Foundations for Model-Based Design

Janos Sztipanovits
ISIS, Vanderbilt University
janos.sztipanovits@vanderbilt.edu

MPSOC 2005
Margaux, France
July 11, 2005

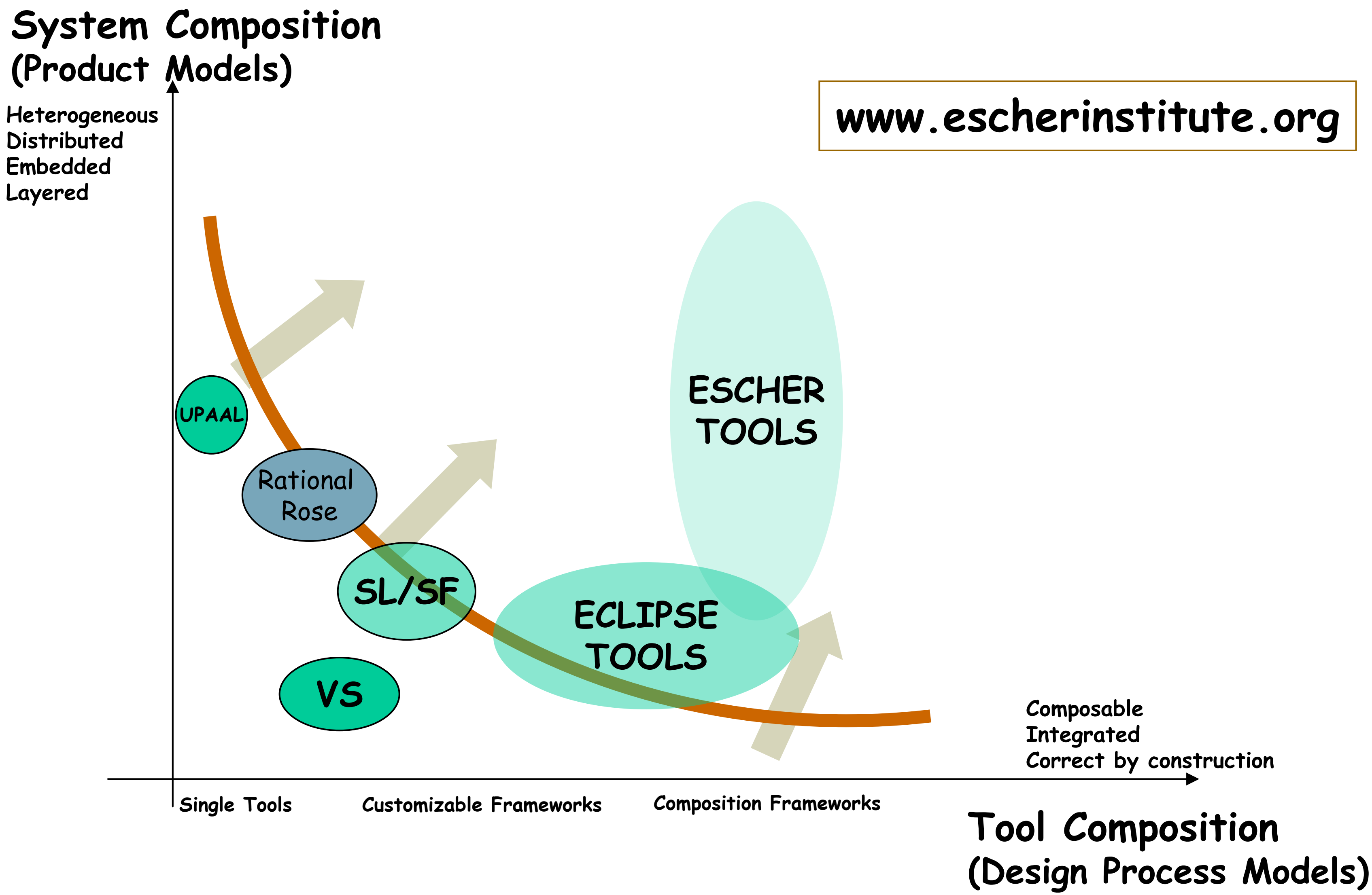
Content

- Introduction to model-based design
- System Composition Dimension
  - Layers
  - Approaches
  - Languages
- Tool Composition Dimension
  - Layers
  - Building Tool Chains
- Metamodeling and Metaprogrammable Tools
- Semantics

Goal and Approaches

- Building increasingly complex networked systems from components
  - Naïve "plug-and-play" approach does not work in embedded systems (neither in larger non-embedded systems)
  - Model-based software design focuses on the *formal representation, composition, analysis and manipulation of models* during the design process.
- Approaches with differences in focus and details
  - MDA: Model Driven Architecture
  - MDD: Model-Driven Design
  - MDE: Model-Driven Engineering
  - **MIC: Model-Integrated Computing**

Two Dimensions of MIC



Content

- Introduction to model-based design
- System Composition Dimension
  - Layers
  - Approaches
  - Languages
- Tool Composition Dimension
  - Layers
  - Building Tool Chains
- Metamodeling and Metaprogrammable Tools
- Semantics

System Composition Dimension: Core Modeling Aspects

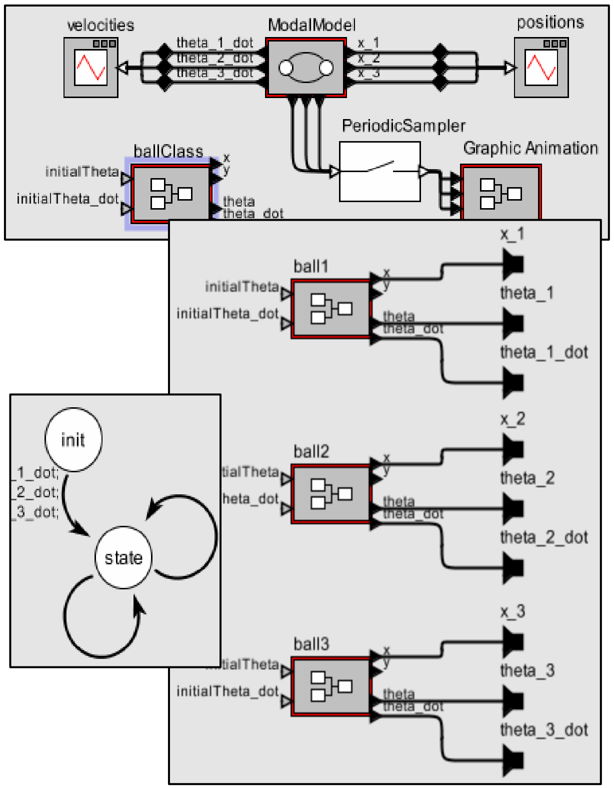
Component Behavior	Modeled on different levels of abstraction: • Transition systems (FSM, Time Automata, Cont. Dynamics, Hybrid), fundamental role of time models • Precise relationship among abstraction levels • Research: dynamic/adaptive behavior
Structure	Expressed as a system topology : • Module Interconnection (Nodes, Ports, Connections) • Hierarchy • Research: dynamic topology
Interaction	Describes interaction patterns among components: • Set of well-defined Models of Computations (MoC) (SR, SDF, DE,…) • Heterogeneous, but precisely defined interactions • Research: interface theory (time, resources,..)
Scheduling / Resource Allocation	Mapping/deploying components on platforms: • Dynamic Priority • Behavior guarantees • Research: composition of schedulers

Examples for Research Approaches

	Ptolemy II (Lee, UCB)	Metropolis (ASV[1], UCB)	IF (Sifakis, Verimag)
Component Behavior	Java Code/ Behavioral Models	Process (Hierarchical, Active Components)	Process (Hierarchical Timed Automaton)
Structure	Hierarchical Module Interconnection	Netlists (port, interface, connection)	Dynamically Created Channels
Interaction	Heterogeneous Models of Computation + Directors	Medium (port, parameter, useport)	Asynchronous Interactions: - P2P - Unicast - Multicast
Scheduling / Resource Allocation		Scheduler (port, parameter)	Dynamic Priorities

1 Alberto Sangiovanni-Vincentelli

Modeling Formalisms Are Different



Ptolemy II

```
medium IntM implements IntWriter,IntReader,IW,IR,IC,IS,IN {
  int storage, space, n;

  IntM(){ space = 1; n = 0;  }

  update void writeInt(int data) {
    await (space>0; this.IW, this.IS; this.IW)
      await (true; this.IC, this.IS, this.IN; this.IC){
        space = 0; n = 1;
        storage = data;
      }
  }

  update int readInt() {
    await (n>0; this.IR, this.IN; this.IR)
      await (true; this.IC, this.IS, this.IN; this.IC){
        space = 1; n = 0;
        return storage;
```

```
process Y {
  port IntReader port0;
  port IntReader port1;
  port IntWriter port2;
  ...
  void thread() {
    int z;
    while(true) {
      await {
        (port0.n()>0 && port1.n()>0;
         port0.IntReader, port1.IntReader;
         port0.IntReader, port1.IntReader)
         {  z = foo(port0.readInt(),port1.readInt()); }
      }
      port2.writeInt(z);
  } }
  int foo(int x, int y) { ... }
}
```

Metropolis

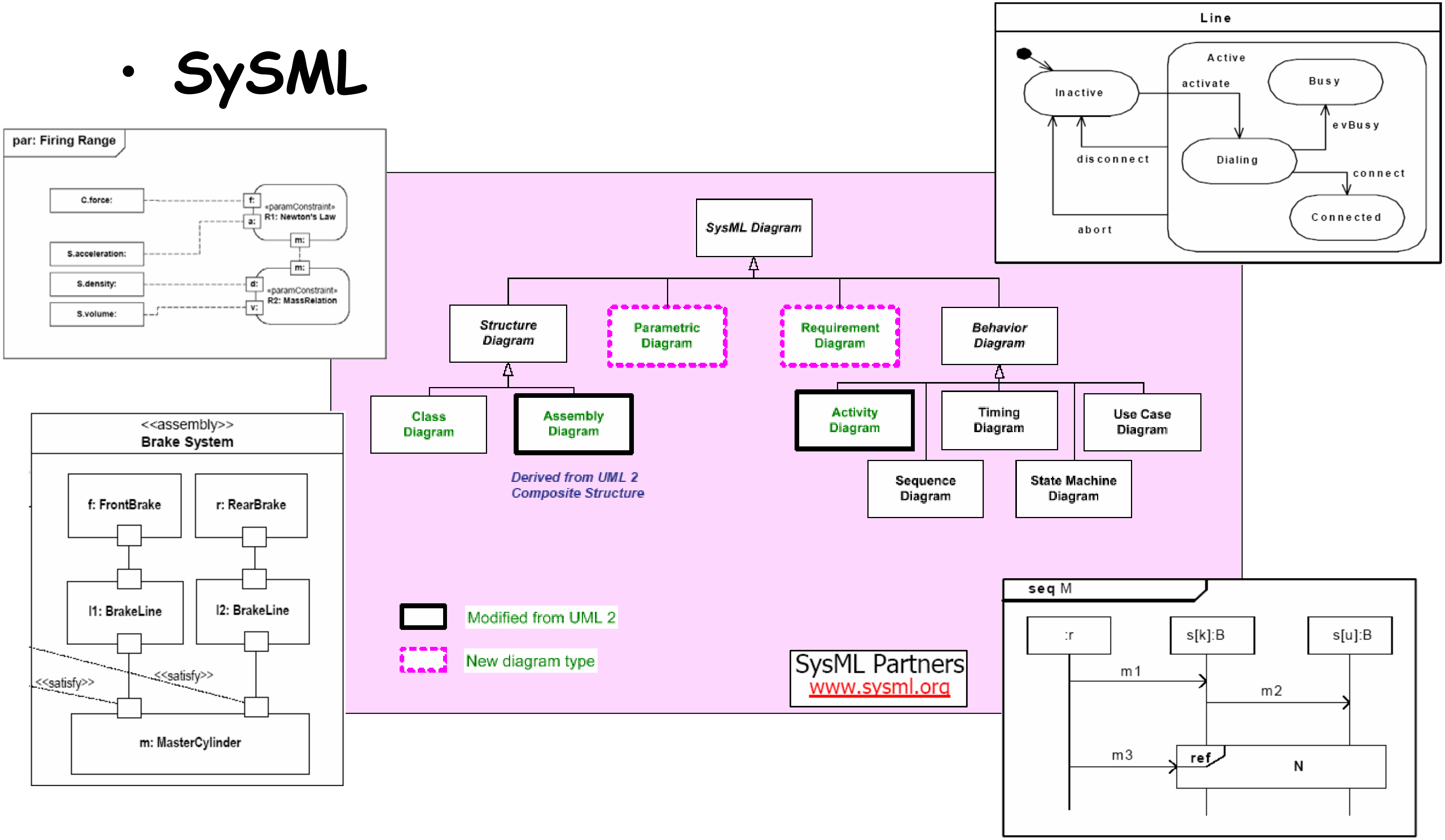
```
state send;
  output sdt(self,m,b) to {receiver}0;
  set t:=  10;
  nextstate wait_ack;
endstate;

state wait_ack;
  input ack(sender,c);
  ...
  when 10 <t<20 ;
  ...
endstate;
```

IF

Emergence of Modeling Language Standards

- SySML



- Others (UML-2; RT-UML, SLML, AADL,…)

Current Status of System/SW Modeling Languages

- The number of new standards is growing driven by competing consortiums and .org-s
- Intended scope ranges from "unified" to "specific".
- Many views them as programming languages
  - Wait for the "Unified One" to ensure reusability of tools
  - Slow down deployment because of the lack of standards
  - Wait for executable models
- Modeling and analysis tools are not integratable (closed camps emerge protected by a "standard").
- **Semantics is largely neglected or left to undocumented interpretations of tool developers.**

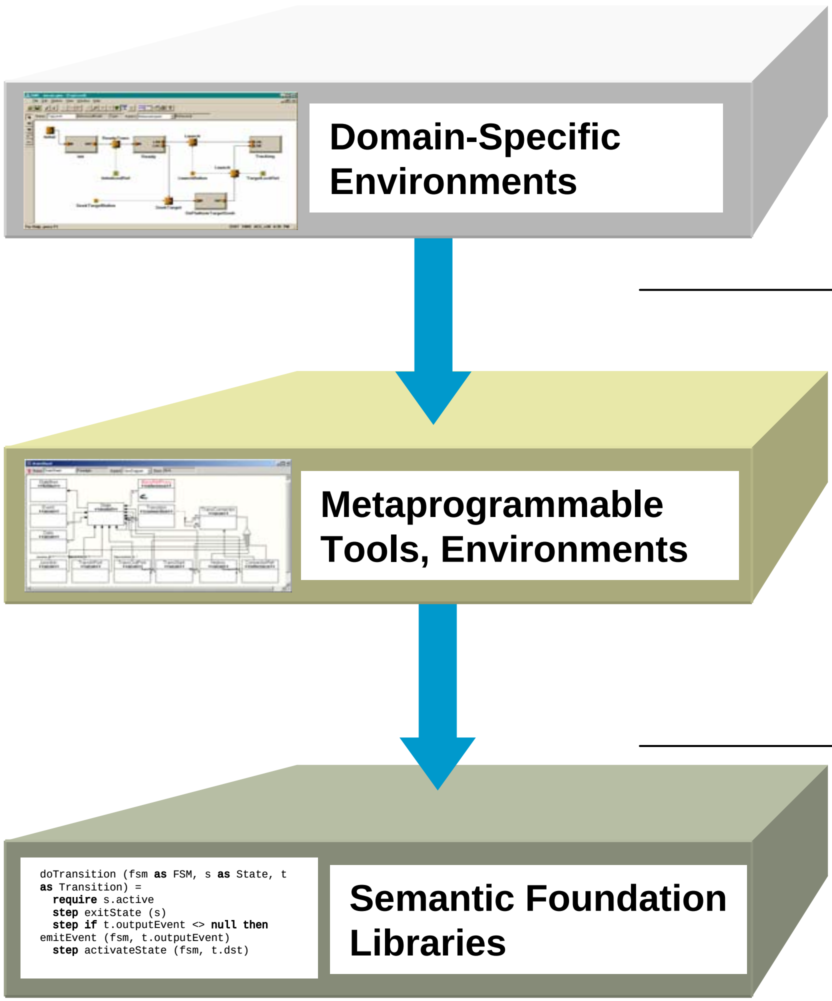
Trends in Modeling Languages

- Increasing acceptance of metamodeling and Domain-Specific Modeling Languages based on standard metamodels (**Meta Object Facility**, **MOF**)
- Emergence of **metaprogrammable tools**
- Desire for solving the "**semantics problem**"
- Better understanding of the role of precise **model transformations** in model-based generators and in building domain-specific tool chains from reusable tools

Content

- Introduction to model-based design
- System Composition Dimension
  - Layers
  - Approaches
  - Languages
- Tool Composition Dimension
  - Layers
  - Building Tool Chains
- Metamodeling and Metaprogrammable Tools
- Semantics

Tool Composition Dimension: Core Modeling Aspects



Modeling Domain Specific Design Flows: Examples in MIC:

- ECSL - Automotive
- ESML - Avionics
- SPML - Signal Processing
- CAPE/eLMS – Learning Technology

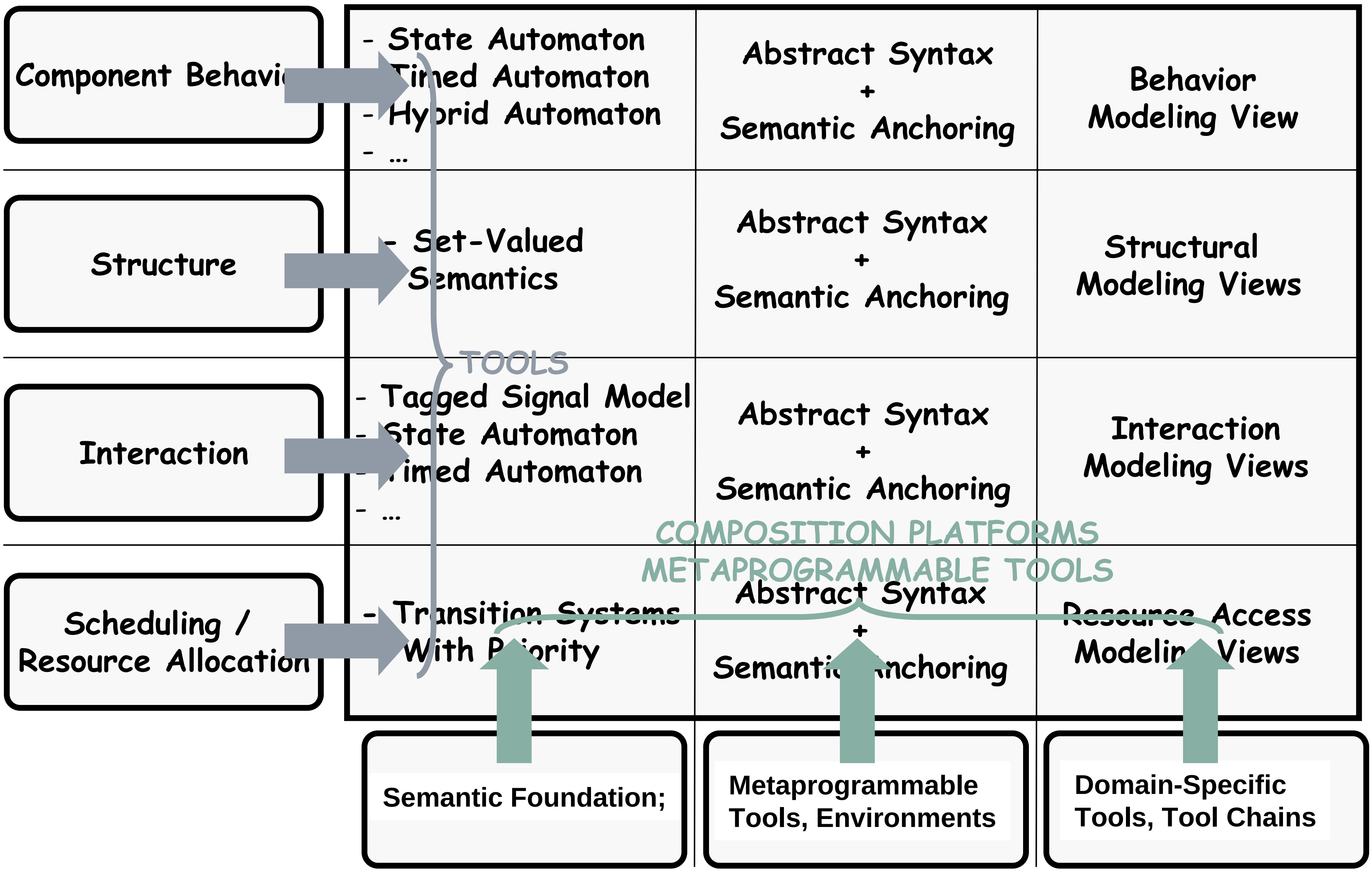
Metamodeling and Metaprogrammable Tools: (mature or in maturation program)

- **GME (Generic Model Editor)**
- **GReaT (Model Transformation)**
- **OTIF (Tool Integration Framework)**
- **UDM (Universal Data Model)**
- **DESERT (Design Space Exploration)**
- **GME-MOF/Meta (Metamodeling Env-s)**

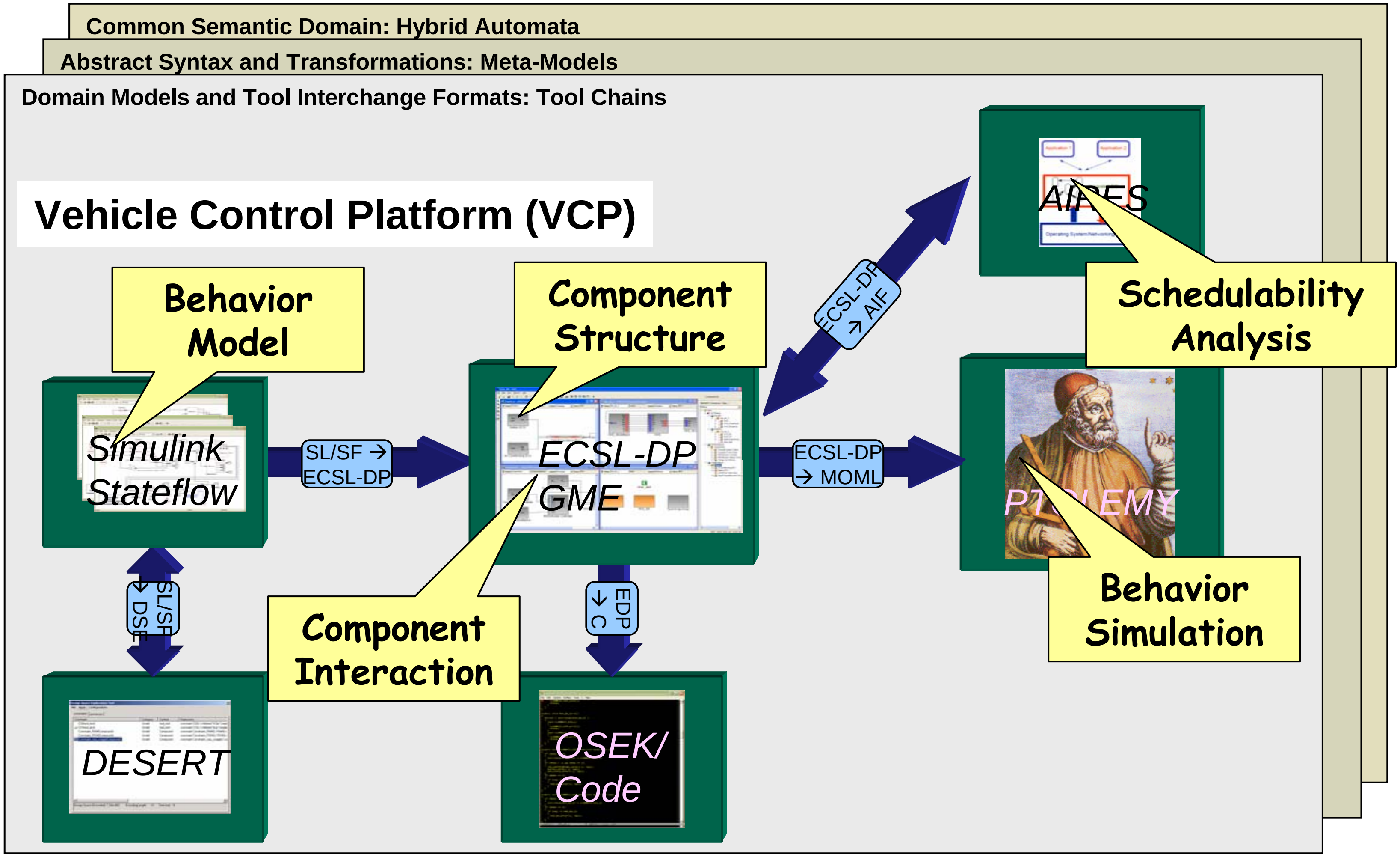
Modeling Semantics (work in progress):

- Semantic “Units”
- Semantic Anchoring

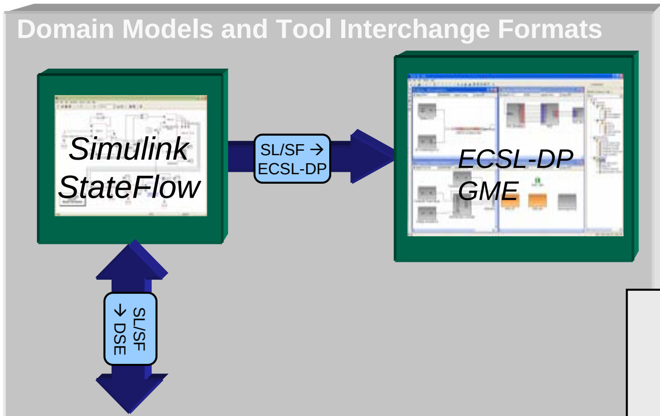
Interrelation with System Composition

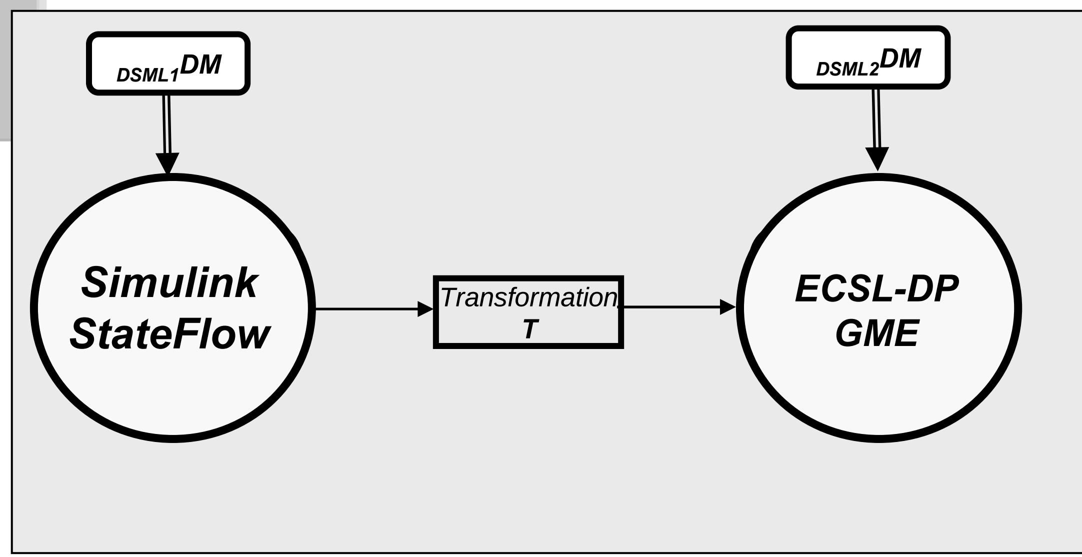


Example Tool Chain: Vehicle Control Platform (VCP)



Constructing Tool Chains: Modeling and Transformations



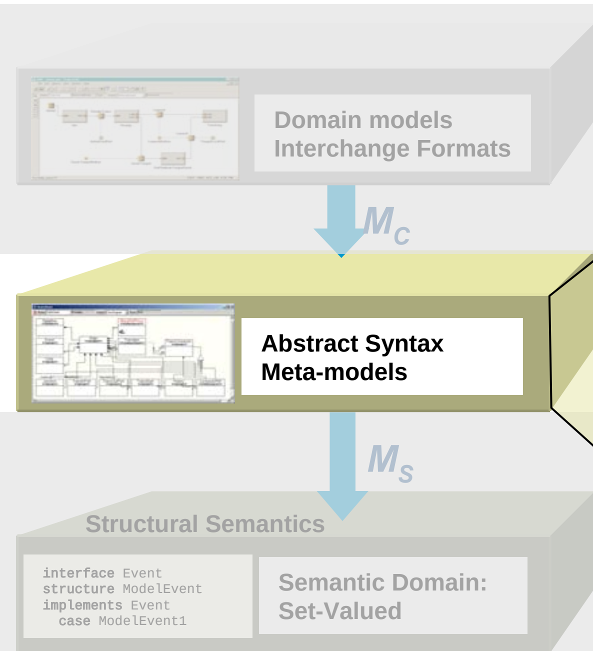


- Large influence of concrete syntax
- No clear role of semantics
- It is not clear what are we doing?

Content

- Introduction to model-based design
- System Composition Dimension
  - Layers
  - Approaches
  - Languages
- Tool Composition Dimension
  - Layers
  - Building Tool Chains
- Metamodeling and Metaprogrammable Tools
- Semantics

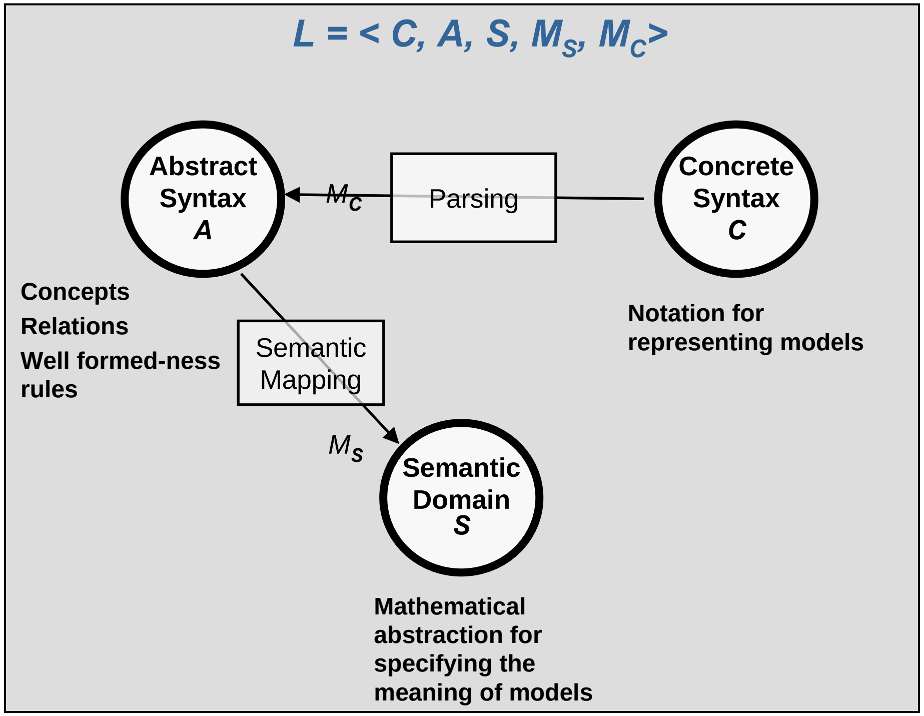
Metamodeling Layer Objectives



- Metamodeling
- Model Data Management
- Model Transformation
- Tool Integration

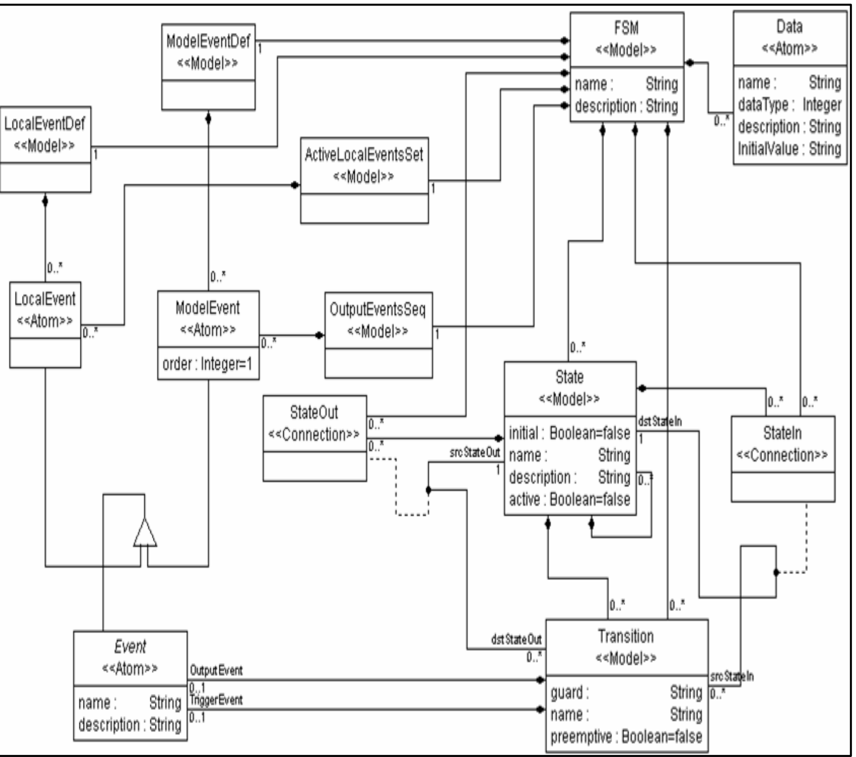
Metamodeling and Domain Specific Modeling Languages

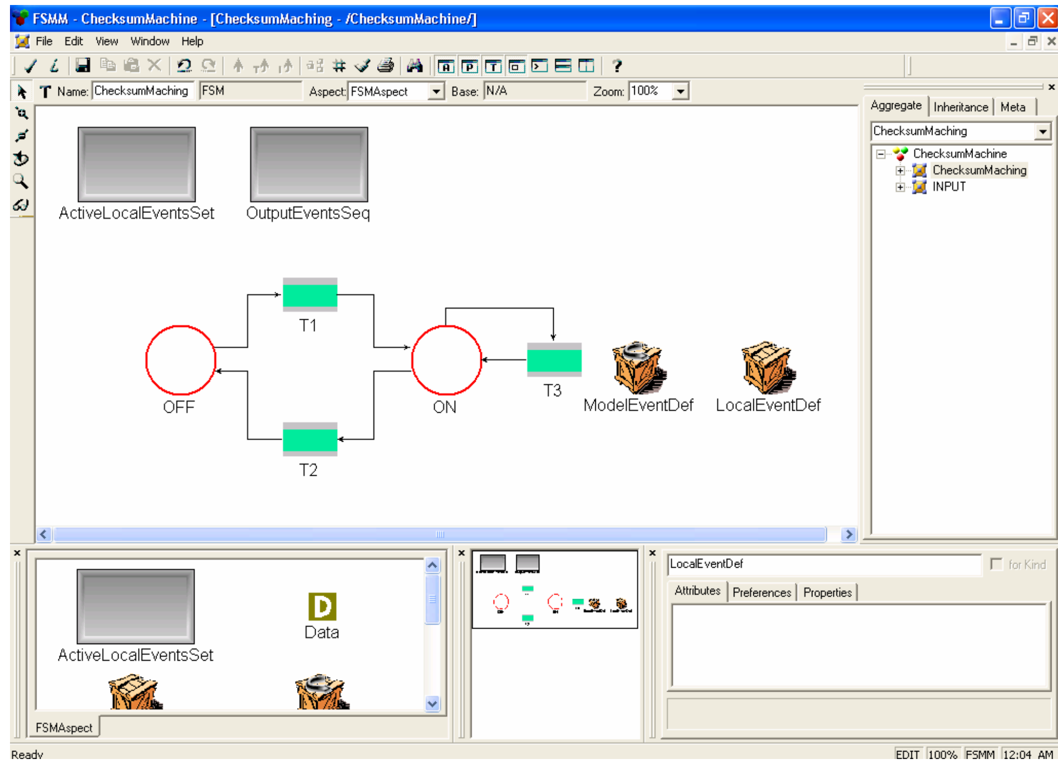
Domain Specific Modeling Language (DSML)



- **Model**: precise representation of artifacts in a modeling language L
- **Modeling language**: defined by the notation (C), concepts/relations and integrity constraints (A), the semantic domain (S) and mapping among these.
- **Metamodel:** formal (i.e. precise) representation of the modeling language L using a metamodeling language L_M.

Modeling Example: Metamodel and Models





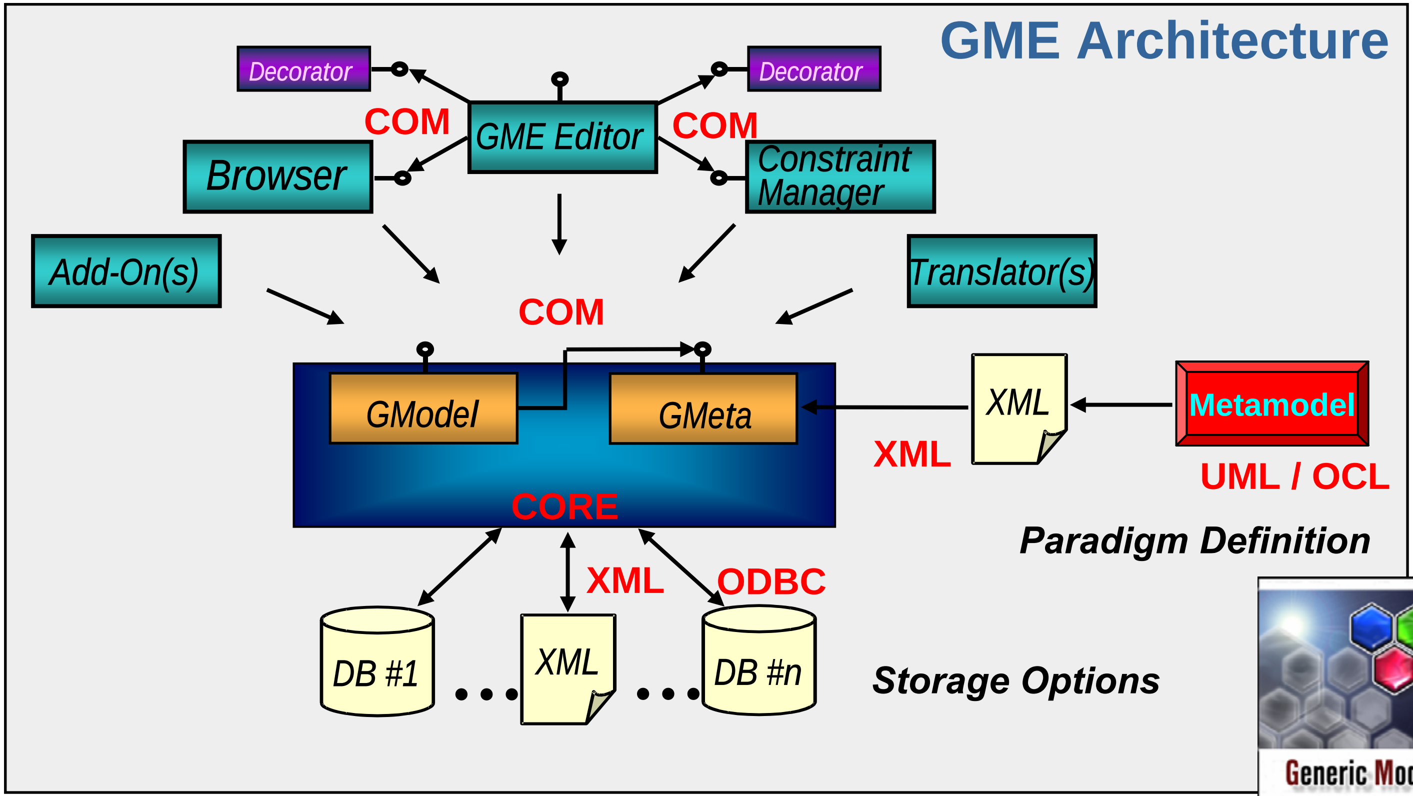
Metamodel:

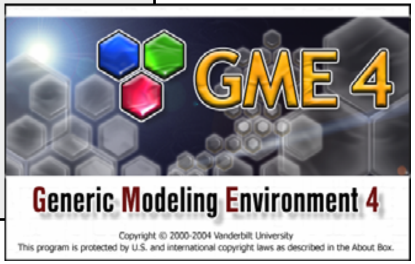
- Defines the set of admissible models
- “Metaprogramms” tool

Model:

- Describes states and transitions
- Modeling tool enforces constraints

Metaprogrammable Modeling Tool: GME



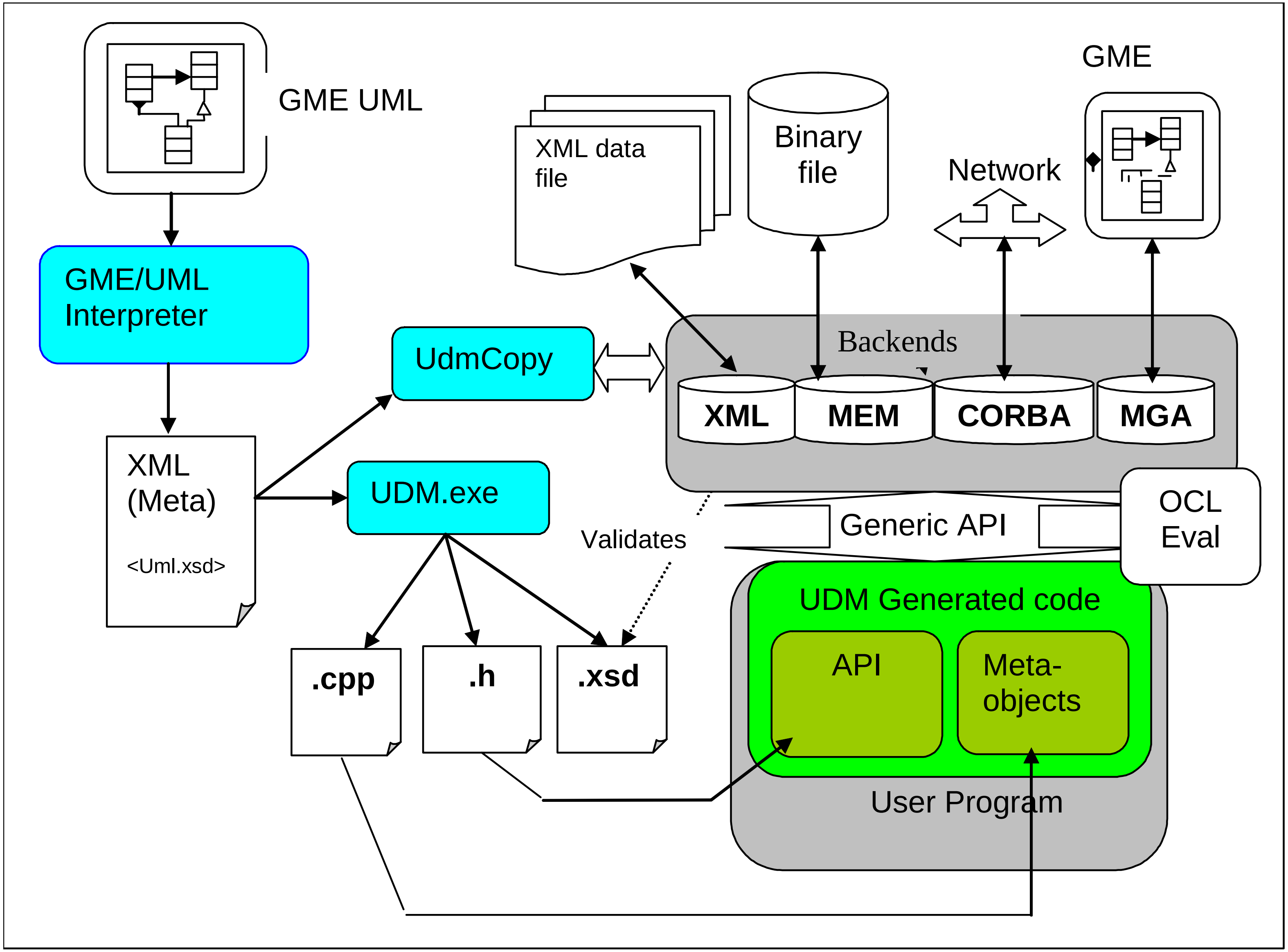


- Configuration through UML and OCL-based metamodels
- Extensible architecture through COM
- Multiple standard backend support (ODBC, XML)
- Multiple language support: C++, VB, Python, Java, C#

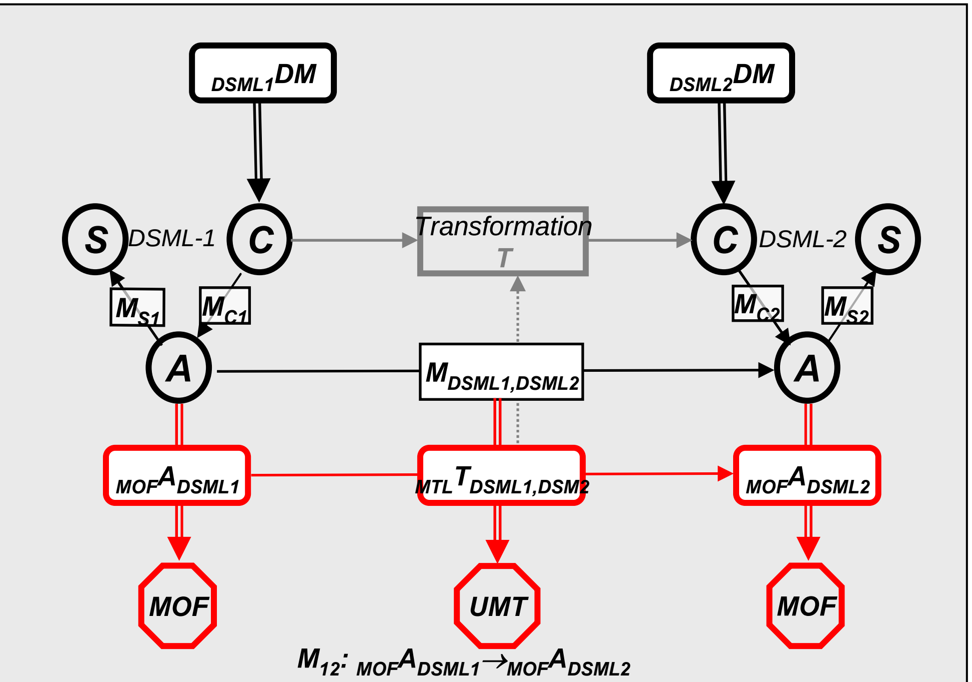
Model Data Management: The UDM Goals

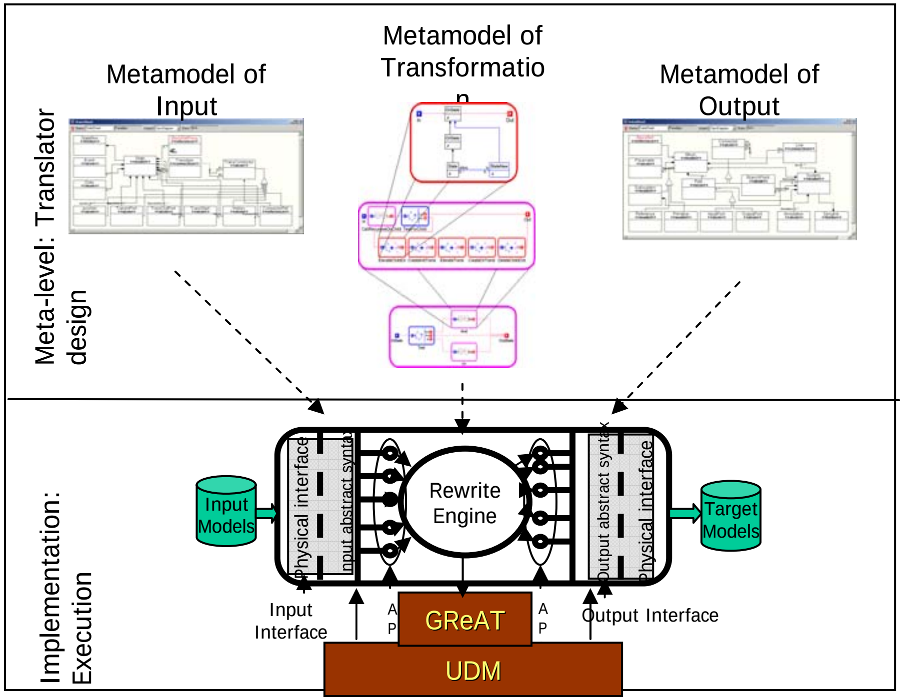
- To have a conceptual view of data/metadata that is independent of the storage format.
- Such a conceptual view should be based on standards such as UML.
- Have uniform access to data/metadata such that storage formats can be changed seamlessly at either design time or run time.
- Generate a metadata/paradigm specific API to access a particular class of data.

Model Data Management: The UDM Tool Suite



Model Transformation: The "Workhorse" of MIC





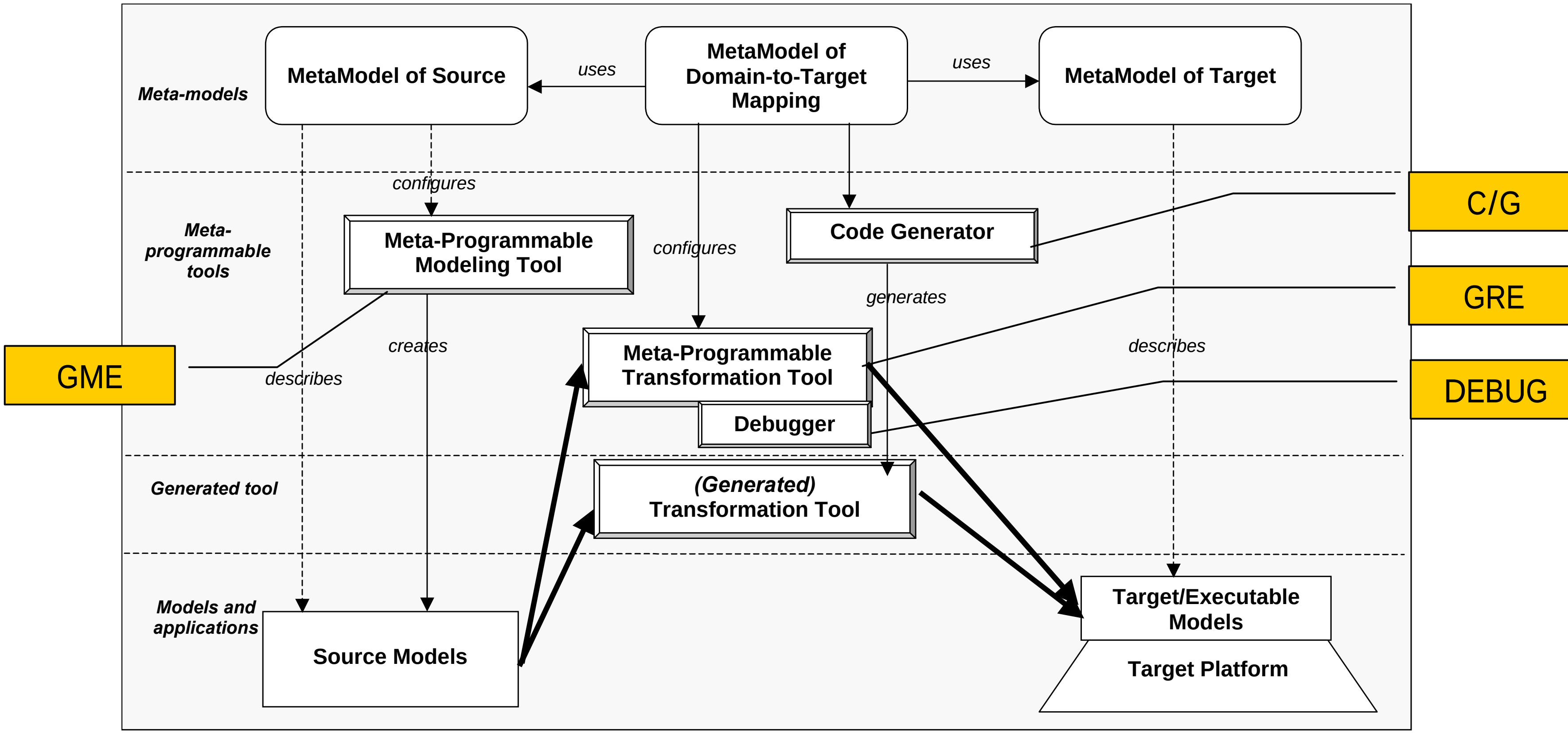
Relevant Use of Model Transformations:

- Building integrated models by extracting information from separate model databases
- Generating models for simulation and analysis tools
- Defining semantics for DSML-s

MIC Model transformation technology is:

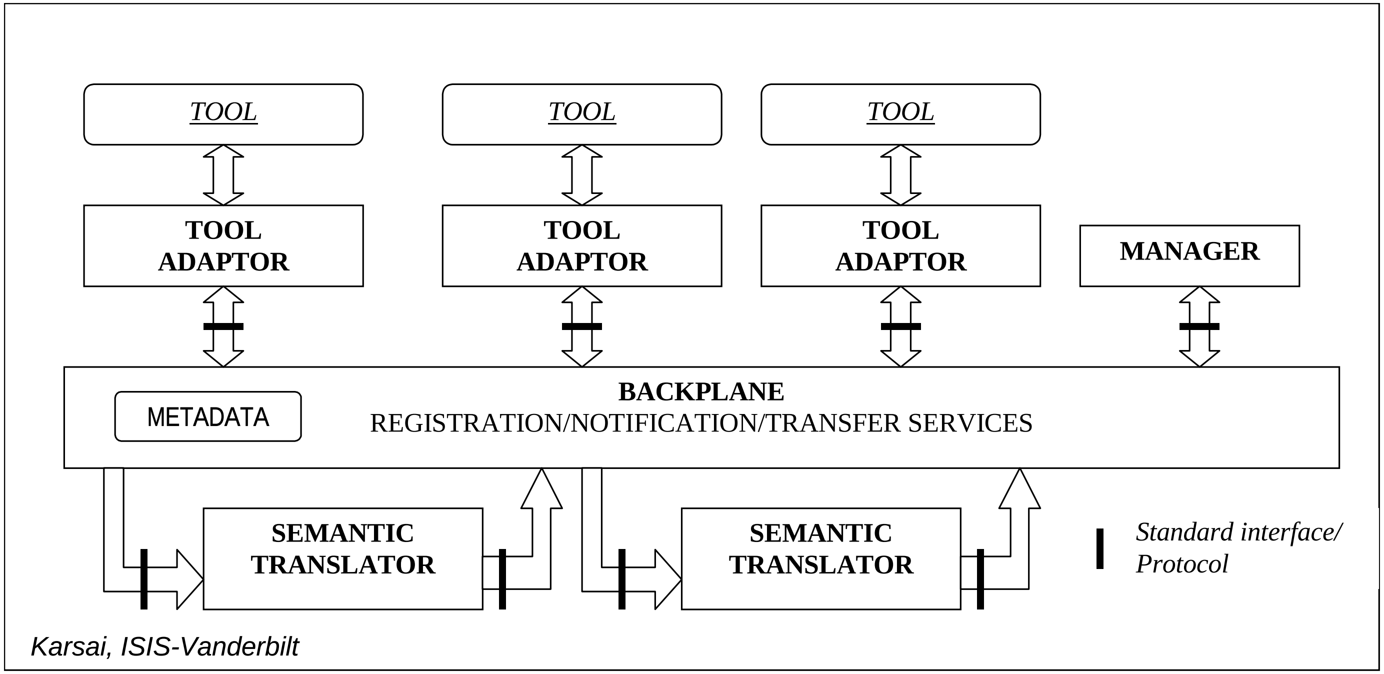
- Based on graph transformation semantics
- Model transformations are specified using metamodels and the code is automatically generated from the models.

Model Transformation: The GReAT Tool Suite



Tools: UMT Language, GRE (engine), C/G, GR-DEBUG

Open Tool Integration Framework: OTIF

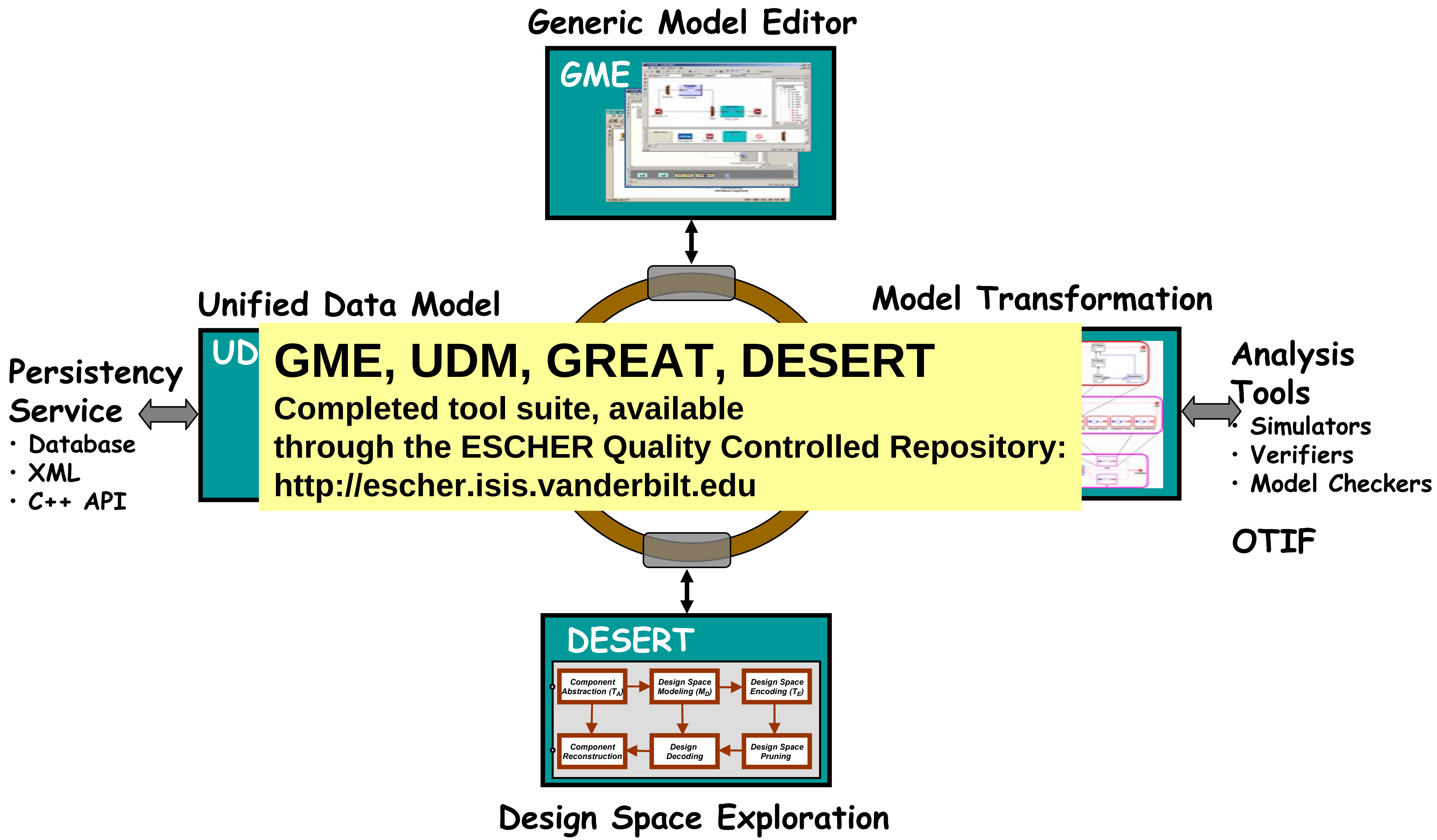


RFP is Discussed at
MIC PSIG
OMG

- **Share models** using Publish/Subscribe Metaphor
- Status:
  - Completed, tested in several tool chains
  - Protocols in OMG/CORBA
  - CORBA as a transport layer
  - Integration with ECLIPSE is in progress

http://www.isis.vanderbilt.edu/Projects/WOTIF/default.html

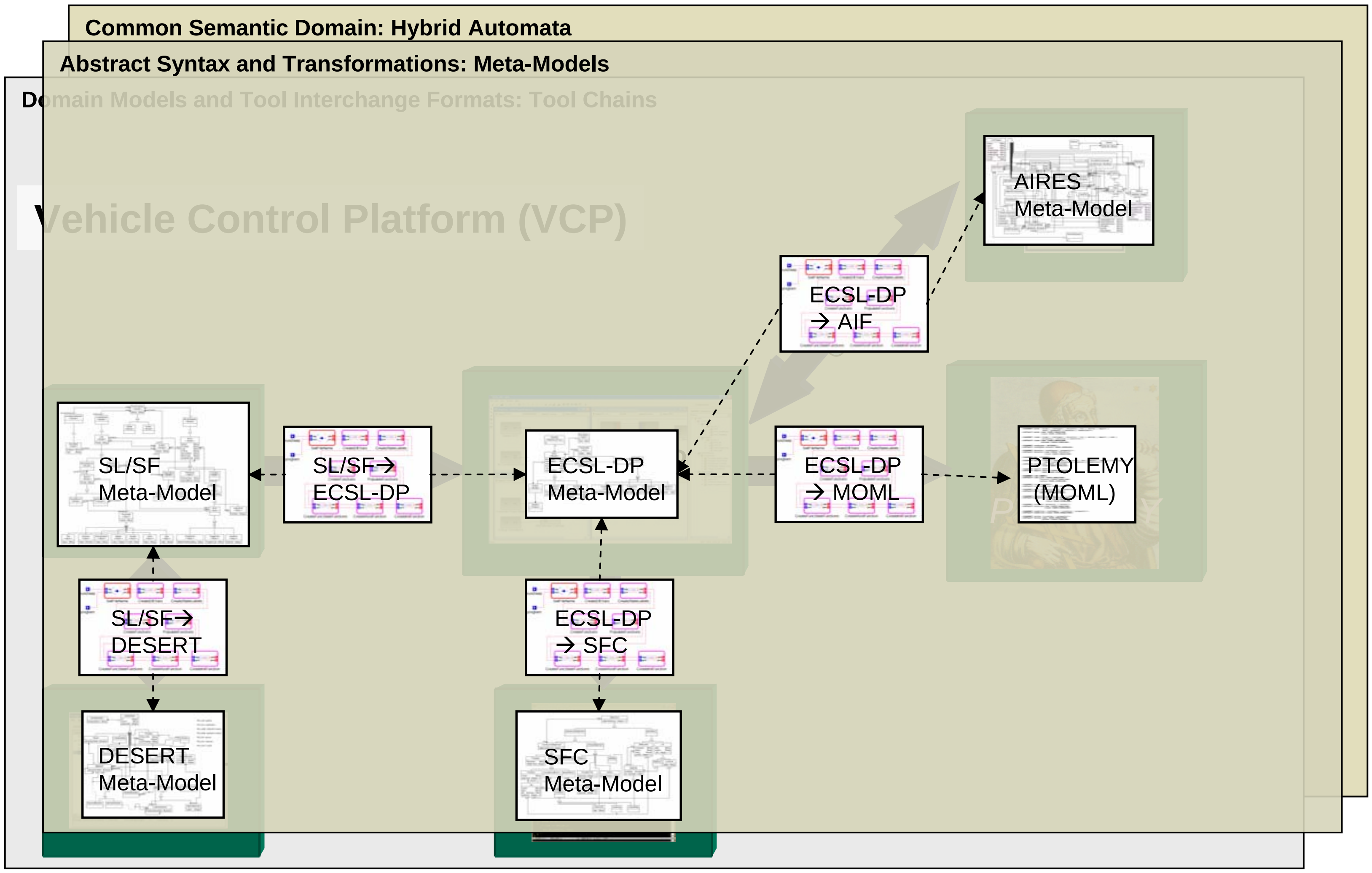
MIC Metaprogrammable Tool Suite



GME, UDM, GREAT, DESERT

Completed tool suite, available through the ESCHER Quality Controlled Repository: http://escher.isis.vanderbilt.edu

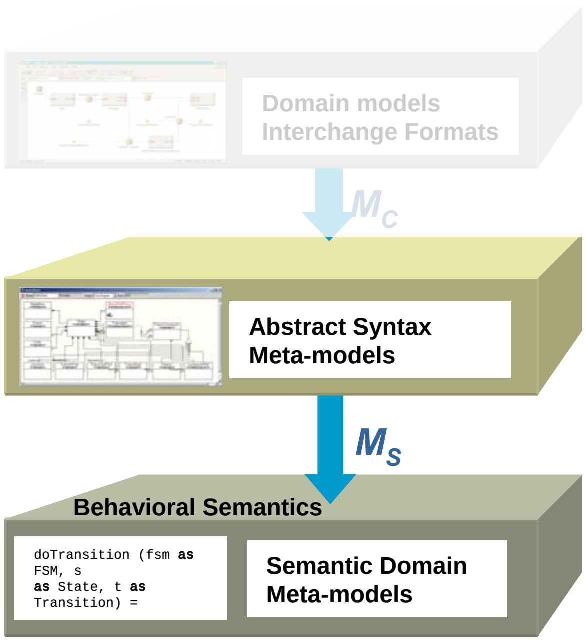
"Backplane View" of the VCP Tool Chain



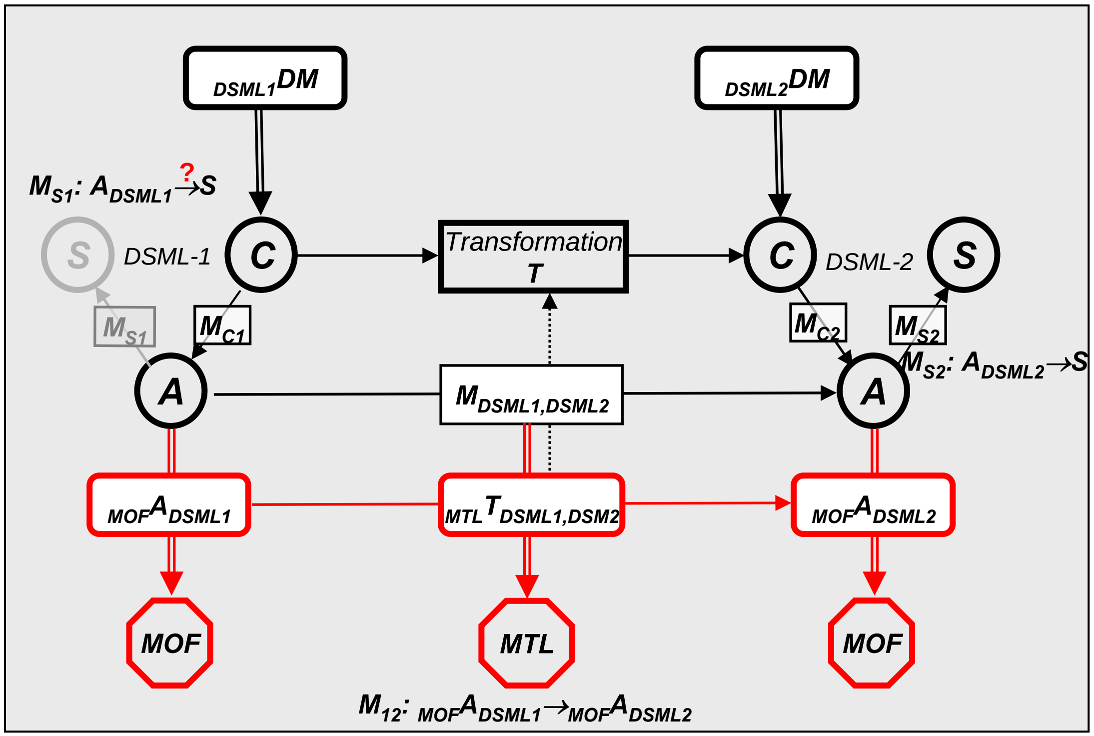
Content

- Introduction to model-based design
- System Composition Dimension
  - Layers
  - Approaches
  - Languages
- Tool Composition Dimension
  - Layers
  - Building Tool Chains
- Metamodeling and Metaprogrammable Tools
- Semantics

How About Semantics?

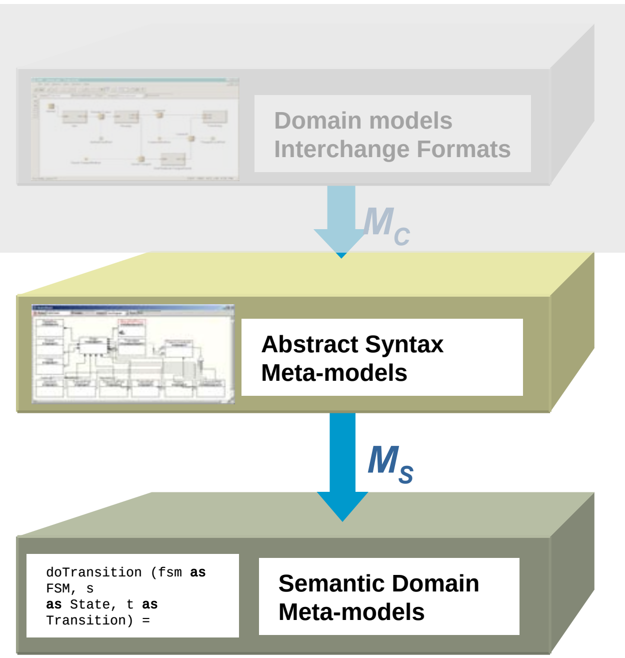


Transformational Specification of Behavioral Semantics

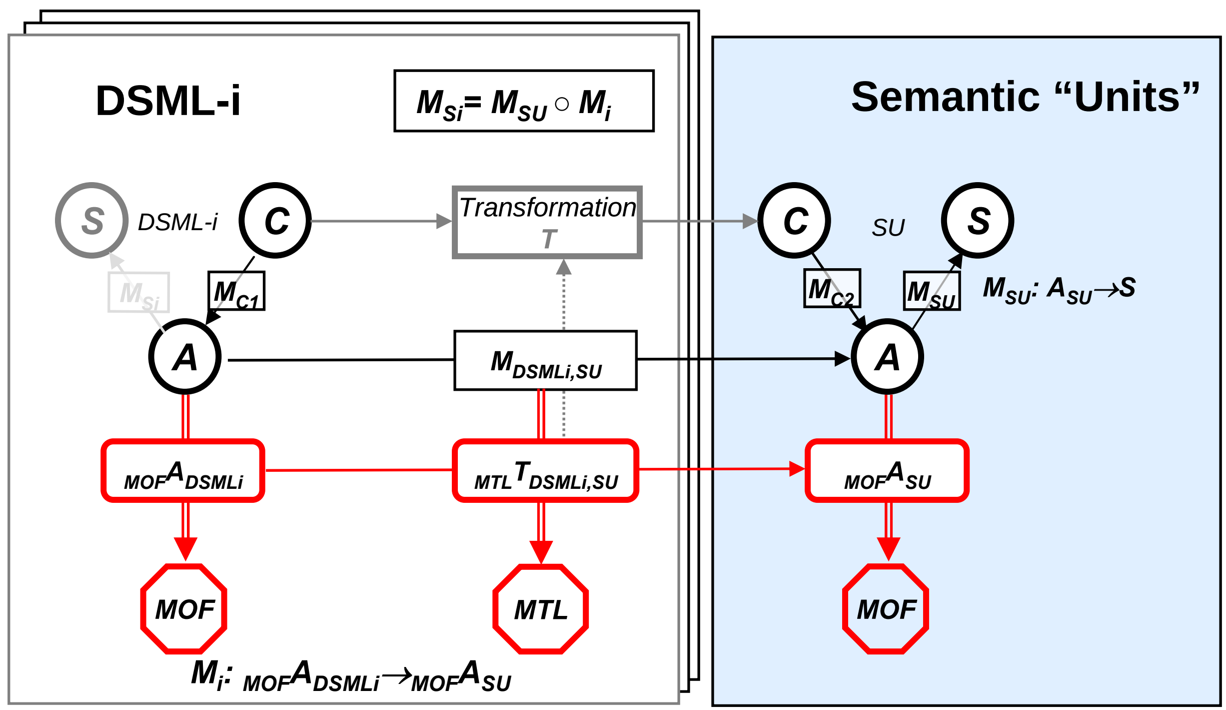


$$M_{S1} = M_{S2} \circ M_{12}$$

Semantic Anchoring



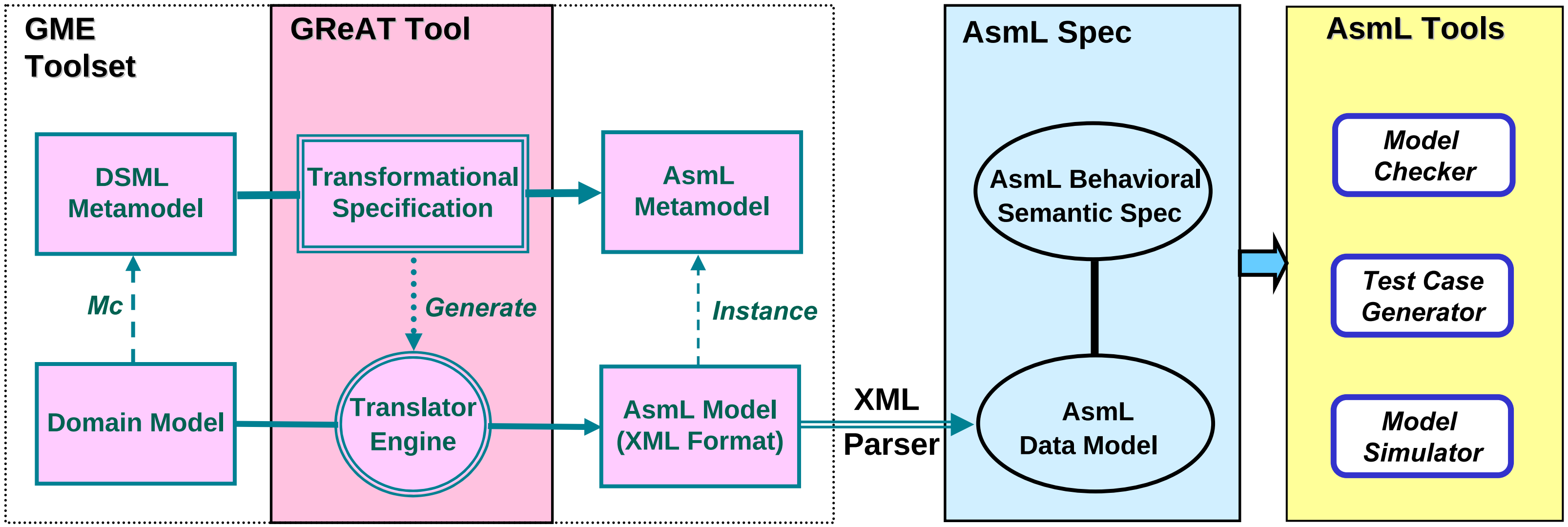
Semantic Anchoring of DSML-s



- The "Semantic Units" are selected common semantics such as MoC-s
- DSML-s or their aspects are anchored to the common semantics using transformations
- The "Semantic Units" are specified in a formal framework

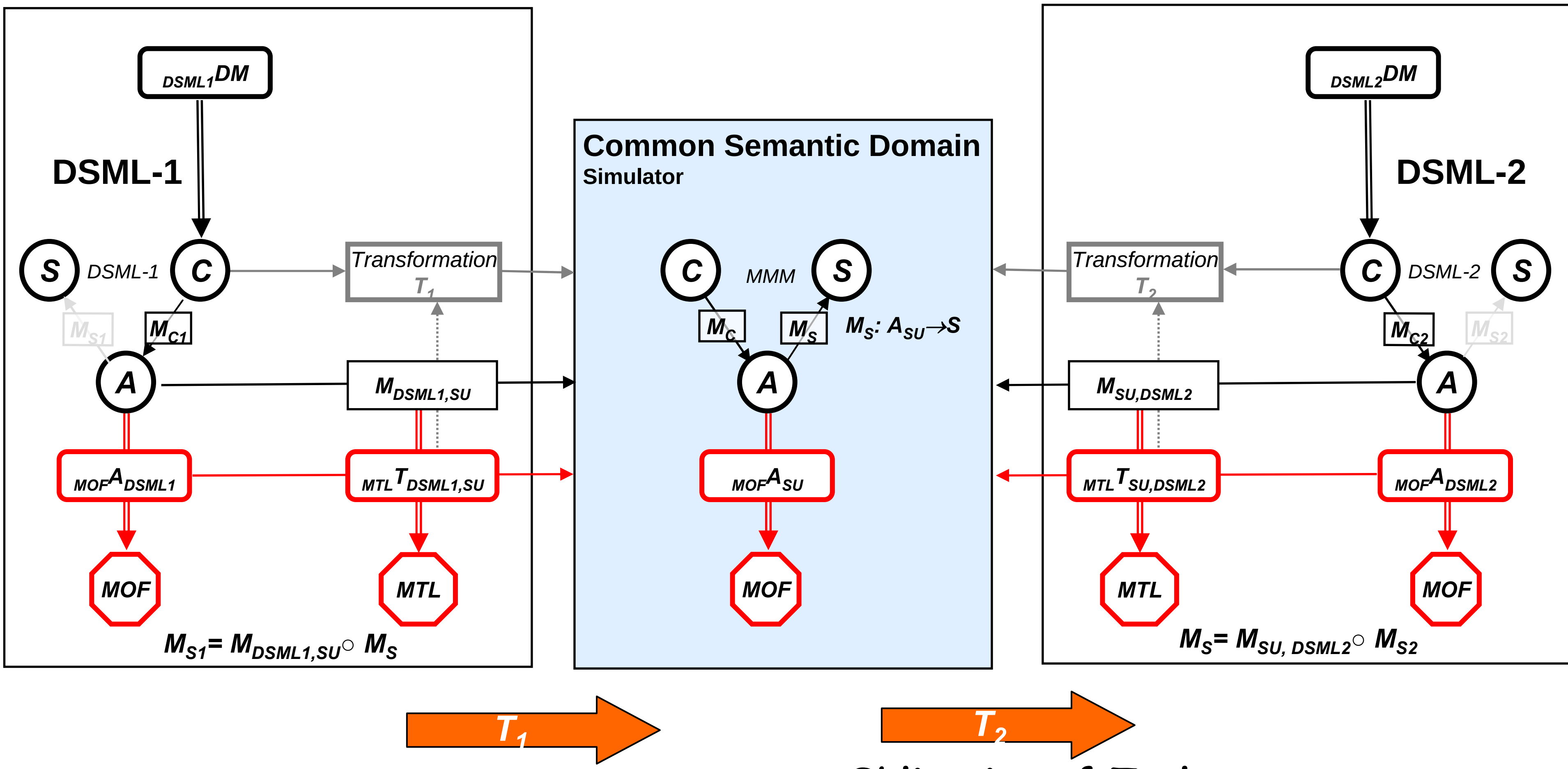
Semantic Anchoring Infrastructure

- Semantic Unit
  - A well-defined operational semantics for core Models of Computation and Behaviors (e.g. FSM).
- Semantic Anchoring
  - Define the semantics a DSML through specifying the transformation specification to a semantic unit.



Semantic Integration of Tools

Modeling Tool **Analysis Tool**



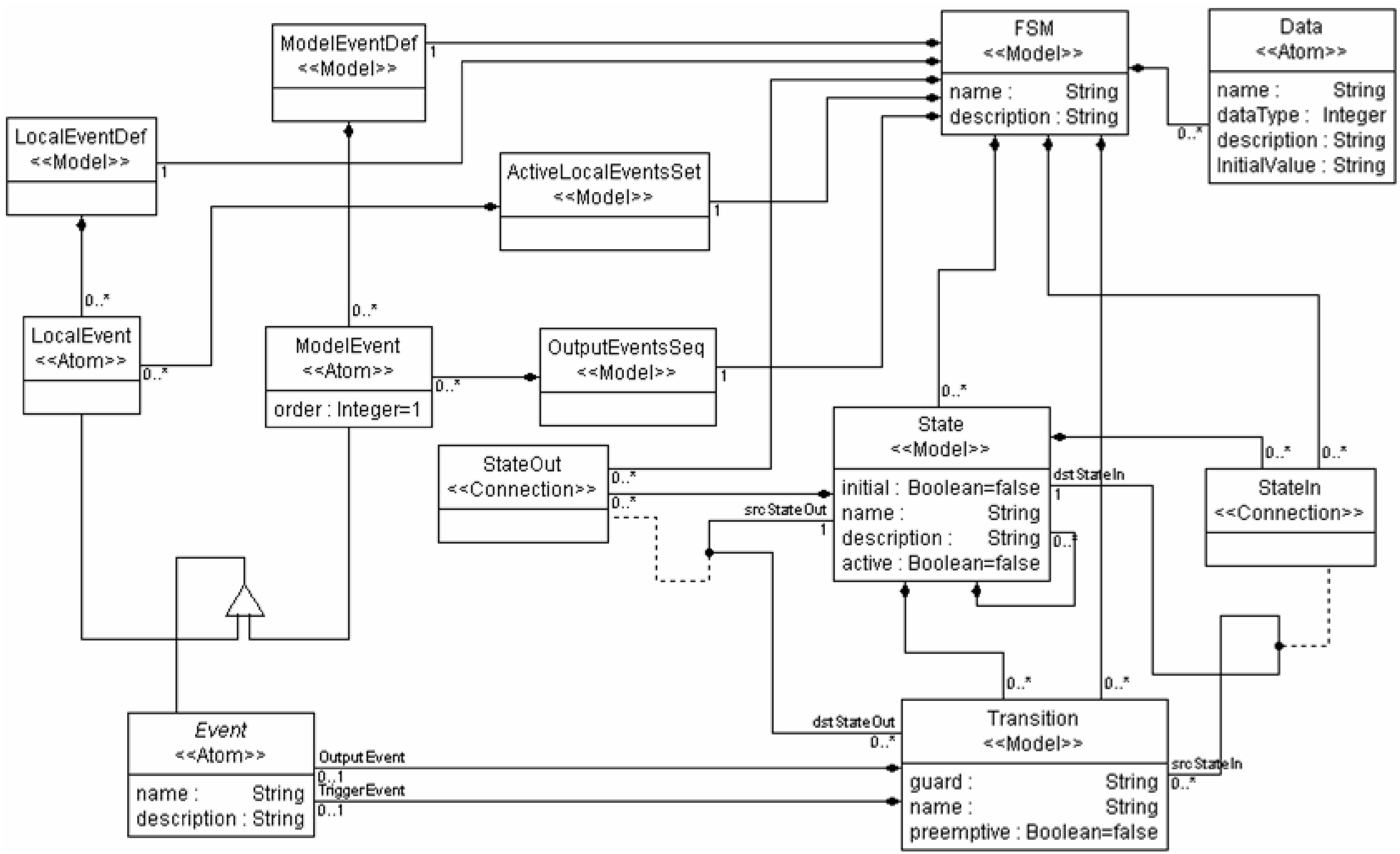
Obligation of DSML Developer

Obligation of Tool Developer

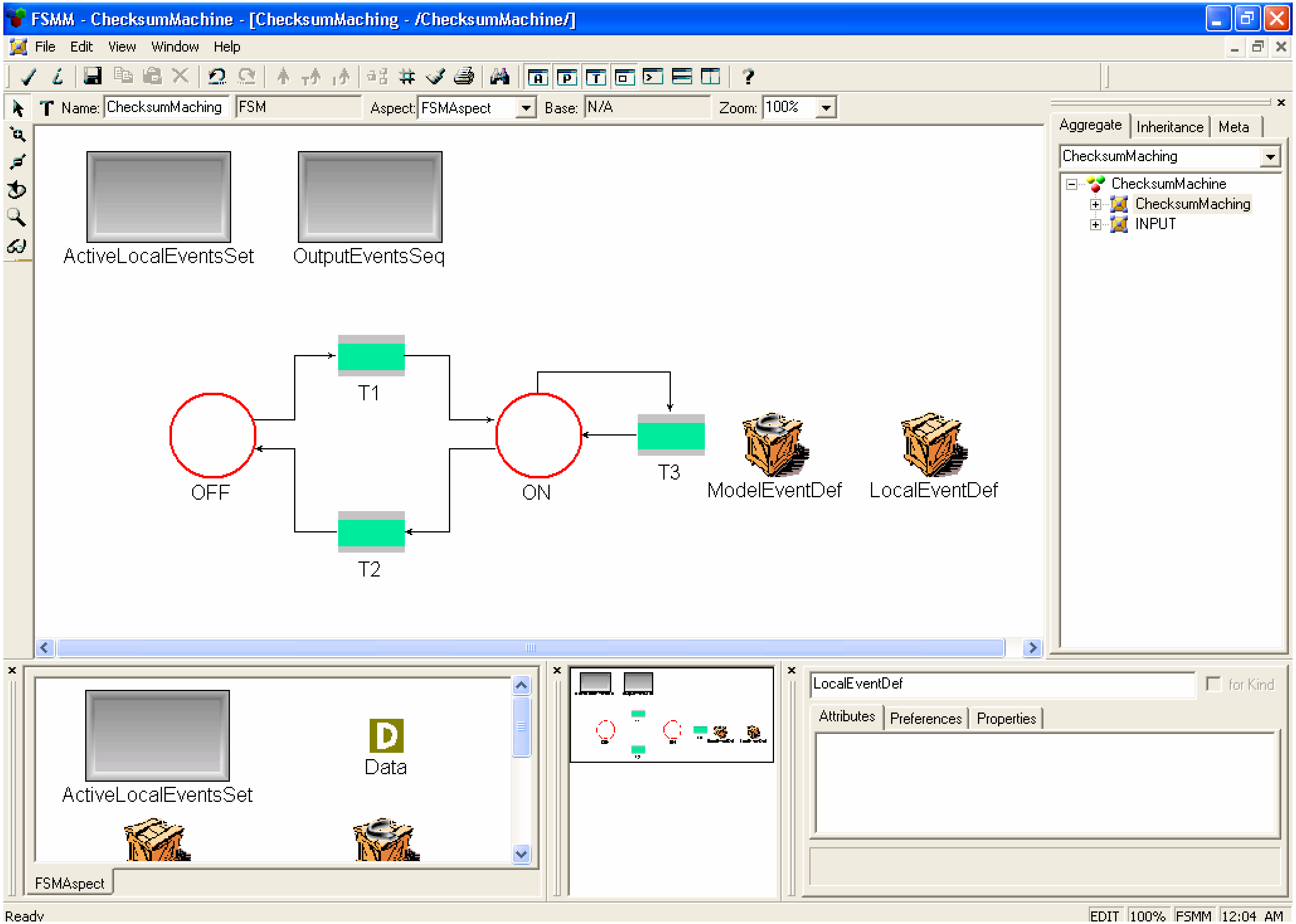
Summary

- "Plug-and-Play" component technology is not sufficient for embedded software of non-trivial size
- Model-based design addresses core issues: it integrates systems and software engineering
- Active research programs in system and tool chain composition have made significant progress in the past five years
- New frontier: explicit semantics

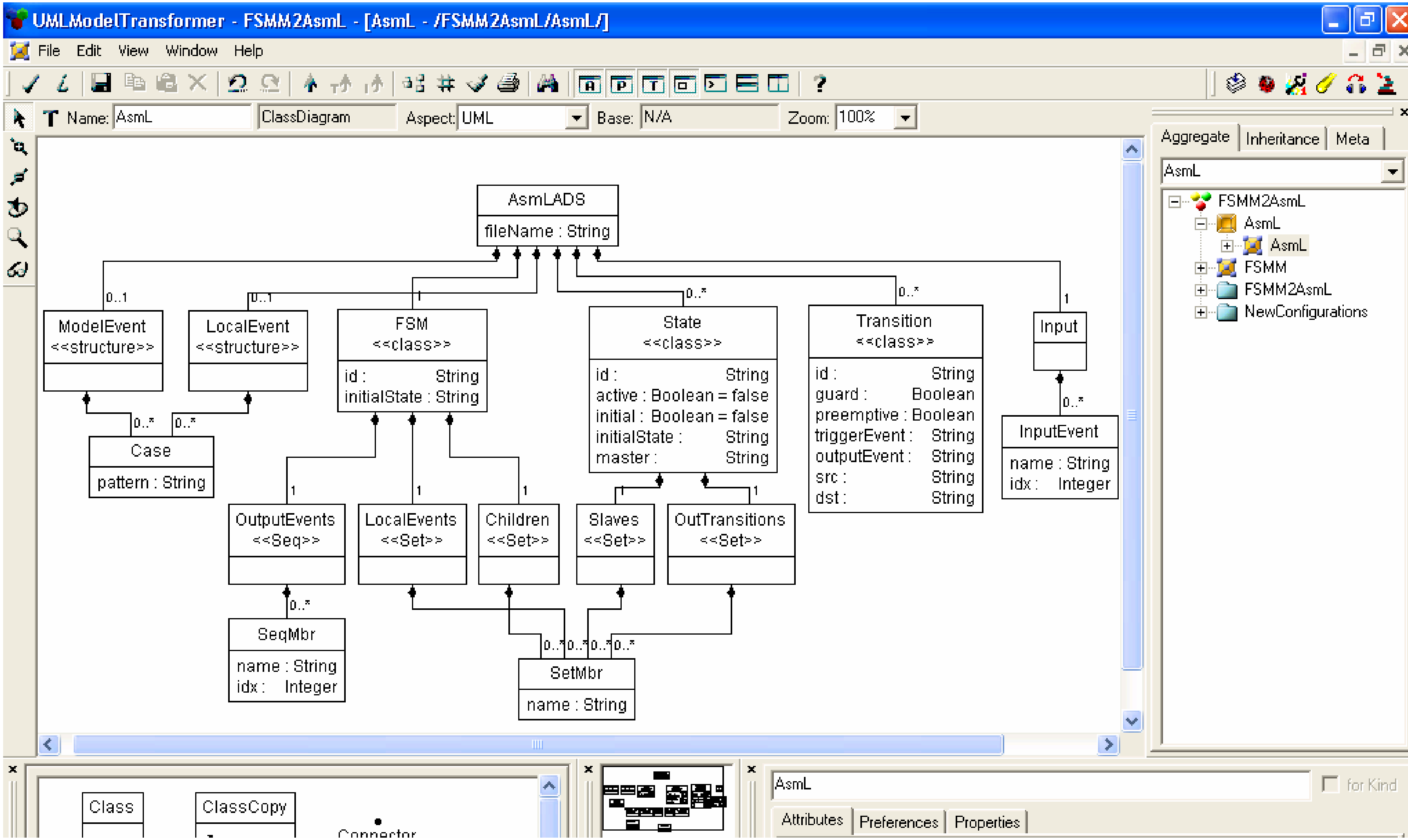
FSM Metamodel



FSM Model


FSMM - ChecksumMachine - [ChecksumMaching - /ChecksumMachine/]
File Edit View Window Help
Name: ChecksumMaching FSM Aspect: FSMAspect Base: N/A Zoom: 100%
ActiveLocalEventsSet
OutputEventsSeq
T1
OFF
ON
T3
ModelEventDef
LocalEventDef
T2
Aggregate Inheritance Meta
ChecksumMaching
ChecksumMachine
ChecksumMaching
INPUT
ActiveLocalEventsSet
Data
FSMAspect
LocalEventDef
for Kind
Attributes Preferences Properties
Ready
EDIT 100% FSMM 12:04 AM

Metamodel for AsmL Abstract Data Model



AsmL Abstract Data Model

Abstract Model

```
structure State
  id      as  String
  initial as Boolean

structure Transition
  id           as String

structure Event
  id           as String

class StateAutomaton
  S as Set of State
  T as Set of Transition
  E as Set of Event
  Connections as Map of <Transition,  (State, State)>
  TriggerEvent as Map of <Transition,  Event?>
  OutputEvent as Map of <Transition,  Event?>
  var CurrentState as State
  var OutputEvents as Seq of Event = []

  GetOutTransitions (s as State) as Set of Transition
    let trans = {t | t in T where Connections(t).First = s}
    return trans


  GetEnabledTransitions (e as Event) as Set of Transition
    let trans = GetOutTransitions(CurrentState)
    let enabledTrans = {t | t in trans where TriggerEvent(t) = e or TriggerEvent(t) = null}
    return enabledTrans

  DoTransition(t as Transition)
    step
      if OutputEvent(t) <> null then
        OutputEvents := OutputEvents + [OutputEvent(t)]
        WriteLine ("OutputEvent " + OutputEvent(t).id)
    step CurrentState := Connections(t).Second
    step WriteLine ("Do transition " + t.id)

  React(e as Event)
    step until fixpoint
     let trans = GetEnabledTransitions(e)
     if Size(trans) <> 0 then
        choose t in trans
          DoTransition(t)
```

AsmL Behavioral Semantic Specifications

```
structure State
  id      as  String
  initial as Boolean

structure Transition
```

```
React(e as Event)
  step until fixpoint
   let trans = GetEnabledTransitions(e)
   if Size(trans) <> 0 then
      choose t in trans
        DoTransition(t)
```

Behavior in Terms of Abstract Model

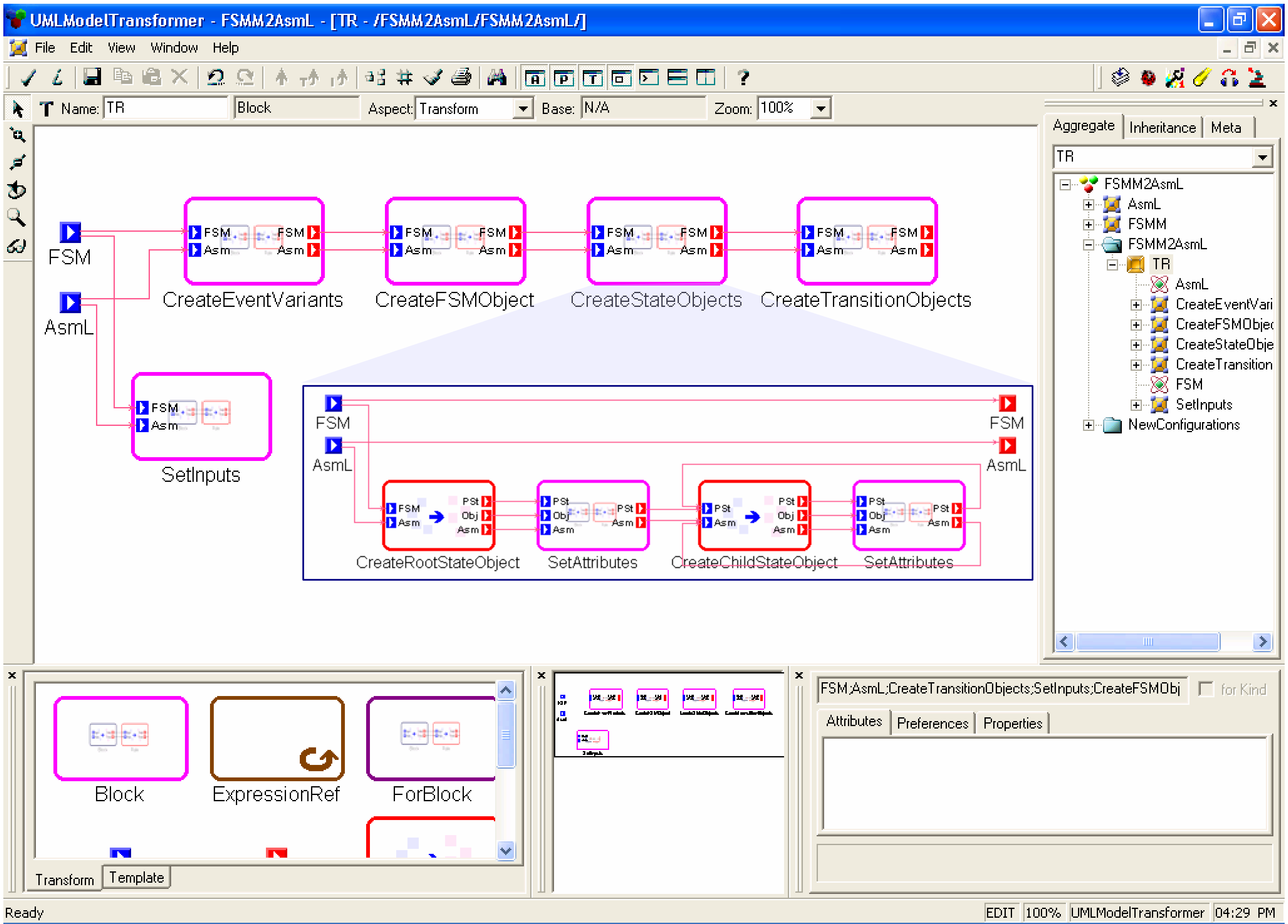
```
GetOutTransitions (s as State) as Set of Transition
  let trans = {t | t in T where Connections(t).First = s}
  return trans


GetEnabledTransitions (e as Event) as Set of Transition
  let trans = GetOutTransitions(CurrentState)
  let enabledTrans = {t | t in trans where TriggerEvent(t) = e or TriggerEvent(t) = null}
  return enabledTrans

DoTransition(t as Transition)
  step
    if OutputEvent(t) <> null then
      OutputEvents := OutputEvents + [OutputEvent(t)]
      WriteLine ("OutputEvent " + OutputEvent(t).id)
  step CurrentState := Connections(t).Second
  step WriteLine ("Do transition " + t.id)

React(e as Event)
  step until fixpoint
   let trans = GetEnabledTransitions(e)
   if Size(trans) <> 0 then
      choose t in trans
        DoTransition(t)
```

Transformational Specifications



AsmL Data Model in XML Format


```
<AsmLADS _id="id988" fileName="" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="UDM\AsmL.xsd">
- <FSM id="ChecksumMaching" _id="id9d5" initialState="OFF">
  + <Children _id="id9f4">
    <LocalEvents _id="id9e5" />
    <OutputEvents _id="id9e0" />
  </FSM>
+ <Input _id="id98b">
+ <LocalEvent _id="id9bf">
+ <ModelEvent _id="id9ad">
- <State id="OFF" _id="ida17" active="false" master="" initial="true" initialState="">
  + <OutTransitions _id="ida5b">
    <Slaves _id="ida3d" />
  </State>
+ <State id="ON" _id="ida18" active="false" master="" initial="false" initialState="ZERO">
+ <State id="ZERO" _id="ida74" active="false" master="ON" initial="false" initialState="">
+ <State id="ONE" _id="ida75" active="false" master="ON" initial="false" initialState="">
  <Transition id="T11" _id="idade" dst="ONE" src="ZERO" guard="true" preemptive="false" outputEvent=""
    triggerEvent="LocalEvent.one" />
  <Transition id="T12" _id="idadf" dst="ZERO" src="ONE" guard="true" preemptive="false" outputEvent=""
    triggerEvent="LocalEvent.one" />
  <Transition id="T13" _id="idae0" dst="ZERO" src="ZERO" guard="true" preemptive="false" outputEvent=""
    triggerEvent="LocalEvent.zero" />
  <Transition id="T14" _id="idae1" dst="ONE" src="ONE" guard="true" preemptive="false" outputEvent=""
    triggerEvent="LocalEvent.zero" />
  <Transition id="T1" _id="idb0e" dst="ON" src="OFF" guard="true" preemptive="false" outputEvent="ModelEvent.start"
    triggerEvent="" />
  <Transition id="T2" _id="idb0f" dst="OFF" src="ON" guard="true" preemptive="false" outputEvent="" triggerEvent="ModelEvent.stop" />
  <Transition id="T3" _id="idb10" dst="ON" src="ON" guard="true" preemptive="false" outputEvent=""
    triggerEvent="ModelEvent.reset" />
</AsmLADS>
```

AsmL Data Model

Instance of the Abstract Model

```
initStateAutomaton() as StateAutomaton
  let S1 = State("S1", true)
  let S2 = State("S2", false)
  let T1 = Transition("T1")
  let e1 = Event("e1")
  let S = {S1, S2}
  let T = {T1}
  let E = {e1}
  let Connections = {T1 -> (S1, S2)}
  let TriggerEvent = {T1 -> e1}
  let OutputEvent = {T1 -> e1}
  let InitialState = S1
  return new StateAutomaton(S,T,E,Connections,TriggerEvent,OutputEvent,
InitialState)
```