DARPA PI Meeting, New Orleans 01/20/01.

Formal Composition Technology for Time-Triggered Systems

N. Shankar

(filling in for John Rushby)

shankar@csl.sri.com

URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory

SRI International

Menlo Park, CA

Overview

- Philosophy
- Background
- Progress
- Outlook

Philosophy

Logic (predicate calculus) is a general-purpose instrument.

Flexible but unhelpful for building models.

Frameworks constrain what can be expressed in terms of what is built-in and what is definable.

But there is a semantic problem when combining or mapping between different frameworks.

Background

Goal

Enhance the design and extension of embedded systems by

Reliably bridging the semantic gap

Between model-based embedded system description frameworks (such as Matlab)

And implementation architectures (such as the Time-Triggered Architecture, TTA)

Through the use of compositional formal techniques.

Model-Based Description Frameworks

Problem-oriented notations that abstract from implementation details.

- Flowgraphs with discrete, continuous transfer functions, and control logic.

Support simulation and code generation.

Examples: Matlab, Matrix_x, Beacon.

Time-Triggered Architecture

- Implementation framework for hard real-time systems
- Time-triggered rather than event-triggered
- TDMA communication.
- Clock synchronization.
- Group membership.
- Transparent fault tolerance.
- Error containment.

Formal Support for Description Frameworks

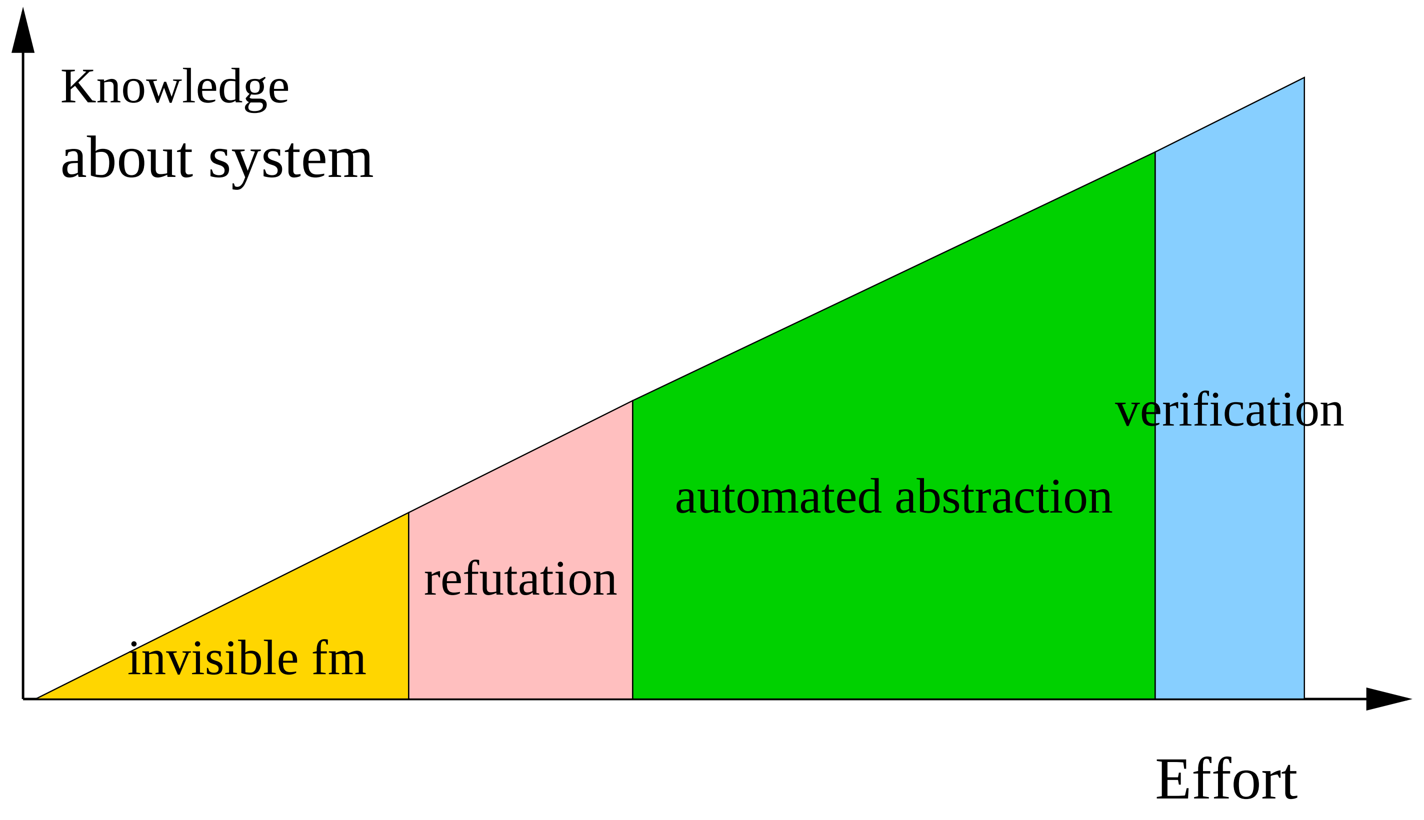
What formal support can be provided for model-based description frameworks?

Can check models for type correctness, absence of bound violations, semantic well-formedness.

Can provide a formal transition system model of flowgraphs and hierarchical state machines with or without continuous dynamics.

Verification of safety, liveness, timeliness, stability.

Invisible Formal Methods



Formal Support for Implementation

Formal transition system model of time-triggered behavior.

Show that mappings high-level models to low-level implementation frameworks preserve high-level behavior.

- Are the interfaces mapped correctly?
- Does the schedule preserve safety, timeliness?
- Are hardware/communication failures masked?
- Are the tasks partitioned — absence of unintended interference?

Challenge: Modular verification of correctness of implementation.

SAL as a Unifying Model

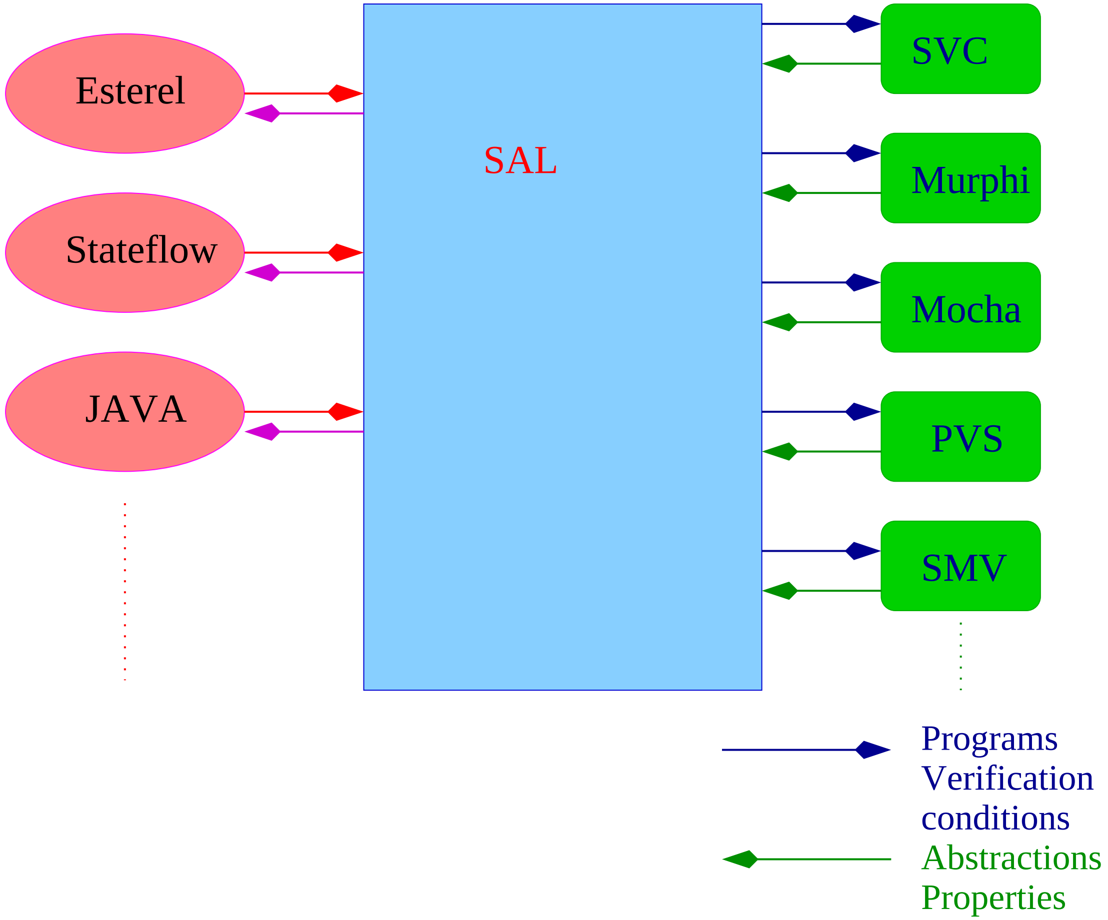
SAL is a transition system model that supports

1. Transitions: Definitions and guarded commands.
2. Modules: input, output, local, global variables.
3. Well-formed synchronous and asynchronous composition of modules.

SAL is a common front-end to theorem provers, model checkers, and program analyzers.

Data and predicate abstraction are key analysis techniques.

SAL Architecture



Progress

Stateflow/Statecharts to SAL

Stateflow is translated into SAL by

- Translating a Stateflow diagrams into *extended hierarchical automata* (HA) [Mikk, Lakhnech, Siegel]
- Translating extended HA into SAL.

The translation is modular and can be adjusted for different Statechart semantics.

Defined and implemented.

Handles hierarchy, nested event broadcasts, but not (yet) junctions.

Stateflow/Statecharts to SAL

Hierarchical automata: An intermediate representation with an automaton for each state machine plus hierarchy embeddings.

Extended HA allow transitions to/from substates.

Modular translation: One SAL module per state machine.

Stateflow/Statecharts to SAL

For each state, a three-valued variable (*inactive*, *active* , *active-and-processing*) indicates if state is activated or suspended.

For each automata, boolean *preempted-in* holds if a module is preempted, and *preempted-out* holds when module preempts.

Similar to **sf2smv** but not restricted to finite state systems.

Analyzing Stateflow/Statecharts

Model checking the SAL output from Stateflow can be used to verify

- Convergence
- Reachability (using PDA)
- Nested event loops.

Theorem proving can be used to establish type-correctness and verify data invariants.

Verification can abstract hierarchy. Need not flatten it.

Outlook

Stateflow/Statecharts to SAL

Deal with the complete langauge; try other Statechart variants.

Apply SAL/SMV model checking to challenge examples.

Apply abstraction techniques for transforming infinite or large state spaces to small, finite ones.

Use the translator for verifying Stateflow properties.

Extend the translation to Simulink.

Challenge: Modular verification using environment models (interface assumptions).

TTA to SAL

TTA implementation of a high-level model involves

- Mapping tasks/subsystems to hosts
- Adding rate, schedule, and communication information
- Introducing fault tolerance

The TTA model can also be viewed as a state machine (see Rushby's proof of group membership).

Verifying Correctness

Given a design M and its TTA implementation T, both modelled in SAL,

The goal is to show that T implements M given fault assumptions F.

Modular implementation: If M is composed of $M_1 \| \ldots \| M_n$, then show that the corresponding T_i implements M_i given interface assumptions I_i.

Relies heavily on invariants and abstractions, and arithmetic theorem proving.

Can exploit predictability provided by TTA.

Summary

Embedded system design: High-level models mapped to low-level implementations.

For safety-critical applications, the correctness of the mapping is crucial.

Verification of the correspondence between design and implementation requires accurate semantic models and modular verification techniques.