Logical Programming Environments

Jason Hickey, Richard Murray

John Hauser
What Formal Methods Offer

• Confidence:
  – reliability: we know what a system is supposed to do, and it does it
  – tools: specifications, code, and verification

• Automation:
  – code analysis, synthesis, and optimization
  – interactive design assistance

• High-confidence design requires systematic and structured approaches at all time scales (run time, design time)
Logical Programming Environments

• A LPE provides an collaborative, interactive design environment

• An LPE includes:
  – A *logical library* where programs, proofs, and reasoning tools can be stored and shared in a collaborative development
  – A *formal compiler* that provides an open platform for producing executable code from programs, specifications, and proofs
  – An *automated reasoning system* that is used to develop formal proofs that programs meet their specification
An example

- Ensemble provides *group communication*
  - Like a multi-point version of TCP
  - Communication is reliable
- Used in NY, Swiss stock exchanges
- French air-traffic control
- Navy’s AEGIS command, control
Formal tools

- Nuprl Logical Programming Environment
- All properties (and meta-proofs of algebra) are formal
Formal automation

- Protocols are pluggable components
- Protocol layers are in ML
- ~70 components, 1000s protocols
- About 30 layers in a protocol; roughly 300 lines of ML each
- Use refinement to verify/synthesize ML code
Applying the LPE to distributed control

- Develop
  - A library of verified control components
  - A hierarchy of languages for cooperative control problems
  - A set of tools and heuristics for automated analysis and synthesis

- Design by successive refinement
  - Requirements propagate down
  - Assumption violations propagate upward (at design time and at run time)
  - Interference prevents straightforward composition
Multi-vehicle wireless testbed

- 8-10 vehicles, integrated computing and communications, including wireless Ethernet (802.11), and Bluetooth

- 2-4 fixed communication nodes, capable of broadcasting on multiple channels

- A set of overhead cameras that can be used to provide position information to the vehicles (perhaps simulating GPS)

- A command console with computing and communication nodes
Multi-vehicle wireless testbed
Current status

- Understand (to some extent)
  - high-level specifications
  - asynchronous communications
  - MPC

- Current focus
  - communication in rapidly-changing networks
  - design models for cooperative control
Multi-vehicle routing

- Network topology is rapidly changing
  - Consensus
  - Message routing
  - Real-time prioritized traffic
  - Make use of topology predictions
Problem formulation for UAV

- Formalize a rejoin
Top-level spec

- The *model* provides the basis for reasoning
- *Languages* provide the connection to syntax
- Top-level specification:

**Mission Objective**

**Assumptions:** \(|\text{operational}_t(V)| \geq 4\)

**Goal:** \(\forall v \in V. \exists t \leq T. \text{operational}_t(v) \Rightarrow |v.pos_t - D| < \epsilon\)
**Second-level refinement**

- Second-level: specify computation as a reactive state machine
- Verify that the decomposition satisfies the spec
Step refinement

• Each state is refined to an executable spec

Choose destination vector

Assumptions: \( \text{bandwidth} > \text{bandwidth}_{\text{min}} \)
Goal:  
Pre : Default  Eff : \( \mathbf{d}_v = \text{projected formation point} \)
Pre : Enemy detected  Eff : Abort
Pre : 2 or more vehicles failed  Eff : Abort

Move into formation

Assumptions: \( \text{bandwidth} > \text{bandwidth}_{\text{min}} \)
Goal:  
Pre : Default  Eff : Continue to reform
Pre : Within tolerance  Eff : Resume formation
Pre : Enemy detected  Eff : Abort
Logical Programming Environment

- The LPE is a framework for supporting formal design
  - *Type theory* is a common language for specification and synthesis
  - Enables *collaborative* development of verified control libraries and design automation tools
  - The *compiler* is an assistant, and the link to executable code
Design layers

Layer 1: Objective
- Goal: $G_1$
- Assumptions: $A_1$

Layer 2: Partitioning
- Goal: $G_2$
- Assumptions: $A_2$

Layer 3: Execution
- Initial plan: ①
- Trap: ②
- Re-plan: ③
- Executable spec

Layer 4: MPC
- $P_{rim1}$
- $P_{rim2}$

Layer 5: OCP
Migration path for legacy code: FC

- Import C programs into a high-confidence, formal environment
- Allow all C programs
  - pointer arithmetic
  - arbitrary coercions
- Map to a safe-functional language
- Add: transactions, migration

\[ e ::= \]
\begin{align*}
\text{let } v : t = a \text{ in } e \\
\text{let } v = s \text{ in } e \\
\text{let } v : t = \text{unop } a \text{ in } e \\
\text{let } v : t = \text{binop } a \text{ in } e \\
\text{let type } \text{typdefs in } e \\
\text{let fun } \text{fundefs in } e \\
\text{let } v : t = f(a_1, \ldots, a_n) \text{ in } e \\
\text{let closure } v : t = f(a_1, \ldots, a_n) \text{ in } e \\
\text{let external } v = (f : t\gamma)(a_1, \ldots, a_n) \text{ in } e \\
\text{if } a_1 \text{ relop } a_2 \text{ then } e_1 \text{ else } e_2
\end{align*}
A formal C compiler

Functional intermediate representation

C code

Front-end

IR

Theorem Prover

Automaton library

Axiomatic semantics
Operational semantics
Type theory semantics
Type theory

Back-end

Optimized machine code
Multi-language environments

- Python front-end
- CDS front-end
- ML front-end
- C front-end

IR Generation
Optimizations
Back-ends

Theorem prover
Reasoning
Automation
Optimization

machine code
Summary

- LPE: leverage existing formal methods and tools for cooperative control problems
  - The goal is to provide a library of verified control primitives, and design automation procedures

- Migration path
  - The compiler provides the guide for migrating code