Automated Reasoning in LabVIEW

An Introduction to the Method/ACL2 System

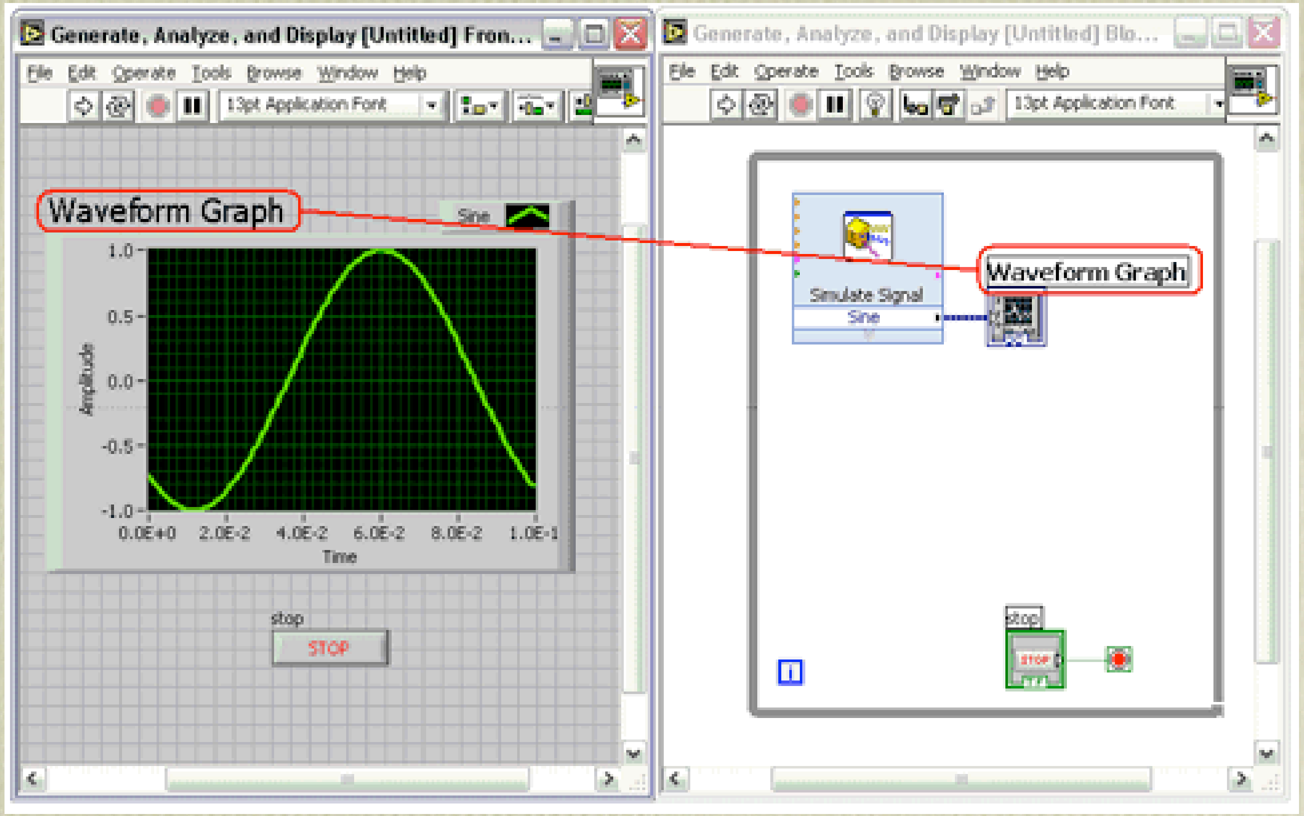
Grant Olney Passmore + ACL2 Seminar

Method/ACL2 work is a group effort together with Jacob Kornerup and Jeff Kodosky.

LabVIEW

- Graphical, concurrent, *data-flow* programming language.
- Pioneered notion of "Virtual Instruments" as software renditions of solutions whose realization was classically in specialized, tediously developed, and expensively manufactured hardware.
- Extremely popular in mission critical testing and automation.
- Can now target any 32-bit microprocessor, and is increasingly popular as a development environment for embedded systems.

LabVIEW: Syntax


Generate, Analyze, and Display [Untitled] Fron...
File Edit Operate Tools Browse Window Help
13pt Application Font
Waveform Graph
Sine
Amplitude
Time
stop
STOP
Generate, Analyze, and Display [Untitled] Blo...
Simulate Signal
Sine
Waveform Graph
stop

- Development of a VI consists of designing, in tandem, a ***front panel*** (user interface) and ***block diagram*** (algorithm) for the function of focus.

LabVIEW: Semantics

- Data-flow determines order of execution.
- ***Block Diagrams*** (algorithms) consist of functions, which are represented as icons, wires that connect these icons, and structures that control execution logic.
- Data flows from one function to the next. So, in a restricted subset of the language, LabVIEW is purely applicative - it is all just functional composition!
- Each function on a diagram does not execute until all of its input terminals have data available for processing.
- Thus, data-flow is rooted in 0-ary functions (constants and user input boxes), and a data-flow partial order may be discerned recursively with 0-ary functions as a base

LabVIEW + A.R.

- Due to its placement as a key development environment for mission critical and embedded systems, LabVIEW is an especially ripe candidate for which a strong application of formal methods and automated reasoning may be very fruitful.
- Began working on this problem in June, 2005. Starting initially with Otter, I proceeded by axiomatizing G diagrams by hand in a Hilbert-style fashion, and producing proofs of simple structural theorems as a first proof of concept.
- Shifted focus to ACL2 to take advantage of its potent brew of decision procedures, operational semantics, and induction heuristics, with the intent of mechanically translating G diagrams into extensionally equivalent, *fully executable,* Applicative Common Lisp forms.
- Two possibilities: Compilation VS. Interpretation. Compilation more natural for proving theorems about diagrams (machine model need not be mapped!).

Method of Attack: *Theorem Blocks*

- Decided to introduce a new meta-linguistic node block into LabVIEW language, the ***Theorem Block.***
- Theorem blocks are sub-diagrams with any number of input terminals and ***no*** output terminals, except a single terminal which may connect only to other theorem blocks (proof planning).
- Input terminals connect to wires on the object diagram. The theorem block then houses asserted constraints upon the relationships between the values of these wires.
- Thus, proposed theorems may be asserted upon the diagram in the same language as the object diagram itself, giving assertions an executable counterpart within LabVIEW, and allowing the same debugging and visualization tools to be used for tracing the flow of data through assertions interactively!
- If theorem can not be proven at compile time, theorem block is compiled into executable as a run-time assertion.
- In this way, Theorem Blocks may be both object and meta theoretic constructs.

G Compiler & Method/ACL2

- In order to prove the contents of theorem blocks in ACL2, we need both a method for translating the high-level G data-structure into a manipulatable acyclic graph, as well as a method for translating an acyclic graph representing a G diagram into a form acceptable to ACL2.

- Translating G diagram into a manipulatable graph is handled by the new G compiler, written in G!

- G compiler gives us a massive, human unreadable representation of a G diagram as a set of nodes together with their terminal wirings. We call this **IGML**.

- Nothing else given in G Compiler output. Data-flow ordering, node rankings, and all node connectivity must be discovered by a new tool: **Method/ACL2.**

Method/ACL2

- A [Theorem Block Annotated G] diagram => ACL2 compiler.
- Written itself in ACL2! (*Very elegant possibilities here*).
- Provides the means to convert G diagrams into an extensionally equivalent ACL2 form, then translate assertions specified on Theorem Blocks upon such diagrams into ACL2 proof searches and theorem definitions (the latter if the former is found, or SKIP-PROOF is used).
- Provides induction heuristics for translating theorem block assertions made upon shift registers into ACL2 proof searches.
- Will allow rewriting strategies, lemma usage, and hints to be specified on G Theorem Blocks as attributes wired to the **Proof Data** sub-node, and uses such data when guiding ACL2 towards a proof.
- Also uses data-flow ordering of wires connecting different Theorem Blocks as an ordering upon theorem definitions, allowing theorems to be proved in a desired sequence.

Strategy I

Lambdas & Combinatorial Explosions

- Once I had sufficient machinery in place for computing data-flow and connectivity, the prospect of extracting *executable*, extensionally equivalent ACL2 forms seemed very elegant.

- My first approach was to then begin at the node(s) with highest ranking, introduce a lambda term to apply them to their inputs, and recur in this fashion, having ACL2 perform Beta-reduction on the final form to extract an executable ACL2 form.

- The only function symbol I would introduce was that of the entire diagram - combinatorial explosion for branching wires!

Strategy II
Every Node is a Function

- Matt Kaufmann suggested I introduce a new 0-ary function for every node, with their executable counterparts disabled to force simplification.
- Using (local (encapsulate foo)) [via defstub] forms to introduce constrained functions for input nodes.
- Allows theorems to introduce rewrite rules in a strategic way, by always targeting function symbols as atomic units susceptible to rewrite.
- No more combinatorial explosion - nothing is in-lined (no more explicit lambdas to be Beta reduced or high-level lambda for the entire diagram!).
- Caveat: We now forgo executable counterparts for non-primitives, but these may be retained using ACL2-PC::S.

ForLoops
Shift Registers and Induction

- ForLoop Structures provide the first focus for inductive theorem proving in Method/ACL2.
- Termination is guaranteed, thus we may focus on developing induction heuristics which focus only on partial correctness (as the difference between partial and total correctness then collapse).
- Shift Registers are the only inductive structure in the language, are available in all looping structures, and thus developing powerful induction heuristics for Shift Registers is my current focus.

Status

- Nearly all applicative aspects of the language implemented.
- Hard problems that are partially solved: Loops with Shift Registers, via our induction heuristics.
- Hard problems yet to be focused upon: Heuristics for proving termination of WhileLoop Structures, methods for handling temporal properties in parallel diagrams.

Part II: *Automating Interpretability*

- Interpretability, conservativity, ordinal analyses, and combinatorial independencies of theories are all extremely fascinating proof-theoretic results.

- Parsons' Theorem - PI_2 conservativity of I-Sigma_1 over PRA is a deep and beautiful theorem.

- Reverse Mathematics program of Friedman and Simpson has a similar approach: Performing "reversals" of *countable, non-set-theoretic* mathematics to find to which set existence axioms interesting theorems are equivalent.

Holy Grail: *Gentzen's Hauptsatz*

- Cut Elimination is at the heart of most of these results in structural proof theory.

- ACL2 has transfinite induction up to epsilon_0, making it a possible environment for formalizing Gentzen's sequent calculus and his Mid-Sequent Theorem.

- Long-term goal: An environment for automating the discovery of relative strengths of combined theorem proving environments, together with a method for translating proof objects of eligible statements between different theorem proving systems.

- Possible future: Interpretability logic?

- Exciting possibilities! A beautiful problem well deserving of a serious effort.