How to Succeed Against Increasing Pressure: Automated Techniques for Unburdening Verification Engineers

James Pascoe, Pierre Kuhn
STMicroelectronics R&D Ltd.
1000 Aztec West, Bristol, UK
+44 (0)1454 46 { 2377 | 2427 }
{ james.pascoe | pierre.kuhn-ext } at st.com

Steve Hobbs
Cadence Design Systems Ltd.
Bagshot Road, Bracknell, UK
+44 (0)1344 8653147
stephenh at cadence.com

***Abstract*—The Release Management System (RMS) is a methodology that has been developed by STMicroelectronics (ST) in collaboration with Cadence. The motivation for this work is to identify and develop automated techniques to relieve verification engineers from routine yet time consuming tasks. In particular, the RMS features formal unreachability analysis[1] and a novel 'gatekeeper' flow that determines optimal sets of regressions to run on designer commits. This paper describes the RMS and focuses on two of its features, namely, the gatekeeper and the unreachability flows. The intention is to enable the reader to implement these ideas in their own environment. To this end, we supply sample scripts in Appendices A and B.**

***Index Terms*—Unreachability analysis, gatekeeper flow, formal verification, continuous integration, automated verification.**

I. Introduction

The scope of verification is changing. Verification teams are under increasing pressure to perform tasks that (until recently) were not considered part of the verification process. In the CPU/GPU team at ST[2], verification engineers are responsible for design integration, release management and providing customer support. These requirements are additional to the group's existing commitments and are not mitigated by increases in timescale or resource. Furthermore, anecdotal evidence suggests that these observations are not specific to ST. Moreover, there is a growing trend for verification engineers to provide services that are not directly related to their core mission. Ultimately, these activities consume resource and impact on verification quality.

This paper addresses a manifestation of this problem (at ST) by describing the Release Management System (RMS). The RMS is an automated build service that includes features for improving verification efficiency. The RMS philosophy is to automate as much as possible, to perform event-triggered error checking and to produce commonly requested commodity data. Thus, the RMS automates tasks such as integrating developer commits, performing coverage analyses and conducting qualification runs. This paper describes the RMS and focuses on two of its features, namely, a novel approach to gatekeeping and a flow to evaluate the reachability of coverage states.

II. Background and Motivation

The CPU part of the CPU/GPU team consists of approximately ninety engineers that are situated in Bristol (UK), Grenoble (France) and Noida (India). The CPU team develops ARM based processor sub-systems for use in mobile computing and consumer based SoCs. Since Q2 2011, the CPU team have been developing (and hardening) a sub-system for a suite of mobile computing platforms. The design work is conducted in Bristol and Grenoble, while verification is performed across all three sites. The Bristol verification team is responsible for block-level and formal work, whilst Grenoble performs Low Power verification and Noida leads the system level testing.

As the project has progressed, the role of the verification team has grown to encompass a number of additional responsibilities. For example, the Bristol team is now responsible for merging individual developer commits into consolidated and regressed global releases. Furthermore, the team in Grenoble now provide a *de facto* customer interface and are responsible for a range of integration and support issues. In addition, our colleagues in Noida are frequently required to produce commodity data such as coverage reports, qualification runs and regression statistics. These activities, while necessary for the project, are time consuming, mundane and are not mitigated by additional resource.

A preliminary study showed that for a typical month[3], the group was spending around 35% of its effort on activities that were not directly related to verification. In particular, the following observations were made:

1) Over 90% of manually merged developer commits required no human intervention i.e. could be merged automatically;

[1]The formal aspects of the RMS are implemented using the Incisive Enterprise Verifier (IEV). The RMS does not exhibit any particular dependence on IEV - other proof engines can be used.

[2]CPU/GPU is a multi-site team that produces processor sub-systems for use in ST's products.

[3]In this context, 'typical' refers to the period where the project is in steady-state. These measurements were made in June 2012.

2) Verification engineers were frequently debugging regression failures that were due to inadequate designer testing prior to commit;
3) Despite significant effort, code coverage only improved marginally;
4) When provided with clear technical feedback, designers improved the quality of their commits;
5) Requests for coverage analyses, qualification runs and regression reports were ad-hoc and disruptive;
6) The proportion of 'dead' code and the reachability of coverage states was not known.

To address these points, the CPU/GPU team have developed the Release Management System (RMS). The RMS philosophy is to automate as much as possible, to be implementable using commonly available tools and to provide a pragmatic interface that can be driven by engineers, managers and scripts. The RMS automates the merging of designer commits. Integrator interaction is only required to resolve conflicts. Furthermore, the RMS implements a 'gatekeeper' flow that was developed with Cadence. This flow performs integrity checks on commits as they are submitted. Feedback is provided to designers when problems arise – the idea being to prevent build errors and regression failures from impacting the main-line. In addition, the RMS automates the collection and delivery of coverage, qualification and regression data. Coverage data is processed with an unreachability flow (also developed with Cadence) to provide a true representation of the coverage that is achievable through testing. Collectively, this data provides the basis for the decision making process.

III. The Release Management System

The RMS operates in a similar manner to a Continuous Integration (CI) server [1]. Continuous integration is a software practice that improves delivery times by reducing integration delays. Developers are encouraged to make small frequent commits which are merged with main-line code and tested in quick succession. Thus, merges tend to be simple and problems with erroneous commits are detected quickly.

The literature shows that there are numerous freely available CI packages. During the literature review for this paper, the authors counted 33 distinct frameworks. Five of the most popular packages are shown in Table I.

While the RMS draws inspiration from this work, its architecture is optimised to support automated processes such as the gatekeeper flow and the reachability analysis. User's interact with the RMS through email. This interface was selected because it is familiar and can be operated by both humans and scripts. In addition, the design teams were already using email to send labels to the Bristol integrator. The RMS server periodically polls a mailbox. When an RMS email is received, `procmail` is used to parse the commands and launch jobs. The RMS server runs under Fedora Core 16.

Name	License	References
CruiseControl	BSD Style License	[2], [3]
Jenkins/Hudson	Creative Commons / MIT	[4], [5]
BuildBot	GNU Public License	[6], [7]
BuildMaster	Proprietary	[8], [9]
LuntBuild	Apache 2.0	[10], [11]

TABLE I
Popular Packages for Continuous Integration

One use case (which has improved both developer efficiency and code quality) is that designers can now submit labels as they leave the office. When they return, a new release and a coverage database are waiting. Alternatively the RMS can reject a label (typically due to a regression failure) and so provides details for the developer to address.

A. Gatekeeper Flow

One problem with integrating developer labels is in determining the 'correct' set of regressions to run. At ST, designers would unknowingly request main-line integration of labels which were later found to contain errors. Although developers were running 'smoke' tests before submission, it was often the case that these tests did not exercise the modified code and so were providing a false sense of correctness. Similarly, developers were unwilling to run the full regressions due to the length of time taken. Thus, a 'gatekeeper' flow was developed to identify an 'optimal' set of regressions i.e. a set of tests that will exercise the modifications whilst being short enough to run interactively.

The functionality of the flow is implemented by a Python script called `targeted_regr.py` (see Appendix A). This script collects data from ClearCase (i.e. the source control system), previous regressions and the compiled design database (see Figure 2). This allows the script to determine which Verilog modules are impacted by the source code modifications. The regression tests are then ranked for their overall coverage contribution to the affected modules. This allows the script to determine the most relevant tests to run. Furthermore, the regression results include information about the past run times for each test. Thus, it is possible to constrain the targeted regression to an approximate time limit.

The gatekeeper flow leverages the capabilities of the Cadence tool-set, however, the script utilises small helper classes to abstract away from the underlying tools. This simplifies maintenance and permits components to be interchanged with

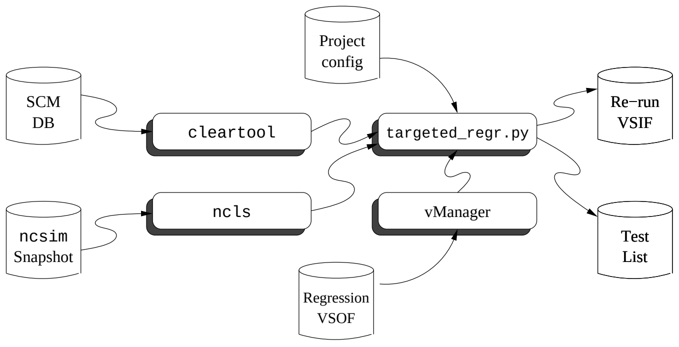

Fig. 2. Targeted Regression Script showing Inputs, Interactions and Outputs

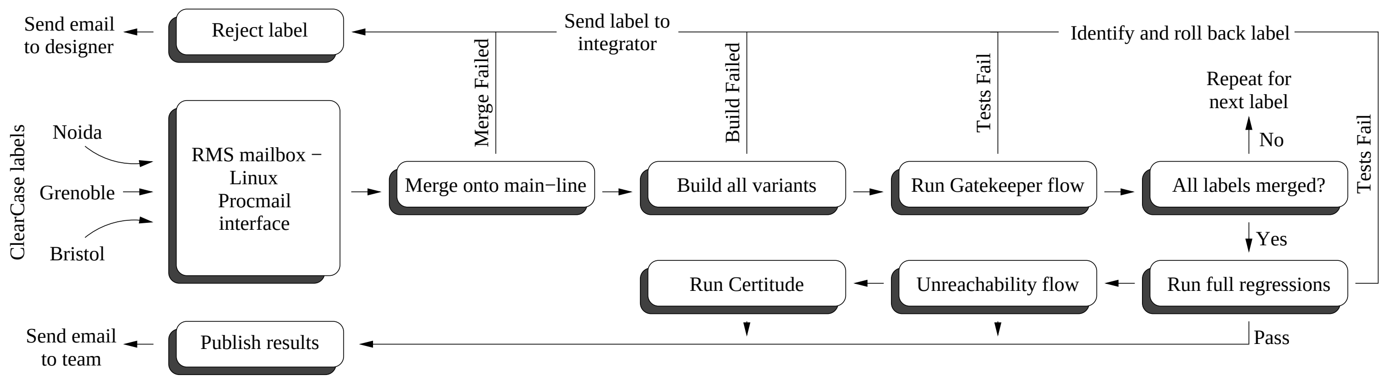

Fig. 1. The Release Management System (RMS) – Showing the Interrelationship between the Gatekeeper and the Unreachability flows

ease. For example, although the team uses ClearCase, any other source control tool can be applied via the project's configuration file. The number of command-line options are minimised to promote genericity and simplicity. Thus, the static and project specific options are stored in a configuration file which can be checked into the project repository. The configuration file is implemented using Python's `ConfigParser` module as this provides a simple, extensible and well documented format. The configuration file specifies:

1) The location of the simulation snapshot;
2) The location of previous regression results;
3) The ranking accuracy (i.e. the trade-off between regression time and test precision);
4) The maximum number of tests in the regression;
5) The maximum amount of CPU time for the regression;
6) The version control suite to use (e.g. ClearCase, SVN).

When the flow is invoked, `targeted_regr.py` interrogates the source repository to retrieve the list of files that are being committed by the designer. Any files matching the user-specified extensions (Verilog by default) are noted. Next, the simulator's compiled database is checked to ascertain which modules are directly affected by each of the modified source files. Using the Cadence Incisive simulator this can be achieved using the `ncls -source` command. The `ncls` command logs every compiled design entity and its corresponding source files, which `targeted_regr.py` stores into a Python dictionary. This facilitates a simple look-up from the changed files back to the module or modules which need to be re-tested.

Once the list of modules is known, a vManager vPlan file is generated. This maps all coverage types (structural and functional) on a module-by-module basis to the vPlan. The vPlan is passed to vManager along with the regression database (i.e. a VSOF). The regression database enables the regression tests to be ranked against the generated vPlan. vManager was adopted in preference to other coverage analysers because vManager supports a powerful yet simple API that allows ranked tests to be filtered according to attributes from the tests. In this context, filtering is based on either one or two attributes, namely, an index into the test list (for limiting the numbers of tests in the run) and cumulative CPU time (to limit the overall run-time). Either limit is optional and is specified in the project configuration file.

vManager also provides benefits in terms of ranking precision (e.g. LOW, MEDIUM or HIGH). This facilitates a trade-off between the accuracy of the results and the time taken to perform the ranking. Note that ranking accuracy is specified in the project configuration file. In addition, vManager also provides the ability to generate a re-run control file (VSIF) which includes the information required to manually launch subsequent regressions based on the randomisation seeds, attributes and scripts from the automated run. This mechanism is particularly useful in the context of constrained random tests. Currently, the CPU team uses a mixture of directed tests (written in C/ASM) and Specman-e, so this flexibility is useful for re-running both types of test in the gatekeeper regression.

Once `targeted_regr.py` has generated the list of regressions, control is passed back to the RMS. Before integrating any labels, the RMS runs the full regression suite on the current release. This provides a source of reference data which is used to detect new failures. Each commit is then merged (in turn) onto the main-line. As each commit is merged, the gatekeeper tests (for that commit) are run and the results are compared to the reference data from the full regression. Any new failures result in the merger being rolled back and details are emailed to the designer.

B. Unreachability Analysis

Each new RMS release is accompanied by a coverage report. This allows designers and verification engineers to frequently monitor progress towards targets. However, not all uncovered items are reachable through functional testing and so the RMS implements an automated reachability analysis to identify and exclude these states. The reachability analysis is particularly useful for identifying redundant RTL.

The reachability analysis is implemented using formal tools. Uncovered items in the coverage database are translated into automatic assertions or 'covers'. These assertions are then 'proved' by `formalverifier` which generates a list of coverage marks. These marks are passed to the coverage tool[4] which excludes the unreachable states from the final results.

[4]In the CPU team, we use the Incisive Metrics Center (IMC).

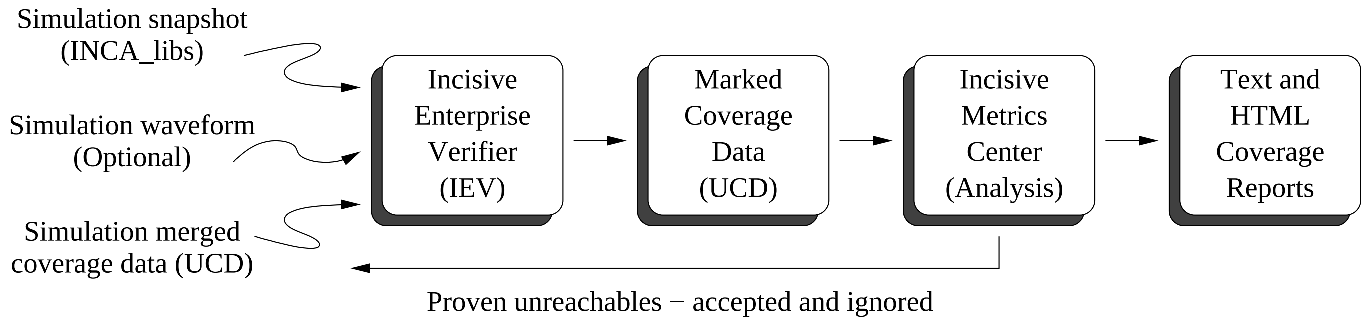
Simulation snapshot (INCA_libs)
Simulation waveform (Optional)
Simulation merged coverage data (UCD)
Incisive Enterprise Verifier (IEV)
Marked Coverage Data (UCD)
Incisive Metrics Center (Analysis)
Text and HTML Coverage Reports
Proven unreachables − accepted and ignored

Fig. 3. Key Steps in the Unreachability Analysis Flow

As with most ASIC developers, ST requires 100% structural coverage. This means that every code block has to be executed, every expression has to be triggered and every net must be toggled. Coverage holes are addressed through improvements in testing or through design reviews. While perfectly valid, design reviews can be time consuming and can lack rigour in the face of time pressure. Thus, the reachability analysis reduces the number of uncovered states and so makes the review process easier.

The first step is to reduce the number of holes that require analysis. Techniques such as Constant Object Marking (COM) and exclude lists can be used to identify design elements that are not expected to be covered. Constant Object Marking is an automatic structural analysis that is applied to toggle coverage by the simulator's coverage elaboration engine prior to simulation. Based on knowledge of the design structure, the elaborator can exclude signals which are tied-off to constant values. This can be useful for certain design styles where a fixed port set has unused optional signals that are tied-off at a higher level. However, COM typically does not provide a large reduction in toggle coverage holes. Furthermore, COM does not work well for other metrics e.g. block or expression coverage as these tend to be too complex for a complete structural analysis in the simulator. Users may augment COM exclusions by specifying certain naming conventions for signals to be excluded e.g. BIST or DFT signals which are typically tested outside of the main verification flow. While useful, both of these approaches are of limited benefit and are not sufficiently complete or robust to fully compensate for the problems described above.

To address these requirements, Cadence has developed a powerful and automated technique that utilises formal property checking to analyse each structural coverage hole. While formal analysis cannot always produce a conclusive proof, this technique significantly reduces the number of holes that the designer must investigate manually. Early prototypes of the flow were tested and used by ST from the outset (circa 2005) [12]. Following the official launch of the unreachability flow in 2011, many ST projects have adopted the technology.

The flow works by taking a merged coverage database (from a normal RTL regression) and extracting formal properties for each coverage hole. Properties are represented in an internal format but are analogous to PSL or SVA cover properties. To maximise automation, the formal tool builds its model of the design using the compiled simulation database and shares the same front-end as the `irun` command. Thus, setup is a case of passing the merged coverage file and the simulation snapshot to the tool (see Figure 3). For each coverage hole, a single property is generated. These properties are then subjected to a formal proof using a variety of algorithms. In addition, the user can accelerate the proof by using multiple CPUs and algorithms in parallel.

Properties that pass indicate areas of code that can be reached by the formal engine. In this case, the tool can provide a waveform which shows an example trace. This can be useful to determine whether the proof is under-constrained i.e. the formal engine is visiting states that are actually unreachable in the design due to a lack of initialisation. More typically, a designer will be interested in analysing the failing properties as these indicate areas of unreachable or 'dead' code.

Once the formal verification has been completed, IEV generates a new coverage database file containing exclusion markers that are distinct from the user defined 'ignore' markers. This allows the designer to load the new database into the coverage analysis tool and apply the unreachable markers to the final report. While it is possible to apply all of the markers in one step, it is generally expected that each marker will be reviewed and appropriate action taken. For example the designer may re-factor the RTL to remove the redundancy.

A key advantage of automating this flow is that the reachability analysis can start earlier in the design cycle. Even if the formal analysis cannot prove every property, the designer has a smaller set of holes to analyse. This is particularly useful as many designers are not comfortable with formal techniques. The unreachability flow improves productivity by providing useful data with no requirement to understand the formal theory. Users can obtain quick (though possibly incomplete) results because unlike traditional formal property checking, the unreachability flow is designed to reduce manual analysis rather than provide exhaustive functional verification. For the purpose of illustration, selected real-world project results are given in Table II. Although the results have been anonymised, they suggest that designs of any size can be tackled. Note that in the final example, the same design was analysed with and without initialisation. This demonstrates how a properly initialised design may exhibit more unreachable coverage items due to the extra constraints that are implicit in the initialisation. Thus, there are fewer holes to analyse manually.

Design	State bits (approx.)	Simulation Holes	% Proven Unreachable	Initialised
1	2000	2116	18%	Uninitialised
2	5000	1073	15%	Initialised
3	11000	19	84%	Uninitialised
4	30000	369	12%	Uninitialised
5	40000	773	8%	Uninitialised
	40000	773	10%	Initialised

TABLE II
EXAMPLE STATISTICS FOR SELECTED UNREACHABILITY ANALYSES

Note that care must be taken to interpret the results from the unreachability flow correctly. As with main-stream property checking, under and over-constraint also apply to unreachability analyses. Under-constraint may lead to dead code being marked as reachable either due to initialisation that is too permissive (e.g. state elements having unknown values) or by failing to constrain external inputs to valid behaviour. By contrast, over-constraint can result in reachable items being marked as unreachable. This means that the designer will be required to perform more manual analysis than is strictly necessary. However, this may be faster than expending extra effort to constrain the analysis more accurately.

In our experience, under and over-constraint, in the context of unreachability analysis, is typically due to the incorrect initialisation of the design i.e. starting the proof from a state other than (the correct) reset. One technique to address this is to sample all state values immediately after reset. These values can then be used to initialise the proof.

IV. RESULTS

The RMS, gatekeeper and unreachability flows have been deployed in the development of a CPU sub-system at ST. This sub-system has been developed for an SoC platform that is targeted at high performance mobile devices and consists of a combination of ARM and ST IP.

The results from the RMS deployment are encouraging. Qualitatively, the RMS has significantly reduced the load on three key engineers. This has meant that engineers who were previously consumed by integration activities are now able to focus on verification. In addition, the gatekeeper flow has meant that designers are running more appropriate tests and verification engineers are performing fewer debug cycles.

The unreachability analysis has proved popular and other ST teams have adopted the flow. The results show that there is very little dead code in the sub-system. This is due (in part) to the ARM IP which contains virtually no redundancy. Table III shows the number of unreachable blocks, expressions and toggles in the sub-system for five releases. These releases occurred at the end of the development phase i.e. just prior to the final deliveries. The values in parentheses indicate the absolute values as a percentage of the total number of states for each category. The results are encouraging and show that almost all of the states are reachable through testing. In addition, the 13.5.0 release was one of the first releases to use the flow. The results from that run identified a number of erroneous cover items that were subsequently excluded.

Release Information			Unreachable States (% of total)		
Tag	Date	Labels	Block	Expression	Toggle
13.5.0	13/08/12	11	412 (0.30)	221 (0.86)	526 (0.14)
13.7.0	29/08/12	15	331 (0.24)	32 (0.24)	528 (0.08)
14.0.0	05/09/12	8	331 (0.24)	32 (0.24)	528 (0.08)
14.4.0	26/09/12	8	332 (0.24)	32 (0.24)	528 (0.08)
14.5.0	04/10/12	10	332 (0.24)	32 (0.24)	528 (0.08)

TABLE III
PROJECT DATA FOR SELECTED RELEASES

V. CONCLUSION

This paper has provided three contributions, namely, in-depth descriptions of the RMS, gatekeeper and unreachability flows. The intention is to provide the reader with enough understanding so that they can deploy these ideas in their own environments. To this end, we have provided sample scripts for the gatekeeper and unreachability flows in Appendices A and B. Details of the RMS are available from the authors.

The flows presented here have not only delivered benefits in terms of tangible effort savings, but have also had positive effects on the team's culture. For example, the engineers in Bristol were significantly more engaged by working on the RMS than they were performing repetitive activities. Ultimately, this project has promoted a spirit of engineering that extends to not only the deliverables, but also to the way in which we work.

REFERENCES

[1] J. L. Gray and G. McGregor, "A 30 Minute Project Makeover Using Continuous Integration," in *Proc. of Design and Verification Conference (DVCon)*, 2012.

[2] M. Parker, *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Apps.* The Pragmatic Programmers, 2004.

[3] J. Schmetzer, "An Introduction to CruiseControl (PowerPoint Slides)," http://www.exubero.com/ccintro/ccintro-s5.html, 2012, last accessed: 25^{th} November 2012.

[4] J. C. H. Page, "Home Page," http://jenkins-ci.org/, 2012, last accessed: 21^{st} November 2012.

[5] M. Moser and T. O'Brien, *The Hudson Book.* Oracle Inc., 2011.

[6] The BuildBot Team, "The BuildBot 0.8.7 Online Documentation," http://buildbot.net/buildbot/docs/current/index.html, 2012, last accessed: 21^{st} November 2012.

[7] E. Software, "The BuildBot Homepage," http://trac.buildbot.net, 2012, last accessed: 25^{th} November 2012.

[8] Inedo, "BuildMaster Overview," http://inedo.com/buildmaster/overview, 2012, last accessed: 21^{st} November 2012.

[9] J. Fisk, "Beginners Guide to Continuous Integration with Sharepoint," http://www.slideshare.net/jamesafisk/continuous-integration-6636466, 2010, last accessed: 21^{st} November 2012.

[10] The LuntBuild Developers, "Build Automation and Management User's Guide," http://luntbuild.javaforge.com/manual/guide/manual.html, 2012, last accessed: 21^{st} November 2012.

[11] ——, "Luntbuild - Build Automation and Management User's Guide," http://luntbuild.javaforge.com/manual/guide/manual.html, 2007, last accessed: 21^{st} November 2012.

[12] G. Faux and J. Müller, "Using Formal Analysis to Improve Dynamic Code Coverage," in *CDNLive EMEA*, April 2008.

Appendix A
targeted_regr.py – Gatekeeper Flow

```python
#!/usr/bin/python
#
# Author: Steve Hobbs (stephenh@cadence.com)
# Date: July 4th 2012
#
import sys
import re
import os
from optparse import OptionParser
import logging
import subprocess
import ConfigParser

class module:
   """ Helper class to represent a Verilog module.
       Currently just wraps around a list of file
           names.
   """
   def __init__(self, name):
      self.name=name
      self.sources=[]

   def add_source(self, file):
      if file not in self.sources:
         self.sources.append(file)

   def depends_on(self, file):
      if file in self.sources:
         return True
      return False

   def dump(self):
      return "Module '"+self.name+"'\n\t"+"\n\t".
          join(self.sources)

class designAccess:
   """ Provides abstracted API for interrogating the
       compiled design database.
   """
   def __init__(self, cdslib, hdlvar):
      self.cdslib=cdslib
      self.hdlvar=hdlvar
      self.modules={}
      self.currentScope=None
      self.ncls()

   def has_module(self, name):
      """ Returns True if the named module exists in
          the design database """
      return name in self.modules

   def add_module(self, name):
      self.currentScope  = name
      self.modules[name] = module(name)

   def dump(self):
      return "\n".join([self.modules[m].dump() for m
          in self.modules])

   def add_file(self, file):
      if self.currentScope != None:
         self.modules[self.currentScope].add_source(
             file)

   def get_associated_modules(self, file):
      """ Returns a list of module names that depend
          on the specified file
      """
      mlist = [ m for m in self.modules.keys() if
          self.modules[m].depends_on(file) ]
      return mlist

   def ncls(self):
      """ Interrogate the Cadence simulator database
          """
      cmd = ['ncls', bits64, '-CDSLIB', self.cdslib,
            '-HDLVAR',self.hdlvar, '-SOURCE',
            '-ABSOLUTE_PATH','-VERILOG',
            '-NOCOPYRIGHT','-NOLOG']
      logger.debug('ncls command is: '+' '.join(cmd)
          )
      ncls = subprocess.Popen(cmd, stdout=subprocess
          .PIPE)
      for line in ncls.stdout:
         # Detect ncls warnings and errors and log
             them
         s = logging.DEBUG
         e = re.search('\*([EWF]),[A-Z]+:', line)
         if e != None:
            if e.group(1) == 'W':
               s = logging.WARNING
            elif e.group(1) == 'E':
               s = logging.ERROR
            elif e.group(1) == 'F':
               s = logging.CRITICAL
         logger.log(s, 'ncls: '+line.rstrip())

         m = re.search('module\s+(\w+).(\w+):(\w+)',
             line)
         if m != None:
            self.add_module(m.group(2))
            continue
         m = re.search('(\S+?)\s+\[lines:', line)
         if m != None:
            self.add_file(m.group(1))
            continue
         m = re.search('Source files:', line)
         if m != None:
            continue
         self.currentScope = None

class vplan:
   """ Helper code to create the vPlan file for
       ranking """
   def __init__(self):
      self.name='targeted_regr'
      self.modules=[]

   def add_modules(self,modules):
      for m in modules:
         self.modules.append(m)

   def get_filename(self):
      return self.name+'.vplan'

   def get_top_perspective(self):
      return 'top'

   def write(self):
      logger.info('Creating vPlan '+self.
          get_filename())
      vp = open(self.name+'.vplan','w')
      vp.write('section "'+self.name+'" {\n')
      # do modules
      for m in self.modules:
         vp.write('section "'+m+'" {\n')
         vp.write('  coverage "metrics" {\n')
         vp.write('    items_pattern : "(hdl,type)'+
             m+'.*";\n')
         vp.write('  }; // '+m+'\n')
         vp.write('}; // '+m+'\n')
      # close vplan
      vp.write('}; // '+self.name+'\n')
      vp.write('perspective "'+self.
          get_top_perspective()+'" {\n')
      vp.write('  top_section : "'+self.name+'";\n')
      vp.write('};')

def try_remove(file):
   """ Helper to catch and warn on removing a file
       from disk """
   try:
      os.remove(file)
   except(OSError):
      logger.warning("Could not remove file '"+file+
          "'")

# Parse command-line arguments
parser = OptionParser()
parser.add_option(
  "-l", "--log", dest="logName", action='store',
      type='string',
  help="Specify the log file name", metavar="FILE",
      default='targeted_regr.log')
parser.add_option(
  "-c", "--config", dest="config", action='store',
      type='string',
  help="Specify a configuration file", metavar="FILE
      ", default='targeted_regr.cfg')
parser.add_option(
  "-v", "--verilog", dest="verilog", action='append'
      , type='string',
  help="Process a Verilog file", metavar="FILE")
parser.add_option(
  "-m", "--module", dest="modules", action='append',
       type='string',
  help="Process list FILE", metavar="MODULE")
parser.add_option(
  "-d", "--debug",
  action="store_true", dest="debug", default=False,
  help="print debugging messages")
(options, args) = parser.parse_args()

# Configure logging
logger = logging.getLogger('tr')
logger.setLevel(logging.DEBUG)
fh = logging.FileHandler(filename=options.logName,
    mode='w')
fh.setLevel(logging.DEBUG)
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
logger.addHandler(fh)
logger.addHandler(ch)

if options.debug == True:
   logger.setLevel(logging.DEBUG)

# Use ConfigParser object to read config file
config = ConfigParser.SafeConfigParser()
config.read(options.config)

cdslib    = config.get('incisive',   'cdslib')
logger.debug('cdslib='+cdslib)
hdlvar    = config.get('incisive',   'hdlvar')
logger.debug('hdlvar='+hdlvar)
bits      = config.get('incisive',   'bits')
if str(bits)=="64" or str(bits)=="64bit":
   bits64="-64BIT"
else:
   bits64=""
logger.debug('bits='+bits64)

vsof      = config.get('regression', 'vsof')
logger.debug('vsof='+vsof)
rankAccuracy = config.get('ranking', 'accuracy')
logger.debug('ranking accuracy='+rankAccuracy)
rankFile = config.get('regression', 'rankfile')
logger.debug('rankfile='+rankFile)
doRun     = config.getboolean('regression', 'run')
logger.debug('run='+str(doRun))
max_tests = config.getint('regression', 'max_tests')
logger.debug('max_tests='+str(max_tests))
max_time  = config.getint('regression', 'max_time')
logger.debug('max_time='+str(max_time))
scmtype   = config.get('scm', 'tool')
logger.debug('scm tool='+scmtype)
scmstatus = config.get(scmtype, 'status')
logger.debug('scm status='+scmstatus)
vlogext    = config.get('incisive', 'vlogext')
logger.debug('vlogext='+vlogext)
rerun_vsif = config.get('incisive', 'rerun_vsif')
logger.debug('rerun_vsif='+rerun_vsif)
rerun_scheme = config.get('incisive', 'rerun_scheme'
    )
logger.debug('rerun_scheme='+rerun_scheme)

# Sanity checks as early as possible:
if os.path.exists(vsof):
   logger.info("Regression VSOF is present")
else:
   logger.critical("Regression VSOF '"+vsof+
      "' is missing, there are no regression results
           to rank")
   exit(1)

# Build list of modules to be ranked/regression
    tested
targetModules={}
# First, any explicitly mentioned modules
if options.modules != None:
   for m in options.modules:
      logger.info("User requested ranking for module
          '"+m+"'")
      targetModules[m]=True
else:
   logger.info("No Verilog modules were specified at
        the command line")

design = None
# Then any modules that are dependants of any
    Verilog files that changed
logger.info("User specified Verilog files to be
    checked")
# Use ncls to get design info into memory
design = designAccess(cdslib, hdlvar)
logger.debug(design.dump())

if options.verilog != None:
   for f in options.verilog:
      logger.info("Finding module dependencies for
          file '"+f+"'")
      for m in design.get_associated_modules(f):
         logger.info("Module '"+m+"' is affected by
             '"+f+"'")
         targetModules[m]=True
else:
   logger.info("No Verilog files were specified at
       the command line")

# Query the SCM tool if no modules or files were
    specified on the command line
if (options.verilog == None) and (options.modules ==
     None):
   logger.info("No Verilog files or modules were
       specified; interrogating SCM tool")
   found = False
   logger.debug("Running SCM status command: "+
       scmstatus)
   scm = subprocess.Popen(scmstatus, shell=True,
       stdout=subprocess.PIPE)
   scm.wait()
   expr="(\S+?\.("+vlogext+"))"
   logger.debug("expr="+expr)
  for line in scm.stdout:
    logger.debug(scmtype+': '+line.rstrip())
    m = re.search(expr,line)
    if m != None:
      #if design == None:
      #   logger.debug("Going to get design info
          from the compiled library")
      #   design = designAccess(cdslib,hdlvar)
      f = m.group(1)
      if f[0] != '/':
        logger.debug("Got a relative path '"+f+"
            ' from SCM; prefixing it to get
            absolute path")
        f = os.getenv('PWD')+'/'+f
      logger.info("Finding module dependencies
          for file '"+f+"'")
      for m in design.get_associated_modules(f):
        logger.info("Module '"+m+"' is affected
            by '"+f+"'")
        targetModules[m]=True
        found = True
  if found == False:
    logger.warning("SCM did not report any changed
        Verilog files")

# Check that all specified modules exist in the
    design
for m in targetModules.keys():
  if design.has_module(m):
    logger.info("Module "+m+" exists in the design
        database")
  else:
    logger.error("Module "+m+" does not exist in
        the design database")
    del targetModules[m]

# Ensure at least one module has been specified for
    ranking,
# once all pruning etc has happened.
if len(targetModules.keys()) == 0:
  logger.critical("No (valid) Verilog modules were
      found or specified, therefore no ranking can
      be done")
  exit(2)

logger.info("Final list of modules: "+' '.join(
    targetModules.keys()))

# Build vPlan for ranking
vp = vplan()
logger.info("Building vPlan '"+vp.get_filename()+"'
    for ranking")
vp.add_modules(targetModules.keys())
vp.write()

# Build ranking script on-the-fly
ecom = open('rank.ecom', 'w')
ecom.write('// Auto-generated - DO NOT EDIT\n')
ecom.write('set notify -severity=IGNORE
    REGISTERING_ATTR_DEFINITION_FILE;\n')
ecom.write('set notify -severity=IGNORE
    DUPLICATE_ATTRIBUTE_DEFINITION;\n')
ecom.write('set notify -severity=IGNORE
    VM_MSG_VPLAN_MISSING_CVR_ITEMS;\n')
ecom.write('setup;\n')
ecom.write('var my_context :vm_context;\n')
ecom.write('var vsof:= vm_manager.read_session("'+
    vsof+'");\n')
ecom.write('my_context = vm_manager.create_context({
    vsof}, "Default");\n')
ecom.write('my_context.read_vplan("'+vp.get_filename
    ()+'");\n')
ecom.write('my_context.set_perspective_by_name("'+vp
    .get_top_perspective()+'");\n')
ecom.write('var tr_rank := my_context.
    create_vplan_attribute("tr_rank", GRADE, "/
    targeted_regr");\n')
ecom.write('my_context.rank_with_accuracy(tr_rank ,
    '+rankAccuracy+');\n')
ecom.write('my_context.add_filter(vm_manager.
    get_attribute_by_name("rank_id"), ">=", "0");\n'
    )
ecom.write('print my_context.get_groups();\n')
if max_time > 0:
  # Implement limit on run-time
  ecom.write('var ccpu_attr := vm_manager.
      get_attribute_by_name("cumulative_cpu_time")
      ;\n')
  ecom.write('my_context.add_attribute(ccpu_attr);\
      n')
  ecom.write('my_context.add_filter(ccpu_attr,
      "<=", "'+str(max_time*60)+'");\n')
ecom.write('var curr_runs := my_context.get_groups()
    ;\n')
ecom.write('print curr_runs;\n')
ecom.write('var name_attr := vm_manager.
    get_attribute_by_name("full_title");\n')
ecom.write('var names := curr_runs.apply(it.
    get_attribute_value(name_attr));\n')
ecom.write('print names;\n')
ecom.write('var f := files.open("'+rankFile+'","w","
    Text file");\n')
if max_tests > 0:
  # Implement limit on number of tests
  ecom.write('for each in names {if (index<'+str(
      max_tests)+') {files.write(f,it)}};\n')
else:
  ecom.write('for each in names {files.write(f,it)
      };\n')
ecom.write('files.close(f);\n')
ecom.write('my_context.create_rerun_vsif("'+
    rerun_vsif+'", vm_manager.obtain_rerun_scheme("'
    +rerun_scheme+'"));\n')
ecom.close()
try_remove(rerun_vsif)
try_remove(rankFile)

# Use vManager to rank the tests against the
    selected modules
cmd = ['vmanager', '-batch', '-command', '@rank.ecom
    ']
logger.debug("Running vManager with: "+' '.join(cmd)
    )
logger.info("Ranking tests. This may take some time,
     please be patient!")
iem = subprocess.Popen(cmd, stdout=subprocess.PIPE)
iem.wait()
for line in iem.stdout:
  # Capture vManager output to the log
  logger.debug('iem: '+line.rstrip())

if options.debug != True:
  try_remove('rank.ecom')

if os.path.getsize(rankFile) > 0:
  logger.info("Ranked test results are ready in
      file '"+rankFile+"'")
else:
  logger.error("Ranking produced an empty test list
      .\n"+
    "Try checking that you have recorded coverage
        for the following modules:\n\t"+
    "\n\t".join(targetModules.keys()))
  exit(3)

if doRun == True:
  # Run a regression directly, if requested
  logger.info("Starting regression")
```

```
    ecom = open('rerun.ecom', 'w')
    ecom.write('// Auto-generated - DO NOT EDIT\n')
    ecom.write('set notify -severity=IGNORE
        REGISTERING_ATTR_DEFINITION_FILE;\n')
    ecom.write('set notify -severity=IGNORE
        DUPLICATE_ATTRIBUTE_DEFINITION;\n')
    ecom.write('set notify -severity=IGNORE
        VM_MSG_VPLAN_MISSING_CVR_ITEMS;\n')
    ecom.write('start_session -vsif '+rerun_vsif+'\n'
        )
    ecom.close()
    cmd = ['vmanager', '-batch', '-command', '@rerun.
        ecom']
    logger.debug("Running vManager with: "+' '.join(
        cmd))
    iem = subprocess.Popen(cmd, stdout=subprocess.
        PIPE)
    iem.wait()
    for line in iem.stdout:
       logger.debug('iem: '+line.rstrip())
    if options.debug != True:
       try_remove('rerun.ecom')
else:
    logger.info("Finished preparing test list. User
        must now start the regression manually")

exit(0)
```

Listing 1. Gatekeeper Flow (Python)

Appendix B
unreachability_analysis.sh – Bash Script

```
#!/bin/bash

date=`date +"%d%b%Y"`
dutModule=<your DUT module>
dutLibrary=<your DUT library>
snapshot=${dutLibrary}.iev_unr_snapshot
covdb=<path to your coverage database>
cds_hdl=<path to your cds.lib and hdl.var files>

if [ ! -d $covdb ]; then
  echo "Failed to find the coverage directory '
      $covdb'. Aborting."
  exit
fi

# Write a new coverage script file
cat > cov_file <<EOF
select_coverage -all -instance <your instance name>
deselect_coverage -all -instance <deselections>
EOF

# Re-elaborate the snapshot
ncelab \
  -access +rwc \
  -NOWARN DLCPTH \
  -coverage all \
  -covfile cov_file \
  -covdut $dutModule \
  -nowarn MRSTAR \
  -timescale 1ns/1ps \
  -update \
  -nowarn CUDEFB \
  ${dutLibrary}.${dutModule} \
  -snapshot $snapshot \
  $cds_hdl

if [ $? -ne 0 ]; then
  echo "Failed ncelab stage, aborting"
  exit
fi

echo ncls -snap $cds_hdl

# Build the model and the unreachability assertions
formalbuild \
  -messages \
  $snapshot \
  $cds_hdl \
  -nohal \
  -enterprise \
  -coverage all \
  -covunit module \
  -covdb $covdb \
  -nowarn PRMFSM \
  -nowarn PRTCON \
  -nowarn BADFSM \
  -nowarn EXTFSM \
  -nowarn NONGEN \
  +fvips

if [ $? -ne 0 ]; then
  echo "Failed formalbuild stage, aborting"
  exit
fi

# run FormalVerifier and produce the refinements
cat > formal_verifier_commands.tcl <<EOF
prove
exit
EOF

formalverifier \
  $snapshot \
  $cds_hdl \
  -enterprise \
  -coverage all \
  -covunit module \
  -input formal_verifier_commands.tcl \
  -sv_lib <your libuvmdpi.so path> \
  -covdb $covdb

if [ $? -ne 0 ]; then
  echo "Failed formalverifier stage, aborting"
  exit
fi

if [ -r iev_ignored_coverage.cf ]; then
  echo "Found the coverage marks file."
else
  echo "Failed to find iev_ignored_coverage.cf,
      formalverifier must have failed. Aborting."
  exit
fi

# Run IMC and generate the coverage reports
cat > imc_reachability_commands.tcl <<EOF
load $covdb
convert_icf iev_ignored_coverage.cf -out
    iev_ignored_coverage.tcl
source iev_ignored_coverage.tcl
save -refinement iev_ignored_coverage.vrefine
load -refinement <path to your refinement file>
report -detail -html -out marked_coverage_${date} -
    source on -overwrite
exit
EOF

imc -batch -init imc_reachability_commands.tcl

if [ $? -ne 0 ]; then
  echo "Failed imc reporting stage, aborting"
  exit
fi
```

Listing 2. Unreachability Analysis Script (Bash)