Detecting TravisCI Misconfiguration

Plan of Attack

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

We can detect TravisCI misconfigurations using ConfigC.

1. Availability of Data

TravisCI has recently released a metadata api to study their tool[**?**].

The first use of this API was analyzing the metadata of builds. It was found that ˜15-20% of failed TravisCI builds are due to "errors". My understanding of the API is that this corresponds to a misconfiguration. This is a large enough number that not only will reducing this be more convenient for users, but it could also significantly reduce server costs for Travis. By using the `tr_duration` field from the API, we can also figure out how much server time we can save[**?**].

2. Feasability Anaysis

The language of TravisCI configuration files can likely be handled by ConfigC. ConfigC works on languages with shallow parse trees, and most of the .travis.yml files is shallow. However, because .travis.yml allows for bash scripts, some error may be out of scope. We want to identify the frequency of the types of errors - if too many are related to poorly formed bash scripts, a naive application of ConfigC will not suffice.

To confirm that the types of most misconfigurations are in the scope of ConfigC, we should compile a collection of common misconfigurations. Usually such a task requires a domain expert. However, in this case we may be able to learn the common errors by using the API and the temporal data from the commits.

We might code an error as the diff between a sequential pair of an erroring commit and a passing commit. As a new direction, we may also explore how useful this data is for learning/verification directly. My guess is that is data will be too noisy, but either way, it should give us a sense of the problem.

To build such a database, scrape the .travis.yml files from each commit for a set of repos. This can be achieved with the `tr_status, git_commit and gh_project_name` fields from the API.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 Learning from Temporal properties

An intersting extension to ConfigC on a theortical side is to look at the temporal relations between commits. The key observation is that with each commit that changes the build status, we can learn highly localized information about our model.

The first step is to build an intermediate representation of the data we will learn. For the sake of efficiency, this data must be structured as a shallow tree. While the travis.yml file can be taken verbatim, we must use a simplification of the program code. Since we are only intrested in the parts of the program effecting build status, we should extract the imported libraries as a list. We will call this summary S_t, a simplified representation of the program which contains only the information relevant to the learning process. The subscript is a tag based on the ordered commit history. To handle branches, add a superscript to indicate the branch, and restart the counter on a branch - merging should be handled easily (not exactly sure how yet).

From this summary we can then build a model $M(P_t)$, which is the full set of possible rules derivable from the model. These possible rules must be given by the user. In [**?**], these were called Attributes, and express constraints such as, ordering of lines, or substring relations on values.

We will denote the build status of S_t with $B(S_t)$. In this application, we consider only the passing and erroring build status, denoted P, and E respectivly. All status that are not `error`, as defined by the Travis API, will be included as passing. For brevity, we denote sequences of build statuses with the following notation:

$S(P_t) = P \wedge S(P_{t+1}) = E \implies S(P_{t,t+1}) = PE$

We know that if a build is passing, then there must not exist any errors. That is, the model must not contain any rules which are breaking. Note we are not, however, garunteed that any rules from a passing commit are necessary.

$S(P_t) = P \implies \forall r \in M(P_t), r \notin Br$

The key insight is that when we commit a break, we can localize the error to one of the lines that changed. Either we removed something that was necessary, or added something that was breaking. We use an inclusive disjunction, since a erroring commit can break multiple things at once. Expressed formally, where ($\setminus$ is the set difference), that is:

$$S(P_{t,t+1}) = PE \implies$$
$$\exists r \in (M(P_t) \setminus M(P_{t+1})), r \in Nec \vee$$
$$\exists r \in (M(P_{t+1}) \setminus M(P_t)), r \in Br$$

We then can combine all these formulas with conjunctions and send it to an SMT solver and magically get an answer. While exstential set operations can be expensive in general for an SMT solver, in our application this is not the case. Because these sets will be small thanks to the practice of making incremental commits

when using source control, these operations will be fairly cheap. In fact, the above implication generalizes to $P_{t,t+n}$, but for efficency we must require that $M(P_t) \setminus M(P_{t+n})$ is managably small. The definition of small here remains to be experimentally determined.

3. Implementation

To build the learning set, we will use the API to build tuples of .travis.yml and repo info. In addition to the .travis.yml file (collected as above), we will need at a minimum `gh_lang and tr_status`. With this `(File,gh_lang,tr_status,...)` tuple, we can then (almost) directly apply ConfigC to detect errors in the shallow part of the tree.

The first (and simplest) requirement is a .travis.yml parser, which is an extension of normal yml parsing to handle travis' ability to include bash scripts. Second, depending on the results from Sec 2, we may also require some extra domain knowledge of the types of errors to be encoded as possible rules. This is a slightly more involved task that requires writing some non-trivial Haskell code - but it really isn't that bad.