

AADL:
UML profile or DSL ?

First workshop "UML and AADL"
Paris, 9 October 2006



DSL

Domain Specific Languages

- As opposed to General Purpose Languages
- "Intentional Programming"
- A DSL requires:
 - an abstract data representation
 - editing tools
 - generation tools
- Not something new:
 - many examples in the Unix world
 - recently promoted by well known SW editors

from DSL to DSM Domain Specific Models

- DSL usually deals with programming activities
- Need to take into account Model Driven Engineering
- DSL + MDE = DSM: Domain Specific Models
(sometimes called *DSML: Domain Specific Modelling Language*)

"Intentional Modeling"

- An abstract modeling data definition (syntax + semantics)
- Editors and concrete data representations (text + graphics)
- Model processing (transformation and production rules)

Illustration:

- the Architecture Analysis and Design Language
- ref. SAE standard AS-5506 (Nov. 2004)

Abstract definition

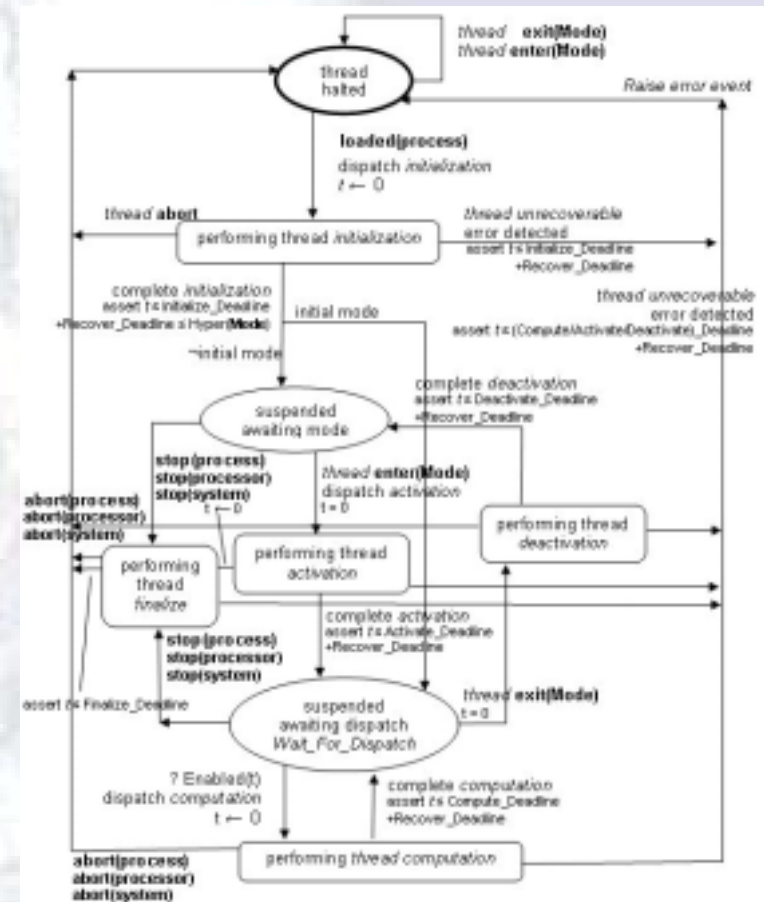
Definition of the syntax:

- BNF
- XML DTD or Schema
- UML metamodel

Definition of the semantics:

- *Static constraints: Legality rules*
- *Temporal semantics: State-Transition models*

Category	Type	Implementation
thread	Features: <ul style="list-style-type: none"> • server subprogram • port • port group • provides data access • requires data access • flow specifications: yes • properties: yes 	Subcomponents: <ul style="list-style-type: none"> • data • Subprogram calls: yes • Connections: yes • Flows: yes • Modes: yes • Properties: yes



Specific issue for Real-Time modeling languages:

- *Actual temporal semantics is platform dependent !!!*

Concrete representations

AADL (core of the standard):

```

process implementation ProdCons.default
  subcomponents
    theProd: thread Prod.Impl;
    theCons: thread Cons.Impl;
  connections
    EventConnection1: event port start -> theProd.start;
    DataConnection1: data port theProd.val -> theCons.val;
end ProdCons.default;
  
```

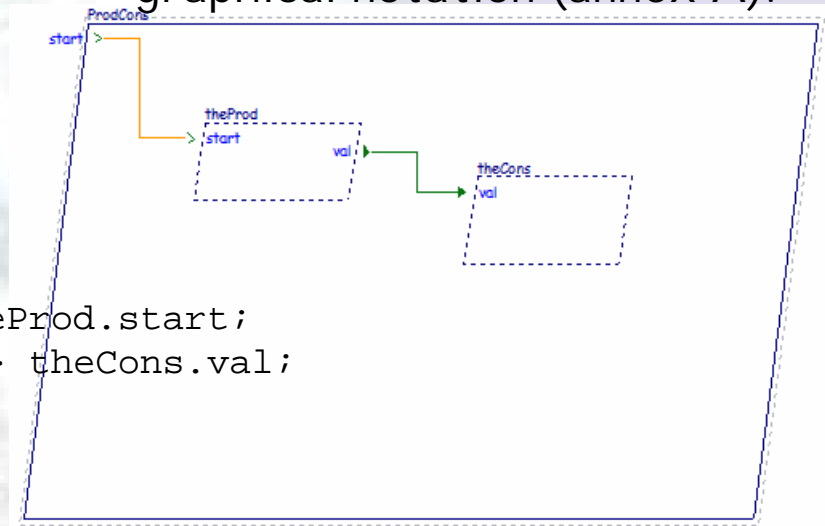
AAXL (annex C):

```

<processImpl name="ProdCons.default" compType="//processType[@name=ProdCons]">
  <connections>
    <eventConnection name="EventConnection1" ... />
    <dataConnection name="DataConnection1" ... />
  </connections>
  <subcomponents>
    <threadSubcomponent name="theProd" classifier="//threadImpl[@name=Prod.Impl]" />
    <threadSubcomponent name="theCons" classifier="//threadImpl[@name=Cons.Impl]" />
  </subcomponents>
</processImpl>
  
```

UML profile (annex B):

graphical notation (annex A):



Note: *The modeling process is not formalized !*

AADL: a DSM ?

- The core standard and its annexes defines:
 - the abstract data model
 - the concrete textual and graphical representations
 - code generation rules
- Domain specific features:
 - "software and hardware architecture of performance-critical real-time systems"
 - AADL components: appropriate building blocks for architecture
 - AADL categories: enable "intentional modeling"
 - AADL annexes and property sets: for yet more domain specific models
- Going further:
 - taking into accounts the actual RT semantics of the platform (***RT patterns***)
 - taking into accounts the development life-cycle (***modeling process***)
 - impact of the toolset (***tool architecture***)

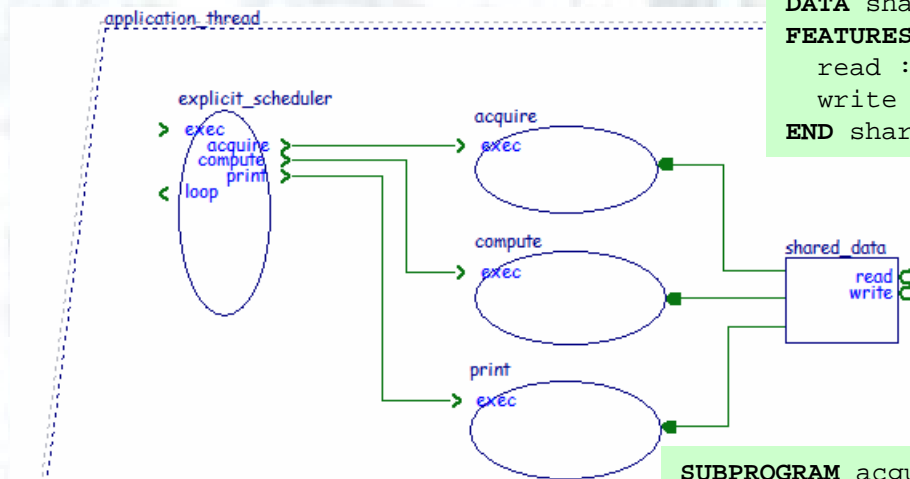
the Real-Time domain

- Examples of Real-Time model patterns in AADL
- Trade off: flexibility vs. predictability
- Depends on platform specific scheduling executive:
 - Case study:
 1. acquire raw data
 2. process raw data
 3. display processed data
 - case 1: cyclic scheduling
 - case 2: preemptive scheduling with clock synchronous dataflows
 - case 3: preemptive scheduling with client synchronous dataflows
 - case 4: preemptive scheduling with asynchronous dataflows
 - case 5: preemptive scheduling with client-server communication

cyclic scheduling

```

THREAD IMPLEMENTATION application_thread.others
SUBCOMPONENTS
  shared_data : DATA shared_data;
PROPERTIES
  Compute_Entrypoint => "explicit_scheduler";
END application_thread.others;
  
```



```

DATA shared_data
FEATURES
  read : SUBPROGRAM read;
  write : SUBPROGRAM write;
END shared_data;
  
```

```

SUBPROGRAM IMPLEMENTATION explicit_scheduler.others
CALLS {
  acquire : SUBPROGRAM acquire;
  compute : SUBPROGRAM compute;
  print : SUBPROGRAM print;
};
END explicit_scheduler.others;
  
```

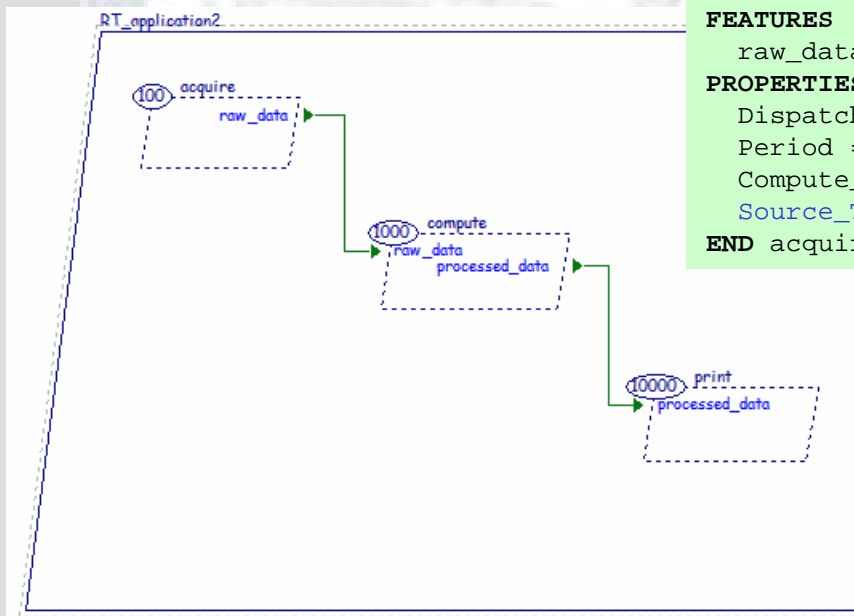
```

SUBPROGRAM acquire
FEATURES
  shared_data : REQUIRES DATA ACCESS shared_data;
END acquire;

SUBPROGRAM IMPLEMENTATION acquire.others
CALLS {
  write : SUBPROGRAM shared_data.write;
};
PROPERTIES
  Compute_Execution_Time => 5 ms .. 10 ms;
  Source_Text => "acquire.c"
END acquire.others;
  
```

preemptive scheduling

with clock synchronous dataflow communications



```

THREAD acquire
FEATURES
  raw_data : OUT DATA PORT T_Flow;
PROPERTIES
  Dispatch_Protocol => Periodic;
  Period => 100 ms;
  Compute_Execution_Time => 5 ms .. 10 ms;
  Source_Text => "acquire.c"
END acquire;
  
```

```

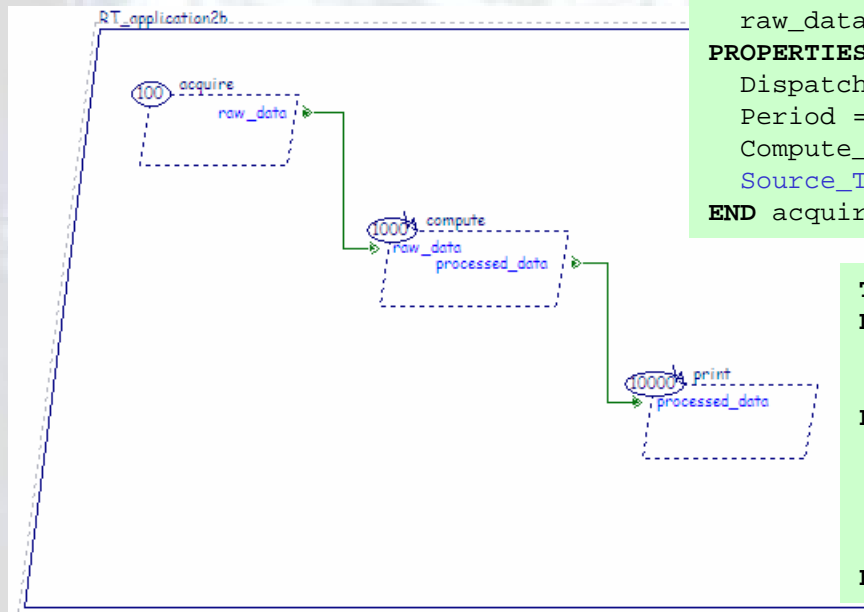
THREAD compute
FEATURES
  raw_data : IN DATA PORT T_Flow;
  processed_data : OUT DATA PORT T_Flow;
PROPERTIES
  Dispatch_Protocol => Periodic;
  Period => 1000 ms;
  Compute_Execution_Time => 200 ms .. 500 ms;
  Source_Text => "compute.c"
END compute;
  
```

```

PROCESS IMPLEMENTATION RT_application2.others
SUBCOMPONENTS
  acquire : THREAD acquire;
  compute : THREAD compute;
  print : THREAD print;
CONNECTIONS
  DATA PORT acquire.raw_data -> compute.raw_data;
  DATA PORT compute.processed_data -> print.processed_data;
END RT_application2.others;
  
```

preemptive scheduling

with client synchronous dataflow communications



THREAD acquire

FEATURES

raw_data : **OUT DATA PORT** T_Flow;

PROPERTIES

Dispatch_Protocol => Periodic;

Period => 100 ms;

Compute_Execution_Time => 5 ms .. 10 ms;

Source_Text => "acquire.c"

END acquire;

THREAD compute

FEATURES

raw_data : **IN EVENT DATA PORT** T_Flow;

processed_data : **OUT EVENT DATA PORT** T_Flow;

PROPERTIES

Dispatch_Protocol => Sporadic;

Period => 1000 ms;

Compute_Execution_Time => 200 ms .. 500 ms;

Source_Text => "compute.c"

END compute;

PROCESS IMPLEMENTATION RT_application2.others

SUBCOMPONENTS

acquire : **THREAD** acquire;

compute : **THREAD** compute;

print : **THREAD** print;

CONNECTIONS

EVENT DATA PORT acquire.raw_data -> compute.raw_data;

EVENT DATA PORT compute.processed_data -> print.processed_data;

END RT_application2.others;

preemptive scheduling with asynchronous dataflow communications

```

THREAD acquire
FEATURES
  shared_data : REQUIRES DATA ACCESS shared_data;
END acquire;

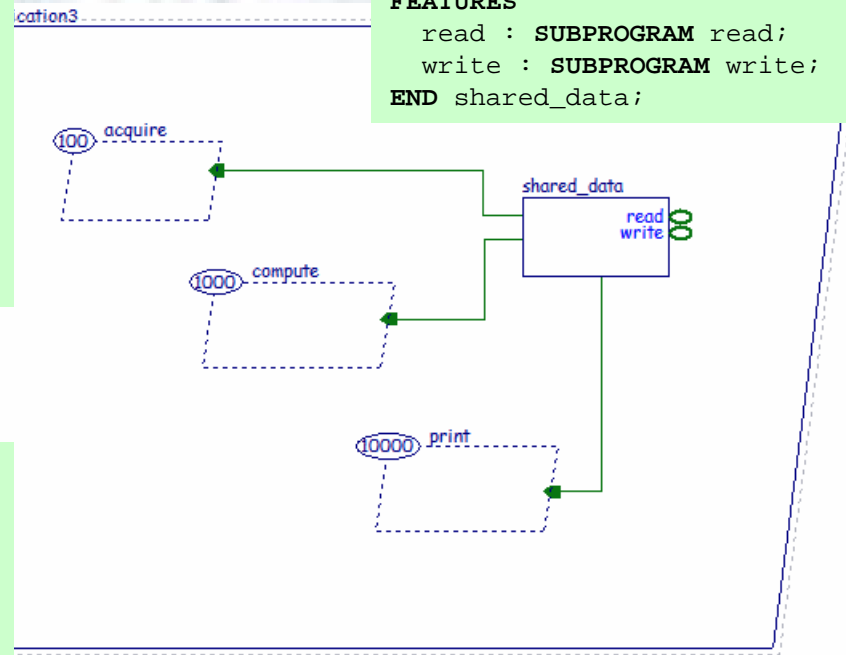
THREAD IMPLEMENTATION acquire.others
CALLS {
  write : SUBPROGRAM shared_data.write;
};
PROPERTIES
  Dispatch_Protocol => Periodic;
  Period => 100 ms;
  Compute_Execution_Time => 5 ms .. 10 ms;
  Source_Text => "acquire.c"
END acquire.others;
  
```

```

PROCESS IMPLEMENTATION RT_application3.others
SUBCOMPONENTS
  acquire : THREAD acquire.others;
  compute : THREAD compute.others;
  print : THREAD print.others;
  shared_data : DATA shared_data;
CONNECTIONS
  DATA ACCESS shared_data -> acquire.shared_data;
  DATA ACCESS shared_data -> compute.shared_data;
  DATA ACCESS shared_data -> print.shared_data;
END RT_application3.others;
  
```

```

DATA shared_data
FEATURES
  read : SUBPROGRAM read;
  write : SUBPROGRAM write;
END shared_data;
  
```



preemptive scheduling with client-server communications

```

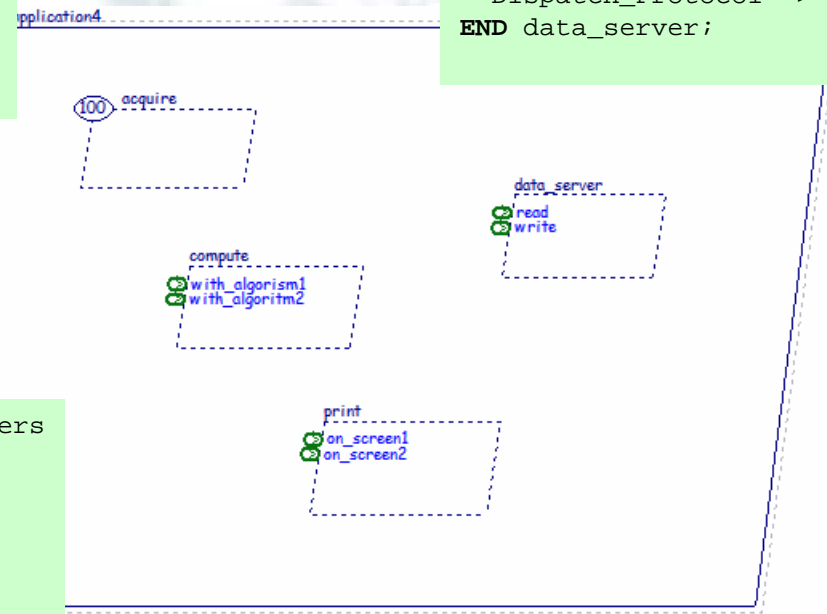
THREAD IMPLEMENTATION acquire.others
CALLS {
  compute : SUBPROGRAM with_algorithm1;
};
PROPERTIES
  Dispatch_Protocol => Periodic;
  Period => 100 ms;
  Compute_Execution_Time => 5 ms .. 10 ms;
  Source_Text => "acquire.c"
END acquire.others;
  
```

```

THREAD data_server
FEATURES
  read : SERVER SUBPROGRAM read;
  write : SERVER SUBPROGRAM write;
PROPERTIES
  Dispatch_Protocol => Aperiodic;
END data_server;
  
```

```

PROCESS IMPLEMENTATION RT_application4.others
SUBCOMPONENTS
  acquire : THREAD acquire;
  compute : THREAD compute;
  print : THREAD print;
  data_server : THREAD data_server;
END RT_application4.others;
  
```



Summary on real-time patterns

- Case 1 (subprograms)

- Fully predictable but:
- Lack of flexibility at design time (very restrictive)
- Lack of flexibility at run time

fully supported by the
core of the AADL

- Case 2 (periodic threads)

- Fully predictable and:
- More flexible at design time (no explicit scheduler to build) but:
- Lack of flexibility at run time

actions in progress

- Case 3 (sporadic threads)

- More flexible at design time (no explicit scheduler to build) and:
- More flexible at run time but:
- Implicit execution model of AADL probably needs to be enriched

- Case 4 (shared data)

- & Case 5 (client-server)

targets of the AADL
behavior annex

- More flexible at design time (no explicit scheduler to build) and:
- More flexible at run time (no fixed communication rates) but:
- More details must be given to insure predictability through RT analysis

Modeling Process

The modeling language must be adjusted for each activity of the life-cycle. example:

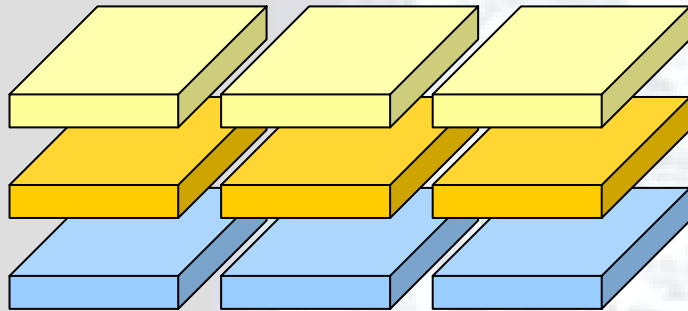
- system engineering: hardware components + systems
- software engineering: software components

Support the right paradigms at each level

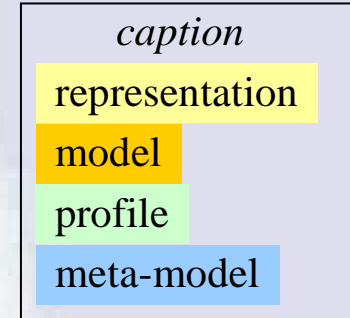
- the tool implementor's concern:
 - instantiate the meta-model*
- the user's concern:
 - build a model instance*
 - create and connect components*
 - use expressive notations*

Strong impact on tool architecture !

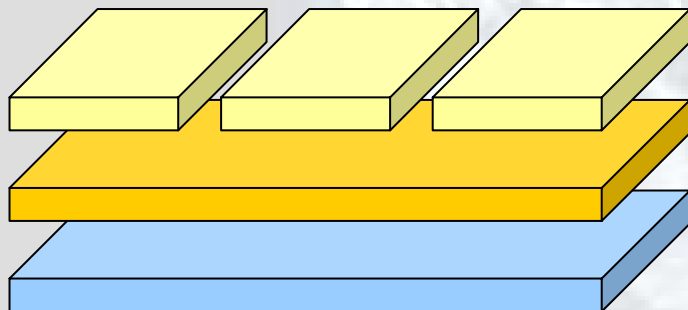
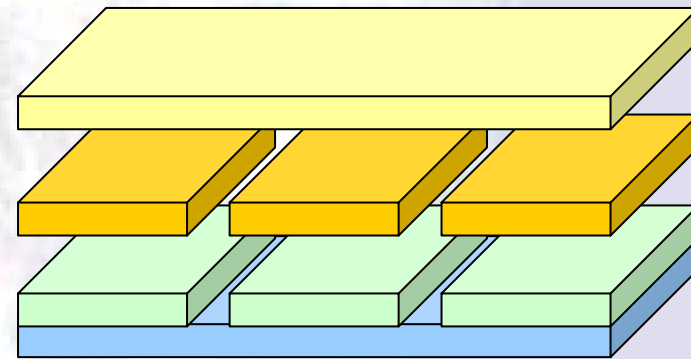
Tool architectures for component based modeling



Vertical architecture
ex: Eclipse based tools
(1 plugin per language)



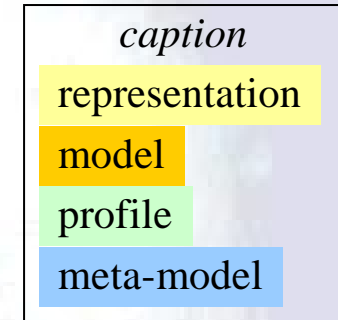
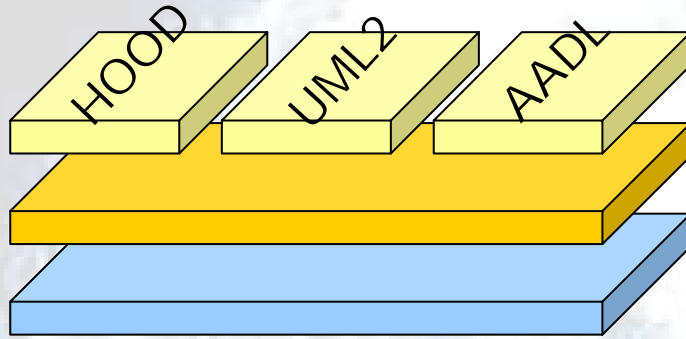
Mixed architecture
ex: UML tools
(common core meta-model and representation)



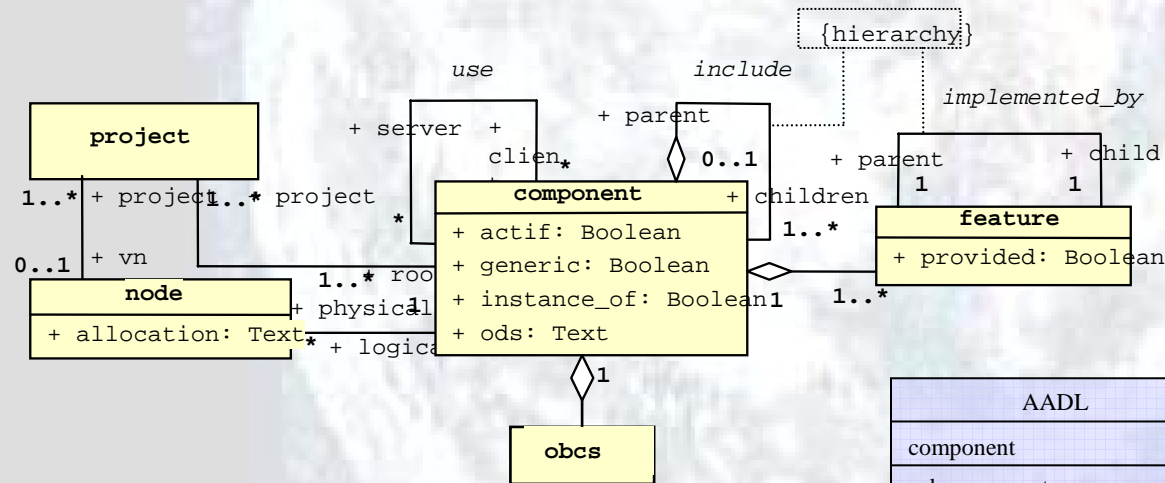
Horizontal architecture
ex: DSM tools
(common model and several projections)

→ *example:*
Stood

unifying models in Stood



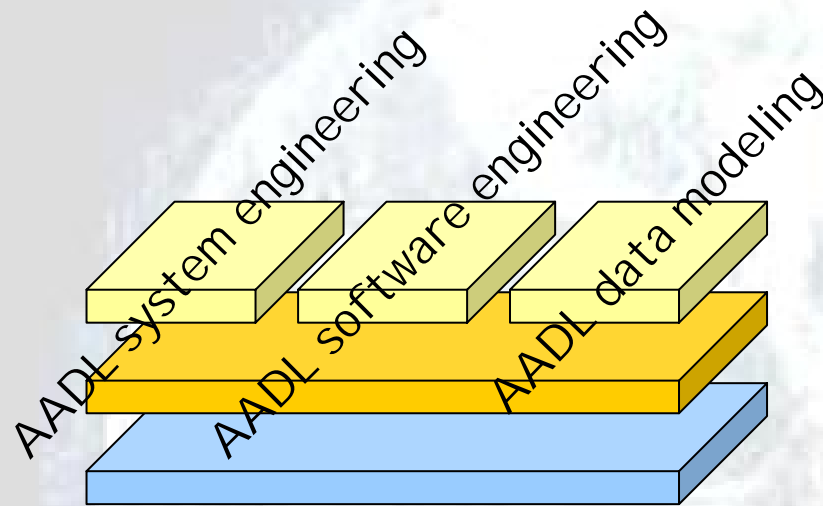
Common "component-based" meta-model



+ mapping rules

AADL	UML 2.0	HOOD
component	component	(parent) module
subcomponent	part	(child) module
features	provided interface	provided interface
	required interface	required interface
containment connection	delegate (provided)	implemented_by
	delegate (required)	use (uncle)
components connection	assembly	use (sibling)

supporting process activities in Stood



<i>caption</i>
representation
model
profile
meta-model

Common "component-based" meta-model

AADL System Engineering:

- system instance
- hierarchy of subcomponents (systems and hardware comp.)

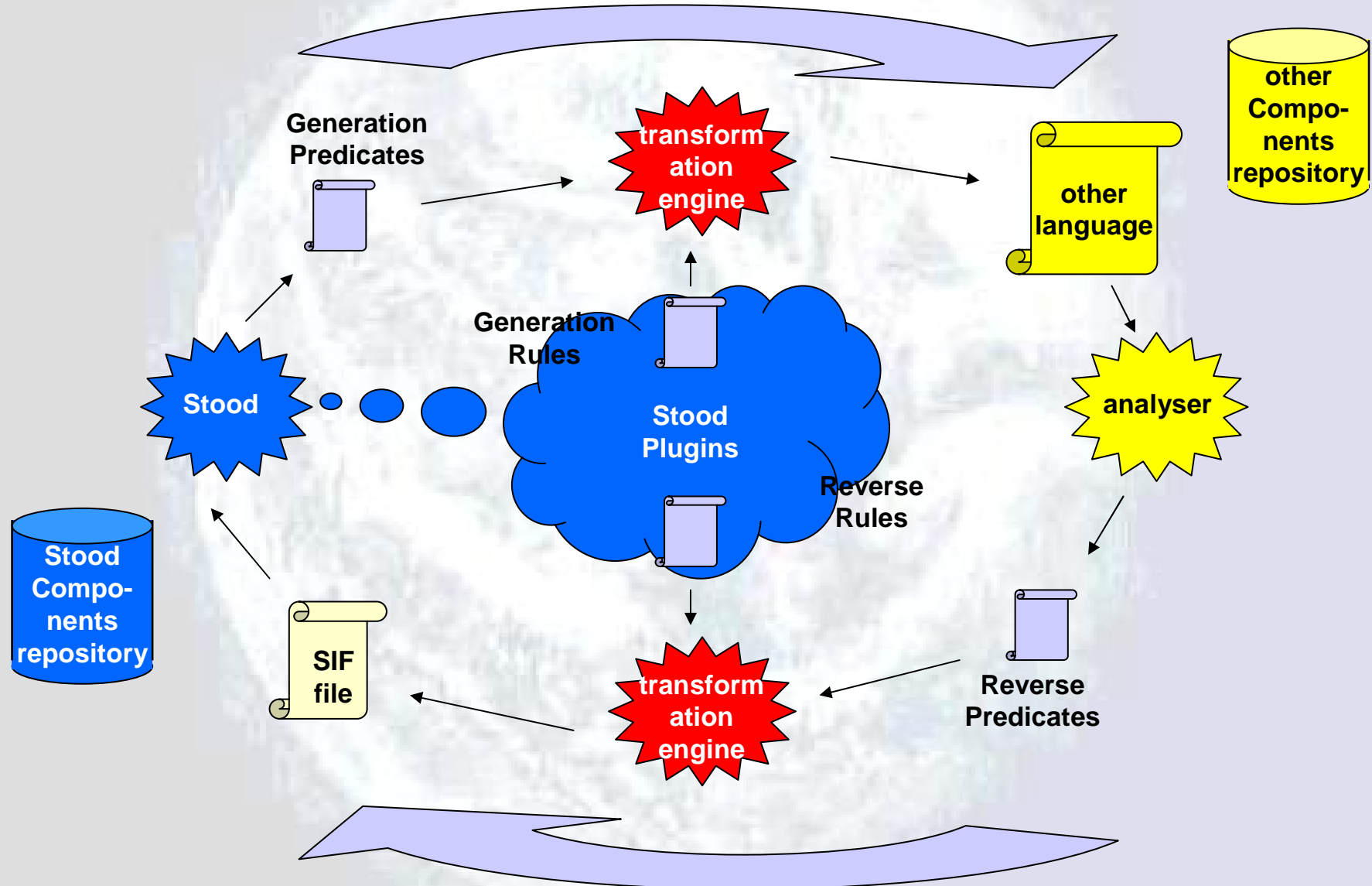
AADL Software Engineering:

- process instance
- hierarchy of subcomponents (threads; data; subprograms)

AADL Data Modeling:

- package
- set of data components

Logical Model Processing in Stood



Advantages of LMP technology:

- pure declarative language (prolog):
 - ⇒ high level of traceability between the formal rules and their implementation
- strong model isolation:
 - ⇒ no way to alter the model while processing it
- homogeneous syntax:
 - ⇒ can replace ATL + OCL
- modular structure:
 - ⇒ one set of rules per user function
 - . target source code generation
 - . source code reverse engineering
 - . design rules verification
 - . models transformations
 - . document generation

Conclusion

- The AADL is at the same time:
 - a Domain Specific Modeling Language (DSM),
 - a XML meta model
 - a UML profile
- Each approach offers specific benefits:
 - DSM best matches user's concerns for modelling activities
 - meta-model is the strongest framework for verification tools
 - UML profile can provide a bridge with the OMG world
- The choice has an impact on tool architecture:
 - custom tools for the DSM (Stood, Cheddar, ...)
 - Eclipse platforms for the meta-model (Osate, Topcased)
 - UML tools for the profile
- However, they all support the same AADL semantics so that models interchange is not an issue.

enough time for a quick demo ?