Chair of
Software Engineering

Automatic Verification of Eiffel Programs

Master Thesis

By: Julian Tschannen
Supervised by: Martin Nordio
Prof. Bertrand Meyer

Student Number: 02-715-498

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf
Informatik
Computer Science

Abstract

Correctness of software systems can be proven by using static verification techniques. Static verifiers such as Spec# and ESC/Java have been developed for object-oriented languages. These verifiers have shown that static verification can be applied to object-oriented languages such as C# and Java. However, these verifiers are not easy to use, as they introduce many new concepts that programmers have to learn. To apply these verifiers to a real project, one has to modify existing code by adding contracts and annotations such as pure method marks or ownership information. Eiffel supports *Design by Contract*. Current libraries and programs are therefore already annotated with contracts.

The goal of this thesis is to develop an automatic verifier for Eiffel which can prove existing code without the need of further annotations. The main features supported by the tool are agents and dynamic invocation. The tool, called *EVE Proofs*, translates Eiffel programs to Boogie and runs a fully automatic theorem prover to check correctness of the code. *EVE Proofs* is integrated in EVE, the Eiffel Verification Environment. This integration enables Eiffel programmers to use it directly from the programming environment. To reason about framing of routines, one needs modifies clauses. To prove existing Eiffel code, we implemented an automatic extraction of modifies clauses. Although the extraction of modifies clauses is limited, it enables us to prove examples with simple frame conditions.

To show the feasibility of *EVE Proofs*, we present some examples which can be successfully verified. The examples include the *command pattern* to show the verification of agents and the *strategy pattern* to show the verification of dynamic invocation.

Acknowledgments

My gratitude goes to my supervisor Martin Nordio for his continuous support, interesting discussions, and valuable feedback, to Prof. Bertrand Meyer for giving me the possibility to work on this interesting topic, and to my family which supported me during my whole time at ETH.

Contents

1 Introduction **9**

2 Verification Methodology **11**
- 2.1 Overview . . . 11
- 2.2 Basic Encodings . . . 12
  - 2.2.1 Routine Specification . . . 12
  - 2.2.2 Routine Calls . . . 13
- 2.3 Dynamic Invocation . . . 15
- 2.4 Agents . . . 17
- 2.5 Automatic Extraction of Modifies Clauses . . . 21
- 2.6 Pure Inference . . . 25

3 Translation to Boogie **27**
- 3.1 Introduction to Boogie . . . 27
- 3.2 Naming conventions . . . 29
- 3.3 Background Theory . . . 30
  - 3.3.1 Basic Definitions . . . 30
  - 3.3.2 Typing . . . 31
  - 3.3.3 Agent Theory . . . 33
- 3.4 Attributes . . . 34
- 3.5 Routine Signature . . . 35
  - 3.5.1 Procedure Definition . . . 36
  - 3.5.2 Arguments . . . 36
  - 3.5.3 Pre- and Postconditions . . . 38
  - 3.5.4 Class Invariants . . . 38
  - 3.5.5 Frame Condition . . . 39
  - 3.5.6 Creation Routines . . . 39
  - 3.5.7 Pure Functions . . . 39
- 3.6 Routine Implementation . . . 41
  - 3.6.1 Structure . . . 41
  - 3.6.2 Object creation . . . 44
  - 3.6.3 Assignment . . . 44
  - 3.6.4 Routine calls . . . 45
  - 3.6.5 Agent Creation . . . 45
  - 3.6.6 Agent Call . . . 45
  - 3.6.7 Check Instruction . . . 46
- 3.7 Control Structures . . . 46
  - 3.7.1 Conditional . . . 46

3.7.2 Loop 47
3.8 Tracing Errors 48

4 User Interface 51
4.1 Starting a Proof 52
4.2 Result Display 52

5 Implementation 57
5.1 Proof Engine 57
5.2 User Interface 59

6 Case Studies 61
6.1 Account 61
6.2 Formatter 62
6.3 Command Pattern 65
6.4 Strategy Pattern 66

7 Conclusions 71
7.1 Conclusions 71
7.2 Future Work 72
7.3 Related Work 74

Chapter 1

Introduction

Software systems are increasing not only in size but also in complexity. Proving that these systems are correct is challenging. This effort consumes a lot of time and resources. Only critical software systems which justify the additional effort are proven these days.

To prove programs, we have to write its specification. The specification language needs to be simple for programmers to use it. Furthermore, these specification languages need to be expressive so that a static verifier can prove the given program. There exist extensions to object-oriented languages to include the specification as part of the program, such as Spec# [16, 12] and JML [17, 9]. Since Eiffel [24, 3, 25] uses the *Design by Contract* method, it has built-in support to write the specification as part of the program. This has the advantage that the programmers are already accustomed to the specification language and that existing libraries and programs are already annotated with contracts.

To have wide-spread use of static verification, the verification has to be integrated in the normal process of writing programs. This means that the static verifier has to become a part of the IDE. This has been done for example with Spec# as part of VisualStudio [11] and ESC/Java [19] as part of Eclipse [2].

The goal of this thesis is to develop an automatic verifier for Eiffel which can prove existing code without the need of further annotations. The tool, called *EVE Proofs*, is integrated in EVE [4], the Eiffel Verification Environment. To verify an Eiffel program, *EVE Proofs* translates the program to Boogie code [22]. The Boogie code is checked by a fully automatic theorem prover [14, 1]. The outcome of the verification is then presented in EVE in a well-arranged way. The user of *EVE Proofs* does not need any knowledge about static verification to use the tool.

Besides basic instructions, *EVE Proofs* handles agents and dynamic invocation. To reason about framing, we have implemented an automatic extraction of modifies clauses. The idea is based on separation logic [32] where frame conditions are extracted from the precondition. Our automatic extraction uses the postcondition to retrieve a list of locations which build up the modifies clause. This approach allows us to prove existing Eiffel code without extending the language with modifies clauses. Although the approach is limited, as all locations which are modified need to be explicitly stated, the automatic extraction proved to be useful for our examples. To handle agents, we have implemented the methodology proposed by Nordio *et al.* [27]. With this method, we are able to prove the *formatter* and *archiver* which are proposed as verification challenges by Leavens, Leino and Müller [21]. Also, we have proven an implementation of the command pattern [20] which uses agents. We also extended the methodology to handle framing for agents. The approach for agents now takes all aspects of the routine specification — pre-, post-, and frame condition — into account. For dynamic invocation, we introduced a methodology which allows the prover to use the dynamic type rather than the static type to evaluate the pre- and postcondition. The frame condition is not yet handled by the method for dynamic invocation. With this methodology, we can prove the strategy pattern.

The translation of Eiffel is not complete yet. Important concepts are still missing. These are generics, exception handling, and expanded types. In addition, some instructions are not yet supported, for example object tests. The translation of the missing language constructs is part of future work.

Outline. In Chapter 2, we explain the methodology used for the verification. Chapter 3 describes the translation from Eiffel to Boogie. The user interface of *EVE Proofs* is then presented in Chapter 4. In Chapter 5, we give an overview of the implementation of the proof engine and the integration in EVE. Chapter 6 shows the examples which we use to evaluate our tool. Finally, Chapter 7 concludes the thesis by summarizing the results, listing possible future work, and showing related work.

Chapter 2

Verification Methodology

In the first two sections of this chapter, we present an overview of the verification process and show the basic encodings of routines in Boogie. The later sections describe the methodology for dynamic invocation, agents, and automatic extraction of modifies clauses. The last section describes an approach we implemented to infer pure method marks.

2.1 Overview

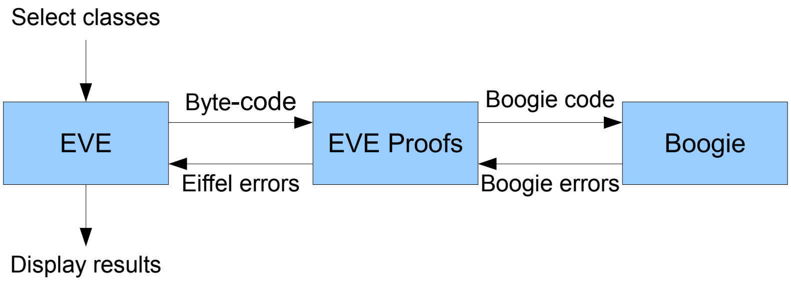


Figure 2.1: Proof Process

The proof process in *EVE Proofs* is very similar to Spec# [16]. Figure 2.1 shows a schema of the general workflow. We start by selecting the classes we want to prove. *EVE Proofs* translates these classes to Boogie code by using the Eiffel byte-code representation. The byte-code is an intermediate representation generated by the EiffelStudio compiler [5]. As the byte-code is only generated for classes which have no compilation errors, this guarantees that the classes we prove contain only valid Eiffel code. Another advantage is that the byte-code representation is simpler than the original AST representation, as for example different ways of writing a creation expression are represented by a single byte-code node. After the translation of the byte-code to Boogie code, *EVE Proofs* runs the Boogie verifier [14] on the generated Boogie code file. To interpret the output of the Boogie verifier, the Boogie code is annotated with additional information to trace the errors reported by Boogie back to the Eiffel program. After the verifier is finished, the results of the verification are presented in a separate tool in EVE.

To prove large systems, we need a modular verification. This means that different parts of the system can be proven individually, and when these parts are put together, the whole system is still correct. For an object-oriented programming language, the proofs are done per routine. Each routine is proven individually, and if all routines are correct, then the whole system is correct. This makes the proofs smaller and reduces the work for the verifier. Another advantage is, that we only have to prove a library once. The clients of the library can just use it and only have to prove their own code.

Modular verification has an important implication: when we prove an individual routine, we assume that all other routines are correct. This allows us to use the specification of called routines without looking at their implementations. As we can have dynamic invocation, the actual routine which is called depends on the dynamic type of the target. The system has therefore to guarantee that each redefined feature adheres to the specification of its origin. For pre- and postconditions, this is already enforced in Eiffel by only allowing weakening of preconditions and strengthening of postconditions.

2.2 Basic Encodings

2.2.1 Routine Specification

The specification of a routine consists of three elements: the precondition, the postcondition, and the frame condition. Eiffel only allows to write the pre- and postcondition in the program. The frame condition can be expressed by a *modifies* clause, which defines the effect a routine can have on the system. In Eiffel, this is not directly represented in the code. In Section 2.5 we present an automatic extraction of modifies clauses. Using this automatic extraction, we can assume in the encoding that we have a modifies clause.

The precondition has to hold prior to the execution of the routine and it is the obligation of the caller to ensure this property. The postcondition has to hold after the execution of the routine body. The routine implementation has to establish the postcondition. The frame condition defines the effect of a routine on the global system. This means that the frame condition denotes which values of the global system may have changed (see Section 2.5 for more information).

Since we are doing modular proofs on the routine level, the routine specification is a key element in the verification. In Boogie, the procedure specification is composed of three elements:

- A list of *require* statements. These conditions have to hold prior to executing the routine. It is the obligation of the client to make sure these conditions hold.

- A list of *ensure* statements. These conditions have to hold after the execution of the routine. It is the obligation of the routine implementation to make sure these conditions hold.
- A list of *modifies* statements. These statements are used to state the effect of the routine call on global variables.

When we encode an Eiffel routine, we use these statements to model the routine contract. The precondition of the routine is encoded as a *require* statement and the postcondition as an *ensure* statement. The third aspect of the routine specification, the frame condition, is also modeled as an *ensure* statement. We are not using the *modifies* statement for this since it is not powerful enough to model abstract modifies clauses as we need them for example for agent calls (see Section 2.4).

Concretely, an Eiffel routine r is encoded as:

$$\begin{array}{l} procedure\ r \\ \quad requires\ precondition_r \\ \quad ensures\ postcondition_r \\ \quad ensures\ frame\_condition_r \end{array}$$

2.2.2 Routine Calls

To encode a call to a routine, we use the procedure call of Boogie. This procedure call is defined in the following way:

Let p be a procedure with a require statement $require_p$ and ensure statement $ensure_p$. When p is called, the require statement has to hold. The require statement is checked by asserting it. When the require statement holds, the ensure statement is guaranteed to hold after the call. Therefore, after the call the ensure statement is assumed to hold.

$$call\ p$$

is therefore interpreted by Boogie as

$$\begin{array}{l} assert\ require_p \\ assume\ ensure_p \end{array}$$

Note that we model the frame condition as part of the ensure statements and is therefore also assumed to hold after the procedure call. Given the encoding of Eiffel routines from the previous section, the call to a routine r is interpreted by Boogie as

$$\begin{array}{l} assert\ precondition_r \\ assume\ postcondition_r \\ assume\ frame\_condition_r \end{array}$$

Listing 2.1: *Eiffel*: Dynamic invocation example

```
deferred class OPERATOR_STRATEGY

  execute (a, b: INTEGER)
    require
      a >= 0 and b >= 0
    deferred
    end

  last_result: INTEGER

end


class ADDITION_STRATEGY inherit OPERATOR_STRATEGY

  execute (a, b: INTEGER)
    require else
      True
    do
      last_result := a + b
    ensure then
      last_result = a + b
    end

end


class CLIENT

  main
    local
      o: OPERATOR_STRATEGY
    do
      create {ADDITION_STRATEGY}o
      o.execute (1, 2)
      check o.last_result = 3 end
    end

end
```

2.3 Dynamic Invocation

Eiffel allows redefined routines to change the contract by weakening the precondition and strengthening the postcondition. When a routine is called, the contract therefore depends on the dynamic type and not just the static type. To prove a program, it may be necessary to use this information and evaluate the contract of the dynamic type.

Listing 2.1 shows an example where the problem is to prove the check instruction on line 36. This program cannot be proven by Spec# for example, as it only considers the static type of feature calls. The static contract of the routine `execute` does not specify any postcondition, as the target of the routine has the deferred type `OPERATOR_STRATEGY`. The dynamic type of the target is `ADDITION_STRATEGY` though, which redefines the routine `execute` and strengthens the postcondition. We now introduce a methodology which is using the dynamic type to evaluate the pre- and postcondition of routine calls.

Abstract Predicates. To prove the example from Listing 2.1, we introduce abstract predicates for the pre- and postcondition of all routines. In particular, for the routine `execute` we introduce the abstract predicates

$$
\begin{array}{l}
pre_{execute}(h_1, o, a, b) \\
post_{execute}(h_1, h_2, o, a, b)
\end{array}
$$

Where the arguments are the heaps h_1 and h_2, the target object of the call o, and the arguments a and b. The heap h_1 denotes the state before the execution of the routine and the heap h_2 the state after the execution. The postcondition predicate takes both heaps as an argument to evaluate `old` expressions. These abstract predicates are now used for the specification of the routine `execute`.

$$
\begin{array}{l}
procedure\ execute(o, a, b) \\
\quad requires\ pre_{execute}(Heap, o, a, b) \\
\quad ensures\ post_{execute}(old(Heap), Heap, o, a, b)
\end{array}
$$

Where o is the target of the routine call and a and b are the arguments of the call. The variable $Heap$ denotes the global heap variable. In the postcondition, we use $old(Heap)$ to access the heap prior to the execution of the routine.

To reason about the abstract predicates, we need to link those predicates to the actual pre- and postcondition of the routine. As we want to verify dynamic invocation, the dynamic type of the target object plays an important role in setting up the connection between the abstract predicates and the concrete contracts. This connection is done by introducing axioms in the theory.

Postcondition. For the postcondition, the axioms state that the abstract postcondition predicate implies the concrete postcondition, depending on the dynamic type of the object. For the routine `execute` the following axioms are generated:

$$\forall h_1, h_2, o, a, b : type(o) <: OPERATOR\_STRATEGY \implies \big(post_{execute}(h_1, h_2, o, a, b) \implies true\big) \tag{2.1}$$

$$\forall h_1, h_2, o, a, b : type(o) <: ADDITION\_STRATEGY \implies \big(post_{execute}(h_1, h_2, o, a, b) \implies true \wedge h_2[o, last\_result] = a + b\big) \tag{2.2}$$

Where $<:$ is the conformance operator and $h_2[o, last\_result]$ is the value of the attribute `last_result` of the object o. Axiom 2.1 is for the routine `execute` of the parent type. As the routine does not have a postcondition, the postcondition predicate just implies *true*. Axiom 2.2 is for the subtype `ADDITION_STRATEGY` and expresses that if the type of the target conforms to `ADDITION_STRATEGY`, then the postcondition predicate implies the combined postcondition of the redefined routine.

Precondition. The axioms for the preconditions use the abstract predicate on the right-hand side of the implication: the axioms state that the concrete precondition implies the precondition predicate, again depending on the dynamic type of the object. For the routine `execute` the following axioms are generated:

$$\forall h_1, o, a, b : type(o) <: OPERATOR\_STRATEGY \implies \big((a > 0 \wedge b > 0) \implies pre_{execute}(h_1, o, a, b)\big) \tag{2.3}$$

$$\forall h_1, o, a, b : type(o) <: ADDITION\_STRATEGY \implies \big(((a > 0 \wedge b > 0) \vee (true)) \implies pre_{execute}(h_1, o, a, b)\big) \tag{2.4}$$

The first axiom is again for the parent type. If the target object conforms to the type `OPERATOR_STRATEGY`, then the actual precondition of the routine `execute` implies the precondition predicate. Axiom 2.4 is for the type `ADDITION_STRATEGY`. For this type, the combined precondition is used for the implication. The axiom expresses that when the type conforms to `ADDITION_STRATEGY` and the combined precondition is satisfied, then the precondition predicate holds.

Application. We can now go back to the example of Listing 2.1. Using the introduced encoding of routine calls from Section 2.2.2 and the abstract predicates, the call on line 35 is interpreted by Boogie as:

$$\begin{array}{l} assert\ pre_{execute}(Heap, o, 1, 2) \\ assume\ post_{execute}(old(Heap), Heap, o, 1, 2) \end{array}$$

To prove the assertion about the precondition predicate, we use the introduced axioms. The type of the object o is `ADDITION_STRATEGY`, thus $type(o)$ conforms to both `OPERATOR_STRATEGY` and `ADDITION_STRATEGY`. Using axiom 2.3 and the information that the arguments are both greater than zero, we can deduce that the precondition predicate holds. As the precondition holds, we can assume that after the call the postcondition predicate holds. To prove the check instruction on line 36 of the example in Listing 2.1, we have to prove that the `last_result` attribute has the value 3. We cannot use axiom 2.1 for this, as the contract of the static type is not strong enough. As the type of the target object is `ADDITION_STRATEGY` though, we can use axiom 2.2. This axiom implies that the value of `last_result` has been set to the addition of the two arguments, which in this case is 3. Therefore we can prove the check instruction.

By introducing this methodology, we are now able to use the pre- and postcondition of dynamic types for the verification. Also, this methodology is modular. Although new axioms are generated which express properties about existing predicates, these axioms do not change the correctness of existing code. All introduced axioms respect the behavioral subtyping and only weaken preconditions or strengthen postconditions. If an existing class has been proven, then the introduction of these new axioms does not change the correctness of the proof.

The frame condition in the context of dynamic invocation is not handled with the current methodology. The extension of the method to handle frame conditions is future work (see Section 7.2).

2.4 Agents

Programs using agents pose two challenges to static verification. First, we need to be able to specify routines that use agents. Second, we need to be able to verify the call to an agent. To tackle these two problems, we use the methodology proposed by Nordio *et al.* [27]. The approach uses abstract predicates which express the pre- and postcondition of an agent to specify routines which use an agent. To verify calls to agents, the methodology introduces assumptions to link these abstract predicates to the concrete contracts of the routine used to create an agent. The example in Listing 2.2 is used to illustrate the method.

Agent Call Specification. The class `CLIENT` declares a routine `apply_agent` which takes an agent and an integer as argument and calls the agent with the given integer. For the call to the agent to be valid, the precondition has to be satisfied with the given integer argument. The problem is, that we cannot know what the precondition of the agent is, as we do not know which agent is passed at runtime. To make sure that the call to the agent is valid, the caller of the routine `apply_agent` has to make sure the argument is valid for the agent.

In Eiffel, the class `PROCEDURE` has a query `precondition` which denotes the abstract precondition of the agent. We can use this query to specify the precondition of `apply_agent` (Listing 2.2, line 43). The meaning of this precondition is, that the caller of `apply_agent` has to make sure that the precondition of the agent

Listing 2.2: *Eiffel*: Agent example

```
class ACCOUNT

  make (a_initial_amount: INTEGER)
    require
      a_initial_amount >= 0
    do
      balance := a_initial_amount
    ensure
      balance = a_initial_amount
    end

  balance: INTEGER

  deposit (amount: INTEGER)
    require
      amount >= 0
    do
      balance := balance + amount
    ensure
      balance = old balance + amount
    end

end


class CLIENT

  main
    local
      a: ACCOUNT
      proc: PROCEDURE [ANY, TUPLE [INTEGER]]
    do
      create a.make (0)
      proc := agent a.deposit

      apply_agent (proc, 70)

      check a.balance = 70 end
    end

  apply_agent (proc: PROCEDURE [ANY, TUPLE [INTEGER]], value
      )
    require
      proc.precondition ([value])
    do
      proc.call ([value])
    ensure
      proc.postcondition ([value])
    end

end
```

is satisfied. Applying the same principle, we can use the routine `postcondition` of the agent to specify the postcondition of the routine `apply_agent`.

To use these queries in the code which calls the agent, the routine `call` of the agent has to use these queries as well. Therefore, the routine `call` needs the following signature:

Listing 2.3: *Eiffel*: Signature of call

```
call (value: TUPLE [INTEGER])
  require
    precondition (value)
  ensure
    postcondition (value)
```

To reason about the specification, the methodology replaces these queries with abstract predicates which model the pre- and postcondition of the agent. These abstract predicates take the agent, the arguments and the heap as an argument and yield the evaluation of the concrete contract. The definition of the predicates is as follows:

$$\begin{aligned}&\mathit{precondition}(a, h_1, \mathit{arg})\\&\mathit{postcondition}(a, h_1, h_2, \mathit{arg})\end{aligned}$$

Where a stands for the agent object, h_1 the heap in the prestate of the call, h_2 the heap in the poststate of the call, and arg the argument to the agent. The postcondition predicate needs both heaps to evaluate old expressions.

With these predicates, we can now prove the routine `apply_agent`. The precondition of the routine (Listing 2.2, line 43) is translated to the abstract precondition predicate $\mathit{precondition}(\mathit{proc}, \mathit{Heap}, \mathit{value})$, where $Heap$ denotes the current heap. To verify the call to the agent, we have to establish the precondition of `call`. This precondition is translated to the abstract precondition predicate as well, specifically to $\mathit{precondition}(\mathit{Current}, \mathit{Heap}, \mathit{value})$. As the target to the call is *proc* and the remaining arguments are the same, both predicates are equal. Therefore the precondition of `call` is satisfied. Conversely, after the call to `call`, the predicate $\mathit{postcondition}(\mathit{Current}, \mathit{old}(\mathit{Heap}), \mathit{Heap}, \mathit{value})$ denoting the postcondition of the routine `call` holds. Using this predicate, the verifier can prove that the postcondition of `apply_agent` on line 47 in Listing 2.2 holds.

Agent Creation. We have specified the routine `apply_agent` using abstract predicates for pre- and postconditions. In the routine `main` of Listing 2.2, we call `apply_agent` with an agent created from the routine `deposit`. To use the specification of `apply_agent`, we need to link the abstract predicates used in the specification to the concrete contracts from the routine `deposit`. We do this by introducing *assumptions* when the agent *proc* is created on line 34:

$$assume\ \forall h_1, arg : precondition(proc, h_1, arg) = pre_{deposit}(h_1, arg)$$
$$assume\ \forall h_1, h_2, arg : postcondition(proc, h_1, h_2, arg) = post_{deposit}(h_1, h_2, arg)$$

Where $pre_{deposit}$ denotes the concrete precondition of `deposit` and $post_{deposit}$ denotes the concrete postcondition of `deposit`. These assumptions are now used to prove the check instruction on line 38. The precondition of the `apply_agent` routine uses the abstract precondition predicate. With the first assumption that is generated, we know that this abstract predicate denotes the precondition of `deposit`, which we see is satisfied. Therefore, as the precondition is established, we can assume the postcondition holds. The postcondition of `apply_agent` uses the abstract postcondition predicate. The second assumption implies that this postcondition predicate is equivalent to the postcondition of `deposit`. We can now use the postcondition of deposit to prove the check instruction.

Modifies Clauses. To define the effect of an agent, we have to reason about the frame condition of the agent. For this we introduce again an abstract predicate:

$$modifies(a, h_1, arg, o, f)$$

The predicate takes the agent a, the heap h_1 denoting the prestate, the arguments, and additionally an object o and field f. The predicate yields true if the agent a modifies the field f of the object o when the agent is called with the given argument. This predicate is used to encode the frame condition of the call to the agent. The specification of the `call` routine of the agent is the following:

```
$procedure\ call(a, arg)$
    $requires\ precondition(a, Heap, arg)$
    $ensures\ postcondition(a, Heap, old(Heap), arg)$
    $ensures\ \forall o, f : not\ modifies(a, Heap, arg, o, f)$
                                 $\Longrightarrow Heap[o, f] = old(Heap)[o, f]$
```

The last line encodes the frame condition. The frame condition expresses, that all fields on the heap which are not modified by the agent are unchanged. To use the abstract predicate, we add an additional assumption when an agent is created which links the abstract modifies predicate to the concrete modifies clause of the routine. For the agent creation on line 34 in Listing 2.2, we create:

$$assume\ \forall h_1, arg, o, f : modifies(a, h_1, arg, o, f) = modifies_{deposit}(h_1, arg, o, f)$$

Where $modifies_{deposit}$ expresses if the routine `deposit` modifies the field f of the object o. Using this assumption and the encoding of the `call` routine, we can reason about the framing of agents.

2.5 Automatic Extraction of Modifies Clauses

The frame condition plays an important role in modular verification. The frame condition denotes the effect of a routine on the system. This means that the frame condition expresses which part of the heap is potentially modified by the call of a routine. When a routine is called, the verifier is invalidating all knowledge about the locations which may have changed. Therefore it is essential to constrain the effect a routine has on the system to preserve as much information as possible.

Spec# [16] and JML [17] have introduced *modifies* clauses to specify the frame condition. The programmer can list the attributes in the modifies clause which are possibly changed by the routine. Eiffel does not offer a way to specify the frame condition. Therefore, the verifier has to assume that the routine may have changed everything. With this approach, the possibility for static verification is very limited as all information is lost after every routine call.

The example in Listing 2.4 illustrates this problem. We have to prove the check instruction on line 31. Using `set_hour`, the attribute `hour` is set to 17. Nevertheless, the check instruction only holds if the routine `set_minute` does not modify the attribute `hour`. The postcondition of `set_minute` only defines what happens to the attribute `minute`. Since we do not have a modifies clause to express the frame condition, the verifier has to assume that the value of `hour` changed. Therefore, the verifier cannot prove the check instruction.

Automatic Extraction. To remedy this situation, we have introduced an automatic extraction of modifies clauses. Our approach uses the postcondition to extract a list of locations which constitute the modifies clause. The postcondition lists the conditions which the routine established. The idea is, that these conditions also mention the locations which are needed to express the postcondition. We then take all the locations which are used in these expressions and construct the modifies clause with these locations. The advantage of this method is, that the language does not have to be extended and that the method can be used with existing code.

We use the person class from Listing 2.5 to illustrate the automatic extraction of modifies clause. To extract the modifies clause from routine `marry`, we look at the postcondition. We need to take all locations from this postcondition. On the first line — `spouse = a_other` — there are the two locations `spouse` and `a_other`. We can ignore the argument, as arguments are local to the routine and cannot be changed. The attribute `spouse` on the other hand is added to the list of locations. The second postcondition is `a_other.spouse = Current`. This postcondition has three locations: `a_other`, `a_other.spouse`, and `Current`. The argument `a_other` and the reference to `Current` can both be ignored as they are again local to the routine. From the second postcondition we thus extract the location `a_other.spouse`. The extracted modifies clause of the routine `marry` is formed by the locations `spouse` and `a_other.spouse`.

With this method, we can look again at the example from Listing 2.4. To prove the check instruction, we need to know if the routine `set_minute` changes the attribute `hour`. When we apply the automatic modifies clause extraction

Listing 2.4: *Eiffel*: Frame example

```
class TIME

  hour, minute: INTEGER

  set_hour (a_value: INTEGER)
    do
      hour := a_value
    ensure
      hour = a_value
    end

  set_minute (a_value: INTEGER)
    do
      minute := a_value
    ensure
      minute = a_value
    end

end


class CLIENT

  main
    local
      t: TIME
    do
      create t
      t.set_hour (17)
      t.set_minute (40)
      check t.hour = 17 end
    end

end
```

Listing 2.5: *Eiffel*: Person class

```
class PERSON

  spouse: PERSON

  marry (a_other: PERSON)
    do
      -- Implementation omitted
    ensure
      spouse = a_other
      a_other.spouse = Current
    end

end
```

Listing 2.6: *Eiffel*: Frame inheritance

```
class PARENT

  a, b: INTEGER

  foo
    do
      -- Implementation omitted
    ensure
      a > 0
    end

end


class CHILD inherit PARENT

  c: INTEGER

  foo
    do
      -- Implementation omitted
    ensure then
      c > b
    end

end
```

to the routine `set_minute`, we see that the modifies clause of `set_minute` contains only the attribute `minute`. Therefore, the verifier can deduce that the call to `set_minute` does not change the attribute `hour` and can prove the check instruction.

Inheritance. In the context of inheritance, the automatic generation has to maintain behavioral subtyping. This means for frame conditions, that a feature in the child is not allowed to change locations from the parent which were not mentioned as being modified by the parent routine. Thus, the list of modified locations can only be extended with attributes introduced by the child class.

This is illustrated with the example from Listing 2.6. In the parent class, the extracted modifies clause of `foo` consists only of the attribute `a`. The class `CHILD` redefines this routine and adds an additional postcondition. Although the additional postcondition lists the two attributes `b` and `c`, only the attribute `c` introduced in the child class is added to the modifies clause. If we would allow the class `CHILD` to modify `b` in the redefined routine, than the routine would violate the frame condition of the parent, thus violating behavioral subtyping.

Listing 2.7: *Eiffel*: Overapproximation

```
   class TIME
 2
     hour, minute: INTEGER
 4
     copy_time (a_other: TIME)
 6     do
         hour := a_other.hour
 8       minute := a_other.minute
       ensure
10       hour = a_other.hour
         minute = a_other.minute
12
         a_other.hour = old a_other.hour
14       a_other.minute = old a_other.minute
       end
16
   end
```

Overapproximation. A drawback of the automatic extraction of modifies clauses is overapproximation. Each location in the postcondition is used for the modifies clause, even if the programmer does not want to change the location. If this situation occurs, one has to add an additional postcondition to state that a value does not change. An example for this is shown in Listing 2.7. The `copy_time` routine does not change any fields of the argument. Nevertheless, the automatic modifies clause extraction adds the location `a_other.hour` and `a_other.minute` to the extracted modifies clause. To assure that these locations were not changed, the two postcondition on line 13 and 14 have to be added.

Encoding. In the translation to Boogie, we are adding the frame condition as part of the postcondition for each routine. Basically, this postcondition expresses that locations on the heap which are not modified by the routine retain their value after the execution of the routine. The formalization is defined as follows:

Let r be a routine, $modifies(r)$ the set of locations modified by r, $Heap$ the set of all locations on the heap, $value(l)$ the value of the location l after executing r and $old(value(l))$ the value of the location l before executing r. Then the frame condition is expressed by:

$$\forall l \in Heap : l \notin modifies(r) \implies value(l) = old(value(l)) \tag{2.5}$$

The formula expresses that all locations not modified by the routine keep their old value. There is a special case if the set of modified locations of a routine is empty. This is the case for side-effect free routines, and this simplifies the formula to:

$$\forall l \in Heap : value(l) = old(value(l))$$

The formula 2.5 is used in the translation to Boogie in Section 3.5.5.

Limitations. The automatic extraction has two limitation: one cannot specify whole object structures which changed and dynamic invocation is not handled. A routine to sort an array would need a modifies clause which lists all locations of the array. In Spec#, this can be done using a special notation [16]. Extending the automatic extraction to handle object structures and dynamic invocation is future work (see Section 7.2).

2.6 Pure Inference

Pure (or side-effect free) routines play a special role in the specification. The execution of the contract of a class must not have any side effects, and therefore only side-effect free routines are allowed as part of the specification. Eiffel does not support pure routines as for example Spec# [16], and therefore this rule is not enforced by the compiler. The verifier is thus trying to infer which routines are side-effect free from the available information. When we know which routines are pure, we can introduce a functional representation for them. The functional form of the routine is necessary to translate the routine specification, i.e. pre- and postcondition and the class invariant to Boogie. The inference mechanism for pure marks allows us to prove existing code which has not been annotated with pure marks.

The automatic inference uses the fact that only pure features are allowed in contracts: All features which are used in a contract are assumed to be side-effect free. Thus, if a feature is used in a precondition, postcondition or class invariant, it is implicitly marked as pure. If a marked routine is being proven, then the prover checks if the routine is side-effect free. As we are doing modular verification, we are not proving routines which are merely referenced by checked code, even if we add a pure mark to a referenced routine.

The problem of this approach is that it is not modular. A feature is only marked as side-effect free when it is used as part of a contract. This does not have to be in the same class or even the same library. During the proof of a class one of its features may not be marked as pure, but together with another system it may be.

The example in Listing 2.8 illustrates the problem. When we prove a class in a modular approach, we assume that all used features from other classes are correct. Therefore, when only class `B` is verified, class `A` is assumed to be correct. As the feature `foo` of class `A` is used in a contract of `B`, it is marked as side-effect free. Looking at the feature it is clear that it does have side-effects, thus violating the inferred property. The violation will only be noticed when both classes are proven together. Only then will the verifier check if the implicitly marked pure feature is really side-effect free.

Listing 2.8: *Eiffel*: Pure Marking Problem

```
class A

feature

  foo: BOOLEAN
    do
      i := i + 1
      Result := i > 10
    end

  i: INTEGER

end


class B

feature

  test (a: A)
    require
      a.foo
    do
    end

end
```

Chapter 3

Translation to Boogie

This chapter describes the translation of Eiffel to Boogie. We first give an introduction to Boogie, show the naming convention used and explain the background theory. The rest of the chapter explains how Eiffel code is translated by showing examples of the translation.

3.1 Introduction to Boogie

This section presents an overview of Boogie. For more details, see [22].

Identifiers. Boogie allows the special characters . (dot), # (hash sign) and $ (dollar sign) in identifier names (among others). Names are case sensitive. The namespace for functions, procedures, and constants is global. Arguments, local variables, and bound variables in forall statements are locally scoped.

Types. There are two built-in types: booleans (`bool`) and integers (`int`) with the usual operators. Additional types can be defined by the user. For example, we can define a type `car` as follows:

Listing 3.1: *Boogie*: Type Example

```
type car;
```

Functions. Function declarations define mathematical functions. The properties of these functions have to be expressed by axioms. The following code segments defines a function which denotes the maximum speed of a car:

Listing 3.2: *Boogie*: Function Example

```
function max_speed(car) returns (int);
```

Variables. To define the state of the program, you can define mutable global variables. These variables can also be one- or two-dimensional arrays. We can define a global array which holds the current speed for all cars:

Listing 3.3: *Boogie*: Variable Example

```
// Array storing the speed for cars
var Speed: [car]int;
```

Axioms. An axiom can define any property using functions, global variables and quantifiers. With this, you can state facts about the system. For example, we can define an axiom to express that the speed of all cars is less or equal to their maximum speed:

Listing 3.4: *Boogie*: Axiom Example

```
// Speed of all cars is less or equal to their maximal speed
axiom (forall c: car :: Speed[c] <= max_speed(c));
```

Procedures. A procedure is basically the signature of a feature. It defines the arguments, result type, pre- and postcondition, and the frame condition. We can define a procedure `accelerate` which takes a car and increases the speed as follows:

Listing 3.5: *Boogie*: Procedure Example

```
// Accelerate a car
procedure accelerate(arg: car);
  // Not yet at maximal speed
  requires Speed[arg] < max_speed(arg);
  // Speed increased
  ensures Speed[arg] > old(Speed[arg]);
  // Speed of argument was modified
  modifies Speed[arg];
```

Implementations. The implementation is separate from the signature definition. If present, Boogie tries to prove that the implementation adheres to the specification of the procedure definition. A possible implementation for the previously declared procedure `accelerate` is shown in the following example:

Listing 3.6: *Boogie*: Implementation Example

```
implementation accelerate(arg: car)
{
  Speed[arg] := Speed[arg] + 1;
}
```

Listing 3.7: *Eiffel*: Naming example

```
class LIST

create
  make

feature

  make
    do end

  count: INTEGER

  is_empty: BOOLEAN
    do end

  extend (a_item: ANY)
    do end

end
```

3.2 Naming conventions

Boogie has a global namespace for functions, procedures and global values. To have unique identifiers, we use a naming convention for the translation. The example from Listing 3.7 is used to illustrate the conventions:

- **Attributes**: The translation of attribute names is `field.CLASS_NAME.attribute_name`. The name for the attribute `count` is therefore `field.LIST.count`.
- **Routines**: All routines are translated to `proc.CLASS_NAME.routine_name`. The routine `extend` of the example is translated to `proc.LIST.extend`.
- **Creation routines**: For creation routines, the translation is `create.CLASS_NAME.routine_name`. The name used in Boogie for the creation routine `make` is `create.LIST.make`.
- **Functional representation**: Side-effect free routines have a functional representation which is created as `fun.CLASS_NAME.routine_name`. Since the routine `is_empty` of the example class is pure, a functional representation is generated with the name `fun.LIST.is_empty`.
- **Precondition predicates**: The precondition predicates for each routine are generated as `precondition.CLASS_NAME.routine_name`. The precondition predicate of the routine `extend` is therefore `precondition.LIST.extend` .
- **Postcondition predicates**: The postcondition predicates for each routine are generated as `postcondition.CLASS.routine`. The postcondition predicate used for the routine `extend` is `postcondition.LIST.extend`.

Listing 3.8: *Boogie*: Reference types

```
// Type definition for reference types
type ref;

// Constant for Void reference
const Void: ref;
```

- **Heap variable**: The global heap uses the name `Heap` starting with an uppercase letter. If the heap is used as a bound variable in a quantifier expression, it starts with a lowercase letter (`heap`).

3.3 Background Theory

The background theory is a Boogie code file which states the global properties for all Eiffel systems. It declares the user-defined types used to represent references, fields and types. It also defines global variables, convenience functions and axioms expressing basic properties of the system, such as the heap model. In each proof, we include the background theory file in the generated Boogie file. The background theory is based on Ballet's [33] theory and has been adapted and extended to handle Eiffel types and agents.

3.3.1 Basic Definitions

Reference Types. Boogie has only the built-in support for the basic types `bool` and `int`. In addition, it allows to have user-defined types. As we are using an object-oriented programming language, we need a reference type. Listing 3.8 shows the definition of the user defined type for references, as well as the constant denoting void references.

Heap Model. An important part of the background theory is the heap model. The code for this is shown in Listing 3.9.

To help with type safety, Boogie supports a special generic type. This is used for the type of fields, which is defined as `type Field _`. The underscore stands for the generic content type. When an Eiffel attribute is translated to Boogie, the Eiffel type of the attribute is used to instantiate the Boogie field type. Boogie also supports two-dimensional maps. This is used to model the heap. The type definition for the heap type is a generic two-dimensional map, where the content type of the map is generic. The heap type instantiates its generic content type using the previously defined field type. That way, the content type of the heap is of the same type as a field which is accessed. This means that unlike generic containers in Eiffel, the content type of the heap changes with the field argument.

The global value `Heap` is an entity of the previously defined heap type. It takes an object reference and a field name and returns the value stored on the

Listing 3.9: *Boogie*: Heap model

```
// Type for fields (with open subtype)
type Field _;

// Type for heap (with generic content type)
type HeapType = <beta>[ref, Field beta]beta;

// Function which defines a heap
function IsHeap(heap: HeapType) returns (bool);

// The global heap
var Heap: HeapType where IsHeap(Heap);

// Allocated ghost field
const $allocated: Field bool;

// Function to check if an object is allocated
function IsAllocated(heap: HeapType, o: ref) returns (bool);
axiom (forall heap: HeapType, o: ref ::
  IsAllocated(heap, o) <==> heap[o, $allocated]);

// Function to check if an object is allocated and not void
function IsAllocatedAndNotVoid(heap: HeapType, o: ref)
  returns (bool);
axiom (forall heap: HeapType, o: ref ::
  IsHeap(heap) ==> (IsAllocatedAndNotVoid(heap, o) <==>
    o != Void && IsAllocated(heap, o)));
```

heap. To define valid objects, a ghost field `$allocated` is added. This ghost field denotes if an object reference is allocated on the heap. If so, the corresponding heap value `Heap[$o, $allocated]` is set to true for an object `$o`. In addition, there are the convenience functions `IsAllocated` and `IsAllocatedAndNotVoid` which are used in the translation for clarity.

3.3.2 Typing

The type hierarchy is also represented in Boogie. Listing 3.10 shows the relevant section of the background theory. A user-defined type `Type` is declared which represents an Eiffel type. Also, the background theory defines the two convenience functions `IsAttachedType` and `IsDetachedType` which are used in the translation to declare the type of objects.

To define the subtype relationship between types, the partial-order relationship `<:` of Boogie is used. For all Eiffel types, a constant of type `Type` is defined and the inheritance relationship is expressed with the order relation.

Listing 3.10: *Boogie*: Typing

```
// Type definition for Eiffel types
type Type;

// Type field (a ghost field)
const unique $type: Field Type;

// Basic types
const unique ANY: Type;

// Function to declare attached types
function IsAttachedType(heap: HeapType, o: ref, t: Type)
    returns (bool);

// Function to declare detached types
function IsDetachedType(heap: HeapType, o: ref, t: Type)
    returns (bool);

// Axiom to state property of attached types
axiom (forall heap: HeapType, $o: ref, $t: Type ::
  IsAttachedType(heap, $o, $t) ==> ($o != Void &&
      IsAllocated(heap, $o) && heap[$o, $type] <: $t));

// Axiom to state property of detached types
axiom (forall heap: HeapType, $o: ref, $t: Type ::
  IsDetachedType(heap, $o, $t) ==> ($o != Void ==> (
      IsAllocated(heap, $o) && heap[$o, $type] <: $t)));
```

Listing 3.11: *Boogie*: Agent theory

```
// Precondition predicate
function routine.precondition_0 (heap: HeapType, agent: ref)
  returns (bool);

// Postcondition predicate
function routine.postcondition_0 (heap: HeapType,
              old_heap: HeapType, agent: ref)
  returns (bool);

// Frame condition predicate
function agent.modifies_0<alpha> (heap: HeapType,
   old_heap: HeapType, agent: ref, $o: ref, $f: Field alpha)
  returns (bool);

// Calling of agent
procedure routine.call_0 (
       Current: ref where IsAttachedType(Heap, Current, ANY)
  );
  requires routine.precondition_0(Heap, Current);
  modifies Heap;
  ensures (forall<alpha> $o: ref, $f: Field alpha ::
     ($o != Void && old(Heap)[$o, $allocated] &&
       !agent.modifies_0(Heap, old(Heap), Current, $o, $f))
       ==> (old(Heap)[$o, $f] == Heap[$o, $f]));
  ensures routine.postcondition_0(Heap, old(Heap), Current);
```

3.3.3 Agent Theory

To encode the agent methodology presented in Section 2.4, we define several helper functions in the background theory. The functions in Listing 3.11 are used for agents without arguments. The number in the suffix denote how many open arguments the agent has. There are similar functions in the background theory file which are used for agents with more arguments.

The functions define the abstract pre- and postcondition predicates, as well as an abstract modifies predicate. The pre- and postcondition predicates denote the require and ensure clause of the routine used to create the agent. To evaluate `old` expressions, the postcondition predicate takes two heaps, one for the prestate and one for the poststate of the routine execution. The modifies predicate takes an arbitrary object and field. The predicate is true, if the field of the object is in the modifies clause of the original routine used to create the agent. These three predicates are used in the specification of the *call* routine of the agent. By using these predicates, we can express that the call to the agent has the same effect as a normal call to the original routine would have. See Section 3.6.5 to see how the predicates are used when an agent is created.

Listing 3.12: *Eiffel*: Person class

```
class PERSON

feature -- Access

  first_name: !STRING

  middle_name: STRING

  year_of_birth: INTEGER

  age: NATURAL

  is_married: BOOLEAN

end
```

Listing 3.13: *Boogie*: Code for basic fields

```
// Attribute name
const unique field.PERSON.is_married: Field bool;

// Attribute name
const unique field.PERSON.year_of_birth: Field int;

// Attribute name
const unique field.PERSON.middle_name: Field ref;
```

3.4 Attributes

The translation for attributes is straightforward. Each attribute of a class is translated to a unique identifier in Boogie which specifies the heap location where the value of the attribute is stored. Special care has to be taken for built in Eiffel types with an additional semantics. This is the case for attached types and natural numbers.

The Eiffel class from Listing 3.12 is used to show the translation for different attribute types.

Basic Attributes. The translation of the basic field types - boolean, integer and reference - is the most simple one. There is just a constant definition for the heap location where the value of this field is stored. The Boogie code in Listing 3.13 shows the translation for the fields `is_married`, `year_of_birth`, and `middle_name` from Listing 3.12. Note that the generic subtype for the heap content type `Field` has been instantiated with the actual field types `bool`, `int`, and `ref`.

Listing 3.14: *Boogie*: Boogie code for natural field

```
// Attribute name
const unique field.PERSON.age: Field int;

// Axiom for type NATURAL
axiom (forall heap: HeapType, $o: ref ::
  IsHeap(heap) && IsAllocatedAndNotVoid(heap, $o) ==>
    heap[$o, field.PERSON.age] >= 0);
```

Listing 3.15: *Boogie*: Code for attached field

```
// Attribute name
const unique field.PERSON.first\_name: Field ref;

// Axiom for attached types
axiom (forall heap: HeapType, $o: ref ::
  IsHeap(heap) && IsAllocatedAndNotVoid(heap, $o) ==>
    IsAllocatedAndNotVoid(heap,
              heap[$o, field.PERSON.first\_name]));
```

Natural Attributes. Fields of type natural have the guaranteed property that they never hold a negative value. As Boogie does not have a specific type for natural values, this property has to be expressed with an additional condition. This can be done with an axiom which states that for all objects on the heap, the heap location for the natural field is always non-negative. The axiom is constrained to only valid heaps and objects. The code in Listing 3.14 shows the translation of the Eiffel attribute `age` from Listing 3.12.

Attached Attributes. Like natural fields, attached fields have an additional property which is guaranteed by the type system. For attached fields, this property states that the field will never point to a void reference. Again, as Boogie does not have a specific type for this, an axiom can be defined which expresses this characteristic. The axiom expresses that for all objects on the heap, the heap location for the attached field is a valid non-void reference. The code in Listing 3.15 shows the translation of the Eiffel attribute `first_name` from Listing 3.12.

3.5 Routine Signature

Since we are doing modular verification on the level of routines, the translation of the routine signature is an essential part of the translation process. We use the example from Listing 3.16 to illustrate the translation of routine signatures. The example shows the interface description of the `make` routine of the class `ACCOUNT`. The Boogie code which is generated for this routine is shown in the Listings 3.17 and 3.18. The following sections explain the generated code.

Listing 3.16: *Eiffel*: Interface of Account

```
class ACCOUNT

create
  make

feature {NONE}

  make (a_initial_amount: INTEGER)
      -- Initialize account with ‘a_initial_amount’.
    require
      a_initial_amount >= 0
    ensure
      balance = a_initial_amount

feature

  -- ... other features

invariant
  balance >= 0

end
```

3.5.1 Procedure Definition

For each Eiffel routine a Boogie procedure is created. The name of the translated routine is generated according to the naming convention (see Section 3.2). The name which is created for the routine `make` from the class `ACCOUNT` is `proc.ACCOUNT.make`. The procedure is defined by using the **`procedure`** keyword (Listing 3.17, line 21). The procedure is followed by the list of arguments and, if the translated routine is a query, the return value. After the signature follows a list of **`requires`**, **`ensures`**, and **`modifies`** statements.

In addition to normal requires and ensures statements, there are *free* requires and ensures. The *free* statement are used by the prover like normal requires and ensures statements, but the free statements are not proven. This means that a **`free requires`** statement is assumed when the procedure implementation is proven, but it is not checked like a precondition when the procedure is called. The **`free ensures`** statement on the other hand is assumed to hold after the call to the procedure, but it is not proven if the implementation establishes this postcondition.

3.5.2 Arguments

To model the object-oriented Eiffel with the procedural Boogie, we add an additional argument to all procedures which denotes the target object of a routine call (Listing 3.17, line 22). This argument is always called `Current`.

Listing 3.17: *Boogie*: Translation of routine *make*

```
// Precondition predicate
function precondition.ACCOUNT.make(heap: HeapType,
  current: ref, arg1: int) returns (bool);

axiom (forall heap: HeapType, current: ref, arg1: int ::
  (arg1 >= 0)
    ==> (precondition.ACCOUNT.make(heap, current)));

// Postcondition predicate
function postcondition.ACCOUNT.make(heap: HeapType,
  old_heap: HeapType, current: ref, arg1: int)
    returns (bool);

axiom (forall heap: HeapType, old_heap: HeapType,
  current: ref, arg1: int ::
  (postcondition.ACCOUNT.make(heap, old_heap,
  current, arg1))
    ==> (heap[current, field.ACCOUNT.balance] == arg1));

// Routine make from class ACCOUNT
procedure proc.ACCOUNT.make(
    Current: ref where IsAttachedType(Heap,Current,ACCOUNT),
    arg1: int
  );
  // Pre- and postcondition predicate
  requires precondition.ACCOUNT.make(Heap, Current, arg1);
  free ensures postcondition.ACCOUNT.make(Heap, old(Heap),
    Current, arg1);

  // Pre- and postcondition
  free requires arg1 >= 0;
  ensures Heap[Current, field.ACCOUNT.balance] == arg1;

  //  Class invariant, routine entry
  free requires Heap[Current, field.ACCOUNT.balance] >= 0;
  // Class invariant, routine exit
  ensures Heap[Current, field.ACCOUNT.balance] >= 0;

  // Frame condition
  modifies Heap;
  ensures (forall<alpha> $o: ref, $f: Field alpha ::
    $o != Void && IsAllocated(old(Heap), $o) &&
    (!($o == Current && $f == field.ACCOUNT.balance))
      ==> (old(Heap)[$o, $f] == Heap[$o, $f]));
```

The arguments to the Eiffel routine are then added as normal arguments. For all reference types, we define the type of the argument in the signature using the typing functions declared in the background theory (see Section 3.3.2).

3.5.3 Pre- and Postconditions

In line with the methodology for dynamic invocation from Section 2.3, we introduce an abstract pre- and postcondition predicate for each routine. For the routine `make`, these are declared on line 2 and line 10 of Listing 3.17. For these predicates, we also define the axioms which define the connection between the abstract predicates and the concrete contracts. The axioms for the pre- and postcondition are declared on line 5 and line 14 of Listing 3.17. We then use these predicates in the specification of the procedure. The precondition predicate is added as a `requires` statement on line 26 and the postcondition predicate as a `free ensures` statement on line 27. In addition to these two predicates, the actual pre- and postcondition of the routine are also added directly to the procedure definition. The precondition is added on line 31 as a `free requires` statement, and the postcondition is added on line 32 as an `ensures` statement.

By adding the precondition predicate as a requires statement, the caller of the procedure has to prove that the predicate holds. On the call site, the caller can use the dynamic type to assert the precondition. With the methodology we introduced, the precondition predicate can hold even if the concrete precondition is violated. This can be the case if the precondition is weaker for the dynamic type. When the verifier is proving the implementation, the requires statement of the procedure are assumed to hold at the beginning of the execution. We add the concrete precondition in the procedure definition as a free requires statement, so that the verifier is using this precondition when the procedure implementation is proven. Since we are using a *free* requires statement, this does not interfere with the call of the procedure. On the call site, only the precondition predicate is proven to hold prior to the call. For the postcondition it is the other way around. When the implementation is proven, the verifier is checking that the ensures statement hold after the procedure execution. Therefore, we use a normal ensures statement for the concrete postcondition. The verifier will then check the concrete postcondition at the end of the procedure. On the call site, the client can use the postcondition predicate which is added as a free ensures statement in addition to the concrete postcondition. Using the postcondition predicate, the caller can use the postcondition of the dynamic type. As the postcondition can only be strengthend, the use of both the concrete postcondition and the postcondition predicate is sound.

3.5.4 Class Invariants

When a routine is called, the target object has to be in a valid state. Therefore, the class invariant of the target object has to hold at the beginning of the routine execution. We encode this by using a free requires statement in the procedure definition (see Listing 3.17, line 35). By using a free precondition, the invariant is assumed to hold when the routine body is proven, but the client does not have to prove this. The invariant can be violated during the execution of the routine, but has to hold again after the execution. This is enforced by adding the invariant as a postcondition to the procedure definition (see Listing 3.17, line 36). The verifier will prove if the routine does establish the class invariant after the execution.

Note that this encoding is too simplistic. It cannot handle routine calls to other objects which may return to the original object. Improvements of the invariant encoding is future work (see Section 7.2).

3.5.5 Frame Condition

Boogie offers the `modifies` keyword to support the encoding of frame conditions. The keyword denotes global variables which may be changed during the execution of the procedure. This semantics is not expressive enough to encode the agent methodology. Therefore we use a different encoding of the frame condition. We always invalidate the whole heap with the `modifies` statement (Listing 3.17, line 40). In addition to the *modifies* statement, we add an additional postcondition which expresses the frame condition. The frame condition for the routine `make` is shown on line 41 of Listing 3.17. This is an encoding of the formula 2.5 from Section 2.5, which expresses that all locations on the heap which were not modified remain unchanged. For this routine, the modifies clause contains only the field `balance`, therefore only this field is allowed to change. As we use a normal `ensures` statement, the verifier is proving that the implementation respects the frame condition.

For pure routines, the encoding can be simplified to `Heap = old(Heap)`.

3.5.6 Creation Routines

The semantics for invariants is different for creation routines. The invariant does not hold at the beginning of the routine. Therefore, we cannot use the previously created procedure definition for creation routines. For each creation routine, we create a special procedure definition. Listing 3.18 shows the translation for the creation routine `make` of the class `ACCOUNT` from Listing 3.16. The only difference to the procedure definition for the routine `make` as a normal routine is the invariant section. The free requires clause which states that the invariant holds at the beginning of the routine is missing.

3.5.7 Pure Functions

A pure function can be used in a contract. In Boogie, expression are used to write the specifications. Expressions cannot contain a procedure call. We therefore need another representation of a pure routine in addition to the translation to a Boogie procedure. This is a functional representation, using a *function* in Boogie. To create the functional representation, we use the pre- and postcondition of the routine. The routine `is_empty` in Listing 3.19 is an example of a

Listing 3.18: *Boogie*: Translation of creation routine *make*

```
// Creation routine make from class ACCOUNT
procedure create.ACCOUNT.make(
    Current: ref where IsAttachedType(Heap,Current,ACCOUNT),
    arg1: int
  );

  // Pre- and postcondition predicate
  requires precondition.ACCOUNT.make(Heap, Current, arg1);
  free ensures postcondition.ACCOUNT.make(Heap, old(Heap),
    Current, arg1);

  // Pre- and postcondition
  free requires arg1 >= 0;
  ensures Heap[Current, field.ACCOUNT.balance] == arg1;

  // Class invariant, routine exit
  ensures Heap[Current, field.ACCOUNT.balance] >= 0;

  // Frame condition
  modifies Heap;
  ensures (forall<alpha> $o: ref, $f: Field alpha ::
    $o != Void && IsAllocated(old(Heap), $o) &&
    (!($o == Current && $f == field.ACCOUNT.balance))
      ==> (old(Heap)[$o, $f] == Heap[$o, $f]));
```

Listing 3.19: *Eiffel*: Pure routine

```
class LIST

feature

  is_empty: BOOLEAN
    require
      True
    do
      Result := count = 0
    ensure
      Result = (count = 0)
    end

end
```

Listing 3.20: *Boogie*: Pure routine

```
// Functional representation of is_empty
function fun.LIST.is_empty(heap: HeapType, current: ref)
  returns (bool);

axiom (forall heap: HeapType, current: ref ::
  true ==> (fun.LIST.is_empty(heap, current) ==
        (heap[current, field.LIST.count] == 0)));
```

pure routine. Listing 3.20 shows the translation to a functional representation in Boogie.

In the translation we first generate a function for the routine. The function takes the heap, the target object, and the routine arguments (if any) as argument. The return type of the Boogie function is the same as the return type of the Eiffel function. In addition to the Boogie function, we define an axiom. The axiom expresses that for all possible arguments to the function, if the arguments satisfy the precondition, the postcondition of the function holds. In the postcondition, the entity `Result` is replaced by the functional representation of the Eiffel query. This replacement expresses that the call to the query establishes the postcondition. The functional representation of queries is used whenever a query is used in a contract.

3.6 Routine Implementation

For each routine, we translate the body of the routine to an implementation block in the Boogie code file. We use the example in Listing 3.21 to show the translation of a routine body as well as the translation of several important instructions. Listing 3.22 shows the Boogie code of the translation. In the following sections, we explain the individual segments of the translated code.

3.6.1 Structure

An implementation block is started with the keyword `implementation`. This keyword is followed by the name of the procedure including the signature. As the routine `main` of Listing 3.21 has no arguments, only the reference to the current object is passed as a parameter.

Following the declaration of the block is the list of local variables. In our translation, there are two kinds of local variables: translation of local variables from Eiffel and variables temporarily used in the translation. The local variables from Eiffel are mapped directly to one local variable in the Boogie implementation. As the Eiffel compiler uses only an index variable to store local variables, the names of locals are not preserved in the translation. The temporary variables are used in the translation of object creation (see Section 3.6.2).

Listing 3.21: *Eiffel*: Implementation example

```
class CLIENT

feature

  main
    local
      a1, a2: ACCOUNT
      p: PROCEDURE [ANY, TUPLE [INTEGER]]
    do
        -- Object creation
      create a1.make

        -- Assignment
      a2 := a1
      my_account := a1

        -- Feature call
      a1.deposit (10)
      my_account.deposit (20)

        -- Agent creation
      p := agent a1.deposit (?)

        -- Agent call
      p.call ([30]);

        -- Check instruction
      check a1.balance = 60 end
    end

  my_account: ACCOUNT

end
```

Listing 3.22: *Boogie*: Implementation example

```
implementation proc.CLIENT.main(Current: ref) {
  var temp1: ref, temp2: ref;
  var local1: ref, local2: ref, local3: ref;

entry:
  // Initialization of locals
  local1 := Void;
  local2 := Void;
  local3 := Void;

  // Object creation
  havoc temp1;
  assume (temp1 != Void) && (!Heap[temp1, $allocated]);
  Heap[temp1, $allocated] := true;
  Heap[temp1, $type] := ACCOUNT;
  call create.ACCOUNT.make(temp1);
  local1 := temp1;

  // Assignment
  local2 := local1;
  Heap[Current, field.CLIENT.my_account] := local1;

  // Feature call
  call proc.ACCOUNT.deposit(local1, 10);
  call proc.ACCOUNT.deposit(
    Heap[Current, field.CLIENT.my_account], 20);

  // Agent creation
  havoc temp2;
  assume (temp2 != Void) && (!Heap[temp2, $allocated]);
  Heap[temp2, $allocated] := true;
  Heap[temp2, $allocated] := ANY;
  assume (forall heap: HeapType, a1: int ::
    routine.precondition_1(heap, temp2, a1) <==> (a1 >= 0));
  assume (forall<alpha> heap: HeapType, old_heap: HeapType,
      a1: int, $o: ref, $f: Field alpha ::
    agent.modifies_1(heap, old_heap, temp2, a1, $o, $f) <==>
    ($o == local1 && $f == field.ACCOUNT.balance));
  assume (forall heap: HeapType, old_heap: HeapType,
      a1: int ::
    routine.postcondition_1(heap, old_heap, temp2, a1) <==>
    ((heap[local1, field.ACCOUNT.balance] ==
      old_heap[local1, field.ACCOUNT.balance] + a1)));

  // Agent call
  call routine.call_1(local3, 20);

  // Check instruction
  assert Heap[local1, field.ACCOUNT.balance] == 30;

  return;
}
```

The starting point for the implementation is denoted by the label `entry` (Listing 3.22, line 5). This is where Boogie will start the execution of the implementation block. The exit point is defined by the keyword **`return`** (Listing 3.22, line 51). As Eiffel does not have a return statement in the language, the exit point of the implementation block is always the last statement before the block is closed.

Right after the entry label, the local variables are initialized to their default values (Listing 3.22, line 7): reference types are set to void, integers to zero, and booleans to false. This is necessary to have the behavior of default initialization as one expects it from Eiffel. Note that the temporary locals are not initialized to default values, since they are always properly initialized at the point where they are used.

The translation of the routine body is added between the initialization of the local variables and the return statement.

3.6.2 Object creation

When an object is created (Listing 3.21, line 11), we need to take an unused heap location where we can allocate the new object. In Boogie, we do this by using a new reference variable and setting it to a random value. This is done by the **`havoc`** statement (Listing 3.22, line 12). Then, we assume that the random value is not the void reference and that the new reference points to a location on the heap which is not yet allocated (Line 13). With this information, the prover can deduce that the reference is different from all other references to existing objects. On the next lines, we set the allocation ghost field and the type of the new object. Finally, we call the creation routine to initialize the new object (Line 16).

The creation of objects always uses a temporary variable to do the initialization. After the creation, we therefore have to assign the new reference to the target of the creation. We do the creation in this way, as there are situation where you do not have a target for the creation, for example if you pass the newly created object directly as an argument of a feature call. By using a temporary variable, we can just pass the temporary variable as the argument in this case.

3.6.3 Assignment

To translate the assignments to locals or attributes on line 14 and line 15 of Listing 3.21, we use the assignment operator of Boogie. In the case of a local variable, we just assign to the corresponding local variable in the Boogie code (Listing 3.22, line 20).

Assignments to attributes are translated as updates on the global heap variable. As Eiffel only allows assignments on the current object, the procedure argument `Current` is always used in Boogie for the reference on the heap. By translating the attribute name, we get the appropriate field name for the second argument to the heap. The translation of the assignment uses again the assignment operator of Boogie (Listing 3.22, line 21).

3.6.4 Routine calls

We use the procedure call of Boogie to translate routine calls. As Boogie is not object-oriented, we pass the target object as the first parameter. The call to `deposit` on line 18 in Listing 3.21 is translated to the procedure call to `proc.ACCOUNT.deposit` on line 24 in Listing 3.22. The target object is the local variable `a1`, which is translated to the local variable `local1` in the Boogie code. Additional arguments to the routine are passed as usual to the Boogie procedure.

The second call which is invoked on the attribute `account` (Listing 3.21, line 19) is again translated to a procedure call in Boogie (Listing 3.22, line 26). As the target object is an attribute of the current class, the target is passed by accessing the heap at the appropriate location.

3.6.5 Agent Creation

The translation for the creation of an agent is probably the most complex one. In the first part we create the agent object, similar to the creation of other objects as it is described in Section 3.6.2. As *EVE Proofs* does not yet handle generic types, we currently set the type for the agent to `ANY`. After the initialization of the object, according to the agent methodology from Section 2.4, we have to generate the assumptions to link the abstract predicates of the agent methodology to the concrete contracts of the agent. These assumptions are shown in Listing 3.22:

- Line 33 shows the assumption to link the abstract precondition predicate (`routine.precondition_1`) to the concrete precondition of the routine `deposit`.
- Line 35 shows the assumption to link the abstract modifies predicate (`agent.modifies_1`) to model the concrete modifies clause of the routine `deposit`. In line with the automatic extraction of modifies clauses, described in Section 2.5, only the attribute `balance` may change.
- Line 39 shows the assumption to link the abstract postcondition predicate (`routine.postcondition_1`) to the concrete postcondition of the routine `deposit`.

3.6.6 Agent Call

The call to the routine `call` of an agent is not translated as a normal routine call. The reason is that the methodology for agents uses different predicates depending on the number of open arguments to the agent. The background theory defines special procedures to call agents with different numbers of arguments (see Section 3.3.3).

Listing 3.23: *Eiffel*: Conditional

```
if condition1 then
  block1
elseif condition2 then
  block2
else
  block3
end
```

In the example in Listing 3.21, the agent on line 22 has one open argument. The call to the agent is therefore translated to a call to `routine.call_1` from the background theory (see Listing 3.22, line 46). The tuple which is used in the Eiffel code as an argument is ignored and its contents are directly passed to the call procedure of the agent. This means that only manifest tuples are supported as arguments to agent calls. This implementation needs to be adapted when generics are implemented and tuples are supported.

3.6.7 Check Instruction

Check instructions of Eiffel are directly translated to `assert` statements in Boogie. The check on line 28 in Listing 3.21 is translated to the assertion on line 49 in Listing 3.22. If a check instruction contains multiple assertions, an `assert` statement is generated for each one of them.

3.7 Control Structures

Boogie supports a nondeterministic jump instruction called `goto`. The `goto` statement lists one or more labels and one of them is chosen at random. We use this jump instruction to translate control flow structures by modeling the control flow graph. The translation we use is directly taken from Ballet [33].

3.7.1 Conditional

To show the translation of the conditional statement, we use the abstract example from Listing 3.23. The translation is shown in Listing 3.24. The translation of the conditions and blocks are designated with the prefix *b*.

For the conditional, we use the nondeterministic `goto` to jump to either the if-branch or the else-branch. When we take the if-branch, we know that the condition has to hold. We model this by *assuming* the condition. In the else-branch the condition does not hold. Again, we model this by an assumption, but this time assuming the negated condition.

To model cascading conditionals, we just consider them as single if-then-else statements where the second conditional is in the else-block of the first

Listing 3.24: *Boogie*: Conditional

```
  goto if_branch1, else_branch2;

if_branch1:
  assume (b_condition1);
  b_block1
  goto end5;

else_branch2:
  assume (!b_condition1);
  goto if_branch3, else_branch4;

if_branch3:
  assume (b_condition2);
  b_block2
  goto end5;

else_branch4:
  assume (!b_condition2);
  b_block3
  goto end5;

end5:
```

conditional.

3.7.2 Loop

To illustrate how we translate the loop, we use the abstract example of Listing 3.25. The Boogie code for the loop is shown in Listing 3.26. Again, the Boogie translation of the Eiffel blocks and conditions are designated by the prefix *b*.

In the translation of the loop, we first execute the statements of the *from* block. This block has to establish the loop invariant, thus we assert that the invariant holds at this point. Then, the control flow either enters the loop if the

Listing 3.25: *Eiffel*: Loop

```
from
  block1
invariant
  inv
until
  exit_condition
loop
  block2
end
```

Listing 3.26: *Boogie*: Loop

```
  b_block1
  assert (b_inv);
  goto loop_body, loop_exit;

loop_body:
  assume (!b_exit_condition);
  b_block2
  assert (b_inv);
  goto loop_body, loop_exit;

loop_exit:
  assume (b_exit_condition);
```

Listing 3.27: *Boogie*: Comment format for error tracing

```
requires a > 0; // pre ACCOUNT:14 tag:a_positive
ensures Result /= Void; // post FACTORY:18
ensures Heap == old(Heap); // frame LIST:is_empty
assert i < n; // loop ITERATOR:24
```

exit condition is not yet met or does not enter the loop at all. This is modeled by using a nondeterministic `goto` to jump to the loop body and the loop exit. If we enter the loop body, we know that the exit condition is not yet fulfilled. Thus, we assume the negated condition before executing the loop body. After the body, the loop invariant has to hold again, which is checked by asserting the loop invariant. After the invariant check, there is again a nondeterministic jump to the beginning of the loop body or the loop exit. This models the behavior of either re-executing the loop body or exiting the loop. After the loop, the exit condition is true. Therefore, we assume that the condition holds after the loop.

3.8 Tracing Errors

In order to trace Boogie errors back to the Eiffel source, we annotate the generated Boogie with location information. Each assertion in Boogie — either a `requires`, `ensures` or `assert` — has a comment describing what type of assertion it is and from which Eiffel source it was generated.

The comments have the format `TYPE CLASS:LOCATION tag:TAG_NAME`, where `TYPE` describes which type of assertion it is, `CLASS` is the classname and `LOCATION` either the line number or feature name in the Eiffel source. If the assertion in Eiffel has a tag, it is also added to the Boogie comment. Listing 3.27 shows some examples of assertions with the corresponding annotations.

The available assertion types are:

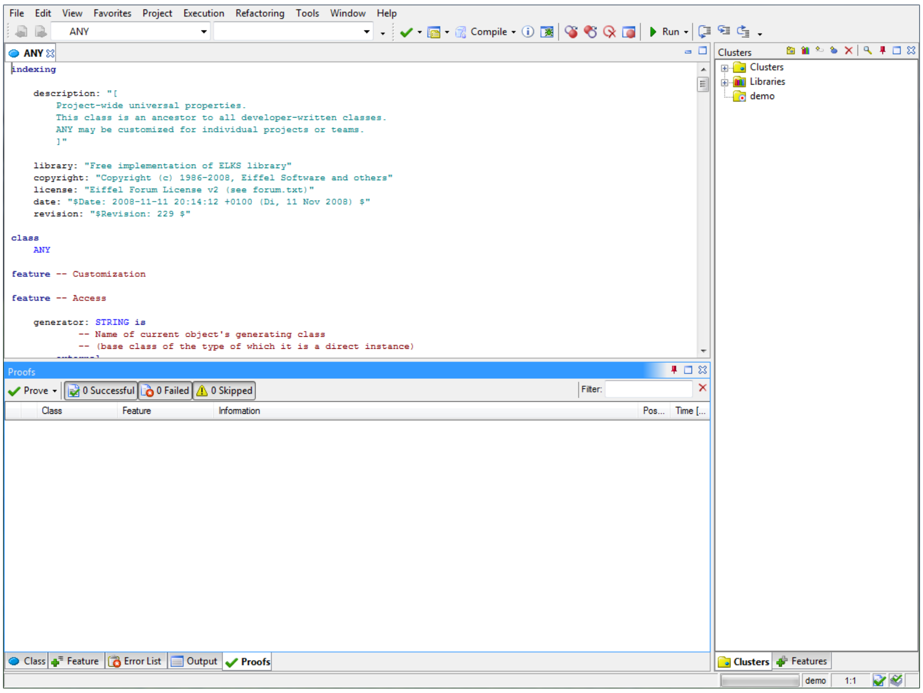
- **pre**: A precondition.

- **post**: A postcondition.
- **inv**: A class invariant.
- **loop**: A loop invariant.
- **check**: A check instruction.
- **attached**: An attachment check for the target of a feature invocation.
- **frame**: A frame condition.

Chapter 4

User Interface

EVE Proofs is integrated in EVE as a separate tool. Figure 4.1 shows a screenshot of EVE with the proof tool in the lower left. This chapter describes how this tool is used and how the feedback of the verification is displayed.



```
indexing

    description: "[
        Project-wide universal properties.
        This class is an ancestor to all developer-written classes.
        ANY may be customized for individual projects or teams.
        ]"

    library: "Free implementation of ELKS library"
    copyright: "Copyright (c) 1986-2008, Eiffel Software and others"
    license: "Eiffel Forum License v2 (see forum.txt)"
    date: "$Date: 2008-11-11 20:14:12 +0100 (Di, 11 Nov 2008) $"
    revision: "$Revision: 229 $"

class
    ANY

feature -- Customization

feature -- Access

    generator: STRING is
            -- Name of current object's generating class
            -- (base class of the type of which it is a direct instance)
```

File
Edit
View
Favorites
Project
Execution
Refactoring
Tools
Window
Help
ANY
Compile
Run
Clusters
Clusters
Libraries
demo
Proofs
Prove
0 Successful
0 Failed
0 Skipped
Filter:
Class
Feature
Information
Pos...
Time [...
Class
Feature
Error List
Output
Proofs
Clusters
Features
demo
1:1

Figure 4.1: Proof tool as part of EVE

4.1 Starting a Proof

There are two ways to start a proof: you can use the proof button or the context menu.

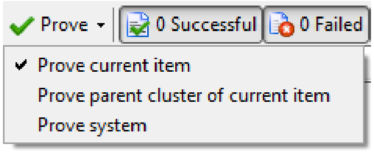


Figure 4.2: Proof button with the drop-down menu to select the mode of operation

Proof Button. The proof button is available in the project toolbar and in the toolbar of the proof tool. It has three different modes of operation which can be selected by the drop-down menu on the button. Figure 4.2 shows an image of the proof button with the drop-down menu. The three different modes are:

- **Prove current item**: The class or cluster which is open in the editor is proven. When you prove a cluster, it will recursively prove all contained classes and clusters.
- **Prove parent cluster of current item**: This mode proves the parent cluster of the currently opened class or cluster. Again, it recursively proves all contained classes and clusters.
- **Prove system**: The last mode proves all classes in the system. This does not include libraries.

When you just press the button, it will execute the last mode which was selected. The last selected mode is indicated in the drop-down menu with a tick mark. In addition to pressing the button, you can also drop a class or cluster pebble on the button to prove the dropped item.

Context Menu. A *prove* menu entry is added to the context menu of all classes, clusters and libraries. This entry starts the proof of the selected item. Figure 4.3 shows an image of the context menus.

4.2 Result Display

Figure 4.4 shows the proof tool after a verification has run. A row is added for each routine which was proven. The row displays the routine, the result of the verification, and how much time it took to verify the feature. Additional information is shown depending on the result of the verification:

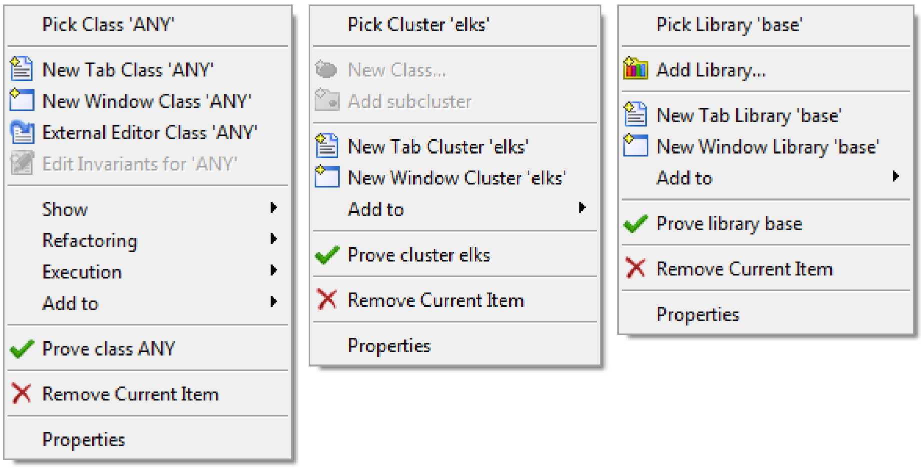
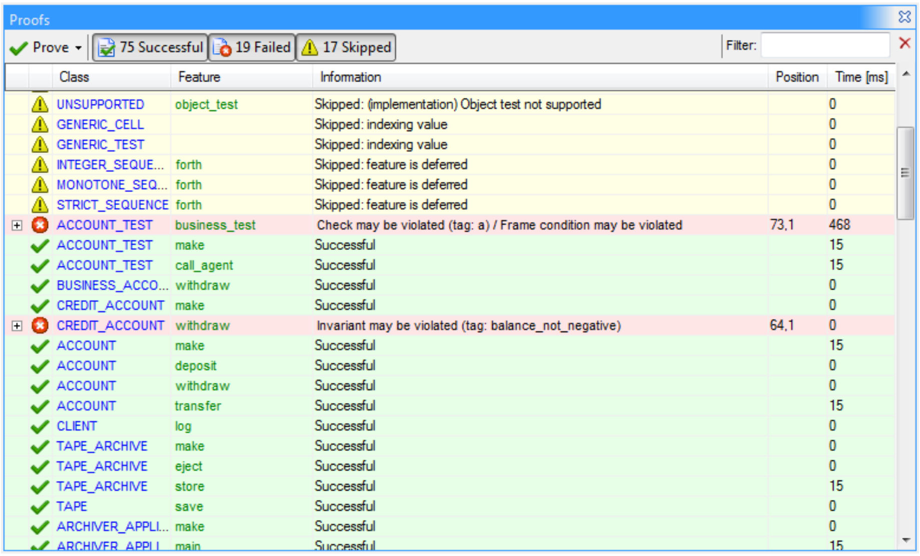

Pick Class 'ANY'
New Tab Class 'ANY'
New Window Class 'ANY'
External Editor Class 'ANY'
Edit Invariants for 'ANY'
Show
Refactoring
Execution
Add to
Prove class ANY
Remove Current Item
Properties
Pick Cluster 'elks'
New Class...
Add subcluster
New Tab Cluster 'elks'
New Window Cluster 'elks'
Add to
Prove cluster elks
Remove Current Item
Properties
Pick Library 'base'
Add Library...
New Tab Library 'base'
New Window Library 'base'
Add to
Prove library base
Remove Current Item
Properties

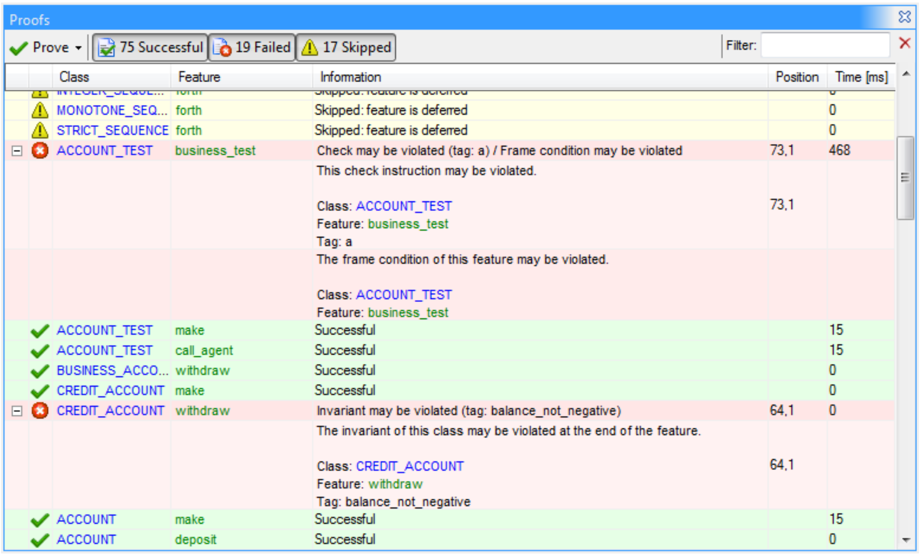
Figure 4.3: Context menu for classes, clusters and libraries which include the *prove* menu item


Proofs
Prove
75 Successful
19 Failed
17 Skipped
Filter:

		Class	Feature	Information	Position	Time [ms]
	⚠	UNSUPPORTED	object_test	Skipped: (implementation) Object test not supported		0
	⚠	GENERIC_CELL		Skipped: indexing value		0
	⚠	GENERIC_TEST		Skipped: indexing value		0
	⚠	INTEGER_SEQUE...	forth	Skipped: feature is deferred		0
	⚠	MONOTONE_SEQ...	forth	Skipped: feature is deferred		0
	⚠	STRICT_SEQUENCE	forth	Skipped: feature is deferred		0
⊞	✗	ACCOUNT_TEST	business_test	Check may be violated (tag: a) / Frame condition may be violated	73,1	468
	✓	ACCOUNT_TEST	make	Successful		15
	✓	ACCOUNT_TEST	call_agent	Successful		15
	✓	BUSINESS_ACCO...	withdraw	Successful		0
	✓	CREDIT_ACCOUNT	make	Successful		0
⊞	✗	CREDIT_ACCOUNT	withdraw	Invariant may be violated (tag: balance_not_negative)	64,1	0
	✓	ACCOUNT	make	Successful		15
	✓	ACCOUNT	deposit	Successful		0
	✓	ACCOUNT	withdraw	Successful		0
	✓	ACCOUNT	transfer	Successful		15
	✓	CLIENT	log	Successful		0
	✓	TAPE_ARCHIVE	make	Successful		0
	✓	TAPE_ARCHIVE	eject	Successful		0
	✓	TAPE_ARCHIVE	store	Successful		15
	✓	TAPE	save	Successful		0
	✓	ARCHIVER_APPLI...	make	Successful		0
	✓	ARCHIVER_APPLI	main	Successful		15

Figure 4.4: Proof tool which shows the results of the verification

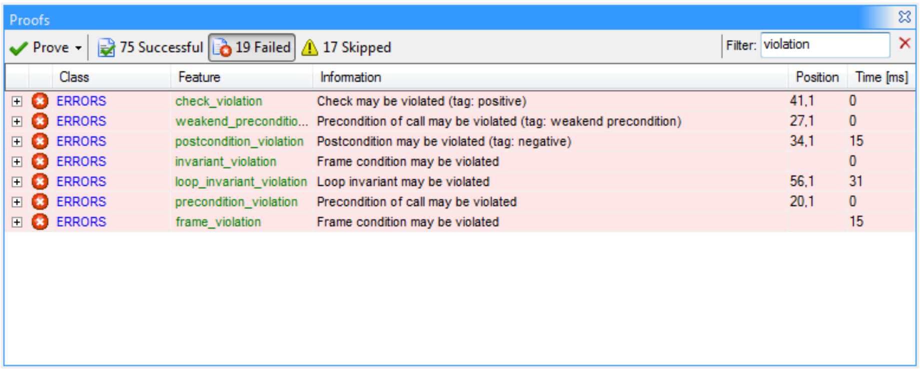
- No further information is given if the feature is proven to be correct. The background color of these rows is set to green.
- If the prover skipped a feature, the reason for doing so is displayed. Rows of skipped features are shown in yellow.
- If the verification failed, the summary for the error or multiple errors is shown. The row can be expanded to show the full error message. Figure 4.5 shows the proof tool with expanded rows for the failed features. Failed rows have a red background color.



		Class	Feature	Information	Position	Time [ms]
	⚠	MONOTONE_SEQ...	forth	Skipped: feature is deferred		0
	⚠	STRICT_SEQUENCE	forth	Skipped: feature is deferred		0
⊟	✗	ACCOUNT_TEST	business_test	Check may be violated (tag: a) / Frame condition may be violated	73,1	468
				This check instruction may be violated. Class: ACCOUNT_TEST Feature: business_test Tag: a	73,1	
				The frame condition of this feature may be violated. Class: ACCOUNT_TEST Feature: business_test		
	✓	ACCOUNT_TEST	make	Successful		15
	✓	ACCOUNT_TEST	call_agent	Successful		15
	✓	BUSINESS_ACCO...	withdraw	Successful		0
	✓	CREDIT_ACCOUNT	make	Successful		0
⊟	✗	CREDIT_ACCOUNT	withdraw	Invariant may be violated (tag: balance_not_negative)	64,1	0
				The invariant of this class may be violated at the end of the feature. Class: CREDIT_ACCOUNT Feature: withdraw Tag: balance_not_negative	64,1	
	✓	ACCOUNT	make	Successful		15
	✓	ACCOUNT	deposit	Successful		0

Figure 4.5: Proof tool where the rows of failed proofs are expanded to show the full error message

The result table can be sorted and filtered. The sorting is selected by clicking the appropriate table header. The rows can be filtered by the result type or by a search string. Three buttons toggle the display for successful, skipped, and failed routines. A string can be entered in the search box to filter the table to only show those rows which have the search string in any of the columns. The image in Figure 4.6 shows an example where the proof tool only shows failed routines which have the string *violation* in their name.



Class	Feature	Information	Position	Time [ms]
ERRORS	check_violation	Check may be violated (tag: positive)	41,1	0
ERRORS	weakend_preconditio...	Precondition of call may be violated (tag: weakend precondition)	27,1	0
ERRORS	postcondition_violation	Postcondition may be violated (tag: negative)	34,1	15
ERRORS	invariant_violation	Frame condition may be violated		0
ERRORS	loop_invariant_violation	Loop invariant may be violated	56,1	31
ERRORS	precondition_violation	Precondition of call may be violated	20,1	0
ERRORS	frame_violation	Frame condition may be violated		15

Figure 4.6: Filtered result table which shows only failed routines which contain the string *violation* in the name

Chapter 5

Implementation

This chapter describes the implementation of *EVE Proofs* and the integration in EVE. The proof engine, which handles the translation of Eiffel to Boogie and runs the theorem prover, is totally separate and only depends on the core of the EiffelStudio compiler. The user interface of *EVE Proofs* needs classes which are not available through libraries and thus this part is directly integrated in EVE.

The following sections give an overview of the implementation of the proof engine and the user interface.

5.1 Proof Engine

The proof engine is responsible to create Boogie code for the input classes, launch Boogie on the created code, and to evaluate the result. The engine is implemented as a separate library. The different parts of the engine are described below.

External Interface. To use the proof engine, clients can use a single class called `EVE_PROOFS` (*facade* pattern [20]). All the classes which should be proven can be added by calling `add_class_to_verify`. The prover is then started with `execute_verification`. This is the whole interaction from the client side.

When the verification is started, the Boogie code generator is invoked for each class which has been added. After the Boogie code is generated, the Boogie verifier is invoked. The results of the verification are not directly returned to the client. Instead, an event service is used where the results are published.

Related class: `EVE_PROOFS`

Boogie Code Generator. The Boogie code generator is the starting point for the translation of Eiffel to Boogie. It can handle whole classes, single features, and types. It further delegates the work to the various writer classes.

Related class: `BOOGIE_CODE_GENERATOR`

Code Writers. The writer classes are responsible to translate the actual Eiffel code to Boogie code. There are various different classes which handle a single aspect of the translation. The translation for the Eiffel code is done by processing the byte code tree - a simplified abstract syntax tree - using the visitor pattern [20].

- `EP_ATTRIBUTE_WRITER`: Creates the Boogie constant to denote the heap location for the attribute.
- `EP_CONSTANT_WRITER`: Translates a constant feature to Boogie.
- `EP_CONTRACT_WRITER`: Given a feature, this class creates the Boogie representation of the pre- and postcondition.
- `EP_EXPRESSION_WRITER`: This class translates all expressions to Boogie.
- `EP_FUNCTION_WRITER`: Creates a functional representation for pure routines and creates the axioms for the pre- and postcondition predicates.
- `EP_IMPLEMENTATION_WRITER`: Processes the body of a routine and creates the Boogie implementation block.
- `EP_INSTRUCTION_WRITER`: Translates single instructions like assignment and conditional statements to Boogie.
- `EP_SIGNATURE_WRITER`: Creates the Boogie code for routine signatures.
- `EP_TYPE_WRITER`: Translates types to Boogie and handles the inheritance relationship.

The writer classes use several helper classes, for example `EP_FRAME_EXTRACTOR`, which handles the automatic extraction of modifies clauses. The frame extraction is used for example in the feature `process_routine_creation_b` of class `EP_EXPRESSION_WRITER` to generate the assumptions needed for the agent framing.

Referenced Features and Types. All used Eiffel features and types are recorded during the creation of the Boogie code. These lists are processed in the end to create the signature for all referenced features and types. The implementation for the routines is omitted though, as the referenced routines are not verified.

Related classes: `EP_FEATURE_LIST`, `EP_TYPE_LIST`

Output Buffering. All created Boogie code is stored in output buffers. Each writer class has its own buffer. These buffers are then subsequently put together to form the final Boogie code.

Related class: `EP_OUTPUT_BUFFER`

Boogie Execution. The Boogie verifier class takes the created code and creates a Boogie code file. This file is then fed to the Boogie executable in a separate process. The Boogie verifier is launched with the */trace* command line argument, which tells Boogie to display more output information about the verified routines. This information is used to improve the display of the results. When Boogie is finished, the output from the execution as well as the input file are given to the output parser for further processing.

Related class: `BOOGIE_VERIFIER`

Output Parser. The output parser is handling the output of Boogie. It takes the output and parses it line by line using a set of regular expressions. For successful proofs, it publishes a *successful* event. When Boogie reports a violation, it looks at the appropriate line in the Boogie code file to trace the error back to the Eiffel program. For each routine which could not be proven, a *failed* event is published along with a list of error messages for this routine.

Related class: `BOOGIE_OUTPUT_PARSER`

Event Service. To report the results of a proof to the proof tool, the EiffelStudio event service is used. This service allows for an arbitrary publisher to post events to multiple unknown consumers.

EVE Proofs uses this service to publish events when a proof was successful, failed, or had to be skipped. All events contain information to identify which routine or class is concerned. The skipped events additionally have the reason for skipping the routine. Failed events have a list of errors passed as part of the event.

By using the event service, it is easy to extend the feedback or do further processing of the information. You can just add a new consumer which listens to the proof events and you will get informed about the proof results.

Related classes: `EVENT_LIST_PROOF_ITEM_I` and subclasses, `EP_EVENT_HANDLER`

5.2 User Interface

The user interface is implemented as an EiffelStudio command and an EiffelStudio tool.

Proof Command. EiffelStudio uses the command pattern [20]. For starting proofs, there is a single command class which starts the verifier. Depending on the input for the command, either the whole system, a library, a cluster, or a single class is proven.

Related class: `EB_PROOF_COMMAND`

Proof Tool. The proof tool is implemented as a subclass of the general event list tool. It is similar to the *errors and warnings* display. The tool listens to incoming events from the event service. When it receives a proof event, it displays the information in the results table.

Related classes: `ES_PROOF_TOOL`, `ES_PROOF_TOOL_PANEL`

Internationalization. All messages which appear in the user interface use the EiffelStudio translation mechanism. This allows to translate the proof tool in the same way as the rest of EiffelStudio.

Related class: `EP_NAMES`

Changes to EiffelStudio. As EiffelStudio does not have a plug-in mechanism, some changes to existing classes were necessary. These changes are:

- Adding the *proof tool* to the menu.
- Adding the *prove* menu item to the context menu for clusters, classes, and libraries.
- Adding the *prove* button to the toolbar.

More information about adding a tool to EiffelStudio is available on the development wiki [7].

Related classes: `EB_DEVELOPMENT_WINDOW_COMMANDS`, `EB_DEVELOPMENT_WINDOW_MAIN_BUILDER`, `EB_DEVELOPMENT_WINDOW_MENU_BUILDER`, `EB_CONTEXT_MENU_FACTORY`, `EB_DEBUGGER_MANAGER`

Chapter 6

Case Studies

This section gives an overview of some of the examples we are able to verify. You will find the source code for the examples in the EiffelStudio repository [6], specifically in the examples directory of the EVE branch [8].

6.1 Account

The account example in Listing 6.1 shows the use of the automatic frame condition extraction for simple programs. The problem is to verify the check conditions on line 11 and 12. The account class which is used is shown in Listing 6.2.

To prove the first check instruction we need to prove that the calls `a2.make` and `a2.withdraw` did not change the `balance` field of `a1`. To reason about the effect of the routine calls on other objects, we need to know the modifies clause of the routines of class `ACCOUNT`. As the postconditions of `make`, `deposit`, and `withdraw` only mention the attribute `balance` of the current object, the frame

Listing 6.1: *Eiffel*: Account client

```
    class CLIENT
 2
      main
 4      local
          a1 , a2: ACCOUNT
 6      do
          create a1.make (100)
 8        create a2.make (200)
          a1.deposit (50)
10        a2.withdraw (180)
          check a1.balance = 150 end
12        check a2.balance = 20 end
        end
14
    end
```

Listing 6.2: *Eiffel*: Account class

```
class ACCOUNT

  make (a_initial_amount: INTEGER)
    require
      a_initial_amount >= 0
    do
      balance := a_initial_amount
    ensure
      balance = a_initial_amount
    end

  balance: INTEGER

  deposit (amount: INTEGER)
    require
      amount >= 0
    do
      balance := balance + amount
    ensure
      balance = old balance + amount
    end

  withdraw (amount: INTEGER)
    require
      amount <= balance + credit_limit
    do
      balance := balance - amount
    ensure
      balance = old balance - amount
    end

end
```

condition of these three routines is equivalent to having a modifies clause which only lists the field `balance` of the current object (see Section 2.5). With this information, the prover can deduce that the call on the object `a2` has no influence on `a1`, and thus the check instruction can be verified. Note that the prover knows that the two objects `a1` and `a2` are different due to the translation of the create instructions which uses an unused heap location for each created object (see Section 3.6.2).

The reasoning to prove the second check instruction is the same. The calls on `a1` have no influence on `a2` due to the automatically extracted modifies clauses. Therefore the prover can verify the second assertion as well.

6.2 Formatter

The formatter example was proposed by Leavens, Leino and Müller [21] as an open problem. The example shows the use of agents with open arguments. Nordio *et al.* [27] have shown how to verify this example. However, they do not implement a tool that can automatically prove this example. In this section, we show this example which can be automatically proven by our tool.

Listing 6.3: *Eiffel*: Formatter class

```
class FORMATTER

  align_left (p: PARAGRAPH)
    require
      not p.is_left_aligned
    do
      -- Operation on 'p'
    ensure
      p.is_left_aligned
    end

  align_right (p: PARAGRAPH)
    require
      p.is_left_aligned
    do
      -- Operation on 'p'
    ensure
      not p.is_left_aligned
    end

end
```

Listing 6.4: *Eiffel*: Paragraph class

```
class PARAGRAPH

  format (proc: PROCEDURE [FORMATTER, TUPLE [PARAGRAPH]])
    require
      proc.precondition ([Current])
    do
      proc.call ([Current])
    ensure
      proc.postcondition ([Current])
    end

end
```

The classes involved are shown in the Listings 6.3, 6.4, and 6.5. This example defines a class `FORMATTER` which has features to change the alignment on the class `PARAGRAPH`. The `PARAGRAPH` class has a routine `format` which takes an agent to format itself. The `format` routine just calls the agent which is passed, using the current object as the argument to the agent. The client class defines a routine `apply_align_left` which takes a formatter and paragraph object. The formatter is used to format the paragraph by calling the `format` routine on the paragraph with an agent created from the formatter object. There are two interesting problems in this example: how to specify the `format` routine of the `PARAGRAPH` class, and how to prove the use of the agent in the `CLIENT` class.

The routine `format` in Listing 6.4 takes an agent and calls the agent with the current object as an argument. For the call to the agent to be valid, the precondition of the agent has to hold. As we do not know the contract of the agent which is called, we move the responsibility to guarantee the precondition to the

Listing 6.5: *Eiffel*: Formatter client

```
   class CLIENT
 2
     apply_align_left (f: FORMATTER; p: PARAGRAPH)
 4     require
         not p.is_left_aligned
 6     local
         proc: PROCEDURE [FORMATTER, TUPLE [PARAGRAPH]]
 8     do
         proc:= agent f.align_left
10       p.format (proc)
       ensure
12       p.is_left_aligned
       end
14
   end
```

client of the routine `format`. For this, we use the query `precondition` offered by the class `PROCEDURE`. In the translation to Boogie, this query is treated special. The methodology for agents introduces precondition predicates for agents which depend on the number of open arguments. The routine `precondition` is now translated to the precondition predicate with the appropriate number of open arguments, which is defined in the background theory (see Section 3.3.3). Also, we use a special translation for the Eiffel routine `postcondition` of the class `PROCEDURE`. The routine `postcondition` is translated to the postcondition predicate for agents, again depending on the number of open arguments. Using the routines `precondition` and `postcondition` of the class `PROCEDURE` and their special translation, we can express the specification of the routine `format` of the class `PARAGRAPH`.

Since we have now specified the routine `format`, we can reason about the client class which creates an agent and uses the `format` routine. The proof of the routine `apply_align_left` of Listing 6.5 is done by using the agent methodology described in Section 2.4. When the agent is created on line 9 in Listing 6.5, the translation is generating assumptions to link the pre- and postcondition of the routine `align_left` to the abstract predicates used for the agent pre- and postcondition. These agent predicates match the pre- and postcondition predicates which are used in the specification of the routine `format`. When the routine `format` is called with the created agent, the prover can use the generated assumptions to deduce that the precondition of `format` is satisfied, and therefore assumes that the postcondition of `format` holds. With the assumption that connects the concrete postcondition of `format_left` with the abstract predicate used in the postcondition of `format`, the verifier can assume that the concrete postcondition of `format_left` holds. Using this assumption, the verifier can prove that the paragraph is left aligned after the execution of `apply_align_left`.

Listing 6.6: *Eiffel*: Command class

```
class COMMAND

  make (a_action: like action)
    do
      action := a_action
    ensure
      action = a_action
    end

  action: PROCEDURE [ANY, TUPLE [INTEGER]]

  execute (a_arg: INTEGER)
    require
      action.precondition ([a_arg])
    do
      action.call ([a_arg])
    ensure
      action.postcondition ([a_arg])
    end

end
```

6.3 Command Pattern

This example shows the command pattern [20, 13]. The code of the class `COMMAND` is shown in Listing 6.6, and the code of the client class is shown in Listing 6.7. The example shows the use of agents as attributes. For this, we define the class `COMMAND` and define an attribute `action` which stores an agent taking an integer as argument.

Looking at the routine `execute` in Listing 6.6, we see that the agent `action` is called with the argument which is passed to `execute`. Like in the *formatter* example, we need to prove that the precondition of the agent holds. We move this obligation again to the client by using the `precondition` routine of the class `PROCEDURE`. After the call to the agent, we know that the postcondition of the routine used to create the agent holds. This information is forwarded to the client by using the routine `postcondition` of the class `PROCEDURE` in the postcondition of `execute`.

Listing 6.7 shows a client of the class `COMMAND`. The client creates an account object and a command using the `withdraw` routine of the account class (the account class is declared in Listing 6.2). The problem is now to prove the check instruction after the call to `execute`.

First, we need to prove that we can execute the command. The precondition of `execute` is the precondition of the agent stored in the command object. The postcondition of `make` ensures that the passed agent is stored in the attribute `action`. Therefore, we can conclude that the precondition of `execute` is the precondition of the routine `withdraw`, which is satisfied by the given argument. This implies that after the call to the routine `execute`, the postcondition of `execute` holds. The postcondition of `execute` is the postcondition of the agent

Listing 6.7: *Eiffel*: Command client

```
class CLIENT

feature

  main
    local
      a: ACCOUNT
      c: COMMAND
    do
      create a.make (100)
      create c.make (agent a.withdraw (?))

      c.execute (50)

      check a.balance = 50 end
    end
end
```

Listing 6.8: *Eiffel*: Operator strategy class

```
deferred class OPERATOR_STRATEGY

  execute (a, b: INTEGER)
    deferred
    end

  last_result: INTEGER

end
```

stored in the attribute `action`. Since we already know that the agent is the routine `withdraw`, we know that the call to the command reduced the `balance` field of the account object `a`. With this information, we can prove the check instruction.

6.4 Strategy Pattern

The last example we present is the strategy pattern [20, 13]. For this, we first define a deferred class `OPERATOR_STRATEGY` which is shown in Listing 6.8. The strategy class has a routine `execute` which takes two integers as argument. The result of the operation is then stored in the attribute `last_result`. A possible implementation of this strategy class is shown in Listing 6.9. The class `ADDITION_STRATEGY` inherits from the deferred strategy class and implements the operation as an addition.

The strategy pattern now defines a context class which uses a strategy object for an internal operation. The advantage of the pattern is, that the strategy can be changed without changing the context class. The context class for this example is shown in Listing 6.10. The routine `apply_strategy` takes also two

Listing 6.9: *Eiffel*: Addition strategy class

```
class ADDITION_STRATEGY inherit OPERATOR_STRATEGY

  execute (a, b: INTEGER)
    do
      last_result := a + b
    ensure then
      last_result = a + b
    end

end
```

integer arguments and delegates the work to the strategy object. To retrieve the result of applying the strategy, the context class offers a query `last_result` which just returns the value of the last operation on the strategy object.

The client class from Listing 6.11 is creating a context object and initializes it with a strategy of type `ADDITION_STRATEGY`. After applying the strategy to the values 1 and 2, we want to prove that the result is 3.

This example shows two problems: we need to specify `apply_strategy` which delegates the work to the strategy object, and we need to prove that the execution of the strategy does more than the static contract specifies.

In the postcondition of `apply_strategy`, we want to express that the execution of `apply_strategy` has the same effect as the call to `execute` on the strategy object. One way would be to write the postcondition of `execute` again in the postcondition of `apply_strategy`. This is not good enough, as this would only cover the postcondition of the static type of `execute`. We want to express that `apply_strategy` has the same effect as the postcondition of the routine `execute` of the *dynamic* type of the strategy object. To do this, we use the postcondition predicates we introduced in Section 2.3. These postcondition predicates denote the postcondition of a routine using the dynamic type of the target object. To express in the postcondition of `apply_strategy` that we ensure the postcondition of the routine `execute`, we just have to ensure the postcondition predicate of the routine `execute`. On line 16 in Listing 6.10 you can see how we encode this in Eiffel: we use the `postcondition` query introduced by agents. This statement is translated into the postcondition predicate of the agent routine, which is in this example the routine `execute`. Using the postcondition predicates introduced for dynamic invocation and the `postcondition` query of agents, we can express the delegation in the postcondition of `apply_strategy`.

Using this specification for the routine `apply_strategy` and the special translation of its postcondition, we can look at the second problem: proving the check instruction on line 9 of Listing 6.11. When the context object is created, we pass a strategy object of type `ADDITION_STRATEGY`. Using the postcondition of `make`, we know that the strategy object is stored in the `strategy` attribute of the context object. Then, the routine `apply_strategy` is called. As the routine has no precondition, we can just assume that the postcondition holds. According to the previously mentioned encoding, the postcondition is actually the postcondition predicate for the routine `execute`. Using the methodology introduced for

Listing 6.10: *Eiffel*: Context class

```
class CONTEXT

  make (a_strategy: OPERATOR_STRATEGY)
    do
      strategy := a_strategy
    ensure
      strategy = a_strategy
    end

  strategy: OPERATOR_STRATEGY

  apply_strategy (a, b: INTEGER)
    do
      strategy.execute (a, b)
    ensure
      (agent strategy.execute).postcondition ([a, b])
    end

  last_result: INTEGER
    do
      Result := strategy.last_result
    ensure
      Result = strategy.last_result
    end

end
```

Listing 6.11: *Eiffel*: Strategy client

```
class CLIENT

  main
    local
      c: CONTEXT
    do
      create c.make (create {ADDITION_STRATEGY})
      c.apply_strategy (1, 2)
      check c.last_result = 3 end
    end

end
```

dynamic invocation, we can use the postcondition of the dynamic type of the strategy object. Since we know that the dynamic type is `ADDITION_STRATEGY`, we can assume the postcondition of `execute` from Listing 6.9 which states that `last_result` has been set to the addition of the arguments. With this information, the verifier can prove the check instruction.

Chapter 7

Conclusions

7.1 Conclusions

In this thesis, we have developed the tool *EVE Proofs* which offers fully automated proofs for Eiffel. In addition to the basic instructions, the tool has an automatic extraction of modifies clauses, a methodology to prove the use of agents, and an approach to use the contracts of the dynamic type for routine calls. The tool has been integrated in EVE.

Verification of Dynamic Invocation. We developed a methodology to prove dynamic invocation. The pre- and postconditions of routines are expressed through abstract predicates. These predicates are then linked to the concrete contracts based on the dynamic type. With this approach, we are able to prove examples which would not be possible by using only the static information.

In addition, these predicates can be used to express delegation. This enabled us to prove the strategy pattern, even if the contract of the concrete strategy is stronger than in the abstract strategy.

Verification of Agents. We were able to implement the proposed methodology to verify agents [27]. This allowed us to prove the formatter and archiver example proposed by Leavens, Leino and Müller [21] as well as the command pattern. In addition, we extended the methodology to handle framing for agents. With this, the approach now considers all aspects of the specification of a routine.

Automatic Extraction of Modifies Clauses. The automatic extraction of modifies clauses which we developed is very basic. The approach is good enough to prove small examples without worrying about framing. The problem of overapproximation and the lack of a method to specify changes to whole object structures requires us to improve the framing in the future.

User Interface. We have integrated the tool in EVE. It can handle multiple classes at once, offering three different modes of operation: *prove a class*, *prove a cluster*, and *prove the whole system*. This makes the tool very convenient to use. The results of the verification are presented in a well-arranged way. This makes the tool usable for all programmers.

7.2 Future Work

This thesis lays the ground for a lot of possible improvements. These can either be more theoretical — providing new methodologies to proof program constructs and translate them to Boogie — or practical — improving user experience and integrating *EVE Proofs* with other tools. Following is a list of possible projects.

Generics. An important translation which is missing at the moment is the translation for generic types. There are two aspects for this: The translation of the definition of a generic class, and the translation of the use of generics. For the definition of generics, the translation to Boogie has to take into account that the class has to be correct for each possible generic derivation. In particular, the class has to be correct for reference types and for expanded types. To translate a call to a routine of a generic, the easiest solution is to create a translation for each different generic derivation. In this way, you know the exact type in the translation which simplifies the generation of the Boogie code.

The implementation of generics will allow a wide variety of examples to be proven, as generics are needed for *EVE Proofs* to run code using the collection classes from EiffelBase.

Exception Handling. Routines with rescue clauses are skipped by the verifier at the moment. Nordio *et al.* [29, 28] have developed an axiomatic and operational semantics for Eiffel which includes exception handling. However, a translation to Boogie needs to be developed.

Expanded Types. *EVE Proofs* does not support expanded types. Only the basic use of integers and booleans is supported, as they are directly available in Boogie as well. Expanded types pose a challenge for verification: each entity which is possibly attached to an expanded type changes the semantics for comparison and reattachment. The equality for expanded types compares the object contents instead of the references and reattachment of expanded types uses the copy semantics instead of the reference semantics. A methodology which can handle this has to be developed and then implemented in *EVE Proofs*.

Switching Methodologies. As there are multiple ways to encode Eiffel to Boogie, it is not always obvious which solution to take. Each solution has its own advantages and drawbacks. To compare different methodologies, we could set up a system to choose between different encodings.

Furthermore, one could add a translation using another theorem prover. This could allow us to prove some programs that Boogie might have problems with. Additionally, the translation to multiple theorem provers could be used to verify different properties of the same program with different theorem provers, exploiting their respective strengths and avoiding their weaknesses.

Framing. The automatic extraction of modifies clauses has limitations. One cannot specify changes for whole object structures like arrays. When generics are implemented, this will be a major restriction for specifying routines. One way to resolve this situation would be to introduce something similar to the asterisk (*) notation of Spec# to denote changes to all fields of an object.

Although we can use the dynamic type for the pre- and postcondition of a routine call, we still use the static type for the frame condition. This can lead to unsoundness in the system. The methodology for framing has to be extended to use the dynamic type as well.

Invariants. The methodology for invariants has to take into account that objects can temporarily violate the invariant, but also that an object can call other objects while being in an inconsistent state. As this is not considered at the moment, the current implementation of invariants can introduce unsoundness in the system. A better implementation is needed which retains the soundness of the system, for example by restricting either the expressiveness of invariants or the points where an invariant can be broken. A look at Spec# is a good starting point [15, 23].

Error Reporting. *EVE Proofs* shows an error for each violation found by Boogie. The error messages are very basic and the additional information about the error is limited. This can be improved in various ways:

- The error messages can be more elaborate.
- Additional information can be added to the error report, for example the extracted modifies clause.
- For conditionals, Boogie provides a trace of the path that has been taken when a violation occurred. This information can be shown to the user to better understand where an error originated.
- The tool could suggest a solution to fix an error, for example adding a new precondition.

Integrate MML. The specification language of Eiffel cannot express quantifier expressions. The mathematical modeling library (MML) [34, 10] can help to overcome this limitation. For this, the MML classes have to be represented in Boogie, and then the calls to the model features can be mapped to the appropriate features in Boogie.

Integrate with Testing. Using the reasoning that a routine which is proven to be correct does not need to be tested, the proof tool can be integrated with the testing facilities of EiffelStudio. As *EVE Proofs* publishes events for the results of the proofs, the testing tools can use these events to stop the tests for features which are successfully proven.

User Experience. There are various ideas to enhance the user experience when using *EVE Proofs*:

- Launch Boogie in an asynchronous way, so that EVE does not block while Boogie is running.
- Store created Boogie code and only translate the classes which changed since the last run of the verifier. This improves the speed of the translation.
- Do incremental proofs by only proving routines which changed since the last run of the verifier. This improves the speed of the verification.
- Run the verifier in the background without the need to explicitly start the proof.

7.3 Related Work

The Spec# programming system [16] is an automatic verifier for C#. The translation of the basic instructions of Spec# to Boogie have been used as a model for our translation. Müller and Ruskiewicz have extended Spec# to handle C# delegates [26]. To tackle the frame problem, Spec# uses ownership. Spec# is more powerful than *EVE Proofs*. However, it is also harder to use. *EVE Proofs* implements a new encoding for inheritance which allows us to prove programs that Spec# cannot prove.

The ESC/Java [19] system has been developed to verify Java classes which use JML [17] for the specification. JML follows a similar approach as *EVE Proofs*, since it does not alter the underlying programming language. Yet, JML itself is already an extension of Java, where the specification has to be added as specially formatted comments.

Distefano and Parkinson have implemented an automatic verifier for Java programs called jStar [18]. This work implements the idea of abstract predicates [30, 31] and handles framing with separation logic [32]. The tool can handle interesting examples such as the visitor pattern.

Smans *et al.* propose an approach with implicit dynamic frames [35]. They implemented the method in an experimental language which adds an accessibility predicate to the specification language. Writing and reading of a field is only allowed if the field is accessible. With this information, they infer the framing properties of a routine from the contract.

Schoeller has developed an experimental automatic verifier for Eiffel called Ballet [33]. Ballet also translates Eiffel to Boogie code. To specify framing, it introduced *modifies* clauses in Eiffel. The general structure of the proof engine we implemented is influenced by Ballet. Also, the translation for control flow structures is taken from Ballet.

Listings

2.1 *Eiffel*: Dynamic invocation example 14
2.2 *Eiffel*: Agent example 18
2.3 *Eiffel*: Signature of call 19
2.4 *Eiffel*: Frame example 22
2.5 *Eiffel*: Person class 22
2.6 *Eiffel*: Frame inheritance 23
2.7 *Eiffel*: Overapproximation 24
2.8 *Eiffel*: Pure Marking Problem 26
3.1 *Boogie*: Type Example 27
3.2 *Boogie*: Function Example 27
3.3 *Boogie*: Variable Example 28
3.4 *Boogie*: Axiom Example 28
3.5 *Boogie*: Procedure Example 28
3.6 *Boogie*: Implementation Example 28
3.7 *Eiffel*: Naming example 29
3.8 *Boogie*: Reference types 30
3.9 *Boogie*: Heap model 31
3.10 *Boogie*: Typing 32
3.11 *Boogie*: Agent theory 33
3.12 *Eiffel*: Person class 34
3.13 *Boogie*: Code for basic fields 34
3.14 *Boogie*: Boogie code for natural field 35
3.15 *Boogie*: Code for attached field 35
3.16 *Eiffel*: Interface of Account 36
3.17 *Boogie*: Translation of routine *make* 37
3.18 *Boogie*: Translation of creation routine *make* 40
3.19 *Eiffel*: Pure routine 40
3.20 *Boogie*: Pure routine 41
3.21 *Eiffel*: Implementation example 42
3.22 *Boogie*: Implementation example 43
3.23 *Eiffel*: Conditional 46
3.24 *Boogie*: Conditional 47
3.25 *Eiffel*: Loop 47
3.26 *Boogie*: Loop 48
3.27 *Boogie*: Comment format for error tracing 48
6.1 *Eiffel*: Account client 61
6.2 *Eiffel*: Account class 62
6.3 *Eiffel*: Formatter class 63
6.4 *Eiffel*: Paragraph class 63

6.5 *Eiffel*: Formatter client 64
6.6 *Eiffel*: Command class 65
6.7 *Eiffel*: Command client 66
6.8 *Eiffel*: Operator strategy class 66
6.9 *Eiffel*: Addition strategy class 67
6.10 *Eiffel*: Context class 68
6.11 *Eiffel*: Strategy client 68

Bibliography

[1] Boogie. `http://research.microsoft.com/boogie/`.

[2] Eclipse Project. `http://eclipse.org/`.

[3] Eiffel Standard Ecma-367. `http://www.ecma-international.org/publications/standards/Ecma-367.htm`.

[4] Eiffel Verification Environment (EVE). `http://eve.origo.ethz.ch`.

[5] EiffelStudio. `http://dev.eiffel.com`.

[6] EiffelStudio SVN repository. `https://svn.origo.ethz.ch/eiffelstudio/`.

[7] EiffelStudio Tool Integration. `http://dev.eiffel.com/Tool_Integration_Development`.

[8] EVE Proofs Examples SVN repository. `https://svn.origo.ethz.ch/eiffelstudio/branches/eth/eve/Src/examples/proofs/`.

[9] Java Modeling Language (JML). `http://www.cs.ucf.edu/~leavens/JML/`.

[10] Mathematical Modeling Library (MML). `http://mml.origo.ethz.ch/`.

[11] Microsoft Visual Studio. `www.microsoft.com/visualstudio/`.

[12] Spec#. `http://research.microsoft.com/specsharp/`.

[13] Karine Arnout. *From patterns to components*. PhD thesis, ETH Zürich, 2005.

[14] Mike Barnett, Bor-Yuh Evan Chang, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. Technical report, Microsoft Research, 2006.

[15] Mike Barnett, Robert Deline, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3, 2004.

[16] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. Technical report, Microsoft Research, 2004.

[17] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, Rustan, and Erik Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, V7(3):212–232, June 2005.

[18] Dino Distefano and Matthew J. Parkinson. jStar: Towards Practical Verification for Java. In Gail E. Harris, editor, *OOPSLA*, pages 213–226. ACM, 2008.

[19] Cormac Flanagan, Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *SIGPLAN Not.*, 37(5), May 2002.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[21] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, 2007.

[22] K. Rustan M. Leino. This is Boogie 2. Technical report, Microsoft Research, 2008.

[23] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods (FM)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer-Verlag, 2005.

[24] Bertrand Meyer. *Eiffel: the language.* Prentice-Hall, Inc., 1992.

[25] Bertrand Meyer. *Object-oriented software construction (2nd ed.).* Prentice-Hall, Inc., 1997.

[26] Peter Müller and Joseph N. Ruskiewicz. A modular verification methodology for C# delegates. In U. Glässer and J.-R. Abrial, editors, *Rigorous Methods for Software Construction and Analysis*, 2007. To appear.

[27] Martin Nordio, Cristiano Calcagno, Bertrand Meyer, and Peter Müller. Reasoning about Function Objects. Technical Report 615, ETH Zürich, 2009.

[28] Martin Nordio, Cristiano Calcagno, Peter Müller, and Bertrand Meyer. A Sound and Complete Program Logic for Eiffel. In M. Oriol, editor, *TOOLS-EUROPE 2009*, Lecture Notes in Business and Information Processing, 2009. To appear.

[29] Martin Nordio, Peter Müller, and Bertrand Meyer. Proof-transforming compilation of eiffel programs. In R. Paige, editor, *TOOLS-EUROPE 2008*, Lecture Notes in Business and Information Processing. Springer-Verlag, 2008.

[30] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL '05*, pages 247–258, New York, NY, USA, 2005. ACM.

[31] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08*, pages 75–86. ACM, 2008.

[32] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium onLogic in Computer Science*, 2002.

[33] Bernd Schoeller. *Making classes provable through contracts, models and frames*. PhD thesis, ETH Zürich, 2008.

[34] Bernd Schoeller, Tobias Widmer, and Bertrand Meyer. Making specifications complete through models. In *Architecting Systems with Trustworthy Components*, volume 3938 of *Notes in Computer Science*. Springer-Verlag, 2003.

[35] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *ECOOP 2009 - Object-oriented Programming, 23rd European Conference, Genova, Italy, July 6-10, 2009, Proceedings*, July 2009.