Propi Plug-in: Semantic Property Tool Support

Eoin O'Connor, *Student Member, IEEE*,

***Index Terms*—Semantic Property, Semantic Property Instance, ...**

I. Introduction

THESE days software projects are larger and more complex than they have ever been. To deal with this it has become common practice to separate a projects design from its implementation. This process is called abstraction. The design is often represented by a modeling language and the implementation by a programming language. This allows a team to display the design in a form that can be understood by programmers and non programmers alike. This in turn allows people that don't have an in depth understanding of a projects implementation to contribute to the design of the project. This is beneficial as the more people that contribute to the design of a project the more likely it is to be a good design.

In this paper we will use the term level. A level is a representation of a project in a specific language. Some examples of levels would be modeling language levels or programming languages levels. One of the major drawbacks of having multiple levels for a project is keeping the levels in sync. When we say in sync we mean keeping the meaning of the levels equivalent.

The development process goes as follows: Firstly, the design team get together and design the modeling level for the project. Then, the programming team write the code level so that it is equivalent to the modeling level specification. This works well for the first iteration of the project. However, during most software projects the code implementation will be changed and refactored many times during the development process. The problem arises when the programmer needs to change the code level in such a way that the two levels become out of sync. Now in order to keep the design team apprised of these changes the programmer has to update the model level to reflect the changes. This is manageable if there are only a few changes but in a project that involves significant changes this quickly become a serious time sink for the programmer. It would be beneficial if when a programmer changes the code level the other levels in the project would automatically get updated.

To enable automatic syncing the levels of a project need to be related in some way. This is where semantic properties come in. Semantic properties (or properties) allow us to give meaning to the comments at different levels. When defining a property we define the form a property should take for each level and how one level refines to another. This means we can compare two instances of a property at different levels and tell if the two instances are equivalent.

Eoin O'Connor is Studying Computer Science at UCD School of Computer Science and Informatics

There already exists consistency checkers that allow users to check between property instances. The limitation of these tools is that they only support checking a set number of pre defined properties. The only way to offer support for more properties is to release a new version of the checker. This means that the properties that are supported are chosen by the consistency checker developers. We feel that consistency checkers would be much more useful to programmers if they could easily add new properties to consistency checkers as they see fit. The aim of the this paper is to create a plug-in that can provide this functionality. By providing tool support for properties we aim to open the full power of the checking tools to a larger group of people and hopefully increase the amount of developers using existing consistency checking programs.

An example of what we would like to do is to add property support to an existing consistency checking program. An example of an existing consistency checker is Beetlz [1]. Beetlz is a consistency (refinement) checker for BON, the Business Object Notation (BON), and JML-annotated Java. With the property support that propi provides Beetlz could be expanded to allow a user to add new semantic properties that will be checked between BON and JML-annotated java. This is very useful when designers want to specify a property that is not already supported by Beetlz. For example a programmer could specify a property at the modeling BON level, and have it maintained at all other implementation levels of the project effortlessly. We aim to make it is easy to define new properties and add them to existing consistency checkers .

Our goal for the propi plug-in was to provide semantic property tool support. We wanted to present the tool support in such a way that it is easy to understand and use whilst also being flexible and powerful enough to support a large set of possible properties. We achieved this in two ways. Firstly, we developed a simple way for a user to define a new property. Properties are defined in simple text files. Secondly, we developed a clean api (application programming interface). The api allows a user to check if an input is a valid instance of the previously defined property and check consistency between instances. One of the crucial points for us was that a programmer new to the propi plug-in could quickly understand and implement their own property after reviewing a few examples.This is crucial because if the plug-in has a steep learning curve it will inevitably lead to a smaller uptake by developers and projects.

II. Background

A. Tools Used

To provide a simple way for a user to define a semantic property we needed a way to parse a simple text file.

We considered writing our own parser but because of time constraints we choose to use an existing data standard for which a parser already exists. After researching what tools were available we short listed XML [2] , Json [3]and YAML [4].

XML covers a large set of uses but is not very human readable. Despite being the more commonly used of the three choices we decided against its use. Poor readability would make our plug-in too difficult to use.

Json is very similar to YAML and is actually a subset of YAML. Json does not support a large set of native data types or use indentation for structure which would help with readability.

YAML has both a large set of native data types and uses indentation for format which make it the most human readable of the solutions we encountered in our research. For this reason coupled with previous experience using YAML in our development team we choose YAML. It provided all the functionality we needed while giving us the cleanest possible interface.

After deciding that YAML was the language we wanted to use we then had to choose a parser. The YAML home page listed a number of options: Jvyaml [5] jyaml [6] yamlbeans [7] and snakeyaml [8]. We choose snakeyaml as it had had the clearest documentation, an active user base and appeared to provide better support than the other options.

B. Tools Studied

We spent a number of days studying and experimenting with Beetlz [1]and BONc [9] . There were two reasons for studying these tools. Firstly, there was significant overlap between their functionality and what we hoped to achieve with propi. Secondly, by analysing these programs we got a good idea of the structure of the tools that we were aiming to support. Both of these steps helped us to design the structure of propi.

III. Implementation

A. The Process

In order to demonstrate what was involved in this project we have broken the work we did up into a number of tasks. Although we didn't necessarily complete each task in this order it makes demonstrating the work we did more straight forward.

A.1 Parsing

As discussed in section II we decided to use snakeyaml to cover the parsing overhead for our project. Snakeyaml is a highly powerful and customizable tool. We created two YAML parses for propi, one for the level definitions and another for the definitions of the refinements between levels.

The main challenge when developing the level parser was getting snakeyaml to recognize the supported objects D.2. After experimenting with snakeyaml we discovered that this was best achieved by writing a custom constructor, resolver and representer for the YAML parser. Although there was documentation for doing this [10] it took considerable effort and time as some of the documentation was misleading or in some cases missing altogether. Our difficulties were compounded by the fact that debugging was made difficult by the structure of snakeyaml. Despite this we managed to write the custom constructor, resolver and representers for the YAML parser. By recognizing and casting the supported objects at this point we made the checking phase **??** easier.

Designing the parser for refinements was far simpler. It needed only one custom type and we had gained much experience from writing the custom constructor, resolver and representer for the level parser. As a result this task was relatively straight forward. The only custom object we used was a enumerator that allowed us to define our transitions, to be used later when checking refinements. The structure we used makes it possibly to add new transitions in the future in a straight forward manor although this is not exposed to the end user.

A.2 Checking

As discussed earlier the YAML parser did do some checking but additional methods were necessary to check for unusual circumstances eg: empty or duplicated inputs.Checking methods were written for both levels are refinements . The methods basically involved iterating through the Maps generated by snakeyaml and checking the values against regular expressions. In the process the methods also construct the levels and refinements for the property.

A.3 Structure

When designing the structure we had to balance the most efficient implementation while also being able to provide the user with an intuitive, simple api. We gave careful consideration to ensue that the full complexity if the program is not exposed to the end user while simultaneously offering enough flexibility to create the largest set of possible properties. Of all the phases we spent the most time designing the structure. We designed the structure about a third of the way through the project, right after the original research and implementation phase was over. In the end we had a structure that exposed a clean and relatively simple public api package to the end user.

A.4 Refinement Methods

IsValidRefinement() and generate() methods were written. The first checks the equivalence of two instances and the second can generate form one instance to an equivalent instance at another level. We found that the structure we had chosen greatly reduced our workload here. However, despite our planning we did have to change some small elements of the stucture to facilitate the writing of efficient refinement methods. The use of these refinement methods are described in the api section C.

A.5 Unit Testing & Documentation

In this project we aimed to produced tests alongside the code for test driven development. We currently have a code coverage of about 83% but aim to improve this the 93-97% region. We also tried to keep the project up to the highest standard of documenting using the check style tool and the sun standards for java doc.

B. Design Decisions

When designing the plug-in we had to make a number of crucial decisions. The majority of these decisions were made in the structure phase **??**. The result of these decisions was that we decided to have a simple dsl to describe new properties and an small api to create and compare instances.

C. API Design

The api package has 4 public classes levelId, scopeId, SemanticPropertyInstance and SemanticPropertyHandler. The first two are just enumerators that represent the allowable levels and scopes for any property. These classes are well documented and new scopes and levels can easily be added. The SemanticPropertyHandler is the main class the user will use. First , the user must create an instance of the SemanticPropertyHandler. Next, we can add semantic properties to the handler with the add(File file) method. The file must be a correctly formatted YAML file as talked about in section A . When we have added all the semantic properties we want we can then create SemanticPropertyInstances for an input string. These are the inputs that we are trying to match from the the various languages. The instances can then be checked against other instances or refined to another instance if this is allowed by the Semantic property. The methods used to achieve this are parse(), ValidRefinement () and generate() respectively.

D. Domain Specific Language (DSL) Design

For the api design the goal was to make a simple domain specific language (dsl) that allowed users to easily define new properties.

D.1 File Structure Overview

First we will go over the basic structure. We define a property in a text file saved with the .yaml extension. For a property we can define levels and refinements. A level is a representation of a property for a certain language. This language levels in our concurrency example are java and BON. The allowed level languages are defined in LevelId C in the public api . A refinement allows us to define how one property level instance refines to another. When defining a semantic property in a file each new level or refinement must start with — and end with ... . Comment lines begin with #. The order in which you define levels and refinements is not important eg:

```
---
#a level can go here
...
---
#refinement between levels can go here
...
---
#another level can go here
...
```

D.2 Levels

Levels must include the following options:

name # The name of the property that this level belongs to.

level # The LevelId of this level.

scope # The ScopeId for this level.

format # A definition of the format that property instances can take at this level.

description # A brief informal description of the level. Does not have any special meaning.

Of these options the most complex is format. The format for a level is the form that all property instances for that level will be matched against. A format is effectively a small, limited regular expression. We are allowed use the following keywords to define the format:

choice: # The input must match one of the list items.

optional: # An option may or may not match the list items.

An simple example of these options in use is the following property definition example :

```
---
name: example3
level: bon
format:
  - choice:
    - a
    - b
    - c
  - optional: d
scope:
  - Module
description: |
 Describes
...
```

Acceptable inputs: could be any of the following "example3 a", "example3 b", "example3 c", "example3 a d", "example3 b d", or "example3 c d". This is an simple example but we feel that this format allows for a large set of potential properties.

When defining the format we can have meaningless words that get matched and objects. The objects allow us to capture meaning. That is to say they have a type, name and value. The objects that are currently supported are nat, int, float, string, expression, description, email, date, version, URL, throwable and class.

D.3 Refinements

Refinements have the following options:

property #The semantic property name.

relation(level1,level2) # Defines the refinement from property level1 to level2.

Refinements that are allowed for the different object types can be any of the following.

For string, description, expression : equals, prefix, suffix, or substring

For int, float & nat: = <= >= < >

For class, throwable, email, date, version, URL : equals or not equals

For optionals: <- -> =

IV. Case Study

A. Propi Use

The best way to understand how propi works is with an case study.

B. Semantic Property Domain Specific Language(DSL) Use

Here is concurrency.yaml. This file represents the concurrency example as specified on the kind website [11]

```
---
---
name: concurrency
level: bon
format:
 - choice:
  - <c1=SEQUENTIAL>
  - <c1=GUARDED>
  - <c1=CONCURRENT>
  - [<c1=TIMEOUT>, <to=nat=0>,
      <e=throwable=java.lang.EXCEPTION>]
  - [<c1=FAILURE>, <f=throwable=java.lang.ERROR>]
  - <c1=SPECIAL>
 - optional: <description=string=''>
scope:
 - Module
 - Feature
description: |
 Describes
...
---
name: concurrency
level: java
format:
 - choice:
  - <c1=SEQ>
  - <c1=GUARD>
  - <c1=PARALLEL>
  - [<c1=TO>, <to=nat=0>,
    <e=throwable=java.lang.Exception>]
  - [<c1=FAILURE>, <f=throwable=java.lang.Error>]
  - <c1=SPECIAL>
 - optional: <description=string='No information.'>
scope:
 - Module
 - Feature
description: |
 Describes
...
---
property: concurrency
relation(bon,java):
 - keyword:
  - c1: c1
  - SEQUENTIAL: SEQ
  - CONCURRENT: PARALLEL
  - GUARDED: GUARD
  - TIMEOUT: TO
  - FAILURE: FAIL
  - SPECIAL: SPECIAL
 - description: prefix
 - n: ">="
 - e: equals
 - f: equivalent
 - owner: ->
```

C. API Use

In this section we will demonstrate how to use the api.

First we create a SemanticPropertyHandler and add the concurrecy.yaml file from above. This will add the concurrency property to the handler.

```
SemanticPropertiesHandler testHandler =
 new SemanticPropertiesHandler();

File conYaml =
 new File("concurrency.yaml");

testHandler.add(conYaml);
```

Next we create an instance for the bon level. This method takes a number of arguments: string input for this instance, the level for the instance and the scope for the instance. This particular call will produce a SemanticPropertyInstance for the bon level.

```
SemanticPropertyInstance bon =
 testHandler.parse("concurrency TIMEOUT 5 java.
     lang.throwable", LevelId.BON_FORMAL, ScopeId
     .Module);
```

Similar to above this will create an instance for the java level. It will also be an instance of the concurrency property with the same scopeId.

```
SemanticPropertyInstance java =
 testHandler.parse( "concurrency TO 4 java.lang.
     throwable", LevelId.JAVA_JML, ScopeId.Module
     );
```

This method allows us to check if one instance is equivalent to another instance. It this case it will return true as the bon instance is equivalent to the java instance as defined in the concurrency property file above.

```
Boolean test = testHandler.isValidRefinement(bon,
     java);
```

If we have an instance for one level and want to generate an equivalent instance for another level we can use the method below. If a definition for this refinement exists it will return a SemanticPropertyInstance. Otherwise it will throw an exception.

```
SemanticPropertyInstance refinedInstance =
    testHandler.isValidRefinement(bon,
    refinedInstance);
```

V. Future Work

A. Time Constraints

Although we are confident that we support the major object types developers will want to use for the sake of completeness we would like to add support for more object types and more complex refinements. If time permitted we would also like to get the test coverage percentage up and write a custom parser as snakeyaml has some limitations as discussed in B.

B. Tool Integration

The potential and success of this tool can not be realized and assessed until tools start to use the propi plug-in. Many traditional consistency checkers could be made extendable by incorporating propi. Propi is not limited to only consistency checkers as talked about in this paper.We

could build on the concurrency example in this paper so that it links up to concurrency checking programs and asses if each instance is as defined.. This kind of automation would be or great benefit to the programming community. Many programmers deal with concurrency issues daily and to insure consistency between all levels of a project for concurrency would reduce debugging time significantly.

If propi is to have any future it must ensure that it integrates effortlessly with programs that would use it. For this reason we plan to integrate this tool with existing tools as a test to see if the api and dsl we designed are easy to use.

VI. Conclusion

We feel that we have achieved our goal as best we could in the time provided.We found some practices worked very well while others could be improved.

A. What Worked Well

For this project we used test driven development and this worked very well. It allowed us to gauge the level of completion at each stage. We wrote tests for bugs as we encountered them and this insured that they didn't resurface as the project progressed.

In the later half of the project we had a number of sessions where we sketched out skeleton code for the public api. This was very useful and allowed us to properly separate the public api from the private implementation. In fact we will do this earlier in future projects.

B. What Didn't Work Well

We spent too long reading and researching at the beginning of the project. It would have been better to start experimenting with code earlier.Also, we did not separate our experimentation and implementation stages well. Our experimentation code just began to become our final implementation. In future projects after a period of experimentation we will start from scratch with a much more structured approach.

We used snakeyaml for a combination of parsing and checking. In hindsight this was a mistake. We should have allowed the parsing to be handled by snakeyaml and written our own code for the entire checking phase.

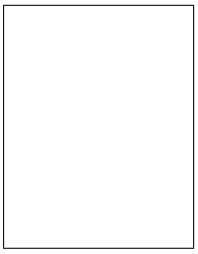
C. Final Thoughts

We set out to bring semantic property tool support to existing tools. We achieved this and are currently in the final phase of testing the project for completeness. We are excited to see how this tool will integrate with other checking tools and are confident it should be relatively straight forward to use.

VII. Acknowledgements

I would like to thank Fintan Fairmichael and Joe Kiniry for their constant support, help and guidance during my internship and when writing of this paper.

VIII. Biography



Eoin O'Connor was born in London, Ontario in 1988. He currently Lives in Dublin, Ireland. He is 22 and about to enter into his final year of Computer Science in University College Dublin.

References

[1] K. Website, "http://secure.ucd.ie/products/opensource/beetlz/."
[2] L. Quin, "http://www.w3.org/xml/."
[3] Unknown, "http://www.json.org/index.html."
[4] C. C. Evans, "Yaml definition."
[5] O. Bini, "https://jvyaml.dev.java.net/."
[6] Unknown, "http://jyaml.sourceforge.net/."
[7] N. Sweet, "http://code.google.com/p/yamlbeans/."
[8] Unknown, "http://code.google.com/p/snakeyaml/wiki/readme."
[9] F. Fairmichael, "http://secure.ucd.ie/products/opensource/bonc/."
[10] Unknown, "http://code.google.com/p/snakeyaml/wiki/documentation."
[11] J. Kiniry, "http://secure.ucd.ie/documents/whitepapers/code_standards/properties.html."