

Proving well-formedness of interface specifications

Geraldine von Roten

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

<http://sct.inf.ethz.ch/>

Fall/2007

Supervised by:

Adam Darvas

Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Contents

1	Introduction	4
1.1	Introduction to the verification system	4
1.1.1	A little example	5
1.2	Terminology	7
1.3	The problem	8
1.4	Goal	12
1.5	Work so far	13
1.6	Outline	13
2	Proving Well-formedness	13
2.1	$MS \downarrow$ and $I \downarrow$	13
2.2	A first approach: Well-formedness of method calls	15
2.2.1	Well-definedness of method calls	16
2.2.2	Well-foundedness of recursion	17
2.2.3	Extending $MS \downarrow$	20
2.3	A second approach: The Δ operator	20
2.3.1	The Δ operator	21
2.3.2	The \mathcal{D} operator	22
2.3.3	The \mathcal{T} operator	23
2.3.4	Using \mathcal{T} to prove well-formedness of specifications	26
2.3.5	\mathcal{D} vs. \mathcal{T} : A comparison	29
2.4	Relative Totalness	29
2.5	Generating axioms from pure methods for source code verification	30
3	Spec# and Boogie	30
3.1	Spec#	32
3.2	Boogie	32
3.2.1	Well-formedness proofs	35
4	Implementation	35
4.1	Extending the Boogie Pipeline	35
4.2	Introduction	36
4.3	Some preliminaries	37
4.3.1	Helper Files	37
4.3.2	The theorem prover	37
4.3.3	Result replacement	38
4.3.4	The StandardVisitor	38

4.4	The call graph	39
4.4.1	Building the graph	40
4.4.2	Checking for circles	40
4.4.3	Assigning order and picking the next node to check	40
4.5	Checking well-formedness of method calls	41
4.5.1	Implicit non-null invariants	41
4.5.2	Checking precondition preservation	41
4.5.3	Consistency	42
4.5.4	Checking well-foundedness of recursions	42
4.5.5	Extending $MS \downarrow$ and $I \downarrow$	43
4.6	Checks using the \mathcal{T} or \mathcal{D} operator	43
4.6.1	The WellFormednessChecker	44
4.6.2	The <i>CheckTorD</i> method	46
4.7	Checking consistency	48
4.8	Recursion on rep fields	48
4.9	extending $MS \downarrow$ and $I \downarrow$	48
4.10	Generating axioms for the regular proof round	49
4.11	Changes to the Spec# compiler	49
4.12	Usage	50
4.13	Known issues	51
5	Examples	51
5.1	Filtering non well-formed specifications	51
5.2	The advantage of the \mathcal{T} operator	51
6	Conclusion and future work	53
6.1	Conclusion	53
6.2	Future work	53
6.2.1	Recursion on references	53
6.2.2	Indirect recursion	54
6.2.3	Visual Studio plugin	54
6.2.4	Lexicographical ordering	54
6.2.5	Arbitrary measure clauses	54
6.2.6	Overridden methods	54

Abstract

Automated software verification systems, that try to prove the correctness of a program, use specifications in the code as a base to verify said program. When this specification includes calls to side-effect free methods, the verification system might want to use the specification of these side-effect free methods to have more information available for its correctness proofs. Unfortunately, the specifications of these side-effect free methods can be ill-formed, which can introduce unsoundness to the verification system.

For some sources of this unsoundness, like unfeasible postconditions, solutions are already found and implemented. Other ill-formedness issues, like non well-founded recursion in specifications, are known and met by syntactic rules that solve them, but are stricter than they need to be, thus disallowing a range of specifications, that are not ill-formed at all.

The goal of this thesis was to find proof obligations that prove, that a given method specification or class invariant is well-formed and thus safe to use for correctness proofs. These proof obligations were then implemented in the automated software verification system Boogie to filter out non well-formed specifications.

1 Introduction

1.1 Introduction to the verification system

All theoretical work in this thesis was done with the intent of it being used with an automated software verification system. A verification system takes the code of a program and statically analyzes it, to determine whether the program works correctly with respect to its specification and also to make sure that no runtime errors will occur.

The specification that the verification systems are using, is connected very tightly to the program code. It appears as method contracts or invariants. Invariants are bound to classes and specify the states of their data. A typical invariant could, for example, say, that a field called *length* is always positive. Method contracts on the other hand are, as their name proposes, bound to methods. They are split into two parts, the precondition and the postcondition. The precondition specifies what the method expects, when it is being called (e.g. that a certain parameter should not be *null*), the postcondition specifies the state the method returns in. The postcondition can, for example, say something about the return value of the method. See figure 1 for an example.

The implementation part of the thesis was done in the Spec# verification system. This verification system was developed by Microsoft Research. It consists of the following two parts:

Spec# [5] is a programming language that uses C# [12] as its base and adds features for method contracts, class invariants and several other things. The Spec# compiler produces CIL (Common Intermediate Language) - which is the byte code for the .NET virtual machine - but, unlike the regular C# compiler, it also includes the contract information that was added to the Spec# program. Spec# can enforce its contracts dynamically, if used without a static verifier.

The second part of the verification system, called Boogie [3], is a program verifier, that takes the CIL program created by the Spec# compiler and attempts to statically prove the correctness of the implementation of all methods in the program with respect to their specifications, using an automated theorem prover. Even though Boogie uses special invariant semantics, this thesis only treats cases, where Boogie follows the usual visible state semantics. For the visible state semantics, Boogie assumes method A to be implemented correctly, if - among other things - the following is true:

- Whenever method A calls any method B, the invariants and the preconditions of method B hold.
- Assuming the precondition of A and all class invariants hold, when the method is called, the postcondition of A and all class invariants hold, when the method returns.

1.1.1 A little example

This section will illustrate, what happens, when a program is fed to Boogie, using a class called *IntegerArray* as an example. An implementation of *IntegerArray* can be seen in figure 1.

The **requires** keyword specifies, what a method assumes to be true, when it is called, i.e. the precondition. In the case of *Add*, the length of the *IntegerArray* needs to be less than the return value of the method *Capacity* and the return value of *Capacity* must be less than the biggest integer. *Capacity* can be used in the specification, because it is marked with the [Pure] attribute and is therefore proved to be side-effect free. The **ensures** keyword on the other hand denotes the postcondition of a method. Note the keyword **result** in the postcondition of *Capacity*: It stands for the return value of the method. **old(length)** stands for the value of *length* before the method was called.

```

class IntegerArray{
  private int length;
  private int capacity = 10;

  invariant length <= capacity;

  IntegerArray()
  {
    length = 0;
  }

  [Pure]public int Capacity()
    ensures result == capacity;
  {
    return capacity;
  }

  void Add(int arg)
    requires length < Capacity();
    requires Capacity() < int.MaxValue;
    ensures length == 1 + old(length);
  {
    length = length + 1;
    //plus more implementation..
  }

  ....
}

```

Figure 1:

We will now take a closer look at the method *Add*: As mentioned above, Boogie needs to prove, that the implementation of *Add* makes sure that the postcondition holds at the end of the method, if the precondition and the invariant hold, when it is called. Proving, that the invariant still holds at the end of *Add* is easy, since it is trivial to see, that adding 1 to `length` still makes it less or equal to `capacity`, if we assume that `length` was less then `capacity` to begin with. Proving that `length == 1 + old(length)` from the implementation, is simple as well. Additionally, for every method that calls *Add*, Boogie will have to prove that, at the point of the call,

`length < Capacity()` and `Capacity < int.MaxValue`.

Basic information, that is the same for every program, like the value of `int.MaxValue`, is stored in a big collection of axioms called the *background predicate*. All axioms in the background predicate are blindly assumed to be true by Boogie, for every proof obligation it verifies. Therefore one must be very careful about what is added to this background predicate.

1.2 Terminology

This section defines some terms, that are used often throughout this thesis, to describe properties of invariants, method specifications, or method calls:

Consistency A method or invariant is consistent, if its specification does not contradict the specifications of any other method or invariant or its own. For simplicity, a method is often called consistent in this paper, when really it is its specification that is consistent.

Well-definedness A specification *spec* is well-defined, if, for all method calls in *spec*, the precondition of the callee is preserved. For method specifications, well-definedness also includes consistency.

Well-foundedness Well-foundedness is a possible property of recursive method calls. Proving recursive method calls to be well-founded is one of the goals of this thesis. For simplicity, a method is often called well-founded in this thesis, when all recursive calls in its specification are well-founded. A method is automatically well-founded, if its specification does not contain a recursive call. Invariants cannot contain recursive calls, because invariants cannot be called. Therefore they are always well-founded.

Well-formedness A specification *spec* is well-formed, if it is well-defined, and well-founded.

1.3 The problem

In the example in figure 1, the specification of the method *Add* contains a call to the the method *Capacity*. This way, the specification of *Add* is independent from the computation of the capacity of the Array and unaffected by changes thereof.

To be able to prove that `length < Capacity()`, the verification system needs to know, the return value of the method *Capacity*. In this case, it needs to know that *Capacity* is simply a getter method for the field *capacity*. This information is stored as an axiom in the background predicate of the verification system. The axiom added to the background predicate for method M would look like

$$P \wedge I_M \Rightarrow Q$$

where P is the precondition of M, Q is the postcondition of M and I_M are all well-formed class invariants that do not depend on M, i.e. that do not call M either directly or indirectly. If not specified, P, Q and I_M are *true*.

In Boogie, the background predicate is stored in an intermediate language called BoogiePL (see section 3 for more information) that has no notion of the heap. Therefore, the heap of each program state is modeled as a two dimensional array in which the first index denotes the object and the second one the field to access. Describing the access to `x.i` in heap *h* would then look like this: $h[x, i]$. The axiom for *Capacity* would thus be (simplified for clarity)

$$\forall o : IntegerArray, h : heap \bullet o! = null \wedge h[o, length] \leq h[o, capacity] \Rightarrow \#Capacity(o, h) == h[o, capacity]$$

The condition $o \neq null$ is a default precondition for all non-static methods. $\#Capacity$ is a function symbol that models the pure method *Capacity*. These function symbols of pure methods take the arguments of the methods they model, plus one argument for the receiver object and one for the heap at the time they are called. *o* and *h* are this receiver object and the heap parameter. When generating the axiom for a method specification, the literal *result* in the specification gets replaced by the function symbol of that method.

Unfortunately, blindly generating these axioms from the specifications of all pure methods can introduce unsoundness to the verification system, as the following example will show:

```

[Pure]int Capacity()
  ensures result == capacity && result == capacity + 1;
{
  return capacity;
}

```

The axiom generated from this method would look like this

$$\forall o : IntegerArray, h : heap \bullet o \neq null \wedge h[o, length] \leq h[o, capacity] \Rightarrow \\ \#Capacity(o, h) == h[o, capacity] \wedge \\ \#Capacity(o, h) == h[o, capacity] + 1$$

Because the two postconditions contradict each other, this can be reduced to

$$\forall o : IntegerArray, h : heap \bullet o \neq null \wedge h[o, length] \leq h[o, capacity] \Rightarrow \\ \text{false}$$

When instantiating this forall quantification with an object of type IntegerArray called *someIntArray* for *o*, we get

$$someIntArray \neq null \wedge h[o, length] \leq h[o, capacity] \Rightarrow \text{false}$$

If instantiated with a non-null object of type IntegerArray and a heap, where $length \leq capacity$, the axiom yields just *false*. Having an axiom, that can be instantiated to yield *false*, in the background predicate makes it possible to prove anything to be *true*, even *false* itself (remember, *false* implies anything and the axioms in the background predicate can be used anywhere in the proof process).

Unattainable postconditions, like in the example above, are a source of unsoundness that can be eliminated by making sure that there actually is an expression that can satisfy the postcondition, a so called witness expression (which can't be found for the above example, since nothing can be `capacity` and `capacity + 1` at the same time). Finding witness expressions for pure methods and not generating axioms from method specifications that don't have a witness is already implemented in Boogie. [8].

Another important class of possibly non well-formed specifications and the focus of this thesis are method specifications that are recursive. A good example for a recursive specification is the computation of the factorial of an integer, as shown in figure 2.

The above specification and implementation of *fact* is correct. One can run into problems with recursive specifications though, if, for example, the

```

class Fact{
  [Measure("p")] [Pure] int fact(int p)
    requires p >= 0;
    ensures p == 0 ==> result == 1;
    ensures p > 0 ==> result == fact(p-1)*p;
  {
    if(p == 0)
      return 1;
    else
      return fact(p-1)*p;
  }
}

```

Figure 2:

measure of the recursion is not decreasing. Note the **Measure** attribute added to *fact*. It specifies this decreasing measure of the recursion. The measure can be any integer expression that decreases with every step of the recursion, to prove, that the recursion is not cyclic. In the *fact* example above, one way of not having the measure decrease would for example be, that the ensures clause that contains the recursive call is replaced with the following:

ensures (p > 0) ==> **result** == fact(p) * p

Axiomatizing that ensures clause would give us the following axiom:

$$\forall o : Fact, h : heap, p : int \bullet o \neq null \wedge p \geq 0 \wedge p > 0 \Rightarrow \\ \#fact(o, h, p) == \#fact(o, h, p) * p$$

This axiom by itself does not do any harm yet. But let's have a look at the method in figure 3. Since the body of *falseMethod* is empty and there are no class invariants, the verification system, would, using weakest precondition calculus, generate and try to prove a proof obligation for the correctness of that method, which says

$$P \Rightarrow Q$$

(The precondition must imply the postcondition).

For *falseMethod* this proof obligation looks like this:

$$p == 3 \wedge \#fact(this, currentHeap, p) == 6 \Rightarrow false$$

Of course, this proof obligation should not pass the theorem prover, because there is no way, a method can satisfy the postcondition *false*. However, there

```

void falseMethod(int p)
    requires p == 3;
    requires fact(p) == 6;
    ensures false;
{
    ...
}

```

Figure 3:

is the axiom for the ill-defined *fact* in the background predicate. If this axiom gets instantiated with *this* (which is never null) and heap *currentHeap* we get:

$$\begin{aligned}
 p \geq 0 \wedge p > 0 &\Rightarrow && \#fact(this, currentHeap, p) == \\
 \#fact(this, currentHeap, p) * p &
 \end{aligned}$$

When adding this axiom to the proof obligation for *falseMethod*, the left-hand side of the implication in the axiom is true (since $p == 3$) and therefore only the right-hand side is needed. The resulting proof obligation looks like this:

$$\begin{aligned}
 \#fact(this, currentHeap, p) == \#fact(this, currentHeap, p) * p \wedge \\
 p == 3 \wedge \#fact(this, h, p) == 6 &\Rightarrow \\
 \mathbf{false} &
 \end{aligned}$$

Note, that the first row comes from the axiom generated from *fact*, the second row comes from the precondition of *falseMethod* and the third row is the postcondition of *falseMethod*.

Since $\#fact(this, h, p)$ is not zero and p is not 1, the part that came from the specification of *fact* can be divided by $\#fact(this, h, p)$ to become $1 == p$, thus resulting in the proof obligation being:

$$1 == p \wedge p == 3 \wedge \#fact(this, h, p) == 6 \Rightarrow \mathbf{false}$$

Since $1 == p$ and $p == 3$ contradict each other, the proof obligation can be reduced to

$$\mathbf{false} \Rightarrow \mathbf{false}$$

which is trivially true. The non well-foundedness of method *fact* thus led to proving that *falseMethod* will exit in a state, where *false* (or anything else that appears in the ensures clause of *falseMethod*), is *true* which is obviously unsound. For the correct specification of *fact*, the division by

$\#fact(this, h, p)$ would not be possible, because the part of the axiom added to the proof obligation of *falseMethod* would be:

$$\#fact(this, currentHeap, p) == \#fact(this, currentHeap, p - 1) * p$$

1.4 Goal

The goal of this thesis is to find the proof obligations an automated software verification system needs, to prove the well-formedness of class invariants and method specifications. These proof obligations are necessary, because non well-formed specifications can lead to unsoundness in the verification system, when they are axiomatized. The second part of the thesis consists of the implementation of those proof obligations in the software verification system Boogie. The main focus is on proving the well-foundedness of recursively specified methods, which is a part of well-formedness.

Since these proof obligations need to be handled by an automated verification system, certain restrictions apply to the syntax of specifications, to make them easier to prove for an automated verification system.

- Only direct recursion can be handled because the proof obligations for indirect recursion are too big to be handled by today's automated theorem provers.
- Because of the problem of defining measures over structures on the heap and proving their decreasing, only recursion on primitive types can be handled, but not recursion on reference types.
- No recursive calls are allowed in preconditions because they would lead to circular reasoning.
- All postconditions of methods must be of the form $Q'_i \Rightarrow Q''_i$. To allow automated theorem provers to find a witness expression for the postcondition, Q''_i must be of form `result :: someExpr`, where `::` is a reflexive operator like `==` or `>=`. Otherwise, finding a witness for the postcondition to prove its feasibility involves an existential operator which, is difficult to handle for automated theorem provers. Such a postcondition could look like this:

```
ensures p > 0 ==> result == 7
```

where `p > 0` is Q'_i , `result == 7` is Q''_i and `7` is *someExpr*. Any recursive calls can only occur in `someExpr`, because recursive calls in Q'_i would lead to circular reasoning.

1.5 Work so far

The problem of non well-formed specifications is not a new discovery. In older verifying systems the problem was addressed, by not axiomatizing pure methods or introducing very strict rules, e.g. totally disallowing recursive specifications. Modern, object oriented verifying systems like provers for JML [11] on the other hand usually ignore the problem of non well-founded recursion completely.

Boogie allows recursion on reference types, if the reference is a *rep* field, since the *rep* relationship cannot have circles (i.e. is always decreasing). See [7].

1.6 Outline

First we discuss the theoretical solutions to the problems explained above in section 2 with subsection 2.2 about well-formedness of method calls and subsection 2.3 about well-formedness in general. Then Boogie is presented in more detail in section 3 and the implementation is discussed in section 4. Finally, we show some examples in section 5 and conclude in section 6.

2 Proving Well-formedness

Up till now, most software verification systems for modern, object oriented languages, decided on their knowledge about a program (the background predicate) once, and then generated proof obligations for the correctness of methods and tried to prove them using the background knowledge they decided on earlier. When axiomatizing pure methods and using them in the background predicate to prove the correctness of a program, additional steps of proving the well-formedness of the specifications of those pure methods have to be inserted *before* the regular prove step. Only once their well-formedness is proved, can their axioms be added to the background predicate and used for correctness proofs. These well-formedness checks can also be applied to non-pure methods to make it easier to find errors in the specifications. Non well-formedness of non-pure methods does not introduce unsoundness though, because no axioms are generated from the specifications of non-pure methods.

2.1 $MS \downarrow$ and $I \downarrow$

The specification of any method M or invariant I can contain calls to pure methods. The specifications of these pure methods can again call other pure

methods. To avoid cyclic reasoning, the well-formedness of all methods, that specification S (either a method or an invariant) calls directly or indirectly, must be proved, before the well-formedness of specification S can be proved. Then, all axioms of methods and all invariants, that don't call S directly or indirectly, can be used to prove the well-formedness of S . This set of axioms is called MS_S , the set of invariants is called I_S . Using MS_S and I_S results in a proof round for each method and invariant in a program.

In this thesis, MS_S , I_S and the correct order of well-formedness proofs are realized with a call graph. The call graph is a forest of trees and contains nodes that all represent either a method or an invariant of the program that is being proved. An edge from node A to node B indicates, that the invariant or the specification of the method represented by node A contains a call to the method represented by node B . Since only pure methods can be called in specifications or invariants, edges will always point to a pure method. By always picking a node, whose children (if existing) are proved to be well-formed already, one makes sure, that the nodes are chosen in a correct order. In the call graph, the sets of method specifications and axioms, that are independent from the specification that is being proved, are called $MS \downarrow$ and $I \downarrow$. Whenever the well-formedness of the specification of a node is proved successfully, either an axiom is added to $MS \downarrow$, or an invariant is added to $I \downarrow$. $MS \downarrow$ and $I \downarrow$ are not an exact realization of MS_S and I_S , since they do not guarantee, that, when the well-formedness of node A is proved, $MS \downarrow$ and $I \downarrow$ contain all axioms and invariants that could be in there, since methods and invariants in other trees of the call graph are all independent of node A , but might not have been proved to be well-formed yet. $MS \downarrow$ and $I \downarrow$ are used extensively in the proof obligations for well-formedness.

$MS \downarrow$ appears in the proof obligations in this paper, but when actually implemented, the axioms for well-formed specifications of pure methods go directly to the background predicate of the verification system and do not appear in the proof obligations explicitly.

Another important task of the the call graph is, to find any existing recursions in the specifications. These recursions form circles in the trees of the call graph. Even though, strictly speaking, trees are not allowed to have circles at all, in our approach of this call graph, direct circles are allowed, while indirect circles are disallowed and lead to all involved methods being marked as not well-formed. A direct circle is a node with an edge pointing to itself (i.e. a method that calls itself), an indirect circle is a circle with more than one involved nodes. Figure 4 shows a simple example of a call graph. When proving the well-formedness of *CommandLineIsCorrect*, the well-formedness proofs for the specification of *FilesPresent* are already evaluated, and an axiom from that specification was added to $MS \downarrow$, if

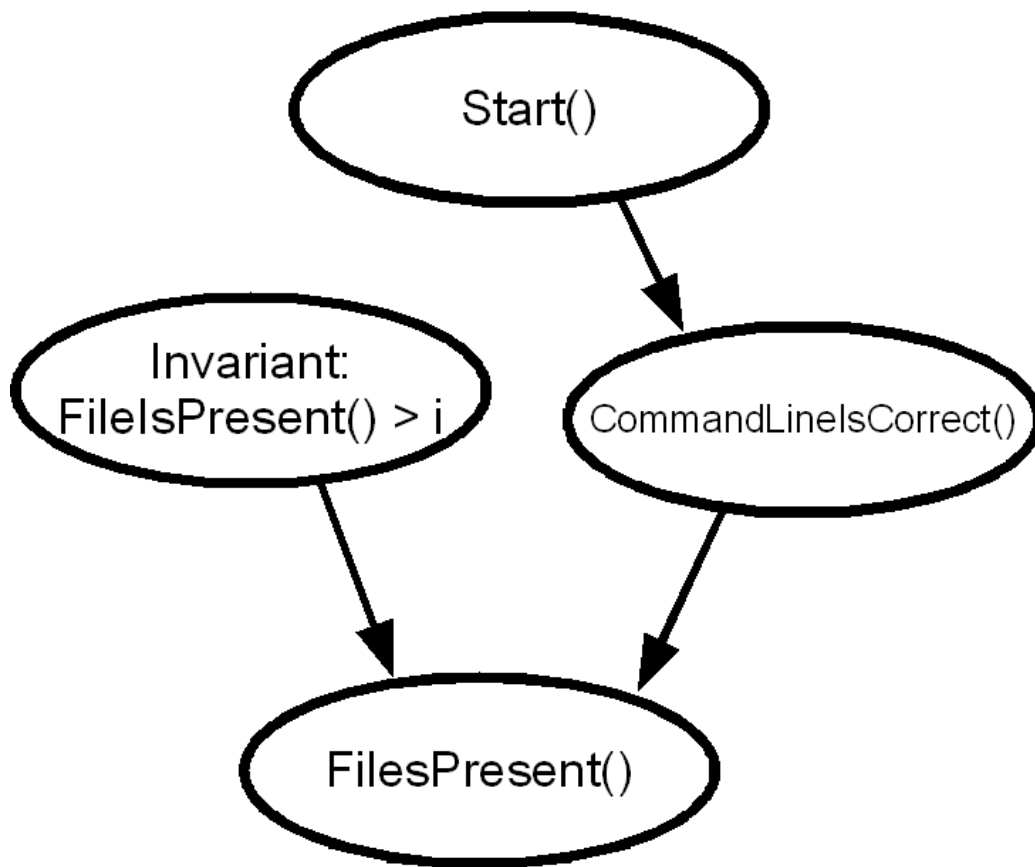


Figure 4:

appropriate. Assuming the invariant is well-formed, it might already be in $I \downarrow$, depending on whether the invariant or *CommandLineIsCorrect* was picked for proving its well-formedness first. There is no rule, which of them should be picked first.

The next 2 sections describe two approaches on how the well-formedness of the individual call graph nodes is proved. The main focus of this thesis is the non well-formedness that is introduced to specifications by the non well-foundedness of recursively specified methods. Hence, as a first step, the proof obligations to prove the well-formedness, which includes well-foundedness, of methods is explained.

2.2 A first approach: Well-formedness of method calls

As a first approach, proof obligations to prove the well-formedness of method calls were developed. This cannot only be used for actual method calls, since other sources of non well-formedness, like field accesses can be seen as method

calls with special preconditions. For field accesses, the precondition is, that the receiver must not be null. For simplicity reasons, we will only talk about actual method calls.

Assume a call graph, as described in section 2.1, including all methods and invariants in a program, was built. We will also assume, that there are no indirect circles in the graph. For each node in the graph, the well-definedness of the specification and the well-foundedness for existing recursive calls is proved. These two things together make up well-formedness. After a small terminology the necessary proofs are explained.

P	The precondition of the method of the current node.
Q_i	The i th postcondition of the method of the current node. Because of our syntactic restrictions, it looks like this: $Q'_i \Rightarrow Q''_i$
I	The invariant of the current node.
P_n	The precondition of the callee of the current call, in which formal parameters are replaced by the actual parameters of the current call.
Q'	The postcondition of the current method, in which <i>result</i> is replaced by the signature of the method.

2.2.1 Well-definedness of method calls

A method call is well-defined, if the preconditions of the callee are preserved. For a specification to be well-defined, all occurring method calls must be well-defined and it must be consistent.

For invariants For invariants, the verification system proves, that, for every method call in the invariant, the precondition of the called method holds. The proof obligations then looks like this:

$$MS \downarrow \Rightarrow (I \downarrow \Rightarrow P_n)$$

Additionally, proof obligations stating, that the invariant is consistent (i.e. does not contradict the rest of the specification) could be added. Unfortunately, this involves an existential quantifier, and can therefore not be proved automatically. Because inconsistent invariants don't introduce unsoundness, since they are not added to the background predicate, this is not a problem.

When all method calls in the invariant are found to be well-formed, the invariant is added with to $I \downarrow$.

For methods The proofs done for methods are a little more complicated than the ones done for invariants. Like with method calls in invariants, the verification system needs to prove, that the precondition of the callee of each method call in the specification of a method holds. However, depending on whether the method call appears in P , Q'_i or Q''_i , the proof obligations look a little bit different.

For method calls in P :

$$MS \downarrow (I \downarrow \Rightarrow P_n)$$

For method calls in Q'_i

$$MS \downarrow (I \downarrow \wedge P \Rightarrow P_n)$$

For method calls in Q''_i

$$MS \downarrow (I \downarrow \wedge P \wedge Q'_i \Rightarrow P_n)$$

This means, that, for method calls in the Postcondition, the Precondition can be assumed, and for calls in Q''_i , Q'_i can be assumed.

Unlike invariants, for methods, an axiom is added to the background predicate. This means, that their consistency needs to be proven to avoid introducing unsoundness. Since the syntactic restrictions introduced by our approach, force every postcondition of a pure method to only express a property of the return value of said method, axioms of two different methods cannot contradict each other. Therefore it is sufficient to proof, that a method specification does not contradict itself. Because all postconditions are of the form $Q'_i \Rightarrow Q''_i$, and the property of the return value is in Q''_i , proving the consistency of a method means proving, that all Q'_i s are mutually exclusive. The proof obligation is

$$MS \downarrow \Rightarrow (I \downarrow \wedge P \Rightarrow \neg((Q'_1 \wedge Q'_2) \vee \dots \vee (Q'_{k-1} \wedge Q'_k)))$$

At this point, the well-definedness of the method specification is proved. Other than for invariants, for method specifications, the well-foundedness needs to be proved as well, so that the axiom for the method cannot be added to $MS \downarrow$ yet.

2.2.2 Well-foundedness of recursion

For all recursive calls in a method specification, it's well-foundedness needs to be proven.

The theory of well-founded recursion A relation R is called well-founded on a set X , iff there are no infinite descending chains. This means, that there is no infinite sequence

$$x_0 R x_1 R x_2 R \dots R x_n$$

A recursion can be defined as follows:

$$f(\vec{x}) = E(f(\vec{x}_1))$$

The important thing about this definition is, that f of \vec{x} is defined by an expression, that itself contains f of some \vec{x}_1 . To prove the well-foundedness of this recursion, one has to prove, that there is a well-founded relation R between \vec{x} and \vec{x}_1 .

For our approach, we want R to be the $<$ operator on the natural numbers, which is a well known well-founded relation. Therefore, we must prove, that \vec{x}_1 is indeed smaller than \vec{x} . Additionally, we need to prove, that \vec{x}_1 stays in the natural numbers. (This is necessary, because in programming we work with integers and not with natural numbers.)

Because parameters of methods are not restricted to the natural numbers and because not all parameters in the vectors \vec{x} and \vec{x}_1 need to decrease, every recursive method must define a so called *measure*. The well-foundedness of the relation between the *measures* of two recursion steps is then proved. The *measure* can simply be one of the parameters or some mathematical expression that uses one or more parameters.

The next two paragraphs will explain, what proof obligations emerge, when moving the rules found in this paragraph to the world of software verification.

Recursion on integers Applying the rules of well-founded recursion to recursion on primitive types (in this case integers) is pretty much straightforward. The specification of method *fact* in figure 2 will be used as an example.

The first thing to be proved for the well-foundedness of a recursive call is, that its measure is decreasing which each step of the recursion. One has to prove that the measure of the recursive call is smaller than the original measure. The proof obligation is

$$MS \downarrow \Rightarrow (I \downarrow \wedge P \wedge Q'_i \Rightarrow E > E')$$

P and Q'_i can be used on the left-hand side of the implication, because the syntactic restrictions force all recursive calls to be in the Q''_i of the postcondition. In the case of the *fact* example, one has to prove that:

$$MS \downarrow \Rightarrow (I \downarrow \wedge p \geq 0 \wedge p > 0 \Rightarrow p > p - 1)$$

```

class IntegerList
{
    IntegerList next;
    int value;

    [Pure] int length()
        ensures next == null ==> result == 1;
        ensures next != null ==> result == next.length() + 1;
}

```

Figure 5:

The second thing needed to prove the well-foundedness of a recursive call is, that the recursion will eventually terminate. One has to prove, that the measure of the recursive call is always bigger than some integer. In this approach we chose the value to be 0, i.e. we want the measure to stay in the natural numbers, but it could be any other number as well. Depending on the chosen value, the measure might have to be different though. The proof obligation for recursion termination is

$$MS \downarrow \Rightarrow (I \downarrow \wedge P \wedge Q'_i \Rightarrow E' \geq 0)$$

In the case of the *fact* example above, one has to prove that:

$$MS \downarrow \Rightarrow (I \downarrow \wedge p \geq 0 \wedge p > 0 \Rightarrow p - 1 \geq 0)$$

Recursion on references A popular example for a recursive method in which the recursion works on a reference, is method *length* that returns the length of a linked list as shown in figure 5. The recursive parameter of *length* is the implicit receiver parameter.

The two things that are needed for the well-foundedness of this recursion are the same as for recursions on primitive types: A decreasing measure and a terminating recursion. In the case of *length*, a possible case of non well-foundedness would be a cyclic list, since this would lead to a cyclic (and thus not decreasing) recursion. For example, an object of type *IntegerList*, where *next* is equal to *this*, would lead to an axiom that could be instantiated to say that

$$\#length(this, currentHeap) == \#length(this, currentHeap) + 1$$

which can be reduced to $0 == 1$.

If the list is indeed acyclic, the termination of the recursion can be proved, because the acyclicity proof yields a proof that *emph null* is eventually reached by the recursion, and the first postcondition of *length* states, that the recursion terminates at that point

To prove the acyclicity of object structures like linked lists, the verification system would need to know some kind of structural information about the object structure and also complete information on aliasing. Unfortunately, proving this is too difficult for today’s automated theorem provers.

One special case of recursion over reference types are recursions, where the recursive receiver parameter is a rep field of the object the method is called on. Since chains of rep relations are defined to be acyclic, this kind of recursion on references is always well-founded. This was already implemented in Spec# for [7].

2.2.3 Extending $MS \downarrow$

If all proof obligations for well-definedness, consistency and well-foundedness for a method specification are found to be valid, an axiom is generated out of the method specification and added to $MS \downarrow$. The axiom for a method looks like this:

$$I \downarrow \wedge P \Rightarrow Q'$$

For invariants, extending $I \downarrow$ already happened after proving the well-definedness of the invariants, since there are no recursive calls in an invariant.

2.3 A second approach: The Δ operator

While the well-formedness proofs shown in the last section do guarantee, that no unsound axioms are generated, they are stricter than they need to be. This means, that they disallow a number of specifications that would not introduce unsoundness to the verification system. The problem with the well-formedness proofs in the last section is, that they require the well-formedness of every single method call in a specification, while there might be method calls whose well-formedness is not required.

Let’s have a look at the following precondition:

```
requires true || SquareRoot(-1) > 6
```

where *SquareRoot* has a precondition saying, that the parameter needs to be greater or equal than zero. The call to *SquareRoot* is not well-formed, since its precondition does not hold. But in this case, the well-formedness of the call to *SquareRoot* is not necessary, since the whole expression is *true*

anyway, and the second part of this OR is never evaluated, because it has no influence on the value of the expression. Using the well-formedness proofs from the last section, the specifications would not pass because of the illegal parameter though.

Proving, that an expression is well-formed as a whole, like we would want to do to the precondition above, is not a new problem and there are many different techniques to handle this. More information on these different techniques can be found at [13].

Proving the well-formedness of a (boolean) expression can be looked at as proving, that the expression will always yield either *true* or *false*. In regular, two-valued logic, this is trivially true. Expressions in program specifications work in three-valued logic though. The third possible result of such an expression is called *bottom* and is represented by the symbol \perp . An expression yields \perp , if it cannot be evaluated. This could, for example, be a null-pointer exception or a precondition violation. The precondition

```
requires SquareRoot(-1) > 6
```

would yield \perp , since it cannot be evaluated because of the precondition violation of *SquareRoot*. The precondition

```
requires true || SquareRoot(-1) > 6
```

on the other hand, yields *true*, because it can be evaluated, even though the same precondition violation occurs. You can see from those two examples, that the well-formedness proofs from section 2.2 are still needed, but not for all cases.

Two of the most popular techniques to solve the problem of the three-valued logic are

- Assigning an unknown but legal value of correct type to non well-formed expressions. This could mean, that *SquareRoot(-1)* is assumed to return an integer. While we don't know this integer, we know, that *SquareRoot(-1) == SquareRoot(-1)*
- Proving, for every expression, that \perp will never occur.

This thesis uses the second approach.

2.3.1 The Δ operator

The Δ operator [1] can be applied to any formula in three-valued logic. The truth table of Δ for formula ϕ looks as follows:

$$\begin{aligned}
\mathcal{D}(P(e_1, \dots, e_n)) &= \bigwedge_{i=1}^n \Delta_e(e_i) \\
\mathcal{D}(\text{true}) &= \mathbf{true} \\
\mathcal{D}(\text{false}) &= \mathbf{true} \\
\mathcal{D}(\neg\phi) &= \mathcal{D}(\phi) \\
\mathcal{D}(\phi_1 \wedge \phi_2) &= (\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2)) \vee (\mathcal{D}(\phi_1) \wedge \neg\phi_1) \vee (\mathcal{D}(\phi_2) \wedge \neg\phi_2) \\
\mathcal{D}(\phi_1 \vee \phi_2) &= (\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2)) \vee (\mathcal{D}(\phi_1) \wedge \phi_1) \vee (\mathcal{D}(\phi_2) \wedge \phi_2) \\
\mathcal{D}(\forall x. \phi) &= \forall x. \mathcal{D}(\phi) \vee \exists x. (\mathcal{D}(\phi) \wedge \neg\phi) \\
\mathcal{D}(\exists x. \phi) &= \forall x. \mathcal{D}(\phi) \vee \exists x. (\mathcal{D}(\phi) \wedge \phi)
\end{aligned}$$

Figure 6:

ϕ	$\Delta(\phi)$
true	true
false	true
\perp	false

Δ returns *true*, when the formula yields either *true* or *false* and it returns *false*, when the formula yields \perp (in our case \perp means, the expression is not well-formed). Note, that the output of Δ is in two-valued logic. This means, that applying Δ to any formula, transforms it from three-valued to two-valued logic in exactly the way we want to transform the expressions in our specifications.

While Δ only exists in theory, the next sections will talk about two special realizations of the Δ operator called \mathcal{D} and \mathcal{T} . \mathcal{D} is an already existing realization of Δ , while \mathcal{T} was developed for this thesis. When applying either of these two operators to an expression, they return a new expression, that is in two-valued logic. This new expression can then be fed to a theorem prover. It will evaluate to *true*, if the original expression was well-formed (i.e. would have evaluated to either *true* or *false*) and to *false* otherwise.

2.3.2 The \mathcal{D} operator

An older implementation of the Δ operator is \mathcal{D} [6]. The definition for \mathcal{D} is shown in figure 6.

A definition for Δ_e can be found in figure 9.

2.3.3 The \mathcal{T} operator

The truth table for \mathcal{T} is

ϕ	$\mathcal{T}(\phi)$
true	true
false	false
\perp	false

It returns an expression that yields *true* if ϕ is true and yields *false*, if ϕ is either *false* or \perp . Unlike other implementations of Δ , \mathcal{T} does not implement Δ directly, but one can see that $\mathcal{T}(\phi \vee \neg\phi)$ is equivalent to $\Delta(\phi)$

The syntax Figure 8 shows the syntax that will be used to define \mathcal{T} . The most significant distinction is the one between Terms and Formulas. A term is either a variable or a method call. Like in the first approach, also field accesses and built-in mathematical operators are seen as method calls. Formulas are either predicates, *true*, *false* or formulas connected by logical operators. In the expression

$$true \vee f(a, b) > c$$

a , b , c and $f(a, b)$ are terms, $f(a, b) > c$ is a predicate (and therefor also a formula) and *true* and the whole expression are formulas. There are two important things to note about this syntax:

- Any expression, that is the argument of a method call, is considered a term. Even when that expression contains a logical operator.
- Predicates are built over terms. Boolean terms (e.g a boolean variable) can appear as a formula without an explicit predicate around them though. To allow this, there is a dummy predicate, that just returns the value the term is surrounds.

The Δ_e operator To prove the well-definedness and well-foundedness of terms, \mathcal{T} uses the Δ_e operator (See figure 9. Δ_e of a variable is always *true*, since a variable is always well-defined and well-founded. Δ_e of a method call is *true* iff Δ_e of all parameters of the method call are *true* and if the preconditions of the called methods hold. In our case the precondition includes

- The explicit precondition.

```

class Circle
{
    int minSize;
    int maxSize;
    bool Comparable;

    [Pure] int getSize()
    {
        ...
    }

    [Pure] bool IsSmallerThan(int size)
    requires this.Comparable;
    {
        ...
    }
    bool EnlargeTo(int size)
    requires size > this.getSize();

    void ExchangeSizes(Circle circle)
    requires circle.getSize > this.minSize &&
           circle.getSize() < this.maxSize &&
           this.getSize() > circle.minSize &&
           this.getSize() < circle.maxSize;
    {
        ...
    }
    void CoverWithCircleOfSize(int size)
    requires this.isSmallerThan(size);
    {
        ...
    }
}

```

Figure 7:

<i>Term</i>	::=	<i>Var</i>		<i>Formula</i>	::=	$P(t_1, \dots, t_n)$
		$f(t_1, \dots, t_n)$				<i>true</i> <i>false</i>
						$\neg\phi$
						$\phi_1 \wedge \phi_2$
						$\phi_1 \vee \phi_2$
						$\forall x. \phi$
						$\exists x. \phi$

Figure 8: Standard syntax of terms and formulas.

$\Delta_e(\text{Var})$	=	true
$\Delta_e(f(e_1, \dots, e_n))$	=	$d_f(e_1, \dots, e_n) \wedge \bigwedge_{i=1}^n \Delta_e(e_i)$

Figure 9: The Δ_e operator

- The receiver cannot be null for non-static calls.
- The well-foundedness of the recursion, if the method is specified recursively.

there is no explicit precondition, if the method call is a field accesses, . Built in operators don't have a receiver object, but some of them do have preconditions. One example for such a precondition would be, that the divisor of an division cannot be zero.

One can see, that the second point is the well-definedness proof and the third point the well-foundedness proof from section 2.2. It is important to see though, that the consistency of methods is not checked by Δ_e , or indeed by \mathcal{T} . It cannot be checked by these operators, because they operate on expressions, while consistency has to be proved for the postcondition of a whole method, which can consists of several expressions.

To show a little example of how Δ_e is used , we will now apply $\Delta_e()$ to the precondition of method *CoverWithCircleOfSize* in class *Circle* in figure 7

$$\Delta_e(\text{this.isSmallerThan}(\text{size}))$$

which gives us, when applying the definition of Δ_e

$$Comparable \wedge this \neq null \wedge \Delta_e(size)$$

Since *size* is a variable, $\Delta_e(size)$ is true. Also, *this* is never *null*. The precondition is therefor well defined, if *Comparable* is true.

The \mathcal{T} operator The definition of \mathcal{T} can be seen in figure 10. When keeping in mind, that \mathcal{T} does not implement Δ directly, but yields *true* if and only if the three-valued expression passed to it evaluates to *true*, this definition is pretty much straight forward for most cases. There two interesting cases though:

- $\mathcal{T}(\phi)$ where ϕ is a Predicate: A predicate P is an expression of type *boolean*. $\mathcal{T}(P(e_1, \dots, e_n))$ is true, iff the predicate yields true and also Δ_e of each term in the predicate is true. An example for a predicate is the precondition

$$size > this.getSize()$$

of method *EnlargeTo* in figure 7 . When applying the definition for \mathcal{T} , we see, that

$$\mathcal{T}(size > this.getSize())$$

is

$$size > this.getSize() \wedge \Delta_e(size) \wedge \Delta_e(this.getSize())$$

$\Delta_e(size)$ is *true* and $\Delta_e(this.getSize())$ is *this \neq null*, which is *true* as well.

- $\mathcal{T}(\neg\phi)$: \mathcal{T} of a negated Predicate is *true*, iff the negation of that formula is *true* and Δ_e of all terms in the predicate is *true*. *true* and *false* get swapped, when they are negated. For the rest of the definition of \mathcal{T} of a negated formula, de Morgans law was simply applied.

2.3.4 Using \mathcal{T} to prove well-formedness of specifications

After all this theory on how the \mathcal{T} operator works, this section will explain, how it can be used to prove the well-formedness of method specifications and invariants. Two important things to remember are:

- A call graph, as described in section 2.1, was built and the well-formedness of the individual nodes must now be proven.

$$\begin{aligned}
\mathcal{T}(P(e_1, \dots, e_n)) &= P(e_1, \dots, e_n) \wedge \bigwedge_{i=1}^n \Delta_e(e_i) \\
\mathcal{T}(true) &= \mathbf{true} \\
\mathcal{T}(false) &= \mathbf{false} \\
\mathcal{T}(\neg\phi) &= \begin{cases} \neg P(e_1, \dots, e_n) \bigwedge_{i=1}^n \Delta_e(e_i) & \text{if } \phi = P(e_1, \dots, e_n) \\ \mathbf{false} & \text{if } \phi = true \\ \mathbf{true} & \text{if } \phi = false \\ \mathcal{T}(\phi_1) & \text{if } \phi = \neg\phi_1 \\ \mathcal{T}(\neg\phi_1) \vee \mathcal{T}(\neg\phi_2) & \text{if } \phi = \phi_1 \wedge \phi_2 \\ \mathcal{T}(\neg\phi_1) \wedge \mathcal{T}(\neg\phi_2) & \text{if } \phi = \phi_1 \vee \phi_2 \\ \exists x. \mathcal{T}(\neg\phi) & \text{if } \phi = \forall x. \phi \\ \forall x. \mathcal{T}(\neg\phi) & \text{if } \phi = \exists x. \phi \end{cases} \\
\mathcal{T}(\phi_1 \wedge \phi_2) &= \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2) \\
\mathcal{T}(\phi_1 \vee \phi_2) &= \mathcal{T}(\phi_1) \vee \mathcal{T}(\phi_2) \\
\mathcal{T}(\forall x. \phi) &= \forall x. \mathcal{T}(\phi) \\
\mathcal{T}(\exists x. \phi) &= \exists x. \mathcal{T}(\phi)
\end{aligned}$$

Figure 11: $\mathcal{T}()$ of Predicates and logical operators

- Applying \mathcal{T} to $\phi \vee \neg\phi$ will give us a new expression that can be evaluated by a theorem prover to *true* if the original expression (here ϕ was well-formed and to *false* if it was not. \mathcal{T} applies the checks from section 2.2 (except for consistency), where appropriate.

First we will return to the example

```
requires true || SquareRoot(-1) > 6
```

We saw earlier, that applying the strict checks from section 2.2 to this example would mark this as not well-formed, even though it is. Applying the $\mathcal{T}()$ operator to the example gives us the following expression (which is used as the proof obligation for the well-formedness of the precondition):

$$\mathcal{T}[(true \vee \#SquareRoot(...)) \vee \neg(true \vee \#SquareRoot(...))]$$

When applying the rule for $\mathcal{T}(\phi_1 \vee \phi_2)$ and for $\mathcal{T}(\neg(\phi_1 \vee \phi_2))$ we get

$$\mathcal{T}[true \vee \#SquareRoot(...)] \vee \mathcal{T}[false \wedge \neg\#SquareRoot(...)]$$

Again, we apply the rules for $\mathcal{T}(\phi_1 \vee \phi_2)$ and for $\mathcal{T}(\phi_1 \wedge \phi_2)$ and get

$$\mathcal{T}(true) \vee \mathcal{T}(\#SquareRoot(...)) \vee (\mathcal{T}(false) \wedge \mathcal{T}(\neg\#SquareRoot(...)))$$

Since $\mathcal{T}(true)$ is *true*, the whole expression evaluates to *true*, no matter what \mathcal{T} of the method call is.

After this example, we will finally show, how the well-formedness of individual nodes of the call graph is proved using the \mathcal{T} operator:

For invariants When applying \mathcal{T} to invariant I , to check its well-formedness, the resulting proof obligation looks like this:

$$MS \Downarrow \Rightarrow (I \Downarrow \Rightarrow \mathcal{T}(I \vee \neg I))$$

If this can be proven by the theorem prover, I is added to $I \Downarrow$

For methods Like in section 2.2, the pre- and postcondition of the method specification are checked individually, because the proof obligations are not

the same for both parts.

Well-formedness of Precondition P :

$$MS \downarrow \Rightarrow (I \downarrow \Rightarrow \mathcal{T}(P \vee \neg P))$$

Well-formedness of Postcondition Q :

$$MS \downarrow \Rightarrow (I \downarrow \wedge P \Rightarrow \mathcal{T}(Q \vee \neg Q))$$

Unlike in section 2.2, the whole postcondition is checked at once, and is not separated into Q'_i and Q''_i , because we are not proving the well-formedness of individual method calls anymore, but the well-formedness of whole expressions.

Since the consistency check is not included in $\mathcal{T}()$, it has to be added separately. The proof obligations for consistency are the same as in section 2.2.

Note, that $\mathcal{D}(\phi)$ could be used anywhere, where $\mathcal{T}(\phi \vee \neg\phi)$ is used, since they are both realizations of the Δ operator. In the implementation part of this thesis, they were both implemented to be able to see the difference between the proof obligations generated by both of them.

2.3.5 \mathcal{D} vs. \mathcal{T} : A comparison

One might wonder, why we bothered to find a new realization of the *Delta* operator, since there already are several of them. We will show the proof obligations of \mathcal{T} and \mathcal{D} for the same specification, for a little comparison. While the output of \mathcal{D} grows exponentially with the size of the expression it is applied to, the size of the output of \mathcal{T} is linear. Figures 13 and 12 show the different sizes of the outputs both operators generate for the precondition P of *exchangeSizes* in figure 7. Note, that \mathcal{D} is applied to P directly, while \mathcal{T} is applied to $P \vee \neg P$

2.4 Relative Totalness

Additionally to the well-formedness checks explained earlier, one might want to prove, that a pure method is total, relative to its precondition. This means, that, whenever the precondition holds, one of the Q'_i s holds. When a method is used in a specification, the well-formedness proofs ensure, that its precondition holds. Relative totalness ensures, that the axiom generated for the pure method can always be applied, when the method is used in a well-formed specification. The proof obligation for relative totalness is

$$MS \downarrow \Rightarrow (I \downarrow \Rightarrow \neg P \vee Q'_1 \vee \dots \vee Q'_n)$$

$$\begin{aligned}
& circle.getSize() > this.minSize \wedge circle \neq null \wedge this \neq null \wedge \\
& circle.getSize() < this.maxSize \wedge circle \neq null \wedge this \neq null \wedge \\
& this.getSize() > circle.minSize \wedge this \neq null \wedge circle \neq null \wedge \\
& this.getSize() < circle.maxSize \wedge this \neq null \wedge circle \neq null \wedge \\
& \vee \\
& circle.getSize() \leq this.minSize \wedge circle \neq null \wedge this \neq null \wedge \\
& circle.getSize() \geq this.maxSize \wedge circle \neq null \wedge this \neq null \wedge \\
& this.getSize() \leq circle.minSize \wedge this \neq null \wedge circle \neq null \wedge \\
& this.getSize() \geq circle.maxSize \wedge this \neq null \wedge circle \neq null \wedge
\end{aligned}$$

Figure 12: Output of $\mathcal{T}()$ operator

Note, that this prove is not part of the well-formedness proof, because generating axioms for underspecified methods will not lead to unsoundness. Warnings for method specifications that are not total with respect to their preconditions might help the programmer to find errors in his/her specification that can lead to non-provable specifications because of missing axioms.

2.5 Generating axioms from pure methods for source code verification

With both approaches for proving well-formedness described in the two sections 2.2 and 2.3, the verification system ends up with a number of methods, some of which are known to be pure and well-formed. For these pure and well-formed methods, it has axioms of the form $I \downarrow \wedge P \Rightarrow Q'$ in its background predicate. These axioms can then be used for the normal round of proving the correctness of the program.

3 Spec# and Boogie

This section will explain the Spec# and Boogie architecture and their interaction to give an understanding of the system, the well-foundedness checks of this thesis, were implemented in. If not stated otherwise, the things explained in this section relate to the original Spec# verification system, before the proof obligations for the well-foundedness checks were added. An

```

(circle ≠ null ∧ this ≠ null ∧
((circle ≠ null ∧ this ≠ null ∧
((this ≠ null ∧ circle ≠ null) ∨
(this ≠ null ∧ circle ≠ null ∧ this.getSize() ≤ circle.minSize) ∨
(this ≠ null ∧ circle ≠ null ∧ this.getSize() ≥ circle.maxSize)))
∨
circle ≠ null ∧ this ≠ null ∧ circle.getSize() ≥ this.maxSize
∨
(((this ≠ null ∧ circle ≠ null) ∨
(this ≠ null ∧ circle ≠ null ∧ this.getSize() ≤ circle.minSize) ∨
(this ≠ null ∧ circle ≠ null ∧ this.getSize() ≥ circle.maxSize)) ∧
(this.getSize() ≤ circle.minSize ∨ this.getSize() ≥ circle.maxSize))))
∨
(circle ≠ null ∧ this ≠ null ∧ circle.getSize() ≤ this.minSize)
∨
(((circle ≠ null ∧ this ≠ null ∧
((this ≠ null ∧ circle ≠ null) ∨
(this ≠ null ∧ circle ≠ null ∧ this.getSize() ≤ circle.minSize) ∨
(this ≠ null ∧ circle ≠ null ∧ this.getSize() ≥ circle.maxSize)))
∨
circle ≠ null ∧ this ≠ null ∧ circle.getSize() ≥ this.maxSize
∨
(((this ≠ null ∧ circle ≠ null) ∨
(this ≠ null ∧ circle ≠ null ∧ this.getSize() ≤ circle.minSize) ∨
(this ≠ null ∧ circle ≠ null ∧ this.getSize() ≥ circle.maxSize)) ∧
(this.getSize() ≤ circle.minSize ∨ this.getSize() ≥ circle.maxSize))) ∧
(circle.size ≥ this.maxSize ∨ this.getSize() ≤ this.minSize ∨
this.getSize() ≥ circle.maxSize))

```

Figure 13: Output of $\mathcal{D}()$ operator

overview of the system can be seen in figure 14.

The three logical parts *compiler*, *byte code translation* and *Verification* (Boogie) are separated into two implementation units: The Spec# compiler and Boogie, with the byte code translation being part of Boogie.

3.1 Spec#

The program, that should be verified is passed to the Spec# compiler as Spec# code. Besides generating CIL code(.NET byte code)out of the implementations in the Spec# code, it also performs some additional type checks, e.g. checks for possible null dereferences or type checks in specification expressions. Since information like method contracts and invariants, which are essential to Spec# are not part of the CIL, this additional information is stored in attributes of the methods and classes they specify. The CIL code generated by the compiler is stored in an executable (.exe file) or in a library (.dll file).

For detailed information on Spec# see [5] and [14].

3.2 Boogie

When Boogie processes an executable or a library generated by the Spec# compiler, it first parsed the CIL in that file back into an abstract syntax tree. It then goes through the whole tree and generates additional nodes from the attributes that contain a serialized form of the contracts in the original Spec# code. From this extended AST, Boogie then builds a second tree representing the BoogiePL equivalent to the original program.

BoogiePL is an intermediate language introduced for Boogie. It is a procedural language that can be used to represent programs written in imperative languages. Since many of the high-level constructs of an object oriented language like classes or the heap don't exist in BoogiePL, using this abstraction step simplifies the further proceedings considerably. Additionally, using this intermediate language, makes it possible to use Boogie for languages besides Spec#, by providing a translator from that language to BoogiePL. From this simpler syntax tree, Boogie then generates all the proof obligations it needs for proving the programs correctness. These proof obligations are then sent to an external theorem prover, which evaluates these to true or false. In our case this theorem prover is Simplify (for more information on Simplify, see [10]), but Boogie also supports other theorem provers like Zap [2].

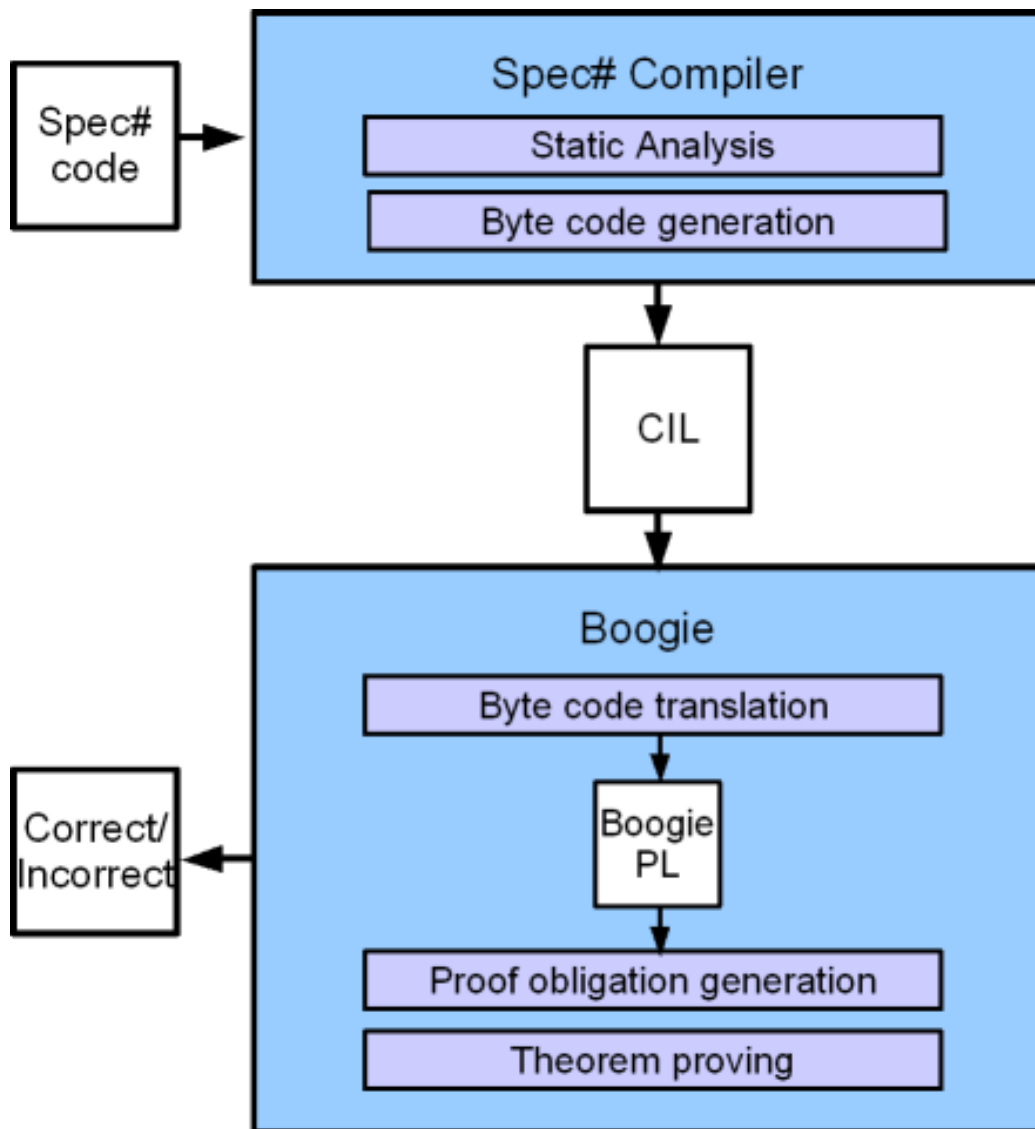


Figure 14: The original Boogie Pipeline

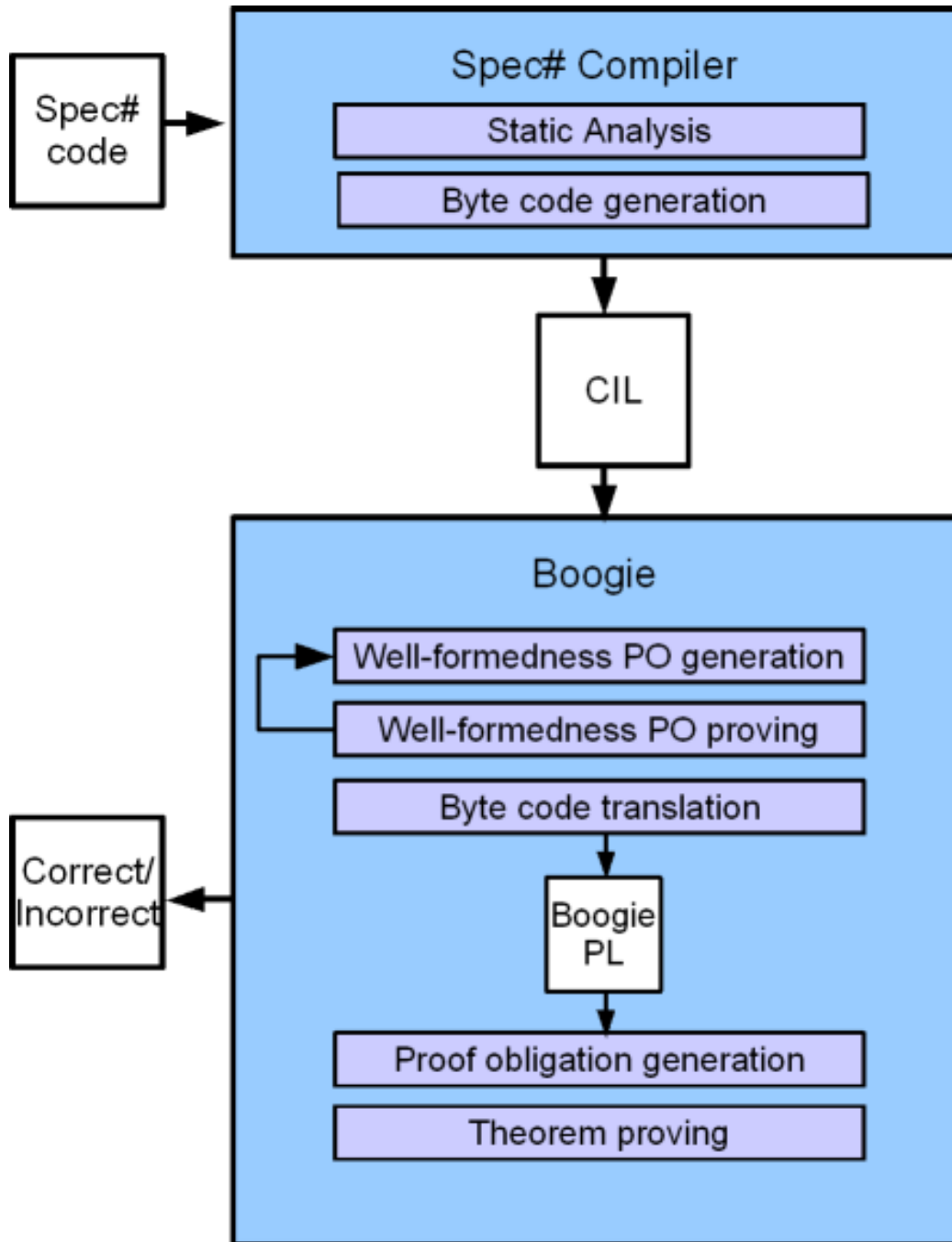


Figure 15: The Boogie Pipeline after adding well-formedness proofs

3.2.1 Well-formedness proofs

The newly added checks for well-formedness of specifications are added to Boogie. Before the byte code translation and the regular proof round, the well-formedness checks are performed on the AST that Boogie parsed out of the executable or the library. The only change to the regular prove round is, that $I \downarrow$ is now added to the axioms for method specifications. Warnings are emitted for invariants and methods whose specifications are found to be not well-formed. Pure methods, that are indeed well-formed, are at this point marked with a special attribute, so that, when translating the AST to a BoogiePL tree, axioms can be generated out of them and added to the background predicate. Figure 14 and 15 show, the difference in the Boogie pipeline before and after the well-formedness proofs were added.

4 Implementation

This section describes in detail the implementation of the concepts explained in section 2 . Almost all code is in new files, very little existing code was changed. The changes that did take place (in the `Spec#` compiler, and in the `Main` method, the byte code translation and the `CommandLineOptions` class of Boogie) is surrounded by `//GVR . . . //end GVR` and can easily be found using the search function.

4.1 Extending the Boogie Pipeline

Boogie takes an .exe or .dll file, parses it into a CIL AST and then verifies the program using that AST. The well-formedness checks are performed before the regular program verification, but they work on the same AST. Working on the same AST is necessary, because the part, that checks the well-formedness, needs to pass the information, what methods are to be axiomatized for the regular prove round, to the part of Boogie that axiomatizes them. This is done by adding a special attribute called *hasWitness* to each method, that should be axiomatized. The attribute is called *hasWitness*, because, when not using the well-formedness checks, it is generated by the `Spec#` compiler for methods, that have a witness, i.e. that have feasible postconditions.

Working on the same AST makes it necessary though, that the well-formedness checks don't change the AST, other than adding said attribute.

4.2 Introduction

The implementation of the well-formedness checks is separated into three different classes.

- CallGraph
- WellFormednessCheckerT
- WellFormednessCheckerD

The most important one is *CallGraph*. A call graph is instantiated in the *Main* class of Boogie, after the AST (which is called *module* in Boogie) is generated out of the file containing the program. The call graph builds itself up, checks itself for circles and then iterates through its nodes and performs well-formedness checks on each of them.

There are three alternatives of how well-formedness can be checked:

- Only checking well-formedness of all method calls.
- Using the \mathcal{T} operator.
- Using the \mathcal{D} operator.

The method *Check* is used for the first variant, *CheckTorD* is used for the other two. Which method and whether \mathcal{T} or \mathcal{D} is used by *CheckTorD* depends on the command line arguments (see section 4.12).

If the \mathcal{T} or \mathcal{D} operator is used to do the well-formedness checks, the call graph uses *WellFormednessCheckerT* or *WellFormednessCheckerD*. They both implement the same interface and implement the \mathcal{T} and \mathcal{D} operator respectively. Note, that *Check* or *CheckTorD* is called on a whole call graph, and not separately on each node.

Additionally to the well-formedness, *Check* and *CheckTorD* also check the consistency and relative totalness for every method in the call graph. When a method is found to be well-formed, consistent and total relative to its precondition, an axiom generated from its specification is added to the background predicate of the verifier. If an invariant is found to be well-formed, it is added to `currentIdown`, which is located in a helper class so it can later be accessed when generating axioms for the regular proof round.

The last part of the implementation is a little change to the byte code translation - which translates the AST to a BoogiePL tree - to correctly add axioms for well-formed methods to the BoogiePL tree.

4.3 Some preliminaries

This section contains the explanation of some important things that are used in several places of the implementation.

4.3.1 Helper Files

The call graph and the well-definedness checker sometimes needs to access parts of the Boogie implementation, that is in a different package and that is marked *protected* and can therefore not be accessed by the call graph or the well-definedness checker. To minimize the change to existing source files, two helper classes were created. One of them, called *SimplifyProverPublic.ssc* is in the *VCGeneration* package and implements a public wrapper for the *SimplifyProver* class. This class is used for everything that is related to the communication with Simplify.

The other helper class is called *CallGraphHelper.ssc* and resides in the *ByteCodeTranslation* package. It is used to have access to methods that need to translate between the CIL AST and BoogiePL ASTs. It is also used to store the $I \downarrow$ s for each method, since they are used in the byte code translation of the regular prove round, and byte code translation does not have access to the package the call graph is in. *CallGraphHelper* also contains two methods that are used by both the call graph and the well-definedness checker.

4.3.2 The theorem prover

Boogie uses an external theorem prover called Simplify to verify the proof obligations it generates. Once simplify is started, it can be sent theorems and it sends back whether this theorem is provable or not. Additionally, new axioms to be added to the background predicate can be sent to Simplify. The well-formedness check starts its own session with Simplify and ends it, before the regular proof round starts. This way, the regular proof round cannot use, or be bothered by, any axioms added during well-formedness checks and the implementation of the regular prove round did not have to be changed more than necessary.

The details of the communication with Simplify are handled by the *VC-Generation* package of Boogie. The well-formedness check only uses the *Check()* and *Feed()* methods to check a theorem or feed an axiom to the prover. To get an expression from a CIL AST (which is of type *System.Compiler.Expression*) to a format that is understood by Simplify, it has to first be translated to BoogiePL, which is done using the *CILTranslator*. BoogiePL expressions, called *Bpl.Expr*, have a method called *VCView()*, that translates them to a

VCEpr. This *VCEpr* is then translated to a string using the *VCEpr* *ExpressionGenerator*. This string can finally be sent to Simplify. A typical example of how an expression is transformed and sent to simplify looks like this.

```
Cci.Expression P = CallGraphHelper.ANDPreconditions(...);
//translate to BoogiePL
Bpl.Expr bplExpr = CallGraphHelper.GetBplExpr(translator, proverExpression);
//use VCView to translate to VCEpr
VCEpr vcExpr = bplExpr.VCView(gen);
//use gen.toSimplify to generate simplify string
string proverFormula = gen.ToSimplifyString(vcExpr);
//send to simplify
outcome = (TheoremProver.Outcome)prover.Check(proverFormula);
```

4.3.3 Result replacement

The postcondition of a method may contain the literal `result`, which represents the return value of the method. For this thesis, postconditions of pure methods must even contain `result`. When translating such a postcondition to BoogiePL, `result` needs to be replaced by the function symbol for the method. The function symbol starts with a `#` and continues with the name of the method and its parameters. The object, that translates CIL to Boogie, needs to know what function symbol *result* should be replaced with. Before translating a postcondition to BoogiePL (e.g. to send it to Simplify) This function symbol must be set. *CallGraphHelper* has a method called *SetResultReplacement* that computes the appropriate function symbol and sends it to the translator.

4.3.4 The StandardVisitor

The Spec# compiler has a class called *StandardVisitor*. In this implementation, there are several places where a class that inherits from *StandardVisitor* is used. *StandardVisitor* is a visitor, to traverse a CIL AST. The implementation of *StandardVisitor* does not do anything except for traversing the AST, but there is a *Visit* method for all kinds of AST nodes (e.g. methods, contracts, method calls, etc.), and a mechanism to automatically find the correct *Visit* method for a node, depending on its type. This means, if the generic *Visit()* method is called for a node of the AST, *StandardVisitor* automatically finds the correct method for the node. E.g. if the node is of type *method*, *VisitMethod()* is called. Each of these visit methods return a AST node. They also call *Visit* on all children of the node they are visiting and replace those children with the result of those calls. For *StandardVisitor*,

the node is returned unchanged, but one can inherit from *StandardVisitor* and override the desired methods, to change the AST, by changing the node or returning a totally different node. The overriding method can also perform a certain action for nodes of certain types. If e.g. one wants to count all methods in a program, one could create a class that inherits from *StandardVisitor* and only override *VisitMethod*. The overriding *VisitMethod* can then increase some variable every time it is called. The implementation of *StandardVisitor* makes sure, that the AST is traversed and all nodes of type Method are reached. If the children of a node should be visited, the overriding method must either call its base method or call *Visit* on all its children. It is usually easier to call the base method, because that way one does not need to know all children of the node.

4.4 The call graph

The class *CallGraph* contains a Dictionary, that contains all *CallGraphNode*s that are part of the call graph. This dictionary maps the *CallGraphNode*s to the IDs of the methods or invariants the nodes represent, so that one can easily find the node of the call graph that represents a given method or invariant.

Each *CallGraphNode* has a reference into the AST to either the method or invariant it represents. To make things easier to understand, we will from now on say call graph node *A* calls call graph node *B*, when in reality, the specification of the node represented by *A* calls the method represented by *B*. Call graph nodes also have an ID, which is the same as the ID of the method or invariant they represent.

Each *CallGraphNode* also has two Dictionaries containing the call graph nodes they call. These Dictionaries map, the call graph nodes to their IDs. The two separated Directories are used, to be able to distinguish between call graph nodes that are called in Q_i'' of a method, and other call graph nodes. This distinction is needed when checking for circles, because direct circles are only allowed in calls that occur in Q_i'' . These two Dictionaries can be seen as the edges of the call graph.

Additionally, there are three lists in every call graph node, that contain pointers to the nodes of the AST that represent the method calls whose targets are stored in the two Directories. These lists are needed, because, when proving the well-formedness of method calls, the calls themselves and not only their targets are needed. Again, these method calls are separated into three different directories, because they need to be treated differently, when proving their well-formedness. (E.g. For calls in the postcondition, the precondition can be assumed.

4.4.1 Building the graph

The call graph uses two visitors to build itself up. Both of them inherit from the class *StandardVisitor*. The first Visitor generates a *CallGraphNode* for each method and invariant of the program and adds them to the call graphs list of nodes. The second Visitor is more complex. It traverses the AST again, and for each method call in a specification, it finds the two nodes in the call graph, that correspond to the caller and the callee, and enters the callee and the method call into the Dictionaries of the caller. The main work is done in the method *VisitMethodCall*. The other methods of the visitor only set some flags, so that *VisitMethodCall* knows, whether the method call it is currently visiting is in a specification, what method or invariant that specification belongs to, and what part of a method specification it is in.

4.4.2 Checking for circles

To check for circles in the call graph, a method visits all nodes in the call graph recursively. From each node n it goes to all nodes that are in the Dictionary of nodes n calls. Every time it visits a node, it marks it as visited. If the method visits a method, that is already marked visited, it knows that it found a circle. It then checks, whether this is a direct circle (the node it just came from is the same as the one it is visiting) and whether the call is in a part of the specification, where direct circles are allowed (i.e. the Q'_i of a method specification). If an illegal direct circle is detected, the method is marked as not being pure, to keep later stages from axiomatizing it. In theory, when an indirect circle is found, all methods that are part of that circle should not be axiomatized. Unfortunately, it's not possible to find all methods in an indirect circle, so the *CheckCircle* method returns false, if one is found. This results in Boogie quitting, since there is no way to correctly continue the proofing process. One might want to change this to just the well-formedness checking quitting, so that Boogie can try to proof the correctness without the axioms of well-formed pure methods anyway.

4.4.3 Assigning order and picking the next node to check

The next step is to pick nodes in an order, so that every node is checked after all nodes it depends on (i.e. that it calls). To do this, all nodes are assigned a number called the order. Every node gets an order, that is higher, than the highest number of all methods it calls, i.e. a number that is higher than all its childrens order. Nodes, that don't call any other nodes, get the order 1. In addition, all nodes that only call themselves, also get the order 1, since they don't depend on any other nodes. To pick the next node to check, the

method *GetNodeWithLowestOrder()* goes through the list of all nodes and picks either the first node that has the same order as the last node, that was picked, or the first node with the next higher order, if no node has the same order as the last one picked. To make sure that no node is picked twice, *GetNodeWithLowestOrder()* uses the *visited* field of *CallGraphNode* to mark the nodes that were already picked. This means, *clearVisited()* has to be run before a round of picking nodes is started.

4.5 Checking well-formedness of method calls

The method *Check* of *CallGraph* is used, when only the well-formedness of method calls should be checked, as described in section 2.2. This means, that well-definedness and well-foundedness of recursion for each method call in a specification and consistency of each method is proved. Note, that, only actual method calls, and no field accesses or mathematical operators like division, are checked. The following sections will describe the different steps of the method *Check*, in the order they appear in the source code. Except for the implicit non-null invariants, all steps are done individually for each node of the call graph. *Check* uses the *GetNodeWithLowestOrder()* method mentioned in the last section to pick the next node to check.

4.5.1 Implicit non-null invariants

The first thing *Check()* does, is to add the implicit non-null invariants to $I \downarrow$. Implicit non-null invariants are invariants stating, that a certain field of a class is never null, because said field is marked to be of non-null type, using the exclamation mark [5]. Since these invariants are always of the form *this.f ≠ null* and thus don't contain any method calls, they do not need to be checked for well-formedness.

4.5.2 Checking precondition preservation

For each method call in one of the three Dictionaries of *CallGraphNode*, the preservation of the precondition of the callee is checked. Depending on where in the specification the call occurs, which can be seen from what Dictionary the call is stored in, The precondition and/or the C'_i 's of the caller can be used to proof this. The most interesting part of the *CheckPreconditionPreservation()* is its use of the *ParameterReplacementVisitor* to get P' , which is the precondition of the callee, with the formal parameters replaced by the actual ones. The *ParameterReplacementVisitor* inherits from the *StandardVisitor*. It takes the preconditions of a node (a list of requires clauses), a Dictionary

```

[Pure] int DivideBy(int a, int b)
requires b != 0;
{
    ...
}

void foo(int c, int d, int e)
requires DivideBy(c,d + e) > 6;
{
    ...
}

```

Figure 16:

that maps the IDs of the parameters of a method to the order of the parameters, and a list containing the expressions of the actual parameters. It then iterates over the nodes in the preconditions, and every time it hits a parameter, it looks up its position in the Dictionary and replaces the parameter node with the corresponding actual parameter. Since this would change the AST, a copy of the preconditions is used instead of the original AST.

When proving precondition preservation in the example in figure 16, one wants to prove, that in the call to *DivideBy* in the precondition of *foo*, the precondition of *DivideBy* is preserved. The *ParameterReplacementVisitor* takes the list of preconditions of *DivideBy*, a Dictionary that maps the ID of parameter *a* to 1 (since it is the first parameter) and the ID of *b* to 2, and a list containing the expressions *c* and *d + e*. It then goes through the precondition of *DivideBy*, finds the node representing *b*, and replaces it by the node for *d + e*, since it found *b* to be the second parameter and *d + e* is the second expression in the list of actual parameters.

4.5.3 Consistency

The next step is to prove the consistency of the node. This is explained in 4.7 because it is also used by *CheckTorD*.

4.5.4 Checking well-foundedness of recursions

To check the well-foundedness of recursive calls in the specification of a node, one has to first find these recursive calls. To do this, for all calls in the nodes Dictionary of calls in the Q_i'' of the method the node represents, the ID of the node is compared with the ID of the callee. If they are identical,

the call is recursive. Only calls in Q_i'' have to be checked, because it is the only place where recursive calls are allowed. Recursive calls in other parts of specifications would have been found when building the call graph. Invariants cannot contain recursive calls, because one cannot call an invariant at all. When a call is found to be recursive, the two proof obligations for a decreasing measure and recursion termination are generated and sent to Simplify individually. The two most interesting things of this part of the implementation is parsing the measure expression and constructing E' .

Parsing the measure Expression The measure expression is stored as a string in an attribute of the method it belongs to. The parser of the Spec# compiler is used to parse this string into an AST. Unfortunately this does not work too well. An AST is generated, but the parser lacks the context of what it is parsing. Thus all identifiers are turned into identifier nodes, even if they are parameters, since the parser does not know what parameters the current method has, or even that it is parsing something belonging to a method. The *AddParametersToParseTreeVisitor* - another class that inherits from the *StandardVisitor* - is used to at least replace identifiers that have the same name as parameters of the method with the nodes corresponding to these parameters. Therefore, parameters can be used in the measure expression. This does not work with method calls or field accesses though. Parsing this string seems to be a rather complex task.

Constructing E' E' is the measure expression of the method, but any formal parameters that are used in it, are replaced by the actual parameters that are used in the recursive call. This works similar to replacing the formal parameters in the precondition by the actual ones to prove precondition preservation. Hence the same visitor is used. In figure 4.5.4, there are two recursive calls, $\text{Fib}(p-1)$ and $\text{Fib}(p-2)$. The measure expression E is simply p . For the first call, E' is $p-1$, for the second one $p-2$.

4.5.5 Extending $MS \downarrow$ and $I \downarrow$

Like consistency, the next step, extending $MS \downarrow$ and $I \downarrow$, is done by *CheckTorD* as well and is therefore described in its own section. See section 4.9

4.6 Checks using the \mathcal{T} or \mathcal{D} operator

Section 4.5 described how well-formedness is proved using the approach to prove well-formedness of all method calls in the specifications. This section

```

class Fibonacci
{
    //computes the p-th Fibonacci number
    [Pure] [Measure("p")] int Fib(int p)
        requires p >= 0;
        ensures p == 0 ==> result == 0;
        ensures p == 1 ==> result == 1;
        ensures p > 1 ==> result == Fib(p-1) + Fib (p-2)
    {
        ...
    }
}

```

will describe the implementation of the well-formedness checks when the command line parameters indicate to use \mathcal{T} or \mathcal{D} .

4.6.1 The WellFormednessChecker

The core of this part of the implementation are the classes *WellFormednessCheckerT* and *WellFormednessCheckerD*. They both implement the very simple interface *WellFormednessChecker*.

```

public interface WellFormednessChecker
{
    Cci.Expression Check(Cci.Expression expr);
}

```

This means, they take an expression and return a different expression. As the names suggest, *WellFormednessCheckerT* implements the \mathcal{T} operator and *WellFormednessCheckerD* implements the \mathcal{D} operator. *CallGraph* has a member of type *WellFormednessChecker*, and, depending on the command line options, this member gets instantiated with either an object of type *WellFormednessCheckerT* or *WellFormednessCheckerD*.

Both classes inherit from the *StandardVisitor* class. Both classes follow figures 10 and 6 showing the definitions of \mathcal{T} and \mathcal{D} respectively. It is important to see though, that the *Check* method of *WellFormednessCheckerT* applies its visitor to $\phi \vee \neg\phi$, while *WellFormednessCheckerD* applies it to just ϕ . This means, that, for both classes, calling their *Check* method, means applying the Δ operator, even though the visitor of *WellFormednessCheckerT* does not implement Δ directly. While the implementation of \mathcal{T} and \mathcal{D} was straight forward for most parts, there were some difficulties in it, due to the

differences in the structure used for the syntax of the definitions of \mathcal{T} and \mathcal{D} and the structure of the AST the visitors have to work on.

In the definitions of \mathcal{T} and \mathcal{D} , the rules for logical operators (e.g. \wedge), predicates (e.g. $<$) and calls to built in operators like the addition are separate. In a CIL AST however, they are all *BinaryExpressions* of different types. The *VisitBinaryExpression* method of the two visitors must therefore handle all of them. This is done using a switch table to distinguish between the different types of binary expressions. The logical operators are all handled separately. The other binary expressions are separated into those that return a boolean (i.e. Predicates) and others. The same is done for unary expressions, where the \neg operator is separated from the others.

Another interesting point is, what is handled in the definitions of \mathcal{T} and \mathcal{D} as $df(e_1, \dots, e_n)$: The preconditions of a function call. In the theory function calls can be actual method calls or applications of built in mathematical functions like addition and division, or even field accesses. In the implementation, these things are handled separately. Field accesses have the precondition, that the receiver must not be null and are handled in *VisitMemberBinding*. Mathematical functions are handled in *VisitBinaryExpression*. Right now, the only handled precondition for these is, that one cannot divide by zero. The biggest work for df is done for method calls though. df of method calls includes the actual precondition added to the method by the programmer, the non-nullness of the target object and, if applicable, the well-foundedness proofs. The actual preconditions and the non-nullness are handled in the visitors directly. For the well-foundedness proofs, the structure of the call graph is needed though. Therefore, they are generated by the call graph and stored in the *CallGraphHelper*, before using the *WellFormednessCheckers*. The proof obligations are stored in a Dictionary that maps them to the IDs of the corresponding methods.

The definition of \mathcal{T} and \mathcal{D} use the operator Δ_e to handle variables, method calls and arguments of method calls. This separated operator makes it easy to handle all formulas differently, when they appear in arguments. *WellFormednessCheckerT* and *WellFormednessCheckerD* on the other hand, are implemented as a single visitor.

There are two additional special things, that only apply to *WellFormednessCheckerT*. The first thing is the field *topLevelPredicate*. In the definition of the \mathcal{T} operator, the values of predicates are used. For expressions like boolean method calls or boolean variables, that are, in the definition, treated as terms, there exists a special predicate, that returns the value of the expression. The values of predicates are only needed for the outermost, or, for an expression tree, topmost, predicate. For the implementation this means, that, for every expression, one needs to know, whether it is in a position,

```

[Pure] int Double(int c)
{
    ....
}
int f(int d, int e)
    requires Double(d) > 10;
{
    ...
}

```

Figure 17:

where its value is needed.

In the precondition of method *f* in figure 17, the top level predicate is the $>$ sign. When its node is processed by the visitor, the whole expression is added to the expression to be returned and the field *topLevelPredicate* is set to false before the children of the node are processed. Hence, when the visitor arrives at the method call, it does not add the method call anymore, since it's already part of the expression added for the $>$ node. After all children of the top level predicate are processed, *topLevelPredicate* is set back to true.

The other interesting thing about *WellFormednessCheckerT* is, how parameters of method calls are handled. The result of any expression, that is the parameter of a method call, is not interesting to \mathcal{T} , since a method can very well return *true*, even if one of its parameter is *false*. For most nodes in the AST, this could be handled, by just setting *topLevelPredicate* to *false* inside method calls, but, since logical operators don't react to *topLevelPredicate*, the additional field *insideMethodCall* was added to *WellFormednessCheckerT*. Figure 18 shows an example, where a logical operator, here the $\&\&$, is used as a parameter.

4.6.2 The *CheckTorD* method

This section will describe what *CheckTorD* does step by step. Remember, the point of this method is, to check the well-formedness of all nodes in a call graph.

The first thing, *CheckTorD* does, is to add the implicit non-null invariants to *currentIdown*. This is done the same way as described for the *Check* in section 4.5.2. After that, the well-foundedness proofs of all recursive methods in the call graph are generated and stored in the *CallGraphHelper* so that *WellFormednessCheckerT* or *WellFormednessCheckerD* can use them, when

```

[Pure] bool XOR(bool a, bool b)
{
    ....
}
int f(bool c, bool d, bool e)
    requires XOR(c && d , e);
{
    ...
}

```

Figure 18:

necessary.

The things below are done to each node of the tree. Like *Check*, *Check-TorD* uses *GetNodeWithLowestOrder* to get the next node to check. To make things more readable, Δ is used in places, where the *Check* method of either the *WellFormednessCheckerT* or *WellFormendnessCheckterD* is used in the code.

- The proof obligations for the well-formedness of invariant I looks like this:

$$I \downarrow \Rightarrow \Delta(I)$$

If I is found to be well-formed, it is added to $I \downarrow$ as described in section 4.9.

- The proof obligation for the precondition P of a method, looks as follows:

$$I \downarrow \Rightarrow \Delta(P)$$

- The proof obligation for the postcondition Q of a method looks as follows:

$$I \downarrow \wedge P \Rightarrow \Delta(Q)$$

where P is the precondition of said method.

- The proof obligations for the relative totalness of a method with precondition P and n Postconditions of the form $Q'_i \Rightarrow Q''_i$ is

$$\neg P \vee Q'_1 \vee \dots \vee Q'_n$$

- The consistency of the method is checked. See section 4.7 for more information.

- If the proof obligations for the method are proved by Simplify, an axiom is generated from the specification of the method as described in section 4.9. Note, that this excludes the proofs for relativ totalness, since this property is not needed for well-formedness.

$MS \downarrow$ is missing from the proof obligations, because it is in the background predicate of Simplify and does therefore not appear in the proof obligations explicitly.

4.7 Checking consistency

Proving the consistency of a method means to prove that all Q'_i s are mutually exclusive. In general, the proof obligation described in section 2.2 have to be generated and sent to Boogie. There are however some cases, where this is not necessary. A method is consistent, if it has either no ensures clause, or only one of them. On the other hand, a method is not consistent, if it has more than one ensures clause and at least one of them has no Q'_i , because the default value for Q'_i is *true*, and nothing is mutually exclusive from *true*. The consistency of invariants is not proven, since it cannot be proved by an automated theorem prover and inconsistent invariants do not introduce unsoundness.

4.8 Recursion on rep fields

A special case of recursion on references is recursion on references that are rep fields of the caller. Accepting recursion on rep fields was so far implemented in the Spec# compiler. (See [7]). The feature, that contains it, has to be turned off to allow recursively specified methods to pass the compiler though. Therefore, the *Check* and *CheckTorD* methods of the CallGraph include a check, that marks methods with recursion on references to be well-formed, if the recursive target parameter is owned by the calling object. Hence, recursion on rep fields still works as it did before.

4.9 extending $MS \downarrow$ and $I \downarrow$

When the specification of a node is proved to be well-formed, its specification is axiomatized and added to the background predicate, if the node represents a pure method, or it is added to $I \downarrow$ if the node represents an invariant.

Well-formed methods For each ensures clause of the method, an axiom is generated from its specification and added to the background predicate of

Simplify. This is done by generating the expression

$$P \wedge \text{currentIdown} \Rightarrow Q$$

This expression then needs to be surrounded by a forall quantifier over all parameters of the method, since this is not done by Simplify automatically. The expression then looks like this:

$$\forall(p_1..p_n) \bullet P \wedge \text{currentIdown} \Rightarrow Q$$

where $p_1..p_n$ is the list of all parameters of the current method. This expression is then transformed to a string and sent to Simplify, as described in section 4.3.2.

Because a new session with Simplify is started for the regular prove round, these axioms have to be added to Simplifys background predicate again at that time. Since `currentIdown` will probably be different by then, it is stored in a Dictionary alongside the ID of the method.

Well-formed invariants `currentIdown` in *CallGraphHelper* represents the $I \downarrow$ used in the theory part. Whenever an invariant I is found to be well-formed, `currentIdown` becomes either I , if `currentIdown` is *null* or $I \wedge \text{currentIdown}$, if it is not.

4.10 Generating axioms for the regular proof round

As mentioned above, all axioms for pure and well-formed methods need to be generated and added to Simplifys background predicate again for the new session started with Simplify for the regular prove round that proofs the correctness of the program. This is done in method *AddAxiomForPureMethod* in class *MethodSignature*. Since the same axiom as before is generated, the $I \downarrow$ s, that were stored when generating the axiom for well-formedness proofs, are taken from *CallGraphHelper*. The axiom is not added to the background predicate directly, but turned into a BoogiePL axiom instead.

4.11 Changes to the Spec# compiler

Only very little is changed in the Spec# compiler. On one hand, support for the *Measure* and *MeasureRep* attributes were added and on the other hand, splitting up invariants was turned off.

Splitting up invariants means, that e.g. an invariant of the form $a \wedge b$ is split up into two invariants a and b . This was previously added to the Spec# compiler to be able to give more detailed information on where a

verification error occurred, but it cannot be used with the well-formedness proofs, because we want the programmer to be able to decide, what part of the class invariant needs to be proven as one unit. I.e. $true \vee x.f$ is well-formed, even if x is *null*. If the invariant gets split up, the second part cannot be proven to be well-formed anymore. Unfortunately, splitting the invariants cannot be turned off by command line arguments, so it was turned off in the code.

All changes in the Spec# compiler are surrounded by `//GVR ... //end GVR`, so they can easily be found using the search function.

4.12 Usage

; This section will explain how to use this implementation. The well-formedness check can only be used from the command line. It is not integrated into Visual Studio yet. There are two different executables, that need to be run to check a program.

The first one is called *ssc.exe*. This is the Spec# compiler. It takes the .ssc file that contains the code of the program to be verified. To allow recursive specifications, the command line argument `/cc-` is needed. To generate a .dll file instead of an .exe file, use `/target:library`. This is useful, when there is no main method. Additionally, the `/debug` parameter is needed. A typical command to start the compiler looks like this:

```
ssc.exe /debug /target:library /cc- examples.ssc
```

The second executable is the verification part called *Boogie.exe*. It takes a .dll or .exe file generated by the Spec# compiler. To use the \mathcal{T} operator, use the argument `/wfT`, for the \mathcal{D} operator use `/wfD` and for well-formedness of method calls use `wfS`. One can also pass a verbose level for the well-formedness checker by appending `:x` to one of those three arguments, where x is the verbose level. The default level is 3;

- Verbose level 1: No output.
- Verbose level 2: Errors are displayed.
- Verbose level 3: Warnings are displayed.
- Verbose level 4: The proof obligations for warning are displayed.
- Verbose level 5: All proof obligations are displayed.

A typical command to start Boogie looks like this:

```
Boogie.exe /wfT:4 examples.dll
```

4.13 Known issues

In theory, the measure expression for a method can be an arbitrary integer expression. In this implementation, only a restricted set of measure expression can be used. The measure expression can contain parameters of the method and build in operators like addition or subtraction, but no method calls or field accesses. This is due to difficulties with parsing the string in the measure attribute to an AST. In particular, no way to give the correct context to the parser was found and so the parser does not know how to interpret anything that is particular to the method or even program the attribute belongs to, except for parameters.

Another issue with this implementation is, that to compare methods when building the call graph, the IDs given to them by the compiler are used. Unfortunately, the compiler assigns different IDs to two methods, even when one of them is overriding the other. This means, that any circles in the call graph or recursive calls in the specification are not found, if the circle or recursion depends on the knowledge, that two methods are connected via inheritance.

Also, implicit non-null preconditions are not used by this implementation. Implicit non-null preconditions are parameters that are marked to be of non-null type using the exclamation mark.

5 Examples

5.1 Filtering non well-formed specifications

Figure 19 shows an example of a non well-formed method specification, that passes Boogie and is axiomatized, when not using the well-formedness checks, because only feasibility is checked. When the well-formedness checks are activated, the non well-formed specification is found and no axioms are generated.

Without the well-formedness checks, the possible precondition violation in the call to *EnoughChewingGumLeft* in the precondition of *ReadyToVent* is not detected and *ReadyToVent* gets axiomatized for the correctness proofs. The specification does not pass the well-formedness checks though.

5.2 The advantage of the \mathcal{T} operator

When taking another look at the specification in figure 19, the programmer might decide to change the specification of *ReadyToVent* to

```

class ChewingGumMachine
{
    float neededMoney;

    [Pure] bool EnoughChewingGumLeft(int neededChewingGums)
    requires neededChewingGums > 0;
    {
        ...
    }

    [Pure] bool ReadyToVent(int numberOfChewingGums, float money)
    requires EnoughChewingGumLeft(numberOfChewingGums);
    ensures result == money > neededMoney;
    {
        return money > neededMoney;
    }
}

```

Figure 19:

```

[Pure] bool ReadyToVent(int numberOfChewingGums, float money)
    requires numberOfChewingGums > 0 && EnoughChewingGumLeft(numberOfChewingGums);
    ensures result == money > neededMoney;

```

This way, the first part of the precondition ensures the well-definedness of the method call to *EnoughChewingGumLeft* in the second part of the precondition. When proving the well-formedness of all method calls, this precondition does still not pass though, because the well-formedness of the method call is proved all by itself, and it is not well-formed without *numberOfChewingGums* > 0.

When using either \mathcal{T} (or of course \mathcal{D}), the precondition does pass the well-formedness checks. Apply \mathcal{T} to the precondition gives us

$$\mathcal{T}(\text{numberOfChewingGums} > 0 \wedge \text{EnoughChewingGumLeft}(\text{numberOfChewingGums})) \vee$$

$$\mathcal{T}(\text{numberOfChewingGums} \leq 0 \vee \neg \text{EnoughChewingGumLeft}(\text{numberOfChewingGums}))$$

This is the same as

$$\mathcal{T}(\text{numberOfChewingGums} > 0) \wedge$$

$$\mathcal{T}(\text{EnoughChewingGumLeft}(\text{numberOfChewingGums})) \vee$$

$$\mathcal{T}(\text{numberOfChewingGums} \leq 0) \vee$$

$$\mathcal{T}(\neg \text{EnoughChewingGumLeft}(\text{numberOfChewingGums}))$$

This can again be transformed to

$$\begin{aligned} & \text{numberOfChewingGums} > 0 \wedge (\text{EnoughChewingGumLeft}(\text{numberOfChewingGums}) \wedge \\ & \text{numberOfChewingGums} > 0) \vee \\ & \text{numberOfChewingGums} \leq 0 \vee \\ & \neg(\text{EnoughChewingGumLeft}(\text{numberOfChewingGums}) \wedge \text{numberOfChewingGums} > 0) \end{aligned}$$

This is always *true*, because, if a is smaller than 0, the third row is *true* and, if a is greater or equal to 0, either the last row or the expression, consisting of the first and the second row, is *true*.

6 Conclusion and future work

6.1 Conclusion

In this thesis we showed an approach for proving the well-formedness of class invariants and method specifications using an automated verification system. The well-formedness of these specifications makes it possible to soundly axiomatize the specifications of side-effect free methods for proving the correctness of programs. The discussed approach includes two different ways of generating the proof obligations necessary to prove the well-formedness of expressions and a description how to apply the generation of these proof obligations to actual method specifications and invariants.

The approach was then implemented for the Spec# verification system. This extends the ability of the verification system to detect non well-formed specifications and allows it to support recursive method specifications.

6.2 Future work

There are several things that could be done to extend this thesis.

6.2.1 Recursion on references

Right now, the well-formedness checker only allows recursion on base types like integers or recursion on references, if this reference is a rep field. In the future, it would be nice to support recursion over regular references. This would involve finding an adequate way of defining measures over heap structures and proving that these measures are decreasing. Additionally, whether an automated theorem prover could handle the resulting proof obligations needs to be ascertained.

6.2.2 Indirect recursion

Also recursion that is indirect, e.g. three methods A , B and C where A calls B , B calls C and C calls back to A is not allowed in the approach presented in this thesis. The problem with indirect recursion is, that the proof obligations become rather large, which makes them hard to prove for an automated theorem prover. It would be nice to either find a way to make the proof obligations smaller or some other way to make them provable by an automated theorem prover.

6.2.3 Visual Studio plugin

Up till now, the implementation of the well-formedness checker has to be used from the command line. Eventually, it should be integrated into Visual Studio, like the versions of the Spec# compiler and Boogie without the well-formedness checks, so that warnings for non well-formed specifications can be seen directly in the program code, like regular warnings and errors. This would involve mapping well-formedness errors back to the position in the code, where it occurs.

6.2.4 Lexicographical ordering

A rather small extension of this thesis would be, to implement support for lexicographical ordering. When lexicographical ordering is used, more than one measure can be defined for a recursive specification. The measure is decreasing, if the i -th measure is decreasing, and measures 1 to measure $i-1$ is unchanged since the last step.

6.2.5 Arbitrary measure clauses

The implementation of this thesis only support very simple cases of measure clauses. One can use mathematical operators like addition or subtraction, literals and the parameters of the method. No method calls or field accesses are allowed, because no way of parsing these from the string in the measure attribute. It would be great, if, in the future, arbitrary strings could be parsed for the measure expression.

6.2.6 Overridden methods

Since the Spec# compiler assigns different IDs to two different methods, even if one of them is overriding the other, the implementation of the well-formedness checker does not recognize recursion when they rely on overridden

methods. A future project could change the implementation to support overridden methods.

References

- [1] J.-R. Abrial and L. Mussat. On using conditional definitions in formal theories. In *Conference of B and Z users on Formal Specification and Development in Z and B*, pages 242–269. Springer, 2002.
- [2] Thomas Ball, Shuvendu K. Lahiri, and Madanlal Musuvathi. Zap: Automated theorem proving for software analysis. 2005.
- [3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. 2005.
- [4] Mike Barnett, Robert DeLine, Manuel Faehndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. 2003.
- [5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. 2004.
- [6] P. Behm, L. Burdy, and J.-M. Meyandier. Well defined b. In *Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 29–45. Springer, 1998.
- [7] Adam Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. 2007.
- [8] Darvas and P. Miller. Reasoning about method calls in interface specifications. *JOT*, 2006.
- [9] Robert DeLine and K. Rustan M. Leino. Boogiepl: A typed procedural language for checking object-oriented programs. 2005.
- [10] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. 2003.
- [11] University Of Central Florida. <http://www.eecs.ucf.edu/~leavens/jml/>.
- [12] Microsoft. <http://msdn2.microsoft.com/en-us/vcsharp/default.aspx>.
- [13] R. Hhnl. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IGPL*, 13(4):415–433, 2005.
- [14] K. Rustan. <http://research.microsoft.com/~leino/papers.html>.