Behavior Models and Composition for Software and Systems Architecture

Mikhail Auguston
Computer Science Department
Naval Postgraduate School
Monterey, California, USA
An architecture description belongs to a high level of abstraction, ignoring many of the implementation details, such as algorithms and data structures.

The architecture plays a role as the bridge between requirements and implementation of a system.

Modeling is an approach to design and verification of system architecture.
• One of the major concerns in architecture design is the question of the behavior of the system
• An architecture specification should be supportive for the refinement process
• Composition operations focus on the interactions between the parts of the system
• An architecture of a system is considered in the context of the environment in which it operates
• The architect needs a number of different views of the architecture for the various uses and users
“Every system has an architecture, whether or not it is documented and understood.”

Monterey Phoenix (MP)

- An approach to formal software and system architecture specification based on behavior models, including concurrency
- A view on the architecture as a high level description of possible behavior of subsystems and interactions between subsystems
- The emphasis on specifying the interaction between the system and its environment
- The behavior composition operations support architecture reuse and refinement toward design and implementation models
- Executable architecture models provide for system architecture testing and verification with tools
Basic Concepts

**Event** - any detectable action in system’s or environment’s behavior

**Event trace** - set of events with two basic partial ordering relations, **precedence** (PRECEDES) and **inclusion** (IN)

**Event grammar** - specifies the structure of possible event traces
Example of an event grammar and event trace

car_race: {+ driving_a_car +};
driving_a_car: go_straight
               (* ( go_straight | turn_left | turn_right ) *) stop;
go_straight: ( accelerate | decelerate | cruise );
Schemas

• The **behavior model** of a system is specified as a set of all possible event traces using a **schema**

• The schema represents **instances of behavior** (event traces), in the same sense as Java source code represents instances of program execution

• A schema contains events called **roots** representing the behaviors of parts of the system (**components** and **connectors** in common architecture descriptions), **composition operations** specifying interactions between root behaviors, and additional **constraints** on behaviors
A 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;

a) Example of composed event trace
b) An architecture view for the schema
Data items as behaviors

Data items are represented by actions that may be performed on that data

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;

a) An example of composed event trace

b) An architecture view
Reuse of schemas

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

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: (* use_S1 *)
ROOT S2: (* use_S2 *);
S1, Main SHARE ALL use_S1;
S2, Main SHARE ALL use_S2;  -- this also ensures that access to each stack is sequential

MAP S1 AS Stack_operation,
    pop FROM S1 AS pop,
    push FROM S1 AS push WITHIN Stack;

MAP S2 AS Stack_operation,
    pop FROM S2 AS pop,
    push FROM S2 AS push WITHIN Stack;
Integrating environment’s behavior

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: insufficient_balance *) FROM ATM_system ADD $y PRECEDES $x ;

COORDINATE (* $x: identification_fails *) FROM Customer,
    (* $y: id_failed *) FROM ATM_system  ADD $y PRECEDES $x ;
Architecture view on the component behavior

A view on the Customer root event behavior as UML Activity Diagram
a) An example of event trace (Use Case) for the ATM_withdrawal schema
b) An architecture view for the ATM_withdrawal schema
Architecture verification & validation

Advantages of Monterey Phoenix approach compared with the common simulation tools are as follows:

- Means to write **assertions** about the system behavior and tools to verify those assertions.
- **Exhaustive search** through all possible scenarios (up to the scope limit).
  - The **Small Scope Hypothesis**: most errors can be demonstrated on small examples.
- Integration of the architecture models with **environment models** for verifying system’s behavior on typical scenarios (Use Cases).
- **Event attributes**, like timing, can be used for non-functional requirements V/V.
- Assigning **probabilities** to certain events makes it possible to obtain statistical estimates for system behaviors.
- Scenarios derived from the model may be used for **implementation testing**
- Interactions of subsystems and environment can be used for detecting **emerging behaviors** of System of Systems
Architecture verification within limited scope

Testing:
A few cases of arbitrary size

Scope-complete:
All cases within a small bound
Brief comparison of MP with SysML and EFFBD activity diagrams

• MP enforces better structured control flow (no unrestricted goto's).
• MP has simple modularization mechanism (composite events) for scalability and readability.
• Extracting different views from MP model (traditional architecture box-arrow diagrams, UML Activity Diagrams, Nassy-Shneiderman diagrams, etc.).
• MP has powerful interaction abstraction. Separation of component's behavior from the interaction between components is an important feature for model reuse (can coordinate different sets of events without changing the model of the component itself).
• MP is focused on the "lightweight" verification – exhaustive trace generation, based on the Small Scope Hypothesis (executable architecture models).
• It is more feasible to perform different kinds of computations on particular instances of event traces; hence, the MP assertion language is more expressive than LTL or other temporal logic languages used in model checking. Neither SysML or EFFBD offer adequate assertion language at this time.
Implementation

- First MP prototype has been implemented as a compiler generating an Alloy relational logic model from the MP schema and then running the Alloy Analyzer to obtain event traces and to perform assertion checks.
- A prototype trace generator converts MP schemas into a C++ code, compiles and runs it. Generation speed reaches $10^4$ events/sec, the search space up to $10^{15}$ traces.
- On-line demo is available at [http://modeling.eagle6.com](http://modeling.eagle6.com)
- MP model checking tool implemented at the National University of Singapore by Dr. Jin Song Dong group.
Access Authentication Model in Eagle 6

Unique Use Cases from the one Authentication Model in MP
Use Case 1: User gets access after one unsuccessful attempt.
Use Case 2: User abandons access request after two unsuccessful attempts.
Using MP to Expose Latent Behavior Before It Happens

1. Behavior description for each system + abstract interaction specification

2. Scenario (use case) generation

3. Scenario visualization, inspection, and assertion checking

4. Modify the design to exclude unwanted scenarios

- e.g., "Are there any instances where a user can gain access to the system after three attempts?"
Publications available at

http://faculty.nps.edu/maugusto/

Questions, please?
Backup slides
Axioms

Mutual Exclusion of Relations

Axiom 1) \( a \text{ PRECEDES } b \implies \neg (a \text{ IN } b) \)
Axiom 2) \( a \text{ PRECEDES } b \implies \neg (b \text{ IN } a) \)
Axiom 3) \( a \text{ IN } b \implies \neg (a \text{ PRECEDES } b) \)
Axiom 4) \( a \text{ IN } b \implies \neg (b \text{ PRECEDES } a) \)

Non-commutativity

Axiom 5) \( a \text{ PRECEDES } b \implies \neg (b \text{ PRECEDES } a) \)
Axiom 6) \( a \text{ IN } b \implies \neg (b \text{ IN } a) \)

Irreflexivity for PRECEDES and IN follows from non-commutativity.

Transitivity

Axiom 7) \( (a \text{ PRECEDES } b) \text{ and } (b \text{ PRECEDES } c) \implies (a \text{ PRECEDES } c) \)
Axiom 8) \( (a \text{ IN } b) \text{ and } (b \text{ IN } c) \implies (a \text{ IN } c) \)

Distributivity

Axiom 9) \( (a \text{ IN } b) \text{ and } (b \text{ PRECEDES } c) \implies (a \text{ PRECEDES } c) \)
Axiom 10) \( (a \text{ PRECEDES } b) \text{ and } (c \text{ IN } b) \implies (a \text{ PRECEDES } c) \)

Event trace is always a directed acyclic graph.
Event grammar

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

$A:: (* B *);$ means an ordered sequence of zero or more events of the type B.

$A:: (B | C);$ denotes alternative

$A:: \{ B, C \};$ denotes a set of events B and C without an ordering relation between them
Alloy Analyzer is a good candidate for implementing the Phoenix Abstract Machine. This is a basic trace for Simple_transaction schema.
Example of refined trace for Simple_transaction. Some redundant PRECEDES relations have been added.
A model of system interacting with its environment and assertions about the system’s behavior.

\[
\text{root Weapon: } (* ( \text{Idle} \mid (\text{Weapon_On} \text{ Shoot} \\
\text{Recharge}) ) ) *)
\]

\[
\text{When } \{ \text{Weapon_hit} \Rightarrow \text{Repair Weapon} \};
\]

\[
\text{Shoot: } (\text{Hit} \mid \text{Miss});
\]

\[
\text{root Control: } (* \text{Generator_On} \text{ Radar_On} \\
\text{Monitoring Radar_off Generator_Off} *)
\]

\[
\text{When } \{ \text{Generator_hit} \Rightarrow \\
\text{GeneratorOff Repair Control}, \\
\text{Radar_hit} \Rightarrow \\
\text{Radar_Off Repair Radar_On Control} \};
\]
A model of system interacting with its environment and assertions about the system’s behavior.

```
root Enemy_missile: (* ( Approaching | Target_detected ) *) Boom
  When { Hit => };

Boom: ( Generator_hit | Radar_hit | Weapon_hit | Miss );
```
A model of system interacting with its environment and assertions about the system’s behavior.
(constraints)

Radar, Enemy_missile *share all* Target_detected;
Radar, Weapon *share all* Weapon_On;
Weapon, Enemy_missile *share all* Hit, Weapon_hit;
Control, Generator *share all* Generator_On, Generator_Off;
Control, Radar *share all* Radar_On, Radar_Off;
(Generator + Radar), Control *share all* Repair;
Control, Generator, Enemy_missile *share all* Generator_hit;
Control, Radar, Enemy_missile *share all* Radar_hit;
A view of the system’s components and environment interaction
Assertions about the system’s behavior.

Assertion 1.
\texttt{not exists Slice(Generator\_off, Radar\_Working);}

Assertion 2.
\texttt{not exists Slice(Generator\_off, Weapon\_On);}

The assertions above can be refuted on relatively small counterexamples of traces within a dozen events.

The following counterexample for Assertion 1 has been found by Alloy Analyzer in less than 2 sec on iMac workstation with 2.8 GHz processor.
Flight model

// main phases
//---------------------------------------------------------
ROOT Preflight: BoardAircraft
    FlightCheck
    DepartureClearance
    Pushback
    IssueGroundInstruction
    Taxi ;
ROOT Takeoff: Clear_for_takeoff
    Liftoff
    Handoff_to_TRACon ;
ROOT Departure: ChangeFrequency
    IssueClearances
    Handoff_to_ARTCC ;
ROOT EnRoute: IssueInstruction
    ( OceanicExtension | Follow_route )
    ChangeFrequency2
    Handoff_to_TRACon2 ;
ROOT Descent: Clear_descent
    Maneuver_toward_Airport ;
ROOT Approach: [ Hold ]
    Clear_approach
    Enter_approach_line
    Handoff_to_tower ;
ROOT Landing: Clear_landing
    Land
    Taxi_instruction
    Taxi_to_gate
    Disembark ;

// main actors
//-----------------------------------------------------
ROOT Passenger: BoardAircraft
    InsideCabin
    Disembark ;
ROOT Pilot:    FlightCheck
    Pushback
    Taxi
    Liftoff
    ChangeFrequency
    ( OceanicExtension | Follow_route )
    ChangeFrequency2
    Maneuver_toward_Airport
    [ Hold ]
    Enter_approach_line
    Land
    Taxi_to_gate ;
ROOT Controller: DepartureClearance
    IssueGroundInstruction
    Clear_for_takeoff
    Handoff_to_TRACon
    IssueClearances
    Handoff_to_ARTCC
    IssueInstruction
    Handoff_to_TRACon2
    Clear_descent
    Clear_approach
    Handoff_to_tower
    Clear_landing
    Taxi_instruction ;
## Flight model

// overlapping between actors and process phases
//-----------------------------------------------

<table>
<thead>
<tr>
<th>Actor Type</th>
<th>Process Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passenger, Preflight</td>
<td>SHARE ALL BoardAircraft;</td>
</tr>
<tr>
<td>Passenger, Landing</td>
<td>SHARE ALL Disembark;</td>
</tr>
<tr>
<td>Pilot, Preflight</td>
<td>SHARE ALL FlightCheck, Pushback, Taxi;</td>
</tr>
<tr>
<td>Pilot, Takeoff</td>
<td>SHARE ALL Liftoff;</td>
</tr>
<tr>
<td>Pilot, Departure</td>
<td>SHARE ALL ChangeFrequency;</td>
</tr>
<tr>
<td>Pilot, EnRoute</td>
<td>SHARE ALL OceanicExtension, Follow_route, ChangeFrequency2;</td>
</tr>
<tr>
<td>Pilot, Descent</td>
<td>SHARE ALL Maneuver_toward_Airport;</td>
</tr>
<tr>
<td>Pilot, Approach</td>
<td>SHARE ALL Hold, Enter_approach_line;</td>
</tr>
<tr>
<td>Pilot, Landing</td>
<td>SHARE ALL Land, Taxi_to_gate;</td>
</tr>
<tr>
<td>Controller, Preflight</td>
<td>SHARE ALL DepartureClearance, IssueGroundInstruction;</td>
</tr>
<tr>
<td>Controller, Takeoff</td>
<td>SHARE ALL Clear_for_takeoff, Handoff_to_TRACon;</td>
</tr>
<tr>
<td>Controller, Departure</td>
<td>SHARE ALL IssueClearances, Handoff_to_ARTCC;</td>
</tr>
<tr>
<td>Controller, EnRoute</td>
<td>SHARE ALL IssueInstruction, Handoff_to_TRACon2;</td>
</tr>
<tr>
<td>Controller, Descent</td>
<td>SHARE ALL Clear_descent;</td>
</tr>
<tr>
<td>Controller, Approach</td>
<td>SHARE ALL Clear_approach, Handoff_to_tower;</td>
</tr>
<tr>
<td>Controller, Landing</td>
<td>SHARE ALL Clear_landing, Taxi_instruction;</td>
</tr>
</tbody>
</table>
Flight model view as Activity Diagram

Activity Diagram

2013/09/20   astah® Evaluation
Use Case (instance of event trace) generated from MP model on Eagle6