Verification condition generator for escjava2

Clément Hurlin
clement.hurlin@gmail.com

October 16^{th} 2005

Abstract

If you want to have a resume, read conclusion first

This document describes the operation of the verification condition generator of escjava2. First we describe how we use the old code, that was designed to produce an unsorted proof (for Simplify), to create a new ast tree, strongly typed, that can be easily used to produce proof for any prover. Design choices are explained, structure of classes and every tricky part of the code is supposed to be detailed here. If you want to modify/continue[1] the work on this new verification condition generator, this isthe more complete (and still absolutely uncomprehensible, uncomplete and full of english mistakes) and unique reference ... Don't hesitate to contact me for further explanations.

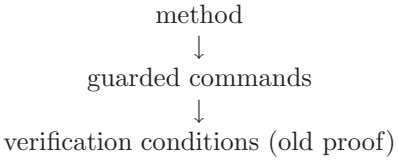
Despite this, this paper intends to show you problems related to implementation of proof generator, ideas used to do it for multiple provers and logics. Thus I hope that it will be done in a much much nicer way in the future because of one or two suggestions coming from this document ...

[1] If you're reading that for it, make sure to read the entire document, it's full of explanations, maybe not well organized, but as exhaustive as I can.

Contents

1 An old tree

First let's explain the global design of escjava2. Each method analysed leads to the generation of a set of guarded commands. The guarded commands represents how the method is supposed to be executed. You can see them by adding the *-pgc* to the *escj* script. The guarded commands are then modified, in a stage called 'desugaring' by the original authors of escjava. The generation of verification conditions[2] takes place after this phase, you can see it by adding the *-pvc* flag to the escjava executable. Originally, the logic of Simplify was completly untyped[3]. The new vc generator use the old proof and type it in order to be able to generate proof for many sorted logics. As the old proof system was fakely typed with the 'as trick', I did the assumption that we can build a strongly typed proof from it (see [**?**]). At the moment, the only new logic supported is the many sorted one for pvs. We can summarize the process this way :



So how the generation is now done ? Given the old tree of the unsorted proof, we generate a tree which is strongly typed, and where you can use dynamic call of functions to easily modify the output of the vc generator. What we call the old tree is the result of parsing the desugared vc, and creating a proof which is independant from any prover. This part of the code was written by the author of escjava, it was designed for Simplify, yet the new vc generator corrects that. Moreover every single node of the new proof ast inherits from the same class. The tricky part of the translation was to extract the information we need in the old tree (like for example, name of variable, sense of the node) and convert it to a lighter, more meaningfull tree that can be easily manipulated no matter what proof we want to generate.

2 Structure of classes

The entire vc generator can be divided in 3 different and independant set of classes. The only method you're supposed to call are in the *VcGenerator*. It was designed as an interface to manipulate the tree of the new proof. All files that deals with the tree are prefixed with a 'T'. The last set of classes is compound of *TypeInfo* and *VariableInfo*. Like you can guess from their name these two classes are used to contain information variables and about types, it will be explained more in details bottom.

2.1 VcGenerator

Publics method of this class are :

- The constructor VcGenerator(ASTNode). The constructor doest nothing except storing the node you give as a parameter. This node is supposed to be the root node of the old proof tree. Then you can call any of the following method :
- generateIfpProof(ASTNode n, boolean dot). This method creates the new tree. The second parameter is just used for outputting the dot representation of the old tree. If you give *true* then the dot file will be generated. You will then be able to retrieve it by calling oldVc2Dot().
- setOutputProof(String). This method is used for setting the type of the proof you want to be generated. At the moment you can give :

[2] verification condition will be abreviated by 'vc' by now
[3] or has only one type depending on how you see it

```
AND(
   (EQ c-1.8 0)
   (EQ |TYPE:9..| T_int)
   (EQ |elems<1>|
     (store elems |tmp!int[]:9.22|
       (store
         (select elems |tmp!int[]:9.22|)
         0 |RES:13.8|)
       )
    )
```

Figure 1: Simplify's proof example

key	unsortedPvs	pvs	sammy	type
c-1.8	c_1_8	c_1_8	c1.8	$integer[4]
|TYPE:9.. |	TYPE_9__	TYPE_9__	TYPE_9__	$Type
|tmp!int[]:9.22|	tmp_int_9_22	tmp_int_9_22	tmp_int_9_22	$arrayReference

Figure 2: map example of variables names storing

- "unsortedPvs" : the vc will be created for the logic which is in ESCTools/docs/Escjava2-Logics/unsorted/current_unsorted_logic.pvs
- "pvs" : the vc are generated for the many sorted logic of pvs (see [**?**]) located in ESCTools/docs/Escjava2-Logics/many-sorted/many_sorted_logic.pvs
- "sammy" : this time it's for the smt many sorted logic, which will be used with smt based prover, like Sammy (see [**?**]).

2.2 The top abstract classes

2.2.1 Static objects belonging to all nodes

The new ast tree makes strong use of abstract classes and overloading of fonctions. The top class from which all node inherits is *TNode*. It contains as well all static fields and functions. I first explain the operation of these static field because it's quite independant from the tree itself. There is a field containing a map describing how renaming of variable is done. This map is called *variablesName*. Let's explain how it is encoded with a little example. Imagine you have a proof for Simplify that looks like :

You can notice that :

- There is one variable, (surely a integer, double or float) named *c-1.8*.
- There is one variable which is a type : *|TYPE:9..|*. You can guess it because it's compared with *T_int* which is a type too. This is important to notice that type have a type (integer is a *FinalType*, a reference is a *ReferenceType*).
- Another variable is *|tmp0!int[]:9.22|* (this is a reference to an array).

Seeing that, the *variablesName* hashmap will look like figure **??** page **??**.

So *variablesName* is a java *HashMap* where key are string which represents name of variable in the old translation of vc. Object stored belongs to class *VariableInfo*(see related section page **??**). So when we encounter a variable name *c-1.8*, if *c-1.8* isn't a key of the *variablesName* field, we entered it as a key in the hashMap, and the associated object is a new *VariableInfo* containing the different renaming of the variable (field *unsortedPvs pvs* and *sammy*). The last field of *VariableInfo* objects is a pointer on a *TypeInfo* object which contains the type of the variable (if known). In order not to waste memory, renaming is done only when we know what kind of proof we will have to generate. The *TypeInfo* class works in the same way as *VariableInfo* in a *HashMap* called *typesName* :

key	unsortedPvs	pvs	sammy
%Reference	S	ReferenceObject	?
%Type	S	ReferenceType	?
integer	S	DiscreteNumber	int
|TYPE:9.. |	S	TYPE_9__	TYPE_9__

Figure 3: map example of types names storing

Why is it done this way ?

1. Having one map to store all variable information is very similar to compilation procedure. So I think this is very natural to handle it this way.
2. Putting it as static field of *TNode* is maybe not very oo oriented, yet it's simple.
3. It allow to have one map not depending of the type of the proof. Moreover renaming is done on the fly, ie only when the type (unsortedPvs, Pvs or Sammy) of proof asked is known.
4. The unicity of old name (the leftest column in our arrays) is guaranteed by the old proof system, thus it should not raise problems.

How to improve it ?

- Looking at the code of *TNode* you can notice that some initialization (call to *init()*) are done as many times as method check, this can be corrected to be reduced to one.
- Think about handling all possible types, and how far we can go. For example I'm quite sure we can fully support arrays, but I'm not sure it will be very easy to find types of maps (associating object to field for example).
- **This is not robust and absolutely experimental**. This is the key point. To continue further this work and think about adding capabilities to escjava2, you should test it against a lot, a lot of tests cases. Be sure to ask Joe to explain what means 'intense' testing...

About types (see figure **??** page **??**)

Like for *variablesName*, keys in typesName (the second array of the exemple) represents the name of the type in the old tree. As the old tree wasn't typed, it was defined during the rewriting of the vc generator. Column numbered higher than one refers to corresponding name of the type in the different logics. For example the first line means that the type 'integer' is equivalent to the type 'int' in the pvs unsorted logic and in the smt lib one, and that it's translated to 'DiscreteNumber' for the pvs many sorted logic. We add a line in this array (it means we create a new *TypeInfo*(see related section page **??**) object) when we encounter a new type. Of course, we don't add the same type two times (this is easily done because key must be unique). For example the type encountered in the old proof called |*TYPE:9..* | is catched by the last line. A question immeditaley raises : what are the types in the old tree ?

We tried to use every information available in the old tree, in order not to be constraint when we generate proofs. At the moment, different types are :

- %Reference
- %Time
- %Type
- boolean
- char

- double
- %Field
- float
- %INTTYPE
- integer
- %Path

As keys in the *typesName HashMap* are String, these types are stored as String. This explains why some of them are prefixed with a %. % is a character which cannot be used in Java for naming a variable or a type. That's why we used to prefix some of our types. Imagine that the user defined a class "Time", then we will try to add to this map with the key "Type". That's why we have to differentiate our type "Time" by prefixing it with a % which we know can't be use as a legal identifier in Java.
As you can guess, when we encounter a type name in the old tree, we check if it's already present in the *typesName* map. If not we add it. That's why, in order to be a little bit efficient, some types are added before any call can be made. There is a function initializing type table in the *TNode* class :

```
    static public void init (){
        typesName = new HashMap();
        variablesName = new HashMap();

        // Predefined types

        $Reference = addType("%Reference", "S", "Reference", "?");
        $Time = addType("%Time", "S", "Time", "?");
        $Type = addType("%Type", "S", "ReferenceType", "?");
        $boolean = addType("boolean", "S", "Boolean", "?");
        $char = addType("char", "S", "T_char", "?");

        ...
}
```

It means that the new object *TypeInfo* is stored in the field *$Reference* and added to the *typesName* during initialization of the class. As these object are stored as *public static field* in the *TNode* class, we can easily use it to set type of variables. Moreover when we want to add a variable which has type '%Reference' we can set its type without having to call a function which check if this type is already in the map. You can just do *this.type = TNode.$Reference* which is a valid call because *$Reference* is a static field of *TNode*. I can imagine that you may wonder why variables name begins with a $, I don't know why too, it just makes easier to remember that they are static. So as a resume, *TNode* class have fields prefixed with a $, that store the types, and some of these types begins with a % to differentiate them from user defined types.

2.2.2 TNode

In this section, I describe features that are more related to the fact that TNode is the super class for all node (ie not talking about static/global variables). This class contains a lot of functions and fields. The different fields are :

Fields

- static protected int id : This field is an identifier which allows to distinguish every single node. It's used when we create the dot graph (see below).

- static protected int counter : This field is used to create different id for each node. It's incremented at each call to the *TNode* contructor.
- protected boolean isroot : This flag indicates if the node is the root one. His utility can be discusssed...
- public TypeInfo type : This field points vers the *TypeInfo* representing the type of this node. It can be null as long as the tree hasn't been typed.
- protected boolean typeSure : When we type the tree, sometimes we can't be sure about the type (for example when inspecting an *AnyEQ* function) of a variable. As long as this boolean isn't set to true, the type can change if we find more informations in another part of the tree.
- static protected int lastType : this field store the type of the proof you last asked for. This is used to launch appropriate proof simplifier visitor for example (see further).
- static protected boolean typeProofSet : this is used in specifications to make sure you have set the proof before calling type-of-proof asked function.
- String label : this will store the name of the current proof generated (like method_X_file_Y for example).

Functions [5]

- public TNode() : this is the default constructor for all node in the proof, it's just increase the counter and the id for the node being created.
- static public void init() : This functions reinitializes *variablesName* and *typesName* before beginning a new proof.
- public void setOutputType(String type) : this is used to set the type of the proof.
- protected void generateDeclarations(StringBuffer s) : As now variables are typed, we should declare it in new proofs. For the example of proof (see figure **??** page **??**), the declaration for the pvs many sorted logic will consist of (I skip the array variable as it's not yet implemented) :

```
c_1_8 : DiscreteNumber
TYPE_9__ : ReferenceType
```

- static public VariableInfo addName(String oldName, String type, ...) : this function is called when the user discover a new variable and want to add it to the variables map. Imagine (still refering to Simplify's proof example of figure **??**) that this is the first you encounter *c-1.8*, you will make a call looking like TNode.addName("c-1.8", "integer"). This call handle adding variable (if not already present in the map) to the map and adding its type to the types map (same notice).
- static VariableInfo(String name) : this function allows to retrieve the *VariableInfo* object associated to this name.
- static public TypeInfo addType(String oldType, ...) : Identical to variable operation, if we encounter a type in the proof, we will add it by calling this function. Imagine we are checking the constructor of a user defined class *A*. We will then do the following call :

```
TName n = new TName();
TNode.addType("A");
n.setType("A");
```

[5] By looking at the code you can notice that I skip some (often overloaded one) function whose sense is trivial

- abstract protected void typeTree() : this function is not defined here because his behaviour is different in all direct subclasses. It is used when you (desperatly) try to type all node of the proof, see section **?? ??**.
- protected void setType(TypeInfo type, boolean sure) : this function is used to set the type of a node. For example, if we are in a node representing the addition of 2 *integers*, we can deduce that all sons should have type integer too. Thus, in such a case, you will write code looking like :
- public String getType() : this function return the type of the node according to the type of proof asked. Ie if the type of the proof is *pvs*. The renaming function used will be the one for pvs.
- public TypeInfo getTypeInfo() : If the current node has a type, ie if it contains a *TypeInfo* object, it will be returned.
- static public void printInfo() : This function is called when you pass the *-pInfo* flag to ESC/Java2. It displays the content of *variablesName* and *typesName* map on standard output.

2.2.3 TFunction

This class, which inherits directly from *TNode*, is used to represent nodes corresponding to function, ie nodes that have sons. That's why it contains a *vector* of *TNode*.

That's why it contains a few additional functions related to son adding/getting :

- void addSon(TNode n)
- void getChildAt(index i) : Notice that it does not throw an exception if you ask for a too big index. It just displays a message.
- protected void typeTree() : This method does nothing except calling it on all sons. As *TFunction* is abstract, ie has no implementation, we have no information about type here.

Classes extending TFunction

A lot of classes are here just to factorize behaviour of function depending on what type they are related to. Let me explain. For example *TBoolRes* extends *TFunction* and represents node returning a boolean (*Boolean Result*). That's why you can see in his constructor :

```
public TBoolRes(){
    type = $boolean;
}
```

Notice that sometimes we are speaking of type of node, sometimes type of variable. In fact, we consider (and this is very natural) that every node has a type. If a node is a variable, it has the type of the variable. If a node is a function, it has the type the functions returns. Thus you can easily typecheck the whole tree.

All abstract classes are in different files (TBoolRes.java, TIntOp.java etc ...). Implementation of classes are all in a central file (TFunction**.j**). For example *TIntegralAdd* which modelizes + (the natural 3 + 5 operator), which inherits from TIntOp (integer * integer → boolean[6]) is in this file *TFunction.j*. In order to split files (thus generate classes) you can launch the perl script *division-j-file.pl*[7]

[6] Name of classes finishing by *Op* indicates that these class returns a boolean

[7] Thanks to Patrice Chalin for this script) on .j files. At the moment there is two .j files, TFunction.j and TLiteral.j. **Important**, note that if you want to modify code of automaticaly generated class, you should modify the .j file, then clean the folder (launching *clean.sh*) and launch *division-j-file TFunction.j* for example. I will detail everything later (see section **?? ??**).

2.2.4 TVariable

First consider that this class means variable and constant. Variables are represented by class *TName*. Constants are represented by class *TLiteral.java*.

constants TLiteral inherits direclty from TVariable and add nothing to it (this is stupid). Above TLiteral you have all classes defined in TLiteral**.j** (again you have to split that file's with Patrice Chalin's script division-j-file). These classes encode differents constants that can appear in proof like :

```
(EQ
   (c-1.8 0)
   (s-2.5 "do not read that")
)
```

In that little proof, you have the constant '0' which an integer (coded by *TInt*) and the constant '"do not read that"' which is a string (coded by *TString*). The advantage of creating as class as constant type is that you can know their type easily.

variables are encoded by the class *TName*. They have an additional field of type *String* where we store their ...name. This name is the same as the one serving as a key in the global map *variablesName*. Notice that if we have a proof looking like :

```
(EQ
   (c-1.8 0)
   (c-1.8 d-5.22)
)
```

We will have two *TName* objects created for variable *c-1.8*. Moreover when we create a *TName* object, we add this name immediately to the global map. It means that every code dealing with TName creation is compound of 2 lines :

```
TName n = new TName("c-1.8");
TNode.addName("c-1.8", "integer");
```

As before adding we check if it's not already there, there will be only one entry corresponding to *c-1.8* in the map. It allows us to be sure the renaming will be always be consistant with the old name (because each *TName* storing *c-1.8* will look at the same entry of the map to get their renaming).

2.2.5 VariableInfo

As discussed and explained before (see figure **??** page **??**), information about variables are stocked in object of type *VariableInfo*. This class contains field for each possible renaming. We can imagine that for each proof asked, the renaming will be done in a different way. When the variable is created, we only fill the old name and wait to see what type of proof will be asked.
That's why you have as many fields as many methods to rename them :

- unsortedPvs → unsortedPvsRename()
- pvs → pvsRename()
- sammy → sammyRename()

At the moment, the only renaming that is partially implemented is the one for the pvs sorted logic. It uses Java regular expression. As all the renaming is done in one fonction for each type of proof, it's very flexible. If you want to change the way the renaming is done, you just have to change this function.
An important thing to consider is to avoid confusion between user defined type and our types. Imagine that the user defined the class 'Reference'. As we already use it in our pvs proof system,

we have to rename it. That's why user defined types are prefixed with 'userDef?'.
Note that as types are variables too, their renaming is done here too. There is a confusion between that and the renaming done in *TypeInfo* (see next paragraph), something has to be fixed here to avoid that.
There should be a reflexion about all this renaming problem before implementing anything because this is now very confuseful. This seems stupid, but between the convention of the old ESC/Java team, mine and yours, it can be hard not to make mistakes.

2.2.6 TypeInfo

This class is very similar to *VariableInfo*. It contains a *String* field for each renaming possible and a rename function for each type of proof too. As for the moment the renaming of types is done in *VariableInfo*, there is nothing more to say here.
Note that predefined types added before each proof are added with their renaming already done (in order to avoid adding conditionnal construction to renaming functions)(look at *init()* method in *TNode*).

2.2.7 TDisplay

This class was intended to centralize outputting of messages. Yet the right way to do the job is to use anonymous class and instantiating function depending on which flags were passed to ESC/Java2. It also use bash special characters to display colors, it can be done in a nicer and more flexible way.
Note that, each call to *TDisplay* function receives the object of the caller in order to indicate what classes we are in. As I didn't find how to display the name of the method currently called, I just passed it as second argument too. This is quite lame, I think this can be done in a nicer way.

3 Pratically speaking, how does it work ?

All source code is located in ESCTools/Escjava/java/escjava/vcGeneration/.

3.1 Flags you can pass to escjava2

- **-nvcg** : this flag enables all other feature. If you do not pass this one, you can't use the **n**ew **vc g**enerator.
- **-pPvs** : Generate **p**roof for the pvs sorted logic (located in ESCTools/Escjava2-Logics/pvs/escjava2.pvs).
- **-pSimplify** : Generate proof for simplify. This was done to compare that the proof system doesn't skip important node (for example we skip label, and handling of *forall* and *exist* isn't supported [8]) by comparing proof generated this way and proof generated by the old system (ie compare output of ./escj /javaFile.java -nvcg -pSimplify and ./escj /javaFile.java -pvc). From previous tests, it seems to work very well (no inconsistancy between both proofs generated (knowing the limitations of the new proof system)).
- **-pErr** : Display errors on error output.
- **-pWarn** : Display warnings on error output.
- **-pInfo** : Display additional messages on error output.
- **-pColors** : used bash special characters to display some colors (absolutely useless).

[8] I didn't make further test but these constructs have been added by David Cok to support *model* keyword but have not been implemented correctly (it uses tag not recognized by escjava2), you should contact him for further details.

- **-pToDot** : output the proof (before simplification) in dot format, and launch dot on it to generate a .ps file [9]. This is **very useful**. Dot files are postfixed with .proof.dot, ps files generated after are postfixed with .proof.dot.ps

- **-vc2Dot** : output the old proof in dot format and launch the appropriate dot command on it (look at Main.java). Fixme, I guess that if you've tried this flag, you've noticed it does not work. That's very easy to fix. As the constructor of ASTNode does nothing special, I need to add a counter to it (to identify each node). Yet the problem is that if I define a constructor in ESCTools/Javafe/java/javafe/ast/hierarchy.j the default constructor is still generated (which clashes with the new one), thus you can't compile. So you need to decomment the constructor in hierarchy.j, compile one time, fix ASTNode.java to comment the old constructor and that's it.

3.2 Generating new ast tree

Translation of the old tree is done in the *generateIfpTree* function in the VcGenerator class. So if you want to modify the way it's done, that's where you have to first look. The translation is similar with the code of the pretty printer for the old logic (ESCTools/Escjava/java/escjava/translate/VcToString.java). It takes non typed object and create a lot of node using inheritance from TFunction to build a new typed tree.
All you have to give to this function is the root node (here named *vcBody*) of the old proof (this is done in Main.java), something like :

```
VcGenerator vcg = new VcGenerator(vcBody, options().pErr,
                                  options().pWarn, options().pInfo,
                                  options().pColors);
```

It leads me to speak of this part. If you carefully read Main.java, you will notice that the root node given to the vc generator is not the same as the one used for the old generation process. In fact the old proof is modified one more time before beeing sent to Simplify. This piece of code looks like :

```
Expr vc = GC.implies(initState.getInitialState(), vcBody);

vc = LabelExpr.make(r.getStartLoc(), r.getEndLoc(),
                    false, Identifier.intern(label), vc);
```

This was done this way because in the old logic, there was a need for a few additional hypothesis for each proof. Ie the proof sent to Simplify was :

$$(\text{additional hypothesis}) \wedge (\text{proof current method})$$

I won't explain, but if you have a good knowledge of Simplify's logic, it will makes sense (you can read [**?**], that's explained), so the additional hypothesis are :

```
(AND
  (EQ |elems@pre| elems)
  (EQ elems
    (asElems elems)
    )
  (<
    (eClosedTime elems)
    alloc)
  (EQ LS
    (asLockSet LS)
    )
```

[9] dot is an open source software developed by graphwiz (AT&T laboratory), you can get it at *www.graphviz.org/* (for Debian users, there is a package *graphviz*

```
(EQ |alloc@pre| alloc)
(EQ |state@pre| state)
)
```

As it's no more useful in new logic, the vc generator use the old proof before adding these hypothesis to it.

The process of translating the old tree is quite crapy. It consists of big switch on tags, and/or conditional statements with *instanceof*. After identifying the type of the old node, I extract information we want by looking at appropriate data structures. This is the main source of errors I think and is not very robust. The rest of operations is much more cleaner. This is not surprising since manipulating the old tree wasn't handy (and that's why a new ast was designed).

3.3 Typing the tree

At this stage, we have a tree which is not completely typed. For example if we encounter a node *ANYEQ* (which stands for "any equality"), we can not have further information on what it represents.
Yet we can deduce a lot of things by visiting the tree. At the moment this is done by calling the function *typeTree* recursively on all nodes (starting from the root one). So imagine we have a proof looking like :

```
(AND
  (isA c-1.8 INTTYPE)
  (EQ c-1.8 d-2.24)
)
```

From the the second line we learn that *c-1.8* is an integer. Then, as the operator *EQ* is used for a lot of things, we can't guess nothing. Yet as *c-1.8* is an integer, and that we know *EQ* is applied on same types sons, we can deduce that *d-2.24* is an integer too.
Depending of the type of the class you're 'in', the function doing the typing (*typeTree*) acts differently. You can know understand the advantage of defining class strongly related to type (I mean defining *TBoolOp* for node returning a boolean, *TIntOp* for node taking *integers* and returning boolean).
To set the type of a node, you have to call the method *setType(TypeInfo, boolean)*. The boolean is here to indicate if you're sure what you're doing. Because of the old implementation sometimes you can't be sure of the type, thus you can pass false as second argument. Then if we have further information later in the proof we will change it. From first experiments, this happens rarely which means the old proof was well typed (which is a good sign and indicates this work is finishable [10]).

3.4 Simplication

Simplification of the proof consists of deleting we know are useless, or deleting operations we will know the result (like boolAnd(true, a, b) = boolAnd(a, b)). For the moment this is only partially done (...). This is done in the *TProofSimplifier* visitor class. For the moment, we only delete *TIs* node. This kind of node gives information about the type of variable. So this is useful to gather type, but as proof are now typed, we can delete these axioms (this information is now contained when declaratig variables). Imagine you have an old Simplify proof looking like :

```
(AND
   (isA
     (c-1.8, INTTYPE)
   )
   (EQ
      (c-1.8 0)
   )
)
```

[10] does it match any english word ?

This will be translated for the pvs sorted logic by :

```
Forall(c-1.8 : DiscreteNumber) :
c-1.8 = 0
```

This operation is non trivial since you have to check that when you delete a node you don't have to delete their parents too. Imagine you have deleted the 'isA' node in the previous proof. What remains is :

```
(AND
   (EQ
      (c-1.8 0)
   )
)
```

It doesn't make sense to have the boolean operator *AND* with only one argument. That's why you can simplify the proof to delete this (now) useless *AND* node and replace it with his son *EQ* (you can notice that the type of the result returned for the super parent of all this proof is still boolean which show that it's consistant.

current implementation only delets *isA* node and *ForAll* and *Exist* nodes (see previous footnote page **??** about that). Yet I think you should continue to modify functions of the *TProofSimplifier* visitor to handle more things.

3.5 Generating the proof

Again this is done by using visitor. There is two independant things. First there is the visitor creating the proof and there is the *StringBuffer class associated with it that does correct indentation[11].
The *TPvsVisitor* class which create proofs for the pvs sorted logic can be easily completed and/or reused for another implementation. There is a few generic function that most functions can call (depending of it's binary function, unary ones, you can call different generic function). Thus the process is very simple to modify. As a comparison, the previous proof generator for Simplify (class ESCTools/Escjava/java/escjava/translate/VcToString.java) was about 850 lines long, and absolutely tricky. Now that all nodes call the same function for displaying their subproof, this is 100 lines long and the pretty printer is 90 lines long.....
This is the main advantage of all this system. You can see that's is very flexible and each part, creation of the tree, simplification, proof generation can be modified independantly.

3.6 Creating graphical representation of proof

As I said before you can add flags *-Vc2Dot* to generate graphic display of the old proof or *-pToDot* to see the new proof tree. Note that the proof outputted is the one before simplification and that you can easily do it at any stage.
As you may have guess, I love to use examples so this part won't avoid it because I think the dot syntax is so simple that you can understand it in a second. So this is the classical aspect of a dot file :

```
digraph G {
BoolAnd140 [label = "BoolAnd\n\[boolean\]"];
BoolAnd140 -> Is141;
Is141 [label = "Is\n\[boolean\]"];
Is141 -> Name142 [color = red];
Name142 [shape=box, label="\[integer\]\n|x:8.25|"];
Is141 -> Name143 [color = red];
```

[11] In order to be more efficient, that will be better not to do indentation and stuff when generating proof for non human reader

```
Name143 [shape=box, label="\[%Type\]\nINTTYPE"];
BoolAnd140 -> Is144;
...
}
```

For further information you can go on *www.graphviz.org*. I can't tell you more than continuing using that, this is very useful to discover errors, see the form of the proof and much more.
Important : see section **??** to see why it does not work to launch -Vc2Dot and how you can fix it in a sec.

4 How to finish the job/generate vcs for another logic/modify the generator ?

Even if it's atm not finished for a single logic, I can already tell what need to be done if you want to :

1. **finish the translation for the pvs sorted logic** : I think there is an example of everything (an example of how you can simplify proof, how renaming is done etc...). You just need to add support for arrays/cast and handle fields (ie pvs's map that simulate *store* function of the old logic). The last thing is the tricky one I think. You then need to verify that the support of type is correctly done (and I think not) because sometimes there can be a confusion between java.lang.Math and Math (I don't really know this is handled in the old representation of the proof and so don't know how to extract it correctly).

2. **generate vcs for another logic** : There is not a lot of things to change. You can add renaming functions to *VariableInfo* and *TypeInfo*. Then you can create a new visitor (and generate prototype of functions automaticaly, see section 'Various stuffs page **??**) that will go through the tree (already simplified, if you want not to do that compare function *simplifyProof* and *pvsProof* in the *VcGenerator* class).

3. **Modify the generator** : you can (and must atm) complete/modify the way the old tree is translated in the function *generateIfpTree* in *VcGenerator.java*. You can modify the way renaming is done in *VariableInfo* and *TypeInfo*.

5 Various stuffs/related files/scripts

clean.sh : this file deletes all classes generated by the division of .j files. It also delete backup files (including the ones generated by latex when editing this file). Note you can rebuild automaticaly the list of files to delete by running :

```
cat TFunction.j | grep class | awk '{print $2}' | sed 's/^T\(.*\)/rm\ T\1.java\ -f/g'
cat TLiteral.j | grep class | awk '{print $2}' | sed 's/^T\(.*\)/rm\ T\1.java\ -f/g'
```

division-j-file.pl : this script, which was written by Patrice Chalin divide .j files into different classes. I don't really know how it works, you should ask him.

generate-visitir-functions : this script generates function for visitors by looking at .j files.

6 Conclusion and possible improvements

This conclusion is divided into two parts. If you want to continue the generator, the first one is for you. If you want to see what I think of a long term project on that, read second section.

1/ Are you the person who have (been) chosen to continue the job ? Hi, we will surely exchange a couple of mails in the incoming days :O]. As I discussed about it with Joe, whatever you may do it won't wide spread around the world. Why ? Because it will be hard to make it very robust, the only thing you can do is to test it against a lot of examples and carefully handle new constructs (array, cast, field ...). Yet I think it's the first system to generate proof for pvs in an automatic way, which can be very interesting.
The first things to do are resumed in the how to section (see page **??**). Don't hesitate to send emails to insult me about the current implementation, I will be happy to help you. Remember that JML constructs have never been tested with jmlrac neither checked with jmlc recently because of the impossibility to check *ASTNode* (the only old class used which is located in ESC-Tools/Javafe/java/javafe/ast/ASTNode.java (but defined in the same directory into file hierarachy.j take care...). That will be very good to correct that too. Remember that there is surely a lot of crappy cases you have to think about.

2/ This job has highlighted a couple of things related to proof generations :

- This is non trivial engineering problem which requires a lot of work far superior to what was remaining in my internship.
- You have to think about of a lot of stupids problems like renaming of variables, clashes with logic variables etc...
- The whole process is very similar to building a compilator. There is (quasi) syntaxic analysis to do first, then generating the proof and simplifying it. The same algorithms/constructs can be reused (visitors, map storing variables name for example).
- You absolutely should separate stages in a very clear way. Thus you will be able to modify each one independantly. Once the process will be robust, think about making it fast but that's all... The example of the old ESC/Java code is a good example of great confusion where you have call to static functions located in other packages everywhere, which makes it very easy to reuse after...
- Build generic stuff in order to manipulate different provers or logics easily. The way the renaming is done is a good example where you just have to define a new function to define a new renaming style.
- Of course, the nicer way to do the job would be to completely rewrite ESC/Java2 to handle type from the beginning and have a good knowledge of every stages. That would be a lot more faster but that's a lot of work too.

A Appendix

The one and only example it is completely implemented for : (Of course I'm not speaking about the proof for default constructor but for the method 'f').

```
class A{

   public A(){};

   //@ requires y != 0;
   static public f(int x, int y){
     return x/y;
   }

}
```

List of non automaticaly generated classes used to modelize types The * indicates that there is no constraint on the number of sons. '?' indicates that the type is not fixed.

TBoolRes : ?* → boolean
TFloatOp : float* → boolean
TIntFun : integer* → integer
TIntOp : integer* → boolean
TRefOp : reference* → boolean
TTypeOp : type* → boolean

Non complete list of automaticaly generated classes used to modelize types (see TFunction.j) The * indicates that there is no constraint on the number of sons. '?' indicates that the type is not fixed.

TBoolImplies : boolean* → boolean
TBoolAnd : boolean* → boolean
TBoolOr : boolean* → boolean
TBoolNot : boolean* → boolean
TBoolEQ : boolean* → boolean
TAllocLT : %Time x %Time → boolean
TAllocLE : %Time x %Time → boolean
TAnyEQ : ? x ? → boolean
TIntegralEQ : integer x integer → boolean

References

[1] The logics and Calculi of ESC/Java2. Original version by K. Rustan, M. Leino and Jim Saxe, 1997.

[2] Sammy is an SMT checker developped by Michæl DeCoser, George Hagen, Cesare Tinelli and Hantao Zhang.

[3] A Tutorial Introduction to PVS, WIFT '95. Judy Crow, Sam Owre, John Rushby, Natarajan Shakar, Mandayam Srivas.