Towards Systematic Model-Based Development of Embedded Systems

Janos Sztipanovits
ISIS, Vanderbilt University

DATE 07
Nice, France
April 20, 2007

Content

- Introduction
  - Basic concepts: platforms, abstractions and DSML-s
- Model Integrated Computing
  - Structural and Behavioral Semantics
  - Metamodel composition
  - MIC Tool Suite
- Making Behavioral Semantics Explicit
  - Semantic anchoring
  - Remarks on composition

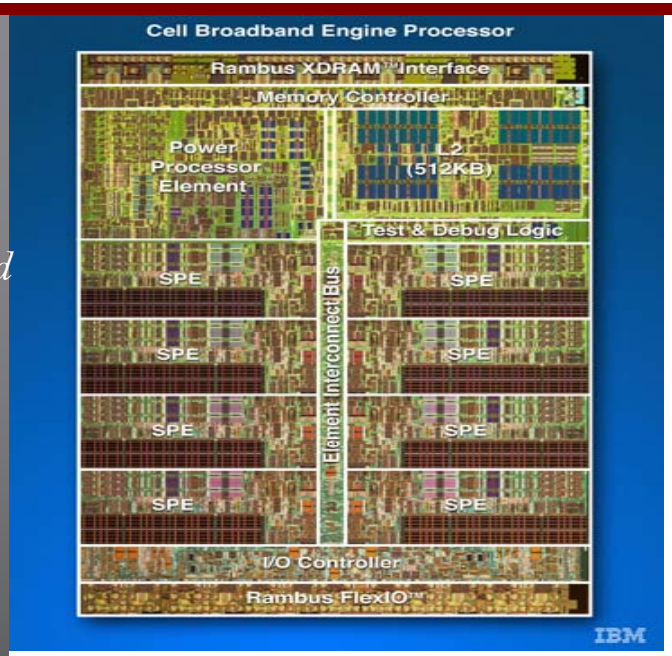
Content

- Introduction
  - Basic concepts: platforms, abstractions and DSML-s
- Model Integrated Computing
  - Structural and Behavioral Semantics
  - Metamodel composition
  - MIC Tool Suite
- Making Behavioral Semantics Explicit
  - Semantic anchoring
  - Remarks on composition

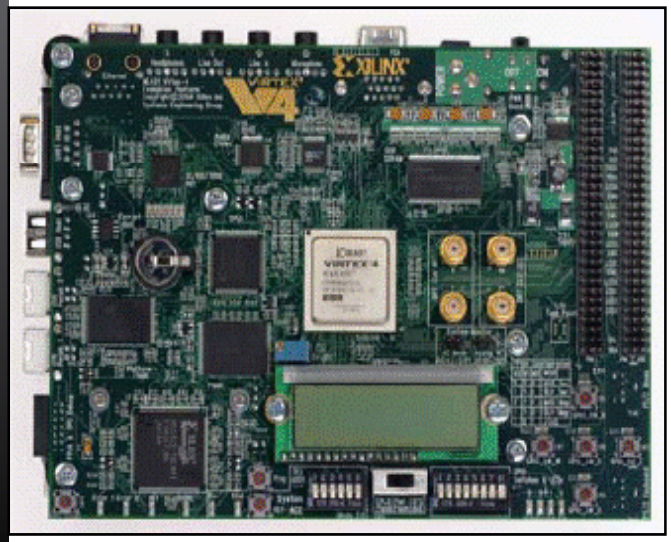
Changing Platforms

Emerging new platforms continue changing requirements for embedded systems design.

The changes in platforms are fundamental:
- *parallel architectures,*
- *customizable HW*



IBM Cell Processor



Kees Vissers, Xilinx, MPSoC 2005

Cell Processor Platform
- Power processor element
- 8 Synergistic processing elements
- Element Interconnect bus

Architecture Constraints
- Single PPE
- Elements must share memory
- Elements must share interconnect bus

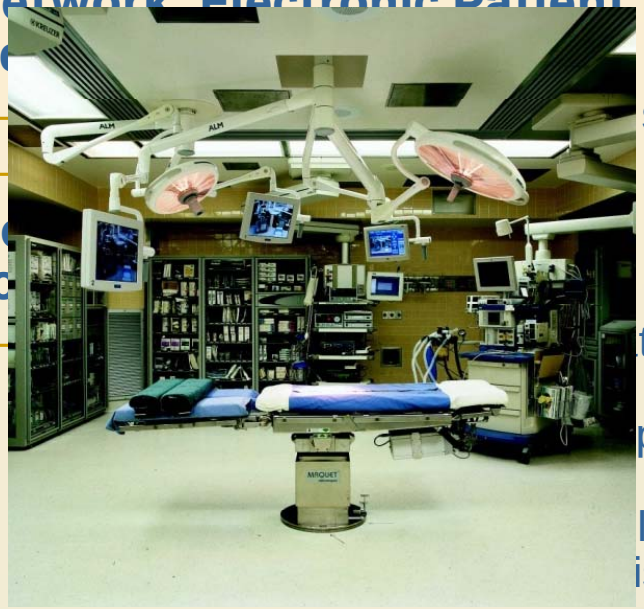
Xilinx DSP Hardware Platform
- Programmable arithmetic fabric
- Hundreds of Xtreme DSP slices
- 100+ multipliers

Flexible Architecture
- Custom HW
- Parameterizable HW IP
- Open HW data format

Changing Applications

- **The share of value of embedded computing components in the application areas expected to reach 30%-40% value by 2010**
- **Applications:**
  - **Automotive Systems**
    - **Light and heavy automobiles, trucks, buses**
  - **Aerospace Systems**
    - **Airplanes, space systems**
  - **Consumer electronics**
    - **Mobile phones, office electronics, digital appliances**
  - **Health/Medical Equipment**
    - **Patient monitori… pumps, artificial or…**
  - **Industrial Automation**
    - **Supervisory Control and D… Acquisition (SCADA) systems… chemical and power plants**
    - **Manufacturing systems**
  - **Defense**
    - **Source of superiority in all weapon systems**

- **National Health Information Network, Electronic Patient Re…**
  - …service
  - …
- **He… co…**
  - …tion),
  - …ps
  - …lity),
  - …is), …



- **Operating Room of the Future (Goldman)**
  - Closed loop monitoring and control; multiple treatment stations, plug and play devices; robotic microsurgery
  - System coordination challenge
- **Progress in personalized medicine: systems biology; disease dynamics, control**

Platforms Are Layers of Abstractions

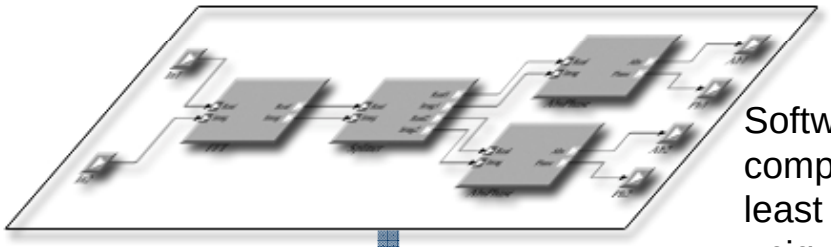
Abstraction layers define platforms.

Abstractions are linked through refinement relations.

Abstraction layers allow the verification of different properties .

Key Idea: Manage design complexity by creating layers of abstractions in the design flow.

(Platform-based design: Alberto Sangiovanni-Vincentelli)

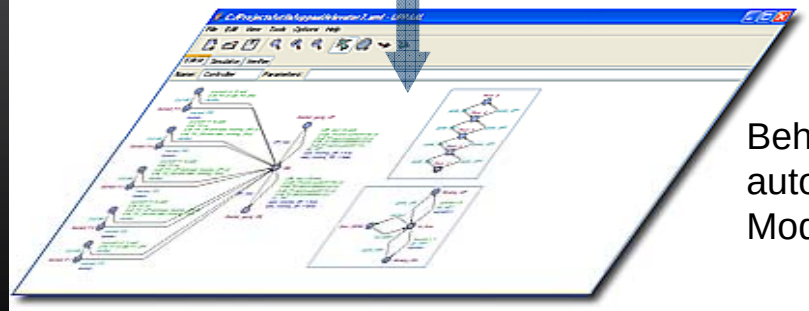


Software architecture defines the composition of functions such that a least fixed point exists and is unique.

Platform mapping

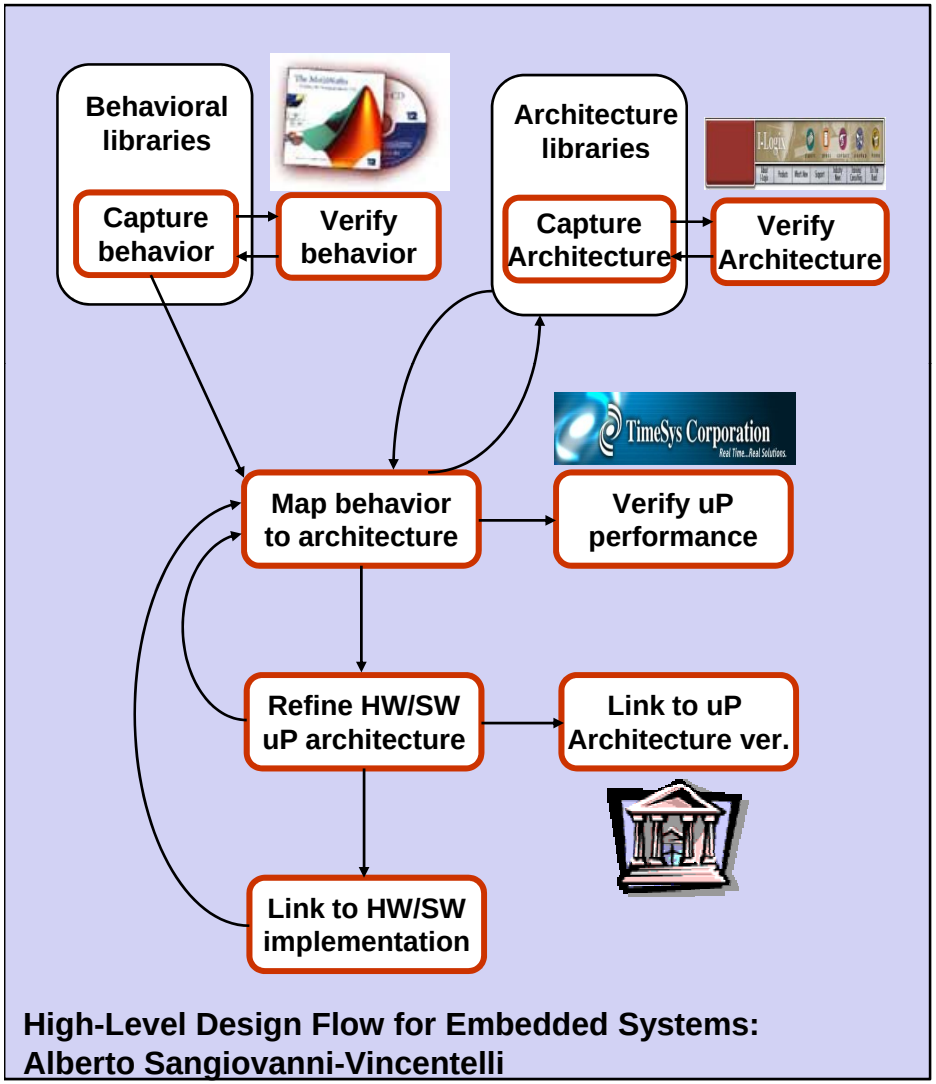
Hardware architecture defines a set of concurrent functional units, where the software architecture can be deployed.

Platform mapping

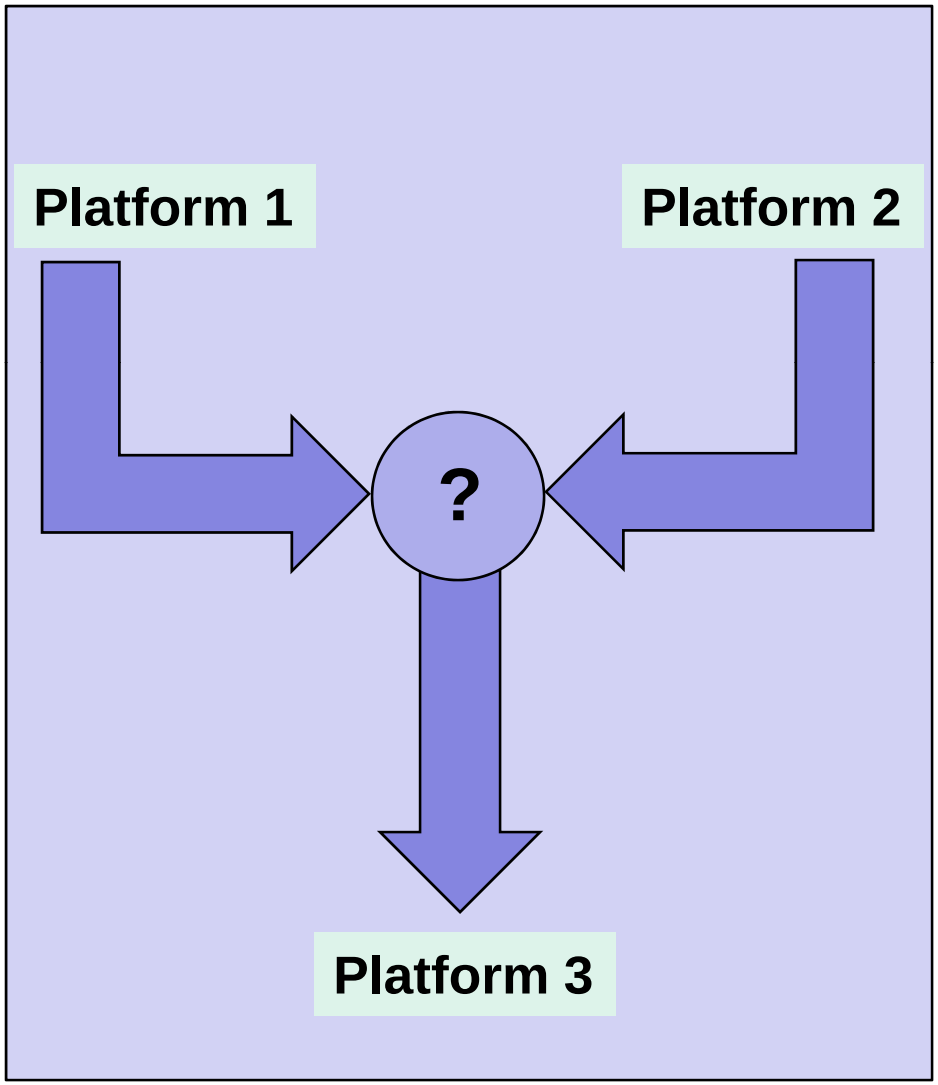


Behavior models define a set of timed automata with local clocks and broadcast. Models can be analyzed with TCTL.

A Basic Pattern in Design Flows

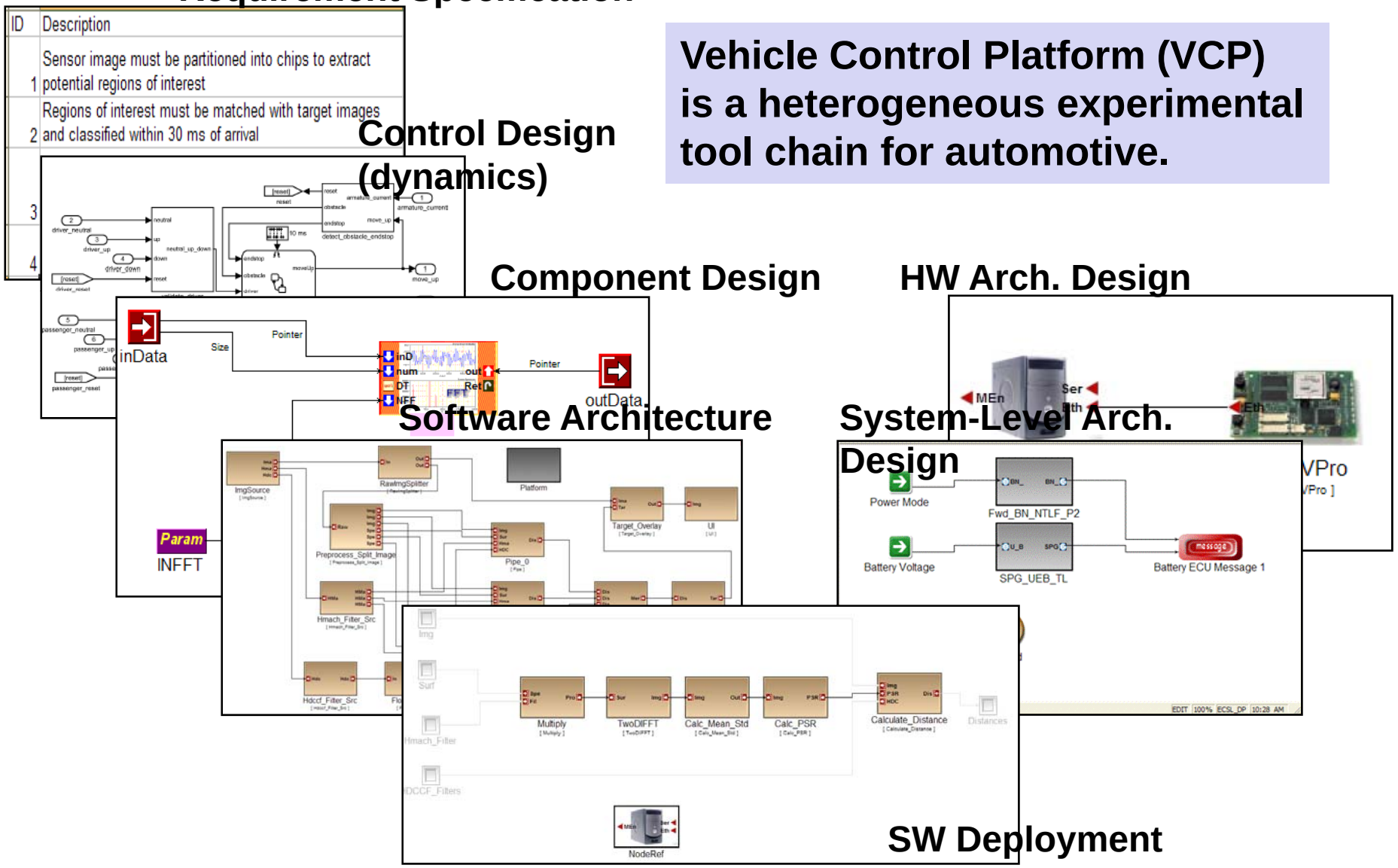


High-Level Design Flow for Embedded Systems: Alberto Sangiovanni-Vincentelli

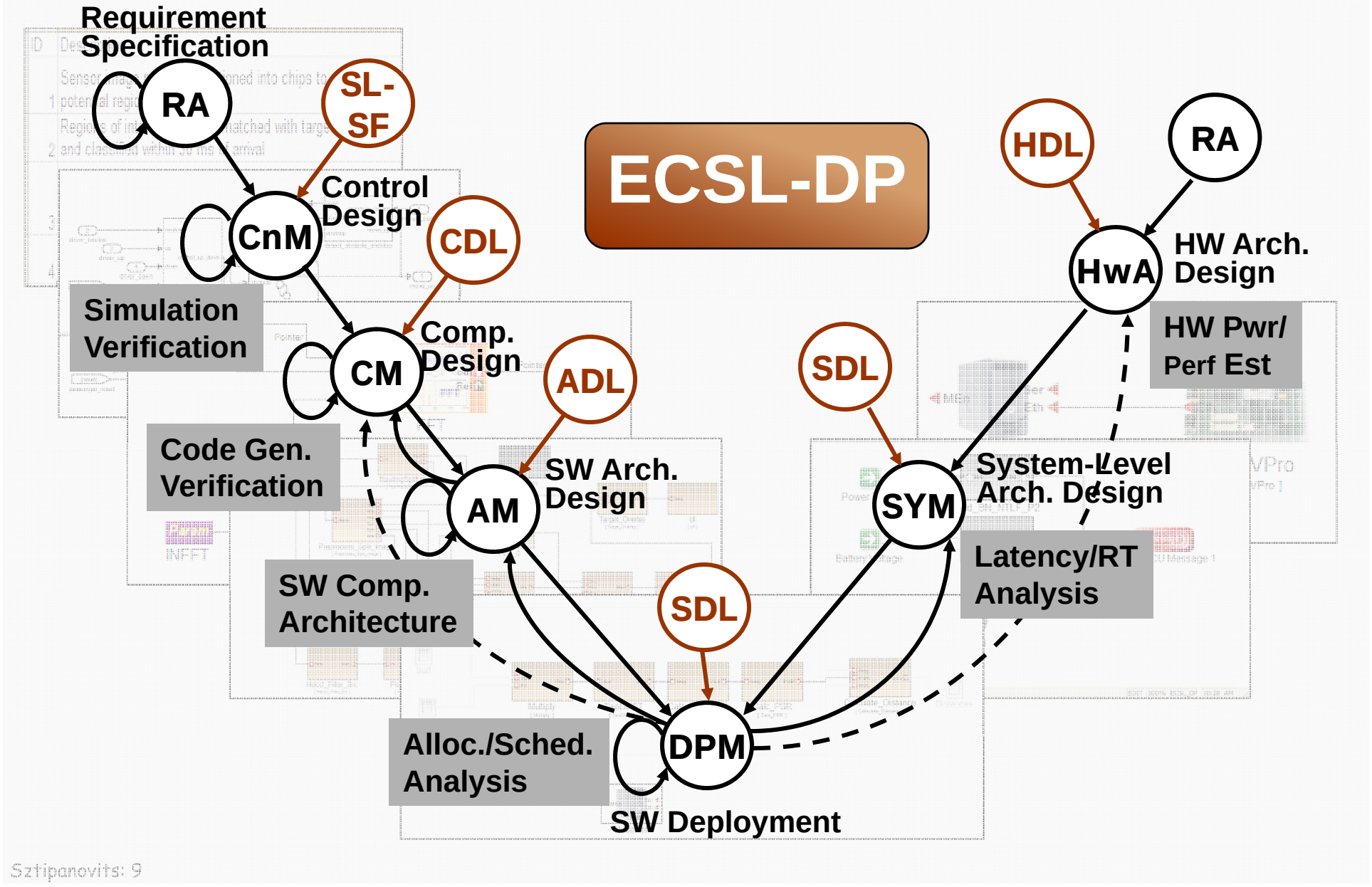


Example: A Simplified Design Flow for Automotive

Vehicle Control Platform (VCP) is a heterogeneous experimental tool chain for automotive.



Design Flow: Tools & Languages

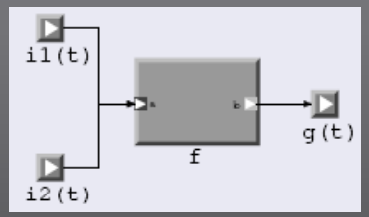


Content

- Introduction
  - Basic concepts: platforms, abstractions and DSML-s
- Model Integrated Computing
  - Structural and Behavioral Semantics
  - Metamodel composition
  - MIC Tool Suite
- Making Behavioral Semantics Explicit
  - Semantic anchoring
  - Remarks on composition

Model-Integrated Computing

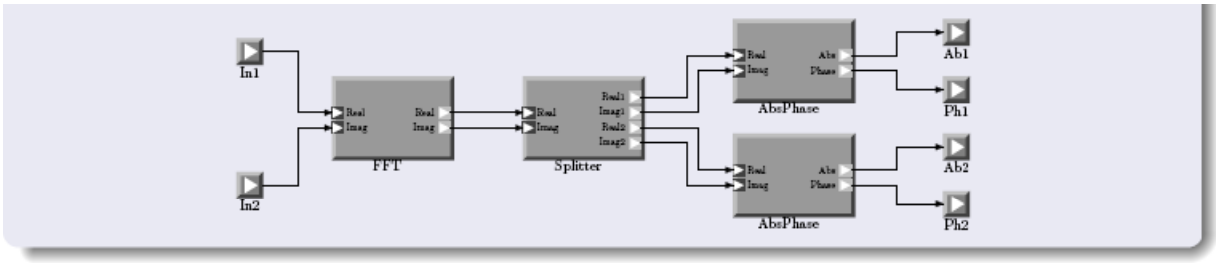
The structural semantics excludes semantically meaningless models.



No operator was provided for composition of values, so this merge model is semantically meaningless in this domain.

Key Idea: Capture intrinsic domain concepts with *domain-specific modeling languages* (DSML-s) and partition DSML-s into *structural* and *behavioral* semantics.

- The **structural semantics** views a model as a structure, and provides a means for calculating which structures are well-formed.



- The **behavioral semantics** defines what the structures do.

1. A block f represents an n-ary map over some value domain. $f : \mathcal{V}^n \to \mathcal{V}^m$.
2. A connection c represents an projection operator: $\pi_{i,m} : \mathcal{V}^m \to \mathcal{V}$, where $\pi_{i,m}(v_0, v_1, \ldots, v_{m-1}) \mapsto v_i$
3. composition is by function composition: $splitter(\pi_{0,2} \circ fft(in1(t), in2(t)), \pi_{1,2} \circ fft(in1(t), in2(t)))$.

Specification of Structural Semantics of DSML-s

Abstract syntax of DSML-s are defined by metamodels. Metamodeling languages provide structural semantics.

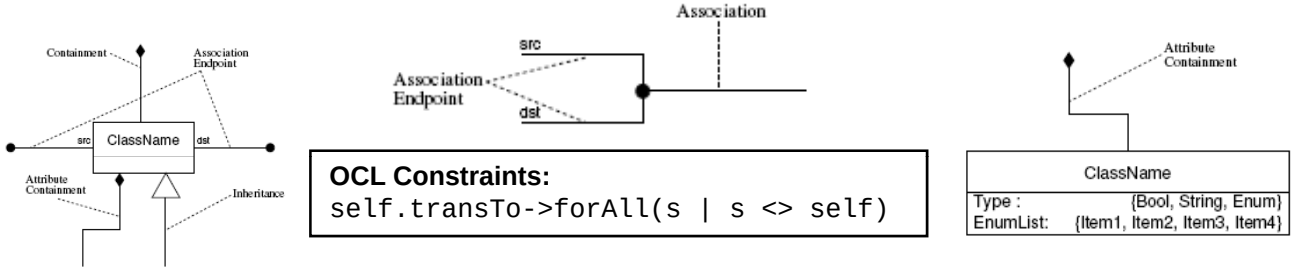
Mathematical framework for structural semantics: Terms Algebra
(Jackson, Sztipanovits EMSOFT 2006)

A metamodeling language is one of the DSML-s: the same tool can be used for modeling and metamodeling.

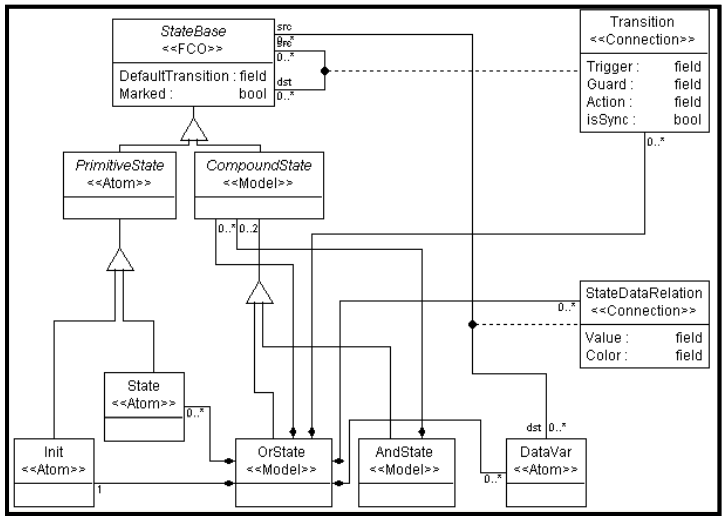
- Metamodels define the structural semantics of DSML-s.

$$L = \langle Y, R_Y, C, ([\ ]_i)_{i \in J} \rangle$$
$$D(Y, C) = \{ r \in R_Y \mid r \models C \}$$

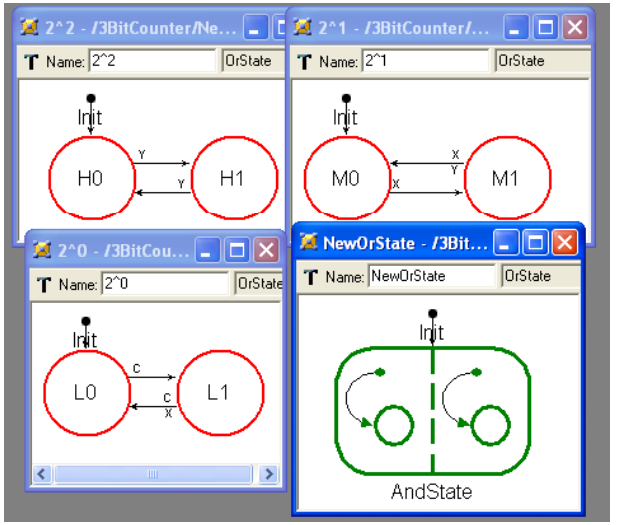
- GME, the metaprogrammable modeling tool of ISIS, supports rapid construction of metamodels and DSML models.



Basic metamodeling notation: UML Class Diagram + OCL



MetaGME metamodel of simple statecharts



Model-editor generated from metamodel

Specification of Behavioral Semantics of DSML-s

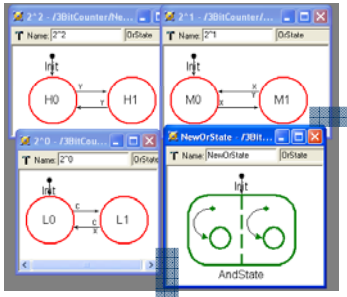
C++ coding permits complex behavioral semantics, but the "specifications" are cluttered with C++ details.

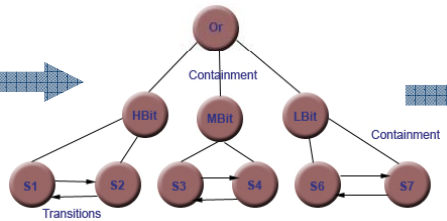
Graph transformations provide a transparent mechanism to attach semantics. However, not all behavioral semantics ca be specified this way.

Semantic anchoring with ASM captures the best of both worlds: Simple graph transformations and simple behavioral specifications.

- Behavioral semantics are defined with model transformations and semantic anchoring.

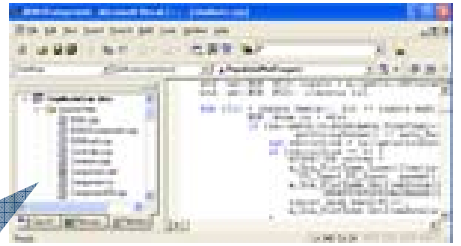
$$[\![\ ]\!]^T : R_Y \mapsto R_{Y'}$$





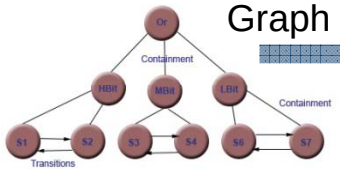
Representation as AST

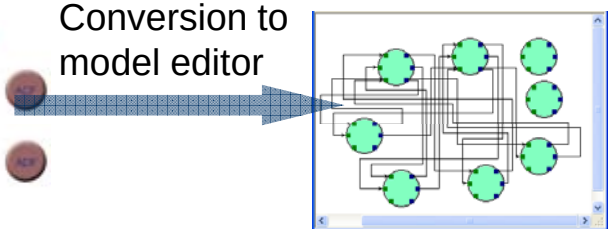
COM + APIs



C++ Interpreter/Generator

Graph rewriting rules





Conversion to model editor


```
fsmReact (fsm as FSM, e as Event) =
  step let s as State = getCurrentState (fsm, e)
  step let pt as Transition? = getPreemptiveTrasition (fsm, s, e)
  step
    if pt <> null then doTransition (fsm, s, pt)
    else
     step
       if isHierarchicalState (s) then invokeSlaves (fsm, s, e)
       let npt as Transition? = getNonpreemptiveTransition (fsm,s,e)
     step
       if npt <> null then doTransition (fsm, s, npt)
```

Canonical definition of FSM behavioral semantics in ASM

```
structure Event
    case on
    case sleep
    case shutdown
    case login
    case logout
ComputerStatus = new FSM([], OFF, {ON,OFF})
OFF      = new State(true, SO, null, {SO, POWEROFF, STANDBY}, {T1})
ON       = new State(false, LOGOUT, null, {LOGOUT, LOGIN}, {T3,T2})
SO       = new State(true, null, OFF, {}, {T12,T11})
POWEROFF = new State(false, null, OFF, {}, {})
STANDBY  = new State(false, null, OFF, {}, {T13})
LOGOUT   = new State(true, null, ON, {}, {T21})
LOGIN    = new State(false, null, ON, {}, {T22})
T1 = new Transition(true, false, Event.on, null, OFF, ON)
T2 = new Transition(true, false, Event.sleep, null, ON, OFF)
T3 = new Transition(true, false, Event.shutdown, null, ON, OFF)
T11= new Transition(true, false, Event.shutdown, null, SO, POWEROFF)
T12= new Transition(true, false, Event.sleep, null, SO, STANDBY)
T13= new Transition(true, false, Event.shutdown, null,STANDBY,POWEROFF)
T21= new Transition(true, false, Event.login, null, LOGOUT, LOGIN)
T22= new Transition(true, false, Event.logout, null, LOGIN, LOGOUT)
```

Translation of GME model to ASM data structures

- Simulation artifacts and test cases can be generated

Content

- Introduction
  - Basic concepts: platforms, abstractions and DSML-s
- Model Integrated Computing
  - Structural and Behavioral Semantics
  - Metamodel composition
  - MIC Tool Suite
- Making Behavioral Semantics Explicit
  - Semantic anchoring
  - Remarks on composition

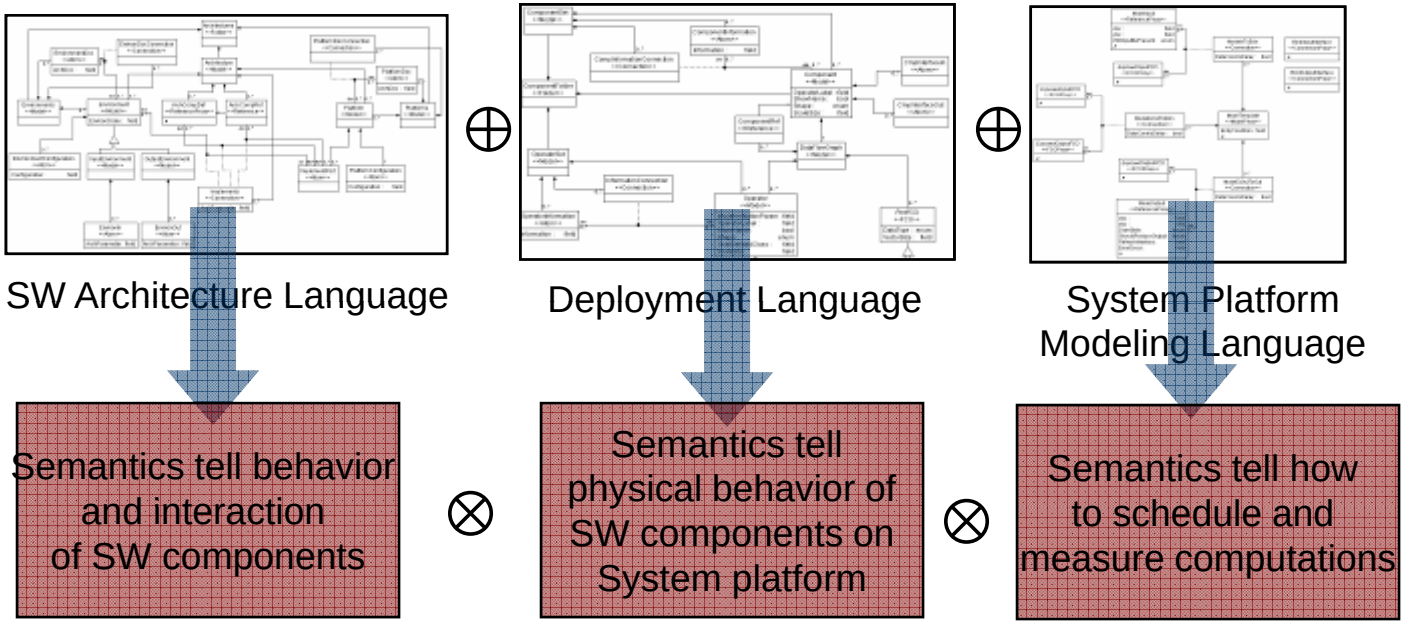
Composing DSML-s

Each view may represent a design concern (separation of concerns).

Design aspects are typically non orthogonal.

Composition of non-orthogonal DSML-s is a significant challenge for the tool infrastructure.

Key Idea: Decompose complex models into aspects. Aspects are composed to create complete specification.



- An architectural specification is created in each view
- The total system is the composition (synthesis) of all the views

Key Question: If properties hold in a view, do they hold in the entire system?

Key Question: How do engineers keep track of cross-aspect interactions?

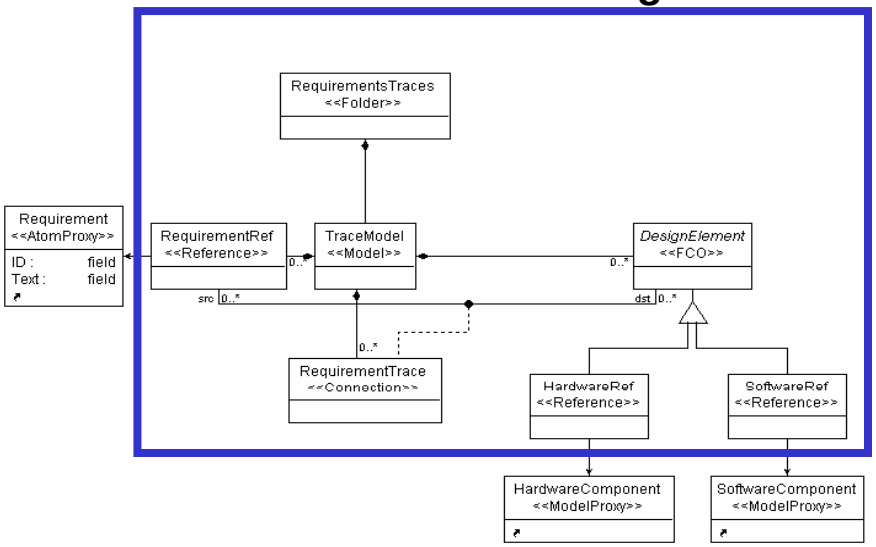
DSML Composition is an Essential Practical Tool

Tool support for DSML (metamodel) composition in GME:

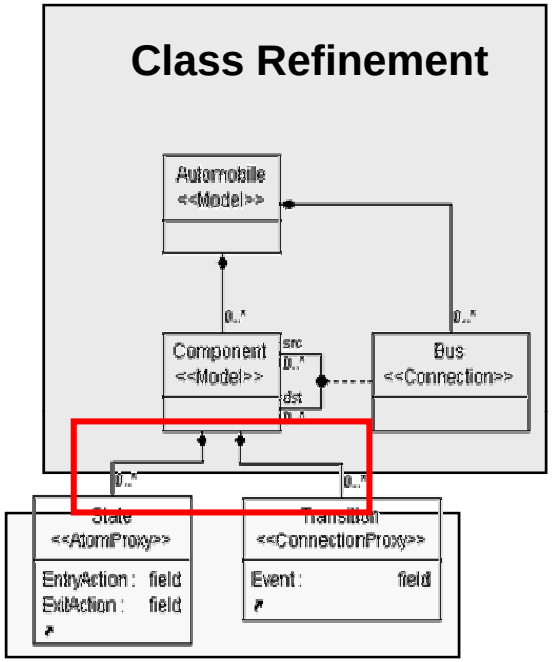
- *Class merge*
- *Metamodel interfacing*
- *Class refinement*
- *Template instantiation*
- *Model transformation*

Understanding structural and behavioral semantics of composed DSML-s is essential.

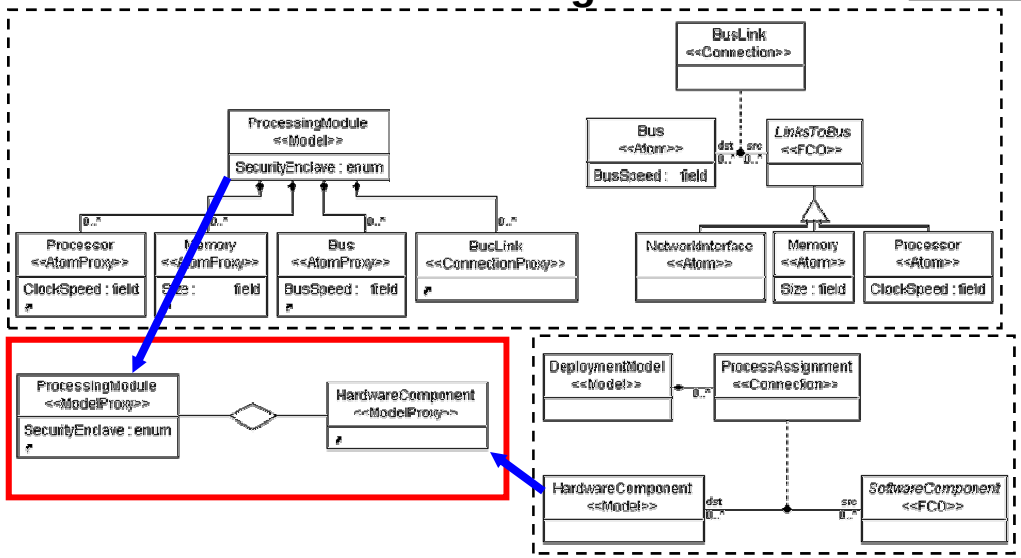
Metamodel Interfacing



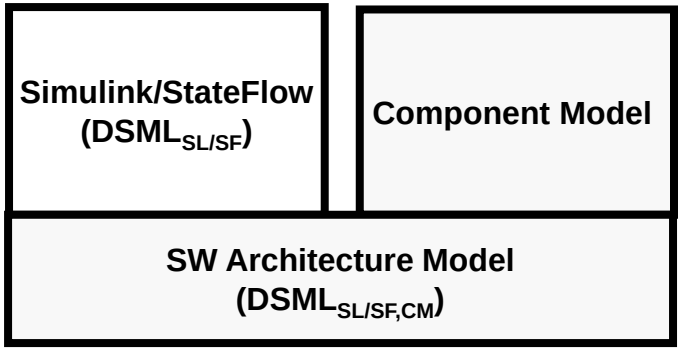
Class Refinement



Class Merge



Embedded SW Architecture Design

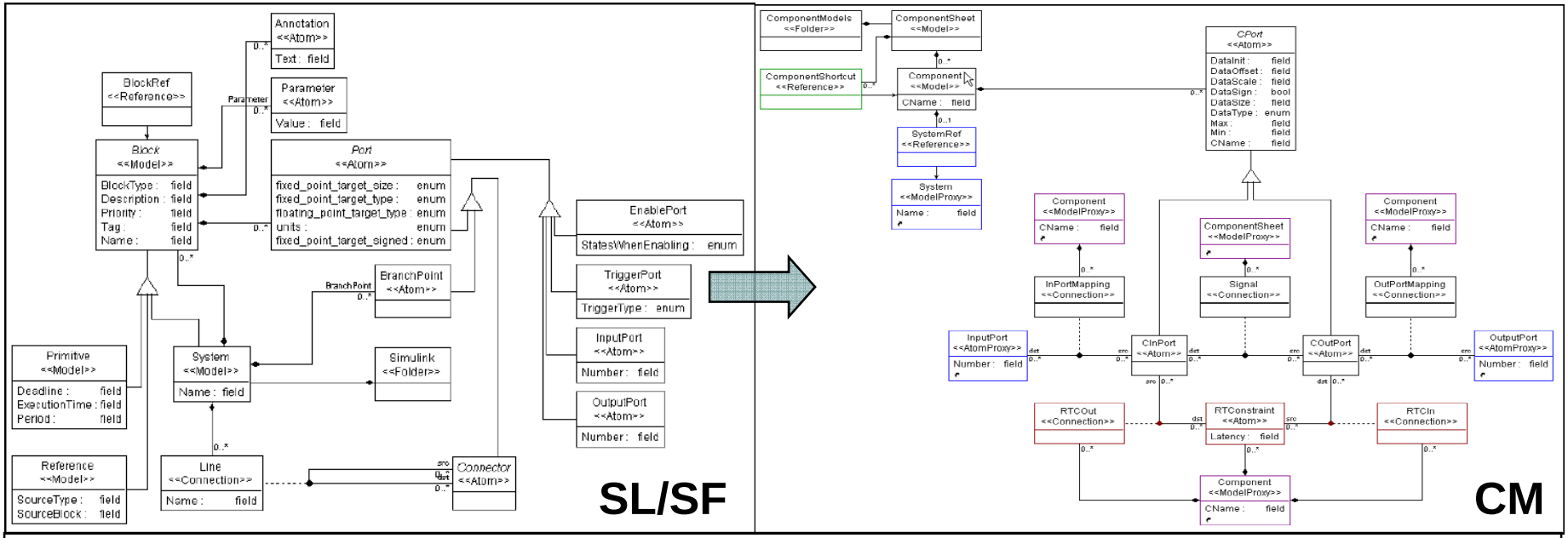


Objective: Optimize the SW architecture by selecting a component model and by allocating functions to components.

Platform: TT Component Model

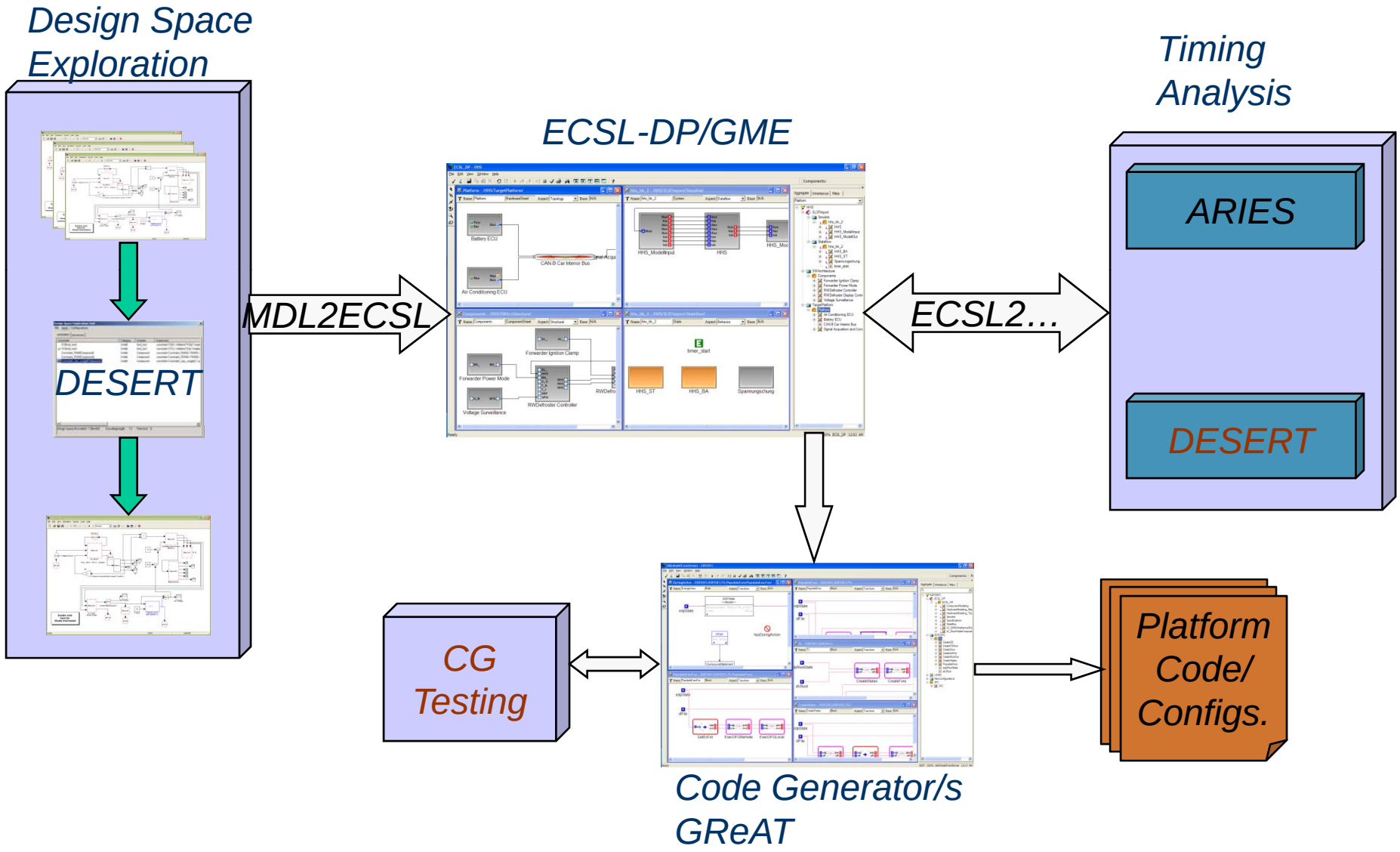
Tools:

GME, GReAT (SL/SF to C generator), C Compiler, WCET Analyzer



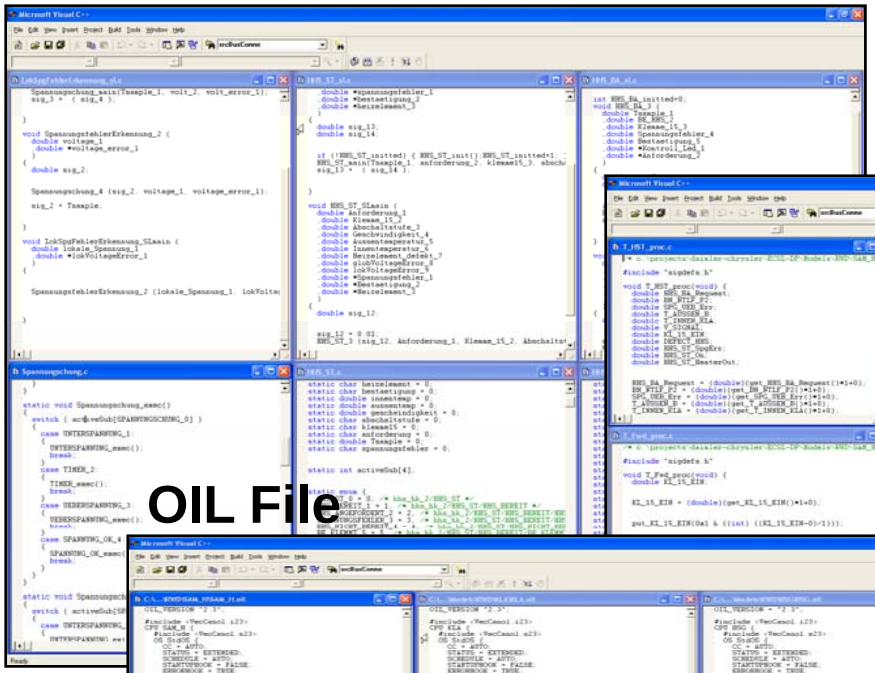
Functional blocks – SW Component Mapping

VCP Tool-Chain: Tools and Interfaces

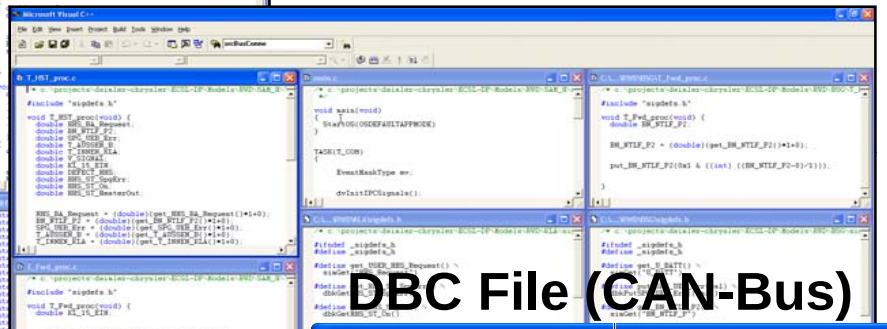


VCP Tool Chain Output

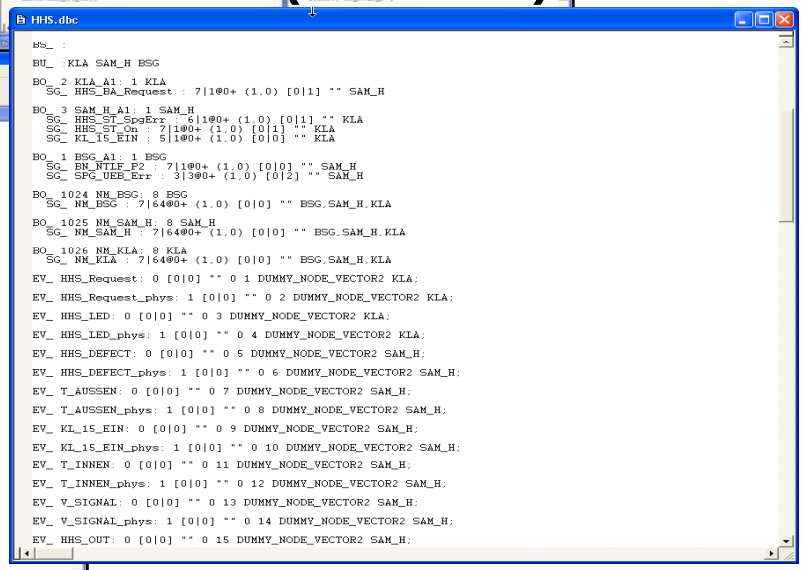
Functional Code



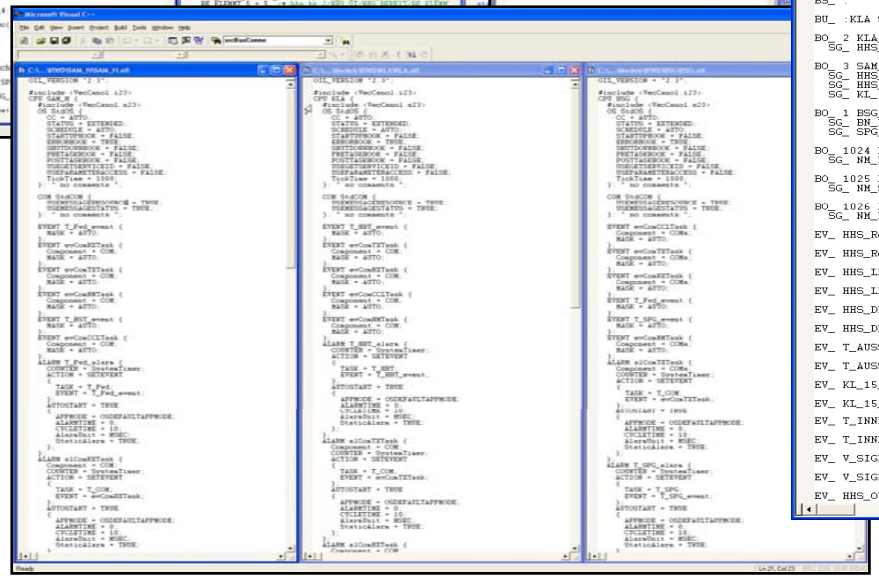
OS/Firmware Glue Files



DBC File (CAN-Bus)



OIL File



Content

- Introduction
  - Basic concepts: platforms, abstractions and DSML-s
- Model Integrated Computing
  - Structural and Behavioral Semantics
  - Metamodel composition
  - MIC Tool Suite
- Making Behavioral Semantics Explicit
  - Semantic anchoring
  - Remarks on composition

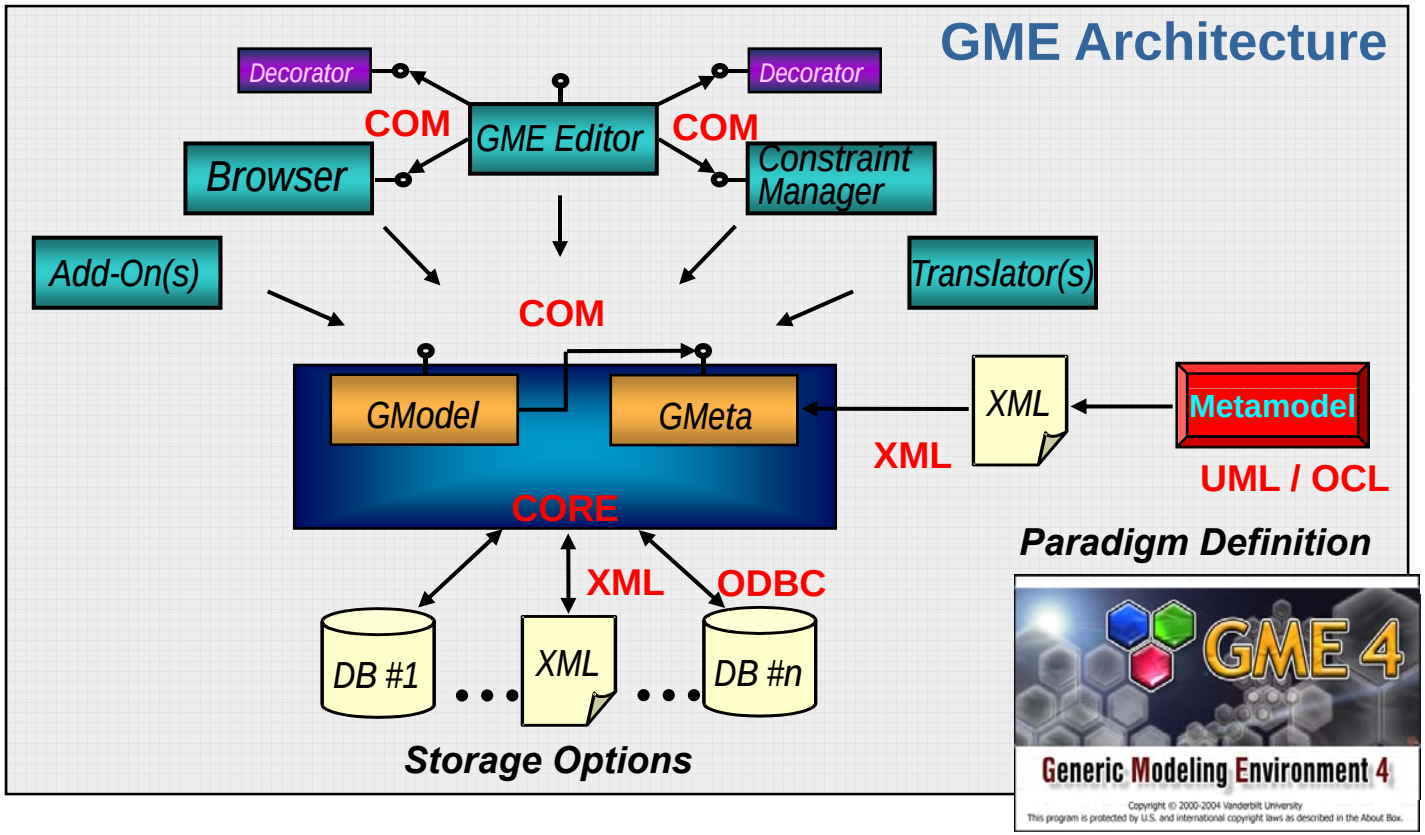
GME: Metaprogrammable Modeling Tool

The Generic Modeling Environment (GME) is used for building metamodels and is customized by metamodels (metaprogrammability).

MetaGME combines language constructs for metamodeling and for specifying concrete syntax of DSML-s.

Metacircularity enables changing metalanguages without changing the tool.

The metamodel for MetaGME is the meta-metamodel – defined in MetaGME.



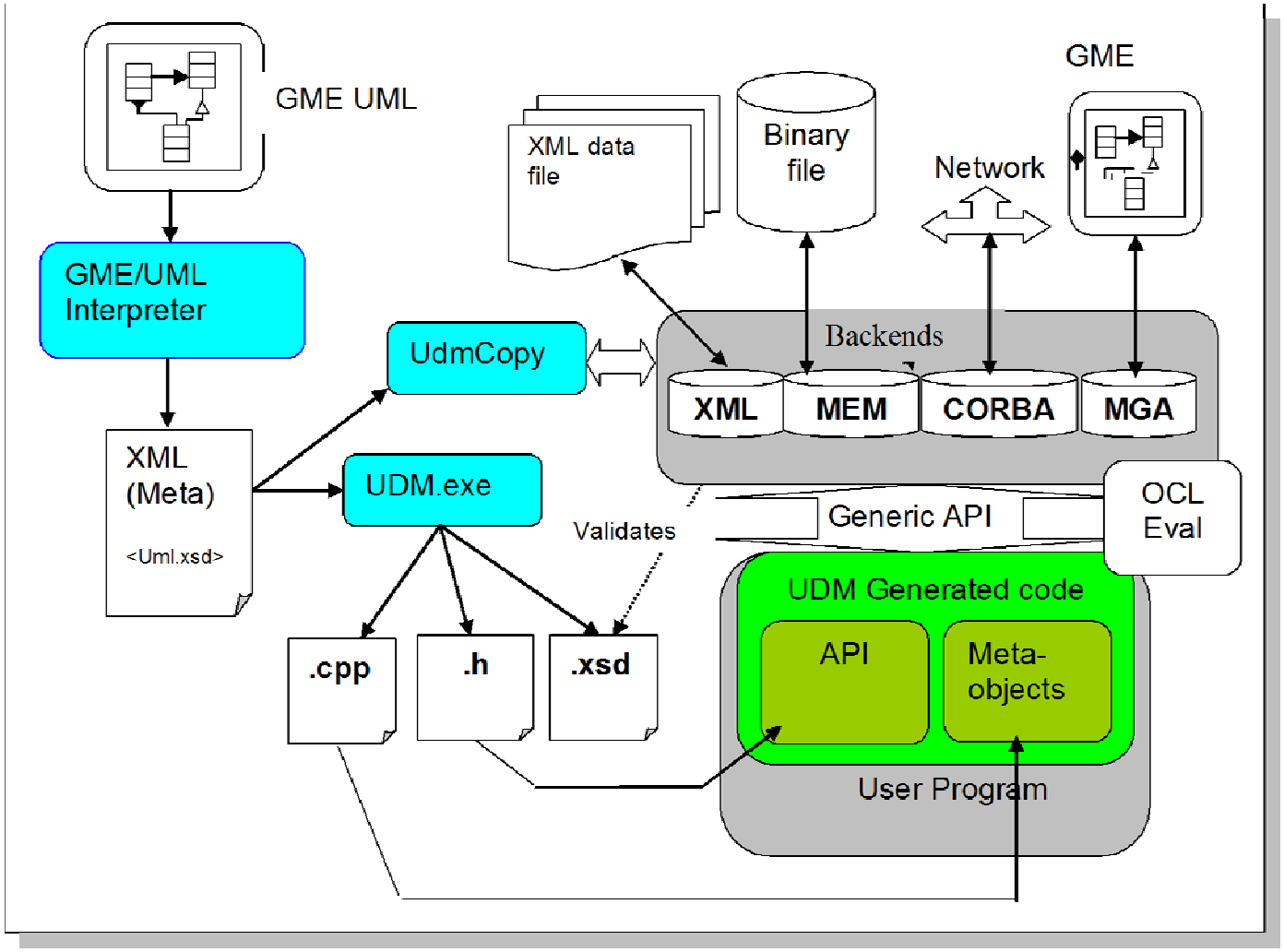
- **Configuration through UML and OCL-based metamodels**
- **Extensible architecture through COM**
- **Multiple standard backend support (ODBC, XML)**
- **Multiple language support: C++, VB, Python, Java, C#**

Model Data Management: The UDM Tool Suite

The goal of UDM is to have a conceptual view of data/metadata that is independent of the storage format.

UDM provides uniform access to data/metadata such that storage formats can be changed seamlessly at either design time or run time

UDM generates a metadata/paradigm specific API to access a particular class of data.

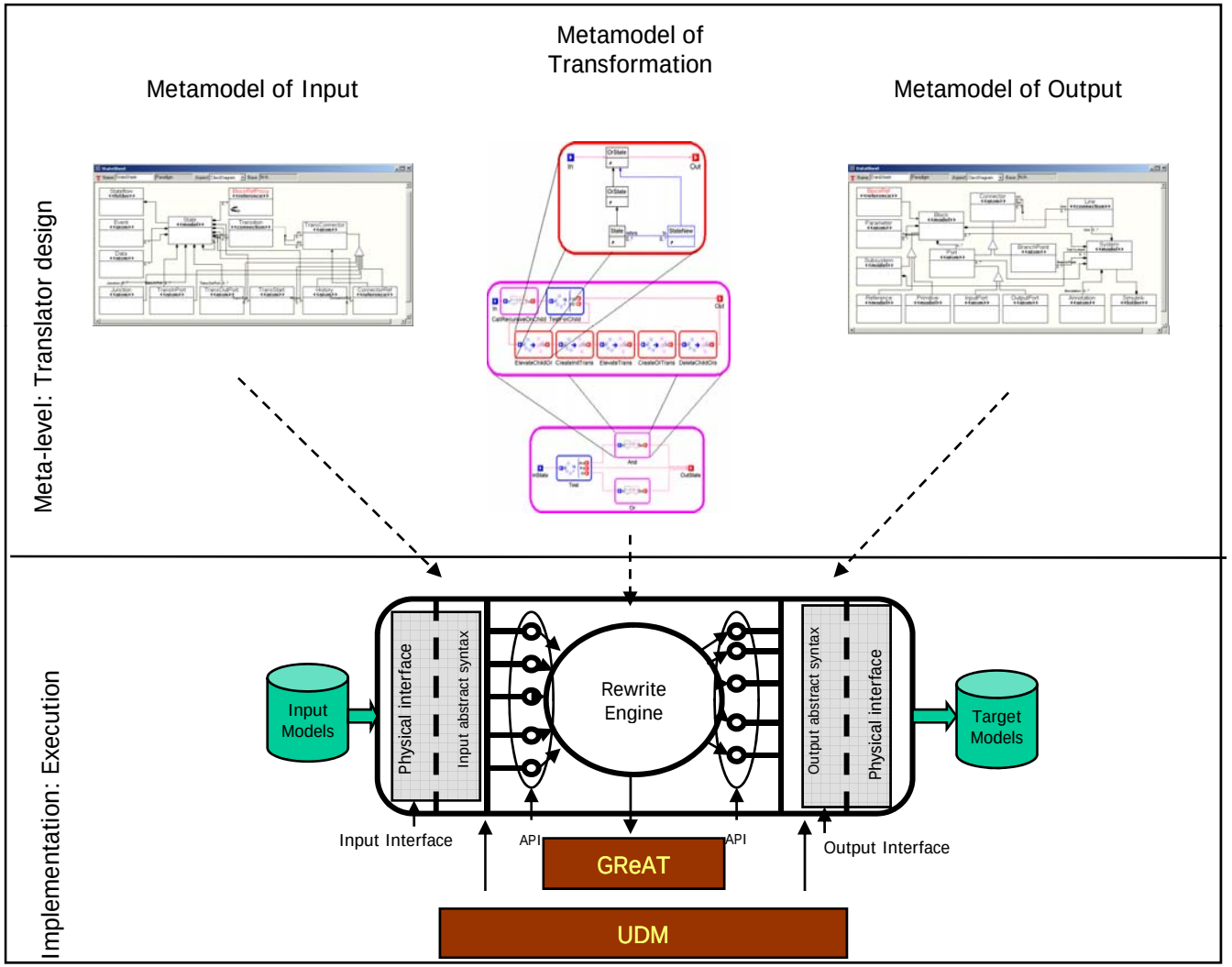


Model Transformation: The "Workhorse" of MIC

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.

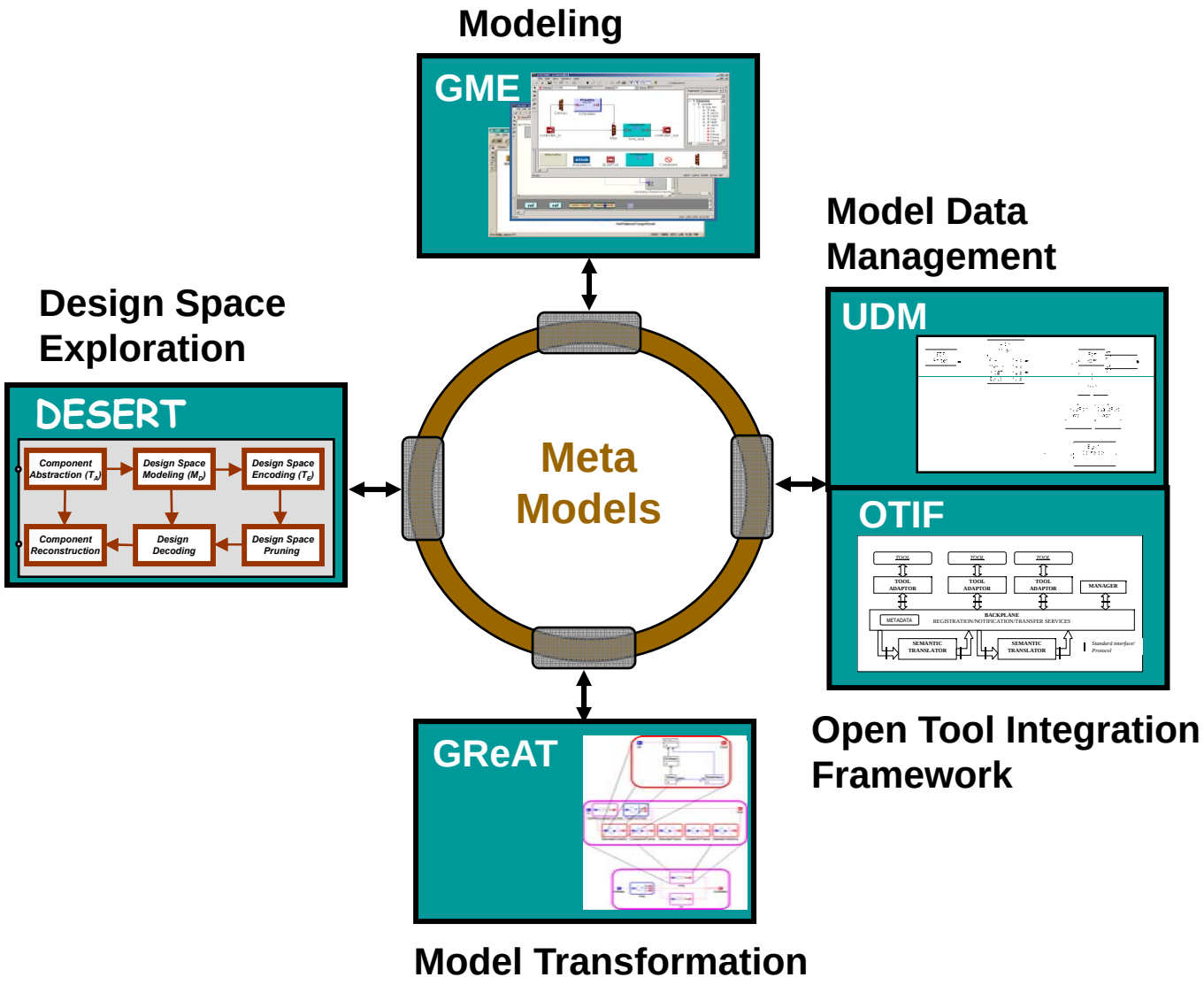
Models of transformations are expressed in a DSML and built in GME.



Integrated MIC Tool Suite

The MIC tool suite is metaprogrammable: each component can be customized to domain specific modeling languages by means of metamodels.

All tools are available from a quality controlled open source repository.



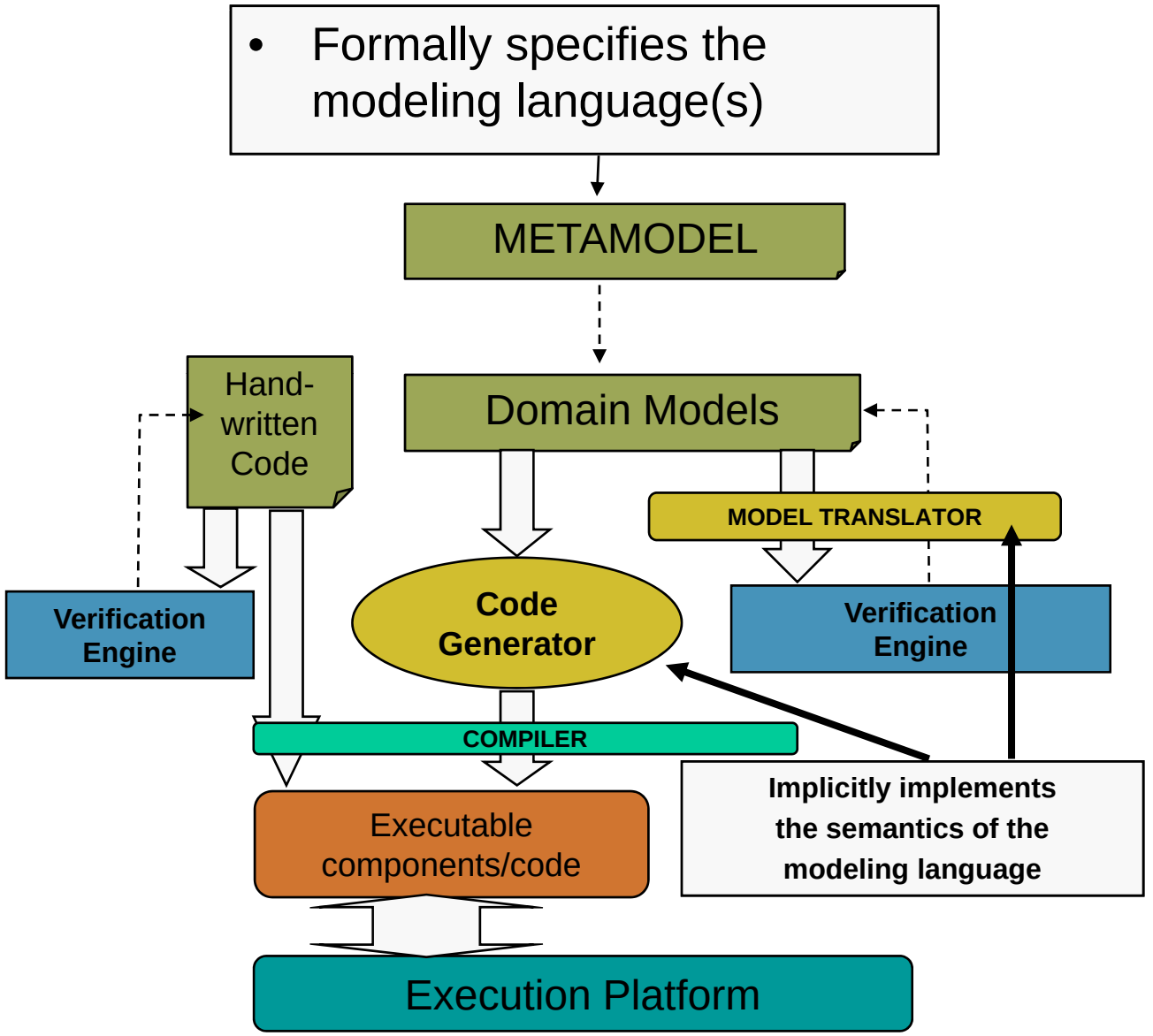
ESCHER Quality Controlled Repository: http://escher.isis.vanderbilt.edu

Model-based Software Development

Essential questions:

How do we know that the model transformations (model translator/code generator) are correct?

How do we know that the verified properties of the models are preserved by the generated code running on the selected execution platform?



Content

- Introduction
  - Basic concepts: platforms, abstractions and DSML-s
- Model Integrated Computing
  - Structural and Behavioral Semantics
  - Metamodel composition
  - MIC Tool Suite
- Making Behavioral Semantics Explicit
  - Semantic anchoring
  - Remarks on composition

Specification of Behavioral Semantics of DSML-s

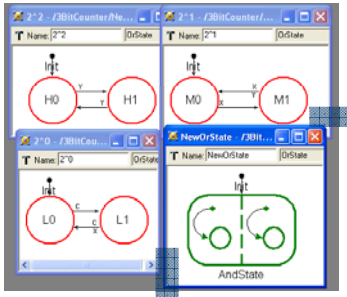
C++ coding permits complex behavioral semantics, but the "specifications" are cluttered with C++ details.

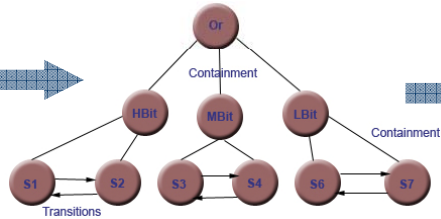
Graph transformations provide a transparent mechanism to attach semantics. However, not all behavioral semantics ca be specified this way.

Semantic anchoring with ASM captures the best of both worlds: Simple graph transformations and simple behavioral specifications.

- Behavioral semantics are defined with model transformations and semantic anchoring.

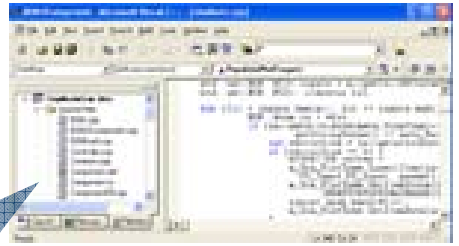
$$[\![\ ]\!]^T : R_Y \mapsto R_{Y'}$$





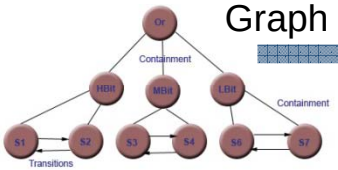
Representation as AST

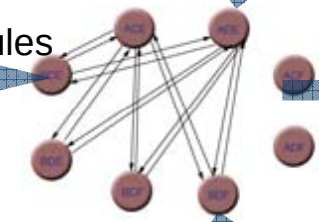
COM + APIs



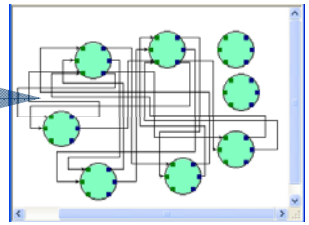
C++ Interpreter/Generator

Graph rewriting rules





Conversion to model editor



```
fsmReact (fsm as FSM, e as Event) =
  step let s as State = getCurrentState (fsm, e)
  step let pt as Transition? = getPreemptiveTrasition (fsm, s, e)
  step
    if pt <> null then doTransition (fsm, s, pt)
    else
     step
       if isHierarchicalState (s) then invokeSlaves (fsm, s, e)
       let npt as Transition? = getNonpreemptiveTransition (fsm,s,e)
     step
       if npt <> null then doTransition (fsm, s, npt)
```

Canonical definition of FSM behavioral semantics in ASM

```
structure Event
    case on
    case sleep
    case shutdown
    case login
    case logout
ComputerStatus = new FSM([], OFF, {ON,OFF})
OFF      = new State(true, SO, null, {SO, POWEROFF, STANDBY}, {T1})
ON       = new State(false, LOGOUT, null, {LOGOUT, LOGIN}, {T3,T2})
SO       = new State(true, null, OFF, {}, {T12,T11})
POWEROFF = new State(false, null, OFF, {}, {})
STANDBY  = new State(false, null, OFF, {}, {T13})
LOGOUT   = new State(true, null, ON, {}, {T21})
LOGIN    = new State(false, null, ON, {}, {T22})
T1 = new Transition(true, false, Event.on, null, OFF, ON)
T2 = new Transition(true, false, Event.sleep, null, ON, OFF)
T3 = new Transition(true, false, Event.shutdown, null, ON, OFF)
T11= new Transition(true, false, Event.shutdown, null, SO, POWEROFF)
T12= new Transition(true, false, Event.sleep, null, SO, STANDBY)
T13= new Transition(true, false, Event.shutdown, null,STANDBY,POWEROFF)
T21= new Transition(true, false, Event.login, null, LOGOUT, LOGIN)
T22= new Transition(true, false, Event.logout, null, LOGIN, LOGOUT)
```

Translation of GME model to ASM data structures

- Simulation artifacts and test cases can be generated

Search for a Formal Framework

- Specification style: Operational semantics
- Solid mathematical foundation
- Tool support for core use cases:
  - Readability (clear syntax and “good-enough” semantics)
  - Validation/exploration of semantics (executable specification)
  - Verification of semantic equivalence (generation of “reference traces”, integratability)

After evaluating several frameworks (Z, TLA+,..)

we selected ASM and the AsmL tool suite.

AsmL has been developed at MSR (Gurevich)

Example Specification : FSM

Abstract Data Model

```
structure Event
  eventType as String
class State
  initial    as Boolean
  var active as Boolean = false
class Transition
abstract class FSM
  abstract property states           as Set of State
    get
  abstract property transitions      as Set of Transition
    get
  abstract property outTransitions   as Map of
    <State, Set of Transition>
    get
  abstract property dstState as Map of <Transition, State>
    get
  abstract property triggerEventType as Map of
    <Transition, String>
    get
  abstract property outputEventType  as Map of
    <Transition, String>
    get
```

Interpreter

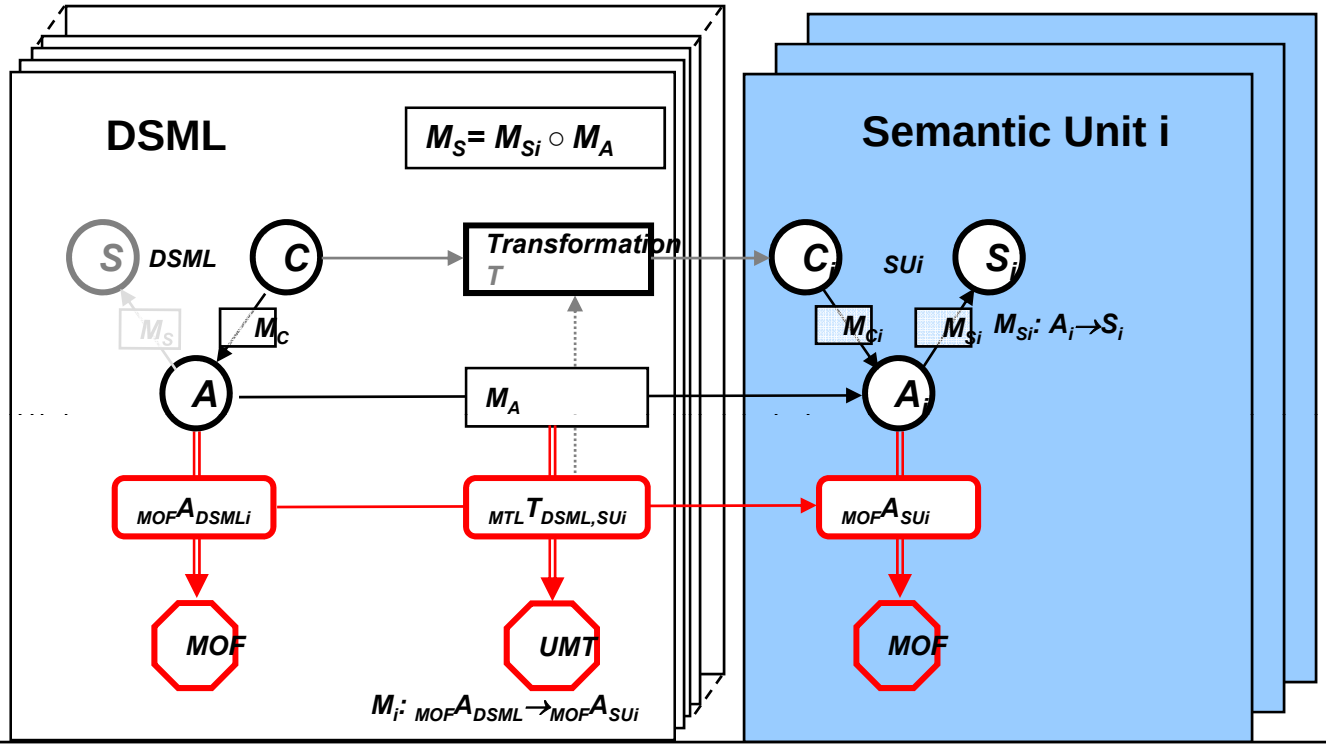
```
abstract class FSM
  Run (e as Event) as Event?
    step
      let CS as State = GetCurrentState ()
    step
      let enabledTs as Set of Transition = {t | t in
        outTransitions (CS) where e.eventType =
        triggerEventType(t)}
    step
      if Size (enabledTs) >= 1 then
        choose t in enabledTs
          step
            CS.active := false
          step
            dstState(t).active := true
          step
            if t in me.outputEventType then
              return Event(outputEventType(t))
            else
              return null
      else
        return null
```

Underlying abstract machine: ASM
Language: AsmL

Content

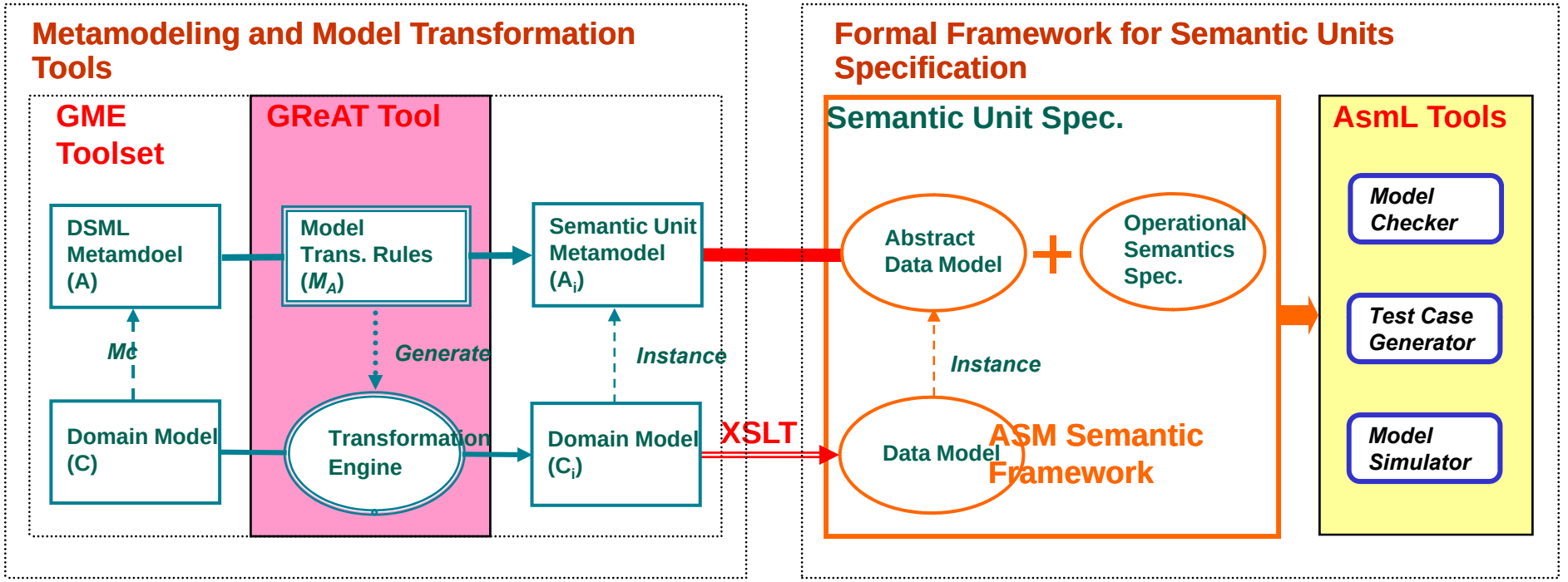
- Introduction
  - Basic concepts: platforms, abstractions and DSML-s
- Model Integrated Computing
  - Structural and Behavioral Semantics
  - Metamodel composition
  - MIC Tool Suite
- Making Behavioral Semantics Explicit
  - Semantic anchoring
  - Remarks on composition

DSML Design Through Semantic Anchoring



- Step 1
  - Specify the DSML by using metamodels.
- Step 2
  - Select appropriate semantic units for the behavioral aspects of the DSML.
- Step 3
  - Specify the semantic anchoring M_A = A -> A_i by using UMT.

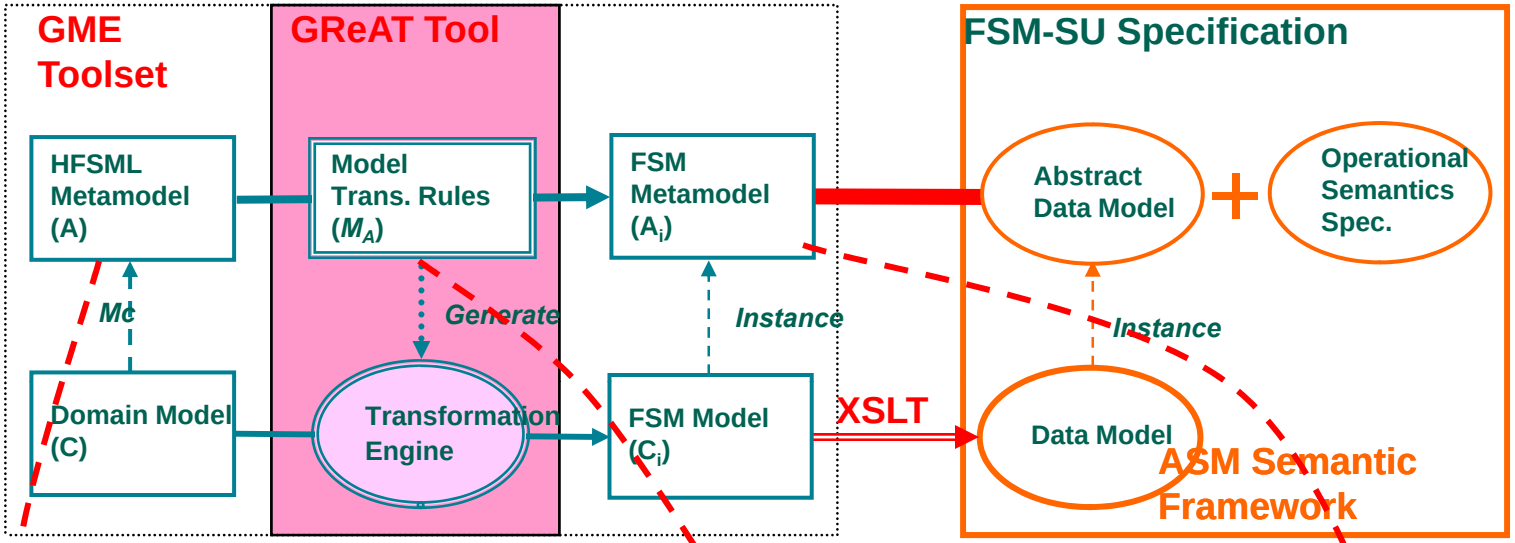
Experimental Tool Suite for Semantic Anchoring

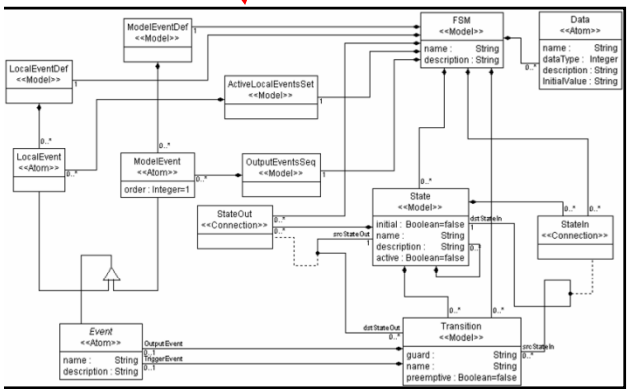


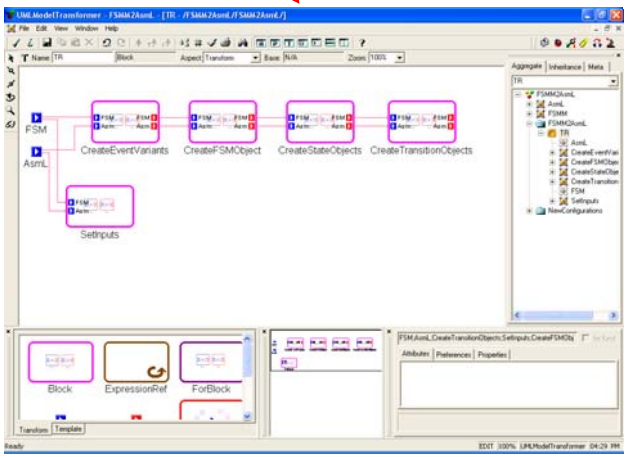
- Metamodeling and Model Transformation Tools
  - GME: Provide a MOF-based metamodeling and modeling environment.
  - GReAT: Build on GME for metamodel to metamodel transformation.

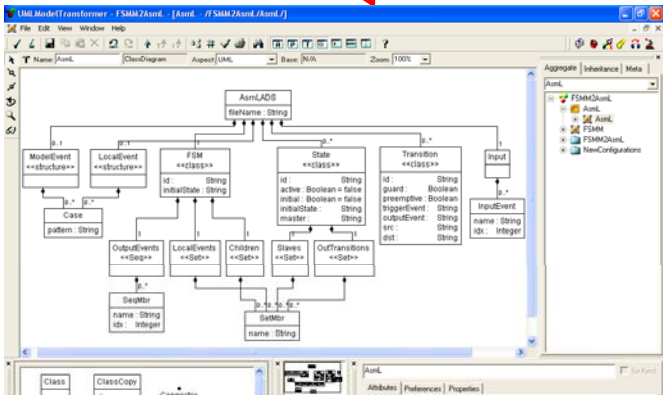
- Tools for Semantic Unit Specification
  - ASM: A particular kind of mathematical machine, like the Turing machine. (Yuri Gurevich)
  - AsmL: A formal specification language based on ASM. (Microsoft Research)

Example: HFSML => FSM-SU

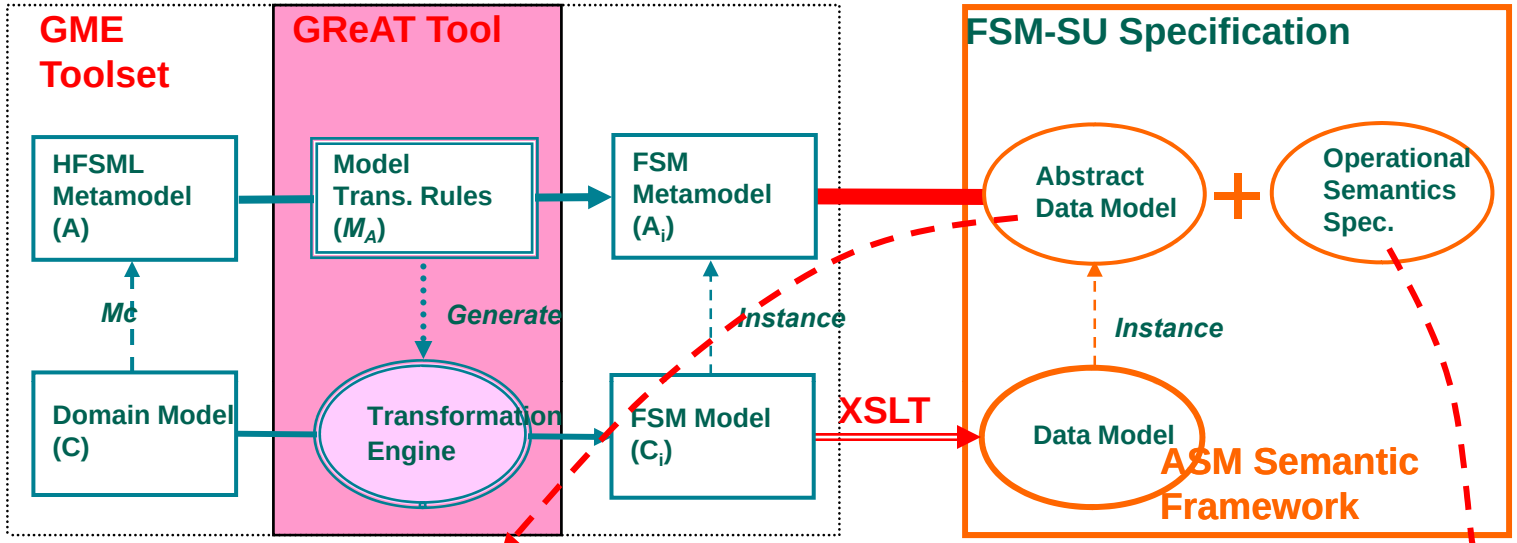








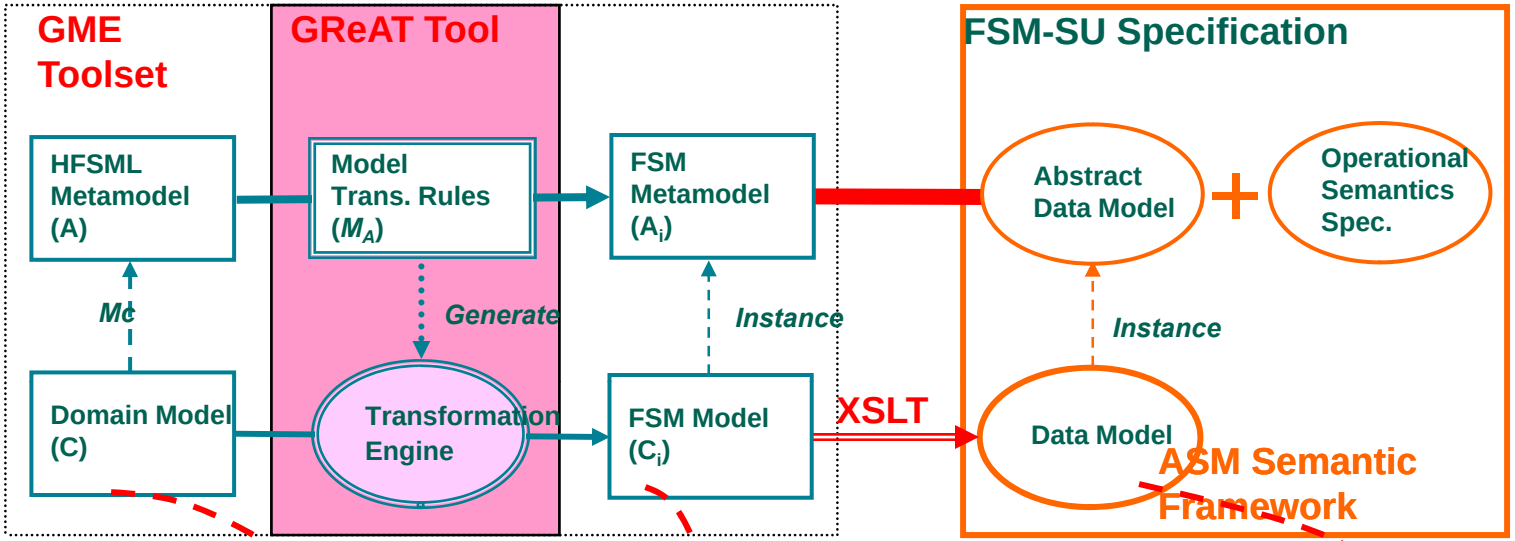
Example: HFSML => FSM-SU

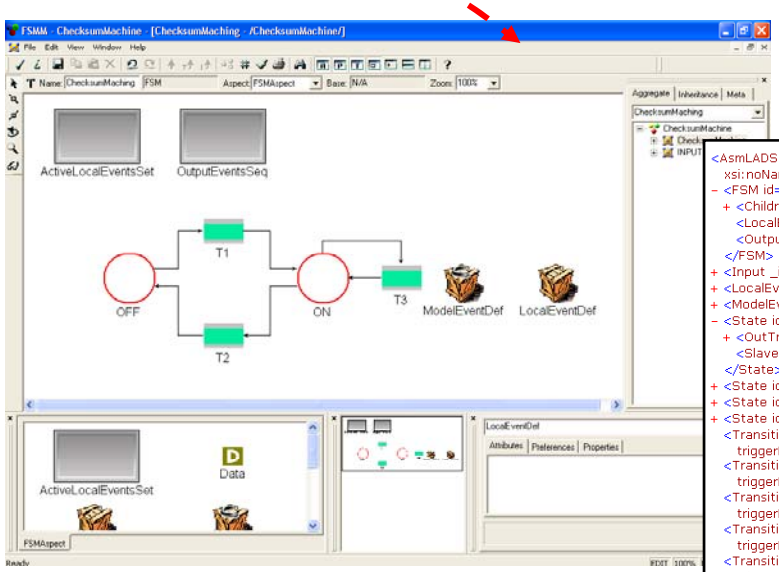



```
structure Event
  eventType as String

class State
  id         as String
  initial    as Boolean
  var active as Boolean = false

class Transition
  id as String

abstract class FSM
  id as String

  abstract property states as Set of State
    get
  abstract property transitions as Set of Transition
    get
  abstract property outTransitions as Map of <State, Set of Transition>
    get
  abstract property dstState as Map of <Transition, State>
    get
  abstract property triggerEventType as Map of <Transition, String>
    get
  abstract property outputEventType as Map of <Transition, String>
```

```
React (e as Event) as Event?
   step
     let CS as State = GetCurrentState ()
   step
     let enabledTs as Set of Transition = {t | t in outTransitions (CS) where
e.eventType = triggerEventType(t)}
   step
     if Size (enabledTs) = 1 then
       choose t in enabledTs
         step
          // WriteLine ("Execute transition: " + t.id)
           CS.active := false
         step
           dstState(t).active := true
         step
           if t in me.outputEventType then
             return Event(outputEventType(t))
           else
             return null
     else
       if Size(enabledTs) > 1 then
         error ("NON-DETERMINISM ERROR!")
       else
         return null
```

Example: HFSML => FSM-SU



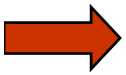



```
<AsmLADS _id="id988" fileName="" xmlns:xsi="http://www.w3.org/
  xsi:noNamespaceSchemaLocation="UDM\AsmL.xsd">
- <FSM id="ChecksumMaching" _id="id9d5" initialState="OFF">
  + <Children _id="id9f4">
    <LocalEvents _id="id9e5" />
    <OutputEvents _id="id9e0" />
  </FSM>
+ <Input _id="id98b">
+ <LocalEvent _id="id9bf">
+ <ModelEvent _id="id9ad">
- <State id="OFF" _id="ida17" active="false" master="" initial="true"
  + <OutTransitions _id="ida5b">
    <Slaves _id="ida3d" />
  </State>
+ <State id="ON" _id="ida18" active="false" master="" initial="false"
+ <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>
```

```
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)
```

Content

- Introduction
  - Basic concepts: platforms, abstractions and DSML-s
- Model Integrated Computing
  - Structural and Behavioral Semantics
  - Metamodel composition
  - MIC Tool Suite
- Making Behavioral Semantics Explicit
  - Semantic anchoring
  - Remarks on composition

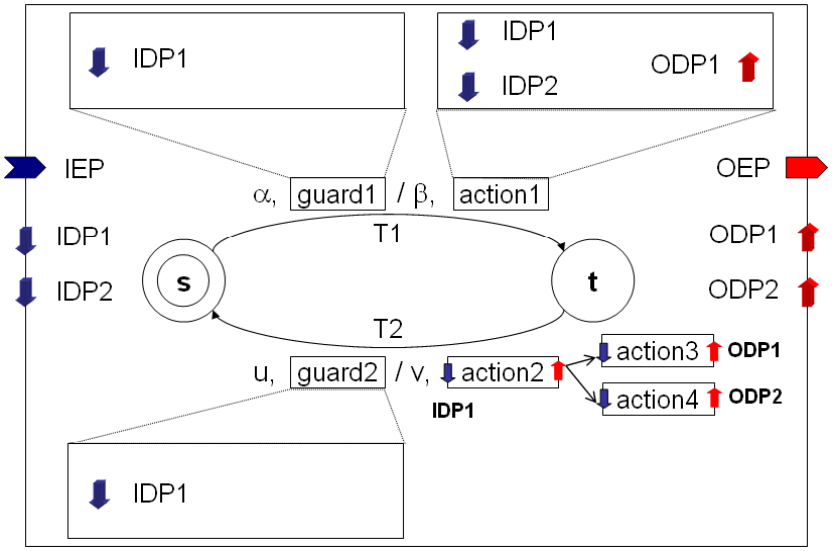


Heterogeneous DSMLs

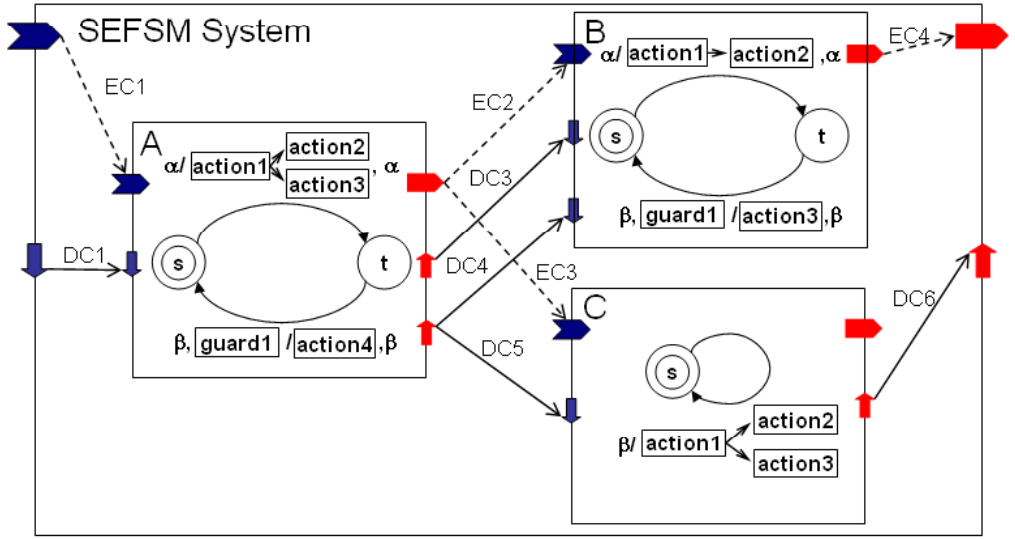
- Heterogeneity of systems
  - Complex systems are composed of heterogeneous components using heterogeneous interactions. Modeling and design of heterogeneous systems is a significant challenge.
- Heterogeneity of tool chains
  - Tool chains supporting domain-specific design flows integrate modeling, analysis and synthesis tools using DSMLs with overlapping semantics.
- The semantics of a heterogeneous DSML is probably not captured by a single predefined semantic unit.

Example: EFSM

- EFSM has been developed by General Motors Research to specify vehicle motion control (VMC) software.
- The SEFSM model is a synchronous reactive system including a set of components communicating through event channels and data channels.
- A SEFSM component is an FSM-based model, which integrates a set of stateless computational functions that consume input data and produce output data.
- Events determine which components are to be activated and the order of activations.

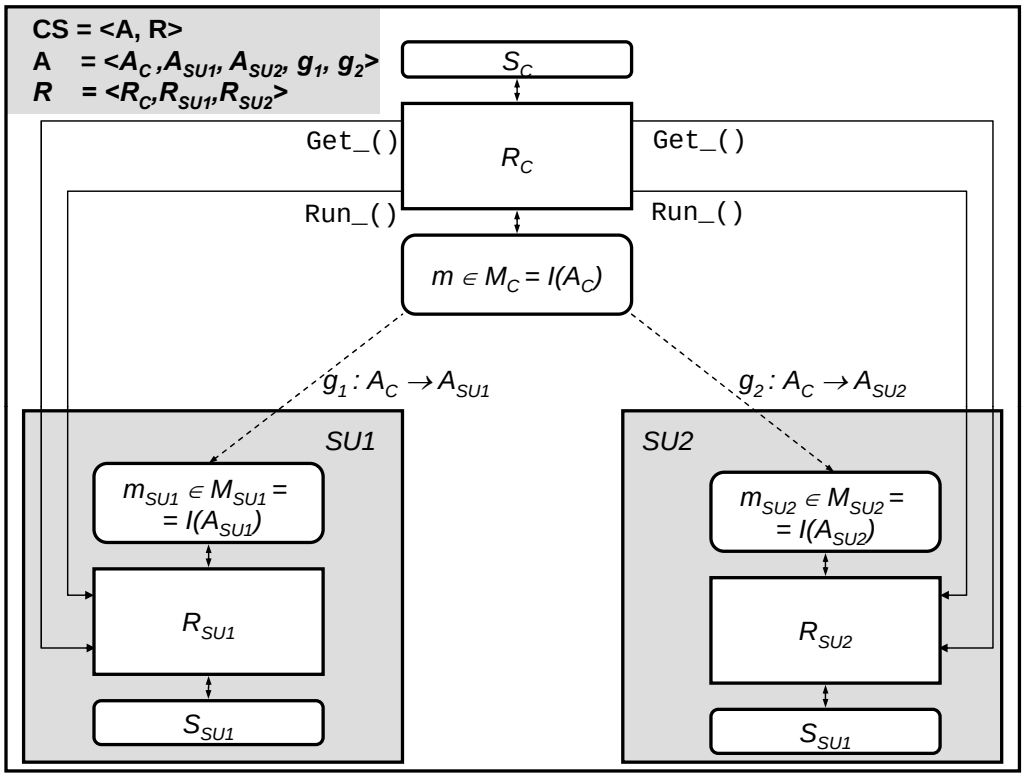


A SEFSM Component Model



A SEFSM System Model

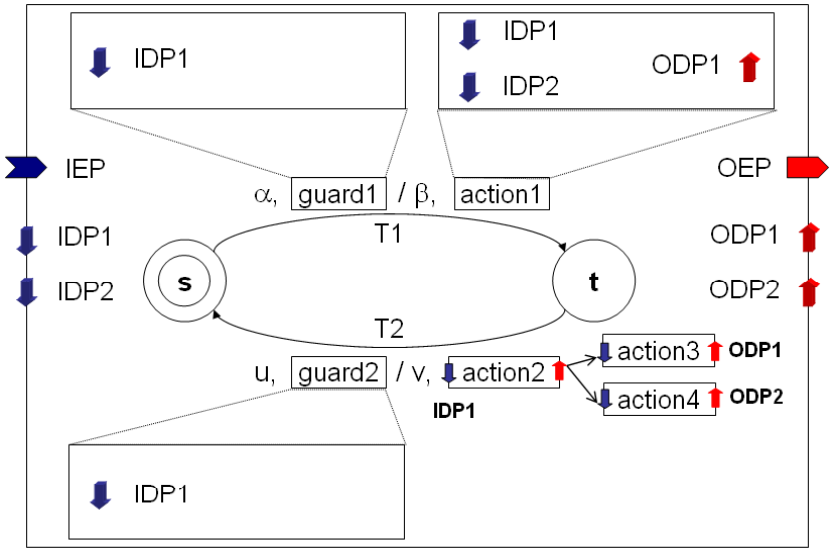
Modular Specification of Semantics



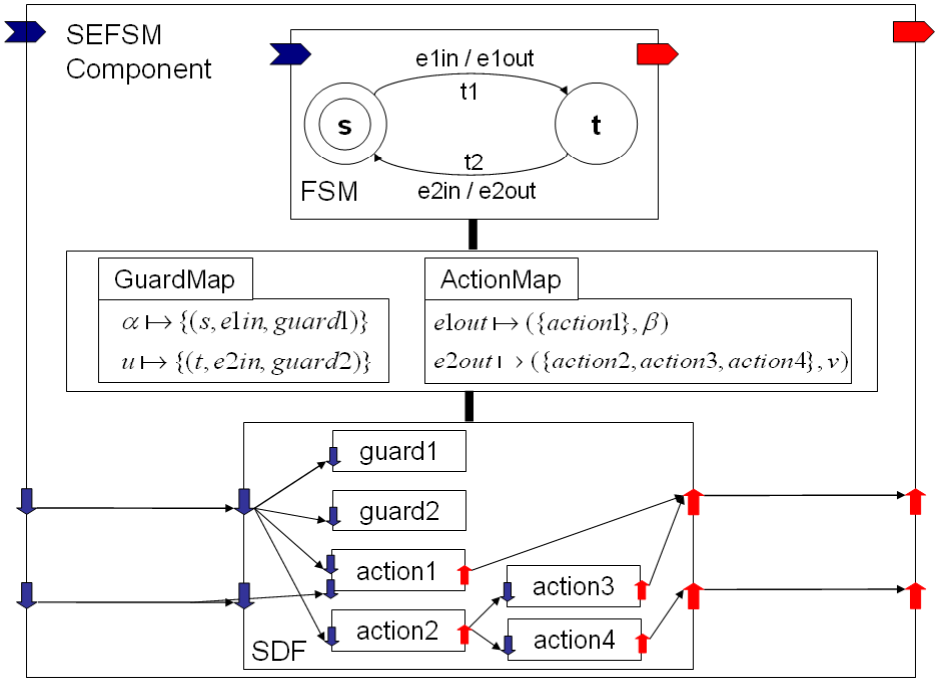
Remark: The behavioral composition specifies a controller, which restricts the executions of actions. Since the behavior of the embedded semantic units can be described as partial orders on the sets of actions they can perform, the behavioral composition can be modeled mathematically as a composition of the partial orders.

- Structural Composition yields the composed Abstract Data Model, $A =< A_C, A_{SU1}, A_{SU2}, g_1, g_2 >$ where *g1*, *g2* are the partial maps between concepts in A_C, A_{SU1}, and A_{SU2} .
- Behavioral composition is completed by the R_C set of rules that together with R_{SU1} and R_{SU2} form the R rule set for the composed semantics.

The Compositional Structure for SEFSM Components



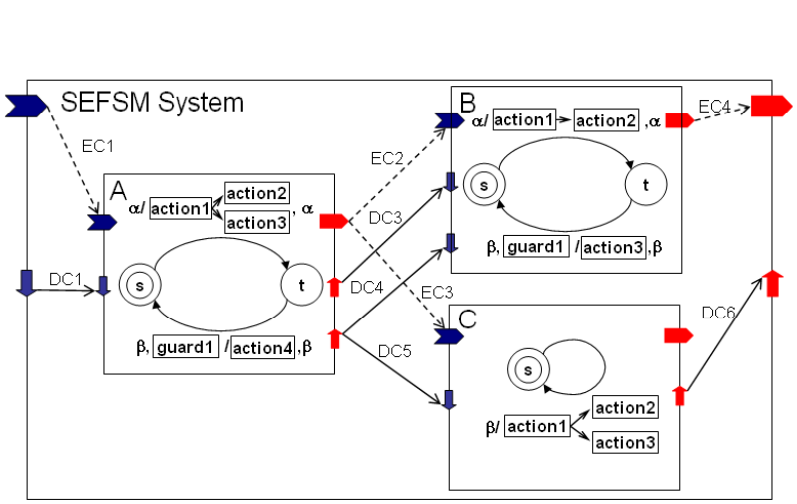
A SEFSM Component Model



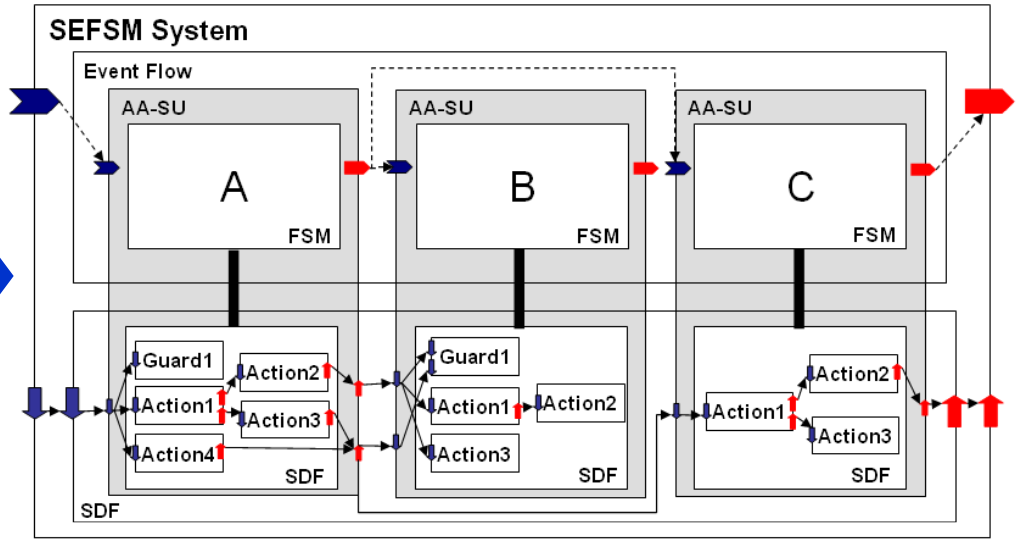
Internal Structure

- The behavior of SEFSM components can be divided into two different behavioral aspects: the FSM-based behavior expressing reactions to events and the SDF-based behavior controlling the execution of computational functions (actions and guards).

The Compositional Structure for SEFSM Systems



A SEFSM Conceptual Structure



A Compositional Structure

- A SEFSM system is composed of a set of components, which communicate with each other through event channels and data channels.
- The semantics of SEFSM systems is defined as the composition of FSM-SU and SDF-SU

Summary

- Rapid changes in implementation platforms and heterogeneity of embedded systems applications demand domain specific approaches.
- Recent advancements in model-based design provide reusable infrastructure for building domain specific tool chains or domain specific extensions to established tool frameworks.
- The key requirement is to make structural and behavioral semantics explicit.
- State-based and event-based semantics play important role in this.