($y: Send) before ($x) SUCH THAT (¬ Has_next(Receive)($y)) <
 max_buffer_size;

Note that **PRECEDES** relation is defined explicitly either in the grammar rule, or by **ADD** composition operation, and is a proper subset of its transitive closure PRECEDES*. The predicate **Has_next(T)(e1)** is true iff there exists an event **e2** of the type **T** in the trace, such that **e1 PRECEDES e2**.

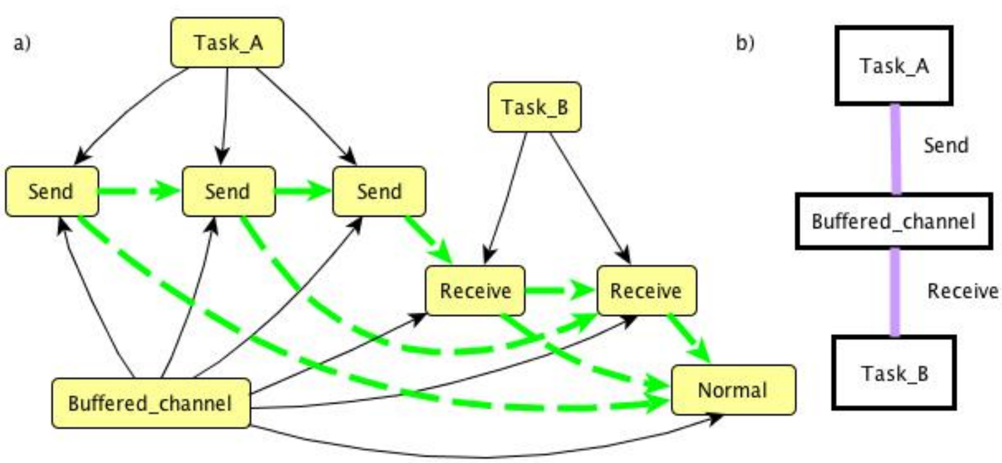


Fig. 4. a) an example of event trace (without overflow) for the Buffered_transaction schema with max_buffer_size = 3.
b) an architecture view for the Buffered_transaction schema.

1.2 Environment's behavior

The following example demonstrates how to integrate the behavior of environment with the behavior of system. The ATM_withdrawal schema specifies a set of possible scenarios of interactions between the Customer, ATM_system, and Data_Base. Each event trace generated from this schema can be considered as a use case example.

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      ADD $x PRECEDES $y ;
COORDINATE   (* $x: request_withdrawal *) FROM Customer,
             (* $y: check_balance *)      FROM ATM_system     ADD $x PRECEDES $y ;
COORDINATE   (* $x: identification_succeeds *) FROM Customer,
             (* $y: id_successful *)  FROM ATM_system         ADD $y PRECEDES $x ;
COORDINATE   (* $x: get_money  *) FROM Customer,
             (* $y: dispense_money *) FROM ATM_system         ADD $y PRECEDES $x ;
COORDINATE   (* $x: not_sufficient_funds *) FROM Customer,
             (* $y: unsufficient_balance *) FROM ATM_system          ADD $y PRECEDES $x ;
COORDINATE   (* $x: identification_fails  *) FROM Customer,
             (* $y: id_failed  *)          FROM ATM_system    ADD $y PRECEDES $x ;
```

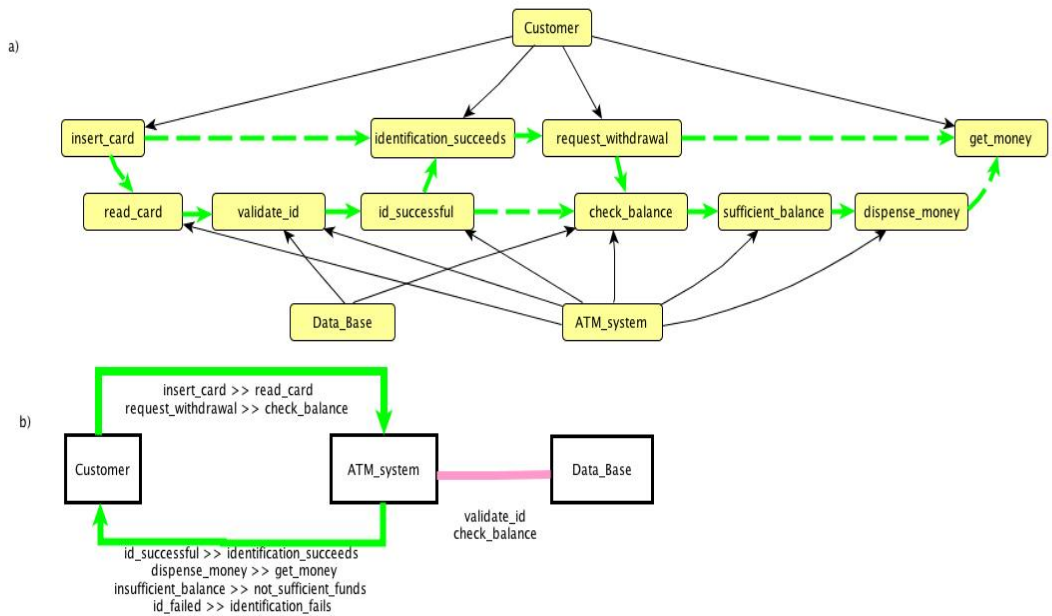


Fig. 5. a) an example of event trace for the ATM_withdrawal schema.
b) an architecture view for the ATM_withdrawal schema.

If the view of the whole system's behavior emphasizing the interaction between the parts (components) can be visualized as in Fig. 5, b), the view of root's standalone behavior can be visualized as an UML Activity Diagram. Since event aggregates (iterations, alternatives, sets) in MP are well structured, it is possible to use Nassi–Shneiderman diagrams as well. This example demonstrates that MP models can be integrated into standard frameworks, like UML, SysML, DoDAF, providing the level of abstraction convenient for architecture models.

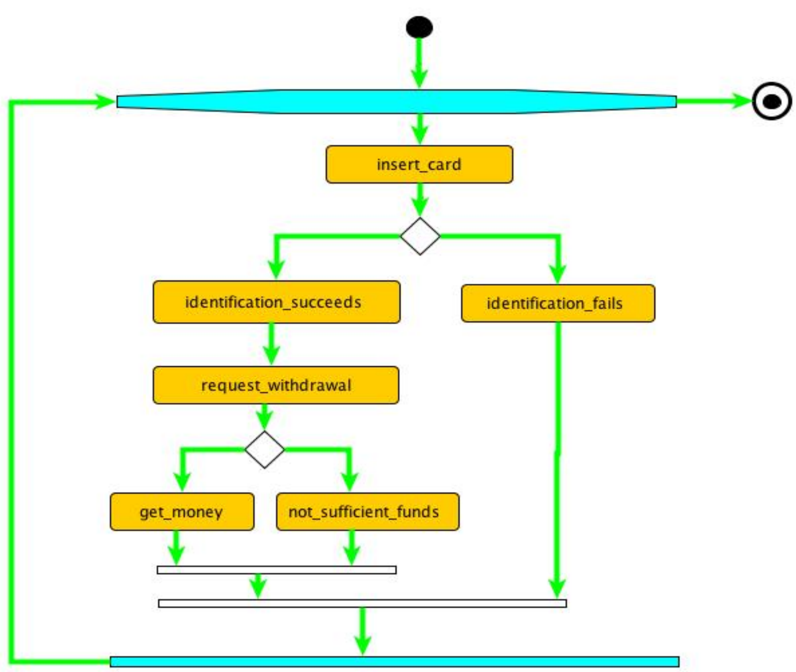


Fig. 6. A view on the Customer root event behavior

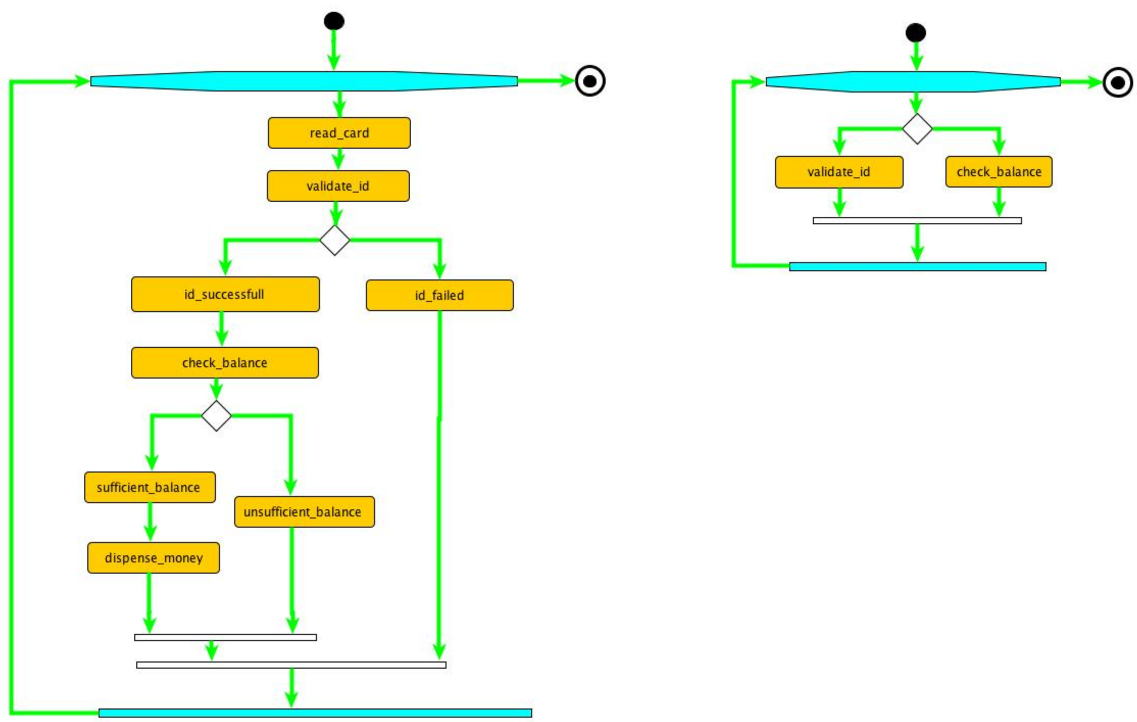


Fig. 7. A view on the ATM_system and Data_Base root events behavior

1.3 Merging schemas

So far, we have seen examples of assembling schemas using previously defined schemas (Example 5). Each schema in the assembly holds its own roots and composition operations (**SATISFIES** filter and interaction constraints, like **COORDINATE** and **SHARE ALL**) within its scope.

The join operation for schemas looks like:
SCHEMA A EXTENDS B
Roots for A
Constraints and composition operations involving roots from both A and B

The resulting schema A joins roots defined in A and roots defined in B, merges within its scope all constraints and composition operations defined in B, and may have additional constraints and composition operations involving all roots. The following example contains Base schema specifying properties for basic relations IN* and PRECEDES*. It is assumed that any MP schema extends on Base. This operation on schemas is inspired by Z schema expressions concept [Spivey 1989].

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.
Each MP schema uses the **Base** as a default extension. As a result, each schema will filter its event traces accordingly, for example, the following schema has empty set of traces, because it violates Axiom 5 for partial ordering.

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

Base specifies a filter for every 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. The purpose of this schema is similar to the purpose of virtual class in OO paradigm.

Example 8.

SCHEMA Base
-- there are no roots, this schema is used only to bring the following filter into derived schema
SATISFIES 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

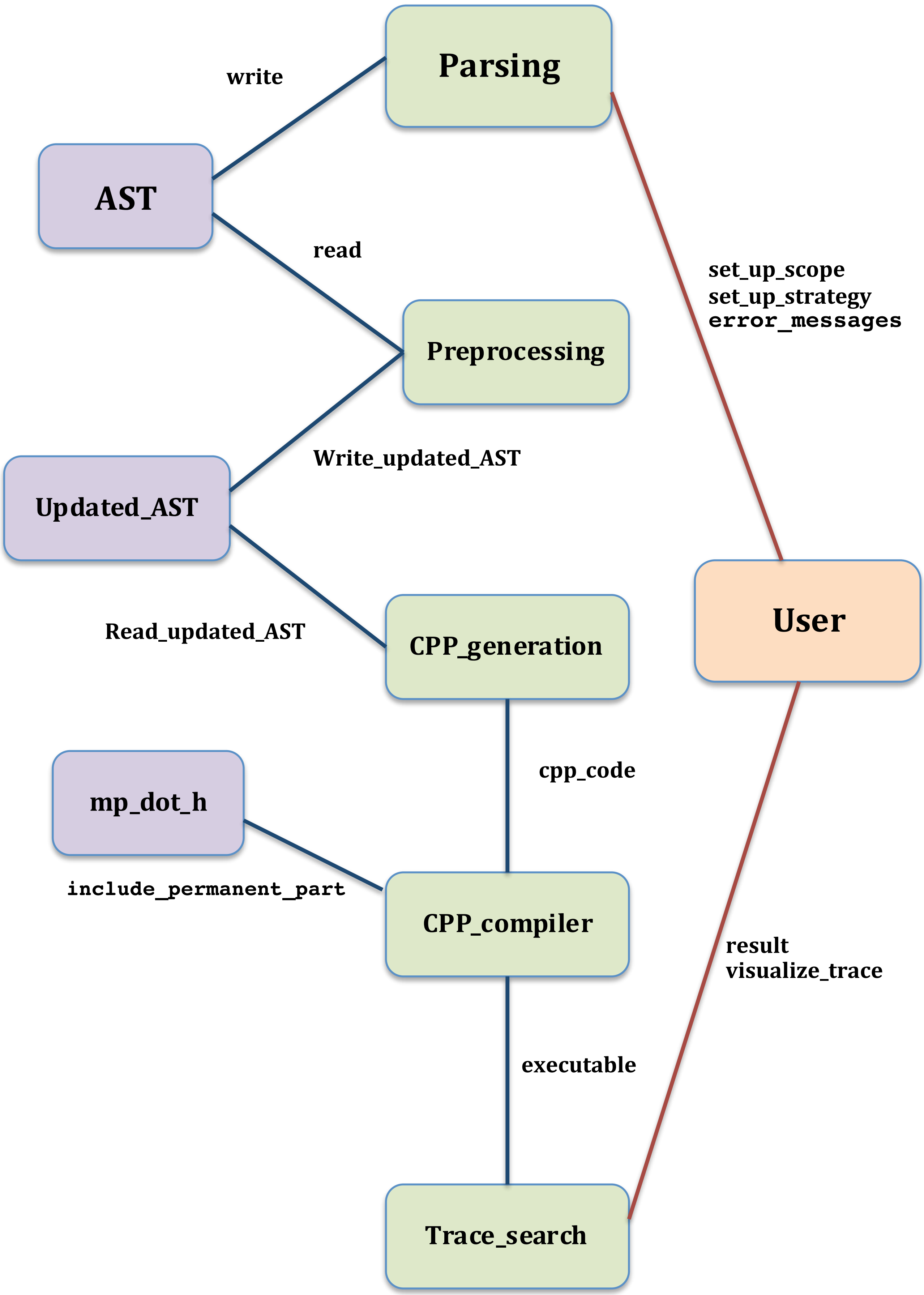
(($a IN* $b) ∧ ($b PRECEDES* $c) ⇒ ($a PRECEDES* $c)) ∧ *-- $Axiom 9)*
(($a PRECEDES* $b) ∧ ($c IN* $b) ⇒ ($a PRECEDES* $c)); *-- Axiom 10)*

Example 9. Architecture model for MP/C++ prototype trace generator

```
/************************************************
     MP/C++ trace generator architecture
     Mikhail Auguston, CS Dept NPS
     May 2011
************************************************/
scope = 3
SCHEMA MP_arch
/*==============================================*/
 ROOT User:      set_up_scope
                 set_up_strategy
                 [error_messages]
                 (* result [ visualize_trace ] *) ;

 ROOT Parsing:   set_up_scope
                 set_up_strategy
                 {ast_building, [ error_messages ] };

  ast_building:  {write, syntax_analysis};
```

```
 syntax_analysis:    {process_roots,
                      process_composites,
                      process_constraints,
                      process_queries     };

    User, Parsing SHARE ALL    set_up_scope,
                               set_up_strategy,
                               error_messages;

ROOT AST: write read;

    Parsing, AST SHARE ALL write;

ROOT Preprocessing: { read, transformations};
    transformations:      eliminating_iterators
                          trace_estimation
                          create_updated_AST;

    create_updated_AST: write_updated_AST;

    Preprocessing, AST SHARE ALL read;

ROOT Updated_AST: write_updated_AST read_updated_AST;

    Preprocessing, Updated_AST SHARE ALL write_updated_AST;

ROOT CPP_generation: read_updated_AST code_generation;

    code_generation: generate_root_predicates
                     generate_event_signatures
                     generate_main_subroutine
                     cpp_code;

    CPP_generation, Updated_AST SHARE ALL read_updated_AST;

ROOT CPP_compiler:   include_permanent_part
                     cpp_code
                     executable;

ROOT mp_dot_h:       include_permanent_part;

    CPP_generation, CPP_compiler SHARE ALL cpp_code;
    mp_dot_h, CPP_compiler SHARE ALL include_permanent_part;

ROOT Trace_search:  executable generate_traces;
    generate_traces: create_signatures
                     sort_segment_lists
                     (*    assemble_a_trace
                           check_global_constraints
                     ( pass_global_check [ perform_queries ] |
                       fail_global_check )
                     *);
```

```
assemble_a_trace:     expand_composite_segments
                      match_root_segments;
perform_queries: (    execute_query show_result |
                      check_assertion
                      [assertion_fails
                       report_assertion_violation ]
                 )
                 [ visualize_trace ];
show_result:                     result;
report_assertion_violation:      result;

Trace_search, CPP_compiler SHARE ALL executable;

Trace_search, User SHARE ALL result, visualize_trace;
```


Parsing
write
AST
read
set_up_scope
set_up_strategy
error_messages
Preprocessing
Write_updated_AST
Updated_AST
User
Read_updated_AST
CPP_generation
cpp_code
mp_dot_h
include_permanent_part
CPP_compiler
result
visualize_trace
executable
Trace_search

Example of work – statistics of event traces generated from the model above.

For scope 3, total 1328 traces generated, with total 79836 events
(average 60.1175 events/trace, max trace length 69);
Initial search space (number of all root trace pairs 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.

Example 10. Two components communicating via unreliable channel.

```
/*
 *  AtoB.mp
 *
 *  Created by Mike Auguston on 3/18/10.
 */
 SCHEMA AtoB
/*****************/
 ROOT TaskA: (*  A_sends_request_to_B
                      ( A_receives_data_from_B   |
                        A_timeout_waiting_from_B  )
             *);

/* assumes that A is the leading actor,
    this model can be modified making A and B to behave similarly
*/

/******************/
 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 *);
/*  request_bounces_back activity simulates the connector's
                                      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 )  *);
```

```
/************ sharing constraints ************/
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;
```

Example 11. Architecture of compiler's front end.

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

Example 11.a. Compiler front end in batch processing mode.

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

```
SCHEMA Lexer
INCLUDE    Token
ROOT Text_Input:      (* (Get_char | Unget_char)  *);
ROOT Output_token_list:     (* Put_token *);
ROOT Token_processing: (* Token_recognition *);
Token_processing, Text_Input SHARE ALL Get_char, Unget_char;
Token_processing, Output_token_list SHARE ALL Put_token;
```

The Input and Output are formalizing our assumptions about input and output streams of events. The structure of the Token_recognition event is defined in the schema Token and is included (reused) in the Lexer schema. It refines the Lexer behavior towards the typical Unix/LEX semantics, when the regular expression in each LEX rule is applied independently, and hence no ordering is imposed. Each RegExpr_Match consumes one or more Get_char events until all finite automata involved in the token recognition enter the Error state, then the winner is selected, and all look-ahead characters beyond the recognized lexeme are returned back into the input stream by Unget_char; the Fire_rule event follows it. As a result of the **include** composition operation the root mark for Token_recognition is localized within the scope of Token_processing event.

```
SCHEMA Token
ROOT Token_recognition:     {* RegExpr_Match *}
                            (* Unget_char *)  Fire_rule;
 RegExpr_Match:  (+ Get_char +);
 Fire_rule:      Put_token;
ALL RegExpr_Match SHARE ALL Get_char;
Number_of( Get_char) in (Token_recognition) >
Number_of( Unget_char) in (Token_recognition);
```

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

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

```
SCHEMA Parser
INCLUDE    Stack;
ROOT Parsing_stack: Stack;
ROOT Input_token_list:     (* Get_token *);
ROOT Output_nodes:    (* Put_node *);
ROOT Parsing:   Push  -- push the start symbol
          (* Get_token (* Reduce *) Shift *) [Syntax_error];
 Shift::          Push ;
 Reduce::  (+ Pop +)  Push  Put_node;
Parsing, Input_token_list  SHARE ALL Get_token;
Parsing, Output_nodes SHARE ALL Put_node;
Parsing, Parsing_stack SHARE ALL Pop, Push;
```

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

```
SCHEMA Stack
ROOT  Stack_operation: (*  ( push | pop )  *);
SATISFIES FOREACH $x: pop FROM Stack_operation
           ( Number_of (pop) before ($x) <
                Number_of (push) before ($x));
```

The constraint reflects the absence of stack underflow.

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

```
SCHEMA Batch_processing EXTENDS Lexer, Parser
DROP Output_token_list, Input_token_list;
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;
SATISFIES Number_of(Put_token) in (Batch)  >=
     Number_of(Get_token) in (Batch);
```

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

The following diagram represents a simplified component/connector view of the Batch_processing architecture.

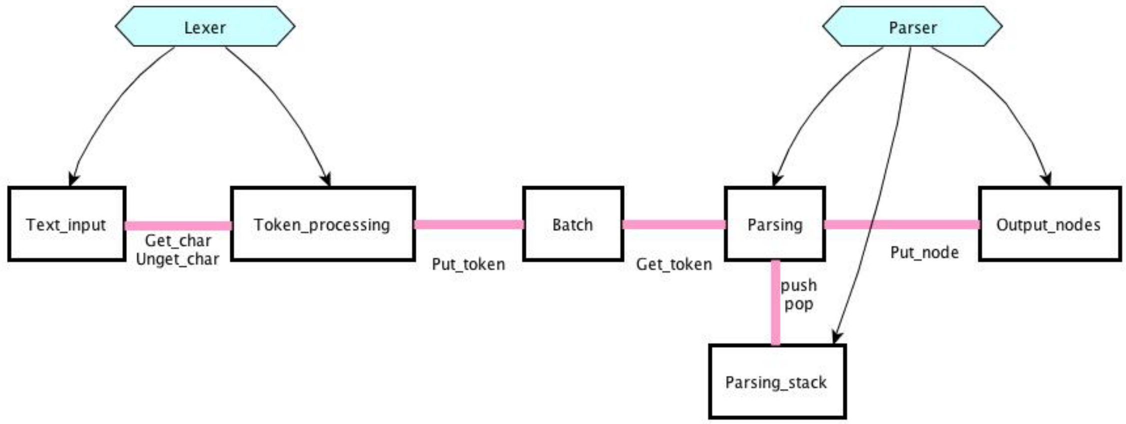


Fig. An architecture view on the Compiler's front end in batch mode.

Example 11.b. Compiler's front end in incremental mode.

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

```
SCHEMA Incremental_processing EXTENDS Lexer, Parser
DROP Output_token_list, Input_token_list;
COORDINATE (* $x: Token_recognition *) FROM Token_processing,
           (* $y: Get_token *) FROM Parsing      ADD $x IN $y;
```

The merged architecture defines a set of event traces where all structuring is inherited from Lexer, Parser, into Incremental_processing schema with the additional constraints for sharing the token processing events.

The composition of the Incremental_parsing schema bears an analogy with the Aspect Oriented Programming approach.

The following diagram represents a simplified component/connector view of the Incremental_processing architecture.

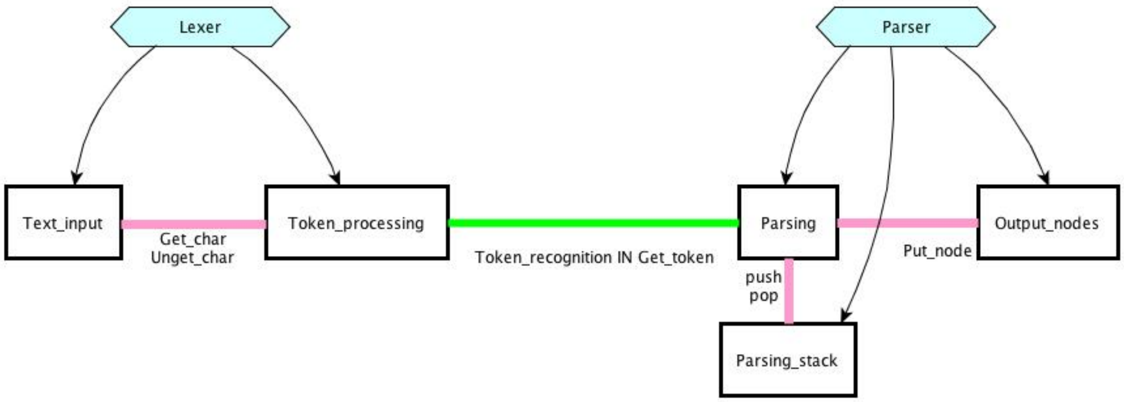


Fig. An architecture view on the Compiler's front end in incremental mode.

Example 12

MP example developed from the problem set in
Denvir T, W. Harwood, M. Jackson, and M Wray, The Analysis of Concurrent Systems Proceedings of a Tutorial and Workshop held Sept. 1983, Cambridge University, LNCS, 207, Springer Verlag, 1985, pp. 97 – 102.

Problem. The channel between endpoints A and B can pass messages in both directions simultaneously until it receives a disconnect message from one end after which it neither delivers nor accepts messages at that end. It continues to deliver and accept messages at the other end until the disconnect message arrives after which it can do nothing. The order of messages sent in a given direction is preserved.

```
SCHEMA P1
ROOT A:  { (*  successful_send_A  *) (* missed_send_A *) ,
           (* receive_A *)                                }
                                               disconnect_A;

ROOT B:  { (*  successful_send_B *) (* missed_send_B *) ,
           (* receive_B *)                     }
                                               disconnect_B;

ROOT Ch: { {* (successful_send_A  receive_B) *}  disconnect_B,
           {* (successful_send_B  receive_A) *}  disconnect_A,
           (* missed_send_A *),
           (* missed_send_B *)                                  };

A, Ch SHARE ALL successful_send_A, missed_send_A,
                receive_A, disconnect_A;

B, Ch SHARE ALL successful_send_B, missed_send_B,
                receive_B, disconnect_B;
```

/ These constraints are supposed to ensure that order of messages sent and received is preserved */*

```
SATISFIES FOREACH $x: receive_A  FROM A
      Number_of(successful_send_B) before ($x) ==
      Number_of(receive_A) before ($x) + 1;

SATISFIES FOREACH $x: receive_B  FROM B
      Number_of(successful_send_A) before ($x) ==
      Number_of(receive_B) before ($x) + 1;
```

Example 13

Problem. Multilayer architecture. An event in top layer deploys several events in the previous layer. Use of 1..n multiplicity coordination.

```
SCHEMA P13
ROOT  Top_layer: (*  do_something  top_event do_anything_else *);
ROOT Bottom_layer: (* something  bottom_event  anything_else  *);

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

This models the case when one top_event contains one or more bottom_events. If needed, more refined iteration multiplicity can be used, like:

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

to limit repetitions of bottom_events.

We can even use multiplicity on each coordination source, as in the following:

```
ROOT Top_layer: (*  top_event     anything_else *);
ROOT Bottom_layer: (* bottom_event   anything_else  *);

COORDINATE
      (*   $x: (*<1..3> top_event *)         *) FROM Top_layer,
      (*   $y: (*<1..4>  bottom_event  *)    *) FROM Bottom_layer
      ADD $x PRECEDES $y;
```

To tell that groups of top_events are placed under PRECEDES with groups of bottom_events.

Example 14

Reuse and extension.

Base specifies a filter for every 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. The purpose of this schema is similar to the purpose of virtual class in OO paradigm.

SCHEMA Base
-- there are no roots, this schema is used only to bring the following filter into derived schema
SATISFIES 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, each schema will filter its event traces accordingly, for example, the following schema has an empty set of traces, because it violates Axiom 5 for partial ordering.

SCHEMA Wrong EXTENDS 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

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, 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

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

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

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

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.

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

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

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

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

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.

JACKSON, M., 2007, Consultancy & Research in Software Development. Past Research Topics. http://mcs.open.ac.uk/mj665/topics.html (last accessed March 2012)

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

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.

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.

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.

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.

SONI, D., NORD, R., HOFMEISTER, C., 1995, Software Architecture in Industrial Applications, in Proceedings of the 17th International Conference on Software Engineering (ICSE 17, Seattle, WA), ACM Press, New York, NY, 1995, pp. 196 – 207.

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