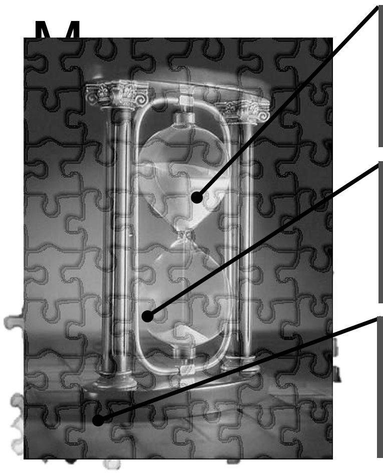
Timed Languages for Embedded Software

Ethan Jackson
Advisor: Dr. Janos Szitpanovits
Institute for Software Integrated Systems
Vanderbilt University

Embedded System Design



Design methodologies and patterns with tools that support pattern reuse: Numerous non-trivial case studies

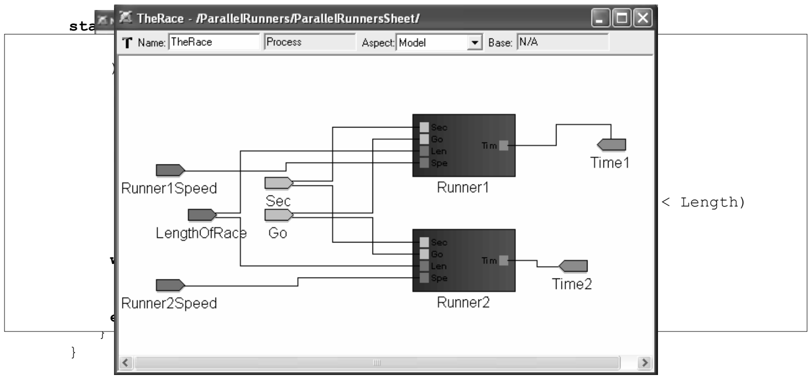
Combine formal abstractions with correct-by-construction programming languages: Provably correct systems, but tool flow may be overly restrictive

Combine formal abstractions with model checking and verification: Provably correct systems, when proof can be produced.

Want to rethink concepts so that well-tested design methodologies help designers better understand correct-by-construction systems and/or build more analyzable systems.

The “Isomorphic” Approach

View MoCs as DSMLs, and describe these languages using MIC. The isomorphic approach makes a one-to-one mapping between a DSML and an already existing language that supports a MoC.



This importation of an existing syntax makes DSML construction and code generation easy, but it does not exercise the full capabilities of MIC.

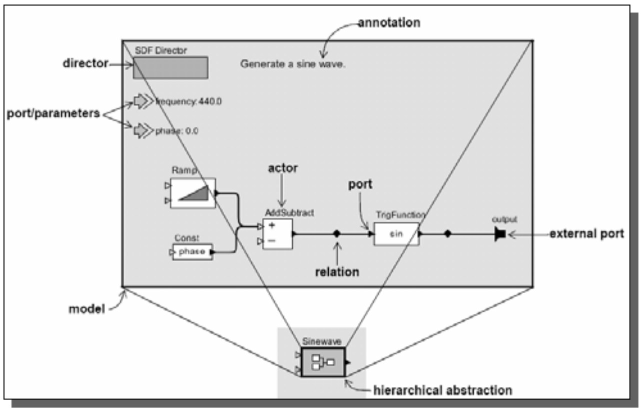
How Could MIC Help?

It is clear that the previous approach does not yield great results, so what should a better approach yield?

1. MIC deals with DSMLs and DSML design patterns. One would like to find some common structure among the languages that express various MoCs, and cast these similarities as patterns in MIC.
2. MIC provides design time constraint checking with OCL. One would like to describe MoCs so that certain rules could check behavioral properties (or conservative approximations of such properties) during design time.
3. MIC provides *aspect* oriented modeling, which partitions the concepts in a DSML. One would like to use aspects to partition MoC concepts, and to modularize available design time reasoning.
4. MIC requires the designer to “anchor” a DSML to a *semantic domain*, which attaches meaning to the *abstract syntax*. One would like to reuse parts of anchorings across similar models of computation.

In Search of Patterns

• Ptolemy recognizes that scheduling under a certain MoC is largely independent of the underlying dataflow. Ptolemy captures this notion with *directors*, which can be swapped out to change to MoC.



• However, it can become tricky to find MoC dependent errors when the scheduler is a "black box".

In Search of Patterns

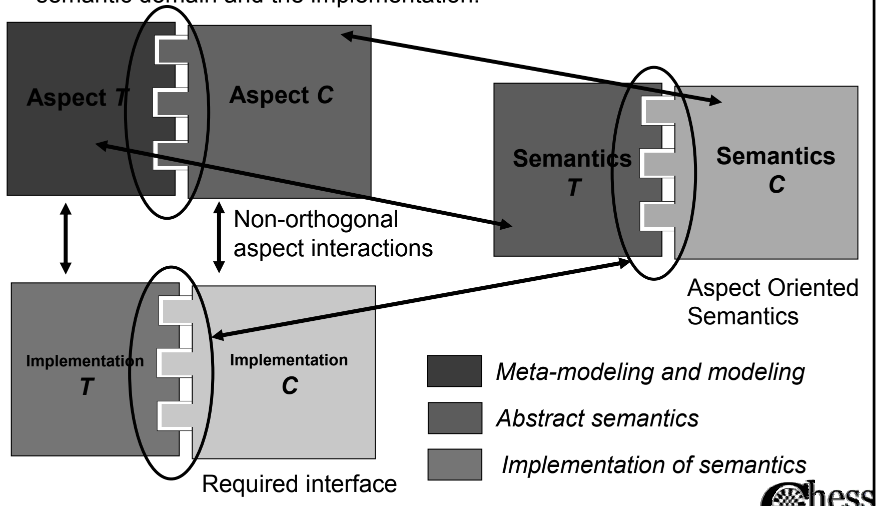
• Giotto treats timing and computation as orthogonal aspects.

```
start Degraded{

   mode Degraded(navigation_output) period 8000 {
            exitfreq 1 do Normal(switch_driver);
            taskfreq 1 do Control(control_driver);
            actfreq 1 do display_actuator(display_driver);
            taskfreq 2 do Navigation(navigation_driver);

   }

   mode Normal(navigation_output) period 8000 {
            exitfreq 2 do Degraded(switch_driver);
            actfreq 1 do display_actuator(display_driver);
            taskfreq 4 do Navigation(navigation_driver);
            taskfreq 1 do Control(control_driver);

   }
}
```

• In Giotto, scheduling is not "black box" because the timing specification is scheduling specific information. However, it is not clear how to extend this model to handle other MoCs.

Interacting Partitions of Time and Data

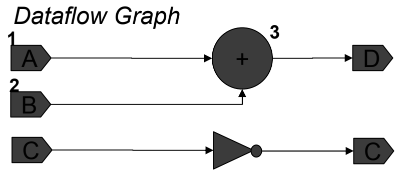
• Using MIC, we can create a DSML for "time models" and a DSML for computation. This is a design pattern that will be reused in the semantic domain and the implementation.

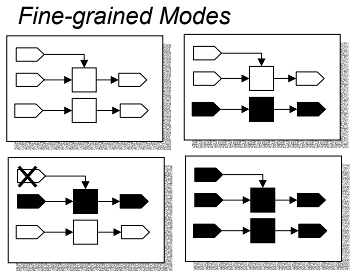


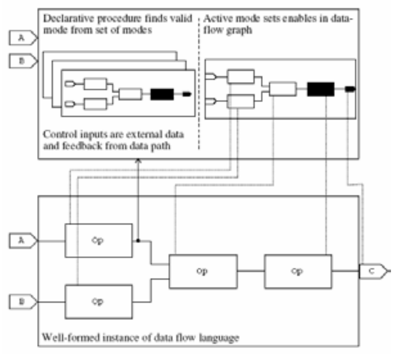
Aspect T
Aspect C
Semantics T
Semantics C
Non-orthogonal aspect interactions
Implementation T
Implementation C
Aspect Oriented Semantics
Required interface
Meta-modeling and modeling
Abstract semantics
Implementation of semantics

Hierarchical Fine-Grained Modal Models

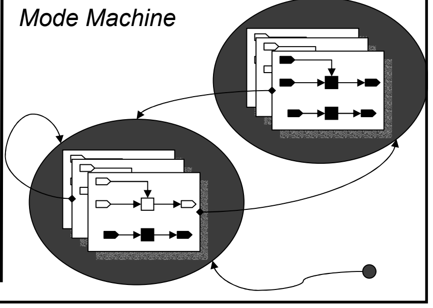
Synchronous Reactive Example





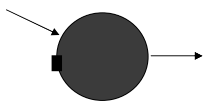


Declarative procedure finds valid mode from set of modes
Active mode sets enables in data-flow graph
Control inputs are external data and feedback from data path
A
B
op
C
Well-formed instance of data flow language



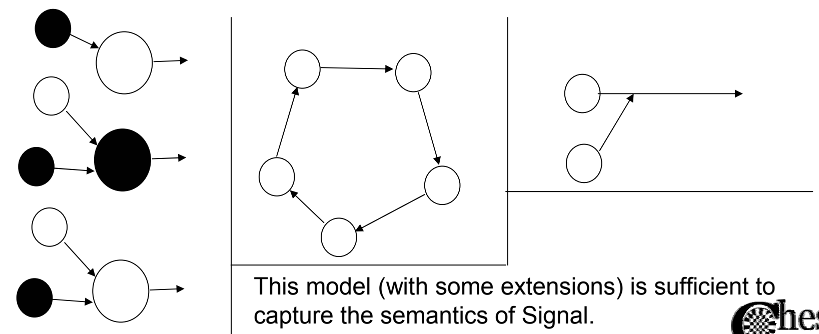
Well-formedness Rules, Accessible Properties

The Dataflow should have no dangling inputs (bounded memory)



The modes should have none of these subgraphs

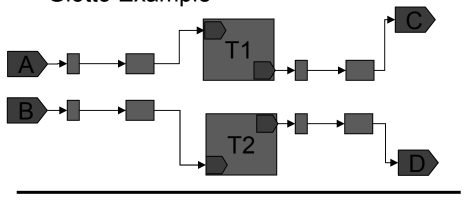
Synchrony Constraints *Causality* *Deterministic Merge*

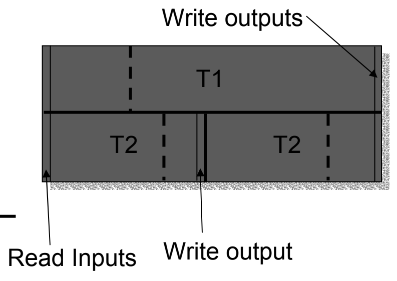


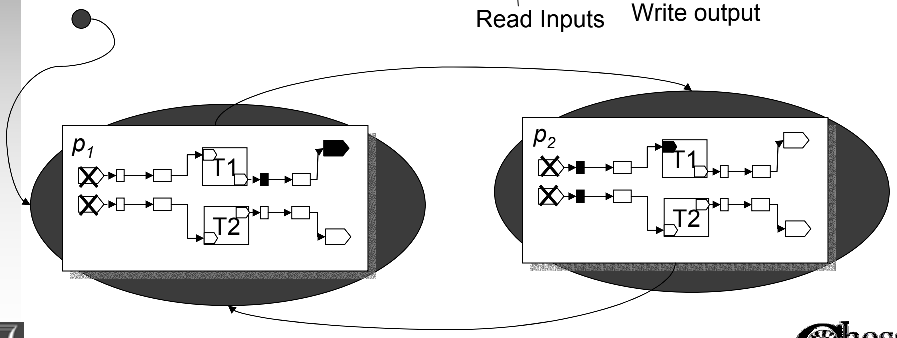
This model (with some extensions) is sufficient to capture the semantics of Signal.

Hierarchical Fine-Grained Modal Models

Giotto Example

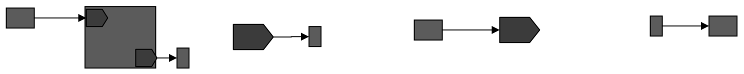






Well-formedness Rules

Dataflow Well-formedness rules ensure communication model

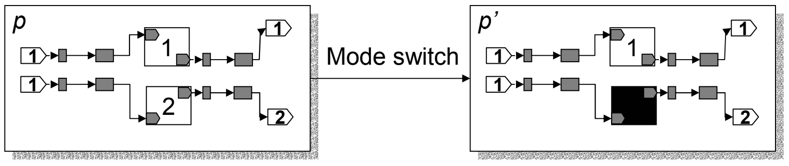


Tasks read from global memories, and senors and tasks write to "private" memories

Actuators read from global memories

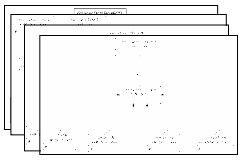
Global memories read from private memories

The fine-grained modal structure is regular, so it does not need to modeled directly.

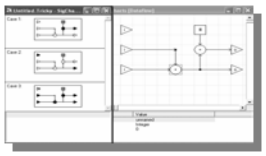


Well-formedness rule: Modes switches should be *well-timed*

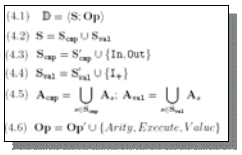
Conclusions



Metamodel patterns define the first class citizens of data and time DSMLs. These patterns are used to construct MoC specific data and time DSMLs.



Aspect support maintains consistency between data and time aspects. Design time well-formedness rules can verify some behavioral property.



(4.1) $\mathbf{D} = \langle \mathbf{S}; \mathbf{Op} \rangle$

(4.2) $\mathbf{S} = \mathbf{S}_{\mathtt{cmp}} \cup \mathbf{S}_{\mathtt{val}}$

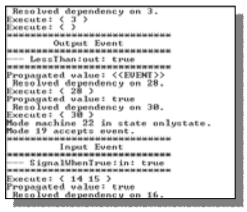
(4.3) $\mathbf{S}_{\mathtt{cmp}} = \mathbf{S}'_{\mathtt{cmp}} \cup \{\mathtt{In}, \mathtt{Out}\}$

(4.4) $\mathbf{S}_{\mathtt{val}} = \mathbf{S}'_{\mathtt{val}} \cup \{\mathtt{I}_{+}\}$

(4.5) $\mathbf{A}_{\mathtt{cmp}} = \bigcup_{s \in \mathbf{S}_{\mathtt{cmp}}} \mathbf{A}_s; \ \mathbf{A}_{\mathtt{val}} = \bigcup_{s \in \mathbf{S}_{\mathtt{val}}} \mathbf{A}_s$

(4.6) $\mathbf{Op} = \mathbf{Op}' \cup \{Arity, Execute, Value\}$

Aspect oriented semantics give an SOS to MoCs by composing a "data" SOS and a "time" SOS.

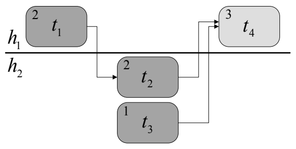


```
 Resolved dependency on 3.
Execute: < 3 >
Execute: < >
******************************
        Output Event
******************************
=== LessThan:out: true
******************************
Propagated value: <<EVENT>>
 Resolved dependency on 28.
Execute: < 28 >
Propagated value: true
 Resolved dependency on 30.
Execute: < 30 >
Mode machine 22 in state onlystate.
Mode 19 accepts event.
******************************
         Input Event
******************************
=== SignalWhenTrue:in: true
******************************
Execute: < 14 15 >
Propagated value: true
 Resolved dependency on 16.
```

The composition is defined over a minimal set of the semantics, so implementations can realize the composition once. This has been realized with TinyModes.

Future Directions

Distributing Giotto programs across many hosts make the instantaneous communication assumption unrealistic. Current research by the Giotto team shows that distributed Giotto programs can be extended with worst case communication times.



Promising results show that when a developer changes the timing properties of a Giotto component, the entire system can be verified using:
• A linear time algorithm in the size of the component
• Timing information local to the host

Perhaps some well-formedness rules can be found that would allow design-time checking of ***distributed*** Giotto programs?

The End

Questions?