Implementing Witness Annotations for VDMJ with Focus on Helping Prove an Existing Payment Protocol Model

Edward Jacobs

School of Computing Science, Newcastle University, UK

Abstract

Formal modelling provides a mathematical approach to help with the specification, development and verification of computer systems. This process is increasingly important in systems with a potentially big impact to users. These models allow us to produce formal proofs to show the correctness of a system. One area which benefits from formal modelling is payment protocols. EMV represents a significant portion of the worlds smart card transactions, however, there exists multiple flaws in the EMV specifications. One such flaw is the vulnerability to relay attacks, where consumers can become a victim of unauthorised transactions. In order to address this issues EMVCo have introduced a Relay Resistance Protocol. This protocol has been successfully modelled in VDM in previous work [2]. VDM is one of the longest standing formal modelling methods. It has grown to include many tools based on its specification language. VDMJ is a Java based extension of the VDM specification language and provides many useful tools, such as user definable annotations. The project aims to explore the process of implementing of a new annotation in VDMJ to assist with the implementation of Witness Annotations, based on the witness proof pattern [11], with the aim of assisting the formal proof of the Relay Resistance Protocol.

Keywords: Formal Modelling, VDMJ, EMV, Relay Resistance Protocol, Witness Annotations, Proof Obligations

1 Introduction

1.1 Motivation

Formal methods are a kind of mathematically based technique used to help with the specification, development and verification of computer systems. These methods allow the modelling of potentially complex systems as mathematical entities, which allows the ability to formally verify the properties of a specified system [12]. This helps to ensure their reliability and robustness, which is particularly advantageous when dealing with systems where security or safety is a critical factor. This becomes increasingly important as they become more powerful and impactful to society.

One area which benefits largely from formal modelling is payment system protocols. Payment protocols exist to help ensure the security, reliability and robustness of the underlying payment systems. EMV [3], produced and maintained by EMVCo, is one of the

[1] Email: E.Jacobs1@newcastle.ac.uk

most common payment types in the world and is the technical standard for many payment cards. This standard not only covers the payment cards, but also the Point of Service (PoS) terminals and Automated Teller Machines (ATMs) which can accept them. However, with the massive use of the EMV standard, adversaries have been able to uncover and abuse vulnerabilities in the specification. One such vulnerability is the relay attack, an attack where two conspiring adversaries can target both a legitimate customer and merchant at the same time. This attack can have a big impact on the victims, with a potentially large amount of money being fraudulently spent. This vulnerability is known by EMVCo and has such a Relay Resistance Protocol has been added to one of the more recent EMV specifications [6]. This protocol aims to secure the vulnerability of the relay attack. A previous student, Jack Maiden, was able to succesfully model the Relay Resistance Protocol [2] using VDM.

VDM (Vienna Development Method) [15,16] is one of the longest standing formal modelling methods available today. It has grown over the years to include many techniques and tools, based on its formal specification languages. One such tool is VDMJ, it "provides basic tool support for the VDM-SL, VDM++ and VDM-RT specification languages, written in Java." [1]. One of the features that VDMJ provides is user definable annotations. These annotations represent a form of metadata that can be added to the specification as comments, used to affect the compiler without directly altering the behaviour of the specification at runtime [8].

VDM specifications are built in terms of models. These models consist of a range of data and functionality, that defines the underlying system. The key activities behind modelling systems in VDM is to both simulate a systems behaviour, allowing it to be tested under various factors, and the ability to generate formal proofs. Proofs are defined in VDM by a number of proof obligations, which are unproven mathematical theorems that, when proven, ensure the correctness of a specification [11]. VDMJ provides a proof obligation generator which automatically generates proof obligations for a given model [17]. Larsen et al. [13], detail how, ''Proof obligations are generated for a range of properties relating to the internal consistency of a model. These include obligations checking for potential run-time errors caused by misapplication of partial operators, consistency of results with postconditions and termination of recursion".

The key proofs for this project are proofs of initialisation and proofs of feasibility. Proofs of initialisation detail how there exists a value, such that the invariant on that value is satisfied. Proofs of feasibility require that for all valid starting states and inputs there exists a valid updated state and result [14]. One of the key impediments when dealing with proofs is the process of existential quantification. The existential quantifier is used in predicate logic to express that the statements it refers to are true for at least one instance. The difficulty with existential quantification can be generating appropriate values that satisfy the proof. In their paper titled "Proof Patterns for Formal Methods" [11], L. Freitas and I. Whiteside introduce the Witness pattern, which when successful can reduce proof obligations allowing them to be more easily proven. They state how, "One of the most important steps to solve feasibility (and reification) POs is finding appropriate witnesses for the outputs and updated state.". They also discuss how "In general, finding a witness is a difficult task". In the paper they go on to suggest two common patterns, to assist with the discharging of some simpler existential proofs, and a third pattern where the proof engineer provides their own arbitrary witness. In a later paper [14], Freitas et al. mention existential witnessing, this builds upon user provided witnesses. In the paper, they describe it as ''a general strategy for progressing with POs with an existential quantifier, by instantiating it with a witness provided by the user.”

This paper focuses on developing a new Witness Annotation for the VDMJ toolset. This annotation allows users to specify a witness in a VDM specification, which is then checked through simulation. This is based on the idea of existential witnessing [14], and aims to be useful for the progression of proof obligations with an existential quantifier. It is arguably easiest for a witness to be specified at the modelling stage, rather than at the stage of proof. This is because the modeller is likely to understand their own specification best, so if another person is responsible for the proof of a model it may be harder for them to generate an appropriate witness. Also, even if a person proves their own system, time can pass between the specification and proving of a model.

After the implementation of Witness Annotations, the plan was to use the existing Relay Resistance Protocol Model [2] to show a full, real world example, to how the Witness Annotation can be used to assist with the process of formal proofs. Due to time restrictions and difficulties found in the implementation stage, this goal is incomplete. However, an example using the model can still be found in the Evaluation section.

1.2 Project Aim and Objectives

1.2.1 Aim

The aim of this dissertation, with respect to the motivation outlined above, is to implement Witness Annotations into the VDMJ toolset. To then use this on an existing payment protocol model [2], to show how it can be useful in assisting formal proofs.

1.2.2 Objectives

The project can be split into several objectives listed below, which help to define the key steps in achieving the aim.

(i) To research and explain the tools and process involved with creating a new annotation in VDMJ.

(ii) To research and explain the Relay Resistance Protocol.

(iii) To implement the Witness Annotation into VDMJ.

(iv) To use the annotation to display how it can help with the proof of formal models, by using it on an existing Relay Resistance Protocol model [2].

2 Background Research

2.1 VDM

VDM (Vienna Development Method) [15,16], is a formal method for the development of computer-based systems and was founded at the IBM Laboratory Vienna in the 1970s. It uses mathematic foundations to model systems and can therefore be used to formally prove their specification. VDM has grown over the years to include many techniques and tools, which are based on its formal specification language, VDM-SL.

VDM specifications are built in terms of models. These models consist of a range of data and functionality that defines an underlying system. It supports user-defined data types, built from base types and constructors. They can be constrained by invariants, which are logical predicates that determine the valid values of the type [13]. The functionality of a model is determined by a number of function and operation definitions. The difference being that operations can be used in conjunction with global state variables, whereas functions can not [13]. Functionality can be restrained via pre- and post-conditions [15]. These are truth valued expressions which specify the conditions that must be met before and after the evaluation of their associated function or operation [17]. Both operations and functions support implicit definitions. Implicit definitions are important for VDMs support of abstraction and allow the ability to characterise the result that should be computed without defining how to do so. This is achieved using post-conditions. The conditions used within a specification generate proof obligations, which can be considered as mathematical formulas. Proof obligations must be discharged to ensure the correctness of a model [15].

2.1.1 VDMJ

In my project I will be working with VDMJ [1]. This provides basic tool support for the VDM specification language and is written in Java. VDMJ is an open-source command line tool, but can be used by Overture [7] to provide a graphical Eclipse IDE interface. However, since the implementation process is slightly different for the Overture environment, for my project I will be sticking to the base command line VDMJ tool.

The VDMJ toolset includes many helpful features for working with the VDM specification languages. This includes: a parser; a type checker; an interpreter; a debugger; a proof obligation generator and a combinatorial test generator with coverage recording [18]. In addition, it provides automatic testing via JUnit support and user definable annotations.

2.1.2 VDMJ Annotations

Annotations were added to VDMJ in version 4.3.0 [8] and represent a form of metadata that can be added to the specification as comments. The annotations in VDMJ were inspired by annotations in Java itself. Annotations in Java can be used to affect the Java compiler without directly altering the behaviour of the program at runtime. As described in Oracles Java Documentation [9], annotations have several uses in Java, namely: information for the compiler, “annotations can used by the compiler to detect errors or suppress warnings.”; compile-time and deployment-time processing, “Software tools can process annotation information to generate code”; and runtime processing, “some annotations are available to be examined at runtime”.

VDMJ provides four standard annotations, ‘Override’, ‘Trace’, ‘NoPOG’ and ‘Printf’. However, the main intent of the feature is for specifiers to be able to easily create new annotations and add them to the VDMJ toolset themselves. As in Java, the aim of annotations in VDMJ is not to directly affect a specifications behaviour, but rather the operation of the tool itself. As listed in the VDMJ annotation guide [8], some examples of how the annotations can affect the follow aspects of operation: the parser, to enable or disable language features; the type checker, to check for overrides or supress warnings; the interpreter, to trace the execution path or variables’ values; the proof obligation generator, to skip obligations for an area of the specification.

For Witness Annotations the aim is to interact with all four of the listed features.

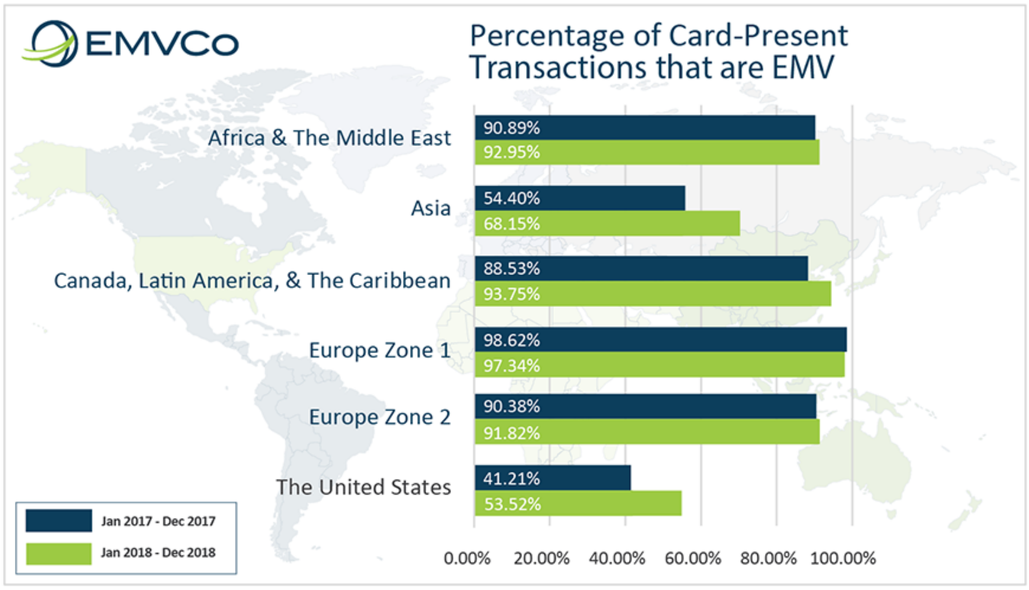

Fig. 1. Percentage of all card-present transactions processed by each member institution that are EMV transactions (Contact or Contactless) [4]

2.2 *Relay Resistance Protocol Model*

The initial decision to work with a payment protocol model for this project came purely from a base interest. However, there are some key reasons as to why this model provides a good base to test my implementation.

Payment protocols exist to help ensure the integrity of payment systems. Mainly, the protocols are present to make sure that the underlying systems are working correctly, without bugs and with minimal ability for adversaries to exploit the system. Payment systems represent a security critical situation, where security is of key concern. As such, they can benefit largely from formal modelling, where these protocols can be modelled, simulated and proven to help verify and validate that they are working correctly. Therefore, I believe the Relay Resistance Protocol, modelled by Jack Maiden [2], is a good choice to display the potential of the Witness Annotations.

2.2.1 *EMV*

EMV (standing for Europay, MasterCard, Visa) is the global standard for most of the worlds chip-based debit and credit cards. It covers not only contact payments, but also contactless and mobile based payments, alongside the Point of Service (PoS) terminals and Automated Teller Machines (ATMs) which utilise them.

EMVCo is the overarching organisation that manages the EMV specifications and related testing. EMVCo's work is overseen by its six member organisations: American Express, Discover, JCB, MasterCard, UnionPay and Visa. It "exists to facilitate worldwide interoperability and acceptance of secure payment transactions." [3]. EMV constitutes the large majority of all card-present transactions in the world, see Fig. 1.

With the massive use of the EMV standard worldwide, adversaries have been able to uncover and abuse vulnerabilities in the specifications. One such vulnerability is the relay attack.

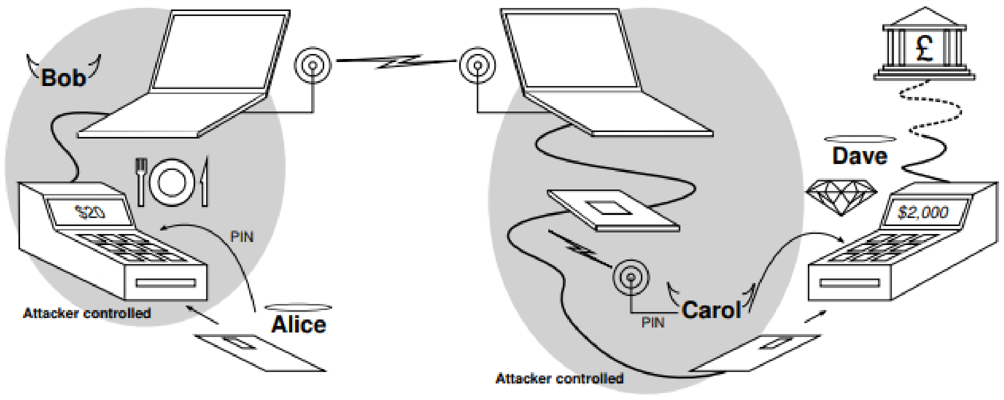

Fig. 2. Relay Attack example [5]

2.2.2 *Relay Attack*

The relay attack works via two conspiring adversaries attacking two unrelated victims at once. The basis for the attack is that a legitimate consumer, say Alice, is paying for an item at fraudulent Bob's store. At the same time the terminal relays the information to a fraudulent consumer Carol, who is completing a potentially much larger transaction at legitimate Dave's store. This example can be seen in Fig. 2.

Although this example shows an attack using a cards chip and a payment terminal, with the increasing adoption of contactless technologies, this attack is also achieved using portable card skimming devices. This combined with the ability for adversaries to repeat the attack multiple times at different locations in quick succession, makes the attack particularly effective. Especially since the legitimate consumer and shop owners will be unaware of the attack having taken place until a later time.

2.2.3 *Relay Resistance Protocol*

The Relay Resistance Protocol was introduced in the EMV 2nd Generation specifications. "The EMV 2nd Generation (2nd Gen) Specifications are a redesign of payment acceptance specifications to promote a flexible common technology platform." [6]. EMV 2nd Gen is a rebuild of the EMV specifications to attempt to improve its overall payment security.

As Jack Maiden describes in his dissertation on the topic of the protocol [2] "Relay Resistance is a protocol described to try and detect when a transaction is under the influence of a relay, and also make it so that fraudsters are deterred away from using the relay attack." As this suggests, the Relay Resistance does not outright stop a relay attack, but rather aims to prevent such a transaction from completing.

The relay resistance is described in the EMV Next Generation Kernel System Architecture Overview [6]. They describe that "The EMV Next Gen architecture can provide protection against simple relay attacks by means of the exchange of two random values, with the time taken by the exchange measured by the Next Gen Kernel". This process is done prior to a secure channel being established. After a secure channel is established, the card can communicate back to the Kernel the random value it originally used, the random value it saw from the Kernel, and the time it took to do the exchange. The Kernel will then evaluate these values, if a mismatch is found, or significant difference in reported timings, it can detect that a relay attack may be occurring. At this stage the transaction can then be prevented appropriately.

More details on the specifics of this protocol can be found in section 3.1.5 of the EMV Next Generation Kernel System Architecture Overview [6].

The aim of Jack's project [2], was to provide research into the Relay Resistance Protocol and to explore any vulnerabilities introduced and removed by introduction of the protocol. One of the objectives to achieve this aim was to create a VDM specification for the protocol's functionality. In this project, the VDM model he specified will be used to help show the results of the Witness Annotation implementation.

3 Implementation

3.1 Overview

This section of the report will cover the design and implementation process of the project. Firstly, the design of the Witness Annotation is detailed. Then the general tools used in creating new annotations are explored, alongside the more specific implementation process of the Witness Annotation.

3.2 Witness Annotation Design

The Witness Annotation is designed to work with type definitions, implicit operations and implicit functions only. Originally it was also designed to work with explicit functions and operations, however, this functionality proved to be mostly redundant due to their nature, and so the decision was made to focus on the implicit equivalents.

When specifying a Witness Annotation on a type definition, one argument is expected to be provided. This argument is a valid 'mk' call of the same type the annotation is associated with. For example,

```
--@Witness(mk_Date(12,5,19))
 Date  ::  day:nat1
           month:nat1
           year:nat
 inv mk_Date(d,m,y) == d <=31 and m<=12 and y<=3000;
```

When specifying a Witness Annotation on an implicit function or operation, up to two arguments are expected. This depends on whether a pre-condition exists on the associated construct. These arguments should be valid calls to the conditions, holding equivalent values if both exist. For example,

```
-- @Witness(post_SQRT(36, 6), pre_SQRT(36))
SQRT(x:nat) r:real
          pre x < 101
          post r*r = x
```

Ideally the user would not have to provide both calls. However, the interpreter stage does not have access to the associated definition. Therefore, prior to this stage the condition calls would need to be correctly built, most likely at the parsing stage. With the time limitations of the project this solution was not further explored, but arguably would be a more desirable solution.

This implementation focuses on the checking and validation of provided witnesses. There is also some proof obligation functionality included, but this acts as a proof of concept, rather than concrete implementation.

3.3 Implementing Annotations

The implementation of a new annotation in VDMJ can be split into four main stages which relate directly to the parser, the type checker, the interpreter and the proof obligation generator respectively [8]. Each of these stages has its own class hierarchy which can be considered as a tree. The names of all classes in a hierarchy begin with an abbreviation associated to the function. The abbreviations are as follows: the parser, AST; the type checker, TC; the interpreter, IN; and the proof obligation generator, PO. These will feature throughout the rest of the paper. At the top of these hierarchies sits a 'Node' class. This class details the common behaviour of all nodes found in the hierarchy. Directly extending the 'Node' class is a layer of production classes. Generally, the production classes represent the various syntactical elements of VDMJ, such as expressions, definitions and annotations themselves. Each of the production classes then has one or more layers of sub-production classes. The sub-production classes represent more specific versions of their parents and extend the parent class. For example, a class representing a statement has a class representing a function statement extending it, which in turn has a class representing an implicit function statement extending that.

Any annotation in VDMJ must have a class in the AST hierarchy to allow it to be parsed correctly [8]. The name of the class determines the name of the annotation. For example, the Witness Annotation would have a class 'ASTWitnessAnnotation' which extends 'ASTAnnotation'. The other three stages are optional and are implemented on a per annotation basis.

Below are more details on the specific role each stage has, in terms of annotations.

3.3.1 Parser

The parser has two main functions relating to annotations in VDMJ. Firstly, all comments are parsed to determine if they are a valid annotation [8]. For an annotation to be valid it must appear at the very start of a comment with the correct syntax and the comment must precede another syntactic construct, being either a class, module, definition, expression or statement. Multiple annotations can be applied to one construct and can be interleaved with textual comments, so long as they exist in their own comment each. The annotation must also have a valid class in the AST hierarchy, otherwise even if the syntax is correct, it is assumed to not be an annotation. Each annotation can also be parsed with a list of arguments, which can be used further through the process.

Secondly, at this stage each type of annotation can individually affect the default parse to handle its own argument syntax and the syntax of the construct it precedes.

After the parse of the specification is completed. The hierarchy of AST objects is converted into an equivalent hierarchy of TC objects [8]. For example, an 'ASTWitnessAnnotation' object would be converted into a 'TCWitnessAnnotation' object.

3.3.2 Type Checker

The type checker is responsible for checking the arguments provided with an annotation. It can check the number of arguments provided, their types and their individual syntax. Alongside checking the arguments provided, the type checker can also check an annotations location in the specification. The location of an annotation is tied to the construct that it precedes. At this stage it's possible to restrict the acceptance of an annotation by which type of construct it related to. For example, the default 'Printf' annotation is designed to only be accepted when preceding either a statement or expression [8].

After the annotations have been type-checked, the process simultaneously splits into the remaining two stages. The hierarchy of TC objects is mapped to an equivalent hierarchy of IN objects and an equivalent hierarchy of PO objects [8]. As such, the interpreter and proof obligation generator classes never interact.

3.3.3 Interpreter

The interpreter is responsible for the evaluation of an annotation's arguments. There are two main uses for this. Firstly, if a definition is provided as an argument, it can be evaluated to check it satisfies any invariants, pre-conditions or post-conditions. Secondly, arguments can be evaluated and compared against other values. In addition to evaluating values it can be used to trace the path of an execution.

3.3.4 Proof Obligation Generator

The proof obligation generator stage allows the annotation to affect how proofs are generated for the construct it relates to. This could be to introduce new proofs, affect existing ones, or to skip the obligations for an area of the specification.

3.4 Implementing Witness Annotations

Witness annotations are designed to interact with all four of the stages. Therefore, it required the implementation of four new classes, 'ASTWitnessAnnotation', 'TCWitnessAnnotation', 'INWitnessAnnotation' and 'POWitnessAnnotation'. The full code for these classes can be found in the Appendix at end of the report.

3.4.1 ASTWitnessAnnotation

As for any annotation to work in VDMJ, the Witness Annotation required the 'ASTWitnessAnnotation' class. However, the witness annotation does not affect the parse. The annotation guide details, "Annotations that don't affect the parse do not have to implement any methods." [8], and so, the class is simply the constructor.

3.4.2 TCWitnessAnnotation

The first main role of the class is to ensure that Witness Annotations are only valid when associated with the correct syntactical categories. This is achieved by overriding the 'before' methods provided in the default 'TCAnnotation' class. There exists one of these methods for each of the syntactical categories of VDM and are called prior to the type check of an annotated element. For my implementation it is only desired for Witness Annotations to work with type definitions, implicit functions and implicit operations. These all fall under the syntactical category of definitions.

```
@Override
public void tcBefore(TCModule module){
        name.report(6020, "@Witness only applies to
            implicit operations, implicit functions
            and type definitions");
}
```

The above is an example of how the annotation is restricted from being associated with a module construct. The 'name.report' is a built in function of VDMJ to throw errors in the type checker stage. It generates an error message with the passed arguments and stops execution of the specification. The 'before' methods for class definitions, expressions and statements follow the same structure as above.

For definitions, since they are supported by the Witness Annotation, the 'after' method is overridden instead. This differs from the before method in the sense that it is invoked after the type check has finished and can easily access the resolved types.

```
@Override
public void tcAfter(TCDefinition def, TCType type,
    Environment env, NameScope scope){
        checkArgs(def, type, env, scope);
}
```

The 'checkArgs' method, invoked above, is a self-designed method holding the logic behind the type checking of a Witness Annotations arguments. Its function can be split into three steps: determining the definition type; checking the number of arguments provided; and checking the types of the arguments provided.

Determining the definition type is a simple task. The following displays how it is done in my implementation.

```
if(def.isTypeDefinition()){
        ...
}
else if(def instanceof TCImplicitFunctionDefinition){
        ...
}
else if(def instanceof TCImplicitOperationDefinition)
    {
        ...
}
```

After determining the definition type, in each case the number of arguments provided with the annotation is checked. This is also relatively simple.

```
if(args.size() != 1){
        name.report(6021, "@Witness on a type
            definition should only have one argument")
            ;
}
```

For a type definition, only one argument is ever expected. For an implicit function or implicit operation, it varies. If a pre-condition exists then two arguments are expected, else only one is expected.

Finally, if there exists the correct number of arguments, the arguments are then type checked themselves. In the case of functions and operations, the arguments are first examined to determine whether they are of the correct form.

```
if(args.get(0) instanceof TCApplyExpression){
        ...
}
```

Then their names are checked for correctness.

```
//Get the expected name (def) and the actual name (
   arg)
String imDefPostName = imDef.postdef.name.toString();
String funcPostArgName = funcPostArg.root.toString();

//Compare the names
if(!imDefPostName.equals(funcPostArgName)){
        ...
}
```

In addition to this, if both pre- and post-conditions exist, their arguments are checked against each other's, ensuring they are equivalent. Finally, an existing type checking method is invoked,

```
args.get(0).typeCheck(env, null, scope, null);
```

This method checks the types of the provided arguments are correct with respect to the associated construct. When dealing with type definitions the only check required is the 'typeCheck'. This ensures the argument is a 'mk' type call if for the correct type, as well as checking the arguments provided with that call.

3.4.3 INWitnessAnnotation

The interpreter class is responsible for evaluating the arguments provided with a Witness Annotation, allowing any applicable invariants and conditions to be evaluated. This is important when determining if a provided witness is valid or not.

By default, the interpreter is designed to only be affected by statements and expressions. "Annotations which apply to classes/modules and definitions do not affect the interpreter, but those that apply to statements and expressions do (since these elements are "executed"). [8]. This means that 'before' and 'after' methods that exist in the default "INAnnotation" class are not applicable to definitions. However, it is important that the arguments provided with Witness Annotations are executed, to determine their condition or invariant satisfiability. The solution for this is to execute the arguments on initialisation. This was not possible with the original VDMJ toolset, since the available initialisation methods were static. As an attempted work around the evaluation was attempted in the class constructor, but this was also not possible. This is because the evaluation method requires a 'context' object to be passed as a parameter, so that any expression and statement evaluation can access the values of variables. In the case of definitions, the ’context’ is still required, although not used, and when the constructor is called it is not possible to access a valid ’context’ object. The working solution required that a new non-static initialisation method was introduced into the framework, this was achieved in accordance of VDMJs author. This new method allowed the individual initialisation of each instance of a ‘INAnnotation’ object, and access to a valid ’context’ object.

With this new initialisation method, it was now possible to evaluate the arguments of a Witness Annotation. For type definitions this is achieved easily by the following.

```
Value v = args.get(0).eval(ctxt);
```

This line will evaluate the ‘mk’ type call. If the invariant is not satisfied an error will be thrown, since it’s not a valid witness.

The process is similar for functions and operations, the above is called for the provided condition arguments. The difference is that the calculated value then needs to be checked. If the conditions are met, then ‘v’ will be true. Otherwise it will be false. If it evaluates to false then the condition has not been met, and hence it’s not a valid witness. This functionality is achieved via the below.

```
Value v1 = args.get(0).eval(ctxt);
if(v1 instanceof BooleanValue){
    BooleanValue vB1 = (BooleanValue) v1;
    if(vB1.value == false){
        String msg = this + ” is a bad witness.
           Postcondition not met.”;
        ExceptionHandler.abort(name.getLocation(),
           6222, msg, ctxt);
    }
}
```

3.4.4 POWitnessAnnotation

The proof obligation class allows the modification and removal of existing proof obligations and generation of new obligations. By overriding the ’after’ method for the definition syntax category, similarly to what was done for ’TCWitnessAnnotations’, it’s possible to access the list of proof obligations VDMJ has generated for each construct. This can be seen below.

```
@Override
public void poAfter(PODefinition def,
   ProofObligationList obligations, POContextStack
   ctxt) {
       for(ProofObligation po : obligations){
              ...
       }
}
```

The implementation of ‘POWitnessAnnotation’ for this project acts as more of a proof of concept than a concrete solution. The below demonstrates how a function satisfiability proof that VDMJ generates can be updated to have a valid witness introduced. This reduces the existential proof, which in turn simplifies the proof process.

```
for(ProofObligation po : obligations){
    if(po.kind.equals(POType.FUNC_SATISFIABILITY)){
        POImplicitFunctionDefinition imDef = (
            POImplicitFunctionDefinition) def;
        String imDefResultName = imDef.result.
            pattern.toString();
        POApplyExpression postArg = (
            POApplyExpression) args.get(0);
        String postResult = postArg.args.get(args.
            size()-1).toString();
        String poValue = po.value;
        poValue = poValue.replaceAll(
            imDefResultName + ":", postResult + ":")
            ;
        poValue = poValue.replaceAll(
            imDefResultName + "\\)", postResult +"
            \\)");
        po.value = poValue;
    }
}
```

3.4.5 Assumptions

The one key assumption with this implementation is that the IN stage is run prior to the PO stage. Since they both map directly from the TC stage it is not guaranteed that this is the case. However, for Witness Annotations it is important that the IN stage is run prior to the PO stage to ensure that only valid witnesses are used in the PO stage. If an invalid witness is present at the IN stage, the specification will stop execution and an appropriate error will be thrown, meaning that it won't be possible to get to the PO stage with an invalid witness; assuming the IN stage is performed first.

4 Evaluation

4.1 Overview

This section presents an evaluation of the Witness Annotation implementation, in the form of both valid and invalid witnesses in use. Firstly, there are some more general working examples, representing the initial results, followed by an example on the Relay Resistance Protocol model [2].

4.2 Initial Results

The following specification was used to initially test the implementation of Witness Annotations. It includes examples of both valid and invalid witnesses for type definitions, implicit functions and implicit operations.

```
types
        -- @Witness(mk_Date(0,2,3))
        -- @Witness(mk_Date(32,2,2010))
        -- @Witness(mk_Date(30,2,2010))
        Date :: day:nat1
                month:nat1
                year:nat
        inv mk_Date(d,m,y) == d <= 31 and m<=12 and y
            <=3000

functions
        -- @Witness(post_SQRT(25,5))
        -- @Witness(post_SQRT(36,6), pre_SQRT(25))
        -- @Witness(post_SQRT(121,11), pre_SQRT(121))
        -- @Witness(post_SQRT(36,5), pre_SQRT(36))
        -- @Witness(post_SQRT(36,6), pre_SQRT(36))
        SQRT(x:nat) r:real
                pre x < 101
                post r*r = x

operations
        --@Witness(post_ADD(10, mk_Register(0),
            mk_Register(25)), pre_ADD(10, mk_Register
            (0)))
        --@Witness(post_ADD(10, mk_Register(0),
            mk_Register(10)), pre_ADD(10, mk_Register
            (0)))
        ADD(i:nat)
                pre i > 9
                post someStateRegister =
                    someStateRegister~ + i

state Register of
        someStateRegister : nat
end
```

Below lists the error, or success, messages displayed for each of the examples. It details how the invalid witnesses are caught. Each of the Witness Annotations from above were run in their own execution of the specification to allow them to progress to the necessary stages for an error or success to be shown.

```
@Witness(mk_Date(0,2,3))
> Error 3327: Value is not of the right type. Actual:
    nat, Expected: nat1

@Witness(mk_Date(32,2,2010))
> @Witness(mk_Date(32, 2, 2010)) is a bad witness.
```

```
   Error 4079: Type invariant violated by mk_Date
   arguments.

@Witness(mk_Date(30,2,2010))
> @Witness(mk_Date(30, 2, 2010)) is a good witness

@Witness(post_SQRT(25,5))
> Error 6023: @Witness on an implicit function with a
    precondition should have two arguments, a post
   condition call followed by a pre condition call.

@Witness(post_SQRT(36,6), pre_SQRT(25))
> Error 6026: @Witness post and pre calls have
   different arguments passed.  Post Argument (0): :
   36, Corresponding Pre Argument: : 25

@Witness(post_SQRT(121,11), pre_SQRT(121))
> Error 6223: @Witness(post_SQRT(121, 11), pre_SQRT
   (121)) is a bad witness. Precondition not met.

@Witness(post_SQRT(36,5), pre_SQRT(36))
> Error 6222: @Witness(post_SQRT(36, 5), pre_SQRT(36)
   ) is a bad witness. Postcondition not met.

@Witness(post_SQRT(36,6), pre_SQRT(36))
> @Witness(post_SQRT(36, 6), pre_SQRT(36)) is a good
   witness

@Witness(post_ADD(10, mk_Register(0), mk_Register(25)
   ), pre_ADD(10, mk_Register(0)))
>  Error 6222: @Witness(post_ADD(10, mk_Register(0),
   mk_Register(25)), pre_ADD(10, mk_Register(0))) is
   a bad witness. Postcondition not met.

@Witness(post_ADD(10, mk_Register(0), mk_Register(10)
   ), pre_ADD(10, mk_Register(0)))
> @Witness(post_ADD(10, mk_Register(0), mk_Register
   (10)), pre_ADD(10, mk_Register(0))) is a good
   witness
```

These results display how the Witness Annotation correctly distinguishes invalid witnesses from the valid ones. At this stage any execution containing an invalid witness will be stopped, with the error. Therefore, only valid witness will be left. These witnesses can now be processed by the ”POWitnessAnnotation” class to perform some desired functionality to assist with simplifying proof obligations. As discussed in the Implementation section, the current implementation of this class acts only as a proof of concept. However, using the currently implemented ’POWitnessAnnotation” class would easily allow for more cases to be added.

Below demonstrates how the valid 'SQRT' Witness Annotation has been used to affect an existing proof obligation. This process was done using the implementation discussed earlier in the paper, which corresponds to the code found in the Appendix at the end of the paper.

```
-- Prior to POWitnessAnnotation
Proof Obligation 1: (Unproved)
SQRT: function satisfiability obligation in 'DEFAULT'
    (test.vdm) at line 17:5
(forall x:nat &
  pre_SQRT(x) => exists r:real & post_SQRT(x, r))

-- After POWitnessAnnotation
Proof Obligation 1: (Unproved)
SQRT: function satisfiability obligation in 'DEFAULT'
    (test.vdm) at line 17:5
(forall x:nat &
  pre_SQRT(x) => exists 6:real & post_SQRT(x, 6))
```

The second proof obligation above gives an example of how the proof obligation can be reduced with the knowledge of the valid witness. This gives an example of an advantage introduced by the Witness Annotation.

I believe these results show that the annotation is successful in providing a easy way for the user to specify a witness, that has been evaluated to ensure its correctness, and can be easily accessed for the use in existential proof obligations. Therefore, with some additions to the 'POWitnessAnnotation' class, the annotation could be setup to automatically adjust proof obligations to help in the process of formal proofs.

4.3 Relay Resistance Protocol Results

Due to time limitations, and earlier difficulties during the project, I was unable to complete this stage of the project. However, below shows an example of how the Witness Annotation has been successfully included with the Relay Resistance Protocol specification.

```
values
        pdol = mk_PDOL(129,10,0,<UK>, <POUND_STERLING
           >);
        AcardNormal = mk_Card(['a','c','a','r','d
           '],125,15,30,10);
        AterminalNormal = mk_Terminal(['a','t','e','r
           ','m'], pdol, 10, 40, 5, 10, 15, 10, 9);
        TransactionState = mk_TransactionState(
           AcardNormal, AterminalNormal, {|->},
           {|->}, 0, 0, 0);

operations
```

```
-- @Witness(post_fcnCSN(AcardNormal,125, TS,
    TS),pre_fcnCSN(AcardNormal,TS))
fcnCSN(a:Agent) n:Number ==
(dcl n: nat := 0;
if (is_Card(a)) then n:= a.seqNum;
return n)
pre is_Card(a) and a.id = aCard.id
post a.id = aCard.id <=> (n = aCard.seqNum
    and n >=1);

--Message displayed from end of IN stage showing it
    was a good witness,
--//@Witness(post_fcnCSN(rrp`AcardNormal, 125, rrp`TS
    , rrp`TS), pre_fcnCSN(rrp`AcardNormal, rrp`TS)) is
     a good witness.
```

With more time, the entire specification could utilise Witness Annotations, and the effect this has on the proof obligations could be explored.

5 Conclusion

This project provides information on the toolset offered by VDMJ, alongside explanations of how to implement new annotations. At the end of this project, Witness Annotations have been successfully implemented, and provide the ability for users to specify witnesses for their definitions. Witness Annotations work with type definitions, implicit functions and implicit operations and correctly parse, type check and interpret. These annotations can be easily accessed to adjust existing proof obligations or create new ones. This can be advantageous by helping reduce the obligations, to allow for simpler proofs. I believe the Witness Annotation provides an easy, yet effective way to allow users to provide valid witnesses at the modelling stage, whilst remaining accessible to assist in existential proofs.

In addition, an existing Relay Resistance Protocol model [2] has been researched. This protocol represents a useful formal model that would benefit from formal proofs. An example of how Witness Annotations can be integrated into the model's specification has been shown. Further work would be needed to fully integrate and evaluate their overall effect.

5.1 Fulfilment of objectives

The aim of this project was detailed by four objectives. This subsection will evaluate these objectives to detail how they've been achieved, or why they have not.

"To research and explain the technologies
and process involved with creating a new annotation in VDMJ"

I believe that throughout the report this objective has been satisfied. The Background section of this paper details the formal method of VDM, as well as the toolset offered by VDMJ, including annotations. The process of creating a new annotation is explained thoroughly during the Implementation section.

"To research and explain the Relay Resistance Protocol"

I also believe that this objective has been satisfied by this report. The Background section introduces the EMV specifications, from which the protocol originates. In addition, the details on the relay attack provides the background understanding as to why the protocol is required. The Background section also explains how the protocol attempts to combat the potential of the relay attack.

"To implement the Witness Annotation into VDMJ"

This objective represented the bulk of the work to be done. The Implementation and Evaluation sections show that Witness Annotations have been successfully implemented into the VDMJ toolset. I set out to support type definitions, functions and operations. During the implementation stage I decided to shift the focus on the implicit versions of functions and operations since I found the explicit versions to be mostly redundant due to their nature.

This objective provided some complications. The main complication came with implementing the INWitnessAnnotation class. This was because the base INAnnotation class only supported the interpretation of expressions and statements, whereas for the Witness Annotation implementation there was a need for definitions to be evaluated. This caused extra time to be taken to find a solution. I was able to overcome this complication by working with the VDMJ author to introduce new functionality into the INAnnotation class, allowing for the evaluation of definitions.

I believe this objective has been achieved. However, there would be some improvements and potential additional features that could be considered. I will outline these in the future work subsection, found in the subsection below.

"To use the annotation to display how it can help with the proof of formal models, by using it on the existing Relay Resistance Protocol model [2]"

This objective is incomplete due to time constraints. The main reason for this was due to the difficulties found with the implementation stage, which was required to be completed prior to this. I do believe I was able to display briefly how this objective would be completed, with an example found in the Evaluation section. With more time a full usage of Witness Annotations would have been conducted on the Relay Resistance Protocol model [2], and further examination into its advantage could have been produced.

5.2 Skills and Knowledge Gained

Before starting the project, I had limited experience with formal modelling and proofs. As such it required me to gain new knowledge into this area. I believe that during the undertaking of this project I have advanced my knowledge in these areas. Below are the skills I believe I have used the most during this project.

Programming: Coming into the project, I feel that programming was one of my strengths. I have also worked extensively with Java, so am very familiar with the language. However, I had not much experience adding to other people's frameworks. As such, I believe I have gained a wider understanding of the approach to take in such cases.

Time Management: With larger projects time management becomes increasingly important, especially with a higher risk of encountering unexpected issues. During this project, my time management has been tested, and unfortunately, I was unable to complete all the goals I set out to achieve. During the project I believe my time management skills have improved.

Research: I have undertaken several projects before with focus on research. In these cases, there have often been many available, and easy to find, sources to reference. For this project, it has been difficult to find suitable references. I believe I have gained some understanding into how research can be complicated, and ability into finding appropriate sources.

Formal Writing: Prior to this project I have felt my formal writing skills have been decent. However, the formal writing of this report has proven challenging. I feel that writing a report of this size can be difficult, with many iterations needed to improve it. I believe this project has provided me with valuable experience with formal writing and has helped me to understand some of its intricacies.

5.3 Future Work

The implementation has several improvements and extensions that could be made. One improvement lies in the usage of Witnesses in proof obligations. I outlined briefly in this project how the Witness Annotation could be used to help progress proof obligations with an existential quantifier. This acted as a proof of concept, rather than a concrete design. With more time and knowledge, a more concrete implementation could be achieved.

Another enhancement the implementation would benefit from is an improved argument syntax. Currently it is required that both a pre- and post-condition argument are provided with a Witness Annotation on a function or operation. I believe, with more time, it would be possible to streamline this to improve usability.

Lastly, one of the more useful improvements would come in the form of Witness Annotation references. The idea is that the result of a prior Witness Annotation could be utilised in a later Witness Annotation. This would improve usability, especially in more complex examples where multiple user definitions may be nested. This feature is relatively complicated to achieve and could contain many nuances. As such, I felt it was out of scope for this project's timeline.

6 Acknowledgements

I would like to thank Nick Battle for his help in adjusting the VDMJ framework to allow for the evaluation of definitions in the 'INAnnotation' class.

References

[1] Battle. N: VDMJ Repository Readme — GitHub, https://github.com/nickbattle/vdmj [Accessed 13th May 2019]

[2] Maiden, J.: EMV's Relay Resistance Protocol in MasterCard Contactless Specification. New-castle University MSc Dissertation (2017)

[3] About EMVCo — EMVCo, https://www.emvco.com/about/overview/ [Accessed 13th May 2019]

[4] EMV Deployment Statistics — EMVCo, https://www.emvco.com/about/deployment-statistics/ [Accessed 13th May 2019]

[5] Drimer, S., Murdoch, S.J.: Keep Your Enemies Close: Distance Bounding Against Smartcard Relay Attacks. (2006)

[6] EMV 2nd Generation — Next Generation Kernel System Overview. Version 1.0, EMVCo (2014). Available as pdf via https://www.emvco.com/emv-technologies/2-gen/ [Accessed 13th May 2019]

[7] Overture Tool Overview — Overture Tool, http://overturetool.org/ [Accessed 13th May 2019]

[8] Battle. N: VDMJ Annotations Guide — GitHub, https://github.com/nickbattle/vdmj/blob/master/FJ-VDMJ4/documentation/AnnotationGuide.pdf [Accessed 13th May 2019]

[9] Java Annotations Tutorial — Oracle, https://docs.oracle.com/javase/tutorial/java/annotations/ [Accessed 13th May 2019]

[10] Overture AST Guide — GitHub, https://github.com/overturetool/overture/wiki/AST-Guide [Accessed 13th May 2019]

[11] Freitas L., Whiteside I.: Proof Patterns for Formal Methods (2014)

[12] Woodcock J., Larsen P., Bicarregui J., Fitzgerald J.: Formal Methods: Practice and Experience (2009)

[13] Larsen P., Battle N., Ferreira M., Fitzgerald J., Lausdahl K., Verhoef M.: The Overture Initiative Intergrating Tools for VDM (2010)

[14] Freitas L., Jones C., Velykis A., Whiteside I.: A Model for Capturing and Replaying Proof Strategies (2014)

[15] Jones C.: Systematic Software Development Using VDM (1995)

[16] Jones C.: Scientific Decisions which Characterize VDM (1999)

[17] Aichernig B., Larsen P.: A Proof Obligation Generator for VDM-SL (1997)

[18] Battle. N: VDMJ User Guide — GitHub, https://github.com/nickbattle/vdmj/blob/master/FJ-VDMJ4/documentation/UserGuide.pdf [Accessed 13th May 2019]

Appendix

ASTWitnessAnnotation

```
package annotations.ast;

import com.fujitsu.vdmj.ast.annotations.ASTAnnotation;
import com.fujitsu.vdmj.ast.lex.LexIdentifierToken;

public class ASTWitnessAnnotation extends ASTAnnotation
{
     public ASTWitnessAnnotation(LexIdentifierToken name){
          super(name);
     }
}
```

TCWitnessAnnotation

```
package annotations.tc;

import java.util.Arrays;

import com.fujitsu.vdmj.tc.annotations.TCAnnotation;
import com.fujitsu.vdmj.tc.definitions.TCClassDefinition;
import com.fujitsu.vdmj.tc.definitions.TCDefinition;
import com.fujitsu.vdmj.tc.expressions.TCExpression;
import com.fujitsu.vdmj.tc.expressions.TCExpressionList;
import com.fujitsu.vdmj.tc.lex.TCIdentifierToken;
import com.fujitsu.vdmj.tc.modules.TCModule;
import com.fujitsu.vdmj.tc.statements.TCStatement;
import com.fujitsu.vdmj.typechecker.Environment;
import com.fujitsu.vdmj.typechecker.NameScope;

import com.fujitsu.vdmj.typechecker.TypeComparator;
import com.fujitsu.vdmj.tc.types.TCType;
import com.fujitsu.vdmj.tc.expressions.TCApplyExpression;
import com.fujitsu.vdmj.tc.types.TCTypeList;
import com.fujitsu.vdmj.tc.types.TCFunctionType;
import com.fujitsu.vdmj.tc.types.TCOperationType;
import com.fujitsu.vdmj.tc.definitions.TCImplicitFunctionDefinition;
import com.fujitsu.vdmj.tc.definitions.TCImplicitOperationDefinition;

public class TCWitnessAnnotation extends TCAnnotation{

     public TCWitnessAnnotation(TCIdentifierToken name, TCExpressionList args){
          super(name, args);
     }

     //TCAfter to ensure the types have been resolved rather than TCBefore
     @Override
     public void tcAfter(TCDefinition def, TCType type, Environment env, NameScope
          scope){
          checkArgs(def, type, env, scope);
     }

     //Method to check the number of arguments provided and the type of those
        arguments
     private void checkArgs(TCDefinition def, TCType type, Environment env,
        NameScope scope){
          //If annotation is on a type definition
          if(def.isTypeDefinition()){
               //Check there's only one argument
               if(args.size() != 1) {
                  name.report(6021, "@Witness on a type definition should only
                       have one argument");
               }
               //If so, type check it
               else{
                    args.get(0).typeCheck(env,null,scope,type);
               }
          }
```

```
//If annotation is on an implicit function
else if (def instanceof TCImplicitFunctionDefinition){

     //Cast to the more specific type
     TCImplicitFunctionDefinition imDef = (TCImplicitFunctionDefinition)
          def;

     //If no precondition exists for this function
     if(imDef.predef == null){
          //Since no precondition, just one argument is expected
          if(args.size() != 1) {
               name.report(6022, "@Witness on an implicit function
                    without a precondition should only have one argument,
                     a post condition call.");
          }
          else{
          //Correct number of arguments, check that the first argument
               is an apply expression
          if(args.get(0) instanceof TCApplyExpression){
               //Cast to the more specific expression type
               TCApplyExpression funcPostArg = (TCApplyExpression) args.
                    get(0);

               //Get the expected name (def) and the actual name (arg)
               String imDefPostName = imDef.postdef.name.toString();
               String funcPostArgName = funcPostArg.root.toString();

               //Compare the names
               if(!imDefPostName.equals(funcPostArgName)){
                    name.report(6024, "@Witness has incorrect function
                         post condition call");
                    name.detail2("Actual: " , funcPostArgName , "
                         Expected: ", imDefPostName);
               }
               //Else, correct post func name, check its arguments
               else{
                  args.get(0).typeCheck(env,null,scope,null); // Just
                       checks scope
               }
          }
          //Else, first argument not an apply expression,
          else {
               name.report(6022, "@Witness on an implicit function
                    without a precondition should have one argument, a
                    post condition call.");
          }
     }
}
//Else, precondition exists for this function
else{
     //Since precondition exists, two arguments are expected
     if(args.size() != 2) {
          name.report(6023, "@Witness on an implicit function with a
               precondition should have two arguments, a post condition
               call followed by a pre condition call.");
     }
     else{
          //Correct number of arguments, check that both arguments are
               apply expressions
          if(args.get(0) instanceof TCApplyExpression && args.get(1)
               instanceof TCApplyExpression){
               //Cast to the more specific expression type
               TCApplyExpression funcPostArg = (TCApplyExpression) args.
                    get(0);
               TCApplyExpression funcPreArg = (TCApplyExpression) args.
                    get(1);

               //Get the expected names (def) and the actual names (arg)
               String imDefPostName = imDef.postdef.name.toString();
               String funcPostArgName = funcPostArg.root.toString();
               String imDefPreName = imDef.predef.name.toString();
               String funcPreArgName = funcPreArg.root.toString();
```

```
                    //Compare the post names
                    if(!imDefPostName.equals(funcPostArgName)){
                         name.report(6024, "@Witness has incorrect function
                              post condition call");
                         name.detail2("Actual: " , funcPostArgName , "
                              Expected: ", imDefPostName);
                    }
                    //Compare the pre names
                    else if(!imDefPreName.equals(funcPreArgName)){
                         name.report(6025, "@Witness has incorrect function
                              pre condition call");
                         name.detail2("Actual: " , funcPreArgName , "Expected
                              : ", imDefPreName);
                    }
                    //Else, correct names, check their arguments
                    else{
                         args.get(0).typeCheck(env,null,scope,null); // Just
                              checks scope
                         args.get(1).typeCheck(env,null,scope,null); // Just
                              checks scope

                         //Check that arguments in both pre and post match (
                              excludes the last argument of post since is
                              output and doesn't exist in pre)
                         int i = 0;
                         for(TCExpression arg : funcPreArg.args){
                              if(!funcPostArg.args.get(i).equals(arg)){
                                   name.report(6026, "@Witness post and pre
                                        calls have different arguments passed
                                        ");
                                   name.detail2("Post Argument (" + i + "): "
                                        , funcPostArg.args.get(i), "
                                        Corresponding Pre Argument: ", arg);
                              }
                              i++;
                         }

                    }
               }
               else {
                    name.report(6023, "@Witness on an implicit function with
                         a precondition should have two arguments, a post
                         condition call followed by a pre condition call.");
               }
          }
     }
}

     //If annotation is on an implicit operation
     else if(def instanceof TCImplicitOperationDefinition){

          //Cast to the more specific type
          TCImplicitOperationDefinition imDef = (
               TCImplicitOperationDefinition) def;

          //If no precondition exists for this operation
          if(imDef.predef == null){
          //Since no precondition, just one argument is expected
          if(args.size() != 1) {
               name.report(6022, "@Witness on an implicit operation without a
                    precondition should only have one argument, a post
                    condition call.");
          }
          else{
               //Correct number of arguments, check that the first argument
                    is an apply expression
               if(args.get(0) instanceof TCApplyExpression){
                    //Cast to the more specific expression type
                    TCApplyExpression operPostArg = (TCApplyExpression) args.
                         get(0);
```

```
                //Get the expected name (def) and the actual name (arg)
                String imDefPostName = imDef.postdef.name.toString();
                String operPostArgName = operPostArg.root.toString();

                //Compare the names
                if(!imDefPostName.equals(operPostArgName)){
                    name.report(6024, "@Witness has incorrect operation
                        post condition call");
                    name.detail2("Actual: " , operPostArgName , "
                        Expected: ", imDefPostName);
                }
                //Else, correct post oper name, check its arguments
                else{
                    args.get(0).typeCheck(env,null,scope,null); // Just
                        checks scope
                }
            }
            else {
                name.report(6022, "@Witness on an implicit operation
                    without a precondition should have one argument, a
                    post condition call.");
            }
        }
    }
    //Else, precondition exists for this operation
    else{
        //Since precondition exists, two arguments are expected
        if(args.size() != 2) {
            name.report(6023, "@Witness on an implicit operation with a
                precondition should have two arguments, a post condition
                call followed by a pre condition call.");
        }
        else{
            //Correct number of arguments, check that both arguments are
                apply expressions
            if(args.get(0) instanceof TCApplyExpression && args.get(1)
                instanceof TCApplyExpression){
            //Cast to the more specific expression type
            TCApplyExpression operPostArg = (TCApplyExpression) args.get
                (0);
            TCApplyExpression operPreArg = (TCApplyExpression) args.get(1)
                ;

            //Get the expected names (def) and the actual names (arg)
            String imDefPostName = imDef.postdef.name.toString();
            String operPostArgName = operPostArg.root.toString();
            String imDefPreName = imDef.predef.name.toString();
            String operPreArgName = operPreArg.root.toString();

            //Compare the post names
            if(!imDefPostName.equals(operPostArgName)){
                name.report(6024, "@Witness has incorrect operation post
                    condition call");
                name.detail2("Actual: " , operPostArgName , "Expected: ",
                    imDefPostName);
            }
            //Compare the pre names
            else if(!imDefPreName.equals(operPreArgName)){
                name.report(6025, "@Witness has incorrect operation pre
                    condition call");
                name.detail2("Actual: " , operPreArgName , "Expected: ",
                    imDefPreName);
            }
            //Else, correct names, check their arguments
            else{
                args.get(0).typeCheck(env,null,scope,null); // Just
                    checks scope
                args.get(1).typeCheck(env,null,scope,null); // Just
                    checks scope

                //Check that arguments in both pre and post match
                //If result is null, can do so as in the function section
```

```
                    (all but last arg in post)
                if(imDef.result == null){
                    for(int i=0; i<operPreArg.args.size(); i++){
                        System.out.println("Post Argument (" + i + "):
                            " + operPostArg.args.get(i) + "
                            Corresponding Pre Argument: " + operPreArg.
                            args.get(i));
                        if(!operPostArg.args.get(i).equals(operPreArg.
                            args.get(i))){
                             name.report(6026, "@Witness post and pre
                                 calls have different arguments passed
                                 ");
                             name.detail2("Post Argument (" + i + "): "
                                  , operPostArg.args.get(i), "
                                 Corresponding Pre Argument: ",
                                 operPreArg.args.get(i));
                        }
                    }
                }
                //Else works differently, due to the state objects needed
                    with operations
                else{
                    //Check all but last in Pre with corresponding post
                    for(int i=0; i<operPreArg.args.size()-1; i++){
                        System.out.println("Post Argument (" + i + "):
                            " + operPostArg.args.get(i) + "
                            Corresponding Pre Argument: " + operPreArg.
                            args.get(i));
                             if(!operPostArg.args.get(i).equals(
                                 operPreArg.args.get(i))){
                                  name.report(6026, "@Witness post and
                                      pre calls have different
                                      arguments passed");
                                  name.detail2("Post Argument (" + i +
                                      "): " , operPostArg.args.get(i),
                                      "Corresponding Pre Argument: ",
                                      operPreArg.args.get(i));
                             }
                        }
                        //Now check last argument in pre with
                            corresponding post (one index prior to that
                             of post)
                        //This is needed due to the state variables
                            passed with pre and post conditions for
                            operations
                        int i = operPreArg.args.size()-1;
                        System.out.println("Post Argument (" + (i+1) +
                            "): " + operPostArg.args.get(i+1) + "
                            Corresponding Pre Argument: " + operPreArg.
                            args.get(i));
                        if(!operPostArg.args.get(i+1).equals(operPreArg
                            .args.get(i))){
                             name.report(6026, "@Witness post and pre
                                 calls have different arguments passed
                                 ");
                             name.detail2("Post Argument (" + (i+1) +
                                 "): " , operPostArg.args.get(i+1), "
                                 Corresponding Pre Argument(" + i + "):
                                  ", operPreArg.args.get(i));
                        }
                    }
                }
            }
            else {
                name.report(6023, "@Witness on an implicit operation with
                     a precondition should have two arguments, a post
                    condition call followed by a pre condition call.");
                }
            }
        }
    }
    }

    @Override
    public void tcBefore(TCModule module){
        name.report(6020, "@Witness only applies to operations, functions and
            type definitions");
    }

    @Override
    public void tcBefore(TCClassDefinition clazz){
        name.report(6020, "@Witness only applies to functions and type
            definitions");
    }

    @Override
    public void tcBefore(TCExpression exp, Environment env, NameScope scope){
        name.report(6020, "@Witness only applies to functions and type
            definitions");
    }

    @Override
    public void tcBefore(TCStatement stmt, Environment env, NameScope scope){
        name.report(6020, "@Witness only applies to functions and type
            definitions");
    }
}
```

INWitnessAnnotation

```
package annotations.in;

import com.fujitsu.vdmj.in.annotations.INAnnotation;
import com.fujitsu.vdmj.in.expressions.INExpression;
import com.fujitsu.vdmj.in.expressions.INExpressionList;
import com.fujitsu.vdmj.in.statements.INStatement;
import com.fujitsu.vdmj.runtime.Context;
import com.fujitsu.vdmj.tc.lex.TCIdentifierToken;
import com.fujitsu.vdmj.values.Value;
import com.fujitsu.vdmj.messages.Console;

import com.fujitsu.vdmj.runtime.ContextException;
import com.fujitsu.vdmj.runtime.ExceptionHandler;
import com.fujitsu.vdmj.values.BooleanValue;
import com.fujitsu.vdmj.in.expressions.INApplyExpression;
import com.fujitsu.vdmj.in.expressions.INMkTypeExpression;

public class INWitnessAnnotation extends INAnnotation{
    public INWitnessAnnotation(TCIdentifierToken name, INExpressionList args)
    {
        super(name, args);
    }

    @Override
    protected void doInit(Context ctxt){
        try{
            //If argument is a mk type expression, dealing with type definition
            if(args.get(0) instanceof INMkTypeExpression){
                //Evaluate it
                Value v = args.get(0).eval(ctxt);
            }
            //Else, dealing with function/operation
            else{
                //Evaluate the first arg
                Value v1 = args.get(0).eval(ctxt);

                //If value is a boolean value, then dealing with implicit
                    functions postc call
                if(v1 instanceof BooleanValue){
                    //Cast to boolean value
                    BooleanValue vB1 = (BooleanValue) v1;

                    //If it evaulates to false, then postc not met
```

```
                        if(vB1.value == false){
                            String msg = this + " is a bad witness.
                                Postcondition not met.";
                            ExceptionHandler.abort(name.getLocation(), 6222, msg
                                , ctxt);
                        }
                    }

                    //If there's a second arg
                    if(args.size() == 2){
                        //Evaluate the second arg
                        Value v2 = args.get(1).eval(ctxt);

                        //If value is a boolean value, then dealing with implicit
                            functions prec call
                        if(v2 instanceof BooleanValue){
                            //Cast to boolean value
                            BooleanValue vB2 = (BooleanValue) v2;

                            //If it evaulates to false, then prec not met
                            if(vB2.value == false){
                                String msg = this + " is a bad witness.
                                    Precondition not met.";
                                ExceptionHandler.abort(name.getLocation(),
                                    6223, msg, ctxt);
                            }
                        }
                        //Else, dealing with explicit function with two arguments
                        else{
                            //Compare the evaluated value to the user provided
                                value
                            if(!v1.equals(v2)){

                                String msg = this + " is a bad witness. It
                                    evaluates but to a different value than
                                    provided.\nActual: " + v1 + " Expected: " +
                                     v2;
                                ExceptionHandler.abort(name.getLocation(),
                                    6224, msg, ctxt);
                            }
                        }
                    }
                }
                //If reaches here, then no "bad witness exception" thrown,
                    therefore is a good witness
                System.out.println(this + " is a good witness");
            }catch(ContextException e){
                System.out.println(this + " is a bad witness.");
                ExceptionHandler.handle(e);
            }
        }
}
```

POWitnessAnnotation

```
package annotations.po;

import com.fujitsu.vdmj.po.annotations.POAnnotation;
import com.fujitsu.vdmj.po.definitions.PODefinition;
import com.fujitsu.vdmj.po.expressions.POExpression;
import com.fujitsu.vdmj.po.expressions.POExpressionList;
import com.fujitsu.vdmj.po.modules.POModule;
import com.fujitsu.vdmj.po.statements.POStatement;
import com.fujitsu.vdmj.pog.POContextStack;
import com.fujitsu.vdmj.pog.ProofObligationList;
import com.fujitsu.vdmj.tc.lex.TCIdentifierToken;

import com.fujitsu.vdmj.pog.ProofObligation;
import com.fujitsu.vdmj.po.definitions.POImplicitFunctionDefinition;
import com.fujitsu.vdmj.po.definitions.POImplicitOperationDefinition;
import com.fujitsu.vdmj.pog.POType;
import com.fujitsu.vdmj.po.expressions.POApplyExpression;
```

```
public class POWitnessAnnotation extends POAnnotation{
    public POWitnessAnnotation(TCIdentifierToken name, POExpressionList args){
        super(name, args);
    }

    @Override
    public void poAfter(PODefinition def, ProofObligationList obligations,
        POContextStack ctxt){
        for(ProofObligation po : obligations){
            if(po.kind.equals(POType.FUNC_SATISFIABILITY)){
                POImplicitFunctionDefinition imDef = (
                    POImplicitFunctionDefinition) def;
                String imDefResultName = imDef.result.pattern.toString();
                POApplyExpression postArg = (POApplyExpression) args.get(0);
                String postResult = postArg.args.get(args.size()-1).toString()
                    ;

                String poValue = po.value;
                poValue = poValue.replaceAll(imDefResultName + ":", postResult
                     + ":");
                poValue = poValue.replaceAll(imDefResultName + "\\)",
                    postResult +"\\)");

                po.value = poValue;
            }
            else if(po.kind.equals(POType.OP_SATISFIABILITY)){
                POImplicitOperationDefinition imDef = (
                    POImplicitOperationDefinition) def;
                if(imDef.result != null){
                    String imDefResultName = imDef.result.pattern.toString();
                    POApplyExpression postArg = (POApplyExpression) args.get
                        (0);
                    String postResult = postArg.args.get(args.size()-1).
                        toString();

                    String poValue = po.value;
                    poValue = poValue.replaceAll(imDefResultName + ":",
                        postResult + ":");
                    poValue = poValue.replaceAll(imDefResultName + "\\)",
                        postResult +"\\)");

                    po.value = poValue;
                }
            }
        }
    }
}
```