Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Practical Inlining in Viper

Bachelor's Thesis

Matthias Schenk (Mat.Nr.: 12-610-531)

November 23., 2022

Advisors: Prof. Dr. Peter Müller, Thibault Dardinier

Programming Methodology Group
Department of Computer Science , ETH Zürich

Abstract

This thesis looks at practical inlining in Viper and wants to improve an inlining prototype so that inlining can be added as a new feature to Viper. In the context of the static inlining tool, error reporting functionality was added and the code-base was revised. Together with the improvements for static inlining, the beforehand non-existing topic of differential inlining was explored. Differential inlining combines resource filters for the store, permission mask, and heap of a Viper state to create barriers before and after inlined code. Through their filters, barriers limit the amount of resources that can pass through them. The resources that do not pass through the barrier are framed around the inlined code. With barriers partial annotations can be applied to a Viper state and the verification results of the different barriers can be combined to derive insights about insufficient annotations. To test the developed theory about differential inlining, a prototype for using differential inlining with method calls was developed. The testing of the prototype gave new insights into the previously developed theoretical part such that parts could be refined and potential improvement identified.

Contents

1 Introduction

Viper [1] is an intermediate verification language and a suite of verification tools for this language. On top of it, it facilitates the development of automatic verifiers for different programming languages like Go (Gobra [2]), Python (Nagini [3]), and more. Viper has a verification-condition-generation-based verifier (*Carbon*) and uses the *Viper* language, the intermediate verification language of the Viper project by the Programming Methodology Group at the Department of Computer Science, ETH Zurich [4]. Carbon uses modular verification. Modular verification means that method or loop bodies are verified in isolation with respect to their annotations[1]. This means that the only interaction/information exchange between methods and called methods or used loops is through their annotations. This thesis wants to introduce a new static inlining feature to the Carbon verifier. Instead of verifying methods and loops separately, this tool inlines the method bodies or unrolls the loop bodies up to a certain depth into the body of some entry method. When inlining, the verifier does not consider any annotations for the inlined code during verification which allows the developer to do a preliminary check of the code before needing to specify any annotations for the inlined body. As an extension to the static inlining feature, this thesis will also explore the consideration of partial annotations through differential inlining. Differential inlining allows the developer to write no annotations, complete annotations, or incomplete annotations and check the behaviour of the program when filters for the resources of the surrounding context are applied. From the verification results when the program is verified multiple times with different resource filters are applied new information about what the program annotations are missing for a successful verification can be inferred. This information can then be relayed to the developer to help improve or correct the already defined annotations.

Static inlining offers a new approach compared to modular verification. Through static inlining it is possible to see if there are enough resources available in the surrounding context for a successful verification or if there are fundamental errors present in the code. This makes development more efficient since writing annotations is often very time and work intensive. The extension of differential inlining is then about helping the developer writing annotations and more easily finding missing parts.

Inlining method calls, or unrolling loops cannot replace the error detection of a fully annotated program. It is not possible to inline a recursive method infinitly, or to unroll a loop an infinite number of times. Static inlining will only ever be able to give a verification guarantee up to a certain depth. This thesis builds upon the work of T. Dardinier [5], who developed an inlining prototype for the Carbon verifier and hence this thesis will also focus on Carbon. The existing prototype does not have error reporting that gives the users information beyond the error messages of the modular Carbon verifier.

The goal of this thesis is twofold. The first goal is to introduce the new static inlining feature to Viper. The second goal is to explore differential inlining. In terms of the exploration the focus will lie on method calls only because loops are analogous to method calls.

The methodology of this thesis consists of defining what behaviour wants to be achieved, then implementing the required functionality and finally testing the implementation to see if it produces our desired output.

[1]The term annotations refers to the pre- and postconditions for methods and to invariants for loops.

As part of introducing the static inlining feature to Viper, the existing prototype had to be improved and brought up-to-date with the current codebase of Carbon and Silver. The improvements focused on the extension of the code documentation and the error reporting. Differential inlining builds upon the static inlining tool but needed to be built from the ground up. Therefore, we first give a theoretical introduction to the concepts of differential inlining and afterwards we discuss the practical approach of developing a prototype, as well as a description of the findings with examples.

The thesis structure is as follows. The first part focuses on the static inlining tool. There is an introductory part to Viper, as a foundation for the explanations and functions of static inlining. After that an overview of the improvements made to the static inlining tool will be given. This includes improvements of the documentation, the added error reporting functionality with a callstack, and high/low confidence flags. The second part will be about differential inlining. The theoretical introduction to the topic, that explains what barriers and filters are and how information can be inferred though critical links between barriers. This will be followed by a section about the practical implementation of differential inlining. After some necessary concepts for the understanding are conveyed, a high-level encoding overview of how the filtering of resources is implemented will be provided and some an example to demonstrate the prototype. The thesis will be concluded in a third part with a discussion of the most interesting findings and problems that were encountered while testing the theory with the implementation.

2 Inlining in Carbon

2.1 Background

One reason Viper is a powerful verification infrastructure is that it uses modular verification. The main method does not have any information about the method body of a called method or a loop body. For method calls, only the preconditions, postconditions and method arguments are visible. For loops, only the invariants and the guards are visible[2]. During the verification of the caller, it is assumed that all called methods and loops verify with their defined annotations. These annotations are then used to restrict the range of possible executions.
Viper builds on separation logic [6], a logic that extends the pre- and postconditions of Hoare logic [7] to reason about heap-manipulating programs. A state in Viper is a triplet consisting of the store, the heap, and the permission mask. The store contains all information on the local variables, the heap contains the values of the fields of references in the program, and the permission mask is a set of mappings from heap locations to permission fractions. A permission fraction p is a rational between 0 and 1 that defines the access rights to heap locations in an execution environment. The fine-grained control of fractional permissions to heap locations is possible because of separation logic. Fractional permissions create three cases in Viper:

1) **p=0** means that there are not enough permissions to read from or write to a heap location.
2) **p∈(0,1)** means that there are enough permissions to read from a heap location but not enough to write to it.
3) **p=1** means that there is full permission and reading from or writing to a heap location is possible.

Fractional permissions are especially useful for reasoning about concurrent heap-manipulating programs since permissions can easily be divided and distributed among threads, and a thread can only write to a heap location if and only if it holds full permission to the desired location. This also implies that no other thread can read from that location until the full permission is released from the writing thread again.
Overall, automatic verification gives developers more confidence in their code compared to traditional testing methods. It is the responsibility of the developer to provide annotations and guide the verification process. Annotations are directed at the three components of a state in Viper and hence contain permission specifications to the heap, constraints on heap values, and information about the store of local variables. Defining these annotations can still be time-consuming since they need to be defined for every method and loop in a program and might need to be adapted whenever changes to the code are made. The more complex the desired properties are, the more time will have to be invested into defining the required annotations. Especially in these cases, one would like to have assurances that the code functions properly before time is spent on annotations.
Therefore, the goal of this thesis is to introduce a new static inlining feature to Viper that can detect errors in the code before annotations are defined. Inlining method calls, or unrolling loops cannot replace the error detection of a fully annotated program. It is not possible to inline a recursive method, or to unroll a loop an infinite

[2]However, the main method identifies which variables are modified (loop targets) and which are not.

number of times. Static inlining will only ever be able to give a verification guarantee up to a certain depth.
Nonetheless, static inlining is very useful for early error detection. The program in Listing 1 contains a division-by-zero error. Without any annotations, it will throw the error that the method might not have enough permissions to *x.f*. The permission error is resolved by adding the invariant in line 8. This results in two new errors for potential division by zero in line 11 and 12. The error in line 11 is not a real division-by-zero error, since *i=0* at the beginning of the loop and *i* increases monotonically. The true error in line 12 is determined only after adding the second invariant in line 9. This is a *fundamental error*. A fundamental error is an error for which there exist no annotations such that the program will verify. These findings can be compared with the unrolled version of the program in Listing 2. The verifier can find the fundamental error without any annotations and reasoning about the behavior of *i*.

Listing 1: Error Detection in Loops

```
0  field f: Rational
1
2  method example(x:Ref, n:Int)
3      requires acc(x.f)
4      ensures true
5  {
6      var i: Int := 0
7      while (i < n)
8          //invariant acc(x.f)
9          //invariant i >= 0
10     {
11         x.f := x.f + 1 / (i + 1)
12         x.f := x.f - 1 / (i - 1)
13         i := i + 1
14     }
15 }
```

Listing 2: Error Detection in Loops (unrolled)

```
0  field f: Rational
1
2  method example(x:Ref, n:Int)
3      requires acc(x.f)
4      ensures true
5  {
6   var i: Int := 0
7   if (i < n) {
8      x.f := x.f + 1 / (i + 1)
9      x.f := x.f - 1 / (i - 1)
10     i := i + 1
11     if (i < n) {
12       x.f := x.f + 1 / (i + 1)
13       x.f := x.f - 1 / (i - 1)
14       i := i + 1
15       if (i >= n)
16       { assume false }
17     }
18  }
19 }
```

This example shows the usefulness of static inlining as a new feature of Viper. Not only does a preliminary verification through static inlining save time in writing annotations, but it also makes debugging easier by narrowing down the source of the failed verification. It answers the question of whether there is a mistake in the code, or if the annotations are not sufficient.

The initial example has shown that static inlining can be used to find fundamental errors in programs. Finding such errors is not always this straightforward with inlining. There are statements and annotations that can create verification errors which in the fully-annotated, non-inlined program would not be an issue. We define verification errors in the inlined program that do not correspond to fundamental errors as *false positives*. Such false positives are side-effects of using separation logic

and permission resources.
Listings 3 and 4 show an example of a false positive due to permission introspection. The function *perm()* returns the permission fractions currently held for a heap location. The program wants to assert that there are no longer enough permission fractions to read from *x.f* after the callee method call. The callee method acquires the permission fractions with the precondition from the caller, but it does not return the permission fractions back to the caller in the postcondition. This is called a permission leak and can, for example, be intentionally used to prevent any further writes or even reads to a reference after a method call. The statement in the callee's body will always verify, and modular verification will also verify Listing 3 with no errors. On the other hand, the inlined program has no longer a permission leak causing the assertion in line 8 of Listing 4 to fail. This implies that it is not a fundamental error. In other words, there exist annotations with which the program verifies and are thus confronted with a false positive.

Listing 3: False Positive

```
0  field f: Int
1
2  method caller(x: Ref)
3      requires acc(x.f, 1/2)
4      ensures true
5  {
6      callee(x)
7      assert perm(x.f) == 0/1
8  }
9
10 method callee(x:Ref)
11     requires acc(x.f, 1/2)
12     ensures true
13 {
14     assert true
15 }
```

Listing 4: False Positive (inlined)

```
0  field f: Int
1
2  method caller(x: Ref)
3      requires acc(x.f, 1/2)
4      ensures true
5  {
6      assert true
7      assert perm(x.f) == 0/1
8  }
```

Having an understanding if static inlining may cause false positives is important. There exist verification-preserving conditions for statements which imply that a statement does not cause any false positives when inlined [5].

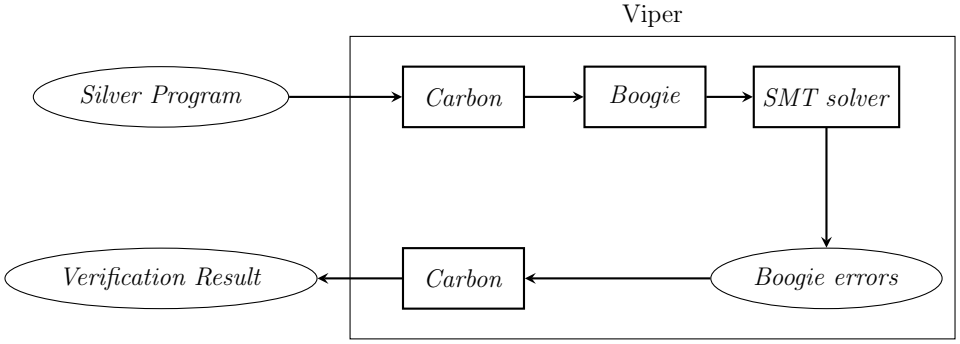
2.2 Static inlining with Boogie

The way the Carbon verifier works (shown in Figurfe 1) is that it takes a program written in the Viper language and translates it into a Boogie [8] file[3] that then gets fed to the satisfiability modulo theories (SMT) solver [4]. To produce the Boogie file, Carbon analyses every statement in the program and adds predefined assertions for the statement types into the Boogie file, together with an error message and error id. An example of an assertion for a read or write statement to a heap location is that there are enough permissions to access that location. Throughout the translation of

[3]Boogie is an intermediary verification language (IVL).

the program, and especially with the inhaling of the pre- and postconditions, the state accumulates statements with conditions that restrict possible traces, meaning execution paths. Inhaling is a Viper statement and inhaling a condition is assuming that condition on the Boogie state [4]. The counterpart of inhaling is exhaling which is an assertion on the Boogie state. Should inconsistent assumptions be made on a Boogie state, then the state becomes inconsistent. An exampel for inconsistent assumptions would be if a variable is assumed to be negative, when in the Boogie state the variable is already assumed to be strictly positve. When Boogie is in an inconsistent state, then anything can be proven because the state can be compared to $false \implies true$. To continue the aforementioned permission example, should there never have been any permissions to that heap location specified, then all traces are possible, including no permission, which will make the assertion fail. Should however full permission to that location been specified then all execution paths that do not have full permission get eliminated. That is the role of the SMT solver. It collects all defined conditions for the state within the boogie file and checks if the defined assertions of the boogie file hold within these constraints. This results in a list of Boogie errors that Carbon translates back into Viper errors and are output to the user as a verification result. Should the list of Boogie errors be empty then the verification was successful.

Figure 1: Verification Pipeline



The produced Boogie file is separated into sections. Starting with the preambles sections of the used Carbon modules that define the functions contained in the modules and their corresponding axioms that are the basis for the verification. After the preamble come the translations of the program methods. In modular verification all defined method calls are collected so that they are translated in their own section, meaning the state is reset and method calls are verified only under the consideration of their pre- and postconditions. A method body will only be translated in the section for that method. Should this method be called inside another method then the section of the caller method will only contain the exhaling of the precondition and the inhaling of the postcondition. Loops are not translated in their own section but are still verified separately of the surrounding context under consideration of their invariants. Just like with method calls, the method in which the loop is defined in will only assert and assume the invariant instead of interacting with the loop body directly.

The user has multiple options available when initiating a verification that may influence the produced Boogie file. Static inlining is activated by setting the static inlining option "*–SI*" together with the maximum depth of the static inlining. When

the static inlining option is set, only the first method in the program will be verified, or if a specific entry method is defined through another option "*–entry*", then only the entry method will be verified. But instead of containing the assertion of the precondition and the assumption of the postcondition, the method body of the called method or the unrolled loop body is directly inserted into the entry method. Below is an example that shows the structuring of sections within a Boogie file with and without inlining of a method *main* and another method *callee* that is called inside the body of main. As is shown in the outline, instead of giving every method in the Boogie file its own section, the method body of callee will be inserted in the section of the entry method. The pre- and postcondition will not be inhaled and exhaled anymore but instead be asserted, which means they will not have any effect on the traces of the program but still will be checked for their definedness and whether they hold or not. This allows two things: Firstly, even more complex called methods can be verified without any annotations. Secondly, the assertion of the pre- and postcondition helps to identify errors in the annotations even if they have no effect on the execution paths.

Listing 5: Section outline without static inlining

```
0  // ==========================================================
1  // Translation of method main
2  // ==========================================================
3          // -- Translation of method body
4
5              // -- Inhaling of precondition of method callee
6              // -- Exhaling of postcondition of method callee
7
8          // -- Translation of method body
9
10
11 // ==========================================================
12 // Translation of method callee
13 // ==========================================================
14         // -- Inhaling of preconditions
15
16         // -- Translation of method body
17
18         // -- Exhaling of postconditions
```

Listing 6: Section outline with static inlining

```
// ================================================================
// Translation of method main
// ================================================================
      // -- Translation of entry method body
      // -- Assertion of precondition of method callee

        // -- Translation of method body of method callee

      // -- Assertion of postcondition of method callee
      // -- Translation of entry method body
```

3 The improved static inlining tool

The main goal for improving the static inlining tool was to implement the error reporting so that the user will receive inlining specific information when errors were found while static inlining is active. The secondary goal was to clean up some parts of the code, extend the documentation and bring the codebase up-to-date with the current version of Carbon and Silver so that the static inlining tool could be integrated more easily into the Viper infrastructure. Extending the documentation helped for this thesis because it required in depth analysis of the code and gave a good basis for working on the primary goal. Merging the static inlining tool with the current codebase of Carbon and Silver required the resolving of a few merge conflicts but did not impact the behaviour of the Carbon verifier. The added documentation should allow new developers a quicker understanding of the static inlining module for further development, use, or research.

3.1 Error Reporting with callstack

The previously existing prototype of the static inlining module did not give error information beyond the information that was provided by Carbon. The primary information that is important to relay to the user when inlining is to give information where in the code the error occurred and at which inlining depth. The goal that was set for the error reporting was to facilitate tracing back what path in the program execution produced the error. For methods this means that the callstack is output and for loops the iteration in which the error occurred. Throughout the translation the inlining module now collects the inlined method calls and unrolled loop iterations and appends them to a callstack variable in the inlining module. When translating a new statement a snapshot of the callstack information is attached to the statement and later gets appended to the error message produced by Carbon. Listing 7 shows a code example for which the inlining messages will be demonstrated. The verification output for Listing 7 is shown in Listing 8 and the following option "–SI 5" was used to run the sbt build tool:

Listing 7: Inlining error messages code example

```
0  field f:Int
1
2  method main() {
3      var i: Int:= 0
4      m1()
5      assert false
6  }
7
8  method m1(){
9      var i:Int :=0
10     while(i<2) {
11         i := i+1
12     }
13 }
```

Listing 8: Static Inlining with option "–SI 5"

```
[info]  [0] Assert might fail. Assertion false might not hold.
    (methodCall1.vpr@6.12--6.17)  SI-depth: 0; Stacktrace: main
    -> m1@5(List()) (fin.) High confidence that real error.
```

To prevent the bloating of the output callstack information, the snapshot of the callstack taken is in a collapsed form by default. This means that if a inlined method call or unrolled loop is verified without an error then all nested method calls or loops are not included in the snapshot. The successful verification of an inlined method call or unrolled loop implies that all nested method calls or loops were also verified successfully.
To give the user more control there is also the option "*–verboseCallstack*". If this option is defined then the snapshots of the callstack will be in non-collapsed form and contain the whole execution path and all loop iterations. The *verboseCallstack* option can also be used to specify loops or methods that should not be collapsed in the snapshots. This allows for a closer inspection if for example a method is called multiple times, potentially nested, but does not always produce an error. The verification output for Listing 7 with the options "—SI 5" and "*--verboseCallstack ()*" produces the output in Listing 9, in which the collstack is presented in non-collapsed form and the verification output for Listing 7 with options "—SI 5" and "*–verboseCallstack (m1)*" produces the output in Listing 10. As can be seen, the output in Listing 10 will not collapse method *m1* but will collapse the loop that verified without error.

Listing 9: Static Inlining with options "–SI 5" and "–verboseCallstack ()"

```
[info]  [0] Assert might fail. Assertion false might not hold.
    (methodCall1.vpr@6.12--6.17)  SI-depth: 0; Stacktrace: main
    -> m1@5(List()) -> Loop@11_id1; iter.: 1 -> Loop@11_id1;
    iter.: 2 -> Loop@11_id1; iter.: 3 -> Loop@11_id1;
    iter.: 4 -> Loop@11_id1; iter.: 5 (fin.) ->
    m1@5(List()) (fin.) High confidence that real error.
```

Listing 10: Static Inlining with options "–SI 5" and "–verboseCallstack (m1)"

```
[info]  [0] Assert might fail. Assertion false might not hold.
    (methodCall1.vpr@6.12--6.17)  SI-depth: 0; Stacktrace: main
    -> m1@5(List()) -> Loop@11_id1; iter.: 5 (fin.) ->
    m1@5(List()) (fin.) High confidence that real error.
```

3.2 High/Low Confidence Reporting

T. Dardinier [5] formulated verification-preserving conditions as part of his static inlining prototype. When these conditions hold and a verification error is detected

then there exists no annotation such that the program will verify modularly. Hence the detected error is a real error with high confidence and will be denoted as such in the error message. Should the verification-preserving conditions not hold then a soundness error is produced. Soundness errors are dependent on the types of statements within the program. An example for when verification-preservation needs to be checked is if the program has permission introspection (see previous section). After a soundness error occurred, the confidence that proceeding errors are real errors is lower. It is possible that it is a real error or it could be that the verification-preserving conditions are not strong enough to capture the behaviour of this program. This is why all errors after a soundness error are denoted as having low confidence. This high and low confidence modification takes place when Carbon receives the verification results of the SMT solver during the transformation of errors from Boogie errors into Carbon errors. Only then can bee seen if the verification-preserving conditions held or did not hold for the program execution.
Listing 11 shows a code example for which the high/low confidence flag will be demonstrated. The verification output for Listing 11 is shown in Listing 12 and the following option "–SI 10" was used to run the sbt build tool. The first error is designated has having high confidence, the soundness error does not have a confidence flag, and the error after the confidence error has the low confidence flag.

Listing 11: Code example high/low confidence

```
0  field f: Int
1
2  method main(x: Ref)
3      requires acc(x.f, 1/1)
4      requires x.f>=0
5      ensures true
6  {
7      assert x.f>=1
8      callee(x)
9      assert false
10 }
11
12 method callee(x:Ref)
13     requires acc(x.f, 1/2)
14 {
15     assert perm(x.f) == 1/2
16 }
```

Listing 12: Output high/low confidence

```
[info]    [0] Assert might fail. Assertion x.f >= 1 might not
    hold. (test_confidenceLevel.vpr@7.12--7.18)  SI-depth: 0;
    Stacktrace: main High confidence that real error.
[info]    [1] FRAMING 1: Statement might not be safeMono
    ([assert.failed:assertion.false] Assert might fail.
    Assertion perm(x_1.f) == 1 / 2 might not hold.
    (test_confidenceLevel.vpr@15.12--15.28)  SI-depth: 1;
    Stacktrace: main -> callee@8(List(x)))  ?
    (test_confidenceLevel.vpr@12.1--16.2)
[info]    [2] Assert might fail. Assertion perm(x_1.f) == 1 / 2
    might not hold. (test_confidenceLevel.vpr@15.12--15.28)
    SI-depth: 1; Stacktrace: main -> callee@8(List(x))
    Low confidence that real error.
```

4 Differential Inlining: Overview

Differential inlining is an extension to static inlining. If the inlined program verifies, then there might exist an annotation for which the program verifies successfully. Developers often have some idea about the annotations required for the desired functioning of their program. Rudimentary permission specifications, like *read and write*, *read only*, or intended permission leaks, can be of great help guiding the verification. An inlining tool should therefore also support partial annotations. T. Dardinier [5] proposes the notion of barriers that consist of a combination of filters for the store, the heap, and the permission mask. Barriers can filter resources and frame them around the inlined body to create states that are dependent on the full or partial annotations of the callee. Differential inlining should help the developer find the missing parts of the annotation by giving more precise information about errors. This can be in the form of the message: "The annotations regarding the permission mask are sufficient, but the annotations regrading the heap are insufficient".
The current prototype for differential inlining focuses on method calls only and not on loops. This decision was made because loops work analogous to method calls.

4.1 Terminology

It is important to state again what a state is in Viper. A Viper state consists of three parts: The store, the permission mask, and the heap [4]. The store contains the traces of all variables in the scope and is a mapping from variables to values. The permission mask contains the amount of permission fractions to heap locations that are currently held and is a mapping from heap locations to permissions. And finally, the heap contains the traces of all heap locations to which the permission mask currently holds permissions and is therefore a mapping from heap locations to values. The heap is dependent on the permission mask. The expression *context* or *execution context* is used synonymously with the state of the execution, and the term *surrounding context* is used when inside the context of the inlined body and is referred to the context before the inlined body.

In the following sections resources and information contained within states will be discussed. Generally speaking, resources are the values and permissions contained within the state, while information is the abstract interpretation of these resources, although, when talking about resources, a distinction is needed between permissions and values of variables or heap locations.
In the setting of permissions, resources refer to permission fraction which can be split but not duplicated. Should a context have 1 permission to a heap location, then these permissions can be split and distributed among two contexts with for example 1/3 and 2/3 permission to that location. What is not possible however, is that the permission gets duplicated, such that two different contexts have 1 permission fractions to the same heap location. In Viper it is essential that there is no more than 1 permission to a heap location among all contexts, otherwise the program is inconsistent.
In the setting of store variable or heap location values, resources refer to traces which can be duplicated but cannot be split up. Should a context have permissions to a heap location $x.f$ with the traces that $x.f >= 0$, then this resource can be duplicated among multiple contexts, i.e. through method calls, without the program becoming inconsistent. However, it is not possible to split the value of $x.f$. If the value of $x.f$

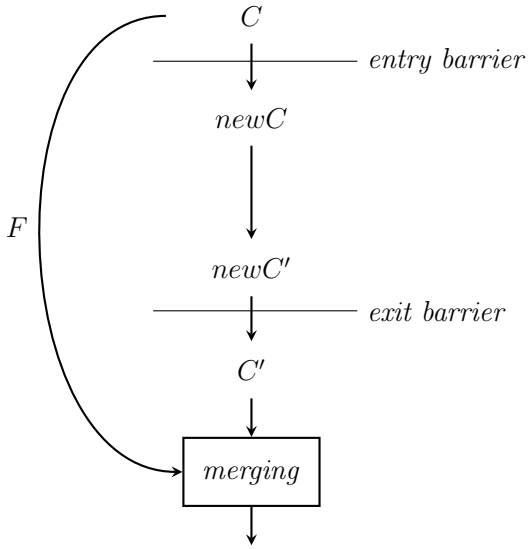
would be passed on to method calls and $x.f == 0$ is assumed in one context and $x.f > 0$ is assumed in the other, then the state would become inconsistent.
This brings us to information contained in states. Information is an interpretation of store or heap resources and is of qualitative nature, as in that there is more information or less information. When looking at traces of a heap location $x.f$ there is more information content about $x.f$ if $x.f > 0$ than when $x.f >= 0$. The amount of traces in $x.f$ is smaller, or in other words the information is more precise, when $x.f > 0$ than when $x.f >= 0$. The same distinction between more or less information does not really make sense for the permission mask. For the permission mask there are three cases: Either there is no permission, there is read permission, or there is write permission. Permissions will not be compared based on their information content, as ranking these cases in terms of information content is not possible.

The last point in the terminology used is the merging of different contexts. Merging means that the resources of two contexts are combined to form a new context. Since permissions can be split (see above) they can also be added together. So to merge the permission masks of two contexts means that the permission fractions of their permission masks get summed up for each heap location. To be able to merge two sets of traces from two contexts, either of a heap location or of a store variable, they need to be consistent. They are consistent if their intersection is not empty. Otherwise they are inconsistent which also makes the resulting state inconsistent. The merge result of two consistent sets of traces from two contexts is their intersection.

4.2 Barriers and filters

A visualization of barriers can be seen in Figure 2. C describes the context of the caller before the inlined method call. The barrier then creates two new contexts from the resources of C. $newC$ contains the resources that pass through the barrier and F contains the resources that are framed around the inlined body. This means that $C = F \oplus newC$. Because of the barrier only the resources in $newC$ are available to the inlined body. This context then might get modified throughout the execution of the inlined body which results in the context $newC'$. Again, not all resources defined in $newC'$ might pass through the barrier after the inlined body, which will determine the context C' in the end. The resources in $newC'$ that do not pass though the barrier will be lost. For the rest of the execution of the caller, it is necessary to restore the framed resources in F after inlining by merging the resources in F and C'.

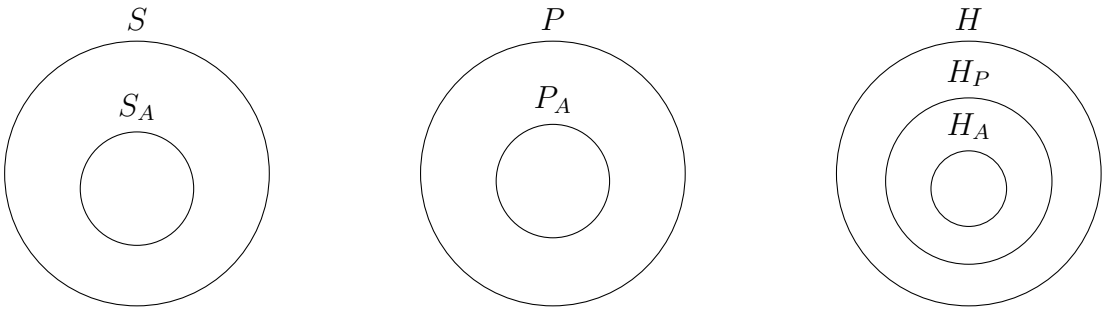
Figure 2: Contexts and Framing



Annotations can transfer at most all resources of a caller to the inlined body and in doing so keep the state of the callee. Thus, barriers cannot create stronger states and only filter resources to create weaker states. This expresses itself through less precise information on local variables, less precise information on the values in the heap, or less permissions. Viper itself does not provide enough control over the state of the program to implement barriers, this is why barriers are encoded in Boogie.

To define barriers the three parts of a state are used: the store, the permission mask, and the heap. For each of these parts filter levels are characterized that specify what resources pass through the barrier and what resources are framed. A barrier is constructed by a combination of one filter level for each resource category. Figure 3 below shows the filter levels used for this thesis. The filter S lets all information in the store of the caller pass through the barrier. The filter P lets all permissions in the permission mask of the caller pass through the barrier and the filter H lets all information in the heap of the caller pass through the barrier. Right away it can be seen that the barrier *(S,P,H)* does not consider any annotations and is equivalent to static inlining that was discussed in the previous section. The subscripts then signify additional filter properties. A is the most important subscript and means that the filter will only allow resources to pass though the barrier that are defined in the annotations. This is also the reason for the subset relation in Figure 3. If annotations are considered for a filter, then the filter is stronger compared to when they are not considered, in the sense that less information passes through the barrier. In other words, it will let fewer permissions or more traces (less information) of variables or heap locations pass through the barrier. However, it is possible that filters from annotations do not create weaker states but instead create states such that the same resources are available inside and outside the inlined body, i.e., $S = S_A$.

Figure 3: Filter levels



In the rest of this subsection examples for the functioning of filters will be presented to consolidate the understanding of filters and what resources they let pass through the barrier.

Store: As already stated, S entails all information about variables in the store from the caller. The next filter level, S_A, considers the annotations about the store. As an example, let the integer $i > 5$ be in the store of the caller and let a callee method take i as argument with the annotation $A := i >= 0$ in the precondition. Then the information that passes through a barrier with filter S will be that $i > 5$, while for a barrier with filter S_A, the information that passes through the barrier will be that $i >= 0$. The annotation can be asserted because the traces contained in the store of the caller are a subset of the traces induced by the annotation. The consideration of the annotation strengthens the filter because it lets less information about i pass through the barrier. Should there however be an annotation $B := i > 10$, then this annotation could not be verified because it is not possible to assert that i is indeed greater than 10. This demonstrates again that a filter cannot generate more information.
With regard to framing, the variables of the store are special since they either are not passed on as method arguments or they are passed on as method arguments but cannot be written to because they are method arguments [4]. This means that the traces of all store variables always are framed around the inlined body and after merging the contexts, like is depicted in Figure 2, the traces of the store variables are the same before and after the method call.

Permission Mask: Like the store, there are only two levels for the permission filter. Either everything passes through the filter or only the annotations are considered. If the inlined program verifies with P but not with P_A, then there exist enough permission resources in the main method, which could be transferred to the callee such that the program verifies. More concretely, let the caller have full permission to a heap location $x.f$ and let the annotations of the callee contain $A := acc(x.f, 1/3)$ in the pre- and postcondition. With the filter P the inlined body would still have full permission to $x.f$ and no permissions would be framed around the inlined body to be merged later. With the filter P_A however, the inlined body only would have $1/3$ permission fractions. The remaining $2/3$ permission fractions are not lost, but framed around the inlined body and merged with the $1/3$ permission fractions after the inlined body. This restores the permissions to $x.f$ to full permission after the method call.

Heap: As mentioned above, H entails all heap information from the caller. The next level H_P filters out any information about values of heap locations a method call does not have at least read permission to within the annotation. Lastly, the filter H_A considers only the annotations about the heap. As an example, let the heap locations $x.f := 5$ and $y.f := 3$ in the heap of the caller and let the callee have the annotation $A := acc(x.f, 1/2) * x.f >= 0$ in the precondition and the annotation $B : x.f >= 1$ in the postcondition. The filter H will let the information that $x.f == 5$ and $y.f == 3$ pass through the entry and exit barrier of the inlined body. The filter H_P lets the information that $x.f := 5$ pass through the entry barrier but lets no information pass through the exit barrier, since A only contains read permission to $x.f$ but not $y.f$ and B has permission for neither $x.f$ or $y.f$. The difference between H_P and H_A is that H_A will also only know the information from the annotation that $x.f >= 0$ and also have no information on $y.f$ after the entry barrier. But after the exit barrier with H_A, in context *newC'*, the context will actually have more information than with H_P since it passes the information that $x.f >= 1$

The framing of heap information is heavily dependent on the permission mask. Because, if the inlined body does not have full permission to a heap location, meaning only read permission, then the verifier will know that that location cannot be written to. The traces of that heap location can be framed around the inlined method call and be merged with the traces of that heap location contained in *C'*. The traces which got framed around it have to be consistent with the traces contained in *C'* or the state would become inconsistent. The merge result of two sets of traces will be their intersection.

On the other hand, if the inlined body does have full permission, meaning write permission, then no traces of that heap location are framed around the inlined body because the frame context does not have any permission to that location. In this case the traces of context *C'* will be the result of the merging of the frame context F and context *C'*.

Listing 13 shows a simple example to demonstrate what resources barriers let pass and what resources are framed.

Listing 13: Code example framing

```
0  field f: Int
1
2  method caller(x: Ref)
3      requires acc(x.f)
4      requires x.f >= 1
5      ensures true
6  {
7      var i:Int := 1
8      callee(x, i)
9  }
10
11 method callee(x: Ref, i: Int)
12     requires acc(x.f, 1/3)
13     requires x.f >=0
14     requires i >=0
15 {
16         assert x.f + i >= 2
17 }
```

The following overview in Table 1 shows the resources contained in the context of the inlined body after the filters were applied and what resources will be framed. It builds upon the notation of contexts from Figure 2. The context C consists of: $[perm(x.f) == 1, x.f >= 1, i >= 1]$, *newC'* equals *newC* because there are no executions that would change the state, and after merging C' with F we will always return to our original state C.

(S, P, H):	newC: perm(x.f) == 1, x.f>=1, i==1 Frame: i=1
(Sa, P, H):	newC: perm(x.f) == 1, x.f>=1, i==0 Frame: i==1
(S, Pa, H):	newC: perm(x.f) == 1/3, x.f>=1, i==1 Frame: i==1, perm(x.f)==2/3
(S, P, Ha):	newC: perm(x.f) == 1, x.f>=1, i==1 Frame: i==1
(S, Pa, Ha):	newC: perm(x.f) == 1/3, x.f>=0, i==1 Frame: i==1, perm(x.f)==2/3, x.f>=1
(S, Pa, Hp):	newC: perm(x.f) == 1/3, x.f>=1, i==1 Frame:= i==1, perm(x.f)==2/3, x.f>=1
(Sa, Pa, Ha):	newC: perm(x.f) == 1/3, x.f>=0, i==0 Frame: i==1, perm(x.f)==2/3, x.f>=1

Table 1: Barrier Comparison

Just to demonstrate, should we instead of $acc(x.f, 1/3)$ have $acc(x.f)$ in the pre- and postcondition, then it would look as follows in Table 2

(S, Pa, Ha):	newC: perm(x.f) == 1, x.f>=0, i==1 Frame: i=1

Table 2: Barrier Comparison

4.3 Gained Information

The essence of differential inlining is to verify the inlined program with different combinations of barriers and narrow down the source of errors through the verification results. The filter levels outlined before result in 12 different filter combinations. Two of these combinations have already been discussed. $\{S, P, H\}$ does not filter any information, and corresponds to the static inlining from section 2 of this thesis. $\{S_A, P_A, H_A\}$ is a barrier that only considers the annotations and hence mimics the behaviour of modular verification.
Below is a short comparison between different inlining barriers to demonstrate the intuition of differential inlining and see what information can be gained by comparing verification results. Table 3 will serve as a guideline for the hypothetical analysis. Each row represents a program with a method main as entry and a non-recursive method callee that is inlined in main. The program was verified with the different barriers from the first row. A check mark indicates that the verification was successful. A "x" indicates that the verification failed.

	(S, Pa, H)	(S_A, P_A, H)	(S, P_A, H_A)	(S_A, P_A, H_A)
1)	✓	✓	✓	✓
2)	x	x	x	x
3)	✓	x	✓	x
4)	✓	✓	✓	x
5)	✓	x	x	x

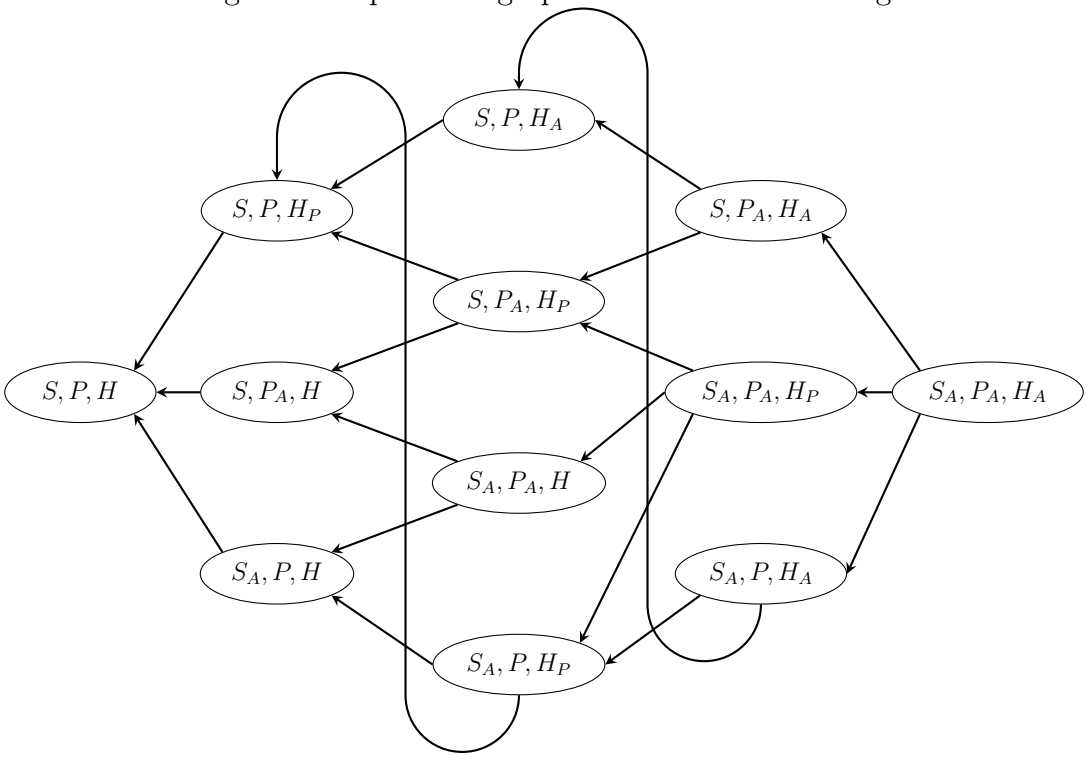
Table 3: Barrier Verification Analysis

1): The inlined program was verified for all barriers. This means that the annotations are sufficient.
2): The code is faulty and needs to be revised before annotations are considered.
3): The information about the program becomes more specific. Verification is possible since the verification succeeds when callee has access to all resources of main. In 3), the program does not verify with the annotations for the store. The attention needs to be focused on the store annotations.
4): This offers more flexibility to the developer. The program can successfully be verified by either changing the annotations for both the store and the heap, only the store, or only the heap.
5): Both heap and store annotations are not strong enough, and both need to be changed. But, as already stated before, the program verifies with a no-scope barrier and hence does not contain a fundamental error.

4.4 Implication Graph and critical links

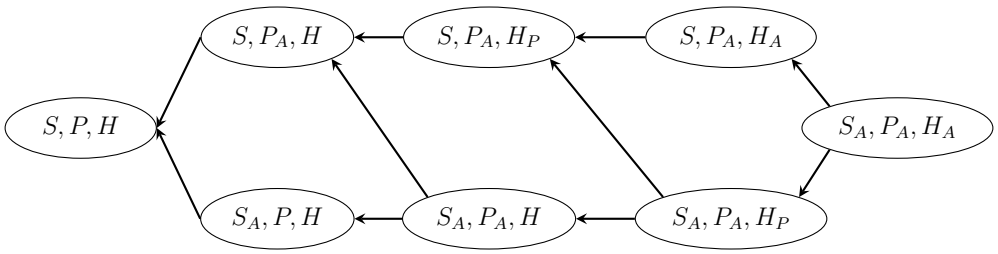
It was the hypothesis of this thesis that, based on the above introduced theory, successfully verified differential inlining barriers can imply the successful verification of other barriers. As was already touched on before, this is founded in the fact that a annotation cannot create a stronger state than the surrounding context since annotations cannot create more resources. They can only pass them on. Therefore, should the inlined program verify in a weaker, less precise state by applying the filters of a barrier, then the inlined program should also verify with a stronger, more precise state by applying less filters. Figure 4 below represents the assumed implications of the different barriers. A node is considered a parent node of a child node if the parent node has an outgoing edge and the same edge is an incoming edge for the child node. As can be deduced by the graph, the parent-child relationship can also be defined by the change of one single filter within the barrier. This does not make all barriers comparable with each other.

Figure 4: Implication graph for differential inlining



After the exploration was tested it was determined that the implication does not hold for barriers that contain the filter P. The implication does hold when you go from the surrounding context into the inlined body but not if you go from the inlined body back into the body of the caller. Because of how the heap in a Viper state is defined, the filter P makes a barrier so transparent that no annotations for heap locations matter since all heap resources pass through the entry and exit barrier. This means that the stronger the barrier with regard to the permission mask, the less heap resources can be framed. The filter P makes the barriers (S, P, H), (S, P, H_P), and (S, P, H_A) equivalent to each other, as well as the barriers (S_A, P, H), (S_A, P, H_P), and (S_A, P, H_A).
With this information a new implication graph can be formulated that is shown in Figure 5 and it reduces the number of viable barriers to 8.

Figure 5: Modified implication graph for differential inlining



This modified graph can also be modeled for a specific example like the code example from Listing 14. Figure 6 represents the verification results of that example. All green nodes imply that the designated barrier verified successfully and all red nodes imply that the barrier failed to verify. As can be seen, all barriers that contain the filter P_A do not verify. This is because the inlined method body does not have access to the heap location $x.f$ if there is no accessibility predicate with strictly positive permissions defined in the annotations. That there might not be enough permission to access $x.f$ does not yet yield any new information compared to when the program would be verified modularly or with static inlining. After the annotation "*requires acc(x.f)*" is added to the precondition the situation become more clear. The new verification results of differential inlining can be seen in Figure 7. The colour coding of the barriers shows that the verification fails as soon as both the annotations for the store and heap are considered, but not if only the annotations for the store or only the annotations for the heap are considered. This is more precise information on the error reason than would be gained with modular verification or static inlining. Modular verification or static inlining would only relay the information that the assertion $x.f + i >= 0$ might not hold.

Listing 14: Differential Inlining implication graph code example

```
0  field f: Int
1
2  method caller(x: Ref)
3      requires acc(x.f)
4      requires x.f >= 1
5      ensures true
6  {
7      var i:Int := 1
8      callee(x, i)
9  }
10
11 method callee(x: Ref, i: Int)
12     requires x.f >=0
13     requires i >=0
14 {
15         assert x.f + i >= 1
16 }
```

Figure 6: Differntial inlining implication graph example result 1

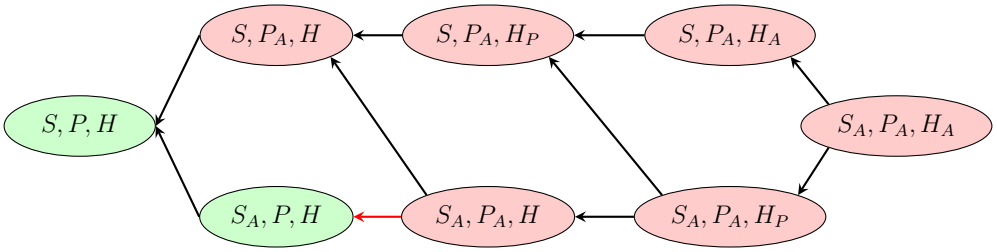
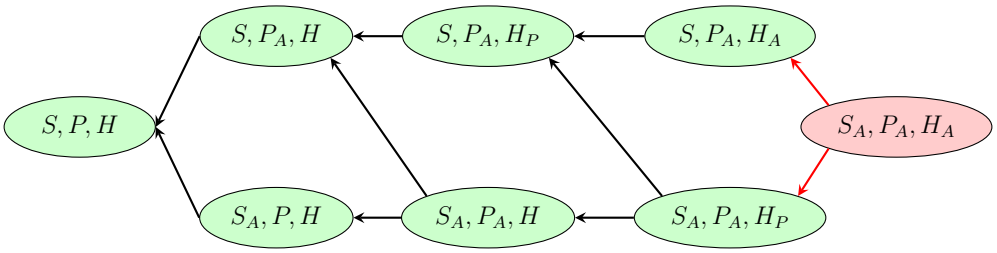


Figure 7: Differntial Inlining implication graph example result 2



Once the verification results of the barriers are available, the important parent-child edges need to be identified for the analysis. The relevant information for the developer is on the one hand all failed barriers and on the other hand the barriers for which all parent nodes failed verification. Should not all parent nodes of a child node have failed, then there exists a barrier with stronger assumptions that still verified. Therefore, the edge between the successful child node and the failed parent node is a *critical link* if all parent nodes of the child node failed verification. The critical link carries the information on which components of the annotations need to be modified for a successful verification with all annotations.

5 Differential Inlining: Encoding

This section will look in more detail at the Boogie encoding that was used for the implementation of the barriers whose behaviour was explained in the previous section. For that there will first be a few remarks on inhaling and exhaling of pre- and postconditions, before a high-level outline for the different barriers will be presented. The section will then close with a few examples that demonstrate the output of the differential inlining implementation.

5.1 Exhaling and Inhaling annotations

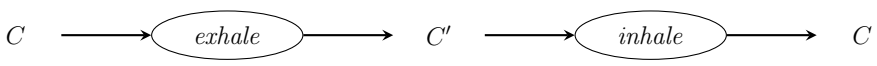
The Listing 5 in section 2.2. showed the outline of modular verification, where the method *callee* is verified separately. During static inlining the pre- and postcondition are asserted but the annotations are not applied to the state, as was seen in Listing 6 in section 2.2. However, when differential inlining is active, partial annotations are applied to the state, as is depicted in Listing 15 below.

Listing 15: Section outline with differential inlining

```
0  // ==========================================================
1  // Translation of method main
2  // ==========================================================
3          // -- Translation of entry method body
4          // -- Exhaling of precondition of method callee
5
6            // -- Inhaling of precondition of method callee
7            // -- Translation of method body of method callee
8            // -- Exhaling of postcondition of method callee
9
10         // -- Inhaling of postcondition of method callee
11         // -- Translation of entry method body
```

Before the high-level outline of the barriers it will be shown that an exhaling and inhaling block of a pre- or postcondition does not change the state under the assumption that there are not any other modifications to that state and heap locations do not get havoced when permissions are fully exhaled. This is necessary in order to be certain that the state for the verification is not changed when exhaling and inhaling a pre- or postcondition during differential inlining. It also shows that there is no need to split the pre- and postcondition into annotations for the store, the permission mask, and the heap when implementing the filters for differential inlining. Figure 8 visualizes the effect of an exhale-inhale block on a context C.

Figure 8: Exhale/inhale figure



Impact on Heap and Store: During differential inlining we exhale and inhale assertions for the annotations about heap locations and store variables while there are assumptions about their traces in the current context. Exhaling asserts the annotations in regard to the traces of the context. If the assertions from the

annotations are weaker then the exhale will succeed. If the assertions from the annotations are stronger then the exhale will fail and an error will be thrown. If inconsistent assertions are exhaled then the exhale will fail and an error will be thrown.
If the inhaled annotations contain weaker assumptions then the traces of context *C'*, then the traces remain unchanged. If the inhaled annotations contain stronger assumptions then the traces in context *C'* will be strengthened and if the inhaled annotations contain inconsistent assumptions then it will result in an inconsistent state. The case that the assumptions from the annotations are stronger or inconsistent will not occur because in such a case the exhale would have failed beforehand.
In summary, either the traces in context C will remain unchanged after an exhale-inhale block or an error will be thrown during exhaling.

Impact on Permission Mask: Exhaling and inhaling has a different effect on the permission mask then it does on the heap. When permissions are exhaled then they are subtracted from the permission mask and when they are inhaled they are added to the permission mask. Exhaling asserts that no more permissions get exhaled than are available in the current context C or an error will be thrown. Afterwards, if the same amount of permissions are inhaled then the permission mask will be the same before and after the exhale-inhale block. Should the permission mask have more than 1 permission fractions to a heap location after the exhale-inhale block then the state is inconsistent. But, this also implies that the state was inconsistent before the exhale-inhale block.

5.2 Encoding steps

Before the high-level overview of the different barrier encodings are discussed, the building blocks of the high-level outlines are presented. The state modifications of the barrier encodings are done in four steps as shown in Listing 16. The steps are: *Save Frame, Modify state before inlined body, Modify state after inlined body,* and *Merge frame with resources after exit barrier*.

Listing 16: Section outline with differential inlining

```
0  // ==================================================
1  // Translation of method main
2  // ==================================================
3      // -- Translation of entry method body
4        // -- Renaming of method callee variables
5
6          // -- Save Frame
7                includes exhaling precondition
8          // -- Modify state before inlined method callee
9                includes inhaling precondition
10
11         // -- Translation of method body of method callee
12
13         // -- Modify state before inlined method callee
14               includes inhaling and exhaling of postcondition
15         // -- Merging of resources
16
17       // -- Translation of entry method body
```

There are two important Boogie functions, definded by Carbon, that are used inside the above mentioned steps. The first function is *SumMask*(*ResultMask*, *Mask1*, *Mask2*) that takes as arguments three variables of type MaskType, add together the permission fractions for each heap location inside *Mask1* and *Mask2*, and stores the sum in ResultMask. This function is used to merge permission resources in the merging step.
The other function is called *IdenticalOnKownLocations*(*Heap1*, *Heap2*, *Mask*). This function takes as arguments two variables of type HeapType, *Heap1* and *Heap2*, and a variable of type MaskType, *Mask*. It then assumes the same traces for both heaps on all heap locations that *Mask* has non-zero permission to. Should Mask have no permissions to a heap location then the traces of both heaps for that location are not assumed to be equal. Assuming the same traces for the two heaps means that the intersection of the set of traces is assumed and the rest are eliminated. If the set of traces have an empty intersection the state becomes inconsistent.

Save Frame: As previously stated, to frame resources: Firstly, the resources to be framed need to be identified; secondly, the framed resources need to be saved in a separate context to be able to merge them again after the inlined body. A variable *FrameHeap* of type HeapType and a variable *FrameMask* of type MaskType are used to frame the resources of the heap and the permission mask around the inlined body of the method call. The FrameHeap gets assigned a snapshot of the current heap before the precondition gets exhaled to avoid that a heap location gets havoced when all permissions are exhaled. Meanwhile, the FrameMask gets assigned a snapshot of the current mask after the precondition gets exhaled so that only the remaining permission resources are framed. Even if a heap location does not get havoced when all permissions get exhaled this does not mean that these resources will be framed because heap resources are merged with the function *identicalOnKnownLocations(Heap, FrameHeap, FrameMask)*. If the FrameMask does not have any permissions to a heap location then the function will ignore these heap locations. In terms of the store, there is no

need to additionally save the store because the variables of the store get renamed before inlining a method call and therefore do not get modified by the inlined body.

Modify state sections: In these sections the state manipulations take place to impose the filters for the entry and exit barriers, meaning that the states after these sections only contain resources that the filters let pass through the barrier. Generally speaking, the permission mask can be prepared by zeroing the mask before inhaling the annotations and the heap locations and store variables can be prepared by havocing them before inhaling the annotations. The filter H_P requires a few more manipulations but they will be explained in more detail in the corresponding outline.

Merge frame with resources after exit barrier: Here, the framed resources and the resources that passed through the exit barrier of the inlined method call are merged to create the context for the remaining program execution.

5.3 Barrier high-level outline

5.3.1 (S,P,H)

The (S, P, H) barrier takes no annotations into account and is the same as static inlining. Hence, there are no additional modifications necessary and only static inlining will be performed. The use of this barrier is to see if there are enough resources in the surrounding context for successful verification. As was discussed in section 4.3., the barriers (S, P, H_P) and (S, P, H_A) are equivalent to the barrier (S, P, H) and therefore are encoded the same way.

Listing 17: High-level outline for barrier (S, P, H)

```
0  // ============================================================
1  // Translation of method main
2  // ============================================================
3      // -- Translation of entry method body
4      // -- Assertion of precondition of method callee
5        // -- Renaming of method callee variables
6
7          // -- Translation of method body of method callee
8
9      // -- Assertion of postcondition of method callee
10     // -- Translation of entry method body
```

5.3.2 (S_A,P,H)

The barrier (S_A, P, H) takes only the annotations with regard to the store into account. Before the store variables can be havoced, the relevant variables need to be collected. This is done with a function in Carbon and not within Boogie. Line 11 of Listing 18 represents the set of relevant store variables even though there is no function *collectStoreVariables()* defined in Boogie. The relevant variables are the renamed formal method arguments without references. References are the pointers to heap locations. Should they be havoced as well, then heap locations could not be

accessed even if there would be enough permissions in the permission mask to do so. Havocing references from the store would cause unwanted side effect with the other filters. Apart form that it is important to mention that the exhaling and inhaling of the annotation with no other modifications results in the same permission mask and heap as from before the exhaling and inhaling, as was shown in section 5.1.
Not all barrier outlines with filter S_A are presented because to get from another barrier outline with filter S to a barrier outline with filter S_A one only needs to add the same modifications as in Listing 12 below.
Further, the high-level outline for (S_A, P, H) is the same as the high-level outline for (S_A, P, H_P) and (S_A, P, H_A) because of the same rationale as to why the outlines for (S, P, H_P), (S, P, H_A), and (S, P, H) are the same.

Listing 18: High-level outline for barrier (S_A, P, H)

```
0  // ============================================================
1  // Translation of method main
2  // ============================================================
3      // -- Translation of entry method body
4        // -- Renaming of method callee variables
5
6            // -- Save Frame
7               FrameHeap:= Heap;
8               FrameMask:= Mask;
9               exhale precondition //no havoc of Heap
10           // -- Modify state before inlined method callee
11              StoreVariables: Set[LocalVar] =
12                  collectStoreVariables()
13              havoc StoreVariables
14              Inhale Precondition
15
16           // -- Translation of method body of method callee
17
18           // -- Modify state before inlined method callee
19              Exhale Postcondition
20              havoc StoreVariables
21              Inhale Postcondition
22           // -- Merging of resources
23              //nothing to merge on Mask
24              //nothing to merge on Heap
25
26       // -- Translation of entry method body
```

5.3.3 (S,P_A,H)

The (S, P_A, H) barrier takes only the annotations into account that modify the permission mask. The remaining permissions, after exhaling the precondition, are saved in the FrameMask to frame them around the inlined body. Exhaling the precondition will not havoc heap locations in case all permissions for a heap location are exhaled because the filter H lets all heap resources pass through the barrier. Although the FrameHeap saves a snapshot of the Heap, it is not within the framed context because the FrameHeap will not be merged with the state after the exit barrier.

To enforce the entry and exit barrier the Mask will be zeroed before inhaling the annotations, while the Heap will not be havoced. After the exit barrier, the Mask will have inhaled the permissions of the postcondition and the resources of the Heap will not have been influenced by the annotations. Therefore it is only necessary to merge the Mask with the FrameMask through the function *sumMask* but it is not necessary to merge resources from the FrameHeap with the Heap through the function *identicalOnKnownLocations*.

Listing 19: High-level outline for barrier (S, P_A, H)

```
0  // ============================================================
1  // Translation of method main
2  // ============================================================
3      // -- Translation of entry method body
4        // -- Renaming of method callee variables
5
6          // -- Save Frame
7             FrameHeap:= Heap;
8             FrameMask:= Mask;
9             exhale precondition //no havoc of Heap
10            FrameMask:= Mask;
11         // -- Modify state before inlined method callee
12            Mask:= ZeroMask;
13            Inhale Precondition
14
15         // -- Translation of method body of method callee
16
17         // -- Modify state before inlined method callee
18            Exhale Postcondition //no havoc of Heap
19            Mask:= ZeroMask;
20            Inhale Postcondition
21         // -- Merging of resources
22            assume sumMask(ResultMask, frameMask, Mask);
23            Mask:= ResultMask;
24            //nothing to merge on Heap
25
26       // -- Translation of entry method body
```

5.3.4 (S, P_A, H_A)

The (S, P_A, H_A) barrier takes the annotations into account for the permission mask and the heap. This barrier high-level outline has the same reasoning for the permission mask as the barrier (S, P_A, H). Here however, the FrameHeap is used to frame heap resources around the inlined body. A snapshot of the Heap is saved in FrameHeap before the exhaling of the precondition and the Heap itself will be havoced before inhaling annotations for the entry and exit barrier. This ensures that only the heap resources specified in the annotations pass through the barrier.
After the exit barrier, the FrameHeap will be used to merge the context after the exit barrier and the framed resources with the function *indenticalOnKnownLocations(Heap, FrameHeap, FrameMask)*. The FrameMask will be zero for all heap locations for which the inlined method body was given all permission of the surrounding context to. For the merging of the context after the exit barrier and the

framed context this means that all heap resources in Heap remain unchanged for these locations. For heap locations where the FrameMask is not zero, after the merge Heap will contain the intersection of heap resources of Heap and FrameHeap.

Listing 20: High-level outline for barrier (S, P_A, H_A)

```
0  // ============================================================
1  // Translation of method main
2  // ============================================================
3      // -- Translation of entry method body
4        // -- Renaming of method callee variables
5
6          // -- Save Frame
7             FrameHeap:= Heap;
8             FrameMask:= Mask;
9             exhale precondition //no havoc of Heap
10            FrameMask:= Mask;
11         // -- Modify state before inlined method callee
12            havoc Heap;
13            Mask:= ZeroMask;
14            Inhale Precondition
15
16         // -- Translation of method body of method callee
17
18         // -- Modify state before inlined method callee
19            Exhale Postcondition //no havoc of Heap
20            havoc Heap
21            Mask:= ZeroMask
22            Inhale Postcondition
23         // -- Merging of resources
24            assume sumMask(ResultMask, frameMask, Mask);
25            Mask:= ResultMask;
26            assume identicalOnKnownLocations(Heap,
27                FrameHeap, FrameMask)
28
29       // -- Translation of entry method body
```

5.3.5 (S, P_A, H_P)

The (S, P_A, H_P) barrier takes the annotations into account for the permission mask and will let only heap resources pass through the barrier for which some positive permission is defined in the annotation. The high-level outline has the same reasoning for the steps of framing resources and merging resources as the barrier (S, P_A, H_A). The difference for this barrier outline lies in the steps that modify the state. To only let the heap resources pass through the barrier, we first zero the Mask and inhale the annotations. Through the inhaling on the zeroed mask the filter P_A is satisfied. Once the Mask contains only the permission resources of the annotations the function *identicalOnKownLocations(FrameHeap, tempHeap, Mask)* can be called with a previously havoced tempHeap. This will assume all heap resources of FrameHeap on tempHeap for heap locations that the annotations specify permissions to. When Heap gets assigned tempHeap the state then fullfills the desired properties of the barrier.

Listing 21: High-level outline for barrier (S, P_A, H_P)

```
// ============================================================
// Translation of method main
// ============================================================
    // -- Translation of entry method body
      // -- Renaming of method callee variables

          // -- Save Frame
             FrameHeap:= Heap;
             FrameMask:= Mask;
             exhale precondition
             FrameMask:= Mask;
          // -- Modify state before inlined method callee
             Mask:= ZeroMask;
             inhale preconditions
             havoc tempHeap;
             assume identicalOnKnownLocations(FrameHeap,
                 tempHeap, Mask);
             Heap:= tempHeap;

          // -- Translation of method body of method callee

          // -- Modify state before inlined method callee
             Mask:= ZeroMask;
             inhale preconditions
             havoc tempHeap;
             assume identicalOnKnownLocations(Heap,
                 tempHeap, Mask);
             Heap:= tempHeap;
          // -- Merging of resources
             assume sumMask(ResultMask, frameMask, Mask);
             Mask:= ResultMask;
             assume identicalOnKnownLocations(Heap,
                 FrameHeap, FrameMask)

      // -- Translation of entry method body
```

5.4 Examples

In the way differential inlining is implemented, all barrier encodings are placed into the same Boogie file. Each barrier is placed into its own section as depicted in Listing 22 below, which shows only three barriers for demonstration purposes. Because all barriers have their own section, their states are independent of each other.

Listing 22: Section outline of differential inlining

```
0  // ==========================================================
1  // Translation of method main_Barrier_SPH
2  // ==========================================================
3        // -- Translation of method body
4
5
6  // ==========================================================
7  // Translation of method main_Barrier_SPaH
8  // ==========================================================
9        // -- Translation of method body
10
11
12 // ==========================================================
13 // Translation of method main_Barrier_SPaHp
14 // ==========================================================
15       // -- Translation of method body
```

By placing all barriers into the same Boogie file, Boogie might for the same statement collect errors multiple times from different barriers. This is why the errors need to be aggregated to avoid redundancy. For every statement that throws an error, the barriers that contained the same error are collected and through them the critical links (see section 4.4.) can be derived for each error. From the final list of Boogie errors, a set of errors is aggregated into a single error if they are for the same statement, i.e. have the same position in the code, have the same callstack that led to the error, and are from different barrier sections. After the aggregation, a differential inlining error message will be appended to the errors. The appended differential inlining error message contains the list of failed barriers for which the error occurred and a recommendation for which part of the annotations needs to be modified to resolve the error.
To give the reader a clearer understanding of the differential inlining output, there is a practical demonstration of an error message produced by differential inlining with the already introduced example from Listing 14 in section 4.4. For the convenience of the reader, the code example of Listing 14 is provided another time in Listing 23 below. The verification output for Listing 23 is shown in Listing 24 and the following options "*–SI 3*" and "*–diffInl*" were used to run the *sbt* build tool:

As can be seen in the verification output, the failed barriers are congruent with the analysis provided in section 4.4. And if the code example is modified with the accessibility predicate *acc(x.f, 1/2)* in the precondition (see Listing 25) then the verification output in Listing 26 is obtained. Also for this output also the failed barriers and critical links are congruent with section 4.4.

Listing 23: Differential Inlining code example 1

```
field f: Int

method caller(x: Ref)
    requires acc(x.f)
    requires x.f >= 1
    ensures true
{
    var i:Int := 1
    callee(x, i)
}

method callee(x: Ref, i: Int)
    requires x.f >=0
    requires i >=0
{
        assert x.f + i >= 1
}
```

Listing 24: Differential inlining verification output for Listing 23

```
[info]   [0] Assert might fail. There might be insufficient
    permission to access x_1.f (test_StoreHeap.vpr@11.1--17.1)
    SI-depth: 1; Stacktrace: main -> callee@8(List(x, i));
[info] Differntial Inlining: Failed in 6 of 8 barriers:
    SPaHa, SPaH, SaPaHa, SaPaH, SPaHp, SaPaHp;
[info]  SaPH succeeded;  SaPaH failed => Enough resources in
    surrounding context; Sufficient annotations for: store;
    Insufficient annotations for: mask, heap
[info]  High confidence that real error.
```

Listing 25: Differential Inlining code example 1

```
field f: Int

method caller(x: Ref)
    requires acc(x.f)
    requires x.f >= 1
    ensures true
{
    var i:Int := 1
    callee(x, i)
}

method callee(x: Ref, i: Int)
    requires acc(x.f, 1/2)
    requires x.f >=0
    requires i >=0
{
        assert x.f + i >= 1
}
```

Listing 26: Differential inlining verification output for Listing 25

```
[info]   [0] Assert might fail. Assertion x_1.f + i_1 >= 1 might
    not hold. (test_StoreHeap.vpr@16.12--16.24)  SI-depth: 1;
    Stacktrace: main -> callee@8(List(x, i));
[info] Differntial Inlining: Failed in 1 of 8 barriers:  SaPaHa;
[info]  SaPaHp succeeded;  SaPaHa failed => Enough resources in
    surrounding context; Sufficient annotations for: store, mask;
    Insufficient annotations for: heap
[info]  SPaHa succeeded;  SaPaHa failed => Enough resources in
    surrounding context; Sufficient annotations for: mask, heap;
    Insufficient annotations for: store
[info]  High confidence that real error.
```

6 Conclusion and Future Work

6.1 Conclusion

The scope of this thesis was to improve the static inlining tool so that it could be integrated as a feature into the Viper infrastructure. Static inlining is used to perform a preliminary check of a program code to see if there are enough resources in the surrounding context for a successful verification of methods and loops. With static inlining fundamental errors can be found without having to specify any annotations for inlined methods or unrolled loops. The goal was to facilitate the further development of the static inlining tool and provide the user of the tool with inlining related error information. Therefore, the improvements to the tool involved the expansion of the documentation, the reworking of some parts of some code, and the introduction of inlining related error messages. When considering what information to relay to the user it crystallised that the most important information are on one hand the output of the callstack and on the other hand the assignment of confidence levels to errors. The callstack is an ordered list that collects all inlined methods or unrolled loop iterations and is extended every time a method is inlined or an iteration of a loop is unrolled. This allows the user to follow the execution of the inlined program. The confidence levels give the user a feedback in case the inlining violates verification-preserving conditions. Low confidence means that the verification-preserving conditions are too coarse to capture the behaviour of the program and as a consequence the verification cannot be sure that it is a fundamental error or if it is a false positive. This signals to the user which errors demand special attention.

Further, differential inlining, an extension of static inlining, was explored for method calls. Differential Inlining uses barriers consisting of filters for the store, the permission mask, and the heap to verify an inlined program with partial annotations. The verification results of the barriers can be compared to derive information about the annotations of inlined method calls. The goal was to implement the barriers in Carbon which first required to formulate the meaning of barriers and filters, and to define what the desired behaviour of a barrier is. This resulted in a hypothesis about how successfully verified barriers imply the successful verification of other barriers. By testing the hypothesis and further analysis it could be determined that the use of filter P has more implications for the amount of resources that can be framed than initially assumed. The more permission resources pass through a barrier, the less heap resources can be framed around the inlined body. This resulted in the barriers (S, P, H), (S, P, H_P), and (S, P, H_A) to be equivalent, as well as the barriers (S_A, P, H), (S_A, P, H_P), and (S_A, P, H_A) to be equivalent. As a consequence, the implication graph for barriers was revised to reflect these findings. After the fundamentals of differential inlining were narrowed down, the barrier encoding outline could be formulated and tested in the implementation.

Static inlining has a good potential to make verification with Viper more efficient and the improvements of the tool that were made in this thesis will help with the adaptation of static inlining. Differential Inlining further bridges the gap between modular verification and static inlining, and can be used as an intermediary step to arrive from a successful verification with static inlining to the complete annotations required for modular verification.

6.2 Future work

There are several areas in which static inlining can still be improved. Verification can be computationally very intensive. A recursive method that spawns multiple children would have exponential growth when inlined. Increase of performance might allow verification up to a greater depth. Another, more sophisticated approach, would be to use stratified inlining, which determines the more interesting traces of the program and selectively explores them in greater detail [9]. Those approaches would increase the effectiveness of finding errors, and the more precise information would increase confidence in the tool.
Another potential area for research is the extension of the verification-preserving conditions. The verification-preserving conditions mentioned before are useful to guarantee the absence of false positives when inlining. Nonetheless, there are still problems with false positives. Programs that cannot produce false positives exist, and yet the verification-preserving conditions are too coarse to capture these programs. Therefore, extending the verification-preserving conditions in a way that more programs could be captured or more statements could be supported, like for example wildcards for permissions, would bring additional value to the tool.

In terms of differential inlining, many more things can be explored, the most obvious being the implementation of loops. As was stated previously, loops are analogous to method call. Hence, the high-level outline provided in this thesis can be useful to implement differential inlining for loops.
Further, the performance of differential inlining can be improved. In the current implementation of differential inlining all barriers are encoded inside one Boogie file. By splitting the barriers into multiple files, the verification of the different barriers could be parallelized which would also allow to explore the usage of a dichotomic search throughout the barrier implication graph presented in section 4.4.
The last research point to be mentioned for future work on differential inlining is the exploration of more or different filters in the barriers. One suggested extension on the filters used in this thesis would be the introduction of a filter S_R. As was expounded in section 5.3.2, the references in the store are not havoced to avoid side effects with the other filters. The filter S_R would also havoc references in the store, it might therefore be interesting to know what exactly are the consequences of havocing references and how it would impact verification results.

List of Figures

List of Tables

Listings

References

[1] P. Müller, M. Schwerhoff, and A. 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.

[2] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV)*, volume 12759 of *LNCS*, pages 367–379. Springer International Publishing, 2021.

[3] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 596–603, Cham, 2018. Springer International Publishing.

[4] Viper tutorial. http://viper.ethz.ch/tutorial/. Accessed: 2022-11-23.

[5] Thibault Dardinier. Beyond the frame rule: Static inlining in separation logic. Master's thesis, ETH Zurich, Zurich, 2020.

[6] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[8] K. Rustan M. Leino. This is boogie 2. June 2008.

[9] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A solver for reachability modulo theories. In *Computer-Aided Verification (CAV)*, July 2012.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Practical Inlining in Viper

Verfasst von (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Schenk	Matthias

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
8700 Küsnacht, 23.11.2022	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.