FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Petr Hudeček

Soothsharp: A C#-to-Viper translator

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	RNDr. Pavel Parízek Ph.D.
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2017

Title: Soothsharp: A C#-to-Viper translator

Author: Petr Hudeček

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Pavel Parízek Ph.D., Department of Distributed and Dependable Systems

Abstract: Viper is a verification infrastructure developed at ETH Zurich. Using this infrastructure, programs written in the Viper language may be analyzed for correctness with respect to assertions and contracts.

In this thesis, we develop a contracts library and a translator program that compiles C# code into the Viper language and thus allows it to be verified. A user may annotate their C# program with these contracts and then use the translator to determine its functional correctness.

The translator supports most C# features, including types and arrays. It also integrates with Visual Studio, showing translation and verification errors to the user on-the-fly.

Keywords: verification, programming languages, contracts, permissions

I would like to thank my advisor, Dr. Pavel Parízek, for promptly providing me with advice and information whenever I needed it, and for guiding me throughout my master's studies.

I would also like to thank Dr. Malte Schwerhoff and Marco Eilers from the Viper team for their advice and cooperation.

Contents

1. Introduction

The field of *software verification* deals with providing guarantees that a given computer program behaves as expected. In the context of research, software verification is understood to encompass only the more formal techniques such as symbolic analysis or model checking. Unit testing in particular is usually not considered a part of software verification.

Software verification is expensive and still often impractical for large-scale projects. For these reasons, it is used only for those systems where the cost of a potential failure is very high, and expending considerable resources on preventing the failure is still a worthy investment. Still even in that case, only the most critical parts of the code are verified.

Over the last decades, software verification has become easier, faster and more powerful. This trend may be expected to continue as better verification methodologies are invented and performance improves. However, at the same time, the software that needs verification has become larger and more advanced. Even at this time it remains impossible to verify many properties of software developed at the dawn of computers, even though we'd be happy to be able to verify those. And in the meantime, advances in software engineering such as more common use of multithreading and the heap have made verification more difficult.

One way to perform software verification is to check program code with respect to *contracts*, i.e. specifications written by the code author that define what the program's expected result is. The program is then verified in a modular way — each method is verified separately and if all methods verify, then the program is deemed correct. For example, if one created a quicksort subroutine, its contract might state "This subroutine always terminates, and when it does, numbers within the input array will be in ascending order." The code that executes the actual quicksort might be complex and hard to understand, but, with an appropriate infrastructure, a tool called a *verifier* could automatically prove that the subroutine's code never breaks its stated contract. If the verifier succeeds, then we can be sure that the code is correct as long as the contract really states what we want it to. But contracts are usually much simpler than the actual code and more easily checked by humans.

Contracts are also the verification methodology chosen by the Viper project [1, 2], developed at ETH Zurich[1]. The project aims, among other objectives, to create an intermediate verification language that is low-level enough that verifiers are capable of proving its contracts, and yet expressive enough that developing front-ends for the language, or programming in Viper directly, is easy.

In this thesis, we aim to develop a transcompiler — a translation program — that takes C# code files as input and produces a file in the Viper verification language. This way, we allow users to formally analyze the correctness of their C# programs. As C# is a more high-level language than Viper, it is often more expressive and convenient to use.

Translating C# syntax and features into Viper makes up the bulk of this thesis. Translating some features is easy: For example, even though the *for* loop is not

[1] *Eidgenössische Technische Hochschule Zürich*, `http://www.pm.inf.ethz.ch/`

available in Viper, a C# *for* loop can be translated as a Viper *while* loop with only minor work by the transcompiler. On the other hand, some features take more effort to translate correctly. For example, Viper has no concept of types or receivers.

In the second chapter, we describe more formally how Viper's verification infrastructure works and give some information on its theoretical basis — permission logic, symbolic execution and SMT problems. Reading the chapter is not required for understanding the rest of this thesis and if the reader is not interested in learning this information, we encourage them to skip it.

In Chapter 3, we define terms used throughout the thesis and give examples of Viper programs and of the translation.

In Chapter 4, we give a formal yet accessible specification of the Viper language. In the following chapters, we will assume the reader can at least read code in Viper. A basic understanding of the Viper language is expected even of the user of the translator.

In Chapter 5, we define the scope of this project and describe what features of the C# language are supported by the translator.

In Chapters 6 and 7, we give details on the implementation of the translator.

In the remaining chapters, we evaluate the translator, discuss its strengths and limitations and our experience, explain the relationship of this thesis with related work, and then conclude.

2. Background

This chapter contains theoretical background for the verification methodology used in the Viper project. This information is in general not necessary for understanding the Soothsharp translator, but it may help one to understand how the Viper language and framework work.

2.1 Viper

The field of software verification is large and there is a number of different verification methods. A one-line description of Viper could be *"Viper verifies correctness of procedural programs written in a high-level language with respect to contracts specified by the user, focusing on multithreading and heap access correctness."*

Viper is not a theorem prover — it reaches neither the theorem prover's power (some programs can be proved to be correct in a theorem prover such as Coq, but not so in Viper), neither its soundness (a program that passes verification in Viper might still contain errors).

Viper doesn't guarantee soundness — for one, it doesn't guarantee termination; and two, it makes some simplifying assumptions such as that integers are unbounded and an overflow cannot occur[2].

However, Viper is sufficiently powerful to prove the correctness of, for example, sorting algorithms or binary search.

The current primary purpose of the Viper project is to serve as a framework that researchers can use to test or develop new verification methodologies. While it is possible to write "proofs" or write Viper programs directly with the goal of ensuring their correctness, and the language was designed with that in mind (for prototyping purposes), that is not the main goal of Viper as a project.

2.2 Viper internals

Figure 2.1 provides an overview of the structure of the Viper project. The most important subprojects are the frontends and the core projects *Silver*, *Carbon* and *Silicon*[3].

Frontends are tools that translate code in a high-level programming language code into Viper code.

Silver is the former name of the *Viper Intermediate Language*. The code repository *silver* still hosts the parser and type checker for the language (and other code common to both backend verifiers).

Carbon and *Silicon* are "backend verifiers," they are responsible for actually verifying Viper code.

[2]In addition, there might be bugs in the verifier. Unlike theorem provers such as *Coq*, there is "trusted core".

[3]We have also considered naming Soothsharp after a chemical element, but the convention seems to be that Viper frontends don't follow this naming scheme.

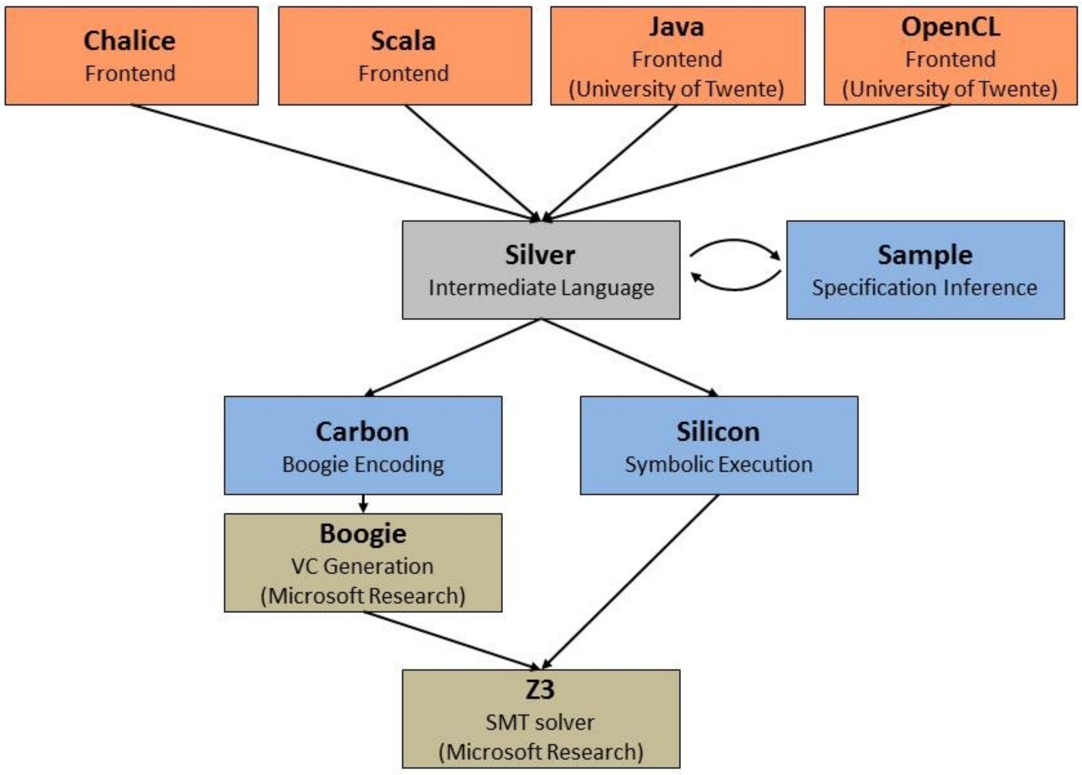


Figure 2.1: Viper architecture [1]

Carbon is a backend that translates a *Viper* code file into a *Boogie* code file which is then translated into queries for the Z3 SMT solver (we explain this process in the next section).

Silicon is a backend that uses a symbolic execution method to generate SMT queries from Viper code, without using Boogie as a go-between step (also explained in the next section).

Each of these backends has its own limitations or bugs, but in theory, they are meant to provide the same verification result for the same Viper code file.

2.3 Verification methodology

A Viper code file ultimately consists of a number of methods and functions that need to be verified. At the time of writing of this text, semantics of the Viper language have not yet been formalized, but we can estimate them: The verifiers are expected to report verification errors if an assertion included in the program is not guaranteed to be true[4].

For example, the verifiers must be able to prove that assertions in each *assert* statement are correct and that method *postconditions* hold, or else they must report a verification error. An error doesn't mean that there is a program execution that would result in the assertion being false, but that the verifier couldn't prove that there isn't.

One verifier (Carbon) works as a translator, similarly to Soothsharp. It takes

[4]The semantics have since been formalized in [3].

Viper code and translates it to Boogie source code. **Boogie** [4] is a Microsoft intermediate verification language and verifier. Syntactically, it is quite similar to Viper itself. The most important difference is that Boogie doesn't use or support fractional permissions. Carbon reduces Viper code to Boogie code and then uses the Boogie verifier to find verification errors in the Boogie code, which are then translated back to Viper errors.

The other verifier (Silicon) performs symbolic execution on the Viper code. Symbolic execution is a kind of static analysis. Each program point is associated with a set of possible program states (by a program state, we mean the values of variables and the contents of the heap, and, in the case of Viper, also known facts and held permissions). For an assertion to hold, each possible program state at that point must satisfy the assertion. To determine whether this is true, Silicon synthesizes SMT queries from the program code that are then fed to **Z3** [5], an SMT solver. This is called "verification condition generation". The answer to those queries is then used by Silicon to produce verification errors.

Carbon (via Boogie) and Silicon both ultimately depend on Z3 for verification. Z3 is a well-known SMT solver, introduced by Microsoft in 2008. SMT solvers receive logic formulas as queries and determine whether they are satisfiable, much like SAT solvers. However, because some common operations in programming (such as integer arithmetic) would result in extremely complex formulas, the SMT solver doesn't reduce them to boolean variables but instead uses a subsystem with some specific knowledge (such as the theory of integer operations) to determine the satisfiability of that formula.

2.4 Reasoning about the heap

An important part of program verification deals with reasoning about locations on the heap. Most verifiers are modular: they verify each method separately. But then, since all methods have access to the heap, each method call has the potential to completely change any value on the heap and you lose almost all reasoning information after the method call.

For example, consider this piece of pseudocode:

```
a = new A();
a.Field = 2;
DoSomething(a);
assert a.Field == 2;
```

The assertion might not be true, because the method *DoSomething* could change the value of *a.Field*. To deal with this, **separation logic** [6] may be used. In separation logic, we consider heap to be a set of distinct regions and for each method, or program point, we maintain information about which regions of the heap the program we can access. Attempting to access other regions of the heap is illegal.

Usually in object-oriented programming, a region would correspond to the memory location of a field of an object, and the heap would be separated into fields you can access and fields you cannot. **Ownership** is a related concept: memory locations, such as fields, may be "owned", which means that they can only be accessed (by reading or writing) via their owner [7]. Many verification methodologies, such as Spec#, use ownership directly.

However, an even more granular approach is possible using **permission logic** and **fractional permissions.** In this paradigm (which is used by Viper), to read a memory location, you must have nonzero permissions (> 0) to it. To write to a memory location, you must have the full permission (1) to it. When memory is first allocated, during object construction, the creator gains the full permission to that memory, and, when calling methods, may then grant a fraction of that permission to that method. If less than the full permission is given (for example, $\frac{1}{2}$ or $\frac{1}{3}$), the called method can only read that memory. If full permission is given (1), the called method can write there as well. Thus, if the caller only gives away partial permission, it is guaranteed that after the method call, that memory location still contains the original value.

Fractional permissions can be used in proofs and never occur in source code [8] but in the case of Viper, they can be used in the code directly.

A use case of fractional permissions that keeps being explored in current research is how they can be used to ensure safe concurrency.

2.5 Roslyn

Before explaining the functioning of our translator, we need to introduce the .NET Compiler Platform, often referred to as “Roslyn”. We make heavy use of its abilities.

Roslyn [9], besides being the host to the C# compiler, also exposes many abilities of the compiler as API. These include lexical and syntax analysis: Roslyn can convert a string or a text file containing C# source code to a syntax tree made of Roslyn classes such as *IfStatementSyntax*.

But they also include semantic analysis. Methods of the Roslyn API can be used to determine which method a call refers to, what is the fully qualified name of an identifier or what is the type of an expression syntax node.

Roslyn can also perform the compilation process in its entirety and determine what compiler errors, if any, would prevent code generation.

More information about Roslyn is available online [10].

3. Overview

The purpose of this thesis is to produce a working software solution that enables developers to write C# code that can be analyzed for functional correctness. To this end, we developed two front-end tools:

- The **csverify.exe** command-line utility. It takes input C# source code files and produces a single Viper source file which it may save on disk or immediately verify using either the Carbon or Silicon verifier. Any errors produced by either the transcompiler or the verifier are linked to a C# line-column location and printed to standard output.
- A **plugin** for Microsoft Visual Studio that translates C# files loaded in the editor and allows the user to have them translated and verified. Any translation or verification errors (reported by a backend verifier) are linked back to nodes in the C# syntax tree and appear in Visual Studio's Error List and as squigglies underlining the errors in the source code.

We also create a contracts library, **Soothsharp.Contracts**, which the user will use in their code to assist the verifier and to specify method contracts.

These, along with the core and the backend, allow the user to verify C# code.

3.1 Example 1 (Maximum)

For example, suppose you have the following C# file:

```
1 class A {
2     public static int Maximum(int a, int b)
3     {
4         return a;
5     }
6 }
```

There is a bug in the above file: the method named *Maximum* always returns the first value, but the compiler cannot know what the method is supposed to do, and so cannot report an error.

With Soothsharp, the user could add *contracts* to the code to specify postconditions that the method body must guarantee:

```
1 using static Soothsharp.Contracts.Contract;
2 class A {
3     public static int Maximum(int a, int b)
4     {
5         Ensures(a > b ? IntegerResult == a : IntegerResult == b);
6         return a;
7     }
8 }
```

The fifth line in this program specifies a method *postcondition*. This one means "If this method was called with arguments a and b, and argument a was greater, then it will be returned, otherwise b will be returned". Postconditions and other Soothsharp contracts are blank methods in C# and so the compiler will still succeed here and leave the bug in the code. However, if the user passes this code to the Soothsharp verifier, an error will trigger:

```
Errors: 1.
5:9 SSIL204: Postcondition of A_Maximum might not hold. Assertion
    (a > b ? res == a : res == b) might not hold. (tmp8116.tmp@2.11)
```

This will alert the user to fix the method, like this, for example:

```
using static Soothsharp.Contracts.Contract;
class A {
    public static int Maximum(int a, int b)
    {
        Ensures(a > b ? IntegerResult == a : IntegerResult == b);
        if (a >= b)
            return a;
        else
            return b;
    }
}
```

Soothsharp will accept this code:

```
> csverify --quiet --silicon example.cs
Verification successful.
```

What happens here is that Soothsharp translates the above code into code written in the Viper Intermediate Language, which looks somewhat like this:

```
method A_Maximum(a : Int, b : Int) returns (res : Int)
    ensures (a > b ? res == a : res == b)
{
    if (a >= b) {
        res := a
        goto end
    } else {
        res := b
        goto end
    }
    label end
}
```

In this particular example, the code is very similar. Of note is only the absence of the *return* statement in Viper and the fact that it names its return values.[5]

This generated code file may then be either printed out or passed to a Viper verifier. In our case, we chose the symbolic execution backend (by using the option `--silicon`). This backend parses a Viper code file and outputs error reports such as the one seen during our first try.

3.2 Example 2 (Binary search)

The example we gave above was very artificial. Verifying such a program is not very useful because the verifier only checks the code with respect to its contracts which still have to be proven to be correct by hand.

However, in the following scenario (adapted from [11]), it is much easier to look at the four lines of contracts at the beginning of the method and see that they are correct than to look for bugs in the entire method:

[5]This is because Viper methods can return more than one value.

```
using Soothsharp.Contracts;
using static Soothsharp.Contracts.Contract;
namespace Soothsharp.Examples
{
    class B
    {
        // Searches a sorted sequence of integers for a value. If
            the value is found, this method returns the index of the
            value in the sequence. If the value is not present, the
            method returns -1.
        int BinarySearch(Seq<int> xs, int key)
        {
            // As a precondition, we assume the sequence is sorted:
            Requires(ForAll(i => ForAll(j => (0 <= i && j <
                xs.Length && i < j).Implies(xs[i] < xs[j]))));
            // The returned integer is either -1 or an index in the
                sequence:
            Ensures(-1 <= IntegerResult && IntegerResult <
                xs.Length);
            // If it's not -1, then the searched value as at the
                returned index:
            Ensures((0 <= IntegerResult).Implies(xs[IntegerResult]
                == key));
            // If it is -1, then the searched value is not in the
                sequence.
            Ensures((-1 == IntegerResult).Implies(ForAll(i => (0 <=
                i && i < xs.Length).Implies(xs[i] != key))));

            // The rest of this method can be assumed to be correct
                if the above contracts are correct
            // and verification passes
            int low = 0;
            int high = xs.Length;
            int index = -1;
            // Binary search follows:
            while (low < high && index == -1)
            {
                Invariant(0 <= low && low <= high && high <=
                    xs.Length);
                Invariant((index == -1).Implies(ForAll(i =>
                (0 <= i && i < xs.Length && !(low <= i && i <
                    high)).Implies(xs[i] != key))));
                Invariant(-1 <= index && index < xs.Length);
                Invariant((0 <= index).Implies(xs[index] == key));

                int mid = (low + high)/2;
                if (xs[mid] < key)
                {
                    low = mid + 1;
                }
                else
                {
                    if (key < xs[mid])
                    {
                        high = mid;
                    }
                    else
                    {
                        index = mid;
                        high = mid;
                    }
                }
            }
            return index;
        }
    }
}
```

They're not perfect contracts. They do guarantee that the result is correct, i.e. that we return a -1 if the value is not found, or the correct index if it is present, but they make no guarantee that the process used to get this index is a binary search, nor that its speed is logarithmic.

Moreover, the contracts don't guarantee *termination* — the method might conceivably run forever. However, if we accept these limitations, the contracts are still a useful tool here. Verifying this code in Soothsharp gives us a guarantee that the method is functionally correct.

You may also note the "*Invariant*" contracts in the loop body. A loop invariant is an assertion that's true at the beginning and end of the loop and also at the beginning of each loop iteration. These invariants are not part of the method's contract but the verifier uses them to be able to prove the method's correctness.

For comparison, this is the Viper code that Soothsharp produces for this example:

```
method B_BinarySearch (this : Ref, xs : Seq[Int], key : Int)
    returns (res : Int)
    requires forall i : Int :: forall j : Int :: (0 <= i && j <
        |xs| && i < j) ==> xs[i] < xs[j]
    ensures -1 <= res && res < |xs|
    ensures (0 <= res) ==> xs[res] == key
    ensures (-1 == res) ==> forall i2 : Int :: (0 <= i2 && i2 <
        |xs|) ==> xs[i2] != key
{
    var low : Int
    low := 0
    var high : Int
    high := |xs|
    var index : Int
    index := -1
    while (low < high && index == -1)
    invariant 0 <= low && low <= high && high <= |xs|
    invariant (index == -1) ==> forall i3 : Int :: (0 <= i3 && i3 <
        |xs| && !(low <= i3 && i3 < high)) ==> xs[i3] != key
    invariant -1 <= index && index < |xs|
    invariant (0 <= index) ==> xs[index] == key
    {
        var mid : Int
        mid := (low + high) \ 2
        if (xs[mid] < key) {
            low := mid + 1
        } else {
            if (key < xs[mid]) {
                high := mid
            } else {
                index := mid
                high := mid
            }
        }
    }
    res := index
}

method B_init () returns (this : Ref) {
    this := new(*)
}
```

3.3 Example 3 (Permissions)

The true strength of Viper, however, lies in its capacity to reason about locations on the heap. A Viper program can access a value stored on the heap only if it has *permission* to access it, and it can overwrite that value only if it has *write permission* to it.

Let's look at this example:

```
 1 using static Soothsharp.Contracts.Contract;
 2 using static Soothsharp.Contracts.Permission;
 3 
 4 namespace Soothsharp.Examples
 5 {
 6     class Data
 7     {
 8         public int Value;
 9     }
10     class C
11     {
12         public static int ReadValue(Data d)
13         {
14             Requires(Acc(d.Value, Wildcard));
15 
16             return d.Value;
17         }
18         public static int ReadValueError(Data d)
19         {
20             Requires(Acc(d.Value, Wildcard));
21 
22             d.Value = 3; // <= This will trigger an error.
23             return d.Value;
24         }
25     }
26 }
```

At line 14, we specify that the method *ReadValue* requires a read permission to heap location *d.Value* ("*wildcard*" here means "any nonzero permission"). This permits the method to access *d.Value* at line 16.

However, the second method requires only a read permission in its precondition, but attempts to write to that location at line 22. It does not have permission to do this, so the verification will fail:

```
Errors: 1.
23:13 SSIL204: Assignment might fail. There might be insufficient
    permission to access d2.Data_Value. (tmp732D.tmp@9.2)
```

This way, one can be sure that a method only modifies heap in a way that's specified by its contract. This is useful both on its own and inside larger proofs.

3.4 Definitions and terms

The following terms are used throughout the rest of this thesis:

- **Viper** ("Verification Infrastructure for Permission-based Reasoning") is both the name of the project and the intermediate language developed as part of the project at the university of ETH Zurich.
  - For many years, the Viper Intermediate Language was called **Silver**. This name persists throughout the source code of both Soothsharp and the Viper project.
- **Silicon** is the name for one of the Viper verifier backends. It uses symbolic execution to prove assertions.
  - **Symbolic execution** is a static code analysis method where facts about the program state are assumed at each point in the control flow graph, and these facts are used to reason about program properties.
- **Carbon** is the name for the other Viper verifier backend. It translates Viper code into the Boogie language and then uses Z3, an SMT solver, to prove the assertions.
  - **Boogie** is a Microsoft intermediate verification language; its code can be compiled into SMT queries understood by Z3
  - **Z3** is a Microsoft SMT solver.
- A **frontend** means, depending on the context, either:
  - A frontend for the Viper language, e.g. Soothsharp or Nagini.
  - A frontend for the Soothsharp translator, i.e. *csverify.exe* or the Visual Studio plugin.
- **Translation** is the process of transforming one or more C# syntax trees into Viper syntax trees.
- **Verification** is the process of using a backend verifier to check assertions in a Viper code file.
- **Roslyn** is the C# compiler library that allows its user to create syntax trees from C# source code and to get the results of the C# compiler semantic analysis from the trees.

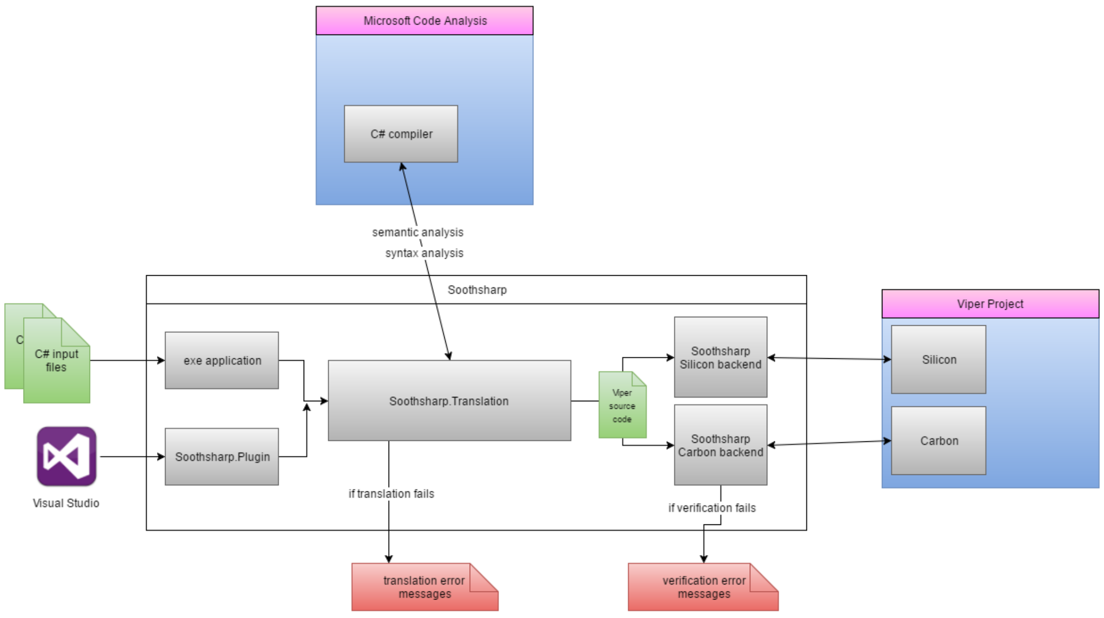


Figure 3.1: Soothsharp architecture

3.5 General architecture

The Soothsharp solution consists of a number of components (see Figure 3.1).

The main project is **Soothsharp.Translation**. This takes C# syntax trees, runs semantic analysis on them (both Microsoft's semantic analysis, and our own) and converts them to Viper syntax trees. Errors may be generated in this process, which would stop the translation. Most of our code is in this project.

To get C# syntax trees into this project, one of two frontends may be used: either the **console application** that generates trees by parsing C# code from files, or the Soothsharp **plugin** for Visual Studio which extracts trees from open documents of a running Visual Studio instance.

The resulting Viper trees are serialized to Viper text which is sent to a backend. The two available backends (Silicon and Carbon) are part of the Viper project.

Messages from these backends are passed back to the frontends and, after some modifications by Soothsharp, are either printed to the standard output or in a Visual Studio window.

Details about this entire process are given in Chapter 6.

4. Viper Intermediate Language

This chapter gives a detailed specification of the Viper language, the target of Soothsharp's translation. The chapter is technical and precise. For a gentler introduction, other documents might be available at the Viper home site [1].

This specification describes the Viper language as it was in November 2016. The language still undergoes active development that sometimes changes even its syntax. We developed this specification during our work on Soothsharp because an official specification for the language does not yet exist. Some examples in this specification are based on personal communication with Dr. Malte Schwerhoff.

Full grammar description of Viper is given as an appendix to this thesis. In this chapter, we first give important basic information and term definitions, and then specify the behavior of individual declarations, contracts, statements and expressions.

In November 2016, semantics of the Viper language have been formalized in [3]. The work assumes some knowledge of the Viper project but it goes into more detail on the semantics of Viper declarations, statements and expressions than we do.

Overview. The Viper language is "an expressive intermediate verification language" [12]. It is a procedural, imperative language with object-oriented elements. Viper programs are not executable, the code is processed by a verifier instead.

Name. Viper is also called the *Viper Intermediate Language.* Previously, Viper was named *Silver, Silver Intermediate Language* and also *Simple Intermediate Language (SIL).* These names are now obsolete, although they all still appear in various documents and source code. *Viper* was originally the name for the entire effort to create a verification framework, but it's recently also come to mean the created language itself.

Extension. Viper code files should have the extension ".vpr", although code files with the extension ".sil" remain common.

Identifiers. Identifiers may contain alphanumeric characters, underscores, dollar signs and must not begin with a digit. The identifier space is mostly flat: fields, functions, methods, domains, predicates and axioms share the same namespace. Local variables, parameters and return value names share the identifier space with global declarations, except that they only have an effect in their scope. It is still a parse error to "shadow" a global identifier with a local one.

Token termination. Definitions (methods, domains, axioms, ...) do not need to be terminated by a newline. For example, `field b : Int field d: Int` is legal. Fields and domain functions may be optionally terminated by a semicolon, but this semicolon is ignored.

Passing verification. A program **passes verification** if its analysis triggers no syntax or semantic errors, and if all of its functions and methods *pass verification.* We can also say that a program or a function or a method **verifies**. This means the same thing.

Types: Viper has the following types:

- **Int**, a signed unbounded integer; specifically, not a 32-bit integer

- **Bool**, a boolean
- **Perm**, a permission value
  - **Rational** is a synonym of **Perm**.
  - A *Perm* value is a rational nonnegative number.
- **Ref**, a reference object
  - All reference objects are assumed to have all fields declared with the *field* declaration.
- Domain types created by *domain* declarations
- **Seq,** an immutable mathematical sequence of elements of a specified type
- **Set**, an immutable mathematical set of elements of a specified type
- **Multiset**, an immutable mathematical multiset of elements of a specified type

Expression. An **expression** is a syntax node that corresponds to the *exp* syntax production. However, the use of this term is discouraged as it may lead to confusion. When possible, the following more specialized terms are used:

- An **assertion** is an expression that's used in contracts, in special Viper statements and as a loop condition and if condition. An assertion will usually contain other expressions. For example, the assertion "2 != 3" contains the subexpressions "2" and "3" which are not assertions themselves.
- A **pure expression** is an expression that has a *type.* It does not contain any *spatial assertions.*
- A **pure assertion** is a pure expression that has the type *boolean.*
- A **spatial assertion** is any other assertion, either an **atomic spatial assertion** or an assertion that combines at least a single *atomic spatial assertion* with other assertions using the operator "`&&`" or "==>".
- An **atomic spatial assertion** is the access predicate *acc* or a user-defined predicate.

Permissions. Permissions are an important concept in Viper. The following terms relate to permissions:

- A **fact** is a logic statement that's known to be true at a point in the program.
- To **assume** a *pure assertion* is to add it to the database of facts at this point. Only boolean assertions can be assumed.
- To **drop** a fact means to remove it from the database of facts and no longer hold it true.
- To **assert** a fact is to check whether the verifier can prove, from its database of facts and by its reasoning, that the asserted fact is true at this point. If the check fails, a verification error is triggered.

- To **inhale** an assertion means to **assume** all pure assertions in the assertion and to **gain the permissions** from that assertion.
  - If you have a permission greater than 1 to a location, you can infer *false*.
- To **exhale** an assertion means to **lose all permissions** from that assertion. If those permissions are not available, a verification error is triggered. The pure assertions in that assertion are **asserted**. Locations which are newly inaccessible are **havocked**.
- To **havoc** a location, variable or value means to lose all information about it.
- You have **read permissions** to a *field* or a *predicate* if you have nonzero permissions to it.
- You have **full permissions** to a *field* or a *predicate* if you have at least permission 1 to it.
  - Any permission greater than 1 causes you to be able to infer *false*.
  - *Full permissions* are sometimes referred to as *write permissions* or even *full write permissions*.

4.1 Example file

This example file (adapted from [13]) that encodes a sorted list of integers might give a good idea of how Viper code files look like in general. All elements in this example are described further below in this specification.

```
field data: Seq[Int]

define sorted(s)
    forall i: Int, j: Int :: 0 <= i && i < j && j < |s| ==> s[i] <=
       s[j]

method insert(this: Ref, elem: Int) returns (idx: Int)
    requires acc(this.data) && sorted(this.data)
    ensures acc(this.data) && sorted(this.data)
    ensures 0 <= idx && idx <= old(|this.data|)
    ensures this.data == old(this.data)[0..idx] ++ Seq(elem) ++
       old(this.data)[idx..]
{
    idx := 0

    while(idx < |this.data| && this.data[idx] < elem)
        invariant acc(this.data, 1/2)
        invariant 0 <= idx && idx <= |this.data|
        invariant forall i: Int :: 0 <= i && i < idx ==>
           this.data[i] < elem
    { idx := idx + 1 }

    this.data := this.data[0..idx] ++ Seq(elem) ++ this.data[idx..]
}
```

4.2 Declarations

A Viper code file (usually with the extension *.vpr*) is a sequence of *declarations.* The declarations are *import, define, domain, field, function, predicate* and *method.* The order of declarations does not matter except for *define* declarations.

The rest of this section specifies the behavior of all declarations.

4.2.1 Import

Syntax: `import "[relative-filename]"`

Processes the declarations in the Viper code file with the given filename as though they were in this file.

Filename is relative to the **primary** file, i.e. the one that is put as a command-line argument, not to the current directory and not to the file where the import declaration was present.

It is an error for a file to import itself directly.

Imports are only ever imported once. If a file imports itself via an intermediary, the second import declaration is ignored.

4.2.2 Define

Syntax: `define [macro-name] [expression-or-block]`

Defines a new *macro* with the name *macro-name* that expands into the given expression or statement block.

Define declarations are processed in text order. If a macro with the same name is already defined, the new declaration is ignored.

Wherever an expression or an identifier might appear, a macro identifier can also appear. In that case, the macro identifier takes precedence and expands first.

A macro may have parameters, in which case the parameters are replaced in the *expression-or-block.* Parameters are replaced as pure source text replacement, i.e. just like C-style macros.

- For expression macros with no parameters, the parentheses may be omitted both at declaration site and at use site.
- For block macros, parentheses are mandatory at both sites.

It is an error to have a cycle of *define* declarations.

4.2.3 Field

Syntax: `field [field-name] :  [type]`

Adds a new field to the database of fields. All reference objects will be considered to have that field. Fields are shared across all loaded files.

4.2.4 Domain

Syntax: `domain [domain-name] {[domain-function]* [axiom]* }`

Syntax: `domain [domain-name] ( [type-argument]* ) { [domain-function]* [axiom]* }`

Adds a new domain to the database of types. A domain may have type parameters. A domain is a type that has no defined operators. The primary way to get a value of a domain type is by the result value of a domain function. This is called "being an **uninterpreted type**".

Other ways for facts to become known about a variable of a domain type exist[6].

A variable of a domain type is not a *Ref* and cannot have fields.

This specification **does not describe** domain type arguments as their definition is still uncertain and they are not very useful. They may be removed in a future version of Viper.

The declaration also adds all nested **domain functions** and **axioms** to the database. Neither for domain functions nor for axioms does it matter in which domain they are defined — they are always global.

Domain function

Syntax: `function [ident] [formal-args] : [type]`

Registers a new global domain function that has no body but has a return type. When called, the domain function will return a value of its return type, with no additional information about it (but see *axioms*).

A function that's called with the same parameters at multiple call sites will always return the same value at all call sites.

If the function is declared **unique**, it merely means that no other domain function can return that value.

Merely declaring a function unique, without ever using it, has an effect. Specifically, if you declare three unique functions returning *Bool*, it is as though *false* was assumed, globally.

Axiom

Syntax: `axiom [ident] { [exp] }`

Registers a new global axiom. The axiom's name is ignored (the parser just checks that it is unique). The axiom's assertion is **assumed** to be true, globally. Usually, one would place quantifier assertions in the axiom. If the axiom is tautologically false, for example, *false* will be assumed globally (and every assertion will be proven).

4.2.5 Subroutines

Syntax:

```
method [ident] [formal-args] [formal-returns]
        [precondition]*
        [postcondition]*
    [block]

```

[6]A value of a domain type can also be obtained as the result value of an abstract function or method or by constraining a fresh variable with the *assume* statement.

```
function [ident] [formal-args] : [return-type]
        [precondition]*
        [postcondition]*
    {
    [exp]
    }

predicate [ident] [formal-args]
    {
    exp
    }
```

Registers a new subroutine (a function, a method, or a predicate). After parsing is complete, all registered subroutines are checked for provability.

The conceptual, most important difference between a function and a method is that the former is *pure* (side-effect free), hence deterministic, and guaranteed to terminate. Due to these properties, functions can be used in specifications, whereas methods cannot.

Specifically, differences between a function and a method are:

- A function's body is a single pure expression. A method's body is a block of statements.
- A function always has one return value, identified by the keyword **result**. A method may have zero or any number of return values.
- A function call may be part of an assertion. A method call is always a statement.

At the beginning of a subroutine's verification, the following become the only known facts:

- All preconditions
- All axioms
- Any facts inferred from global sources such as the uniqueness of functions

To verify a subroutine, the tool starts at its entry point and then proceeds as though by symbolic execution. If all possible execution paths that lead to the exit point pass verification, and all postconditions pass verification for all these paths, then the subroutine passes verification, otherwise it doesn't.

A subroutine need not have a body. In that case, we say that it is **abstract**. An abstract subroutine's body is deemed to be such that it ensures that all the postconditions hold.

A predicate's assertion's type must be **Bool** or it must be a spatial assertion.

In addition, for a subroutine to verify successfully, its preconditions and post-conditions must be well-formed[7].

[7]The most common well-formedness violation is when a contract accesses a field for which it does not have sufficient permission.

4.2.6 Predicates

Predicates behave as subroutines syntactically, during their declaration, but their use is quite different at call sites.

A predicate is, basically, a packaged (usually spatial) assertion that contains permissions. This assertion may be "unfolded" by the *unfold* statement or the *unfolding* expression, and then the contained assertion may be used to access fields.

For non-recursive predicates, there is little advantage to using predicates instead of a *define* declaration which doesn't require folding and unfolding. However, there may also be *recursive* predicates.

For example the following predicate is recursive [14]:

```
predicate lseg(this: Ref, end: Ref)
{
    this != end ==>
    acc(this.data) && acc(this.next) && acc(lseg(this.next, end)) &&
    unfolding acc(lseg(this.next, end)) in this.next != end ==>
        this.data <= this.next.data
}
```

That predicate means "write permissions to the object *this* and to the entire list of objects reachable from *this* by means of the field *this.next*, transitively, until the object *end* is reached".

Folding and unfolding is used to prevent loading all the permissions and all the facts into memory where they're not necessary since the verifier might not be able to reason about that. Instead, only a single "layer" of the predicate is unfolded and exposed to the verifier, which is often all it needs. In the example above, even if the method has *lseg(a, end)* assumed, it still can't access *a.data*. After *unfolding lseg*, it could access *a.data* but not *a.next.data* — it would need to unfold twice to access that field.

More information can be found under *fold* and *unfold* further below.

4.3 Contracts

There are several types of contracts — preconditions, postconditions and invariants. Contracts are assertions.

4.3.1 Precondition

A precondition has an effect at the beginning of a subroutine and just before a method call.

- At the beginning of a subroutine declaration, the subroutine **inhales** all preconditions.
- Just before a method call, the caller **exhales** all preconditions of the method.

4.3.2 Postcondition

A postcondition has an effect at the end of a subroutine and just after a method call.

- At the end of a subroutine's verification, the subroutine **exhales** all postconditions.
- Just after a method call, the caller **inhales** all postconditions of the method.

4.3.3 Invariant

An invariant has an effect in *while* loops. See the statement "while" further below.

4.4 Statements

The following statements exist: *local variable declaration, local definition, local variable assignment, field assignment, object creation, assert, assume, inhale, exhale, fold, unfold, goto, label, if, while, method call, fresh, wand, package, apply, constraining.*

The statements *goto, label, while, if* and *constraining* are involved in execution path modification. Otherwise, all statements merely execute and the execution continues in text order.

In the following, "**local**" means "local variable, parameter or return value".

The statements **wand, package, apply, fresh** and **constraining** are not used in Soothsharp. The first three deal with *magic wands*, an advanced concept in separation logic. The *fresh* statement deals with havocking permission values and the *constraining* statement may be used to implement abstract read permissions. None of these statements will be described in this specification, nor are they used within the Soothsharp program.

4.4.1 If

If statement consists of a *pure assertion*, a *then block*, and optionally any number of *elsif* branches and optionally a final *else* block.

The *if statement* causes the control graph to branch. For each block, a new path is created.

At the beginning of the *then* block, the statement's *pure assertion* is **assumed**. At the beginning of each *elsif* branch, negations of the preceding *pure assertions* are **assumed** and the current branch's *pure assertion* is **assumed**. At the beginning of the *else* block, the negation of all preceding *pure assertions* is **assumed**. At the end of each of the blocks, all of these assumptions are dropped and execution transfers unconditionally to the code point after the *if statement.*

4.4.2 While

While statement consists of a *pure assertion*, a number of *invariants* and a *block.*

The *while statement* causes the control graph to branch into two branches.

- In the first, all invariants are **exhaled** ("an invariant must **hold on entry**"), then all local variables used in the loop body are **havocked**, then all invariants are **inhaled**, then the negation of the *pure assertion* is **assumed**, and execution transfers to the end of the loop.
- In the second, **all permissions are dropped**, then all invariants are **inhaled**, then the *pure assertion* is **assumed**, then the loop body occurs, then all invariants are **exhaled**, then execution ends.

No Invariants on Abrupt End: Jumping out of a loop abruptly (i.e. by a *goto* statement) will *not* cause the invariant to have an effect.

No Termination Check: There is no check that the loop condition can ever be violated, i.e. whether the program terminates or not is not checked by Viper verifiers.

4.4.3 Label

Label marks a point that can be referenced by a *labeled old expression* and by the *goto* statement. It has no semantic meaning on its own.

4.4.4 Goto

Goto transfers control unconditionally to the target label in the same method.

If a statement is unreachable because of a *goto* jump, then *false* is **assumed** for that statement.

4.4.5 Var (local variable declaration)

Declares a new *variable* with a specified *type* and optionally assigns an *expression* as its value. Note that in Viper, object creation and method call are *statements*, not *expressions*. If no expression is given, this does not mean that the value is random or none; it merely means that the verifier does not yet know what it is.

A variable cannot be accessed before it is declared.

4.4.6 Define (local definition)

Works as a global *define* declaration, with the following differences:

- The *macro* only exists in this method.
- The *define* statement can be anywhere in the method, even after its use.
- A local definition shadows a global define declaration without an error or a warning. A local definition thus takes precedence.

4.4.7 Local variable assignment

Sets the value of the local variable being assigned to the value of the right-hand side of the assignment (a *pure expression*). The local and the pure assertion must have the same type.

Note that facts are associated with *values*, not with *variables*. For example, thee following code verifies successfully:

```
method test() {
    var x: Int
    assume 0 < x
    var y: Int := x

    x := x - 10000
    assert 0 < y
}
```

4.4.8 Field assignment

Sets a *pure expression* as the value of a field. The field and the pure expression must have the same type.

The method must have full write permissions to the field at this point, and it keeps these permissions after the assignment.

4.4.9 Object creation

Creates a new *reference object* and sets it as the value of a local.

In addition, for all specified fields, **full write permissions** to those fields are **inhaled**. If the *new(*)* variant is used, full write permissions are **inhaled** for all fields in the database of fields (for the newly created object only).

4.4.10 Assert

The *assertion* is **asserted** (see the beginning of Chapter 4 for definitions).

The verifier must prove that the assertion is provable, i.e. its pure assertions are true and that you have permissions for all spatial assertions in the assertion. If it doesn't, then **verification fails**.

4.4.11 Assume

The *assertion* is **assumed**.

The pure assertion is added to the database of facts. The current implementation, however, considers *assume* to be a synonym of *inhale* and allows spatial assertions to be assumed as well (they are, however, inhaled).

4.4.12 Inhale

The assertion is **inhaled**.

Any pure assertions in the *assertion* are **assumed**, and permissions to any spatial assertions are **gained**.

4.4.13 Exhale

The assertion is **exhaled**.

Any pure assertions in the *assertion* are **asserted**. For spatial assertions, the permissions to those elements **are lost**. If you don't have those permissions, then **verification fails**.

4.4.14 Fold

Syntax: `fold [predicate-call]` *or* `fold [access-expression]`

The access expression must be to a user-defined predicate.

The "*fold [predicate-call]*" syntax is equivalent to "*fold acc([predicate-call], write)*".

The assertion within the predicate is **asserted** but, for all access expressions within the predicate, the permission amount checked is first multiplied by the permission amount given in the fold statement.

Then, permissions to the assertion in that predicate are "lost" and cannot be used until the predicate is unfolded again, *but* the verifier can still assume that others don't have that permission access to this predicate. This may be relevant in some situations.

Also, permissions to the predicate itself **are gained**, based on the permission amount given.

Example of "fold":

```
field f : Int
predicate p(k : Ref) {
    acc (k.f, 1/2)
}
method test(l : Ref) {
    assume acc(l.f, 1/2) // We have half permission
    fold acc(p(l), 1/2) // We only have 1/4 permission left,
    // but we also have 1/2 permission to p(l)
    assert acc(l.f, 1/4) // succeeds
    assert acc(l.f, 1/2) // fails!
}
```

4.4.15 Unfold

As *fold*, except that instead of *asserting* and "losing" permissions, it **inhales** the assertion within the predicate, as modified by the multiplier given as part of the unfold statement. Also, these permissions to the predicate itself are **exhaled**.

4.4.16 Method call

The following happens in order:

- First, the *preconditions* of the called method are **exhaled**.
- Second, return value receiving variables are **havocked**.
- Third, the method's *postconditions* are **inhaled** .

4.5 Expressions

The following expressions exist: *literal, local, parenthesized expression, arithmetic expression, conditional expression, old expression, labeled old expression, apply old expression, permission literal, perm-expression, accessibility predicate, function application, field access, predicate access, inhale-exhale expression, unfolding, folding, applying, packaging, forall, exists, seq constructor, set constructors, sequence length, let-in expression, forperm expression.*

The expressions **apply old, applying, forperm** and **packaging** are not used within Soothsharp and are not explained in this document.

4.5.1 Local

Syntax: `[identifier]`

Represents a local variable, parameter or return value and returns its value. It is not possible to refer to locals within access expressions — the program always has permission to access locals.

4.5.2 Parenthesized expression

Syntax: `( [expression] )`

Does the same as *[expression].* This is used for clarity or operator precedence.

4.5.3 Literals

The **Bool** literals are **true** and **false**.

The **Ref** null literal is **null**.

The **Int** literals are all integers written as decimals.

The keyword **result** represents the return value of a function.

There are four **Perm** literals:

- **none** which is the same as 0/1;
- **write** which is the same as 1/1;
- **epsilon** which is deprecated and should not be used
- **wildcard** which is a special permission literal that represents any non-zero permission.

4.5.4 Operator expression

An operator expression consists of one or two operands and an operator. The operator must be able to work on those operands. The following operators exist:

- Unary plus, unary minus: works on integers and rationals
- Unary negation: works only on booleans
- `subset`: works on sets
- `in`: works on a set, multiset, sequence and its member type

- `++`: works on two sequences of the same type
- `union`, `intersection`, `setminus`: works on two sets or multisets of the same type
- `--*` (magic wand): not described in this specification
- \ (division), `%` (modulo): works only on integers
- `+`,`-`,`*`: works on integers and rationals
- <,<=,>,>=: works on integers and rationals
- ==, `!=`: works on any two pure expressions of the same type
- `||`, <==>: works on booleans
- `&&`: works on booleans and spatial assertions
- ==>: left-hand side must be a boolean, right-hand side may be a boolean or a spatial assertion

If both operands of the "`&&`" operator are boolean assertions, the resulting operator expression is pure and boolean. If at least one of them is a spatial assertion, then the resulting operator expression is spatial and has no type and "combines" all boolean and spatial assertions from its operands (recursively). The same holds for the ==> operator.

4.5.5 Conditional expression

Syntax: `[test] ? [then] : [else]`
If the pure assertion *test* is true, then the result is *then*, otherwise the result is *else*. *then* and *else* must have the same type. If one of them is spatial, then the other must be either spatial or boolean.
If either *then* or *else* is spatial, the expression is spatial, otherwise it's pure.

4.5.6 Old expression

Returns the value of its argument, which must be pure, as it was at the beginning of the subroutine (if no label is specified) or at the specified label.

4.5.7 Permission access (`perm`)

Returns the *Perm* value that represents the permissions the method has on the *field* or *applied predicate* at that point.

4.5.8 Inhale-exhale expression

This expression should be part of a precondition or postcondition. It contains two assertions.
If the inhale-exhale expression is *inhaled*, then only the first assertion is **inhaled**.
If the inhale-exhale expression is *exhaled* or *asserted*, then only the second assertion is **exhaled** or **asserted**.

4.5.9 Access expression (`acc`)

An access expression is used within contracts and *assert, inhale, exhale* and *assume* statements. It means that the program has permissions to access the specified location at that point. If no permission amount is specified as the second argument, the permission amount is 1/1 (write permissions).

The first argument may be a field access or a predicate access.

4.5.10 Function application

Syntax: `[functionname] ([arguments])`

Represents the return value of a function. Functions are deterministic and pure: the verifier will assume that two function applications with the same arguments return the same value. A function application works in many respects as a method call, except that it is pure and can be used within expressions, even more complex expressions.

At the point where a function is used, the function's preconditions are **asserted**, but not exhaled.

4.5.11 Field access

Syntax: `[object].[fieldname]`

Access a field of an object. Field access may appear within access expressions, assignments or directly as a value.

4.5.12 Predicate access

Syntax: `[predicate]([arguments])`

Also referred to as "predicate call" in this specification, it represents an *applied instance* of the predicate. It may appear within access expressions and fold/unfold statements. This is how one refers to a predicate outside their definition.

4.5.13 Folding/unfolding

A *folding* expression is as a normal expression, but before its "evaluation" begins, the specified predicate is folded as if by the *fold* statement, and when the expression finishes evaluating, the predicate is unfolded again.

Notably, it can be used within functions.

An *unfolding* expression acts analogously.

4.5.14 Quantifiers

There are two quantifiers: **forall** and **exists**. While they can be used anywhere a boolean expression is permitted, they only make sense as assertions.

A **forall** assertion is true if the inner assertion holds for all possible instances of the specified types. An **exists** assertion is true if the inner assertion holds for at least one combination of values of the specified types.

The verifier cannot easily reason with quantifiers: You might have an assumed assertion and yet the verifier might not use it. This is because the verifier only applies quantified expressions when they are *triggered*.

Each quantifier may be associated with one or more triggers. A trigger is a syntax element, such as a function application or a field access. When the verifier reaches this element, it instantiates the quantified expression for that case.

If no trigger is specified for a quantifier, the Viper program may try to infer the trigger automatically, but this is not always successful.

4.5.15 Mathematical object constructor

Creates a new mathematical sequence, set or multiset, either from a type or from an explicit sequence. You must specify one or the other, but not both. Mathematical objects are immutable, although you may create new mathematical objects from old ones using operators such as "`++`", "`subset`" or a special sequence constructor.

The following special sequence constructors exist:

- `sequence[index]`: Returns the element at index *index*. Sequences are zero-based. This doesn't create a new mathematical object.
- `sequence[..count]`: **Take**. Returns a new sequence with the first *count* elements.
- `sequence[count..]`: **Drop**. Returns a new sequence without the first *count* elements.
- `sequence[dropcount..takecount]`: **Take & Drop**. Returns a new sequence as though first the **take** and then the **drop** operations were performed.
- `[minimum..maximum)`: Returns a new `Seq[Int]` that contains each integer starting with *minimum* and ending just before *maximum*.

4.5.16 Sequence length

Syntax: `|seq|`
Returns the number of elements in a sequence, set or multiset.

4.5.17 Let-in

Syntax: `let [identifier] == ( [value] ) in [expression]`
Assigns the value (in parentheses) to the *identifier*, as though the identifier were a local variable inside the *expression*. The *value* must be a pure assertion and **must be in parentheses**. The *identifier* uses the global namespace.

5. Supported C# features

The C# language is extensive and it would take significant time to create a program that would translate it perfectly. For some features, it would even be impossible because they cannot be encoded in Viper. In this section, we go through the C# language specification and we state, for each feature, whether our translator should support it or not.

For starters, we will not support any C# feature from version 6.0 onwards. This is mostly because no public specification for C# 6 exists, and also because these new features are mostly syntax improvements with little research value.

In the following comprehensive list of C# features, taken from the specification, black items are supported, but red underlined items are explicitly out of scope (any feature introduced in C# 6 or later is out of scope). If an item is supported, it still might not be *fully* supported. For example, variable declarations might support only "`int i; int j`" and not "`int i,j;`". The **bold** captions are names of chapters of the C# language specification.

In the parentheses after each unsupported feature is a character that encodes the reason why the feature is unsupported. The meanings of these characters are given after the list.

- **Lexical structure:** pre-processing directives (U)
- **Basic concepts:** member access, scopes, name hiding, qualified and short names, memory model (D), unsafe code (D)
- **Types:** value types, default constructors, int, bool, reference types, object, byte, sbyte, short, uint, long, char (D), float (V), double (V), decimal (V), dynamic (D), string (D), boxing (U), constructed types (D), expression trees (D)
- **Variables:** instance fields, array elements, value parameters, local variables, static fields (N), reference parameters (N), output parameters (N), default values (N)
- **Conversions:** conversions (N)
- **Expressions:** operators, member lookup, overload resolution, literals, simple names, parenthesized expressions, member access, invocation, element access, this access, increment, decrement, new, ternary operator, assignment operators, constant, dynamic binding (D!), operator overloading (N), base access (D), typeof (D), checked (V), unchecked (V), default value (N), anonymous method (D), null coalescing operator (N), LINQ query (D)
  - **Operators**: as supported by Silver
- **Statements:** empty statement, block, labeled statement, local variable declaration, local constant declarations, expression statement, if, while, do, for, goto, return, switch (S), foreach (D), break (S), continue (S), throw (D), try (D), checked (V), unchecked (V), lock (D), using (U), yield (D!)
- **Namespaces:** namespaces, using aliases, extern aliases (U), nested namespaces (U),

- **Classes:** classes, instance member, constants, constructors, partial classes (S), generics (D), inheritance (D), GetType (D), static member (D), nested types (S), properties (S), events (D), indexers (S), static constructors (D), destructors (V), iterators (D), async functions (D!)
  - **Fields**: fields, readonly (N), volatile (D), field initializer (N)
  - **Methods**: methods, overloading, overriding (D), extension methods (N)
  - **Operators**: integer operators, custom operators (N)
- **Structs:** structs (N)
- **Arrays:** arrays, array creation expression, array element access, array length, array initializer, array covariance (D), arrays with inheritance (D)
- **Interfaces:** interfaces (D)
- **Enums:** enums
- **Delegates:** delegates (D!), lambda functions (D!)
- **Exceptions:** exceptions (D)
- **Attributes:** Soothsharp attributes, custom attributes (U)
- **Silver functionality:** field, function, contracts, predicate, method, new object creation, assert, assume, inhale, exhale, fold, unfold, Seq, sequence concatenation, old expression, permissions (none, write, wildcard, fractional), perm(), acc(), unfolding, forall, exists, domain (NC), domain function (NC), domain axiom (NC), fresh (NC), constraining (NC), Set (NC), union (NC), intersection (NC), setminus (NC), set membership (NC), subset relation (NC), triggers (D), Multiset (NC), domain type instance (NC)

In the next chapter, we will describe how all of these C# features are translated into Viper.

5.1 Rationale

We support a reasonable subset of C# — it is possible to code useful programs and algorithms using only the features that are supported by the Soothsharp translator. We would still like to provide some explanation behind why the excluded features were not implemented.

- (U) "uncommon": These features are almost never used in real C# code which is why they should be low-priority.
- (D) "difficult": It would be too difficult to implement this, therefore we decided to put it out of scope of this thesis.
  - (D!) "extremely difficult"

- (V) “Viper”: It is impossible to encode this in Viper. These features mostly deal with integer overflow/underflow, affected by the *checked* and *unchecked* contexts, and would require us to implement a bounded numeric data type in Viper which would be difficult; floating-point numbers are also unsupported. As it is, Viper only supports unbounded integers.
- (S) “syntax only”: This is basically syntactic sugar and can be easily replaced by other, supported, C# constructs. The statement *switch* may be replaced by *if*, *break* and *continue* may be replaced by *goto*. Nested types may be put on the top level. Properties and indexers can be replaced by fields and methods.
- (N) “not interesting”: Translating these features holds no research interest and it’s not indispensable. They don’t add strength to the C# language and with some effort, the user might be able to replace them.
- (NC) “not C#”: These are Viper features that we might backport into C# but we chose not to, because either they would be difficult to use or difficult to implement.

For some C# features that are difficult to implement in Soothsharp, we would like to present the reasons for that:

Unsafe code: Unsafe code makes use of memory pointers and allows for pointer arithmetic. There is fundamentally no way to make this verifiable using Viper.

Inheritance: Support for inheritance, by which we mean casting, the *typeof* operator, and overriding methods is possible but would take time. Casting and *typeof* could be implemented by adding a field to every instance that would determine its class and by adding axioms for subtyping, for example as a Viper domain (this is the way Scala2Sil [15] does this). Implementing overriding methods is a harder problem because we cannot permit arbitrary preconditions and postconditions, and so there would be inherent limits on what an overridden method could do. This is explained in more detail in [16].

Static fields: Viper expects no global state to exist and, aside from constants, doesn’t have the syntax to express it. The way around this would be to add an extra parameter to each method that would contain a “global state object”, which would have, as fields, all the static fields in the program.

Delegates: Delegates allow the user to basically store method pointers. At a point when the delegate is executed, what do we know about the result of the method call and of its arguments? How do we know we have enough permissions to call all the possible methods in the delegate? A possible solution would be to only allow delegates to point to methods that have been somehow marked as possible targets in a method’s precondition. Then, when we call that delegate, we can exhale the conjunction of preconditions of possible targets, and inhale the disjunction of postconditions of possible targets, and thus maintain soundness. To our knowledge, no Viper frontend, or any similar tool, has tried to do this yet.

6. Translation

In this chapter, we explain how the translation process operates, step-by-step.

Soothsharp attempts to translate C# code, as closely and accurately as possible, to Viper. However, some unsoundness will be introduced, even if Soothsharp did not have any bugs. For example, C# bounded integers (*System.Int32*) are translated as Viper mathematical integers (*Int*) and overflow problems are therefore ignored.

We will point out these problems where they occur in the translation phase, but much like the Viper team itself, we are unable to provide a rigorous proof of the translation process's correctness.

An explanation of why we chose to perform translation this way can be found at the end of this chapter.

6.1 Definitions

We will use the following phrasing in the translation process description.

- To "trigger an error" is to add it to the list of errors that will be returned to the frontend.
- "[This] may not trigger any errors." means that we don't expect errors to be triggered during [this] procedure. An exception may still occur but if it does, it is a bug and the translation process will usually be aborted.
- "[An item] is remembered" means that somewhere, [the item] is stored in memory (or at least some information about the item is stored) and that information will come into play later in the process.
- A silvernode is a node of the Viper abstract syntax tree. A silvertree is the syntax tree. The *master silvernode tree* is the concatenation of Viper syntax trees of all files that are translated. The term "silvernode" comes from a previous name for the Viper language when it was still called the "Silver Intermediate Language".
- A sharpnode is a node of our own C# abstract syntax tree.
- A Roslyn node is a node of Microsoft .NET Compiler Platform's abstract syntax tree.
- "[An item] is ignored" means that nothing is done to it in this phase, and if it's a translation phase, it translates into nothing.
- A silvername is an identifier in the Viper language that corresponds to an identifier in C# code. Again, the name "silver" comes from an earlier name for the Viper language.

6.2 Overview of the translation process

The translation process takes as input one or more C# syntax trees and may output, depending on settings, Viper code files, translation or verification errors or the information that the input was successfully verified.

The process consists of 14 steps. Some of them (marked in bold) are named phases that are detailed further below.

1. The frontend supplies one or more C# syntax trees. A tree may be marked for *full translation* or only for *signature extraction* (see 6.12).
   (a) If the frontend is the *csverify* executable, the user will supply *.cs* source files and *.dll* reference assemblies. The front-end will use the Roslyn API to create a syntax tree from the files. All files, even if malformed, will output a syntax tree. Reference assemblies will not create any trees but will be passed to the compiler in order to make semantic analysis work correctly. Otherwise each source file creates a single tree. Full verification may be launched if the user requests it via a command line option.
   (b) If the frontend is the Visual Studio plugin background scan, it will supply the syntax tree of a single document. No reference assemblies will be loaded and other documents in the project will also be ignored. The tree will always be marked for full translation. Full verification will always be launched.
2. A new translation process is created.
   (a) The translation process determines what reference assemblies are associated with the process and adds the *mscorlib* assembly and the *Soothsharp.Contracts* assembly, if not already associated.
   (b) In addition, the *System.Core* assembly will be referenced. None of the types in that assembly can actually be used by a verifiable program. However, the directive "*using System.Linq;*" is generated by default in Visual Studio and referencing this assembly will prevent a C# compiler error on that line.
3. The Roslyn object *CSharpCompilation* is created. It contains all the trees and all the references.
4. The Roslyn semantic model is created, starting semantic analysis for all trees simultaneously.
5. If any error occurred during semantic analysis, the process is aborted and ends with failure.
6. The following phases occur, in order, for all syntax trees. First, the first syntax tree is fully processed, then the next one, and so on.
   (a) **Conversion phase.** The tree is converted to Sharpnode intermediate representation ("the sharpnode tree"). This phase cannot trigger any errors. If it throws an exception anyway, the translation of this tree is aborted.
   (b) **Collection phase:** The sharpnode tree is scanned for types and fields, which are collected into the translation process. Any errors are remembered.

   (c) **Main phase:** The sharpnode tree is scanned for methods which are converted into a single Viper Intermediate Language syntax tree ("a silvernode tree"). Main phase occurs also for trees that are marked for signature extraction only but all methods in those trees are considered to be marked *[Abstract]*. Any errors are remembered.

7. All silvernode trees are concatenated into a single master silvernode tree.
8. **Global addition phase:** Additional silvernodes are added to the master tree that are global for the translation process; these are, for example, field definitions or additional code for arrays. No errors may trigger.
9. **Optimization phase.** The master silvernode tree is optimized and some silvernodes may be removed. However, semantics will remain the same.
10. **Name assignment phase.** Identifiers are assigned their names in the Viper source code ("silvernames"). These are guaranteed to be unique across the master silvernode tree.
11. **Postprocessing phase.** Postprocessing is launched for the master silvernode tree. The postprocessing deals mainly with pretty indentation.
12. If there are any remembered errors, they are sent to the frontend and the process is aborted.
13. The translation is reported as successful.
14. **Verification.** If the process is marked for full verification, the master silvernode tree is converted to source code text and passed to the chosen backend.
    (a) The Carbon or Silicon verifier runs, producing output.
    (b) The verifier output is parsed and a list of verification error messages is extracted from it.
    (c) If these error messages are associated with a line and column ("location") in the Viper source code, they instead become associated with a C# syntax tree node that caused that Viper code location to exist.
    (d) If there are any error messages, they are sent to the frontend.
    (e) If not, the verification is reported as successful.

The translation process for expressions on the left-hand side of assignments and prepending method calls were the two most difficult parts of this system to implement. We give more details on this in Section 8.1.

6.3 Conversion phase

During the *conversion phase*, Roslyn nodes are transformed into sharpnodes. This is useful because Roslyn nodes contain a lot of information that is unnecessary for us, don't sometimes contain information that we need, and most importantly, cannot be extended with virtual methods.

This phase will not normally trigger errors. If it does anyway (due to a bug), the conversion phase may throw an exception which is caught and put to the user as error SSIL103 and the translation of this tree is aborted (the collection and main phases do not occur).

Roslyn nodes are converted mostly on a one-for-one basic. For example, *NamespaceDeclarationSyntax* (a Roslyn node) becomes *NamespaceSharpnode*. *ClassDeclarationSyntax* becomes *ClassSharpnode* etc.

If a Roslyn node is encountered that we either don't have a sharpnode for (for example, a struct declaration) or that we don't have a sharpnode for in this context, a *DiagnosticSharpnode* or *UnknownSharpnode* is created. The purpose of these special sharpnodes is to trigger an error later on in the main phase. We do not trigger the error right here because we want the conversion phase to be free of errors and to have it happen entirely within constructors.

At some locations, nodes of many different kinds may occur. Specifically, these locations are:

- where a class member is expected (there may be a method declaration, a field declaration, a constructor declaration, or a declaration that we cannot convert);
- where a statement is expected; and
- where an expression is expected.

The static class *RoslynToSharpnode* calls an appropriate sharpnode constructor based on the *kind* of the Roslyn node in each of these three locations. For example, in a location where an expression is expected, the method *RoslynToSharpnode.MapExpression* might see that the node has the kind *GreaterThanExpression* and so it will call the constructor of *BinaryExpressionSharpnode*, with appropriate arguments.

Each sharpnode will retain a reference to the Roslyn node that it was transformed from. This node will be called "original node". The purpose of this is twofold:

- First, during the main phase, this reference is how we can access semantic information about the node (e.g. the type of a variable).
- Second, if an error triggers in this sharpnode, the original node will serve as the location of the error that's reported to the user (Roslyn nodes keep information about location in a code file; sharpnodes don't).

6.4 Collection phase

In the collection phase, we walk through the sharpnode tree using a depth-first search and we don't use any context.

Depending on the type of the sharpnode we're traversing, different actions may occur. For compilation units and namespaces, we simply descend into their children. *ClassSharpnode* elements, however, register themselves and all of their fields. The search does not descend into methods or fields.

Registering a class means:

- Depending on the attributes and verification settings (see Section 6.12), the class might or might not be collected. If it's not collected, it's ignored and none of its fields are registered either.
- Otherwise, the class is remembered. The remembered information is the class name, what fields it contains and whether the class is static. This information will be used in the global addition phase (Section 6.6) to create initializer code.

Registering a field means:

- If the field is a constant, it is ignored (because constants are inlined later on in the main phase).
- If the field is static, an error triggers.
- Otherwise, the field's name becomes its base silvername (see Section 6.8), we determine the field's Viper type (or trigger an error if the type translation fails) and the field is remembered.

6.5 Main phase

In the main phase, we also walk through the sharpnode tree using a depth-first search algorithm. However, this time, sharpnodes themselves are responsible for translating their children; also, whenever a sharpnode starts being translated, it receives a *context* providing additional information.

The main phase only occurs for a tree with no compiler errors. If the C# compiler library detects an error in the tree, the tree is not translated.

When a sharpnode is being translated, it will usually first translate its children and then use the results of their translation to provide its own Viper output.

6.5.1 Context

Depending on the context and on the type of the sharpnode, different actions may occur when that sharpnode goes through the main phase.

When a sharpnode causes its child to be translated, it may modify the context for that child. The modified context may then be propagated further down the tree that has that child as a root.

The context includes the following flags that are common for the entire tree and are never modified by sharpnodes:

- *MarkEverythingAbstract* which is set when the tree is to be translated as signatures-only, without translating method bodies (this happens with the `--assume` option of *csverify.exe*)
- *VerifyUnmarkedItems* which is set when classes and methods with neither the *[Verified]* nor *[Unverified]* attributes should be considered marked *[Verified]*. If not set, then they should be considered marked *[Unverified]*.

The context also includes these properties which may be modified by sharpnodes:

- *IsFunctionOrPredicateBlock*, which is set when the sharpnode being translated is part of a C# method that is meant to be translated to a Viper function or predicate, rather than a Viper method.
- *PurityContext* which may be set to one of three options[8]:
  - *PurityNotRequired*: Translated expressions may result in Viper statements.
  - *Purifiable*: The expression must not translate into a statement, but it will be possible to prepend a statement (see *Prepending statements* below, Section 6.5.5).
  - *PureOrFail*: If an expression would translate into a statement, an error triggers.

6.5.2 High-level constructions

Compilation units, namespace declarations and class declarations are translated as a concatenation of their children's translations.

Depending on verification settings (Section 6.12), a class may be ignored.

Class members are handled thus:

- Methods are translated (see below).
- Constructors are translated (see below).[9]
- Fields are ignored.
- Any other members or nested types trigger an error.

6.5.3 Methods and constructors

A C# method, in general, has the following form:

```
[attributes]
[modifiers] [return-type] method-name ( [parameters] ) {
    [contracts]
    [statements]
}
```

All modifiers except for **static** are ignored. Access modifiers don't need to be translated at all, and Soothsharp doesn't support inheritance so virtual methods are considered the same as normal methods.

A C# method may have the *[Pure]* attribute. Such a method is called "*declared pure*" and will be translated into a Viper function. It may also have the *[Predicate]* attribute which will cause it to be translated as a Viper predicate. Other methods are impure and are translated as Viper methods. A method may not be declared *[Pure]* and *[Predicate]* at the same time.

A method may also have the *[Abstract]* or *[SignatureOnly]* attribute which causes it to be translated as an abstract Viper method, function or predicate.

[8]In this chapter, by "purity", we mean "not having side-effects", as opposed to purity as used by Viper where it means "having a type and not being *spatial*"

[9]There is currently a bug where a class with multiple user-defined constructors fails to translate.

The same effect happens if the file that contains the C# method is marked for signature extraction only (see 6.12). An abstract Viper method, function or predicate merely lacks a body (the part from the opening brace to closing brace, including the braces, is excluded).

Verification settings may also cause a method to be ignored altogether.

The method name is translated as an identifier. The return type, unless void, is translated as a type (see *type conversion*, Section 6.13).

A parameter, in C#, is `[type] [name]`, which is translated as `[name]' : [type]'`.

Method

An impure method is translated into Viper thus:

```
method [name]' ( this : Ref, [parameters]' ) returns ( res :
    [type]')
    [contracts]'
{
    [statements]'
    label end
}
```

If the C# method's return type is *void*, the **returns** clause is not present. If the method is static, the "`this : Ref`" parameter is not present. The statements are translated in a *PurityNotRequired* context.

Function

A method declared *[Pure]* is translated thus:

```
function [name]' ( this : Ref, [parameters]' ) : [type]'
    [contracts]'
{
    [statements]'
}
```

It is an error to declare a method without a return type pure. If the method is static, the "`this : Ref`" parameter is not present. The method's body is translated in a *PureOrFail* context and must only contain contracts and a single return statement.

Predicate

A method declared *[Predicate]* is translated thus:

```
predicate [name]' ( this : Ref, [parameters]' )
    [contracts]'
{
    [statements]'
}
```

It is an error to declare a method with a return type other than *bool* a predicate. If the method is static, the "`this : Ref`" parameter is not present. The method's body is translated in a *PureOrFail* context and must only contain contracts and a single return statement.

Constructor

A constructor is translated into a Viper method as above but there are a couple of differences.

The return type is always *Ref*. The base silvername of the Viper method will be the translated class name followed by the tag "ctor" (see also 6.8). The return type is then named "`this`".

At the beginning of the constructor, the class's initializer is called.

A constructor is therefore translated thus:

```
method ClassName_ctor ([parameters]') returns (this : Ref)
    [contracts]'
{
    this := ClassName_init ()
    [statements]'
}
```

Unlike the default constructor (which we call "initializer"), a user-defined constructor will not return full write permission to all its fields by default. If that is something the user wants, they need to add the *Ensures* postconditions themselves. This allows for more control by the user.

6.5.4 Statements

In this section, we'll show how C# statements are translated.

An apostrophe after a word means "the translated version of this". The word might be an identifier, an expression, a statement or something else that can be translated.

Table 6.1 summarizes how statements are translated into Viper.

- **Empty statement** is ignored.
- **Block:** A C# block is translated into a block silvernode, with all C# statements within translated, with the following caveats:
    - If any of the children are contracts:
        - If the block is the main code block of a loop or method, they are returned to the parent as user-specified contracts. They are not translated as Viper statements.
    - If the block is the main block of a method declared pure, it must only have a single child that's not a verification condition and that must be a return statement. If not, an error triggers.
- **Do/while/if/for:** No special comment, but see *Prepending statements* below. The expressions are translated in a *Purifiable* context.
- **Goto/labeled statement:** Translation is straightforward.
- **Local variable declaration:** A local variable declaration may declare only a single variable. This is a limitation of Soothsharp that could be removed in the future.

Statement	**C# code**	**Viper code**
Empty statement	;	(ignored)
Block	{ code }	{ code' }

Do	**do** { code } **while** (condition)	code'
		prepended-statements
		while (condition') {
		code'
		prepended-statements
		}

While	**while** condition statement;	prepended-statements
		while (condition') {
		statement'
		prepended-statements
		}

Expression statement	expression;	expression'

For	**for** (initializers; condition; incrementors) block	initializers'
		prepended-statements
		while (condition') {
		block'
		incrementors'
		prepended-statements
		}

Goto	**goto** label;	goto label'

If	**if** (condition) statement	prepended-statements
		if (condition') {
		statement'
		}

If (full)	**if** (condition) statement **else** elseStatement	prepended-statements
		if (condition') {
		statement'
		} else {
		elseStatement'
		}

Labeled statement	label: statement;	label label'; statement'
Local declaration	type identifier;	var identifier' : type'
Local declaration and initialization	type identifier = expression;	var identifier' : type'; prepended-statements; identifier' := expression'
Return	**return** expression;	prepended-statements *(in methods:)* res = expression' goto end *(in functions:)* expression'
Unknown	(node generated during conversion phase)	(an error triggers)
End of method		*(in methods:)* label end

Table 6.1: How statements are translated

- **Return statement:** The expression is translated and the result stored in either the *res* output variable (if in method) or merely returned (if it's a function or a predicate). In methods, the context for the expression is *Purifiable*. In other cases, it's *PureOrFail*.
- **Expression statement:** The expression is translated. An error may trigger when the expression is not a valid Viper expression.
  - If the expression is a verification condition, the statement translates into nothing but will return the verification condition.
  - The expression is translated in a *PurityNotRequired* context and may result in a Viper statement.
- **End of method:** At the end of each method, the label **end** is added to Viper code. Because Viper has no *return* statements, this label is referred to by *goto* statements created when translating C# return statements. This only happens for methods that are not translated into functions or predicates because functions and predicates don't contain statements in Viper.

6.5.5 Prepending statements

In Viper, expressions cannot have side-effects. By side-effects, we mean the following actions:

- creating a new object;
- assigning a value to a variable or field;
- calling a method; and
- changing the state of the verifier by an inhale, exhale or similar statement

Viper guarantees this syntactically[10] as, unlike in C#, all of these actions (object creation, assignments, method call and inhale statement) are statements and not expressions. Statements are allowed to have side-effects.

However, C# permits, for example, calling methods from within expressions, and we would like to allow this if possible. To do so, when a C# *expression* that would be translated into a Viper *statement* is encountered, we move the translated expression to a code location *just before* where the original expression is and "prepend" it. For example, look at the following code.

```
if (methodname() == 2) {
    body();
}
```

In Viper, this would not be legal. Therefore, we translate the code into this:

```
var _tmp1 : Int;
_tmp1 := methodname();
if (_tmp1 == 2) {
    body();
}
```

[10]In the case of methods vs. functions, semantic analysis is also required to determine whether the subroutine call is pure.

As another example, suppose there there is a C# method with the signature `static int Return4()` and we used it in a statement `Contract.Assert(Return4() == 4)`, that statement will be translated thus:

```
var _tmp1 : Int
_tmp1 := Return4()
assert _tmp1 == 4
```

In this case, the user may have intended for the translation to be

```
assert Return4() == 4
```

To have that code, *Return4* would have to be declared *[Pure]* so that it translates into a Viper *function*. Since it's declared as a method, and method calls are not expressions in Viper, we must store the return value in a temporary variable.

We must not forget this to maintain soundness:

- In loops, these prepended statements must be added before each evaluation of the condition. That usually means before the loop and at the end of the loop body.
- If multiple subexpressions within an expression wish to prepend statements before the location, the subexpression that would be evaluated first in C# must also be evaluated first in Viper.

This still is not enough to maintain the same semantics as C#. In the case of binary boolean operations, C# has short-circuit evaluation. By prepending statements, we force all method calls to evaluate even if they weren't evaluated in C#. A similar problem occurs in the case of the conditional expression, where a branch may be evaluated even if the condition doesn't point to it. This may be the subject of future work.

6.5.6 Expressions

In this section, we'll show how C# expressions are translated.

An apostrophe after a word means "the translated version of this". The word might be an identifier, an expression or something else that can be translated.

Table 6.2 gives a summary.

For expressions that include operators, an example operator (usually plus) was used in the table but the translation works similarly with other operators.

In the purity column,

- *pure* means "this expression is translated to a Viper expression" and so it can occur in any purity context
- *stmt* means "this expression translates to a Viper statement", and so it can only occur inside *PurityNotRequired* context, or it must be prepended or trigger an error

Some notes are needed for specific rows of the table:

Expression name	**Example C# code**	**Equivalent Viper code**	**Purity**
Sequence concatenation	seq1 + seq2	seq1’ ++ seq2’	pure
Division	operand1 / operand2	operand1’ \ operand2’	pure
Another binary operator	operand1 + operand2	operand1’ + operand2’	pure
Unary operator	-operand	-operand’	pure
Compound assignment	res += operand	res’ := res’ + operand’	stmt
Conditional expression	cond ? ifyes : ifno	cond’ ? ifyes’ : ifno’	pure
Diagnostic expression	(node generated earlier)	(triggers an error)	pure
This	this	this	pure
Array index	expr[index]	*complex, see further down*	depends
Array length	expr.Length	|expr’.arrayContents|	pure
Array initializer	{expr, expr2, ...}	*complex, see further down*	stmt
Sequence element	seq[expression]	seq’[expression’]	pure
Increment / decrement	expression++	expression’ := expression’ + 1	stmt
Literal	literal (e.g. 2 or true)	literal (e.g. 2 or true)	pure
Object creation	new classname([arguments])	*complex, see further down*	stmt
Parenthesized expression	(expression)	(expression’)	pure
Assignment	lvalue = expression;	lvalue’ := expression’	stmt
Unknown expression	(node generated earlier)	(triggers an error)	pure
Identifier	name	*complex, see further down*	pure
Member access	expression.name	*complex, see further down*	pure
Invocation	expression([arguments])	*complex, see further down*	special
Soothsharp namespace	*various*	*complex, see further down*	depends

Table 6.2: How expressions are translated

- **Plus.** In Viper, sequence concatenation uses the operator ++ but in C#, we use + for this (because in C#, ++ is the increment operator). The translator uses semantic analysis to determine whether sequences or integers are added and emits a Viper operator accordingly.
- **Division.** In Viper, the integer division operator is “\” so that it can be differentiated from the fractional permission operator “/” that results in a Perm value.
- **Literal.** Some literals are not legal in Viper, such as floating-point numbers. These trigger an error.

Some of these expressions translate as statements but contain subexpressions. For all of these, those subexpressions are translated in a *Purifiable* context.

6.5.7 Arrays

To support arrays, we store their elements in Viper mathematical sequences. However, sequences are immutable, but arrays are not. Because of this, whenever an array is modified, we replace the old sequence by a new one. An array is thus represented by a reference object that has only a single field: a mathematical sequence of its type[11].

In Soothsharp, we support only single-dimensional integer arrays but support for other array types could be added in the future (we describe a possible solution at the end of this section).

If the use of an array is detected in the C# source code, the following Viper lines are added in the global addition phase:

```
field arrayContents : Seq[Int]
define arrayAccessPermission(array) acc(array.arrayContents)
define arrayWrite(array, index, value) { assert index >= 0; assert
    index < |array.arrayContents|; array.arrayContents :=
    array.arrayContents[..index] ++ Seq(value) ++
    array.arrayContents[(index+1)..]; }
function arrayRead(array : Ref, index : Int) : Int
     requires acc(array.arrayContents, wildcard)
     requires |array.arrayContents| > index
{
     array.arrayContents[index]
}
```

The following C# constructs are supported that are relevant to arrays:

- Array indexing
- Array initializer
- Array length

An array initializer is impure. If an array is initialized thus:

```
int[] variable = {2, 4}
```

It is translated into:

```
var _tmp1 : Ref
_tmp1 := new(arrayContents)
_tmp1.arrayContents := Seq(2, 4)
var variable : Ref
variable := _tmp1
```

An array length expression is pure and is translated into Viper's sequence length expression.

As for indexing, the translation depends on whether we write into the array or read from it. We write into an array if the indexing is on the left side of an assignment expression, and we read from it otherwise.

If we read, the indexing is translated into the *arrayRead* function. If we write, *the entire assignment expression* is translated into the *arrayWrite* macro. Here note that *arrayRead* is pure, but *arrayWrite* translates to statements.

[11]This is also the approach taken by Nagini, the Python frontend, for Python lists. Other ways of encoding arrays exist, such as [17].

arrayRead is a Viper function which means that it's pure and yet it can have preconditions. In this case, its use guarantees that we never access an array index that's out of bounds. Viper functions are pure and don't consume permissions. Therefore, after we read from an array, we still possess our original permissions to the array.

In order for a Viper verifier to allow access to the *arrayContents* field, we must have permissions to it. We will obviously have it if we create the array in the method but what if it's passed down as a parameter? For those situations, we introduce the contract method *AccArray* which grants access to that field (this contract can only be used on array-type variables). This might be improved in the future by having the *Acc* predicate automatically grant access to the *arrayContents* field if it's an array.

To extend arrays for non-integer elements, we would need to create separate Viper fields, functions and macros for each possible array type. Specifically, these would be *Int*, *Bool* and *Ref* (there is no reasonable use of a *Perm* array, and arrays of *Seq* and *Set* are also not very useful, especially if we can wrap them in objects). The translator could then select appropriate functions and macros based on the element type of the array.

6.5.8 The Soothsharp namespace

Soothsharp users will include the *Soothsharp.Contracts* library in their projects. The methods in this library have empty bodies (or merely return a literal value to satisfy the type checker) but translate directly into Viper constructs.

For example, consider the following method call:

```
Soothsharp.Contract.Inhale(
 Soothsharp.Contract.Acc(this.fieldName)
)
```

It would get translated into:

```
inhale(acc(this.fieldName))
```

Soothsharp namespace translation *takes precedence.* All identifier expressions, member access expressions and invocation expressions are first checked to see whether they don't have a special meaning given by this namespace, and only if not are they translated according to the general translation rules.

The tables in this section give translation for all "special" translations.

Table 6.3 shows translations for members of the main class.

The three "contract" methods (Requires, Ensures and Invariant) don't translate to Viper statements but are instead remembered by the enclosing block sharpnode and attached to the loop or subroutine that contains them.

Table 6.4 shows how extension methods residing in the *Soothsharp.Contracts* namespace are translated. *bool1* and *bool2* must be boolean expressions.

Table 6.5 shows how sequences are translated and Table 6.6 translates members of the *Permission* class.

Soothsharp.Contract.	**Viper code**	**Purity**
IntegerResult	res/**result**[1]	pure
Result<T>()	res/**result**	pure
Old(value)	**old**(value')	pure
Ensures(postcondition)	**ensures** postcondition'	contract
Requires(precondition)	**requires** precondition'	contract
Invariant(invariant)	**invariant** invariant'	contract
Assert(obligation)	**assert** obligation'	stmt
Assume(assumption)	**assume** assumption'	stmt
Fold(predicate)	**fold** predicate'	stmt
Unfold(predicate)	**unfold** predicate'	stmt
Folding(predicate, expr)	**folding (**predicate'**) in** expr'	pure
Unfolding(predicate, expr)	**unfolding (**predicate'**) in** expr'	pure
Inhale(expression)	**inhale** expression'	stmt
Exhale(expression)	**exhale** expression'	stmt
Acc(loc)	**acc(**loc'**)**	pure
Acc(loc, perm)	**acc(**loc', perm'**)**	pure
ForAll((i) => assertion)[2]	**forall** i' **:** type' **::** assertion'[3]	pure
Exists((i) => assertion)[2]	**exists** i' **:** type' **::** assertion'	pure

Table 6.3: How Soothsharp.Contract members are translated

[1] Within predicates and functions, *result* is used. Within methods, *res* is used.
[2] These are the only locations where Soothsharp permits a lambda expression.
[3] *type'* is the type of the identifier *i*, deduced by C# semantic analysis

C# code	**Viper code**	**Purity**
bool1.Implies(bool2)	bool1' ==> bool2'	pure
bool1.EquivalentTo(bool2)	bool1' <==> bool2'	pure

Table 6.4: How extension methods are translated

C# code	**Viper code**	**Purity**
new Seq<T>()	Seq[T']()	pure
new Seq(arguments)	Seq(arguments')	pure
seq.Length[1]	|seq'|	pure
seq.Contains(value)	value' **in** seq'	pure
seq + seq2	seq' ++ seq2'	pure
seq[index]	seq'[index']	pure
seq.Drop(value)	seq'[value'..]	pure
seq.Take(value)	seq'[..value']	pure
seq.TakeDrop(drop, take)	seq'[drop'..take']	pure

Table 6.5: Translation of sequences

[1] In this table, *seq* means "an expression with a Seq type".

C# code	**Viper code**	**Purity**
Permission.Write	write	pure
Permission.Wildcard	wildcard	pure
Permission.Half	1/2	pure
Permission.None	none	pure
Permission.FromLocation(loc)	**perm**(loc')	pure
Permission.Create(num, den)	num' / den'	pure

Table 6.6: Translation of the Permission class

6.5.9 Identifier expression and member access expression

Soothsharp. If one of the translations in the Soothsharp special translations table applies, that translation takes precedence.

Constants & Enums. If an identifier or a member access represents a constant (declared with the *const* keyword) or an enumeration member, it is translated to Viper as the value of the constant. In C#, enumerations are based on integer types, so the value will be an integer. Constants may either be integers, booleans or the null literal, which will be translated as the appropriate Viper type. Information from the Roslyn semantic model is used to determine the value of a constant.

Otherwise: A simple identifier expression is translated as the identifier's silvername; a member access expression *container.name* is translated as *container'.name'*. These are pure. If no container is given, but the identifier refers to an instance field, it is translated as *this.name'* instead.

6.5.10 Object creation and invocation

Object creation and Viper method invocation can occur in *PurityNotRequired* and *Purifiable* contexts. In a *PurityNotRequired* context, they are translated as statements. In a *Purifiable* context, that statement is prepended, the return value (i.e. the created object or the method's return value) is stored in a temporary local variable, and that variable is then used in the place of the original context.

Viper function and predicate calls can occur in any context.

Soothsharp. If one of the translations in the Soothsharp special translations table applies, that translation takes precedence.

Method invocation. Method invocation is translated in a straightforward way, regardless of whether the method was translated to a method, a function or a predicate. The translation schema depends on whether the method group is given as an identifier or a member access, and whether it's a static or instance method. The schemas are summarized in Table 6.7.

Kind	**Method group**	**C# code**	**Viper code**
static	identifier	name(arguments)	name'(arguments')
instance	identifier	name(arguments)	name'(this, arguments')
instance	member access	exp.name(arguments)	name'(exp', arguments')
static	member access	Type.name(arguments)	name'(arguments')

Table 6.7: Translating method calls

However, if the method translates into a Viper method, then it must be translated as a statement. Viper also requires that if the method has a return value, then that value *must* be assigned to a variable, even if it's not used. To handle this, we always use a temporary variable to store the result (unless the method has no return value, in which case a direct call is permitted and required).

Object creation. When a C# constructor is called, it is handled as though it were a call to a Viper method in the above section, except that it always returns a value (the created object) and that the name of the called method is determined based on the constructor.

If the constructor is the default constructor of a class, the name is *ClassName_init*. Otherwise, the name is *ClassName_ctor* (see Section 6.8). Information about whether a default or non-default constructor is called is taken from C# semantic analysis.

6.6 Global addition phase

For each type collected during the collection phase, the following steps are performed:

Adding fields. Each declared field is transformed into the silvernode "field [identifier] : [type]". The identifier includes the class name. [type] is the Viper type corresponding to the C# type.

If the type is a value type and cannot be correctly translated, an error would have triggered during the collection phase. If the type is a C# type whose methods or fields are not translated to Viper, no errors trigger at this point either, but they will trigger at the points where the user attempts to use these untranslated methods or fields.

Adding initializers. An initializer Viper method is created that creates a new object and returns it, and returns write permissions to all fields of the new object to the caller. This method is used whenever an object is created in C# using the default constructor, or at the beginning of user-defined constructors.

Adding array code. Additional code to support arrays is added here, as explained in Section 6.5.7.

6.7 Optimization phase

The following optimization steps are performed for the master silvernode tree:

- **Removing excess blocks.** If multiple blocks are nested in each other without using a loop or if statement, the redundant blocks are removed. Nested blocks are illegal in Viper; thus, this is not merely optimization, but a *requirement* for the translation to work correctly.

- **Removing empty silvernodes.** This eliminates redundant line breaks in the Viper code.

- **Removing final goto/label sequence.** Normally, the *return* statement is translated into Viper as `goto end` and the end of a method is translated as `label end`. However, for methods that contain only a single *return* statement, when it's last statement of the method, this "goto end; label end" sequence is redundant and clutters the Viper code. Because such methods are common, we eliminate this sequence where possible.

6.8 Name assignment phase

During previous phases, Viper identifiers may be needed, and in that case, a **base silvername** is created for each of them. The translation process is responsible for always assigning the same base silvername for syntax nodes that refer to the same identifier.

- Sometimes, a phase needs to create a temporary Viper identifier (for example, for a local variable that stores an unused return value of a method). The **base silvername** of these temporary identifiers is "`_tmp`".
- Otherwise, a phase might need to create a Viper identifier that corresponds to a C# item such as a class, a variable or a method. The base silvername of these identifiers is the partially qualified C# name of that symbol, **silverized** and perhaps with an additional **tag**.
  - A *partially qualified name* of a C# symbol is the fully qualified name of that symbol but without namespaces. We found that namespaces clutter the resulting Viper code and don't add any value in relevant cases.
  - *Silverization* is removing all characters not accepted in the Viper language. For example, the C# identifier "Hudeček" would be reduced to "Hudeek" and the C# identifier "π" would be reduced to an empty string. In addition, dots are replaced with underscores.
  - Sometimes, a tag will be added to differentiate among Viper identifiers that share the same C# symbol. This is the case of initializers (these get the suffix tag "`_init`") and constructors (these get the suffix tag "`_ctor`") because these special methods share the C# symbol of their enclosing class.

In the name assignment phase, we want to ensure that silvernames don't conflict with each other. To ensure this, we convert base silvernames into their final forms by appending a number (starting with `2`) to a silvername if that base silvername is already used an all forms with lesser numbers are already used as well. Thus, for example, the fifth temporary variable in the resulting Viper file would be named "`_tmp5`", and the second overload of the method *Foo.Example* would be named "`Foo_Example2`".

Keywords of the Viper language are also thought of as "being used" for the purposes of this phase. So, for example, even the first variable declared with the name "`axiom`" will have the silvername "`axiom2`".

6.8.1 Rationale

We designed this name translation process with these goals in mind:

- The silvername must be as similar as possible to the C# name.
- C# can use a wider set of characters in identifiers. These characters that are not legal in Viper will be omitted or otherwise translated away.
- There must not be ambiguity. For example, if there is an overloaded method, some characters will be appended to at least one of its overloads in order for the two methods to have a different silvername.
- Scopes exist in Viper, too; the translator should recognize this and not needlessly mangle names that would not be in conflict in Viper source code (not implemented, see *Future work*, Section 8.4).

6.9 Postprocessing phase

Postprocessing calculates the number of spaces that must be added in front of each line in the resulting Viper code so that the code is properly indented in more complex methods.

6.10 Verification

The master silvernode tree is serialized to text, written to a file and that file is sent to a backend verifier. The backend verifier's standard output is parsed, line-by-line, and either errors are found, or the file passes verification.

If any errors are found, they are associated with a line number and a column number in the Viper code where the error starts. These are converted to the C# location of the original error and shown to the user (see Section 7.4).

6.11 Producing errors

Any errors created by the translation process are fed back to the frontend. The *csverify* frontend will print them to the console and the Visual Studio plugin frontend will show them in the Error List and as squigglies.

Errors are almost always associated with a Roslyn syntax node, whether they come from translation or verification. This is converted to a line and column number in *csverify* and the Error List, but the entire erroneous syntax node is highlighted if using the plugin.

We assign error codes to various types of errors. These error codes are shown in the Error List window or are printed to standard output.

- Error codes from 100 to 199 represent an error during the translation.
- Error codes from 200 to 210 represent an error during verification.
- Error codes from 300 to 310 represent an internal error of Soothsharp.

6.12 Verification settings

A C# class or method may be marked with attributes (*[Verified]*, *[Unverified]* and *[SignatureOnly]*) or be unmarked. If it's marked by multiple of these attributes, a translation error is triggered.

Depending on user settings, unmarked nodes may be considered *[Verified]* or *[Unverified]*. The user may set some source code files to be marked for signature extraction only. If they do, then any *[Verified]* attributes on methods in those files are overridden and all *[Verified]* and unmarked methods are considered to be *[SignatureOnly]*.

- An *[Unverified]* node is ignored during both collection and main phase.
- A *[Verified]* node is subject to both collection and main phase.
- A *[SignatureOnly]* node is subject to the collection phase and to a modified variant of the main phase. This modified variant causes it to be translated as though it were abstract.

6.13 Type conversion

Types in C# are converted to Viper types according to Table 6.8.

C# type	**Viper type**
integer types	Int[1]
System.Boolean	Bool
Soothsharp.Permission	Perm/Rational
Soothsharp.Seq<T>	Seq[T]
an array type	Ref[2]
System.Void	*should never be translated*
any other value type	*translation error*
any other type	Ref

Table 6.8: How types are translated

[1] In Viper, an integer is unbounded, but in C#, it's a 32-bit integer. We assume that overflow will never occur.
[2] See Section 6.5.7 on arrays.

The type system and compiler of C#, along with the lack of conversion operators in Soothsharp, guarantee that type members are accessed properly. The way we build Viper forms of initializers and constructors also (redundantly) ensures that the program only has permission to access its actual fields and not fields of other types.

Value types that have no equivalent in Viper, such as floating-point numbers, are excluded from Soothsharp. Viper is not designed to deal with floating-point math.

All integer types are translated as `Int` — Viper doesn't deal with bounded integers.

Viper allows also for *domains* and the types *Set* and *Multiset* but these are never generated by the Soothsharp translator.

6.14 Design choices

It has been obvious from the very beginning that we wanted to do a source-level translation. Viper and C# are languages with very similar syntax. In addition, the frontends that came before us — Scala2Sil and Chalice2Silver — both use source-level translation, as does Nagini[12] (although we weren't aware of that at the time).

The other option would be a bytecode-level translation. However, there we would run into problems: Almost nobody explored the translation of bytecode into Boogie-like languages and there would be little existing experience to draw on. The code we would produce would also be necessarily larger and slower, and we would need to implement a stack for local variables, among other issues.

Using Roslyn and translating the abstract syntax tree is preferable to translating the source files as text strings because the parser work is already done by the compiler. We were learning Roslyn API as we went along so there are a few inelegant constructions in our code, but using Roslyn was the right idea.

Most C# and Viper constructs have a one-to-one correspondence. For example, most arithmetic expressions don't need to be translated at all. The while loop and the if condition are quite similar as well, as is the block-based structure of the code. For these reasons, we chose to translate each syntax node individually, as opposed to performing more complex translations.

However, some nodes cannot be so easily translated. Let's take contracts (preconditions and postconditions), for example. A simple node translation of a method declaration might translate the C# code block into a Viper code block, translate the method declaration header and put that together. However, preconditions and postconditions are included in the C# code block as statements. They don't result in actual Viper code themselves: at the very least, they can't be put in the same Viper code block as all the other statements in the method. The result of their translation has to be given to the method declaration which puts them between the Viper header and the Viper method body.

That is not a very pretty solution, but it's an acceptable one. We were unable to find any other feasible solution.

Then, there was a decision to be made on what to translate C# code into. One option was to have C# nodes generate Viper code directly, as text. That would greatly simplify the project as we wouldn't need the Viper tree nodes at all. Instead we chose to generate a Viper syntax tree first, and then serialize that tree into Viper code.

The reason for that is to maintain links between C# and Viper code. We must unfortunately communicate with the verifier by sending it a text file and reading its standard output. What it gives us is a list of verification errors associated with a line and column of the Viper code given as input. But when the user is writing C# code in an IDE, or even in a text file, they don't want to see what

[12]Nagini is a Viper frontend for Python, see Section 9.3.

generated line caused the verification to fail: they want to see what part of their own code triggered the error.

By maintaining a link between C# syntax nodes and the Viper syntax node they translated into, we can then go back from the line and code of the Viper code to the Viper syntax node and finally to the C# syntax node, and if necessary, from there, to C# line and column.

That said, we believe our Viper syntax tree is overly complex: we describe some of this in Section 8.3.

7. Implementation

Most code in Soothsharp project is called as part of the translation process, which follows the specification above.

The **conversion phase** happens entirely within constructors and cannot fail. The translation process begins by calling the constructor of a *CompilationUnit-Sharpnode* and from there, construction proceeds until the entire *CompilationUnitSharpnode* is constructed. The **main phase** happens via calling the *Translate* method on nodes in the constructed tree.

This chapter describes parts of the code where the meaning is less obvious.

7.1 Code overview

The source code contains the following folders and projects:

- **Soothsharp.Contracts**
  - library that users of Soothsharp must reference in their projects to be able to write contract code
- **Soothsharp.Translation.Tests**
  - unit and system tests for Soothsharp, see Section 7.7
- **Soothsharp.Translation**
  - the bulk of the code, responsible for translating C# code to Viper and for reporting verification errors back
  - **Backends**
    - contains classes that launch backend verifiers (Carbon or Silicon), pass them Viper code files and recover their output
  - **Diagnostics**
    - contains a list of all possible errors that may occur when using the Soothsharp translator (in Roslyn, a diagnostic is an error or a warning)
  - **Exceptions**
    - custom exceptions thrown during translation
  - **Translators**
    - some parts of the translation process are complex enough that we added special classes for them, we call these classes "translators", see Section 7.1.3
  - **Trees**
    - **CSharp**
      - contains nodes that make up the converted C# tree; all files start with `CompilationUnitSharpnode` inside the subfolder `Highlevel`

        * **Silver**
            * contains nodes that make up the translated Viper tree

- **Visual Studio Plugin**
    - **Soothsharp.Plugin**
        * an intermediary between the Visual Studio plugin and the `Soothsharp.Translation` project
    - **Soothsharp.Plugin.Vsix**
        * a VSIX package that can be installed as an extension into Visual Studio
- **Soothsharp.Rewriter**
    - a console application that removes all contracts from a code file; using this application is one way of facing the issue posed by the recursive predicates problem, see Section 7.8
- **Viper**
    - this folder contains Viper binaries and scripts that are used by Soothsharp as backend verifiers
- **csverify**
    - command-line utility for running Soothsharp
- **csverify GUI**
    - a Win32 GUI application that sets command-line arguments and runs *csverify.exe*
- **Examples**
    - some example code files that use Soothsharp for verification
- `soothsharp.sln` (the main solution file)
- `README.md`
- `LICENSE.txt`

7.1.1 Csverify.exe

Csverify.exe may be run directly from the command-line or launched from the GUI application. Its role is to accept options (described in the user documentation, see B.3), process them, run the translation process and report its results to standard output.

7.1.2 Soothsharp.Contracts

This library is meant to be used by the end user and it exposes Viper features to C#.

Viper methods, functions and predicates are all created from C# methods, so attributes are used to differentiate between them. The *Contract* static class is a container for most standalone keywords of Viper. The *Permission* and *Seq* classes expose the Viper types *Perm* and *Seq*.

Other attribute classes are used by *csverify.exe* to determine which parts of the code should be verified.

Classes, methods and properties from this library must be translated in a special way. The translator recognizes them by their name. For example, any method with the fully qualified name *Soothsharp.Contracts.Contract.Requires* is translated as a *requires* clause. This makes Soothsharp more resilient to errors that might occur if, for example, the user has a different build of the *Soothsharp.Contracts* library.

7.1.3 Translators

Some parts of the translation are factored out to separate classes which we call "translators".

Type translator. This class converts C# types to Viper types, for example, it converts `System.Int32` to `Int`. Types that cannot be translated cause a translation error.

Subroutine builder. This class takes a C# method declaration or a constructor and creates from it a Viper method, function or predicate, abstract or non-abstract.

Seq translator. This class translates *Soothsharp.Contracts.Seq* into Viper *Seq*.

Constants translator. This class translates declared constants (such as "`const int a = 2`") by inlining them in Viper code.

Arrays translator. This class adds Viper code related to arrays and translates array reads and array writes.

Contracts translator. This class translates the various members of *Soothsharp.Contracts.Contract* into keywords of the Viper language.

7.2 Setup

To build Soothsharp from source, the user should follow this process:

1. Clone locally the repository `https://github.com/Soothsilver/soothsharp`[13]

2. Follow the installation guide in Section B.2 to add required Viper components to the system.

[13]You may also copy the source code from the attached DVD.

3. Rebuild the solution. *If the fails, right-click the solution in the Solution Explorer and click "Restore NuGet packages", then try to rebuild the solution again.*

Then the user may run the "csverify GUI" project and find examples in the "Examples" project.

7.3 Using backend verifiers

About Nailgun. There are two ways to run the backend verifies on Windows. The most straightforward way is to use the batch files provided by the Viper project. However, both verifiers are written in Scala and run on the Java virtual machine which requires the JVM to be initialized each time, which takes significant time.

To allow for shorter verification time, the official Viper IDE plugin for Visual Studio Code makes use of Nailgun [18], "a client, protocol, and server for running Java programs from the command line without incurring the JVM startup overhead". The Nailgun server is launched, once, with class files of both verifiers in its classpath. From then on, the nailgun client ("ng.exe") can be used to execute entry points in those class files. This cuts back on the time required for verification.

Originally, we used the batch files for verification and the code that facilitates this is still present in the classes *CarbonBackend* and *SiliconBackend*, but eventually we migrated to using nailgun because of its speed-up which is relevant especially for automated tests. The interaction with nailgun is handled by *CarbonNailgunBackend* and *SiliconNailgunBackend*.

Starting the Nailgun server. Whenever Soothsharp attempts verification, it first launches the Nailgun server using a batch file (`startviperserver.bat`) provided by the Viper team. If the server is already running, the batch file does nothing. This file is only launched once per lifetime of the translator, which is useful for running batches of automated tests.

Running the verifiers. To verify Viper code, Soothsharp saves that code to a temporary text file and passes it as an argument to a verifier class started by means of the Nailgun client. The verifier, via Nailgun, sends its standard output back to Soothsharp, which can then recreate error messages with information about C# code, as seen in the following section.

7.4 Converting Viper errors to C# errors

During the conversion phase, Roslyn syntax nodes are associated with sharpnodes, and during the main phase, the created silvernodes are associated with the sharpnodes that created them. A silvernode is either atomic (such as *EmptySilvernode* or *IdentifierSilvernode*) or complex, which means that it consists of one or more child silvernodes (for example, *SubroutineSilvernode*).

An atomic silvernode always implements *ToString* which returns its Viper representation. A complex silvernode's *ToString* method concatenates the Viper representations of all children of the silvernode. Thus, a tree of silvernodes may be converted to Viper code by calling *ToString* on its root node.

When that is done, the verifier examines the Viper code and returns a list of error messages, where each message is associated with a line and column of the Viper code.

To determine which line and column of C# code each message corresponds to, the children of each complex silvernode are expanded, until the line and column of the Viper error is reached, and the earliest atomic silvernode that contains that line and column is deemed to have caused it. Then, either that silvernode itself or one of its parents is linked to a sharpnode, and each sharpnode is linked to a Roslyn node and from there we get the line and column of the C# code that caused the error.

7.5 Roslyn

Microsoft.CodeAnalysis (better known under its codename *Roslyn*) is a .NET class library that can be used to analyze C# code [9]. The library exposes almost everything that's done by the C# compiler: syntax analysis, semantic analysis and even emitting IL code.

In this project, we make use of its syntax and semantic analysis and, in the rewriter program, of the ability to modify the abstract syntax tree.

A major limitation of Roslyn is that it doesn't permit interacting with the C# compiler itself. It is not possible modify the abstract syntax tree between the time the user starts compilation and the time IL code is emitted.

Documentation for Roslyn is available on the Internet [10]. In the rewriter project, we use its Rewriter class to modify the abstract syntax tree.

In the translator proper, we use Roslyn more extensively. We use it to perform syntax analysis and obtain a syntax tree from C# code given as a string (using *CSharpSyntaxTree*) and perform semantic analysis on it. The results of the semantic analysis are stored within *SemanticModel* which is accessible from the context in the main phase of the translation.

Roslyn nodes contain other Roslyn nodes as members, and we convert all of these to instances of our classes in the conversion phase, because we don't need all the members of those nodes, and conversely, we need properties and methods not present in the Roslyn classes.

7.6 Visual Studio plugin

We have created a small plugin for Visual Studio 2015 that allows the user to translate and verify their C# code files from within the Visual Studio IDE. User documentation for this plugin is in the appendix.

The plugin uses Roslyn extension points to run the translation process whenever Visual Studio determines that the syntax tree has changed.

7.7 Automated tests

The unit test project, *Soothsharp.Translation.Tests*, contains four kinds of automated tests:

- traditional unit tests: these are used on the algorithms library
- syntax tests (*SyntaxTest*): these run the translation of a C# code file into Viper code using the translator; for a test to pass, the translation must succeed
- compare tests (*CompareTest*): these also run the translation only, and a compare test passes if the Viper code output of the translation is the same as the expected output
- systemwide tests (*SystemwideTest*): these run both translation and verification (using Carbon); the test passes if the translation and verification errors that actually triggered match the expected ones

Full translation tests are the best way of ensuring that the translation works as expected, so we focused on those tests.

Compare tests were a failure: in the end, we only kept a single compare test. The problem with these tests is that a minor change in the translation program might invalidate many compare tests and force us to generate the expected output again.

For many test files, we only perform the syntax/translation test, and not the full systemwide test, because the errors we test for in those files would reveal themselves already during translation, and doing translation only is faster than running the verifier as well.

We have about 40 tests in total.

7.8 Recursive predicates problem

We encountered a significant problem with recursive predicates in that a program that uses recursive predicates, if it's run as an executable application, will crash because of a *StackOverflowException* or another exception, such as *NullReferenceException*.

Recursive predicates (see Section 4.2.6) are a feature of Viper and permit reasoning about the heap in situations where an object refers to other objects of the same kind, such as in a linked list.

Here's why this is a problem. Contracts in Soothsharp are written as expression statements with a method call as the expression. Although the method bodies of *Requires*, *Ensures* etc. are empty, and so have no effect on execution, their arguments are still evaluated. That leads to two issues:

- First, evaluating an argument might throw an exception. Suppose, for example, that we need a precondition of "the first argument can't be null and its field is nonzero". We might encode that, for clarity, as two preconditions: "`Requires(arg != null);`" and "`Requires(arg.field != 0);`". This will pass both compilation and verification, but, when it's called with a null argument, a *NullReferenceException* will be thrown. In this particular case, that might be okay, since we're not supposed to call a method with a failed precondition, but in other cases, that might be a problem.

- Second, more importantly, to encode Viper predicates, Soothsharp uses C# methods. But then, if a contract contains a predicate and that predicate contains itself in its body (i.e. it is recursive), during execution, an infinite recursion will occur. Verification will still succeed, because Silicon/Carbon do not execute code — they perform analysis. The launched executable file, however, would fail due to this infinite recursion.

The first problem can be partially solved by a couple of tricks. For example, we may have Soothsharp methods and properties return non-null dummy objects that can perform any operation. This cannot be used every time, though, and as for the second problem, we don't see a way (without significant changes to Soothsharp) to have recursive predicates in C# code and yet have the code be correctly executable as a functional Win32 program. This problem doesn't affect the verification itself.

Other tools, such as the Scala frontend or Code Contracts, use rewriting to remove contracts from the code before compilation. Specifically, the Scala frontend project created a plugin for the Scala compiler that modifies the abstract syntax tree of the Scala program by removing the contracts before it's passed to the next phase of compilation. Code Contracts use a binary rewriter at the IL level to move contracts to appropriate points. None of these options can be done with the technology we have (mostly because Visual Studio and *csc.exe* do not allow interfering with the compilation process).

We have, unfortunately, discovered this problem late during work on this thesis and only had time to implement two less satisfactory solutions. We present all the possible solutions to this problem that we know of here:

1. **Add a value that's *false* in C# and yet *true* in Viper.** Add a new constant that can be used in C#, such as "`Contract.Truth`". In C#, this constant would evaluate to *false* but in Viper, it would be translated as *true*. C# uses short-circuit evaluation. If the user added this constant as the first operand of a conjunction in a contract, the rest of the contract would not evaluate and wouldn't throw an exception nor would it cause infinite recursion. However, the user would need to write longer code. For example, instead of "`Requires(RecursivePredicate(this));`", the line "`Requires(Truth && RecursivePredicate(this));`" would be needed.

2. **Do not support recursive predicates.** This solves the second (more serious) problem. Recursive predicates aren't necessary for many programs. For example, *InsertSort* can be verified without using them. Still, these predicates are useful very often and they are needed for linked lists or graphs. In addition, Soothsharp already supports the verification of recursive predicates and verification is the primary contribution of this project, so it would be a waste to not include them.

3. **Preprocessor directives.** Using the `#if` and `#endif` directives, we could separate contracts from the executable code. Contracts would be

run for Soothsharp verification, and the executable code for the compiler. However, these directives would need to be written by the users themselves. They couldn't be in the Soothsharp contracts library, and especially when using the expressions *Folding* and *Unfolding*, which are important for recursive predicates, code could be littered with `#if` blocks and become unreadable.

4. **Create a rewriter.** Before generating the final executable file, the user could run a rewriter program on their code that would remove contracts from the code. Only the code that comes out of the rewriter would be compiled. As far as we know, it is not possible to integrate this into Visual Studio. We could have this program replace the *csc.exe* compiler, but that is problematic (*csc.exe* is complex and has many features and options).

We have decided to implement solutions 1 and 4. The user may use the *Contract.Truth* static property in contracts to ensure that they're not evaluated at runtime. Most contracts are simple and don't need it, but for contracts with recursive predicates, it is required.

Alternatively, the user may run the *Soothsharp.Rewriter* program on their code which generates identical code except that all contracts are removed and the *Folding* and *Unfolding* expression have their permission part removed.

We recognize that this is an obstacle to the use of Soothsharp as an end-user tool. For larger or more complex programs, this extra work on the part of the user may be excessive. We outline a possible path forward to remove this issue in the *Future work* section (8.4).

7.9 Algorithms library

A library of basic algorithms that have been verified using Soothsharp can be found in the project *Examples*. They can be included in a Soothsharp verification of other programs, either directly or as assumed files. More information can be found in the User documentation (Section B.7).

8. Discussion

In this chapter, we discuss what features proved to be the most difficult or surprisingly time-consuming to implement. We also give limitations of the current translator, share insights from the development process and offer ideas on how our work can be improved in the future.

8.1 Notable difficulties

The work on Soothsharp encompassed a number of areas that proved more difficult to understand or implement than we anticipated. We don't include the planning or architecture work in this section.

Understanding the Viper language. At the time when development of Soothsharp started, there was no specification of the language available. A research paper outlining the principles existed, and there was an (outdated) grammar description, but nowhere near enough material to create a translation program from.

We spent significant time compiling a specification of the language in order to make it possible to create a sound translator. To do this, we analyzed the Scala source code of the parser, experimented with the verifiers on simple files and also exchanged private communication with Dr. Malte Schwerhoff, a researcher at ETHZ, one of the authors of Viper.

During this work, we have reported several bugs and inconsistencies in the parser, the grammar and the verifiers to developers of the Viper project.

While there is still no specification of the language as such, the semantics of a significant part of Viper have since been formalized in [3].

Scale of the C# language. Traditionally, one wants to verify at the lowest possible language level. For example, the Code Contracts static verifier Clousot runs its verification on the IL code emitted by the C# compiler. This makes it possible to handle even the very large number of features supported by the C# language with less effort.

However, Viper is a language that presents many similarities to C# that can be exploited and many features are easier to verify in C#-style code than in assembly-style (for example, loops rather than jumps). This also has the side-effect of making the resulting Viper code more readable.

This means, however, that we have to translate language with a specification that takes 500 pages. Even with the reduced scope (see Chapter 5), the sheer scale of the project caused it to take a lot of time.

Assignments. Our translation process wants to translate each node on its own, without knowing anything about its parent. That works most of the time but for assignments (and some other cases), it doesn't. If an expression is on the left-hand side of an assignment, it cannot be translated the same way as if it were on the right-hand side. This is mostly because Viper assignments are statements where the left-hand side is fixed and doesn't allow arbitrary expressions. C# assignments, by comparison, are just expressions.

Prepending method calls. In Viper, method calls are statements. In C#, they are expressions. The code `int a = MethodCall(2) +`

`MethodCall(4)` is legal in C#, but not in Viper, where it would have to be translated as:

```
tmp1 := MethodCall(2)
tmp2 := MethodCall(4)
a := tmp1 + tmp2
```

This is common to do in compilers, but it's a little more complicated in Soothsharp, because the method calls can't just be prepended before the expression, but before the *outer statement* that contains the expression. For example, if the method call is part of an *if* condition, the method call must go in front of the *if* statement.

8.2 Aesthetics

A notable limitation of the Soothsharp project is the way contracts must be used by the end user. In languages designed primarily for verification (e.g. Viper or Spec#), keywords such as *requires* and *ensures* are first-class keywords of the language. Frontends for Viper, however, add contracts using the tools that already exist in the language, usually method calls. This is the case not only for Soothsharp, but also for Scala2Sil, Nagini and Code Contracts[14].

The lack of these keywords and other syntax elements makes it less convenient for the programmer to use Soothsharp, and causes the code to become longer. Two cases where this is the most obvious are *Implies* and *Contract.Result.* Where in Viper one can write an implication as `expression1 ==> expression2`, in C#, `(expression1).Implies(expression2)` is needed. Note the parentheses that are almost always required.

Because C# doesn't allow generic methods to infer their type arguments from context, the type argument of *Contract.Result()* must be always given explicitly. Longer contract lines are thus difficult to read.

One additional important aesthetics problem deals with throwing exceptions in contracts and with recursive predicates. Normally, contracts are evaluated by the runtime during execution and this may cause exceptions or infinite recursion to occur. To prevent this, either the rewriter program needs to be used (which cannot be done directly from Visual Studio) or the *Contract.Truth* member has to be put manually in all problematic contracts. We have described this in detail in Section 7.8 .

8.3 Code limitations

If we were to create the translation program from scratch now, there are some things that would be better done differently.

A better design of the class trees. Both class trees, for C# and for Viper, are somewhat complex, having intermediate base classes for statements and expressions. Especially for Viper, they are a little haphazardly used and not especially necessary and they could be removed to make the code cleaner and simpler.

[14]Another option is to add contracts as comments, but that is even less appealing — any IDE support is lost that way.

Better passing of information though the tree. When converting the C# tree to the Viper tree, an instance of a context class is passed to each node and an instance of *TranslationResult* is returned. Especially the *TranslationResult* class contains a number of members that are not useful for all nodes. Because of tree structure of the nodes, a *TranslationResult*'s contents have to be copied (or appended) in each node to the next *TranslationResult*, until a final *TranslationResult* is output as the result of translation. However, there is a potential for bugs in this process if some members of the class are forgotten. Here, a better architecture might help.

Method prepending. C# allows method calls inside expressions, Viper doesn't. To make this work, we invented a method of extracting method calls from expression and prepending them in front of the outer statement. That is okay, but our implementation is not very clear and prone to mistakes, because it requires prepending code at multiple locations in Soothsharp source code.

8.4 Future work

The Soothsharp project might be improved in the following ways:

- **Multithreading.** While Viper contracts and permissions have some use even in single-threaded programs, they offer the most benefit when used to guarantee data race freedom in parallel programs. Soothsharp can be extended to allow the user to run methods in parallel.
- **First-class contracts.** Instead of encoding contracts as a method calls, create a superset of the C# language that would add keywords such as *requires* and *ensures.* This is possible by creating a customized C# compiler but distributing it is not well-supported. This is a difficult extension but it would eliminate two of the most important issues in Soothsharp: its aesthetics problem and the recursive predicates problem.
- **Better name translation.** Viper supports scopes, albeit in a limited way. If a variable named `a` is declared in two different methods, currently Soothsharp will translate the first as `a` and the second as `a2`, but there is no reason why both couldn't be named `a`.
- **More object-oriented features.** There is a lot of possible improvements regarding object-oriented programming support. In the future, Soothsharp might want to support inheritance, interfaces and the *is* operator.
- **More Viper features.** Viper contains some useful features that are not exposed via Soothsharp. Notably, these are the mathematical constructs *Set* and *Multiset*, and magic wands. Magic wands are a little tricky, but *Set* and *Multiset* could be implemented analogously to *Seq.*
- **Static fields.** Soothsharp doesn't support static fields yet. There should be a way to add support for these into the translator, perhaps by creating a global Viper object to hold static fields and passing this object to all methods.

- **Class libraries.** .NET offers a large number of classes and methods that can be used by C# programs, but these are not annotated by contracts and thus cannot be used in Soothsharp-verified programs. Features could be added that would allow us to annotate these classes (to which we don't have source code access).
- **Automatic contract inference.** Currently, we require the user to specify a lot of contracts by hand, as Viper does. However, it might be useful to infer some preconditions or postconditions automatically, as it's done by Code Contracts.
- **Exceptions.** C# makes use of exceptions which break traditional control flow, and — more importantly — can cause a method to not return a value of its stated return type. This presents some difficulties but in the Nagini frontend, these were overcome.
- **Integration with another IDE.** Soothsharp, as it stands now, is not well suited for large industrial projects consisting of many files. Writing code to be checked by Soothsharp might possibly be better done in an editor such as Visual Studio Code than Visual Studio proper.
- **More proper integration.** Sometimes Soothsharp forces the user to use less complex expressions. For example, it is not possible to use the result of a method call in an l-value (the method call must be before the assignment expression). This can be done automatically by Soothsharp in the future.

A full frontend that would translate as much C# as possible into Viper is probably not a useful effort but some of the options given above would likely be beneficial, if only to further validate the effort of the Viper project.

9. Related Work

In this chapter, we would like to present a number of projects in some way related to the kind of verification we attempt, either for C# or other languages, within the Viper project or outside it.

There is a large number of such tools available[15] and we cannot elaborate on all of them here, but those that are the most related will be discussed.

9.1 Code Contracts

Code Contracts [19] is a Microsoft tool for expressing preconditions, postconditions and invariants in C# code.

Code Contracts are used much like our own *Soothsharp.Contracts* library. In fact, the naming of classes and signatures of methods in that library (*Contract, ForAll, Result*) are directly inspired by Code Contracts and made to resemble them as much as possible.

There are two ways Code Contracts can be used — with a static checker or a binary rewriter. The static checker [20] is able to output Visual Studio warnings for some violations of contracts, but not for all — for example, it is unable to prove most assertions with a quantifier. The binary rewriter encodes preconditions as guards that throw an exception if not met. In a similar way, postconditions will then throw an exception if not satisfied.

Code Contracts operate on the bytecode level [21].

Our project (and Viper), as opposed to Code Contracts, is able to statically prove more complex assertions and can reason about permissions or ownership of objects on the heap. Both projects share the same flaw of clumsy syntax in C# (see Section 8.2).

9.2 Spec#

Spec# [22] is an abandoned collaboration project between ETH Zurich and Microsoft Research. It is a superset of C# 2.0, its syntax a blend of C# and the language used by intermediate verification languages such as Viper.

Lessons learned from the development of Spec# were applied when Code Contracts were created, but the project was abandoned around 2008 [23].

Spec# verifies code similarly to how we do it, by translating the C# (Spec#) code into Boogie and then running an SMT solver. Spec# has a notable advantage of having elegant syntax as opposed to our tool or Code Contracts. It uses ownership to reason about heap locations which is a less powerful technique than permission logic, although similar. There were attempts for Spec# to also check whether a method terminates but these were only really developed for Dafny, a sort-of successor to Spec# [24, p. 5].

[15]For example, AutoProof, Dafny, Frama-C, KeY2, Spec#, VCC, VERL (no permissions), Bedrock, Chalice, FCSL/Coq, Grasshopper, Infer, jStar, SmallFloot, VeriFast, Ynot [16, pp. 17–18].

9.3 Nagini

Nagini[16] is a frontend for the Viper project that translates Python source code to the Viper language and then verifies it. Nagini is currently developed by Marco Eilers at the university of ETH Zurich. Its capabilities are similar to what Soothsharp offers for C#, but more advanced in some areas.

In particular, Nagini handles class inheritance with some success, and supports try/catch blocks and exceptions.

We would like to thank Marco Eilers for his advice and for sharing his experience on programming a Viper frontend.

9.4 Other frontends

Other frontends for Viper already exist.

There is a frontend for Chalice [26], which is a verification language itself. That was the example frontend for the Viper project. Chalice and Viper are similar in many respects.

There is a frontend for Scala [15] from which we took some inspiration. Like us, the Scala frontend uses compiler libraries to get access to the syntax tree of the original language and then translates them mostly at the abstract syntax tree level. Like us, it supports only a subset of the Scala language. Because the frontend was written in Scala, just like the Viper framework itself, it could interface with the verifier directly. Soothsharp, on the other hand, has to rely on creating a text file and communicating via a command-line interface.

Finally, frontends for Java and OpenCL have been developed at the University of Twente. These are said to be the most comprehensive frontends, but we were unable to gain access to them.

[16]No public document or website is available for citation, but the project is part of the VerifySCION initiative [25].

10. Conclusion

We have developed Soothsharp, a transcompiler that converts C# source code files into code of the Viper Intermediate Language. It supports a reasonable subset of the C# language, works from within the Visual Studio IDE and makes it more convenient to write verifiable programs.

Soothsharp proves that the Viper framework is a solid base upon which one can build frontends that help writing verifiable programs, even for languages quite different from the Viper language itself or the language it's programmed in, Scala.

We have encountered no fundamental issues that would prevent advanced verification of C# programs using a future version of Soothsharp.

However, there are obstacles that prevent our work from being used as-is by end users, because the program misses some features needed for more complex projects, such as exceptions or inheritance. These features could be added in the future.

Bibliography

[1] Viper website.
`http://www.pm.inf.ethz.ch/research/viper.html`.

[2] Peter Müller, Malte H. Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[3] Cyrill Martin Gössi. A formal semantics for viper. 2016.

[4] Boogie website. `https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/`.

[5] Z3 homepage. `https://github.com/Z3Prover/z3`.

[6] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Werner Dietl and Peter Müller. *Object Ownership in Program Verification*, pages 289–318. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[8] Colin S. Gordon, Matthew Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. Technical report, October 2012.

[9] Roslyn website. `https://github.com/dotnet/roslyn`.

[10] Roslyn documentation. `https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview`.

[11] Binary search (Seq) example.
`http://viper.ethz.ch/examples/binary-search-seq.html`.

[12] Malte Hermann Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution.* PhD thesis, 2016.

[13] Sorted list example. `http://viper.ethz.ch/examples/sorted-list-immutable-sequence.html`.

[14] Linked list example. `http://viper.ethz.ch/examples/linked-list-predicates.html`.

[15] Bernhard F Brodowsky. Translating Scala to SIL. Master's thesis, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2013.

[16] Malte Hermann Schwerhoff. Talk on Viper, 2016.
`http://malte.schwerhoff.de/talks/prague16_slides.pptx`.

[17] Viper alternative array encoding.
http://viper.ethz.ch/examples/max-array-standard.html.

[18] Nailgun website. http://www.martiansoftware.com/nailgun/.

[19] Code Contracts website. https://www.microsoft.com/en-us/research/project/code-contracts/.

[20] Manuel Fahndrich and Francesco Logozzo. Static Contract Checking with Abstract Interpretation. Springer Verlag, October 2010.

[21] Rustan Leino and Mike Barnett. To Goto Where No Statement Has Gone Before. In *VSTTE 2010*, August 2010.

[22] Spec# website. https://www.microsoft.com/en-us/research/project/spec/.

[23] Spec# vs Dafny, online discussion.
http://specsharp.codeplex.com/discussions/569610.

[24] Rustan Leino and Peter Müller. Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs. In *Advanced Lectures on Software Engineering*, volume 6029 of the series Lecture Notes in Computer Science, pages 91–139. Springer Berlin Heidelberg, March 2010.

[25] VerifySCION.
http://www.pm.inf.ethz.ch/research/verifyscion.html.

[26] Chalice website.
http://www.pm.inf.ethz.ch/research/chalice.htm.

A. Contents of the DVD

- **source**
  - source code for the Soothsharp solution as detailed in Section 7.1; this is also available at `https://github.com/Soothsilver/soothsharp`
- **binaries**
  - **contracts**
    - `Soothsharp.Contracts.dll` (the class library that contains contracts)
  - **tools**
    - `csverify.exe` (the translator program)
    - `rewriter.exe` (the rewriter program)
    - `Soothsharp Graphical Verifier.exe` (a GUI for the translator)
    - `Soothsharp.Plugin.vsix` (the Visual Studio plugin)
    - various dll files requires by the above programs
  - **viper**
    - this folder contains tools of the Viper framework; these files must be in your `PATH` environment variable
  - **boogie**
    - this folder contains compiled Boogie binary files
- **class-documentation**
  - generated class documentation for the project
- `thesis.pdf` (this document)

B. User documentation

B.1 Overview

Soothsharp is a tool for verification of C# code with respect to contracts. It makes use of the Viper framework as its backend. You will need a certain familiarity with software verification and Viper-like methodologies to use Soothsharp effectively.

Soothsharp consists of the following projects. They are described in more detail further below.

- **csverify.exe:** This command-line tool can translate C# files to Viper code, or verify them immediately.
- **csverify GUI:** This program is the graphical user interface to *csverify.exe*.
- **Visual Studio Plugin:** If you install this plugin in Visual Studio, translation and verification errors will be shown in the Error List within the IDE.
- **Soothsharp.Contracts:** This C# class library contains contracts that you will need to use in your verifiable C# code. Add this library as a reference to your projects.
- **Examples:** This folder contains example code files that demonstrate how to use Soothsharp.
- **rewrite.exe:** This program removes contracts from the C# code it receives and writes the remaining code to standard output.

B.2 Installation

All binary tools, except for the plugin, may be used out of the box, by running the executable.

To install the plugin in Visual Studio, double-click the *vsix* file. You must have Visual Studio 2015 installed.

To make use of verification, you must first ensure that Soothsharp can locate the backend files and Nailgun. To do this, add the folder *Viper*[17] to your PATH environment variable.

In addition, some additional programs that are required by Viper must be installed. For both verifiers, you will need Java 8 and the Z3 solver. For Carbon, you will also need Boogie.

You may download and install Java 8 from its official website[18].

You may download the Z3 prover from its GitHub page[19]. The recommended version of Z3 for the version of Viper tools supplied with this thesis is 4.4.0.

[17]The folder contains the files *ng.exe*, *startviperserver.bat*, *silicon.jar*, *carbon.jar* and *nailgun-server-0.9.1.jar*.

[18]`https://www.java.com/en/download/`

[19]`https://github.com/Z3Prover/z3/releases`

When you do, you must create an environment variable named *%Z3_EXE%*. That variable should hold the path to the *z3.exe* executable.

To run the Carbon verifier, you will need Boogie. You may download Boogie's source code from its GitHub page[20] and build it. Alternatively, you may use the binary files attached to this thesis. Then, you must create an environment variable named *%BOOGIE_EXE%*. That variable should hold the path to the *boogie.exe* executable.

These variables are how the Viper verifiers locate the tools they depend on. Alternatively, you play all Z3 and Boogie files in the Viper tools folder which should have the same effect.

Finally, the Viper tools themselves must be located under a path that is pure ASCII and doesn't contain non-ASCII characters or spaces.

It is possible to replace the attached Viper tools with newer versions, but that process is more difficult.[21]

B.3 csverify.exe

The main executable of the project, *csverify.exe*, takes C# code files as input and outputs Viper code or error messages.

The tool has many command-line options:

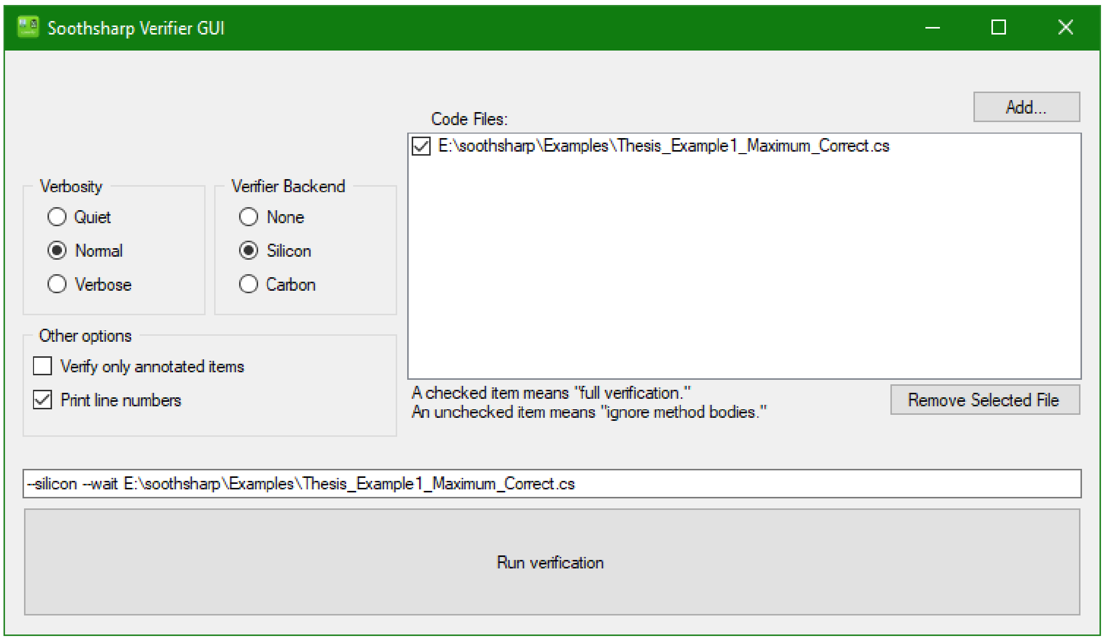
```
Usage: csverify.exe [OPTIONS] file1.cs [file2.cs ...]

Options:
-?, --help, -h             Shows this message.
-v, --version              Shows that the version of this program is 1.0.6352.
                            25168.
-V, --verbose              Enables verbose mode. In verbose mode, additional
                            debugging information is printed and more
                            details are written for each error message.
-q, --quiet                Enable quiet mode. In quiet mode, only the
                            resulting Viper code or error messages are shown.
-r, --reference=ASSEMBLY.DLL
                            Adds the ASSEMBLY.DLL file as a reference when
                            doing semantic analysis on the code. mscorlib
                            and Soothsharp.Contracts are added automatically.
-a, --assume=CLASS.CS      Translates the file CLASS.CS to Viper and prepends
                            it to the main generated file, but its methods
                            and functions won't be verified - their
                            postconditions will be assumed to be true.
-w, --wait                 When the program finishes, it will wait for the
                            user to press any key before terminating.
-O, --only-annotated       Only transcompile classes that have the [Verified]
                            attribute, and static methods that have the [
                            Verified] attribute even if their containing
                            classes don't have the [Verified] attribute.
-o, --output-file=OUTPUT.VPR
                            Print the resulting Viper code into the OUTPUT.VPR
                            file.
-l, --line-numbers         Print line numbers before the Viper code
-s, --silicon              Use the Silicon backend to verify the Viper code.
-c, --carbon               Use the Carbon backend to verify the Viper code.
```

To disable an option that's turned on by default, add a minus sign after it, like this: `--line-numbers-` or `-l-`. By default, line numbers are printed, and neither the verbose mode nor the quiet mode are active.

[20]https://github.com/boogie-org/boogie

[21]Some documentation is available at `https://bitbucket.org/viperproject/documentation/wiki/Home`.



Code Files:

[✓] E:\soothsharp\Examples\Thesis_Example1_Maximum_Correct.cs

A checked item means "full verification."
An unchecked item means "ignore method bodies."

Verbosity: Quiet [✓ Normal] Verbose

Verifier Backend: None [✓ Silicon] Carbon

Other options:
[ ] Verify only annotated items
[✓] Print line numbers

```
--silicon --wait E:\soothsharp\Examples\Thesis_Example1_Maximum_Correct.cs
```

Figure B.1: Screenshot of the GUI application

To output only Viper code and nothing else, use the options `--quiet --line-numbers-`.

B.4 csverify GUI

The Soothsharp Verifier GUI allows you to configure the verifier using a graphical interface. The verifier itself (*csverify.exe*) must be in the same folder as the GUI program.

The following options are exposed (see Figure B.1) :

- **Verbosity.** In quiet mode, only the translated Viper code will be written out. In verbose mode, many debugging messages will be written out.
- **Verifier backend.** Determines whether verification also occurs, and if yes, which backend verifier is used.
- **Code files.** Determines what C# code files will be verified, and whether they're fully verified or whether only their signatures are extracted.
- **Other options.** Allows the configuration of other options of the *csverify.exe* program.

The text box shows the arguments that will be passed to the command-line tool based on selected options. You may modify these arguments before running verification.

The *Run verification* button will launch *csverify.exe* with the given arguments.

B.5 Visual Studio plugin

There is also a Visual Studio plugin that automatically translates and verifies C# code using Soothsharp and the Carbon Nailgun backend verifier.

To install it, double-click the distributed *vsix* file.

Then, as long as that plugin is not deactivated, Soothsharp will automatically translate and verify C# code files open in Visual Studio. There is no per-project setting: the plugin is either active or not, so unless you're using Soothsharp, you will likely want to keep the plugin deactivated.

Translation errors are reported as errors in Visual Studio (with red squigglies). Verification errors are reported as warnings in Visual Studio (with green squigglies).

B.6 Soothsharp.Contracts

The **Soothsharp.Contracts** library should be added as a reference to each project that uses Soothsharp. The *csverify.exe* tool adds it as a reference automatically.

It contains the static *Contract* class which exposes functionality of the Viper language, some attributes, the classes *Seq* and *Permission* and some extension methods. All of these are in some way translated to Viper. Some of them have some functionality in C# as well (such as the Permission type) but most methods have empty bodies and do nothing when executed by the runtime.

More information can be found in the class documentation of the project.

See the example files provided to get a good overview of how contracts are used.

B.7 Examples

This folder contains several files annotated with Soothsharp contracts. The folder *Algorithms* is of particular interest and it contains the following methods:

- Arithmetic.Max
- Arithmetic.Min
- Arithmetic.Abs
- Search.GetSmallestNumber
- Search.BinarySearch
- SeqUtils.ArrayToSeq
- Sorting.InsertSort

In addition, the following classes are available:

- VerifiedTuple
- SortedList

- Node (a node in a directed acyclic graph)

All of these are proved correct using Soothsharp and their documentation is visible to IntelliSense.

B.8 rewrite.exe

The *rewrite.exe* tool accepts one C# code file as input: either put its filename as the only argument, or send it to *rewrite.exe* using standard input (terminated by *end-of-file*).

The rewriter will remove all contracts from the source code (i.e. calls to the methods *Assume, Assert, Ensures, Exhale, Inhale, Invariant, Requires, Fold* and *Unfold* on *Soothsharp.Contracts.Contract*). Semantic analysis is used to remove them correctly even if aliases are used.

In addition, the expressions *Folding* and *Unfolding* are removed and replaced with their second argument only. That way, the first argument — which is only relevant in Viper — doesn't remain in the code that needs to be compiled into a binary, but the executable portion of the code remains.

Use *rewrite.exe* to avoid running contract code.

B.9 Troubleshooting

- If a program fails with an error, make sure that all the DLL files that accompanied it are present in its directory.
- If verification fails, or conversely, unexpectedly succeeds, make sure that the Viper folder is in your PATH environment variable and that you have Java, Z3 and Boogie installed and that the environment variables *Z3_EXE* and *BOOGIE_EXE* are set.[22] It is not sufficient for the Viper files to be in the current directory, they need to be in the PATH.
- If verification still doesn't work, check that all executable files are in paths that don't contain non-ASCII characters (such as Czech letters) or spaces.
- There is a bug in Viper that causes problems when the current directory is on a different drive than the Viper tools. To avoid this, run verification from a folder that's on the same drive.
- Additional documentation for Viper tools is available online[23]. You may also contact the author of this thesis for assistance.

[22] Programs copy the environment as they are launched. You may need to restart programs after you set these environment variables to have them apply.

[23] `https://bitbucket.org/viperproject/documentation/wiki/Home`

C. Formal grammar of Viper

```
// Declarations
sil-program ::=
    (
        import    |
        define    |
        domain    |
        field     |
        function  |
        predicate |
        method
    )*

import ::=
    "import" relative-path

relative-path ::=
    \A(?:[\w-]+\/?)+\z

define ::=
    "define" identifier [ "(" parameter^,* ")" ] expression-or-block

parameter ::= ident

expression-or-block ::=
    exp |
    block

domain ::=
    "domain" domain-name
    "{"
        domain-function*
        axiom*
    "}"

domain-name ::=
    ident |
    ident "[" ident^,* "]"  //e.g. Seq[T]

domain-function ::=
    ["unique"] function-signature [";"]

function-signature ::=
    "function" ident formal-args ":" type

axiom ::=
    "axiom" ident "{" exp "}" [";"]

field ::=
    "field" ident ":" type [";"]

function ::=
    function-signature
    precondition*
    postcondition*
    "{"
        exp
    "}"
// Semicolon not permitted

precondition ::=
    "requires" exp

postcondition ::=
    "ensures" exp

invariant ::=
    "invariant" exp

predicate ::=
    "predicate" ident formal-args "{" exp "}" |
    "predicate" ident formal-args
// Semicolon not permitted

method ::=
    "method" ident formal-args [formal-returns]
        precondition*
        postcondition*
    [block]

formal-args ::=
    "(" formal-arg^,* ")"

formal-arg ::=
    ident ":" type

formal-returns ::=
    "returns" formal-args

// Statements
block ::=
    "{" statement-with-optional-semicolon* "}"

statement-with-optional-semicolon ::=
    stmt [";"]

stmt ::=
    // local variable declaration with an optional initial value
    "var" ident  ":" type [":=" exp] |

    // local definition
    define |

    // local variable assignment
    ident         ":=" exp |

    // field assignment
    field-access  ":=" exp |

    // object creation (all fields)
    ident         ":=" "new(*)" |

    // object creation (specified fields)
    ident          ":=" "new(" ident^,* ")" |

    "assert" exp |
    "assume" exp |
    "inhale" exp |
    "exhale" exp |
    "fold"   acc-exp |
    "unfold" acc-exp |

    "goto" ident    |            // goto statement
    "label ident    |            // a goto label

    if-statement           |
    while-statement        |
    call-statement         |
    fresh-statement        |
    wand-statement         |
    constraining-block

if-statement ::=
    "if" "(" exp ")"
        block
    ("elsif" "(" exp ")"
        block
    )*             // any number of elseif branches
    ["else"
        block
    ]              // optional else branch

while-statement ::=
    "while" "(" exp ")"
        invariant*
    block

call-statement ::= // method call [with return target]
    [ident^,* :=] ident "(" exp^,* ")"

fresh-statement ::=
    "fresh" ident^,*

wand-statement ::=
    "wand" ident ":=" exp |
    "package" magic-wand-exp |
    "apply" magic-wand-exp

constraining-block ::=
    "constraining" "(" ident^,* ")"
    block

// Expressions
binop ::=
    "==" | "!=" |                  // equality operators
    "==>" | "||" | "&&" |"<==>" |  // boolean operators
    "<" | "<=" | ">" | ">=" |      // ordering
                                   //  (integers and permissions)
    "+" | "-" | "*" |              // arithmetic operators
                                   //  (integers and permissions)
                                   // also int*permission
    "\\" | "\%" |                     // arithmetic division
                                      //  and modulo
    "\/" |                            // permission division
                                      //  (of two integers)
    "--*" |                           // magic wand

    "union" | "intersection" | "setminus"  //set operators
    "++"  |           // sequence concatenation
    "in"  |           // set/multiset/sequence membership
    "subset"          // subset relation

unop ::=
    "!" |            // boolean negation
    "+" | "-"        // integer and permission

exp ::=
    "true"  | "false" |           // boolean literal
    integer |                     // integer literal
    "null"   |                    // null literal
    "result" |                    // result literal in
    //  function postconditions
    ident    |                    // local variable read

    "(" exp ")" |

    unop exp |                    // unary expression
    exp binop exp |               // binary expression
    exp "?" exp ":" exp |         // conditional expression

    "old" "(" exp ")"             // old expression
    "[" ident "]" "(" exp ")"     // labeled old expression
    "lhs" "(" exp ")"             // apply old expression

    "none"     |               // no permission literal
    "write"    |               // full permission literal
    "epsilon"  |               // epsilon permission literal
    "wildcard" |               // wildcard permission

    "perm" "(" loc-access ")" | // current permission
    //  of given location

    acc-exp  |                    // accessibility predicate

    ident "(" exp^,* ")" |                 // [domain] function
       application
    "(" ident "(" exp^,* ")" ":" type ")" // typed function
       application


    field-access         |  // field read
    predicate-access     |  // predicate access

    "[" exp "," exp "]" |         // inhale exhale expression
    "unfolding" acc-exp "in" exp | // unfolding expression
    "folding" acc-exp "in" exp
    "applying" ( "(" magic-wand-exp ")" | ident ) "in" exp
    "packaging" ( "(" magic-wand-exp ")" | ident ) "in" exp

    // quantification
    "forall" formal-arg^,* "::" trigger^,* exp |
    "exists" formal-arg^,* "::" exp |

    seq-constructor-exp      |
    set-constructor-exp      |

    seq-op-exp  |
    "|" exp "|" |                 // length of a sequence; or
       set/multiset cardinality

    let-in-exp  |
    forperm-exp

magic-wand-exp ::=
    exp // except that it must not be <==> or ==>
let-in-exp ::=
    "let" ident "==" "(" exp ")" "in" exp

forperm-exp ::=
    "forperm" "[" ident^,* "]" ident "::" exp

seq-constructor-exp ::=
    "Seq[" type "]()"     | // the empty sequence
    "Seq(" exp^,* ")"     | // explicit sequence (must not be empty)
    "[" exp ".." exp ")"    // half-open range of numbers

set-constructor-exp ::=
    "Set" "[" type "]" "(" ")"             | // empty set
    "Set" "(" exp^,* ")"                   | // explicit set
    "Multiset" "[" type "]" "(" ")"        | // empty multiset
    "Multiset" "(" exp^,* ")"              | // explicit multiset

seq-op-exp ::=
    exp "[" exp "]"          |  // sequence lookup
    exp "[" ".." exp "]"     |  // take n first elements
    exp "[" exp ".." "]"     |  // drop n first elements
    exp "[" exp ".." exp "]" |  // take and drop
    exp "[" exp ":=" exp "]" |  // update sequence at

trigger ::=
    "{" exp^,* "}"  // a trigger for a quantification

acc-exp ::=
    "acc" "(" loc-access ["," exp ]")" //access
// default is write
loc-access ::=
    field-access | predicate-access

field-access ::=
    exp "." ident |               // field access

predicate-access ::=
    ident "(" exp^,* ")"          // predicate access

// Types
type ::=
    "Int" | "Bool" | "Perm" | "Ref" | "Rational" | // primitive
```

```
        types
    "Seq" "[" type "]" |                              // sequence type
    "Set" "[" type "]" |                              // set type
    "Multiset" "[" type "]" |                         // multiset type
    ident [ "[" type^,* "]" ]                         // [instance of
        a generic] domain type

// Identifiers
ident ::=    // regular expression for an identifier
    "[a-zA-Z$_][a-zA-Z0-9$_']*"

Expression operator priority
============================
(operators are in general, right-associative)

ternary conditional operator
<==>
==>
--*
||
&&
== !=
<= >= < > in
++ + - union intersection setminus subset (left-associative!)
* / \ % (left-associative!)
field-access seq-op-exp
other expressions
```