The Möbius Framework - with an application to Stochastic Process Algebras

William H. Sanders

Performability Engineering Research Group
Coordinated Science Laboratory and Electrical and
Computer Engineering Department
University of Illinois at Urbana-Champaign
whs@crhc.uiuc.edu

http://www.crhc.uiuc.edu/PERFORM

IFIP Working Group Meeting - Stenungsund, Sweden
July 2001

Project funded in part by the Motorola Center for High-Availability System Validation, under the umbrella of the Motorola Communications Center, and National Science Foundation Next Generation Software Program

Model-Based System Validation

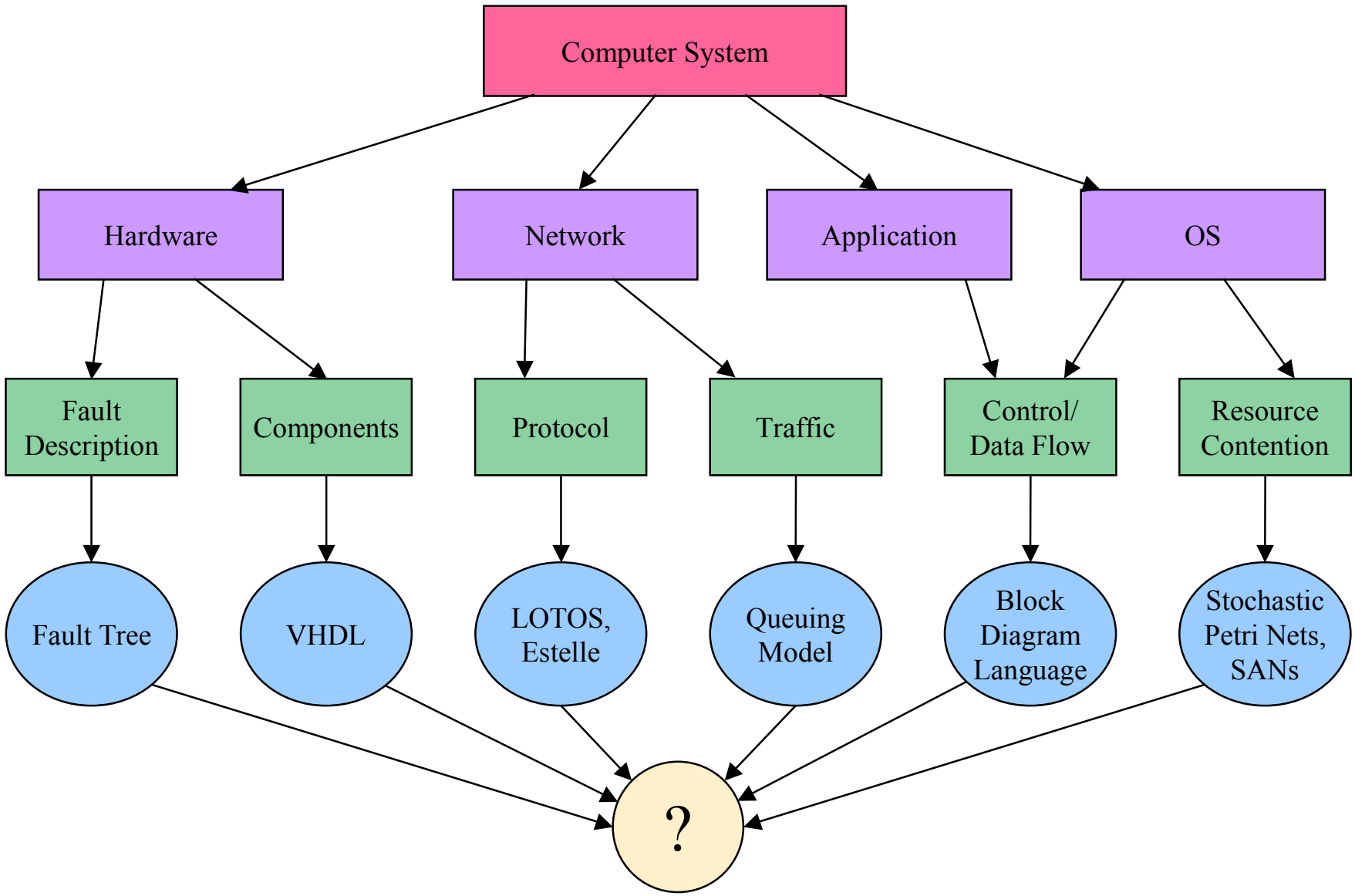
- Goal: Determine, early in the design phase, the performance, dependability, and performability of large-scale distributed systems.

- Experience:
  - Building high-level CAD tools for such evaluation since 1984: METASAN, *UltraSAN*, and Möbius
  - *UltraSAN* modeling software licensed to approximately 250 academic sites.
  - *UltraSAN* commercially licensed to several companies, including Intel, Bell Communications Research, US West Advanced Technologies, IBM, Motorola, Vysis, 3M, NASA, Bell Northern Research, and Technicatome
  - Current research directed toward efficiently solving models of very large systems, using scalable ways to compose and connect models together (Möbius Project).

Integrated Modeling Frameworks are Needed!

- No single formalism is best for representing all parts of a distributed computing/communication system
  - Computer hardware, networks, protocols, and applications each call for a different representation
  - Even within a “class” of application, different industry segments use very different ways of representing a particular design
- No single solution method is adequate to solve all models
  - Discrete-event simulation is efficient in many cases, but is extremely slow in others (e.g., significant, but rare events (like faults and buffer overflows), or extreme system complexity)
- Research in new modeling methods and tools is significantly hampered by the close link between model specification and model solution methods, and the closed nature of existing tools

Modelers Need Heterogeneous Models



Related Work

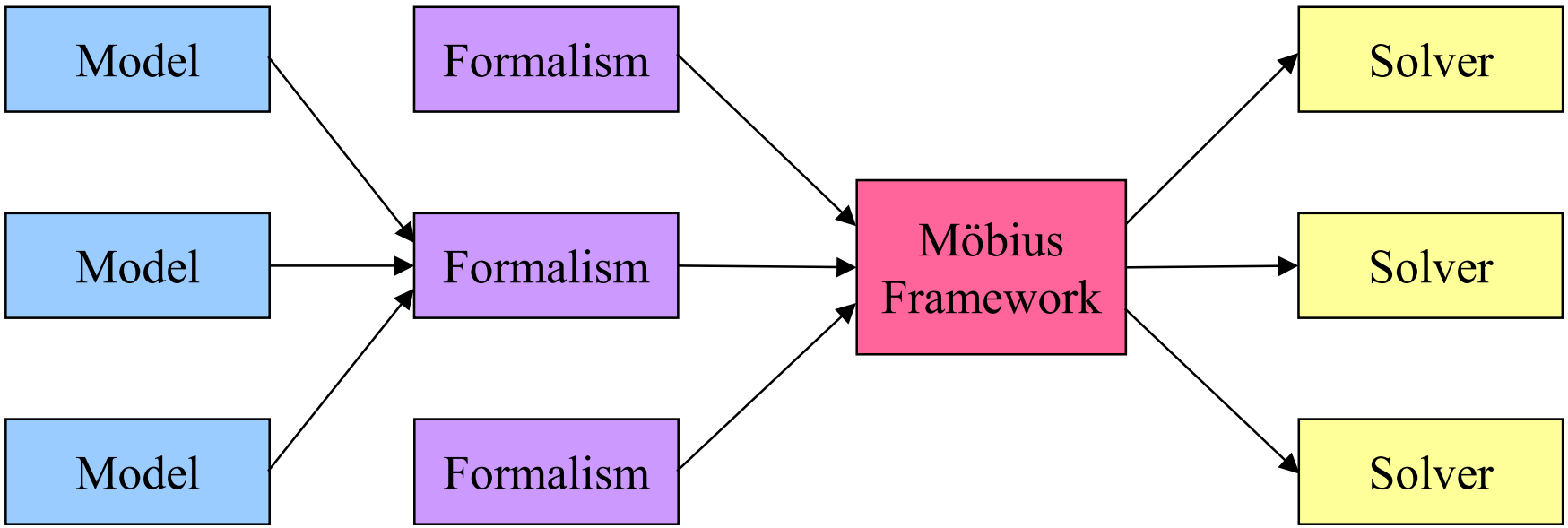
- Single formalism, multiple solution methods
  - e.g., DyQN-Tool+, GreatSPN, *UltraSAN*, TANGRAM-II
- Integrated software environment
  - ISME, IDEAS, Freud
- Multi-formalism multi-solution
  - SHARPE, SMART

Möbius Project Research Goal

- Development of tools to predict the performance, dependability, and performability of distributed computing/communication systems
  - Such systems are complex combinations of:
    - Computing hardware
    - Networks
    - Operating systems
    - Software

  (First goal was *not* to prove logical system properties, although this may be possible within framework (and what I would like to learn more about at this workshop!))
- We believe such tools can be realized by:
  - Developing a framework/tool that supports multiple modeling formalisms, at multiple levels of detail and abstraction, and multiple model solution methods
  - Developing new model representation and solution methods (within the framework, and implemented in the tool) that scale well with increasing system complexity

Möbius Framework



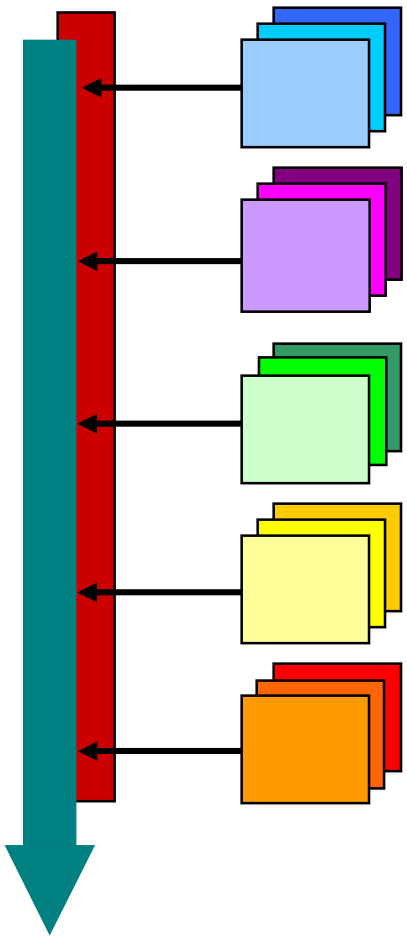
- Model: An abstract representation of some system
- Formalism: A modeling language
- Framework: A “language” in which modeling languages may be expressed

The Möbius Framework ...

- Expresses most existing modeling languages (except some simulation languages)
- Retains the ability for efficient solution
- Facilitates homogeneous modeling
- Is a vehicle for researching new model composition, connection, reward specification, and solution methods

Model Categories in the Möbius Framework

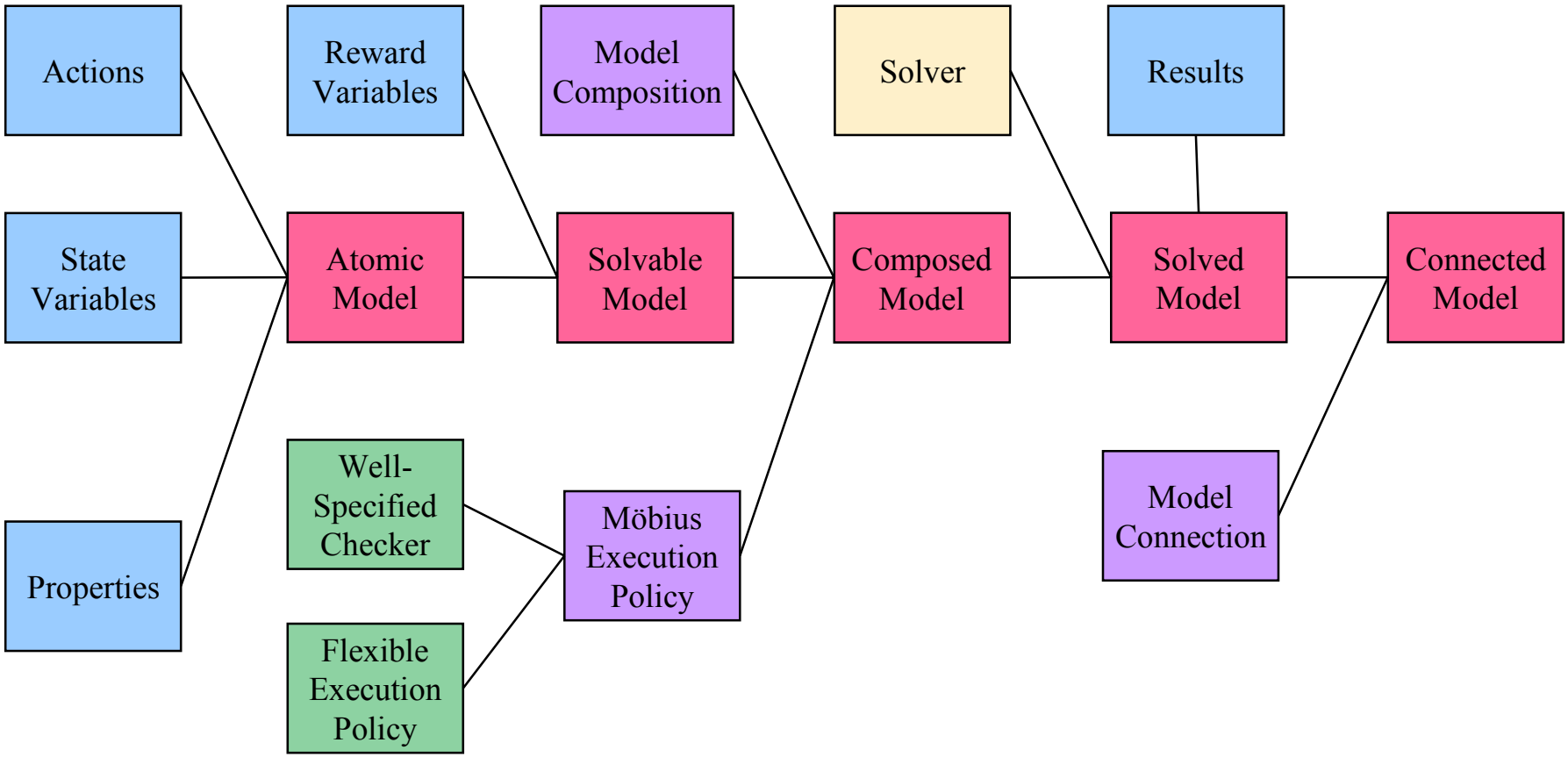
Submodel Interaction



Model Specification

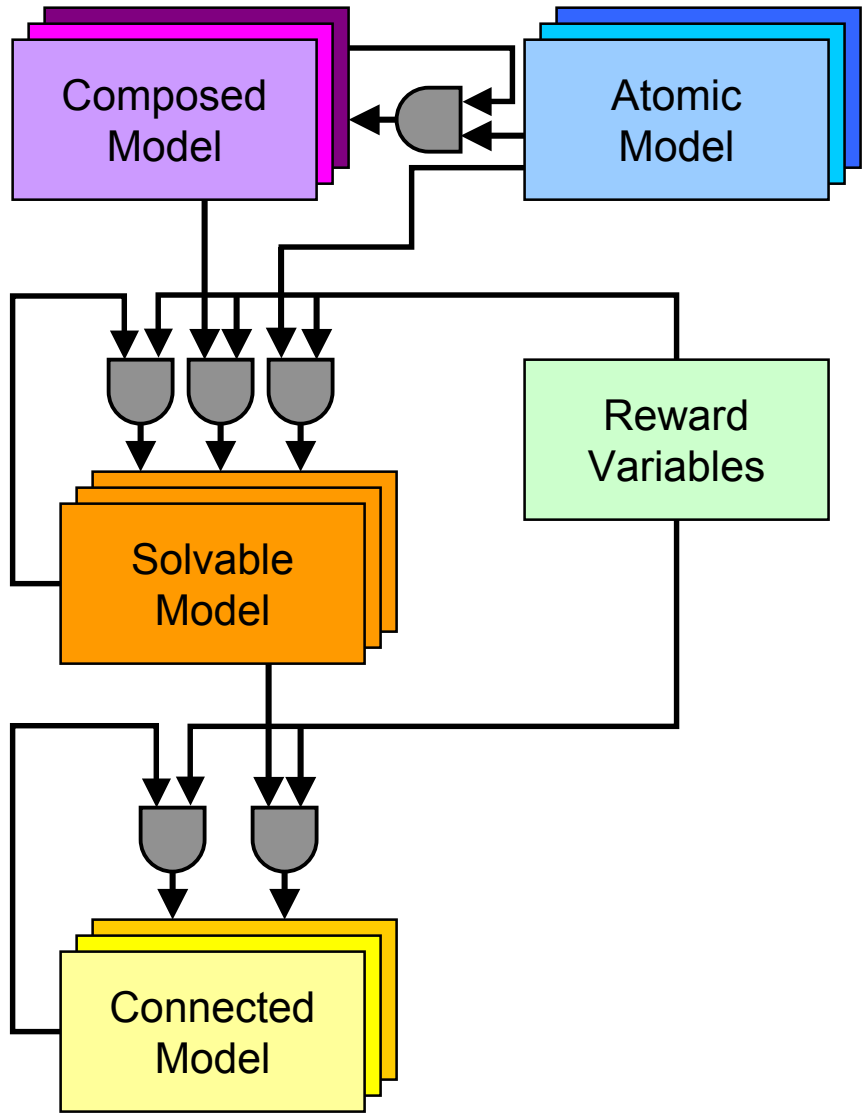
Framework Component	Example Formalisms
Atomic Model	DSPN, GSPN, Markov chain, Queueing Network, SAN, SAN, SPA, other SPN extensions, Domain-specific formalism
Composed Model	Graph interconnection Kronecker Composition (SAN), Replicate/Join, SPA Domain-specific formalism
Solvable Model	Rate/Impulse reward variables Path-based reward variables Domain-specific formalism
Connected Model	Fixed-point governor Acyclic model composer
Study Specifier (generates multiple models)	Range and Set Variation Non-linear optimizer

Möbius Framework Components

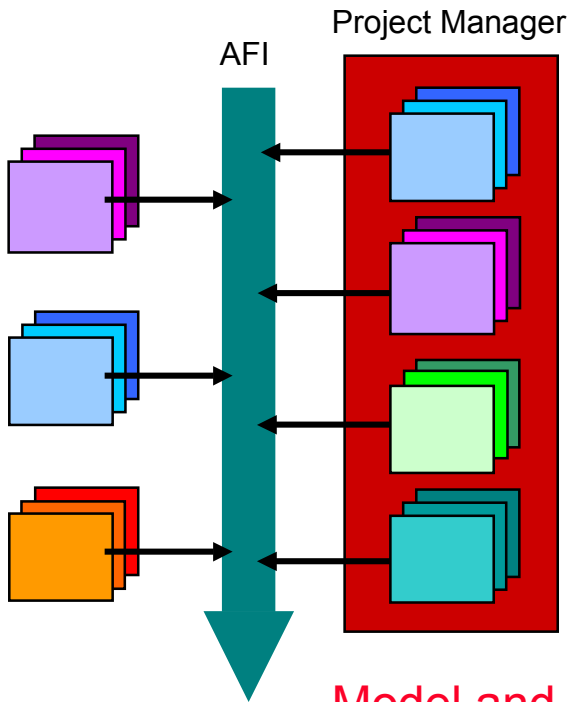


Model Construction Process

- Models expressed in different framework components can be combined to form larger models
  - One or more atomic and/or composed models form a composed model
  - A atomic, composed, or solvable model, together with reward variables, form a solvable model
  - One or more solvable or connected models, together with reward variables, form a connected model
- The model construction process retains the structure of each constituent model, facilitating efficient solution.



Abstract Functional Interface Facilitates Interaction of Models and Solution Engines



Simulation, State-Space Generation, and Analytic/Numerical Solvers

Model Specification, Composition, Reward Definition, and Connection Formalisms

Abstract Functional Interface Interactions

Model and Solver Editor Interactions

- The abstract functional interface allows models to affect each other and be acted on by solvers without understanding model semantics
- A project manager maintains consistency when constructing new models, performance/dependability variables, and studies from existing models

Technical Details: Abstract Functional Interface (AFI)

The abstract functional interface is a set of methods defined on a set of "base classes" that all models must implement to interact within the Möbius framework.

- The AFI acts as a communication interface between multiple, possibly heterogeneous, models.
- Solvers communicate with models by calling methods in the AFI.
- The AFI can preserve special properties of models implemented in particular formalisms since it only specifies an interface, not internal representation, unlike the integrated framework approach which converts models to a universal formalism.
- The AFI makes Möbius extensible, by hiding formalism-specific details from interacting models or solvers, making is possible to add modeling formalisms that interact with models described in other formalisms and solvers, without changing them.

Model Support of the Abstract Functional Interface: State Variables, Actions, and Properties

- Formally, a model in the Möbius framework is a set of "state variables," a set of "actions," and set of "properties"
- State variables "contain" information about the state of the system being modeled
  - They have a type, which defines their "structure"
  - They have a value, which defines the "state" of the variable
- Actions prescribe how the value of state variables may change as a function of time
- Properties specify characteristics that may effect the solution of a model
- Other models and solvers may request information regarding or change to state of a model's state variables, actions, and groups via the abstract functional interface
- The format of this information is determined by the structure of a model's state variables and attributes of its actions

State Variable Specification in the Möbius Framework

- The set of all state variable types is denoted *T*.
- The set *T* is constructed by repeated application of the following rules:

$Z \in T.$

$\Re \in T.$

$S \in T.$ Set of names of state variables

If $t \in T$, then $2^t \subset T.$ Restriction of a type

If $t \in T$, then $2^t \in T.$ Sets of types (unordered)

If $t_1, t_2 ..., t_n \in T$, then $t_1 \times t_2 \times ... \times t_n \in T.$ Ordered lists of types

Action Specification in the Möbius Framework

- An action's attributes specify how and when it changes state:

$$\text{Enabled}: \Sigma \rightarrow \{\text{true}, \text{false}\}$$

$$\text{Delay} : \Sigma \rightarrow (\Re \geq \rightarrow [0,1])$$

$$\text{Effort} : \Sigma \rightarrow (\Re \geq \rightarrow [0,1])$$

$$\text{Interrupt}: \Sigma \rightarrow \{\text{true}, \text{false}\}$$

$$\text{Rank} : \Sigma \rightarrow Z^{+} \cup \{\textit{Undefined}\}$$

$$\text{Weight} : \Sigma \rightarrow \Re^{\geq} \cup \{\textit{Undefined}\}$$

$$\text{Complete} : \Sigma \rightarrow \Sigma$$

$$\text{Policy}: \Sigma \rightarrow \{\text{DDD}, ..., \text{PPP}\}$$

Model State

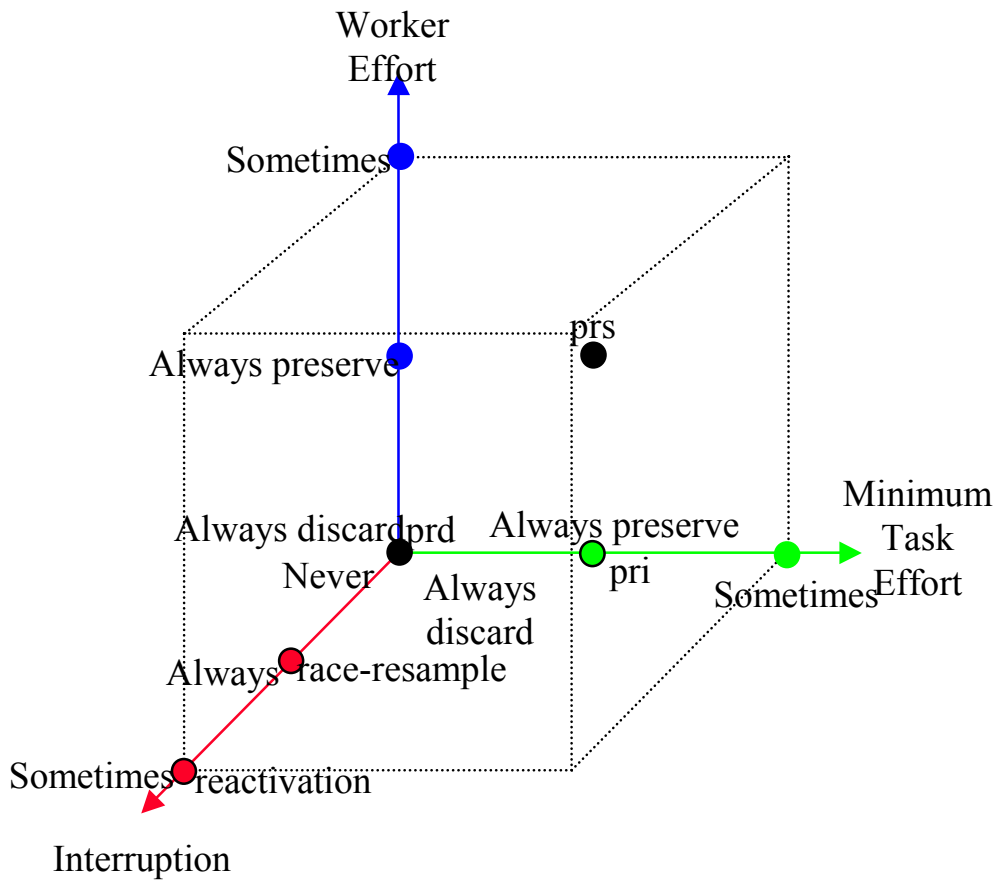
- A model's state is the state of all its:
  - State variables, and
  - Actions

- Generally speaking, the set of all possible states of a model is uncountable, since certain state components are continuous

- If all action time distributions in a model are exponential, and if all marking dependent actions depend only on the model's current state, then the continuous component of action state can be ignored, and the resulting behavior is Markov

Model Execution in the Möbius Framework

- Models change state by the firing of actions, according to formalism-specific rules hidden by AFI
- The execution policy is defined on a per action and per state basis
  - Actions can be interrupted, reset, continue with a different distribution, or continue with time same distribution
  - Generalizes and unifies existing execution policies
  - Details can be found in technical report ...



Model Composition

- Model composition formalisms permit the construction of models from other models by sharing state variables or actions between constituent models
- New model implements AFI, just like an atomic model
- State variable sharing can be of two types:
  - Equivalence sharing, where a state variable or "part" of state variable from one model is identified with a state variable or "piece" of state variable in another model (information flow is bi-directional)
  - Functional sharing, where the state of a state variable in one model is defined to be a function of another submodel's state (information flow is one-direction)
- Two complete or partial state variables can be equivalently shared if:
  - Their structure is the same (as defined by the state variable specification syntax presented earlier - in short, they are of the same type).
  - They have the same initial value

Model Connection

- Model connection formalisms permit the construction of model solutions from a set of models by exchanging “results” between the models
- The abstract functional interface provides the infrastructure necessary to build connection formalisms, but none are currently implemented
- More work needs to be done in this area
- Potentially, “results” can be:
  - The mean and/or variance of a the a performance variable
  - The density or distribution function of a performance variable (e.g. exponential-polynomial distribution)
  - Some automatically-constructed more-abstract model representation, e.g.,
    - Hidden Markov model
    - Markov-modulated Poisson processes

An Example Atomic Modeling Formalism: PEPA

PEPA, developed at the University of Edinburgh, is a stochastic process algebra intended for performance evaluation.

PEPA differs from other Möbius formalisms in several ways.

Issues:

- How to build an AFI for PEPA
- What is a useful and intuitive notion of state?
- How to construct an equivalence sharing relationship
- Can we construct new composed model formalisms?

What are Process Algebras?

- View as a programming language for describing models
- Central aims:
  - Compositionality – a methodology for systematically building the complex from the simple
  - Concurrency – *built-in for free*, as a consequence
- Prominent representatives:
  - For research: CCS [Milner], CSP [Hoare]
  - For applications: LOTOS (ISO Std. 8807) e.g. the study of communications protocols

```
process Spec :=
 enter.exit.Spec
endproc


process Peterson[p1_enter,
p1_exit, p2_enter, p2_exit] :=

hide
 flag1,flag2,…

in
 (Proc[…] <flag1,…> Proc[…])

endproc

…
```

What is PEPA?

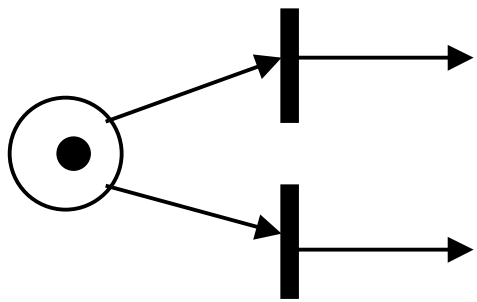
- PEPA stands for "*Performance Evaluation Process Algebra*"
- Primitive process algebra *actions* become timed PEPA *activities*:

```
enter.exit.Spec  <---> (enter,r).(exit,s).Spec
```

- r and s are the parameters of exponentially distributed random variables which determine the time it takes for each activity to complete
- What are the primitives for building PEPA models?

PEPA Combinators

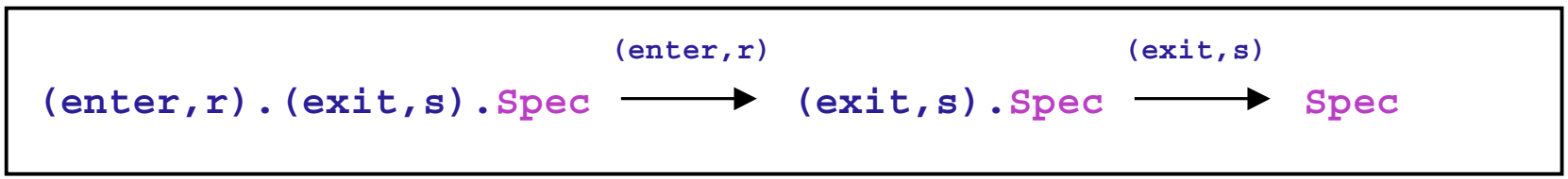
1. Prefix: given an activity `(a,r)`, and a process `P`, `(a,r).P` is a process which performs the activity `(a,r)` and then becomes `P`
2. Choice: `P + Q` is a process which expresses competition between `P` and `Q`. It is analogous to the following SAN fragment:



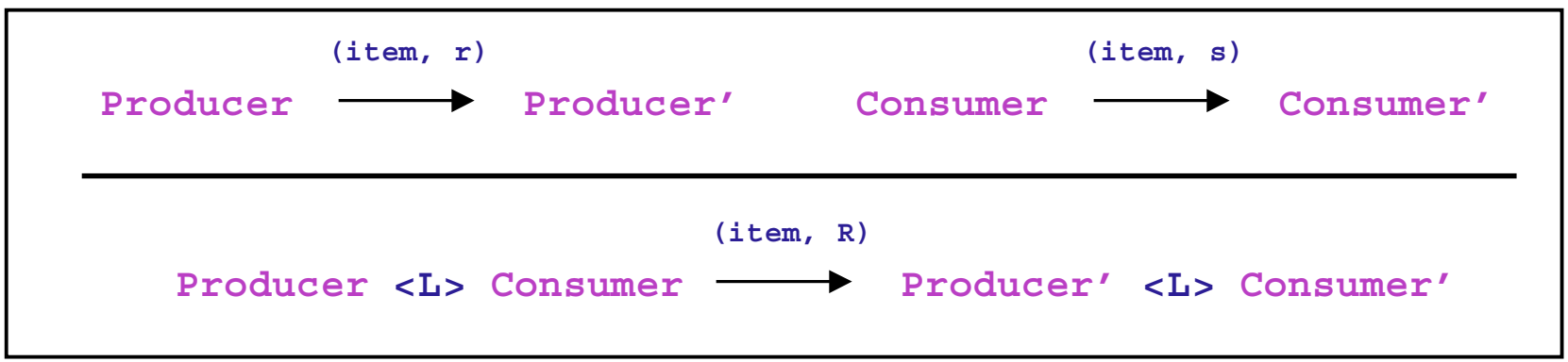
3. Cooperation: given processes `P` and `Q`, and a set of activity names `L`, the process `P <L> Q` expresses the parallel composition of `P` and `Q` with synchronization on `L` activities; c.f. increasing the number of tokens in a SAN place
4. Hiding: given a process `P`, and a set of activity names `L`, the process `P/L` hides those names in `L` from further interaction

The Underlying Model

- The model evolves from state to state by performing activities:



- Rules are used to calculate the behavior of processes from their subcomponents. Assume `item` is in `L`; then



- Leads to direct simulation, or an analytical solution

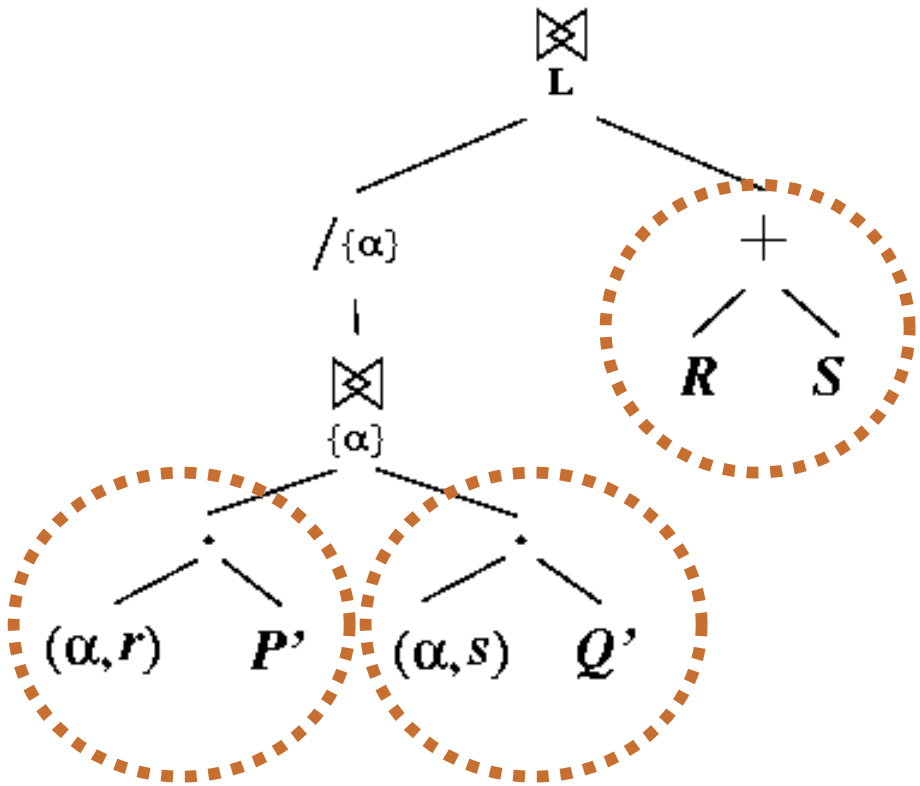
Representing PEPA State

- A PEPA process evolves over time according to operational transition rules
- We consider cyclic PEPA only, meaning that no dynamic combinator (prefix and choice) ever governs a static combinator (cooperation or hiding)
- This ensures that the structure of the PEPA term does not grow unboundedly over time
- Not a serious restriction : cyclic PEPA is always used in “real-life” models anyway

Let’s illustrate with an example

Representing PEPA State : Example

```
System := ((P <α> Q)/{α}) <L> (R + S)
   P   := (α,r).P'
   Q   := (α,s).Q'
```



Representing PEPA State (ctd.)

- We could enumerate the states of each dynamic (circled) component; **but** there is no *compelling* way for a partner model to use this data
- Instead, we use partial information about the state of each dynamic component, and..
- We require the modeler to specify this information

This is done by extending PEPA to provide PEPA$_k$.

PEPA$_k$ adds value-passing to PEPA.

PEPA$_k$: Adding Value Passing

We extended PEPA with the following features:

- Formal Parameters: variables now have an arity; may be instantiated with parameters e.g. a defining eqn:

$$P[x,y] := (\alpha, r).P'$$

- Guards: the behavior specified by a process expression is only enabled if the guard evaluates to true in the current state
- Value-Passing: values may be communicated between dynamic components using activities

We decided additional features would not have increased the usefulness of the language.

Theory is well-known! e.g. LOTOS (no stoch.), MLOTOS, EMPA…

PEPA$_k$: An Example

```
Queue[m,s,n]    :=  if (m < n) then (in,u).Queue[m+1,s,n]
                 +  if (m > 0) then (out,v*min(s,m)).Queue[m-1,s,n]
```

We provide a PEPA semantics in order to understand PEPA$_k$ processes. For this example, it generates a set of definitions over s and n including:

$$Queue_{0,s,n} := (in,u).Queue_{1,s,n}$$

$$Queue_{i,s,n} := (in,u).Queue_{i+1,s,n} + (out,v*i).Queue_{i-1,s,n} \quad 0 < i < s$$

$$Queue_{i,s,n} := (in,u).Queue_{i+1,s,n} + (out,v*s).Queue_{i-1,s,n} \quad s <= i < n$$

$$Queue_{n,s,n} := (in,u).Queue_{n-1,s,n}$$

- No surprises here!
- To understand activity-based values passing, we translate $(in?x,u)$ to a sum over activities (in_i,u) ; then we require input activities to always be matched by an output over the lifetime of the process
- With above condition, we can show that we are not working with a new process algebra – this reduces the proof burden

Mapping to the AFI : State Variables

- For AFI state variables, we provide the modeler with PEPA*k* dynamic component process parameters
- The modeler may then create an equivalence sharing relationship with a partner model, for example, a SAN
- If the partner changes a shared state variable, the PEPA*k* process parameter's value is changed, and the future behavior of the PEPA*k* process may be changed. However the behavior is altered in a **meaningful** way!

  For example, `Queue[m,s,n]` would export 3 state variables – simple!

Mapping to the AFI : Actions

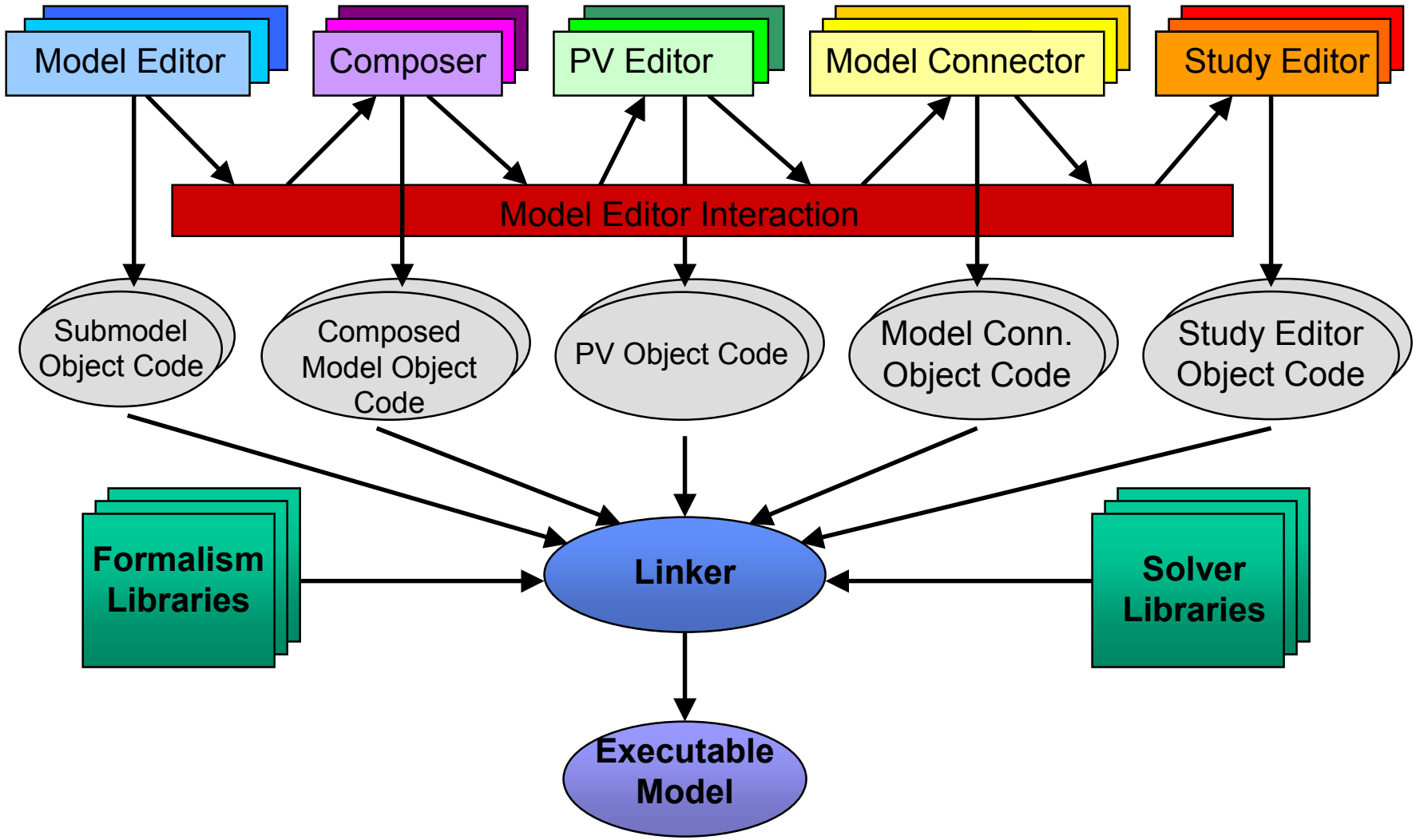
- AFI actions may be used to specify impulse rewards on models
- For AFI actions, we make available all possible combinations of activities enabled due to cooperations between static PEPA$_k$ components
- We could choose all possible activities from each dynamic component; this cuts down proliferation of actions, but loses modeling power (no way to specify impulse as a result of a particular cooperation)
- We could choose all activities that may be enabled over the lifetime of a process, but this cannot be calculated without exploring a state space

Our proposal is a **good compromise!**

Implications

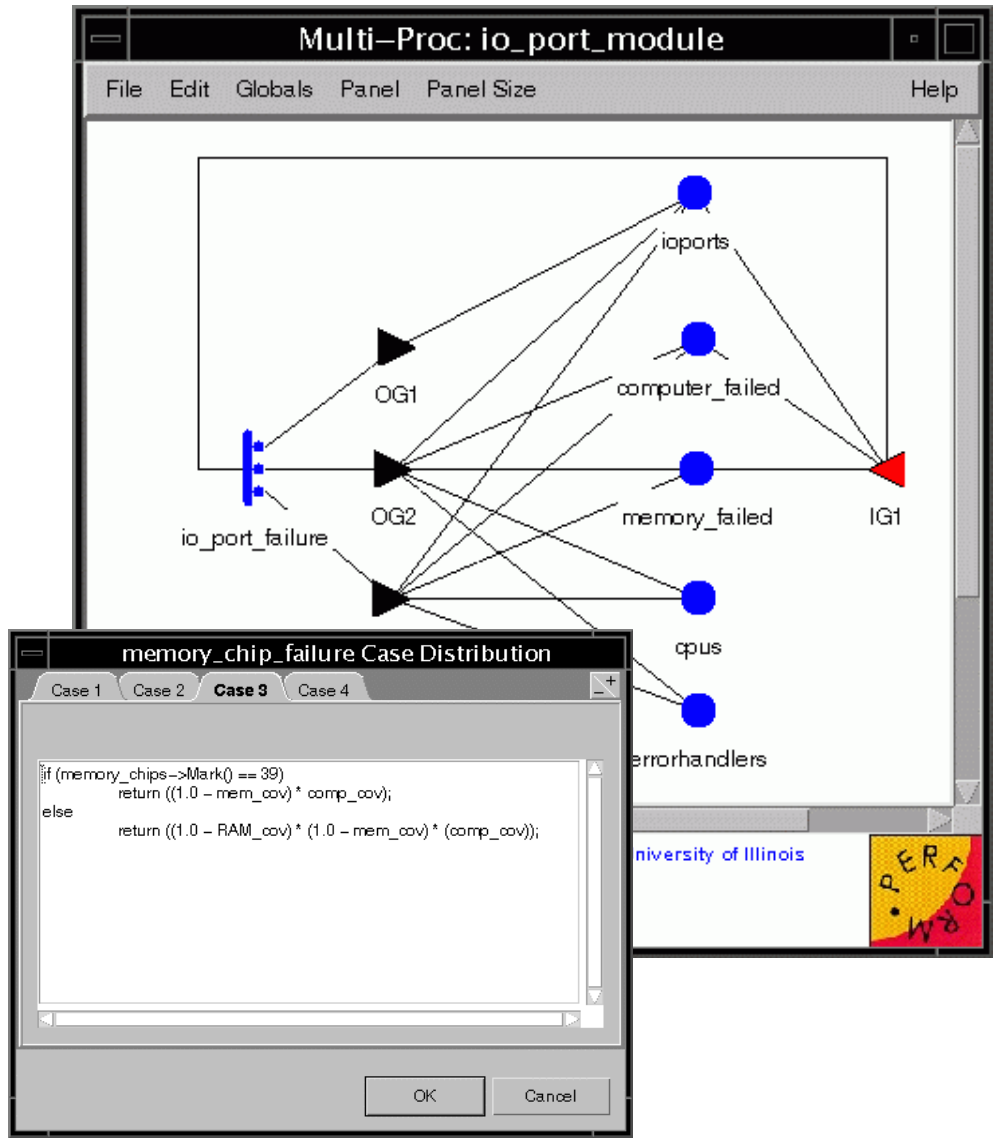
- The *UltraSAN*/Möbius modeler may now use PEPA$_k$ models; they may be composed with any other *UltraSAN*/Möbius formalism; may have rewards specified over them; and may be solved analytically or by simulation
- What happens to process algebra equivalences in an equivalence sharing relationship?
- Can we construct a new composed model formalism that employs **both** equivalence sharing and more traditional process algebra action sharing? To what extent can we exploit model symmetries **and** "observational" equivalence?

Möbius Tool Architecture



Atomic Model Construction

- Each atomic model editor permits the specification of a particular atomic model formalism -- editors can be graphical or textual.
- Möbius toolkit provides Java building blocks for editor construction, easing editor implementation.
- Each editor must generate model representation that can be "executed" by Möbius solvers; note that the representation can be formalism specific.
- Together, the code emitted by the editor and formalism "library" must implement the AFI to models written in the formalism.

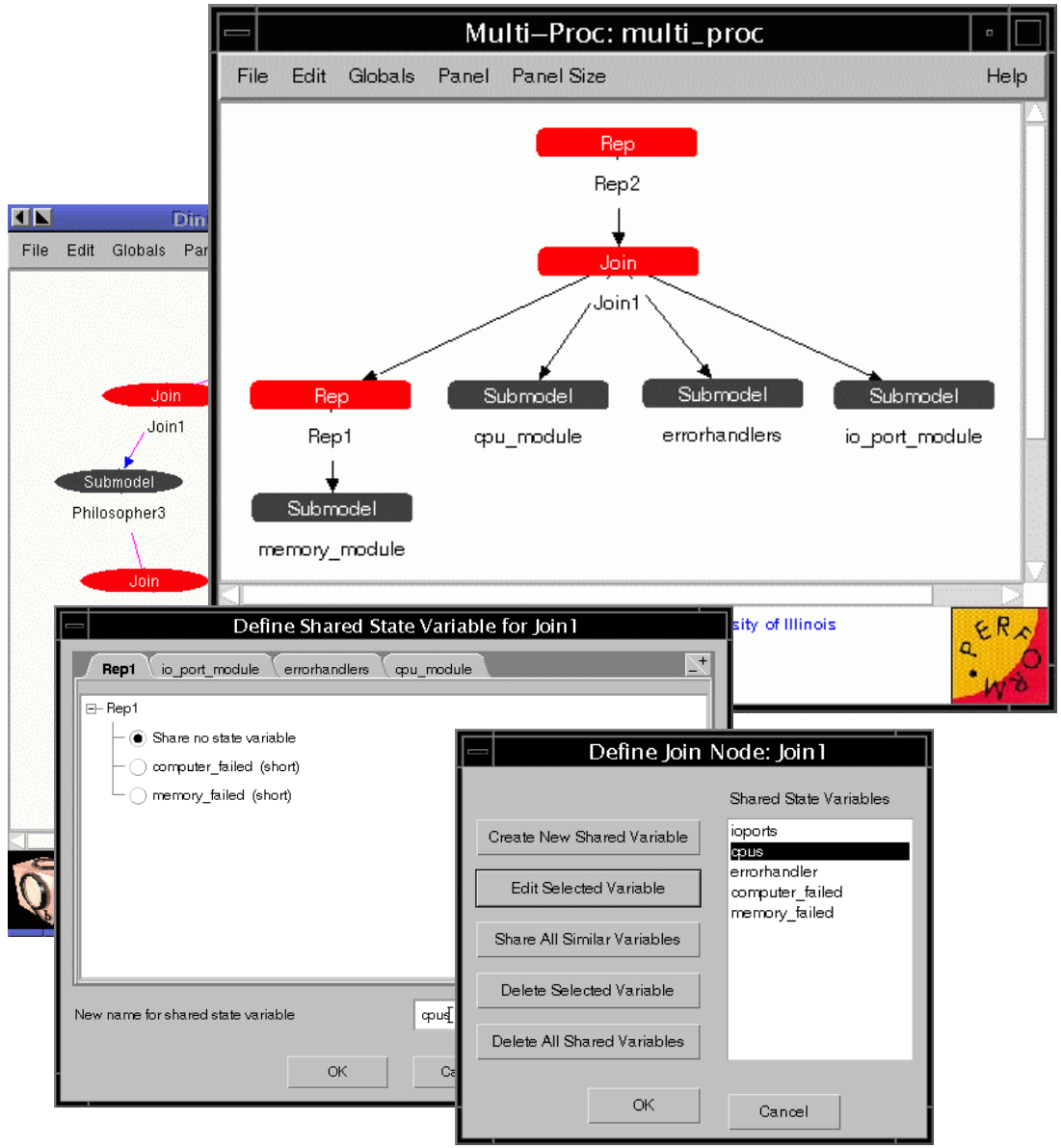


Implementation of Abstract Functional Interface as Base Class - Example Action Base Classes

bool Enabled()	Determines whether the action is enabled in the current state.
double Weight()	Weights are used to determine the probability of selecting an action from the set of enabled actions in the current state
double Rate	Returns the rate with which an exponentially timed action fires.
bool ReactivationPredicate()	Determines whether an action is reactivatable.
Bool ReactivationFunction()	Determines if an action, whose ReactivationPredicate is true, should restart after a state change in which the action is still enabled.
double SampleDistribution()	Samples the action's distribution and returns the action's time to completion.
double* ReturnDistributionParameters()	Returns the set of distribution parameters. The number of parameters is determined by the action distribution function.
BaseActionClass* Fire ()	Defines how the action changes the state of the model.

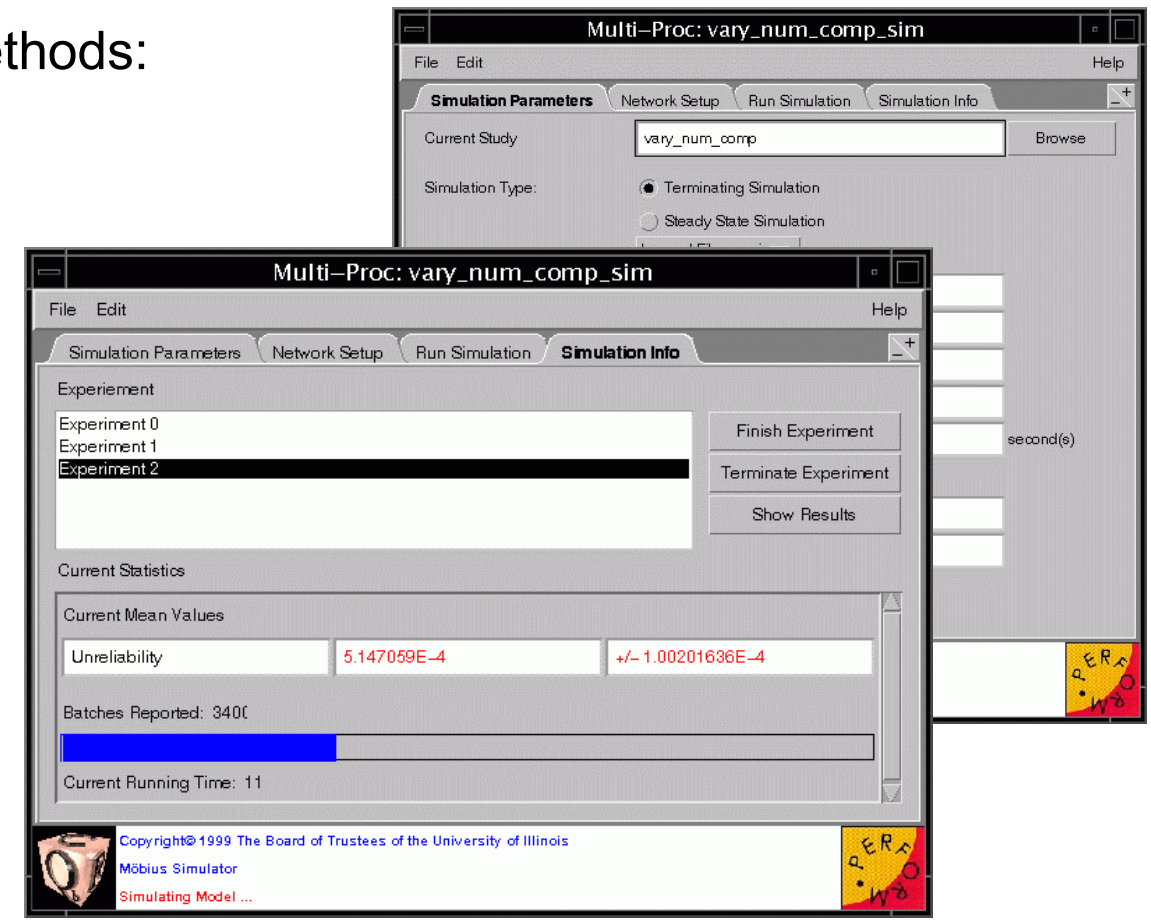
Composed Model Construction

- Composed model formalisms take models (atomic or composed) as input, and generate a model.
- Generated model implements AFI (just like an atomic model) so composed model can be further composed.
- Special properties of composition formalism (e.g. symmetry) are hidden by AFI, so can be automatically exploited by solvers and applied when only partially present in a model.
- AFI allows composed submodels to interact with one another (e.g., by sharing state variables) without understanding details of other submodel formalisms.



Simulator Use of Base Classes

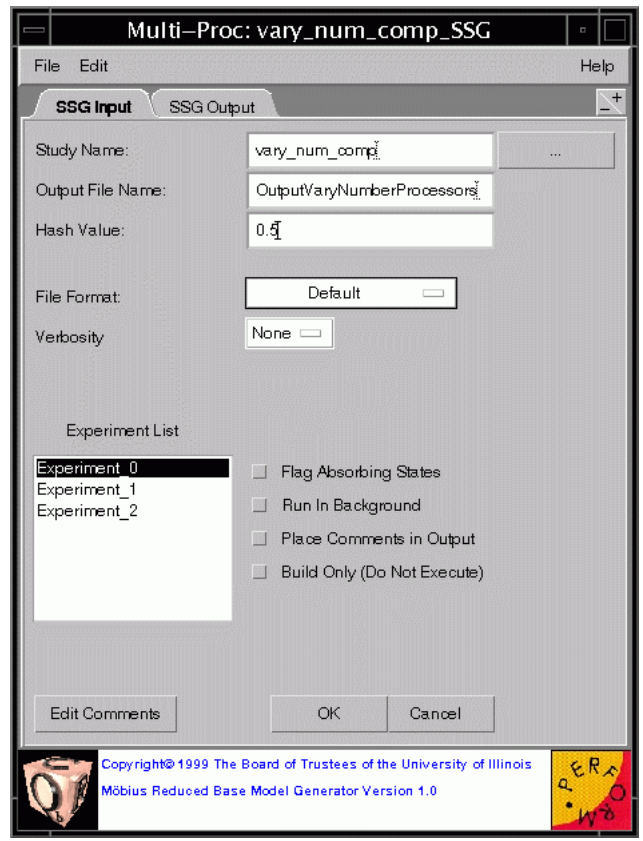
- The simulator interacts with a solvable model using abstract functional interface; Simulator does not know what formalism or formalisms it is simulating!
- Example base-class methods:
  - Base Model
    - CurrentState( )
    - SetState( )
    - ListOfActions( )
  - Base Action
    - Enabled( )
    - Fire( )



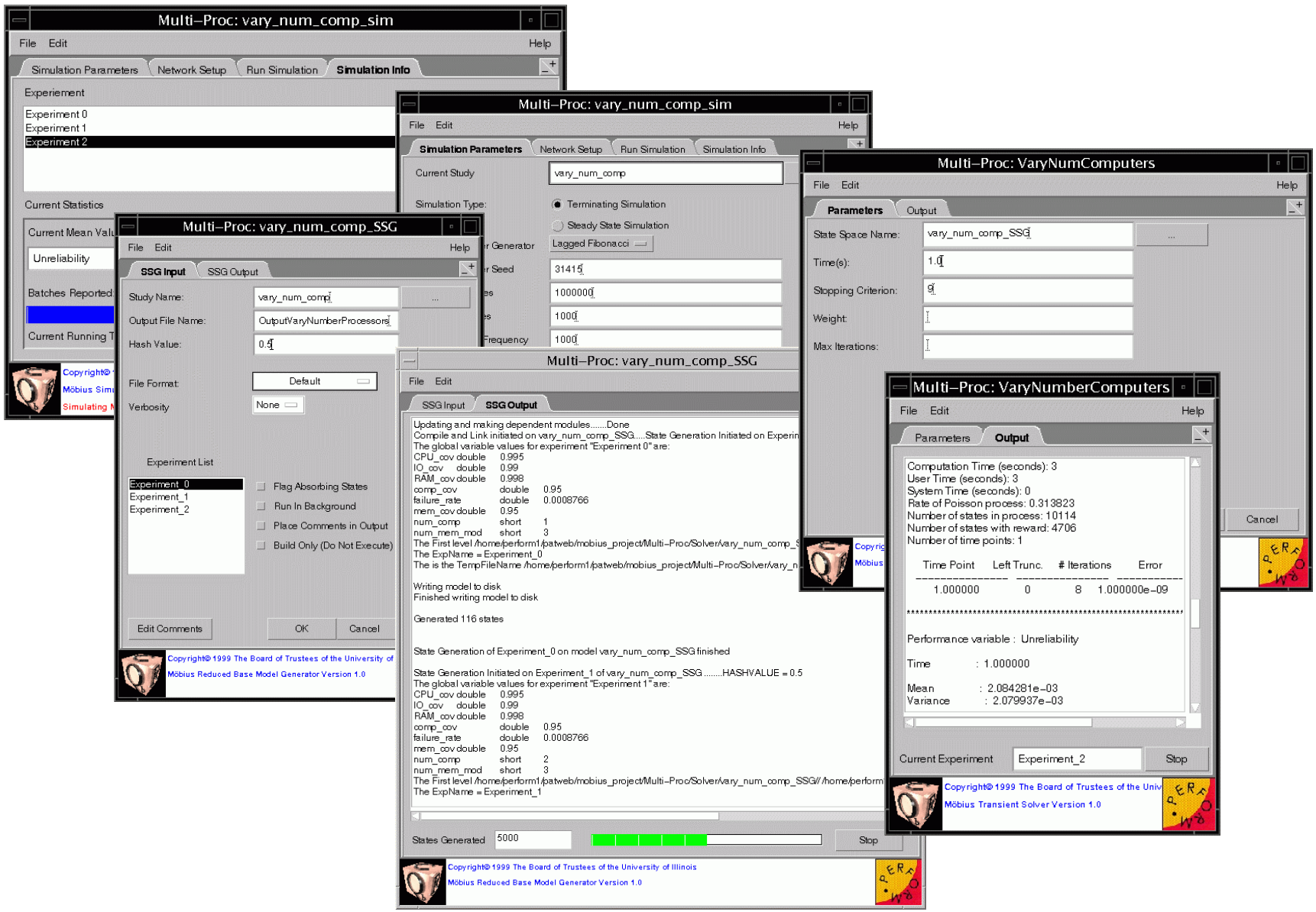
State-Space Generator Use of Base Classes

```
States = ∅
NewStates = TheModel.CurrentState();
ActionSet = TheModel.ListOfActions();
While(NewStates != ∅)
   TheModel.SetState(NewStates.getState())
   EnabledActions = ∅
   For all Action ∈ ActionSet
      If(Action.Enabled())
         EnabledSet = EnabledSet + {Action}
   End For
   While(EnabledSet != ∅)
     EnabledAction = EnabledSet.getAction()
     EnabledSet = EnabledSet – {EnabledAction}
     EnableAction.Fire()
     If(TheModel.CurrentState() ∉ States)
        NewStates = NewStates +
              {TheModel.CurrentState()}
   End While
   States = States +
              {TheModel.CurrentState()}
End While
```

- Generic “Fire” method changes the model’s state in a formalism-specific way
- Base class methods StateSize() and CompareState() allow state-space generator to manage state space in a generic fashion



Model Solution Engine Graphical User Interfaces



Möbius Project Status

- Completed definition and implementation of abstract functional interface.
- Completed 3 Atomic Model formalisms: Balls and Buckets, PEPA stochastic process algebra, Stochastic Activity Network
- Implemented 2 Composition formalisms:Replicate/Join, Graph
- Implemented one PV specification method: Reward Variables
- Completed range, set, and Design of Experiment Editors
- Developed results database, and integrated with formalisms and solvers
- Developed new connection method that uses distribution sharing
- Completed multiple solution methods and integrated them into Möbius: Steady-state simulator, Terminating simulator, State-space generator, Uniformization and adaptive uniformization transient solvers, Iterative steady-state solver (SOR), Direct steady-state solver (LU Decomposition), Deterministic/ Exponential steady-state solver