

# IEGen Development Procedures

Alan LaMielle

May 17, 2010

## 1 IEGen Overview

IEGen, the Inspector/Executor Generator, is a tool for generating inspector/executor functions that implement transformed irregular computations. The input is a computation specification file (.spec) and a sequence of transformations to apply to that computation. The output is C code that implements the transformed computation. Details of the techniques used to produce this output are described in my thesis.

## 2 Development Practices

IEGen is developed using a TDD (test-driven development) style and utilizing a bug tracker (Redmine). Consider the issues in Redmine to be the larger feature pieces and descriptions of bugs to be fixed. Consider tests as subcomponents of the issues. When developing IEGen, in general the workflow should be:

1. Add or select an issue in Redmine (feature addition or bug fix) as your current focus of work: 'implement set/relation parsing from a string', 'add depth-first visiting for the AST'.
2. Write unit tests that utilize this feature or exhibit the bug being focused on: 'parsing this string should produce this string as output', 'visiting this AST instance should create this string'. Write many tests for new features, write an appropriate number of tests for a bug (this will depend on the particular bug). The tests written in this step will all fail initially.
3. Implement the necessary code to make the tests written in step 2 pass.
4. Have someone check your work and make a comment to that affect in the redmine issue.
5. Close the issue selected in step 1.

This general workflow should be followed as closely as possible. At times it may not make sense depending on the piece being worked on, but for the most part this is appropriate. Also, 'commit early, commit often'. The smaller the

changes you make the better. Force yourself to break your development into extremely small chunks. Unit tests and issue tracking help with this. Do not commit code that doesn't build and/or passes all tests. Clean up the output of Valgrind (checking for memory leaks) before committing as well.

### 3 Code Style Guidelines

The following guidelines should be adhered to when developing for this project:

- New C++ files should have the extension `.cc` and headers should have the extension `.h`.
- Indent code with spaces, not tabs. 3 spaces is preferred, but be consistent within a file (and preferably project wide).
- Use 80 character width lines or less.
- Document your code using doxygen comment strings (see existing code for examples).
- Commit logs should follow the standard git commit format:  $\leq 50$  characters on the first line, skip a line, and write a more detailed commit message that wraps at 80 characters after the summary line. Refer to redmine issues in the commit log (references `#...` or closes issue `#...` for example).

### 4 Git

Git is a bit different from svn: svn is a centralized version control system. This means that there is one repository that everyone commits to and updates from. This works well in some environments where you know that the server will always be available and you want to control access to the code and where it is stored. However, there are drawbacks to such a system, such as having one central point of failure. Git, on the other hand, is a distributed version control system. Every user of git *has their own independent repository*. What does this mean? When you want to start working on a project, you make a copy (or clone in git speak)—this is equivalent to checking out from an svn repo. Cloning makes a complete copy of the whole repository, history and all. You can now commit locally, examine history locally, and do everything else one would like to do with a version control system. Most of the operations (like looking at history/logs) in svn required network access to talk to the server. This is not the case with git as you have all of that information locally. The benefits of a distributed VCS are quite numerous.

Getting changes from/to another repository is called pushing and pulling in git. For convenience, my server will be hosting the 'main' repository: this is the repo that we will use as an intermediate point for propagating changes to/from

one another. This is similar to svn, but is not strictly necessary. If we could easily reference our individual repositories, we could push/pull from each other directly.

What you need to do (a basic git tutorial): My server is using public/private keys to authenticate users. This means you will need to give me your public key so that you can be authenticated. If you aren't familiar with this, please read through this 3 part article: <http://www.ibm.com/developerworks/linux/library/l-keyc.html> It is very important that you keep your private key secure and use a strong passphrase so even if someone obtains the private key, it will be useless to them.

To generate a key pair: `ssh-keygen -t dsa`

Before you can do anything, you have to send me your public key.

Once this is done, you can clone the IEGen repo: `git clone git@iegen.com:iegen`  
This will clone the repository (make a copy of the repo locally on your machine).

Basic workflow:

- `git pull` (grabs the latest changes from iegen.com)
- (make changes to files)
- `git add [changed files]` (adds the changed files to the 'staging area' to make them ready to be committed)
- `git commit` (commits the changes locally)
- `git pull --rebase` (syncs with the repo at iegen.com and replays and local commits after the remote commits)
- `git push` (pushes any local changes to iegen.com)

Before doing your first 'git commit', you will need to tell git your name and email so that it will be included in the commit info. There are two ways to do this. The first is to place the attached file in `~/.gitconfig` (don't forget the dot before gitconfig) and change the appropriate fields. This way will work across all repos unless you specify different configurations in a per repo basis. The other way is to use two commands to set your name and email just for the one repo:

```
git config user.name Your Name
```

```
git config user.email your@email.com
```

Git log messages: Git log messages have a convention that some tools look for and I'd like to stick to. The first line is  $\leq 50$  chars and should be a summary of the commit. If more space is needed, skip one line and provide a more detailed description of the commit. When I commit, vim is brought up and it has syntax highlighting that reminds me of this convention. See some of my commit messages in the logs for examples.

Binaries that are produced from other files in the repository do not belong in the repo. It is not such a big deal in our case (though it does add unnecessary size to the repo which I'd like to avoid), but in larger projects (like enlightenment,

```
[user]
  name = Your Name
  email = your@email.com
[core]
  whitespace=fix,-indent-with-non-tab,trailing-space,cr-at-eol
  pager=/usr/bin/less -R
[color]
  branch = auto
  diff = auto
  status = auto
  ui = true
[color "branch"]
  current = yellow reverse
  local = yellow
  remote = green
[color "diff"]
  meta = yellow bold
  frag = magenta bold
  old = red bold
  new = green bold
[color "status"]
  added = yellow
  changed = green
  untracked = cyan
[alias]
  st = status
  ci = commit
  br = branch
  co = checkout
  di = diff
  lg = log -p
```

Figure 1: Example gitconfig

xmms2, and other open source projects), people will flip out if you commit binaries that don't belong.

That's about it. A good cheat sheet is here: <http://zrusin.blogspot.com/2007/09/git-cheat-sheet.html>

Again, before any of this will work I need your public key. Send it to me and you can clone the repo.

## 5 Misc

Just as with IEGen, a huge piece of the tool is the backend, the representation of sets and relations. We're hoping ISL will be able to do what we want after adding support for UFSs. I don't know how much can actually be done on the rewrite until we have something like ISL. I propose that we assume we're going to use this library. We may be able to write a wrapper layer around ISL so that we can swap out backends if necessary.

For IEGen itself, here are the main components: -Input: some representation in the form of the MapIR that represents the computation to transform -Transformation application: apply transformations in sequence to the MapIR and build the IDG -Inspector code generator: generate code for the inspector based on the IDG -Executor code generator: generate executor code -Backend representation library: ISL currently is the choice

I can break each of these components down into smaller blocks too. The following data structures will be needed to support these components:

- MapIR: Structure storing statements, etc.
- Transformation hierarchy: structures for representing different transformation types
- IDG: structure (nodes, graph) for representing the IDG
- Code IR: Maybe we can use the Rose AST for this, but in IEGen currently I generate an IR rather than text for code generation. I then traverse this IR to generate the code. The idea is that it allows us to potentially generate code for different languages.

Implementation detail questions:

- How reliant should we be on using classes?
- Should we use structs rather than C++ classes?
- What will our memory management strategy be? When do we dynamically allocate structures vs. local (stack) variables.
- Code format/style guidelines. (Tabs vs. spaces, where to braces go, etc. This ties in with the tools below.)
- Commit log style guideline.

Development tools and methodologies:

- Testing: we will practice test-driven development whenever possible. We will setup a unit-testing system and a regression testing system. Unit tests will be utilized to ensure smaller more fine-grained functionality is working as it should. Regressions tests will be used to ensure that larger components are working properly.
- IDE: We will attempt to utilize Eclipse and it's C/C++ development tools.
- Version control: Git integrates with Eclipse, so we will utilize git to do version control.
- Documentation: We will focus on creating good documentation of all of the sub-components we develop. Currently it looks like you are already familiar with Doxygen, so we should count this in as the default tool. Maybe there are other tools that work well with Eclipse that we can look at though.
- Bug tracking: I think we've both been really happy with Redmine for this task. We'll continue using this unless we see a strong motivation to switch.
- Google Wave: We've both seen that this can be a really powerful tool for certain tasks. (How)Should we integrate this into our development practices?