Cooperating Languages - Phase One Report

Hesam Samimi

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Research Memo RM-2012-002

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Cooperating Languages
Phase One Report

Hesam Samimi

May 11, 2012

Cooperating Languages Framework Status

The initial prototype of *CooperatingLanguages* is available in Squeak.

- It's a framework with a base high-level relational language just to specify problems declaratively. It then allows those problems to be solved declaratively or imperatively by hosting solvers or other languages that are incorporated in.
- Currently we interact with the framework, e.g. set up the environment, invoke programs, etc. in Squeak. We should come up with a better UI, for example a graphical representation as Alan B*suggests, within a web browser, etc.
- It works on top of Yoshiki's *KSObjects*, so it enjoys all the functional reactive (*FRP*) goodness. That is, dependencies are handled automatically and objects can tick and change with time.
- The universal base language in its most expressive form includes the following so far:
  - A hierarchy of class definitions, at top of which is the *Thing* class.
  - Each class definition only includes *attributes* and *constraints* (which are inherited by subclasses)
  - Attributes can be *objects*, *primitives* (*Bool*, *Int*, *Real*), or parameterized sets $Set < T >$. This can be generalized to arbitrary relations in the future.
  - Constraints are in *first-order logic* and can involve *basic linear and nonlinear arithmetic*, *relational joins*, *aggregates* such as *sum* or *size*, as well as *existential* and *universal quantification*.
  - *frame conditions* can also be specified using an explicit listing of *modifiable objects* and *modifiable attributes*.
- We already have incorporated three solvers in it: *Cassowary* (linear arithmetic), *Kodkod* (relational first-order SAT-based constraint solver), and *Z3* (SAT Modulo Theory solver).
- We have streamlined how new languages and solvers (as libraries or external) can be added in. All it takes is:
  - 1. Write an OMeta parser to transform the AST from our universal problem specification language into one suited for hosted language
  - 2. Say how to run it, whether natively with a library or externally via system call

*Alan Borning, University of Washington, consulting to VPRI, Spring 2012.

- 3. Write an OMeta parser to apply the resulting model (or state) from the hosted program's environment onto our base environment

– We will let our future examples motivate which other languages / solvers to bring in

The Bridge Circuit Example

The bridge circuit in Fig 1 is the most interesting example we've made to work so far. All three solvers can handle the same exact problem, so it's interesting.

Figures 2, 3, and 4 list different parts of the program. First, Fig. 2 specifies the general model for the program, in this case physical properties of analog electronic devices. Second, Fig. 3 instantiates objects in this model to specify the specific circuit problem shown in Fig. 1. This may be better done using a graphical UI. Finally, Fig. 4 tells the framework what specific problem to solve in this setting, using what solver, and what objects and attributes are modifiable by the program.

Note that the specific problem we're solving as shown in Fig. 4 is *true*, a tautology. This is because in this case other than the class constraints that are automatically part of the problem specification there is no other constraints that we're adding.

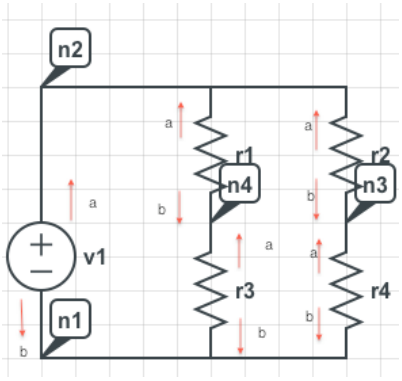

Fig. 1. Bridge Circuit

```
class Node: Thing
        Attributes
                voltage: Real
                leads: Set (Lead)
        Constraints
                sum(leads.current) = 0.0
class Lead: Thing
        Attributes
                node: Node
                current: Real
class TwoLeadedThing: Thing
        Attributes
                lead1, lead2: Lead
        Constraints
                lead1.current + lead2.current = 0.0
class Resistor: TwoLeadedThing
        Attributes
                resistance: Real
        Constraints
                lead1.node.voltage - lead2.node.voltage = resistance * lead2.current
class Battery: TwoLeadedThing
        Attributes
                voltage: Real
        Constraints
                lead1.node.voltage - lead2.node.voltage = voltage

var v1: Battery
var r1, r2, r3, r4: Resistor
var v1a, v1b, r1a, r1b, r2a, r2b, r3a, r3b, r4a, r4b: Lead
var n1, n2, n3, n4: Node
```

Fig. 2. Bridge Circuit - Problem Model

```
"instantiate objects and set fixed attributes..."

n1 := Node new voltage: 0.0; yourself.
...
...
```

Fig. 3. Bridge Circuit - Problem Instance

```
hostProgram: 'true'
in: #Z3
modifies: #(n2 n3 n4 v1a v1b ... r4b 'Node.voltage' 'Lead.current').
```

Fig. 4. Bridge Circuit - Hosting The Program

Onwards

- Decide what examples we would like to do next. Candidates are a pendulum, a bridge, etc.
- Design a neat environment for user to interact with the framework, instead of the current Squeak-based interaction, such as the dual text / graphic representation of classes that Alan B recommends
- Work out how examples where objects time tick should work
- Incorporate an objective version of Cassowary for constraints over real-valued attributes?
- Specification of *cost optimization*, *soft constraints*, and *fix-point* problems in the base language, and plugging into solvers that handle such problems
- motivated by examples, come up with cases of actually "cooperating" solvers