System Architecture Specification Based on Behavior Models

Mikhail Auguston, Clifford Whitcomb
Computer Science Department, Systems Engineering Department
Naval Postgraduate School
Monterey, California, USA

ICCRTS 2010
WWW.NPS.EDU
One of the major concerns in Systems Architecture design is the question of the behavior of the system.

We suggest an approach for building system behavior models based on the concepts of event and event traces.

This yields executable architecture models and the ability to reason about system’s behavior.
Objectives

- To make architecture models **executable** on an abstract machine, so that it becomes possible early in the system development phase to perform **testing and verification** of the top level system design and to enable automatic **assertion** verification.

- To provide a method and tools for extracting **multiple views** from the architecture models (e.g. DODAF views).

- To provide the method for system **stepwise refinement** from the top level architecture models to the detailed design and implementation models, supported by tools for sanity checks and refinement **consistency checks**.

- To provide formalism for specifying system’s **environment models**, so that the system architecture can be tested and verified in the interaction with its environment, supporting the system **safety assessment** by identifying the hazard states that may emerge from such interaction.
• An approach to formal software system architecture specification based on **behavior models**.

• The behavior of the system is defined as a set of events (**event trace**) with two basic relations: **precedence** and **inclusion**.

• The structure of event trace is specified using **event grammars** and other constraints organized into schemas.

• The **schema** framework is amenable to stepwise architecture refinement, reuse, composition, visualization, and application of automated tools for consistency checks.
**Event** - any detectable action in system’s or environment’s behavior

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

- Graph grammar
- Both basic relations are partial orderings
- Event trace is always directed acyclic graph
Example of an Event Grammar

Shooting_competition:: {* Shooting *};

Shooting:: (* Shoot *);

Shoot:: Fire ( Hit | Miss);
Example of event trace

Shooting_Competition

Shooting

Shoot_a_gun

Fire

Miss

Hit

iccRTs 2010
WWW.nps.edu
Simple_transaction

_______________
root TaskA:: Send;
root TaskB:: Receive;
root Transaction:: Send Receive;
_______________
TaskA, Transaction share all Send;
TaskB, Transaction share all Receive;

If $X$, $Y$ are root events, and $Z$ is an event type
$X$, $Y$ share all $Z \equiv \{ v: Z | v \text{ IN } X \} = \{ w: Z | w \text{ IN } Y \}$

The schema defines a set of event traces, i.e. the behavior model.

Example of an event trace
Multiple_synchronized_transactions

root TaskA:: (* Send *);
root TaskB:: (* Receive *);
root Connector:: (* Send Receive *);

TaskA, Connector share all Send;
TaskB, Connector share all Receive;

Example of an event trace
**Basic(S)**. This set contains only traces, which satisfy all schema’s constraints, and have only events and relations imposed by the schema’s grammar rules and Axioms.

The process of generating traces from Basic(S) defines the **semantics** of the schema S.

**Schema is executable** if there exists an Abstract Machine able to generate all traces from Basic(S). The schema represents instances of behavior (event traces), in the same sense as Java source code represents instances of program execution.
Alloy Analyzer is a good candidate for implementing the Phoenix Abstract Machine. This is a basic trace for Simple_transaction schema.
Predicate

\[
\text{CONNECTED}(X, Y) = \text{exists a } ((a \text{ IN } X) \text{ and } (a \text{ IN } Y))
\]

may be used to extract simple diagrams from the schemas:

```
Simple_transaction
```

```
TaskA ----> Transaction ----> TaskB
```
Client_server

root Client:: {* Request *};
root Server:: {* Provide *};
root Connector:: Initialize
    {* ( Request Provide) *} Close;

Client, Connector share all Request;
Server, Connector share all Provide;
The User schema represents the environment behavior in which the Calculator operates.

User

Use_calculator:: (* Perform_calculation *);

Perform_calculation::
    Enter_number
    Enter_operator
    Enter_number
    Enter_number
    Request_result;

Enter_number:: (+ Press_digit_button +) ;
System behavior model

Calculator

Calculator_in_action:: (* Single_calculation *);

Single_calculation:: Get_number Get_operator Get_number
  IF (Get_operator.operation == ‘+’) THEN
    / Single_calculation.result =
    Get_number[1].value + Get_number[2].value; /
  ELSE
    / Single_calculation.result =
    Get_number[1].value – Get_number[2].value; /
  Show_result;

Get_number:: / Get_number.value= 0; /
  (* Get_digit
    / Get_number.value =
    Get_number.value * 10 + Get_digit.value;/ *) ;

Show_result:: /show_result(ENCLOSING Single_calculation.result );/ ;
The following schema defines the interaction between the User and the Calculator by establishing a connection between events in the environment and in the system.

**Connection**

Press_digit_button:: /Get_digit.value = Press_digit_button.value;/

Get_digit;

Enter_operator:: / Get_operator.operation = Enter_operator.operation;/

Get_operator;

Request_result:: Show_result;

The model of a calculator interacting with the environment.

**merge User, Calculator, Connection;**

Calculator, Connection share all Get_digit, Get_operator, Show_result;

User, Connection share all Press_digit_button, Enter_operator, Request_result;
A counterexample for assertion:

\textbf{not exists} \text{Slice(Generator\_off, Radar\_Working);}