Simpler Specifications for Resource-Manipulating Programs

by

Zachary Grannan

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

April 2023

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Simpler Specifications for Resource-Manipulating Programs

submitted by **Zachary Grannan** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Alexander J. Summers, Associate Professor, Computer Science, UBC
Supervisor

William Bowman, Assistant Professor, Computer Science, UBC
Supervisory Committee Member

Abstract

Many program verifiers allow specifications to be written in terms of program states. The specification language of a program verifier is typically a superset of the source language, extended with additional constructs such as first-order quantifiers. This is a pragmatic choice because it allows the same language to be used for both implementation and specification. This kind of language is ideal for properties that can be easily expressed in terms of program expressions.

However, programs that manipulate *resources*, such as money, are best described in terms of how they operate on resources rather than in terms of program state. This is because properties that are implicit when reasoning about resources must be made explicit in specifications written using program expressions. As a result, specifications of resource-manipulating programs tend to be complex, are difficult to compose, and are hard for non-experts to understand.

In this thesis, we present a methodology for extending a modular program verifier to support resource reasoning in its specifications. Users can specify where resources are created and destroyed, and assert properties about the resource state. The verifier ensures that resource operations cannot fail, and that the asserted properties hold.

Our methodology does not require first-class support for resources in the source language itself, and does not impose requirements on how resources are represented in the program. Instead, users can define *coupling invariants* to relate changes in the resource state to changes in the program state; these invariants are enforced by the verifier. Coupling invariants enable simpler specifications because they enable users to describe changes in program state in terms of resource operations.

We implement our methodology as an extension to the Prusti program verifier. To evaluate our technique, we use our extended version of Prusti to verify properties of a real-world blockchain application. We show that, compared to a more-classical specification implemented without resource reasoning, our methodology enables more concise specifications.

Lay Summary

To obtain greater confidence in their code, developers can use program verifiers: software tools that check if a program satisfies user-specified properties. This process requires the developer to formally specify such properties in an unambiguous specification language.

Verifying programs that manipulate resources, such as money, is challenging because details that are implicit in human reasoning must be made explicit in such specifications. For example, while it is obvious to humans that depositing money into one's own bank account will not change the balances of other depositors' accounts, this property must be made explicit in a formal specification.

In this thesis, we present a methodology to extend a program verifier with resource reasoning capabilities, eliminating the need to reify such implicit properties in specifications. We implement our methodology as an extension to the program verifier Prusti, and show that it facilitates simpler specifications for real-world resource manipulating code.

Preface

This thesis is original work by the author, Zachary Grannan. Some text in the paper has been adapted from a conference submission [13] describing the same work, authored by Zachary Grannan and Alexander J. Summers.

Table of Contents

List of Tables

List of Figures

Acknowledgements

First of all, I would like to thank my supervisor, Alex Summers, who has taught me an incredible amount over the past two years. It is not an exaggeration to say that the feedback and advice you have given me has been the most valuable aspect of my education so far.

In addition, I would also like to thank William Bowman for reading the thesis and providing feedback.

I would also like to thank everyone in the Software Practices Lab at UBC. It is incredibly rare to have the opportunity to work alongside such dedicated, intelligent, friendly, and humorous people. I look forward to getting to know you all better in the coming years.

I would like to thank the team at Informal Systems for supporting the development of this work. In particular, I would like to thank Romain Ruetschi, for presenting the real-world verification problems that motivated the design of the work presented in this thesis, and for providing valuable feedback throughout its development.

Finally, I could not have completed this work without the support of my family, especially my wife Chiao-Ju and son Matthew. Thank you for putting up with me throughout the process, especially during the times when I have been entirely unreasonable, and for reminding me to take a break every once in a while.

Chapter 1

Introduction

The goal of program verification is to ensure that a program will always operate in accordance with a formal specification. In practice, this requires encoding the desired properties in the specification language of a program verifier that will be used to check conformance to such properties. The difference between our intuitive, human language description of desired properties and their representation in the specification language is called the *semantic gap* [14]. When this gap is small, it is easier to write specifications.

In modular program verifiers, specifications are written and checked for each function separately. Specification preconditions describe requirements on the program state at function call sites, and postconditions describe the corresponding guarantees about the program state at return sites; the verifier reasons about function calls with respect to these specifications rather than the function's implementation. For example, the specification for a function `sort()` that sorts a list in-place could be written as an expression relating the values of the input list both before and after the call; namely, stating that the latter should be a sorted version of the former. This design effectively minimises the semantic gap for properties naturally expressed as relations on program states, such as sortedness.

However, properties about programs that manipulate resources, such as money, are often phrased intuitively in terms of the creation, destruction, and transfer of resources. As these properties aren't directly connected to program state, they can only be encoded indirectly, in terms of how the resources are represented in the code. For example, an implementation of a bank for multiple accounts could represent balances using a variable `balances: Map[AccountId, u32]`. Movements of money between accounts in the bank (resource operations), would be implemented as updates to `balances`, and specifications would need to be expressed in terms of changes to `balances`, rather than directly in terms of movements of money.

As a result, there is a large semantic gap between the language used intuitively to describe resource-manipulating programs (namely, resource operations), and the specifications of program behaviour, which depend on how the resource operations are implemented in the program. The

semantic gap results in two significant issues. First, the specifications, which are written in terms of program states, do not resemble the high-level properties they intend to convey. As a result, the specifications are more difficult for non-experts to interpret.

Second, properties which are implicit in our intuitive reasoning about resources must be made explicit in formal specifications. This issue is particularly significant in the context of modular verification, where specifications for each function are checked separately. In the context of the above example, the postcondition for a function `deposit(acct, amt)` must specify not only that the balance of `acct` will increase by `amt`, but also that *the balances of other accounts remain the same*. The latter property, which specifies what parts of the program state remain unchanged after an operation, is known as a frame condition [18]. In our high-level understanding, such frame conditions do not need to be stated: we know intuitively that depositing money into one account will not change any others. However, specifications written in terms of program state must encode these conditions explicitly, as the implicit knowledge is absent.

In this thesis, we present a novel specification methodology that narrows this gap by introducing custom resources and a language of resource properties and operations as *first-class elements* of our specification language. Our methodology allows resource-related properties to be expressed directly, and eliminates the need to reify properties that are implicit in our intuitive understanding of resources, while building in ubiquitous properties by default, e.g., that the amount of a resource should remain constant unless explicitly created or destroyed.

A key aspect of our design is that specifications for resource operations in a given program are decoupled from how the program chooses to interpret resource operations in its implementation. Users can define *coupling invariants* that connect resource operations to changes to the program state. These invariants are checked by the verifier, enabling one to prove the correctness of the implementation of resource operations in the code. Coupling invariants are enforced bidirectionally: changes to resources must be accompanied by corresponding changes to the program state, and vice-versa. This coupling enables program specifications to be written in terms of resource operations, rather than in terms of program state: the change to the program state is effectively computed from the resource operations. As a result, specifications written in our methodology describe the changes to program state more concisely than those written directly in terms of program state.

We begin by introducing the relevant background in Chapter 2. In Chapter 3 we introduce our approach and demonstrate how it can simplify specifications. In Chapter 4 we explain our design. Chapter 5 describes our implementation of resource reasoning as an extension to the Rust program verifier Prusti [3]. In Chapter 6 we apply our methodology and implementation to verify a key component of a real-world Rust implementation of a cross-chain token transfer protocol. We perform a comparative evaluation of our methodology, showing that it enables users to write shorter and simpler specifications for resource-manipulating programs (Chapter 7). Chapter 8 describes related work, and we conclude in Chapter 9.

Chapter 2

Background

2.1 Rust

Rust is a modern systems programming language that features an advanced type system and other language features to facilitate high-level abstractions without sacrificing performance. In particular, Rust incorporates an ownership model that restricts what memory locations are accessible at any point in a program. The ownership model enforces that memory can only be accessed via references, which can either be *mutable* (permitting mutation to the memory at the referenced location) *immutable* (permitting reading of the memory, but not writing). Rust ensures that only a single mutable reference to a given memory location can exist at any time. These restrictions allow Rust to ensure a variety of memory safety properties statically.

2.2 Program Verification

Program verification is concerned with showing that a program behaves in accordance with a formal specification. Specifications are typically written in a separate language than the source program, although often there is a considerable overlap. A program verifier is used to determine statically whether the program upholds the specification. Fig. 2.1 shows an example of a Rust function annotated with specifications written in the syntax of the Rust program verifier Prusti [2] (discussed later in Sec. 2.4). Given the annotated program as input, Prusti would report that the function implementation satisfies the specification: all executions of this function with inputs satisfying the precondition will return a result that satisfies the postcondition.

2.3 Modular Program Verification and the Frame Problem

In modular program verification, each function in the program is verified separately with respect to its specification. This enables compositional reasoning: when analysing function calls, the verifier

```
1 #[requires(cur != u32::MAX)]
2 #[ensures(result > cur)]
3 fn next(cur: u32) -> u32 { return cur + 1; }
```

Figure 2.1: A function written in Rust, with specifications expressed as annotations in the syntax of Prusti. The annotation on line 1 specifies that the `next()` function has a precondition that the input `cur` must not equal the maximum value for the `u32` datatype. The annotation on line 2 specifies a postcondition of `next()`: the result of the function must be greater than the input `cur`.

```
1 #[requires(cur < 100)]
2 fn next2(cur: u32) -> u32 {
3     let tmp = next(cur);
4     next(tmp)
5 }
```

Figure 2.2: A Rust function that calls the function `next()` from Fig. 2.1 multiple times. When checking the body of this function, a modular verifier would consider the specification of `next()` rather than its implementation.

considers the specifications of the called functions rather than their implementations.

Modular verification enables more efficient verification, as each function implementation is only checked once. Additionally, it enables a separation between interface and implementation; for example, the author of a function may provide a less precise specification if they don't want clients to rely on the function's implementation details.

On the other hand, because the specification of a function is written once and for all, the reasoning of the verifier is more conservative: properties that hold when considering function calls with respect to their implementation are not necessarily provable when looking solely at their specifications. For example, in Fig. 2.2, the precondition to the call to `next()` on line 4 will always be satisfied, as the result of the first call to `next()` will return at most `100`, and therefore could not equal `u32::MAX`. However, the postcondition of `next()` states only that the result of the function must be greater than its input. A modular verifier would only be able to show that the variable `tmp` is some number greater than the variable `cur`, and therefore could not ensure the appropriate precondition `tmp != u32::MAX` for the call to `next()` at line 4.

One consequence of this imprecision in modular verification is the so-called *frame problem* [18], which relates to the need to specify what aspects of the overall environment remain unchanged after an action (in this case, the effect of a function call). For example, the specification for the function `erase_at()` in Fig. 2.3 indicates only that calling the function will set the element at the provided index to `0`. It does not specify that the remaining elements in the array remain the same, and therefore the verifier cannot rule out that calls to `erase_at()` would change any of the other elements in the array.

```
#[requires(index < 10)]
#[ensures(arr[index] == 0)]
fn erase_at(arr: &mut[u32;10], index: usize) { arr[index] = 0; }
```

Figure 2.3: A Rust function that sets the `index`'th element of the array `arr` to the value `0`. The specification describes this effect, but does not also specify that the other elements of the array remain unchanged.

```
#[ensures(forall(|i: usize| i != index && i < 10 ==>
  arr[i] == old(arr[i])
))]
```

Figure 2.4: An appropriate frame condition for the function `erase_at()` depicted in Fig. 2.3. The frame condition is expressed as a postcondition stating that all other elements of `arr` besides `arr[index]` remain unchanged.

To address this problem, it is necessary to add *frame conditions* that specify what parts of the state remained unchanged. An appropriate frame condition for `erase_at()` is expressed as an additional postcondition, shown in Fig. 2.4. The requirement to explicitly include such frame conditions makes specifications more complex and challenging to write.

2.4 Prusti

Prusti [2] is a modular program verifier for Rust. Function specifications can be written by adding the annotations `#[requires(...)]` and `#[ensures(...)]` to express function pre- and postconditions respectively. Prusti's specification syntax is an extension of Rust's expression language. Notably, Prusti allows for universal and existential quantifiers to appear in specifications, and also supports `old()` expressions, which can be used to refer to the previous value of an expression. `old()` expressions typically appear in function postconditions; `old(e)` refers to the value of `e` in the precondition.

A key feature of Prusti is that it leverages the guarantees provided by Rust's ownership type system to automatically generate and prove some frame conditions and other properties automatically. Prusti accomplishes this by translating the Rust program into the intermediate verification language Viper [20], and using Viper's permission-based reasoning capabilities to automatically synthesise proofs of the properties guaranteed by the borrow checker. These proofs are invisible to Prusti users, but they facilitate simpler specifications for Rust code, as demonstrated in Fig. 2.5.

Prusti's design eliminates the need to write frame conditions for invariants that are guaranteed by Rust's type system. However, users must still write higher-level frame conditions for properties that are not directly encoded in the type system, as demonstrated in Fig. 2.6.

```
1  #[requires(*x < u32::MAX)]
2  #[ensures(*x == old(*x) + 1)]
3  fn shift(x: &mut u32) {
4    *x = *x + 1;
5  }
6
7  #[requires(*x < u32::MAX)]
8  fn adjust_point(x: &mut u32, y: &mut u32){
9    shift(&mut x);
10   prusti_assert!(y == old(y));
11 }
```

Figure 2.5: Rust functions that verify with Prusti. The asserted expression on line 10 will always hold, because Rust's borrow-checker ensures that the references `x` and `y` will not alias. Prusti encodes the function `adjust_point()` as a Viper method with exclusive permission to the heap locations corresponding to `x` and `y`. Prusti can verify the assertion, because the encoded method does not give up permission to `y` in the call to `shift()`.

```
1  struct Point {x: i32, y: i32};
2
3  #[ensures(p.x == old(p.x) + s)]
4  #[ensures(p.y == old(p.y))]
5  fn shift_x(p: &mut Point, s: i32) {
6    p.x = p.x + s;
7  }
```

Figure 2.6: The `shift_x()` function from the original Prusti paper [2]. The frame condition `p.y == old(p.y)` is necessary, because Rust's type system does not preclude `shift_x()` from modifying `p.y`.

2.5 Blockchain and Smart Contracts

A blockchain is a software implementation of a distributed ledger: an append-only data structure that is replicated among multiple computers in a network, without the need for a central authority. A blockchain stores data as an ordered list of immutable *blocks*; furthermore, each block contains a list of transactions. The immutable and decentralised nature of blockchains make them useful for implementing cryptocurrencies: no centralised authority can prevent transactions from being included in the chain, and transactions that have been registered on the chain (i.e., included in a block on the chain) cannot be changed or removed.

Each block also contains metadata regarding the consensus state, which allows clients to independently verify the state of the blockchain by downloading all blocks and confirming the validity of the consensus data. Users can interact with a blockchain by running clients known as *full nodes*, which save the entire history of blocks and perform this validation.

```
interface ERC20 {
  function totalSupply() external view returns (uint256);
  function balanceOf(address who) external view returns (uint256);
  function transfer(address to, uint256 value) external returns (bool);
  ...
}
```

Figure 2.7: Key functions for Ethereum's ERC-20 interface, in the Solidity language. A user, or smart contract, can call the `transfer()` function to send tokens to another account.

However, because running a full node must maintain the entire blockchain state, it requires considerable storage space and bandwidth to operate. Therefore, users wishing to interact with the blockchain often use *light clients*, which do not require downloading the entire blockchain state. Light clients typically connect to a full node to obtain only the necessary data for a particular use case. For example, a smartphone application allowing users to check their balance and send cryptocurrencies could be implemented as a light client that only downloads the necessary blocks to obtain the transactions for a single account.

2.5.1 Smart Contracts

A smart contract is a program that is stored on a blockchain. Execution of a smart contract is performed on the blockchain; or, more precisely, by each full node involved in the consensus. As a result, smart contract behaviour must be deterministic to ensure that each node yields the same result for every computation and maintains a consistent blockchain state. This precludes smart contracts from making network requests, for example, as there is no guarantee that the response to a request will be the same for different nodes. However, smart contracts are typically allowed to maintain a limited amount of persistent state.

Many blockchain systems define their own language for smart contracts, although some allow smart contracts to be written in general-purpose programming languages such as Rust. Smart contracts define a programmatic interface that is used by users and other smart contracts. These interfaces are often standardised. For example, every token on the Ethereum blockchain is defined as smart contract that implements the ERC-20 interface (shown in part in Fig. 2.7).

A user can deploy a smart contract to a blockchain by uploading the code in a transaction. Once deployed, the code of a smart contract is typically immutable: this ensures that even the creator of the contract cannot change its behaviour. Users interact with deployed smart contracts by submitting a transaction to the blockchain. This transaction is analogous to an RPC call: it identifies which smart contract function to execute and includes data for appropriate parameters. The smart contract execution occurs when the transaction is included in a block.

Smart contracts are often used to control cryptocurrencies programmatically. Because smart

contracts are immutable, users can inspect a contract's code prior to interacting with it, and be certain that it will execute according to its implementation. On the other hand, their immutable nature also means that bugs discovered after deployment cannot be fixed. Verification of smart contracts is valuable for this reason.

2.6 The Inter-Blockchain Communication Protocol

The Inter-Blockchain Communication Protocol (IBC) [11] is a protocol for communication between different blockchains. As blockchains generally do not provide a mechanism to establish direct connections to other blockchains, the protocol relies on external *relayers*: programs that run clients for multiple blockchains, and can submit transactions to each chain.

IBC enables the development of *cross-chain* applications, which perform operations spanning multiple blockchains: These applications can take the form of smart contracts, or they can be built-in to the blockchain software implementation itself; in both cases execution occurs on the chain[1].

When an application on chain A wishes to send a message to chain B over IBC, it records on A its intention to send the message to the recipient, along with the message data. When a relayer downloads the block that includes the intention on chain A, it submits a transaction to chain B with the attached message data, along with a proof that the message originated on the other chain.

The protocol does not require that the relayer be trusted: when a chain receives a relayed message, it independently checks that the message originated from the other chain. The receiving chain accomplishes this by running a light client for the opposite chain to verify the proof attached to the relayed message. The protocol requires at least one functioning relayer to be active to ensure liveness.

One important use case for cross-chain applications is to move assets from one chain to another. It is important to ensure the correctness of such applications, because bugs in their implementations could cause users to lose funds. Furthermore, the immutable nature of blockchains means that such losses of funds usually cannot be reversed. In our evaluation (Sec. 7), we focus on verifying an implementation of a cross-chain token transfer application.

[1]The key technical distinction between the two approaches is that applications implemented as part of the blockchain can be upgraded in new versions of the full node software, while smart contracts typically cannot.

Chapter 3

Resource Reasoning

In this section, we introduce our methodology for reasoning about resource-manipulating programs. We begin by highlighting the key issues encountered when writing specifications for such programs directly in terms of program states. We then show that our methodology allows us to specify the same program properties more directly, eliminating these issues.

Throughout this section, we consider the task of writing specifications for a Rust implementation of a multi-account bank, shown in Fig. 3.1. The struct `Bank` manages the balances for multiple accounts in the field `map`, each account is identified by a corresponding `AcctId`. Clients of the `Bank` cannot access `map` directly; instead, they interact with the `Bank` object via the functions `balance()`, `deposit()`, and `withdraw()`, defined within `impl Bank {...}` block.

The specifications we want to write for `Bank` serve two purposes. First, we want to show that the implementation of `Bank` is correct, i.e., at a high-level, that the bank correctly tracks account balances and updates them in accordance with corresponding deposits and withdrawals.

Second, these specifications also serve a purpose for clients of the `Bank`, such as the `transfer()` function in Fig. 3.5. The specifications should prevent clients from interacting with the bank in disallowed ways, for example, by attempting to overdraft an account by attempting to withdraw an amount larger than the balance. Furthermore, the appropriate specifications for `Bank` are necessary for writing specifications of the client functions themselves. Intuitively, it's not possible to write a precise specification for `transfer()` if the bank functions `withdraw()` and `deposit()` are left unspecified.

Although the bank implementation we present is substantially less complex compared to what would be seen in real-world code, even for this simple example, the correct specifications using the standard approach (i.e., in terms of program states) are challenging to write correctly. In the following section, we identify the issues that arise when writing the specification.

```
type AcctId = u32;
struct Bank { map: U32Map<AcctId> };

impl Bank {
  pub fn balance(&self, acct_id: AcctId) -> u32 {
    self.map.get(acct_id).unwrap_or(0)
  }
  pub fn deposit(&mut self, acct_id: AcctId, amt: u32) {
    let bal = self.balance(acct_id);
    self.map.insert(acct_id, bal + amt);
  }
  pub fn withdraw(&mut self, acct_id: AcctId, amt: u32) {
    let bal = self.balance(acct_id);
    self.map.insert(acct_id, bal - amt);
  }
}
```

Figure 3.1: Rust code that defines a `Bank` struct and an implementation of banking operations. Account balances are stored using a `U32Map` that maps `AcctId`s to `u32` values. Deposits and withdrawals update the values in that map.

```
#[ensures(self.balance(acct_id) == old(self.balance(acct_id) + amt))]
fn deposit(&mut self, acct_id: AcctId, amt: u32) { ... }
```

Figure 3.2: An under-specified specification for the function `deposit()`, including a postcondition stating that the balance of the account `acct_id` increase by `amt`.

3.1 Three Issues of Specifying Resource-Manipulating Programs

A typical specification for the `Bank` implementation in Fig. 3.1 would define postconditions for `deposit()` and `withdraw()` in terms of how they change the result of calls to `balance()`. For example, a possible (but ultimately insufficient) postcondition for `deposit()` could specify that calling `deposit()` increases the value of `balance()` by `amt`, as shown in Fig. 3.2.

Although the postcondition always holds, it is not sufficient to describe the behaviour of `deposit()`, as demonstrated by the example in Fig. 3.3. Intuitively, we would expect that the assertion on line 6 should never fail. However, using the postcondition for `deposit()` from Fig. 3.2, Prusti would fail to verify the above code. The reason is that the postcondition only specifies that the value of `balance(acct_id)` should increase by `amt`, but does not specify that the value of `balance()` when applied to any other `AcctId` parameter must remain unchanged. Therefore, Prusti cannot prove that the call `bank.deposit(acct_id2, 10)` on line 5 does not change the balance of `acct_id1`.

The solution, therefore, is to add a frame condition to the postcondition, specifying that the balance of all other accounts remain unchanged. The `withdraw()` function also requires similar frame conditions. A correct such specification for `deposit()`, and the analogous specification for

```
1 let acct_id1 = AcctId(1);
2 let acct_id2 = AcctId(2);
3 let init_balance = bank.balance(acct_id1);
4 bank.deposit(acct_id1, 10);
5 bank.deposit(acct_id2, 10);
6 assert!(bank.balance(acct_id1) == init_balance + 10);
```

Figure 3.3: Rust code that deposits $10 into two distinct accounts. Using the specification of `deposit()` from Fig. 3.2, Prusti would not be able to prove that the assertion on line 6 should always hold.

```
#[ensures(
  forall(|acct_id2: AcctId| if acct_id == acct_id2 {
    self.balance(acct_id) == old(self.balance(acct_id)) + amt
  } else {
    self.balance(acct_id2) == old(self.balance(acct_id2))
  })
)]
fn deposit(&mut self, acct_id: AcctId, amt: u32) { ... }

#[requires(self.balance(acct_id) >= amt)]
#[ensures(
  forall(|acct_id2: AcctId| if acct_id == acct_id2 {
    self.balance(acct_id) == old(self.balance(acct_id)) - amt
  } else {
    self.balance(acct_id2) == old(self.balance(acct_id2))
  })
)]
fn withdraw(&mut self, acct_id: AcctId, amt: u32) { ... }
```

Figure 3.4: Specifications for the `deposit()` and `withdraw()` functions. Each specification includes a frame condition indicating the value of `balance()` for all other account identifiers remains unchanged; specification lines used to encode the frame condition are highlighted in red.

`withdraw()`, are presented in Fig. 3.4. Adding frame conditions makes the resulting specifications more complex. Most of the lines of specification code for the two functions are concerned with describing what remains unchanged rather than what does change.

If such frame conditions were only necessary for the `Bank`'s associated functions `deposit()` and `withdraw()`, i.e., if they could be written once and for all, the situation would not be that bad. Unfortunately, this is not the case: similar frame conditions are also necessary to write precise specifications for clients of the `Bank`, such as the `transfer()` function in Fig. 3.5. The postcondition for `transfer()` must also specify that the balances of accounts other than `from` and `to` are preserved.

Ideally, the specification for `transfer()` could be derived by conjoining the postconditions of `withdraw()` and `deposit()`; however, the resulting postcondition is not valid. The postcondi-

```
fn transfer(bank: &mut Bank, from: AcctId, to: AcctId, amt: u32) {
  bank.withdraw(from, amt);
  bank.deposit(to, amt);
}
```

Figure 3.5: A client function of `Bank` that moves a specified amount of funds from one account to another.

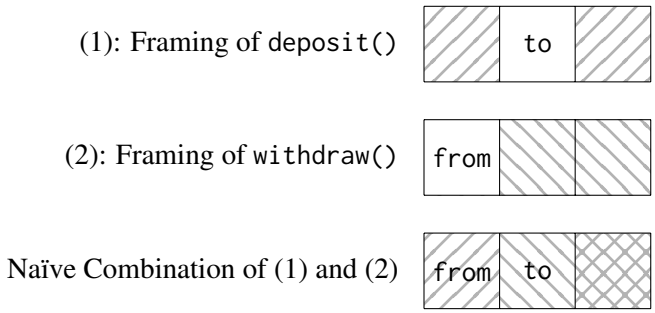


Figure 3.6: A graphical representation of the frame conditions for functions `deposit()` and `withdraw()`, as well as the resulting frame condition that would be obtained by directly conjoining the Prusti expressions used to define them. Each box indicates a distinct account, boxes with lines in the background indicate that the account should remain unchanged. The naïve combination of frame conditions for `deposit()` and `withdraw()` results in all accounts being unchanged.

tion of `withdraw()` requires that all accounts besides `from` remain unchanged; the postcondition of `deposit()` requires all accounts besides `to` to be unchanged. When `from` and `to` are distinct, the conjunction of these postconditions would require all balances to be unchanged, shown graphically in Fig. 3.6. This would not be a valid postcondition for `transfer()`, which has the effect of changing the balances of accounts `from` and `to`.

As written, the postconditions for `deposit()` and `withdraw()` cannot be directly combined, because the frame conditions for each function do not compose. The appropriate postcondition of `transfer()` is presented in Fig. 3.7. The postcondition does not directly use the specification expressions from `deposit()` or `withdraw()`, because their frame conditions are overly restrictive.

This demonstrates the first issue:

Issue 1: Specifying resource operations using program states requires frame conditions

In the case of `transfer()`, we cannot directly write a specification by composing the specifications of `deposit()` and `withdraw()` due to their frame conditions. However, composability issues also arise even without consideration of frame conditions, as demonstrated by the code in Fig. 3.8, which performs two deposit operations in sequence.

In this case, additional complexity arises because the expression describing the resulting program state must separately consider the cases when the two account identifiers passed to `deposit2()`

```
#[requires(from != to && bank.balance(from) >= amt)]
#[ensures(forall(|acct_id: AcctId|
  if acct_id == from {
    bank.balance(acct_id) == old(bank.balance(acct_id)) - amt
  } else if acct_id == to {
    bank.balance(acct_id) == old(bank.balance(acct_id)) + amt
  } else {
    bank.balance(acct_id) == old(bank.balance(acct_id))
  }
))]
fn transfer(bank: &mut Bank, from: AcctId, to: AcctId, amt: u32) { ... }
```

Figure 3.7: The specification for the `transfer()` function, which makes calls to `withdraw()` and `deposit()` in sequence. The postcondition of this function does not directly incorporate the postconditions of `withdraw()` and `deposit()`.

```
#[ensures(if(acct_id1 == acct_id2) {
  bank.balance(acct_id1) == old(bank.balance(acct_id1) + 3)
} else {
  bank.balance(acct_id1) == old(bank.balance(acct_id1) + 1) &&
  bank.balance(acct_id2) == old(bank.balance(acct_id2) + 2)
} && forall(|acct_id: AcctId| (acct_id != acct_id1 && acct_id != acct_id2) ==>
    bank.balance(acct_id) == old(bank.balance(acct_id))
))]
fn deposit2(bank: &mut Bank, acct_id1: AcctId, acct_id2: AcctId) {
  bank.deposit(acct_id1, 1);
  bank.deposit(acct_id2, 2);
}
```

Figure 3.8: The implementation and Prusti specification for a function that performs two deposit operations in sequence. When `acct_id1` and `acct_id2` are the same, this function performs two deposit operations on the same account.

refer to the same account, or different accounts. When `acct_id1` and `acct_id2` are the same, the result of `balance()` for the referenced account increases by three, otherwise, the result for `acct_id1` increases by one and `acct_id2` by two. Therefore, we cannot write the appropriate specification simply by composing the postconditions of `deposit(acct_id1, 1)` and `deposit(acct_id2, 2)`. A function depositing to three accounts would be substantially more challenging to specify, as the change to values of `balance()` differs depending on the aliasing between account identifiers.

The example presented in Fig. 3.7 demonstrate the second issue:

Issue 2: Specifications of resource operations using program states do not easily compose

Specifications of the `Bank`'s associated functions also dictate how clients are allowed to interact with the bank. For example, `withdraw()`'s precondition `self.balance(acct_id) >= amt` of

```
1 fn merge(bank: &mut Bank, source: AcctId, dest: AcctId) {
2   bank.deposit(dest, bank.get_balance(source));
3   bank.withdraw(source, bank.get_balance(source));
4 }
```

Figure 3.9: A function intended to move funds from account `source` into account `dest`. When `source` and dest refer to the same account, this function has the unintended effect of removing all funds from `source`.

the `withdraw()` prevents client functions from overdrafting an account. However, specifications written in terms of program states cannot reliably prevent all kinds of unintended interactions with the Bank. This is demonstrated by the buggy function `merge()` in Fig. 3.9, that intends to merge two accounts by moving all funds from one account to another.

When `source` and `dest` refer to distinct accounts, this function behaves correctly. However, in the case where `source` and `dest` refer to the same account, `merge()` would remove all funds from the account: the call to `deposit()` on line 2 increases the balance of `source`, but the entire balance of `source` is withdrawn on the following line. The specifications for `withdraw()` and `deposit()` in Fig. 3.4 would permit this function to be written: note that `deposit()` has no precondition, and the precondition for `withdraw()` is trivially satisfied.

Issue 3: Specifications using program states cannot easily enforce resource properties

We now show how our methodology addresses all three issues.

3.2 Simplifying Specifications with Resources

The key to simplifying specifications for the `Bank` is to describe functions in terms of their operations on resources, rather than their changes to the program state. However, although we are considering code that handles money, money is not represented as a first-class notion in the implementation. For example, the interface of the `Bank` itself provides no indication that the `u32` parameter `amt` of `deposit()` and `withdraw()` should be treated as a monetary value.

Instead, our methodology provides new features that allow users to *reify* resources, and operations on these resources, as part of the specifications. In the remainder of this section, we demonstrate how we use our methodology to write a specification of the `Bank` implementation.

3.2.1 Representing Resources in Specifications

The core idea of our approach is to enable program behaviour to be specified directly in terms of notions of resource relevant to the program. *Resources* in our methodology are conceptually a pair of a *resource type* and a *resource amount* (an integer). Resource types are organised into

parameterised *resource kind*s: a named *resource constructor* with fixed arity (possibly zero) and corresponding parameter types; each instantiation of these parameters yields a distinct *resource type*. For example, we might use a resource kind `Money` with a single parameter representing an account ID to represent currency known to be deposited in the corresponding bank account.

In our Prusti implementation, such a resource kind is declared as a Rust struct annotated with a `#[resource_kind]` tag, listing its parameter types; resource types are denoted by instantiating the struct with appropriate Rust expressions. For example, we can declare a resource constructor for the amount of money belonging to a particular account as follows:

```
#[resource_kind] struct Money(AcctId);
```

Resources of the same type are always aggregated to the sum of their amounts, and resources of different types are incomparable. In our example, the fact that for two *different* account IDs `id1` and `id2`, the corresponding resource types `Money(id1)` and `Money(id2)` are different encodes the property that money in different accounts should not be fungible; withdrawing money from your account should have a meaning distinct from withdrawing from someone else's!

Resource amounts are represented in our methodology as rational numbers. Inside of specifications, types that can be converted into rational numbers (for example, `u32` types) can be used as resource amounts. To refer to a resource in our specifications, we use the syntax `resource(`*rtype*`,` *amt*`)`, where *rtype* is a resource type, and *amt* is the amount. For example, `resource(Money(a_id), amt)` refers to an amount `amt` of money for the account associated with the identifier `a_id`.

3.2.2 Writing Specifications with Resource Operations

Our methodology supports two *resource operations*: *creation* and *destruction*; each operation is associated with a resource. Our specifications should reflect that, as far as `Bank` is concerned, a deposit corresponds to the creation of money, and a withdrawal corresponds to its destruction. We can express the former property by adding the specification macro `produce!(resource(Money(acct_id), amt))` to the body of the `deposit()` function, and the latter by using an `consume!()` macro with the same resource argument in the `withdraw()` function.

Intuitively, it is only possible to destroy a resource if it exists. Therefore, our methodology requires that the resource `resource(Money(acct_id), amt)` is available to destroy at the point of the `consume!()` call inside `withdraw()`. We can ensure this by requiring the resource inside the function precondition. Analogously, we also specify that `deposit()` makes this resource available by expressing it as a postcondition. Fig. 3.10 presents a version of the `Bank` implementation with the appropriate resource operations.

```
impl Bank {
  #[ensures(resource(Money(acct_id), amt))]
  fn deposit(&mut self, acct_id: AcctId, amt: u32) {
    ...
    produce!(resource(Money(acct_id), amt));
  }

  #[requires(resource(Money(acct_id), amt))]
  fn withdraw(&mut self, acct_id: AcctId, amt: u32) {
    ...
    consume!(resource(Money(acct_id), amt));
  }
}
```

Figure 3.10: The specification of `Bank`, described in terms of the resource `Money`.

3.2.3 Verification and Resource State

We now present how these specifications are interpreted inside our methodology. During verification, our methodology considers a hypothetical *resource state*, which maps resource types to amounts. Conceptually, creating a resource updates the entry corresponding to its type by adding the appropriate amount, and destroying a resource has the opposite effect.

Each function is verified separately; the initial resource state consists of the resources made available in the precondition. At function calls, the resources required by the function's precondition are removed from the resource state, and the resources returned by the postcondition are added. The verifier ensures that all amounts in the resource state are positive at every point in the body of the function, and that at the return of the function, there are sufficient resources available in the resource state to satisfy the resources that the function makes available via its postcondition.

3.2.4 Connecting Resource and Program State

The specifications introduced so far suffice to characterise the behaviour of the program in terms of its resource operations. Reifying these resource transfers makes certain properties of the program more apparent. For example, it is impossible to overdraft an account because bank withdrawals require a resource that can only be acquired from a corresponding deposit.

However, we have not yet specified any functional properties related to the `Bank`. To do so, we define how changes in the resource state correspond to changes in the program state. For the bank, this means that when a `Money` resource is created or destroyed, there should be a corresponding change in the value of `bank.balance()`.

Writing such a specification requires introspection on the amount of resource in the resource state. This can be done using the expression `holds(`r`)`, which returns the amount of resource type r that is present in the state.

```
#[invariant_twostate(forall(|a_id: AcctId|
  holds(Money(a_id)) - old(holds(Money(a_id))) ==
  PermAmount::from(self.balance(a_id)) - PermAmount::from(old(self.balance(a_id)))
))]
```

Figure 3.11: A two-state invariant for the `Bank` struct. This invariant is enforced at the end of each function involving a `Bank` object; expressions wrapped in `old()` are interpreted using their value at the beginning of the function, and the function `PermAmount::from()` explicitly converts an Rust integer type to a rational number. The invariant ensures that changes to the `Money` resource are reflected as corresponding changes in the value of `balance()`.

We establish a connection between resource and program state by asserting that the difference in `balance()` between the start and end of each function involving `Bank` is the same as the difference in `holds()` expressions for the corresponding `Money` resource. We express this assertion using a *two-state invariant* on the `Bank` struct, shown in Fig. 3.11.

The two-state invariant is asserted at the end of every function involving the `Bank`. To demonstrate the effect of this invariant, we can consider how it applies to the `deposit()` function (depicted graphically in Fig. 3.12). The expression `old(holds(Money(a_id)))` refers to the amount of the resource type `Money(a_id)` present in the resource state at the beginning of the `deposit()` function. As `deposit()` does not require any resources in its precondition, the initial resource state is empty, and therefore this value is `0` for all values of `a_id`.

The `deposit()` function produces the resource `resource(Money(acct_id), amt)`. Hence, we have `holds(Money(acct_id) == PermAmount::from(amt)` and `holds(Money(a_id) == PermAmount::from(0)` for all other account identifiers `a_id`. Therefore, the effect of invariant is to assert that after calling `deposit(acct_id, amt)`, the value of `balance(acct_id)` increases by `amt` relative to its previous value, because `holds(Money(acct_id)) - old(holds(Money(acct_id)))` `== amt`. The invariant also enforces that the value of `balance()` for other accounts remain unchanged, as `deposit()` does not produce or consume any other resource. In other words, the effect of the invariant is to specify exactly the same postcondition that had to be written manually for `deposit()` in Fig. 3.4.

In summary, the two-state invariant specifies that the implementation of `Bank` updates `balance()` appropriately when resource operations occur. Therefore, there is no need to explicitly describe the changes to `balance()` in specifications, and therefore the complex specifications from Fig. 3.4 are unnecessary.

The two-state invariant also applies to client functions of the bank, allowing us to produce simplified specifications for functions such as `transfer()`. Fig. 3.13 shows the specifications for the `transfer()` function implemented using our methodology; unlike the original version presented in Fig. 3.7, the specification does not require the user to define frame conditions for this function.

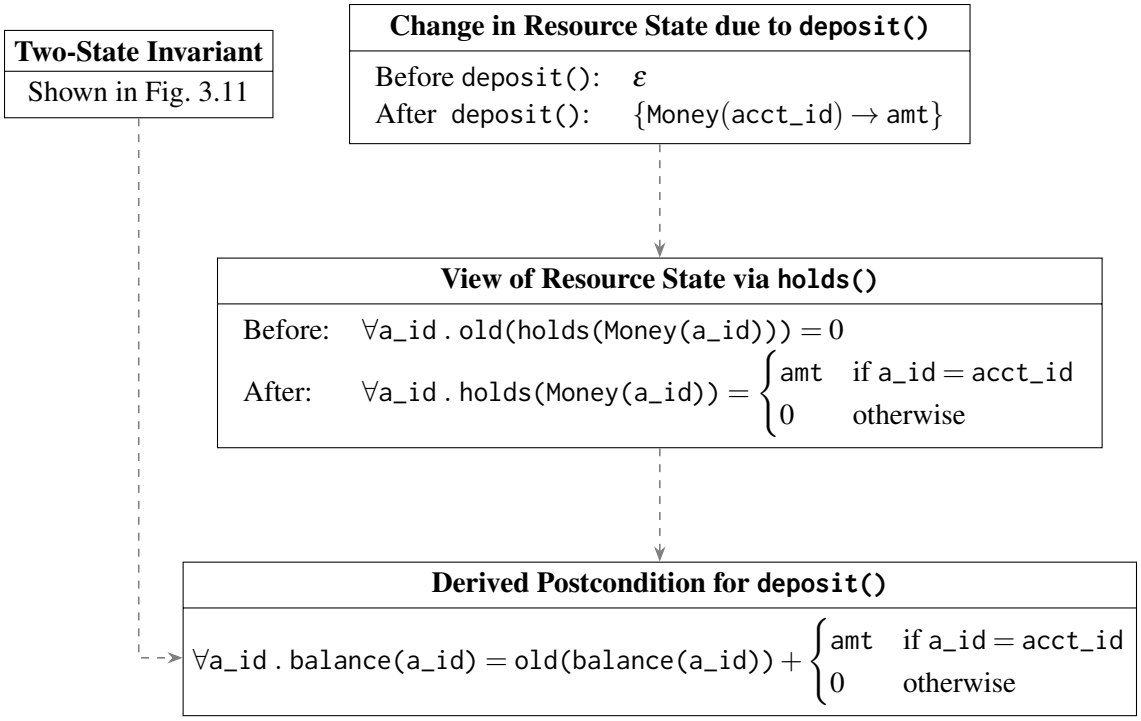


Figure 3.12: A diagram indicating how the two-state invariant from Fig. 3.11 combines with resource operations of `deposit()` (Fig. 3.10) to derive a postcondition specifying the effect `deposit()` has on the value of `balance()`.

```
#[requires(from != to)]
#[requires(resource(Money(from), amt))]
#[ensures(resource(Money(to), amt))]
fn transfer(bank: &mut Bank, from: AcctId, to: AcctId, amt: u32) { ... }
```

Figure 3.13: The specification of `transfer()`, described in terms of the resource `Money`.

Furthermore, the specification for `transfers()` (with the exception of the extra precondition `from != to`) is a simple composition of the specifications of `withdraw()` and `deposit()` from Fig. 3.10.

Overall, our methodology enables simpler specifications, as demonstrated by the comparison of `Bank` specifications in Fig. 3.14: specifications written using resource operations substantially shorter and more directly convey the intended behaviour of the functions they describe.

```
impl Bank {
  #[ensures(
    forall(|acct_id2: AcctId| if acct_id == acct_id2 {
      self.balance(acct_id) == old(self.balance(acct_id)) + amt
    } else {
      self.balance(acct_id2) == old(self.balance(acct_id2))
    })
  )]
  pub fn deposit(&mut self, acct_id: AcctId, amt: u32) { ... }

  #[requires(self.balance(acct_id) >= amt)]
  #[ensures(
    forall(|acct_id2: AcctId| if acct_id == acct_id2 {
      self.balance(acct_id) == old(self.balance(acct_id)) - amt
    } else {
      self.balance(acct_id2) == old(self.balance(acct_id2))
    })
  )]
  pub fn withdraw(&mut self, acct_id: AcctId, amt: u32) { ... }
}
```

(a) Specifications of Bank expressed using program states

```
impl Bank {
  #[ensures(resource(Money(acct_id), amt))]
  pub fn deposit(&mut self, acct_id: AcctId, amt: u32) { ... }

  #[requires(resource(Money(acct_id), amt))]
  pub fn withdraw(&mut self, acct_id: AcctId, amt: u32) { ... }
}
```

(b) Specifications of Bank written using our methodology

Figure 3.14: A comparison of the specifications in the Bank using our methodology, and without using our methodology

Chapter 4

Design

The core idea of our approach is to enable program specifications to be expressed in terms of resource operations. To write such specifications, the user must not only learn the specification language, but also to understand the semantics of resources. Therefore, a key consideration for our design is to ensure the language and semantics of resources is easy to learn and also appropriate for specifying the kind of properties that users care about, for example, that the program correctly tracks the amount of real-world resources. In particular, our design must make it possible to relate resource operations to program state, which is necessary to specify correctness properties about the program's implementation of resource operations.

We also want to ensure that our approach is sufficiently general, so that it can be applied to real-world programs and programming languages. Therefore, we aim for a design that does not rely on first-class support for resources in the source language. For example, Rust's ownership model allows values to have some resource-like properties: by default, values of user-defined `struct` types cannot be duplicated. The Rust library even takes advantage of these properties to ensure, for example, that file handles (which are represented using Rust values) are automatically closed once the handle is discarded. However, requiring that real-world resources be represented using resource-like type in Rust code would restrict the applicability of our methodology. For example, it could not be used to specify the bank interface in Fig. 3.1, as the monetary amounts are represented in the implementation as values of the primitive type `u32`, which are allowed to be duplicated.

Therefore, we opted for a design that did not require support for resources in the source language. In the remainder of this chapter, we present the design we came up with, based on the above criteria.

4.1 The Semantics of Resources

Because most general-purpose programming languages do not have first-class resources; our design develops a resource semantics for use in specifications that does not require resource support in

the source language. As a consequence, the meaning of specifications depends not only on the semantics of the source language, but also the resource semantics.

4.1.1 Resource Types and Amounts

In the semantics, we describe a resource by two properties: the *type* of the resource, and its amount. Resource types are instantiations of fixed-arity *resource constructors* applied to zero or more parameters; instantiations of different constructors or different parameter values yield different resource types. Resource types instantiated via the same resource constructor belong to the same *rkind*. For example, a resource constructor `Stock`, applied to a string value, would yield a resource type referring to stock for a given company. Stocks for different companies would have different types but share the same kind.

The motivation for this decomposition is to enable intuitive reasoning about the combination of multiple resources, by aggregating the resources by their type. For example, suppose you have two apples and receive one fruit. If the type of that fruit happens to be an apple, then you have three apples, however if that fruit happens to be an orange, then you have two apples and one orange. In other words, resources of the same type are fungible.

The resource type can also be used to represent *real-world* ownership. For example, a bank application could consider the money in different accounts to be of different types, by declaring a resource constructor that takes an account identifier as a parameter. This encoding reflects the property that the money in different accounts should not be directly fungible: withdrawing money from an ATM should certainly move money out of your account, rather than the account of someone else.

Resource amounts are represented in our semantics as symbolic rational numbers, and resources of the same type are aggregated by taking the sum of their amounts. We make this choice because rational numbers are sufficient to represent amounts of real-world resources, and verifiers (especially those based on SMT solvers) typically support reasoning about rational numbers. In principle, it could be possible to consider a more general representation of resource amounts, allowing them to be encoded as an arbitrary type associated with a composition operation, where the type and operation define a *permission algebra* [8], i.e., the operation is associative, commutative, and cancellative. However, we have not considered implementing a semantics with this more general resource model, as it requires encoding user-defined composition operations into the underlying program logic.

4.1.2 Resource State

The *resource state* is a symbolic map from resource types to resource amounts, representing the resources available at a given point in the program. Our semantics does not treat the resource state as a single globally accessible map, however. Instead, we consider each stack frame to have its own resource state, with resources transferring between frames at call sites and return sites. The state of

each frame is initialised with only the resources passed to it from its caller. This implementation of resource state is more suitable for modular verification: each function can be checked independently, because it always considers the same initial resource state.

4.1.3 Resource Operations

Our resource semantics defines two *resource operations*: *creation* and *destruction*. The most basic usage of these operations is to create or destroy an amount of a particular type of resource. These operations can also denote the conditional creation / destruction of a resource, only having an effect when a certain property holds.

In our methodology, the verifier interprets a creation operation as an action that increases the amount of the resource in the resource state for the corresponding resource type; destruction operations have the opposite effect. Resource operations are checked by the verifier to ensure that they will not result in negative amounts for any resource kind in the resource state (which is only possible for destruction operations). If the verifier cannot prove that a destruction operation will maintain this invariant, then a verification error is raised.

4.1.4 Resource State Expressions

An important aspect of our methodology is the ability to introspect on the resource state; this functionality is key to connecting resource and program state. Our methodology accomplishes this via *resource state expressions*, expressions that refer to the amount of a particular resource type present in the resource state (in our Prusti implementation, these take the form of `holds()` expressions). Resource state expressions are treated the same way as standard program expressions inside specifications. They can also be combined with standard program expressions, which is the key to their use in establishing coupling invariants, as described later in Sec. 4.2.

4.1.5 Using Resources in Function Specifications

When `holds()` is used in function specifications, we define its semantics differently, considering the fact that the caller and callee resource states may differ. In such positions, our technique interprets `holds()` expressions with respect to the *amount of resource transferred so far* by the corresponding pre-/post-condition; this notion has the same meaning for both caller and callee.

Fig. 4.1 illustrates this semantics. The first precondition `#[requires(resource(Money(a), 1))]` at line 1 specifies a resource to be transferred from caller to callee. As that is the only resource transferred so far, the (commented) assertion `holds(Money(a)) == 1` holds at ①. The subsequent line specifies the same resource transfer again, resulting in `holds(Money(a)) == 2` (②).

The first postcondition `old(holds(Money(a))) == 2)` at ③ concerns (via `old`) the amount of `Money(a)` transferred *from the caller* (this refers to the overall transfers made by the preconditions

```
#[requires(resource(Money(a), 1))] // holds(Money(a)) == 1 ①
#[requires(resource(Money(a), 1))] // holds(Money(a)) == 2 ②
#[ensures(old(holds(Money(a))) == 2)]                      ③
#[ensures(holds(Money(a)) == 0)]                           ④
#[ensures(resource(Money(a), 1))] // holds(Money(a)) == 1  ⑤
fn take2return1(bank: &mut Bank, a:AcctId){ bank.withdraw(a, 1); }

#[requires(resource(Money(a),3))]
fn client(bank: &mut Bank, a: AcctId){ take2return1(bank, a); }
```

Figure 4.1: A program demonstrating resource operations and resource state expressions in specifications.

together), while the second (④) states that no money has been transferred by the *postconditions* up to this point. At ⑤, located after the postcondition transferring `resource(Money(a),1)` to the caller, the commented assertion `holds(Money(a)) == 1` would instead be true.

4.2 Connecting Resource and Program State

So far, our semantics have primarily been concerned with the resource state, a concept that exists only in the verifier. At runtime, there is no resource state, resource operations have no effect, and resource state expressions are undefined.

At the same time, we often wish to show that the resource operations described in the specifications (which have no inherent connection to any program behaviour) change the program state in a particular way. For example, when a bank registers a deposit or a withdrawal (a resource operation), it should update the user's balance (a property of the program state). The relationship between the balance and resource operations is bidirectional: each balance update should correspond to a resource operation, and (non-trivial) resource operations should result in a change to the balance[1]. If one were to happen without the other, there is likely a bug in the program: the verifier should enforce this relationship so that such bugs cannot occur.

To enforce such relationships, we allow users to establish *coupling invariants*, that connect *changes* in resource state to changes in the program state. Taking this approach requires deciding where the coupling invariant should be enforced. Enforcing the invariant at every pair of program points could be overly restrictive because updates to resource and program state do not occur in a single command. More generally, updating the program state after a resource operation could require multiple instructions and function calls, during which time the invariant would not hold.

To permit this flexibility, we choose to enforce invariants at the end of a function execution, interpreting invariants as additional, implicit postconditions. The resource and program state are allowed to be out of sync (with respect to the invariants) within the body of the function, but must

[1]Examples of trivial operations include a transfer of amount `0`, and a transfer conditional on the expression `false`

be in sync by the end of the function. This treatment of invariants ensures that calls to other functions within the body, however, update the program state in accordance to their own resource operations. Therefore, functions only need to maintain the invariants related to their own resource operations, and not those of the functions they call.

In the following section, we present the implementation of our design as an extension to the program verifier Prusti.

Chapter 5

Implementation

We implemented our verification technique in the Rust program verifier Prusti. Prusti verifies Rust code by encoding the program and specifications into the intermediate verification language Viper [20], which supports permission-based reasoning using a variant of implicit dynamic frames [24]. Although Prusti does not allow users to directly access Viper's permission primitives in specifications, it uses Viper permissions to encode the guarantees provided by Rust's ownership model. Our methodology uses Viper's permission-based reasoning capabilities to extend Prusti with support for resource reasoning.

We begin this chapter with a high-level overview of the Viper language (Sec. 5.1), and Prusti's encoding of Rust programs and specifications into Viper (Sec. 5.2). We then describe how we extended Prusti to support resource reasoning (Sec. 5.3) and coupling invariants (Sec. 5.4).

5.1 The Viper Intermediate Verification Language

Viper is an imperative intermediate language, in the spirit of Why3 [5] and Boogie [16]. Program verifiers can translate programs and specifications into Viper, and the encoded specifications can then be checked by a Viper verifier.

In this section, we provide an overview of a subset of the Viper language, the syntax of which is presented in Fig. 5.1.

5.1.1 Data Types and Pure Expressions

Viper is statically typed and provides built-in datatypes such as booleans (`Bool`), mathematical integers (`Int`), and rational numbers (`Perm`). Viper has two different kinds of expressions: *pure expressions*, which are similar to standard programming language expressions, and *impure expressions*, which we describe in Sec. 5.1.3.

Variables and constants are pure expressions in Viper. Pure expressions are also used to represent standard operations such as integer arithmetic and boolean negation (these are omitted in

T	::=		**Type** (Sec. 5.1.1)
	|	`Int`	*mathematical integers*
	|	`Bool`	*booleans*
	|	`Perm`	*rational numbers*
	|	...	
e	::=		**Pure Expression** (Sec. 5.1.1)
	|	x	*variable*
	|	`true | false | 0 | 1 | ...`	*constant*
	|	`old[`l`](`e`)`	*old expression*
	|	`perm(`$P(e_1,\ldots,e_n)$`)`	`perm()` *expression* (Sec. 5.1.5)
	|	...	
I	::=		**Impure Expression** (Sec. 5.1.3)
	|	`acc(`$P(e_1,\ldots,e_n)$`, `e`)`	*accessibility predicate*
	|	I `&&` I	*conjunction*
	|	e `=>` I	*implication*
	|	`forall` x `:` T `::` I	*quantified permission*
	|	e	*pure expression*
s	::=		**Statement** (Sec. 5.1.2)
	|	`var` x	*variable declaration*
	|	x `:=` e	*assignment*
	|	`assert` e	*assertion*
	|	`assume` e	*assumption*
	|	`label` l	*label*
	|	`inhale` I	*inhale* (Sec. 5.1.4)
	|	`exhale` I	*exhale* (Sec. 5.1.4)
	|	...	
d	::=		**Declaration**
	|	`method` $p(x_1,\ldots,x_n)$ $\{s_1,\ldots,s_m\}$	*method* (Sec. 5.1.2)
	|	`predicate` $P(x_1:T_1,\ldots,x_n:T_n)$	*abstract predicate* (Sec. 5.1.3)
	|	...	

Figure 5.1: A subset of the Viper intermediate language

Fig. 5.1 for brevity). `old[`l`](`e`)` expressions refer to the value of an expression e at a previous point l in the program. `perm` expressions are described in Sec. 5.1.5.

5.1.2 Methods and Statements

Viper methods are similar to functions, or static methods, in an imperative language. Methods contain a list of variable parameters, and their body is a list of statements. Viper methods can also include pre- and postconditions, however as their behaviour is not relevant to our implementation we do not consider them further[1].

[1] Sec. 5.2 describes how Prusti encodes pre- and postconditions of Rust function into Viper without these constructs

Viper supports statements for variable declarations and assignments, as well as `assert` and `assume` statements. The statement `label` l defines a program point l, which can be referenced in `old[`l`](`e`)` expressions to refer to the value of a Viper expression at that point inside a method. `exhale` and `inhale` statements (described in detail in Sec. 5.1.4) are used with impure expressions, which are described in the following section.

5.1.3 Impure Expressions and Abstract Predicates

Abstract predicates are used in Viper's permission model: intuitively, every point in a Viper method is associated with some (possibly zero) amount of *permission* for every abstract predicate. An abstract predicate is defined using the syntax `predicate` $P(x_1 : T_1, \ldots, x_n : T_n)$, where P is a string identifier, and $x_1 : T_1, \ldots, x_n : T_n$ are the names and types of the arguments used to instantiate the predicate. A predicate is instantiated by applying the constructor P to expressions of the appropriate types.

The Viper statements `inhale` I and `exhale` I (discussed further in Sec. 5.1.4) update the permissions to predicates at the point they appear in a method by adding or removing permissions respectively. The parameter I is an impure expression that describes what permissions should be added or removed. Impure expressions are treated differently than pure expressions, for example, they cannot be used as parameters to a method or function or assigned to variables.

The simplest impure expression is an *accessibility predicate*, written `acc`$(P(e_1, \ldots, e_n), e)$. The expression denotes permission e to the predicate $P(e_1, \ldots, e_n)$, where e is an expression of type `Perm`, the type of rational numbers.

Impure expressions can be combined using the operator `&&`, which is similar to the separating conjunction $*$ in separation logic. For example, the expression `acc`(p_1, e_1) `&&` `acc`(p_2, e_2) describes an amount e_1 of the predicate p_1 and a *separate* amount e_2 of the predicate p_2. In particular, when $p_1 = p_2$, the expression is equivalent to `acc`$(p_1, e_1 + e_2)$.

Impure expressions can also be made conditional using the implication operator `=>`. The expression b `=>` `acc`(p, e) describes the amount e of the predicate p when b is `true` (and describes an amount 0 of that predicate otherwise).

Finally, Viper also supports *quantified permissions*, which allows using quantifiers inside impure expressions. For example, the expression `forall` $x : T ::$ `acc`$(P(x), a)$ denotes the amount a of the predicate $P(x)$ for every value of x with type T.

Pure, `Bool`-typed expressions can also be used as impure expressions. Their interpretation in the context of `inhale` and `exhale` statements are discussed in the following section.

5.1.4 `inhale` and `exhale` statements

The statements `inhale` and `exhale` take an impure expression I as input, and update the permissions to predicates at the point they appear in a method by adding or removing the permissions described

```
1 predicate p(x: Int)
2 method example(x: Int){
3   inhale acc(p(x), 9 / 1)
4   exhale acc(p(x), 6 / 1)
5   exhale acc(p(x), 5 / 1) // ERROR: Exhale might fail
6 }
```

Figure 5.2: An example demonstrating the behaviour of `inhale` and `exhale` in Viper. The statement at line 5 causes Viper to raise an error, because there is only three permission to `p(x)` at that point in the program.

```
1   predicate p(x: Int)
2   method unsound(x: Int) {
3     assume perm(p(x)) == 1 / 1
4     assert false
5   }
```

```
1   predicate p(x: Int)
2   method possible(x: Int) {
3     inhale acc(p(x), 1/1)
4     assume perm(p(17)) == 1 / 1
5     assert x == 17
6   }
```

Figure 5.3: Examples demonstrating the behaviour of `perm()` expressions; both programs verify in Viper.

by I. The verifier checks that `exhale` statements will not fail, i.e., that sufficient permissions exist at the point of the `exhale` statement. Fig. 5.2 shows an example of the usage of these statements in Viper.

When either statement is applied to a term containing pure expressions (i.e. those without any accessibility predicates) `inhale` has the effect of *assuming* these expressions, `exhale` will *assert* these expressions. Effectively, this means that `inhale` and `exhale` are equivalent in meaning to `assume` and `assert` respectively when applied to pure expressions.

5.1.5 `perm` expressions

`perm` expressions allow introspection on the current permissions to a predicate: the expression `perm(`p`)` returns a value of type `Perm` (the type of rational numbers), indicating the current permission held to p. `perm` expressions are pure expressions; like other pure expressions, they can be used as part of `assume` and `assert` statements, but cannot change the state. The example in Fig. 5.3 demonstrates the usage of `perm` expressions.

Prior to the `assume` statement on line 3 for the left program, there is no permission to any predicate. The statement on line 3 assumes that there exists `1 / 1` permission to the predicate `p(x)`, which is impossible. Therefore, the statement on line 3 is equivalent to `assume false`. The `assume` statement on line 4 for the figure on the right assumes that there is `1 / 1` permission to `p(17)`. At that point in the program, there is only permission to `p(x)`. Therefore, the assume statement on line

$$\left[\!\left[\begin{array}{l}\texttt{\#[requires}(f_{pre}\texttt{)]}\\ \texttt{\#[ensures}(f_{post}\texttt{)]}\\ \texttt{fn } f(x_1 : T_1, \ldots, x_n : T_n)\ \{s_1; \ldots; s_m\}\end{array}\right]\!\right] \rightsquigarrow \begin{array}{l}\texttt{method } f(x_1 : [\![T_1]\!], \ldots, x_n : [\![T_n]\!])\{\\ \quad \texttt{inhale } [\![f_{pre}]\!];\ [\![s_1]\!]; \ldots; [\![s_m]\!];\ \texttt{exhale } [\![f_{post}]\!]\\ \}\end{array}$$

$$[\![f(t_1, \ldots, t_n);]\!] \rightsquigarrow \begin{array}{l}\texttt{exhale } [\![f_{pre}[x_1 := t_1, \ldots, x_n := t_n]]\!]\\ \texttt{inhale } [\![f_{post}[x_1 := t_1, \ldots, x_n := t_n]]\!]\end{array}$$

Figure 5.4: Prusti's translation rules for function definitions and function calls into Viper. The pattern $t[x_1 := t_1, \ldots, x_n := t_n]$ denotes the substitution of each variable x_i with the expression t_i in the Prusti expression t

4 is equivalent to `assume x == 17`.

5.2 Prusti's Viper Encoding

Having provided the relevant background of Viper, we now briefly describe how Prusti encodes a Rust program and its accompanying specifications into Viper. For the sake of clarity, we present a simplified description of Prusti's encoding, excluding aspects that are not relevant to our extension.

Prusti encodes some Rust types directly into corresponding Viper types. For example, the Rust `bool` type is converted into Viper's `Bool` type; and Rust integer types are converted to `Int` in Viper[2]. Prusti can also translate deterministic, side-effect free expressions Rust expressions into corresponding Viper expressions. When encoding a `prusti_assume!()` or `prusti_assert!()` statement, Prusti converts the argument into a Viper expression e and emits a corresponding `assume` e or `assert` e statement.

Prusti's encoding of Rust function definitions and function calls is somewhat less direct. Fig. 5.4 shows Prusti's encoding of functions into Viper. When Prusti encodes a function definition into a Viper method, instead of encoding function pre- and postconditions as corresponding pre- and postconditions of the method, Prusti instead inhales the precondition at the beginning of the method body and exhales the postcondition at the end.

Although Viper includes a statement for method calls, Prusti does not use it in its translation of function calls. Instead, Prusti translates function calls by replacing the call with an exhale of the function's postcondition followed by an inhale of the precondition.

One benefit of this indirect translation is that it enables more flexibility in the translation of function pre- and postconditions, as opposed to using Viper's method call statements. In the following section, we show how we take advantage of this flexibility in our implementation of resources.

[2]Prusti also separately inserts bounds checks for values of machine integer types

5.3 Resource Encoding

An overview of our translation from our resource specifications into Viper is shown in Fig. 5.5. The (overloaded) syntax $[\![]\!]$ denotes the translation of Prusti declarations, Rust statements, and Rust types into Viper. We omit the translation of types, which is unchanged from the prior Prusti encoding [3]. Prusti's encoding of pre-existing features is unchanged, except for adding additional `label` statements (used for encoding our `holds()` expressions).

The syntax $[\![]\!]^o_c$ denotes the encoding of a Prusti expression in the context c, o. The context changes the way that `holds()` expressions are translated into Viper. The first element, c, represents the *current context*: either *no label* ε, or a *signed label* $-l$ or $+l$ (denoting whether we are currently removing or adding resources). The second element, o, represents the *old context* taking the form of either $old(l)$ or $cur(l)$. These contexts prescribe where (with respect to which existing label) generated `perm()` expressions should be evaluated and how they should be composed. For example, the context $+l', old(l)$, is used to encode `old(holds())` expressions that occur in the postcondition of a called function; and are computed by taking the difference of the `perm()` expressions taken before and after exhaling the function precondition, corresponding to labels l and l' respectively.

5.3.1 Resources and Resource Operations

As described previously, resource types (via their Resource constructors) are declared in Prusti as a Rust struct, annotated with the tag `#[resource_kind]`. We encode resource constructors as abstract predicates in Viper; the fields of the struct are encoded as the arguments to the abstract predicate. We chose this encoding because abstract predicates in Viper are used to model resources in Viper's resource model. Accordingly, we translate `resource()` expressions directly into `acc()` expressions in Viper. Our `produce!()` and `consume!()` statements are encoded as `inhale` and `exhale` statements in Viper; these provide our desired semantics directly.

5.3.2 `holds()` Expressions

Resource state expressions (Sec. 4.1.4) are encoded in Prusti using `holds()` expressions, which return a result of the Prusti type `PermAmount`, representing rational numbers. `PermAmount` values can also be constructed from Rust integer types such as `i32` and `u32`.

When they appear inside the body of a function (for example, as part of a `prusti_assert!()` statement), `holds()` expressions are translated into `perm()` expressions in Viper. However, the translation of `holds()` expressions is more complicated when they appear in the pre- or postcondition of a function. As described in Sec. 4.1.5, `holds()` expressions that appear in those contexts refer to the resources that are transferred via the pre- or postcondition respectively. Therefore, we cannot always encode `holds()` expressions as `perm()` expressions directly. In the remainder of this section, we describe the encoding of `holds()` expressions in pre- and postconditions for function

Declarations

$[\![\#[\texttt{resource\_kind}]\ \texttt{struct}\ R(T_1,\ldots,T_n);]\!] \rightsquigarrow \texttt{predicate}\ R(\texttt{arg1} : [\![T_1]\!], \ldots, \texttt{argn} : [\![T_n]\!])$

$$\left[\!\!\left[\begin{array}{l}\#[\texttt{requires}(f_{pre})]\\ \#[\texttt{ensures}(f_{post})]\\ \texttt{fn}\ f(x_1 : T_1,\ldots,x_n : T_n)\ \{s_1;\ldots;s_m\}\end{array}\right]\!\!\right] \rightsquigarrow \begin{array}{l}\texttt{method}\ f(x_1 : [\![T_1]\!], \ldots, x_n : [\![T_n]\!])\{\\ \quad \texttt{inhale}\ [\![f_{pre}]\!]_{\varepsilon}^{cur(l_\varepsilon)};\ \texttt{label pre};\\ \quad [\![s_1]\!];\ldots;[\![s_m]\!]\\ \quad \texttt{label post};\ \ \texttt{exhale}\ [\![f_{post}]\!]_{\texttt{-post}}^{cur(\texttt{pre})}\\ \}\end{array}$$

Statements

$[\![\texttt{produce!}(t);]\!] \rightsquigarrow \texttt{inhale}\ [\![t]\!]_{\varepsilon}^{cur(\texttt{pre})} \qquad [\![\texttt{consume!}(t);]\!] \rightsquigarrow \texttt{exhale}\ [\![t]\!]_{\varepsilon}^{cur(\texttt{pre})}$

$$[\![f(t_1,\ldots,t_n);]\!] \rightsquigarrow \begin{array}{l}\texttt{label}\ l_{pre};\ \texttt{exhale}\ [\![f_{pre}[x_1 := t_1,\ldots,x_n := t_n]]\!]_{-l_{pre}}^{cur(l_\varepsilon)}\\ \texttt{label}\ l_{post};\ \texttt{inhale}\ [\![f_{post}[x_1 := t_1,\ldots,x_n := t_n]]\!]_{+l_{post}}^{cur(l_{pre})}\end{array}$$

Expressions

$[\![\texttt{old}(t)]\!]_{c}^{cur(l)} \rightsquigarrow [\![t]\!]_{c}^{old(l)}$

$[\![R(t_1,\ldots,t_n)]\!]_{c}^{o} \rightsquigarrow R([\![t_1]\!]_{c}^{o},\ldots,[\![t_n]\!]_{c}^{o}) \qquad [\![\texttt{resource}(r,t)]\!]_{c}^{o} \rightsquigarrow \texttt{acc}([\![r]\!]_{c}^{o},[\![t]\!]_{c}^{o})$

$[\![\texttt{holds}(r)]\!]_{\varepsilon}^{cur(l)} \rightsquigarrow \texttt{perm}([\![r]\!]_{\varepsilon}^{cur(l)}) \qquad [\![\texttt{holds}(r)]\!]_{\varepsilon}^{old(l)} \rightsquigarrow \texttt{old[}l\texttt{](perm(}[\![r]\!]_{\varepsilon}^{old(l)}\texttt{))}$

$[\![\texttt{holds}(r)]\!]_{+l'}^{cur(l)} \rightsquigarrow \texttt{perm(}[\![r]\!]_{\varepsilon}^{cur(l)}\texttt{) - old[}l'\texttt{](perm(}[\![r]\!]_{\varepsilon}^{cur(l)}\texttt{))}$
$[\![\texttt{holds}(r)]\!]_{+l'}^{old(l)} \rightsquigarrow \texttt{old[}l\texttt{](perm(}[\![r]\!]_{\varepsilon}^{old(l)}\texttt{)) - old[}l'\texttt{](perm(}[\![r]\!]_{\varepsilon}^{old(l)}\texttt{))}$

$[\![\texttt{holds}(r)]\!]_{-l'}^{cur(l)} \rightsquigarrow \texttt{old[}l'\texttt{](perm(}[\![r]\!]_{\varepsilon}^{cur(l)}\texttt{)) - perm(}[\![r]\!]_{\varepsilon}^{cur(l)}\texttt{)}$
$[\![\texttt{holds}(r)]\!]_{-l'}^{old(l)} \rightsquigarrow \texttt{old[}l\texttt{](perm(}[\![r]\!]_{\varepsilon}^{old(l)}\texttt{))}$

Figure 5.5: Encoding of Prusti resource constructs into Viper. Labels l_{pre} and l_{post} are assumed to be fresh. The label l_ε denotes an arbitrary fresh label that is never referenced, i.e., a placeholder label used where `old` expressions are not permitted.

definitions and function calls.

When Prusti encodes a Rust function definition into a Viper method, it encodes the precondition using an `inhale` statement, which is used as the first statement in the body of the method. `holds()` expressions in the precondition should refer to only the resources transferred into the function. In this case, this corresponds to the permissions to abstract predicates that are `inhaled` via the precondition, and therefore, `holds()` expressions are equivalent to analogous `perm()` expressions in this case.

Prusti encodes the postcondition of a function definition analogously as an `exhale` statement inserted at the end of the method body. To determine the difference in `perm()` expression prior to the beginning of the postcondition, a label l in inserted before the encoded postcondition. To represent the permissions transferred by the postcondition, `holds()` expressions are computed as

```
#[resource_kind]
struct R();

#[requires(resource(R(), x)
       && (holds(R()) == x))]
#[ensures(resource(R(), x+1)
      && (holds(R()) == x+1))]
fn inc_res(x: u32) {
  produce!(resource(R(), x));
}

#[requires(true)]
#[ensures(true)]
fn client() {
  inc_res(0);
}
```

```
predicate R()

method inc_res(x: Int) {
  inhale acc(R(), x) && perm(R()) == x
  inhale acc(R(), 1)
  label l0
  exhale acc(R(), x+1) &&
   (old[l0](perm(R()))-perm(R()) == x+1)
}

method client() {
  inhale true
  label l1
  exhale acc(R(), 0) &&
    (old[l1](perm(R()))-perm(R()) == 0)
  label l2
  inhale acc(R(), 0+1) &&
   (perm(R())-old[l2](perm(R())) == 0+1)
  label l3
  exhale true
}
```

Figure 5.6: Translation of a Prusti program with resource specifications into Viper. To simplify the presentation, casts between `PermAmount` and `u32` values are omitted in the Prusti code.

the difference of `perm()` expressions at l and the current point, a shown in Fig. 5.6.

Prusti encodes function calls by exhaling the precondition and subsequently inhaling the post-condition. To track the difference of permission between the calls, labels are inserted prior to the corresponding `inhale` and `exhale` statements; and `holds()` expressions are translated as the difference between `perm()` expressions at various points; as can be seen in the encoding of the Rust function `client()` in Fig. 5.6.

5.4 Coupling Invariants and Reborrowing

We directly encode coupling invariants (declared with `#[invariant_twostate(`t`)]` annotations on Rust structs) as postconditions on the corresponding translated Viper methods. One restriction of our current methodology is that we do not support Rust functions that *reborrow* mutable references, returning to the caller a live mutable reference to e.g., the internals of the `Bank`. An example of such a function is shown in Fig. 5.7, where a mutable references to one of the balances is handed out to the caller. Technically, this requires that the *client code* would become responsible for maintaining the `Bank`'s two-state invariant. For untrusted client code (such as smart contracts running on a blockchain infrastructure), this should not be relied upon, and rejecting such functions (as we currently do) is the right approach. A similar issue and its solution has been discussed in the context

```
struct Bank { balances: HashMap<AcctId, u32> }
impl Bank {
  fn get_balance_ref(&mut self, acct_id: AcctId) -> &mut u32 {
    self.balances.get_mut(acct_id).unwrap()
  }
}
```

Figure 5.7: A function that performs a reborrow. The variable `self` is inaccessible at the end of the function call and the returned reference can modify its internal state; the Bank's two-state invariant cannot yet be re-established.

of *single-state* invariants for Prusti [3]. However, for *trusted* client code (e.g., other verified layers of the same software stack), we believe an adaptation of this idea to support reborrowing with our two-state invariants, as future work (it remains to consider exactly which pairs of states we would use to enforce our two-state invariants in potential extension). This feature has not been needed in practice when applying our methodology to examples (likely because this additional reliance on client code for correctness is often not desirable for such programs).

With this encoding in place (and implemented), we can verify Rust code specified in our methodology directly and automatically, as we evaluate in the following sections.

Chapter 6

Case Study

In this case study, we consider the core of a Rust implementation [15] of a cryptocurrency token transfer application running on the Interblockchain Communication Protocol (IBC) [11]. The token transfer application enables tokens to be sent from one blockchain to another, without the need for a trusted intermediary; a bug in the application could inadvertently cause tokens to be destroyed or duplicated.

In Sec. 6.1 we present an overview of the token transfer application. In Sec. 6.2 we describe the properties that are verified. We present our specification with resources in Sec. 6.3.

6.1 The Fungible Token Transfer Application

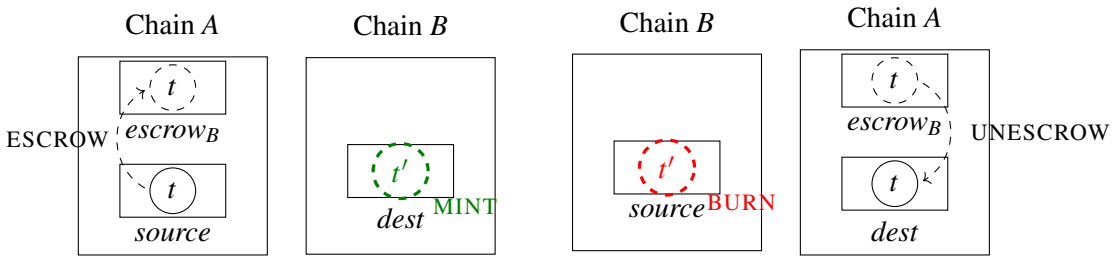
IBC enables communication between applications running on different blockchains. One important use case for the protocol is to enable tokens from one blockchain to be transferred to another. IBC defines an interface for performing such transfers in the Fungible Token Transfer specification [12].

We begin this section with a high-level overview of the protocol, and describe the relevant interfaces and data structures of the Rust implementation in Sec. 6.1.2.

6.1.1 Overview

The token transfer specification allows tokens to be sent bidirectionally between two heterogeneous blockchains, i.e., blockchains with potentially different implementations and consensus mechanisms. The specification requires that each chain track token ownership for accounts on its own chain, but not for accounts on other chains. The application effectively has access to the ledger on each chain; allowing it to mint and burn tokens for arbitrary accounts on that chain. Therefore, the application "transfers" a token between chains by making ledger updates on both chains.

A naïve token transfer implementation could operate by destroying the tokens on one chain and creating them on another. However, the naïve approach has two issues. First, a chain may not have the capacity to mint some kinds of tokens, i.e., tokens with fixed supply. Second, this approach



(a) Operations performed when sending a native token on A to another chain B. The token is escrowed on A and a voucher is minted on B.

(b) Operations performed when sending a voucher token back to its originating chain. The voucher is burned on B and the native token is unescrowed on A.

Figure 6.1: Operations of the Token Transfer Application.

would allow an exploit allowing unrestricted minting of a token on one chain to allow obtaining an arbitrary amount of that token on any connected chain. In particular, if a chain A had a vulnerability allowing arbitrary minting, an attacker could mint on A a token that exists on chain B, and then send it to chain B.

Instead, the fungible token transfer application performs a token transfer from chain A to chain B by first sending the token to a special *escrow account* on chain A, and then minting a *voucher token* on chain B (shown in Fig. 6.1a). The denomination of the voucher token is created by prefixing the denomination of A's token with the identifier of the channel connecting the two chains.

Intuitively, the voucher token on B corresponds to the escrowed token on A. In particular, when the application transfers the voucher token back to A, it is burned (destroyed) on chain B, and the tokens in the escrow account in chain A are sent to the recipient (as shown in Fig. 6.1b). Effectively, voucher tokens are equivalent in value to the native token they represent, as the owner of a voucher token has the ability to redeem it for the native token on the originating chain. The advantage of this design is that it does not require the token transfer application to mint or burn *native* (i.e., non-voucher) tokens. Because this design ensures that native tokens are never minted, exploits on one chain cannot use the protocol to inflate the supply of native tokens on other chains.

When a voucher token is transferred to a chain other than its originating chain, it is treated the same as any other token. This enables tokens to be transferred transitively across multiple chains. For example, suppose A, B and C are blockchains, and B can transfer tokens with both chains A and C. Then, a token on A can be sent to C by first making a transfer from A to B, and sending the voucher minted on B to C. The token on C is a voucher for the token on B, which itself is a voucher for the token on A.

6.1.2 IBC Data Structures and Interfaces of the Rust Implementation

Fig. 6.2 presents the structures to represent tokens. The struct `PrefixedCoin` consists of a token denomination and an integer amount; this type is used in the bank interface to specify which token

```
struct PrefixedDenom {trace_path: TracePath, base_denom: BaseDenom}
struct PrefixedCoin {denom: PrefixedDenom, amount: u32}
```

Figure 6.2: Data Types used in the Token Transfer Application

```
impl BankKeeper {
  fn send_tokens(&mut self, from: AccountId, to: AccountId, coin: PrefixedCoin);
  fn burn_tokens(&mut self, from: AccountId, coin: PrefixedCoin);
  fn mint_tokens(&mut self, to: AccountId, coin: PrefixedCoin);
}
```

Figure 6.3: The `BankKeeper` interface

should be minted, burned, or transferred. The struct `PrefixedDenom` refers to a token denomination: `base_denom` is the name of the token on the originating chain, and `trace_path` denotes the sequence of token transfers necessary to exchange a token of this denomination with the one on its original chain.

To verify the properties specified in Sec. 6.2, we are primarily concerned with how the token transfer application modifies ledger balances; specifically, by minting, burning, or transferring tokens. This functionality is implemented by the `BankKeeper` interface of Fig. 6.3.

`AccountId` is a Rust struct that refers to an account, which can be an account on the local chain, or an account on a remote chain. The `BankKeeper` interface only handles accounts on the local chain; but the token transfer application considers both types.

6.2 Properties of the Fungible Token Transfer Application

For our case study, we focused on verifying two key properties of the token transfer application:

- **Two-Way Peg:** Round-trip transfers of tokens should maintain the state of all account balances.
- **Preservation of Supply:** For a given native token, the sum of the native token and all derived voucher tokens in non-escrow accounts should remain constant.

These properties are based on descriptions from the IBC specification [12] (the two-way peg property is phrased slightly differently in the original specification). These properties only hold under the assumption that neither chain experiences consensus failures. For example, if chain A sent a token to chain B, a subsequent consensus failure on chain A could change the history on A such that the transfer action was not recorded. In such a scenario, the token would then be be usable on both chains, thereby inflating the overall supply.

```
#[resource_kind]
struct Money(BankID, AccountId, PrefixedDenom);
```

Figure 6.4: The Prusti encoding of a resource representing an amount of a token (which may be a voucher for a token originating on another chain) for a particular account in a particular bank.

```
#[resource_kind]
struct UnescrowedCoins(BankID, BaseDenom);
```

Figure 6.5: The Prusti encoding of a resource representing an amount of unescrowed token with a particular base denomination, held by a particular bank.

6.3 Encoding the Specification with Resources

We begin the presentation of our specification by describing the representation of resources. First, we note that the two properties above refer to different aspects of tokens: the two-way peg property is concerned with the tokens in each account, while the preservation of supply is concerned with the sum of unescrowed tokens across all accounts. The second property does not distinguish between native and voucher tokens, however this distinction is relevant for the first property. Therefore, we write our specifications in terms of two different resource kinds; each kind corresponds to a different view on the tokens.

Because tokens are stored on a single chain, we index both resource kinds by a value of type `BankId`, which identifies the chain that the token resides on. This indexing will be necessary to establish coupling invariants (described later in Sec. 6.3.2): including the `BankId` as part of the resource kind ensures that we can specify how operations on tokens are reflected as changes in the state of the corresponding chain.

To ensure the two-way peg property, we consider two tokens as having distinct resource types if they differ in denomination or if they have different owners. The encoding of this resource in Prusti is shown in Fig. 6.4. With this resource definition, we can ensure the two-way peg property by showing that round-trip transfers do not change the amount of tokens of this type in the resource state.

The second property requires showing that the total amount of all unescrowed tokens of a particular base denomination remain unchanged. For this property, we use a different resource kind, such that tokens having the same type when they are in the same bank and have the same base denomination. This encoding is shown in Fig. 6.5.

6.3.1 Annotating `BankKeeper` with Resource Operations

In our specification, we use a Rust macro to express an operation on tokens as resource operations involving both `UnescrowedCoins` and `Money`. The resulting macro is presented in Fig. 6.6.

```
macro_rules! transfer_money {
  ($bank_id:expr, $to:expr, $coin:expr) => {
    resource(Money($bank_id, $to, $coin.denom), $coin.amount) &&
    if !is_escrow_account($to) {
      resource(UnescrowedCoins($bank_id, $coin.denom.base_denom), $coin.amount)
    } else { true }
  }
}
```

Figure 6.6: A macro that specifies what Prusti resources are changed (i.e., either created or destroyed) in response to token operations. An operation on tokens will always change the `Money` resource, and also changes the `UnescrowedCoins` resource if the target account is not an escrow account.

```
impl BankKeeper {
  #[requires(from != to)]
  #[requires(transfer_money!(self.id(), from, coin))]
  #[ensures(transfer_money!(self.id(), to, coin))]
  fn send_tokens(&mut self, from: AccountId, to: AccountId, coin: PrefixedCoin);

  #[requires(transfer_money!(self.id(), from, coin))]
  fn burn_tokens(&mut self, from: AccountId, coin: PrefixedCoin);

  #[ensures(transfer_money!(self.id(), to, coin))]
  fn mint_tokens(&mut self, to: AccountId, coin: PrefixedCoin);
}
```

Figure 6.7: The `BankKeeper` annotated with the appropriate resource operations.

We can then use the `transfer_money()!` macro to annotate `BankKeeper`'s associated functions with appropriate resource operations, as presented in Fig. 6.7.

6.3.2 Establishing Coupling Invariants for the `BankKeeper` implementation

Annotating `BankKeeper` with resource operations is sufficient to enable specification of the token transfer application in terms of resource operations. However, this alone does not enable us to express any properties about the program state. Ultimately, we wish to prove the properties not just in terms of the way we chose to model resources, but also as properties of the program itself.

To prove such properties in terms of program state, we establish two-state invariants on the `BankKeeper` function, which describe how changes to `Money` and `UnescrowedCoins` in the resource state correspond to changes in functions `balance` and `unescrowed_coin_balance` respectively. These invariants are shown in Fig. 6.8.

```
#[invariant_twostate(forall(|acct_id: AccountId, denom: PrefixedDenom|
  holds(Money(self.id(), acct_id, denom)) -
  old(holds(Money(self.id(), acct_id, denom))) ==
  PermAmount::from(self.balance(acct_id, denom)) -
      PermAmount::from(old(self.balance(acct_id, denom)))))]

#[invariant_twostate(forall(|coin: BaseDenom|
  holds(UnescrowedCoins(self.id(), coin)) -
  old(holds(UnescrowedCoins(self.id(), coin))) ==
  PermAmount::from(self.unescrowed_coin_balance(coin)) -
      PermAmount::from(old(self.unescrowed_coin_balance(coin)))))]
```

Figure 6.8: Invariants connecting the resources `Money` and `UnescrowedCoins` to the methods `balance()` and `unescrowed_coin_balance()` respectively.

6.3.3 Verifying the Application Logic

We now focus our attention to the logic of the token-transfer application itself. The application performs a token transfer from a chain A to chain B in two steps. The first step is performed by calling the function `send_fungible_tokens()` (Fig. 6.9) on chain A, which disposes of the tokens on A by either escrowing or burning them. In the second step, it effectively causes the function `on_recv_packet()` to be called on chain B, which produces the tokens by either minting or unescrowing tokens[1].

Fig. 6.9 is a simplified version of the function `send_fungible_tokens()` that performs the first step of a cross-chain token transfer. The arguments `port` and `channel` identify the chain that tokens will be transferred to. The condition on line 8 checks if the token being sent is a voucher for a token on that chain. If so, then the voucher is burned on this chain (and the next step will unlock the corresponding token on the other chain). Otherwise, the token is moved to an escrow account on this chain.

Fig. 6.10 presents the specification for this function. The precondition reflects the fact that calling `send_fungible_tokens()` will transfer the token out of the sender's account. The postcondition indicates that when the token did not originate on the opposite chain (i.e., the chain identified by `port` and `channel`), it will be transferred to the escrow account.

We now consider the behaviour of the receiving end of the chain. Fig. 6.11 shows the code. The conditional `packet.is_source()` is true when the received token is a voucher for the token originating on this chain. If so, `on_recv_packet()` transfers the token from the escrow account to the receiver. Otherwise, the token originates on the other chain, and a voucher token is minted into the receiver's account. The corresponding specifications are presented in Fig. 6.12.

[1]Technically, `on_recv_packet()` is a callback that is triggered when a relayer sends a message to chain B, however for verification we consider this as a regular function call.

```
fn send_fungible_tokens(
  bank: &mut Bank,
  coin: &PrefixedCoin,
  sender: AccountId,
  port: Port,
  channel: ChannelEnd,
) {
  if coin.denom.trace_path.starts_with(port, channel) {
    bank.burn(sender, coin);
  } else {
    bank.send(sender, escrow_address(channel), coin);
  };
}
```

Figure 6.9: The first step of the fungible token transfer.

```
#[requires(transfer_money!(bank.id(), sender, coin))]
#[ensures(!coin.denom.trace_path.starts_with(port, channel) ==>
    transfer_money!(bank.id(), escrow_address(channel), coin))]
```

Figure 6.10: The specification for the function `send_fungible_tokens()`.

6.3.4 Verifying the Desired Properties

With the above specifications, we can then prove that the token transfer application satisfies the properties described in Sec. 6.2.

To prove the two-way peg property, we construct a method that performs a round trip token transfer, and show that the balances of all accounts are unchanged after the transfer. The relevant specifications are presented in Fig. 6.13 (the body of the method is omitted for brevity). We verify the preservation of supply property by showing that the supply is preserved after an arbitrary transfer; the specifications are presented in Fig. 6.14. The specifications of both properties are expressed in terms of the functions `balance()` and `unescrowed_coin_balance()` respectively.

In conclusion, we've shown that our methodology can be applied to a real-world resource-manipulating program. Using our methodology, we were able to prove two important properties about the token-transfer application: that it maintains a two-way peg and preserves total token supply. Our technique allow us to verify the desired properties in a straightforward manner by describing resource operations directly, without having to write frame conditions.

```
fn on_recv_packet(bank: &mut Bank, packet: &Packet) {
    let coin = packet.get_recv_coin();
    if packet.is_source() {
        bank.transfer_tokens(
            escrow_address(packet.dest_channel),
            packet.data.receiver,
            &coin
        );
    } else {
        bank.mint_tokens(packet.data.receiver, &coin);
    };
}
```

Figure 6.11: The second step of the fungible token transfer, executed on the receiving chain.

```
#[requires(packet.is_source() ==> transfer_money!(
    bank.id(),
    escrow_address(packet.dest_channel),
    packet.get_recv_coin()
))]
#[ensures(
  transfer_money!(bank.id(), packet.data.receiver, packet.get_recv_coin())
)]
```

Figure 6.12: The specification for the function `on_recv_packet()` from Fig. 6.11.

```
#[ensures(forall(|acct_id2: AccountId, denom: PrefixedDenom|
    bank1.balance(acct_id2, denom) == old(bank1).balance(acct_id2, denom)))]
#[ensures(forall(|acct_id2: AccountId, denom: PrefixedDenom|
    bank2.balance(acct_id2, denom) == old(bank2).balance(acct_id2, denom)))]
fn round_trip(bank1: &mut Bank, bank2: &mut Bank, coin: &PrefixedCoin,
    sender: AccountId, receiver: AccountId, ...) { ... }
```

Figure 6.13: The relevant specifications for the two-way peg property. The function `round_trip()` performs a token transfer from account `sender` to `receiver`, and then performs the same transfer in the opposite direction.

```
#[ensures(forall(|c: BaseDenom|
      bank1.unescrowed_coin_balance(c) + bank2.unescrowed_coin_balance(c) ==
  old(bank1.unescrowed_coin_balance(c) + bank2.unescrowed_coin_balance(c))
))]
fn transfer(bank1: &mut Bank, bank2: &mut Bank, coin: &PrefixedCoin,
    sender: AccountId, receiver: AccountId, ...) { ... }
```

Figure 6.14: The relevant specifications for the supply preservation property. The function `transfer()` performs a token transfer from account `sender` to `receiver`.

Chapter 7

Evaluation

To evaluate our technique, we consider how specifications written using our methodology compare to alternative specifications written in terms of program states. We consider three research questions:

RQ1 (*Conciseness*): Are our specifications *smaller* than the alternative?

RQ2 (*Complexity*): Are our specifications *simpler* than the alternative?

RQ3 (*Verification Time*): Do our specifications *require more time to verify* than the alternative?

To answer these questions, we compare the specifications we developed in Sec. 6 to an alternative version we constructed, that verifies the same implementation using specifications written without using our methodology. In addition, we also apply our methodology to a simplified implementation of the IBC Non-Fungible Token (NFT) Transfer application [25], again comparing the results to an alternative version of the specification. The NFT transfer application has a similar architecture to the token transfer application. We wrote the implementation ourselves, based on pseudocode in the specification. We are not aware of a functioning Rust implementation of the protocol; ultimately we intend to develop our prototype into a verified reference implementation. A key difference in the application of our methodology between the two protocols, is that we use resources to model the *permission* to change ownership of a token, rather than modelling the token itself.

We now consider each research question in order, describing how we compared the specifications and presenting our results. We discuss the results in Sec. 7.4.

7.1 Conciseness

To measure conciseness, we compared the number of lines of code used to specify resource-related operations in both specifications. Lines that are unrelated to resource operations, which are identical in both versions, are not considered in the comparison. Table 7.1 provides a comparison between

Context	Usage	Without Resources	With Resources
TT-Resources	Resource Constructors	N/A	4
	Coupling Invariants	N/A	21
	`transfer_money!()` macro	N/A	10
TT-Bank	`burn()`	27	1
	`mint()`	26	1
	`send()`	41	2
TT-App	`send_fungible_tokens()`	13	3
	`on_recv_packet()`	19	6
TT-Properties	`send_preserves()`	16	26
	`round_trip()`	19	23
NFT-Resources	Resource Constructors	N/A	2
	Coupling Invariants	N/A	5
	`transfer_tokens!()` macro	N/A	5
NFT-Bank	`burn()`	8	2
	`mint()`	9	3
	`send()`	8	3
	`create_or_update_class()`	4	0
NFT-App	`send_nft()`	12	13
	`on_recv_packet()`	17	11
NFT-Properties	`round_trip()`	21	24
Total		240	165

Table 7.1: Comparison of the number of specification lines required for the specifications written with and without resources. Rows prefixed with `TT-` refer to the token transfer specifications, those prefixed with `NFT-` refer to the NFT transfer specifications.

the number of specification lines required for both specification versions. In total, the specifications concerned with resource properties (including invariants and helper macro definitions) consist of 165 lines of code. Encoding the equivalent specifications without resources required 240 lines.

Our methodology requires fewer lines of code to encode the specifications for the token transfer application's `BankKeeper` implementation, as well as the two main functions of the fungible token transfer application. Our specifications of the final properties require more lines of code, because they also indicate the resource operations of `send_preserves()` and `round_trip()`. However, the trade-off is acceptable, because these extra lines also make the specification more expressive compared to the alternative specification. Furthermore, a larger difference would be seen if we considered a program with more functions, because the invariants only need to be written once. The function `create_or_update_class()` in the NFT transfer application does modify token owner-

ship, and therefore does not require any specifications in our specification; in contrast, the alternative specification requires four lines of code to explicitly state this property.

7.2 Syntactic Complexity

More concise specifications are not necessarily simpler or more desirable. For example, a longer specification may be preferable to a shorter one if the latter involves complex nesting of conditionals, implications, and quantifiers. To evaluate syntactic complexity, we considered the AST nodes of `#[requires()]` and `#[ensures()]` clauses, as well as those of associated specification-related expressions [1]. We quantify the complexity of an AST using three metrics: the total number of nodes in the AST, the maximum depth of the AST, and the number of unique node types occurring within the AST. We classify the node types by differentiating arithmetic and comparison operators, and among Prusti constructs such as `forall` and `old`, while disregarding children in the classification and treating variables and constants uniformly. For example, the expression `a + (b - c)` has three different node types: identifiers (`a`, `b`, and `c`), addition, and subtraction. An overview of the different node types we consider is included in the appendix (Table A.1).

We present our results in Table 7.2. Our results demonstrate that the specifications written using our methodology are syntactically simpler in most cases. In particular, specifications related to `BankKeeper` compare favourably to the alternative w.r.t. all three metrics: specifications are less than $1/10^{\text{th}}$ the size, $1/3^{\text{rd}}$ of the depth, and contain less than $1/5^{\text{th}}$ as many unique nodes. The only area where our specifications are markedly more complex is with respect to the size of the specifications for `send_preserves()` (79 vs 47) and `round_trip()` (72 vs 53). As mentioned in the prior section, this is due to the requirement to indicate the resource operations of the functions in additional to the program-state properties. We note that although the size of the coupling invariants is relatively large (comparable with the size of `mint()` and `burn()`), the invariant only needs to be written once, regardless of the number of functions in the program. Therefore, we expect that we our methodology would compare even more favourably if applied to larger programs.

7.3 Verification Time

We compared the runtime of the specification written using our methodology, and the alternate version. For each version, we performed five runs, all runs were performed a 10-core Apple M1 Max. Our results are presented in Table 7.3. The verification time for the NFT transfer application is similar for both versions (59.88s vs 57.04s). There is a larger difference w.r.t. token transfer application: the specification using resources is 27% slower (83s vs 65.12s). This is most likely due to the overhead of casting between the types used to represent resource amounts and the types represent-

[1] For example, the function `burn_tokens_post()`, which expresses the postcondition of `burn()`, is also used in the specification of `send_fungible_tokens()`. We associate the body of `burn_tokens_post()` towards the count of `burn()`, rather than `send_fungible_tokens()`.

Context	Usage	Without Resources			With Resources		
		Size	Depth	Uniq.	Size	Depth	Uniq.
TT-Resources	Resource Constructors	N/A	N/A	N/A	7	2	2
	Coupling Invariants	N/A	N/A	N/A	90	9	9
	`transfer_money!()` macro	N/A	N/A	N/A	26	7	8
TT-Bank	`burn()`	92	10	12	5	3	2
	`mint()`	84	10	11	5	3	2
	`send()`	136	11	14	10	3	2
TT-App	`send_fungible_tokens()`	34	6	7	20	6	5
	`on_recv_packet()`	45	6	7	20	5	4
TT-Properties	`send_preserves()`	47	7	9	79	7	10
	`round_trip()`	53	6	8	72	6	7
NFT-Resources	Resource Constructors	N/A	N/A	N/A	4	2	2
	Coupling Invariants	N/A	N/A	N/A	42	10	10
	`transfer_tokens!()` macro	N/A	N/A	N/A	9	4	5
NFT-Bank	`burn()`	29	8	7	10	3	3
	`mint()`	37	8	7	18	3	3
	`send()`	31	8	7	16	3	3
	`create_or_update_class()`	14	6	5	0	0	0
NFT-App	`send_nft()`	56	9	8	50	6	8
	`on_recv_packet()`	67	9	8	55	7	6
NFT-Properties	`round_trip()`	65	6	7	81	6	8

Table 7.2: Comparison between the syntactic complexity of specifications written with and without resources. Each column considers the ASTs of the specification expressions for the function in that row. **Size** refers to the total number of nodes in the ASTs. **Depth** refers to the height of the tallest AST. **Uniq.** refers to the number of distinct node types occurring in the ASTs.

ing bank balances in the underlying Viper encoding: changing the specification to instead express balance using Viper's permission amount types (i.e., rational numbers as opposed to integers) eliminates the difference (resulting in timings of 88.74s vs 90.38s respectively). As future work, we believe it could be possible to reduce the performance overhead associated with such casts.

7.4 Analysis and Discussion

Our evaluation shows that specifications written in our methodology compare favourably to specifications written in terms of program states. Our specifications require fewer lines of code and syntactically simpler. Although our specifications sometimes require more time to verify, the increase in time is a reasonable trade-off to make for the simpler specifications. Furthermore, our

Application	Without Resources (mean / sd)	With Resources (mean / sd)
Token Transfer	65.12s / 0.15s	83.00s / 0.78s
NFT Transfer	57.04s / 0.23s	59.88s / 0.36s

Table 7.3: Comparison of verification time for the specifications written in the different styles. Results are presented for five runs of the verifier.

evaluation indicates that the increase is due to the behaviour of the underlying Viper verifier rather than a fundamental consequence of using our methodology.

In addition, we conjecture that our methodology enables users to write specifications that are easier to interpret than specifications written in terms of program states. Our conjecture is based on the comparison of specifications written for the same functions considered in the evaluation, for example as in Fig. 7.1. However, as interpretability is subjective, formally evaluating this aspect would require a user study, which he have not yet performed. Performing such a study could be a potential avenue of future work.

```
pub fn transfer_tokens_post(&self, old_bank: &Self, from: AccountId, to: AccountId,
    coin: &PrefixedCoin
) -> bool {
    self.unescrowed_coin_balance(coin.denom.base_denom) ==
        if (is_escrow_account(to) && !is_escrow_account(from)) {
            old_bank.unescrowed_coin_balance(coin.denom.base_denom) - coin.amount
        } else if (!is_escrow_account(to) && is_escrow_account(from)) {
            old_bank.unescrowed_coin_balance(coin.denom.base_denom) + coin.amount
        } else {
            old_bank.unescrowed_coin_balance(coin.denom.base_denom)
        } &&
    forall(|acct_id2: AccountId, denom2: PrefixedDenom|
        self.balance_of(acct_id2, denom2) ==
            if(acct_id2 == from && coin.denom == denom2) {
                old_bank.balance_of(from, coin.denom) - coin.amount
            } else if (acct_id2 == to && coin.denom == denom2){
                old_bank.balance_of(to, coin.denom) + coin.amount
            } else {
                old_bank.balance_of(acct_id2, denom2)
            }
    ) && forall(|c: BaseDenom| implies!(c != coin.denom.base_denom,
      self.unescrowed_coin_balance(c) == old_bank.unescrowed_coin_balance(c)
    ))
}
```

(a) The postcondition for `BankKeeper`'s `send_tokens()` function, specified in terms of program states.

```
#[requires(transfer_money!(self.id(), from, coin))]
#[ensures(transfer_money!(self.id(), to, coin))]
```

(b) The specification of `send_tokens()` written using resource reasoning constructs. The macro `transfer_money!()` is defined in Fig. 6.6.

Figure 7.1: Specifications written with and without resources for `send_tokens()`.

Chapter 8

Related Work

8.1 Type and Effect Systems

Effect systems [17] extend a type system to include the side-effects via effect types, which describe the side-effects an expression is allowed to perform. The resource operations in our methodology are similar in the sense that they define imperative operations: functions with a `resource()` annotation in their precondition have the effect of consuming a resource; when the same annotation appears in the postcondition, the function has the effect of producing the resource.

There are, however, two key differences between our methodology and the effect typing methodology. First, effect types typically describe an over-approximation of the runtime side-effects that an expression can perform. In contrast, the specifications in our methodology precisely describe the transfer of resources. Second, the methodologies differ with respect to what they are meant to track. The effect type of a function is intended to capture all side-effects that could occur during its execution. Using effect types to track the resource operations performed by a function would involve considering every operation that occurs within the function. In contrast, the resource operations in our function specifications only describe the resources that must be transferred into the function (via the precondition), and those that will be returned to the caller (via the postcondition). The specifications summarise the effect of all resource operations that occur within the body, effectively hiding the implementation details.

8.2 Separation Logic

Separation logic [22] enables verification of heap-manipulating programs using local reasoning: assertions describe only the relevant part of the heap, rather than the heap as a whole. Extensions of separation logic facilitate abstraction with user-defined predicates [21]. Variants of separation logic also support reasoning about varying permissions to heap locations, via fractional permissions and counting permissions [6].

Our methodology facilitates separation-logic style reasoning about resources: function specifications refer to a local resource state. However, our methodology does not use resources to model the heap itself. Additionally, our `holds()` construct, which allows introspection on the local resource state, does not have an analogue in separation logic.

The Viper intermediate language [20] supports separation-logic style reasoning, which it uses to model the heap, as well as abstract predicates. Viper also supports introspection via its `perm()` expressions. However, in contrast to our `holds()` construct, Viper's `perm()` expressions cannot be used directly in method pre- and postconditions. Viper requires that `perm()` expressions used in these contexts be nested within *inhale-exhale* assertions. Viper's inhale-exhale assertions consist of two expressions, indicating how the assertion should behave when inhaled and exhaled respectively. These expressions can introduce unsoundness (i.e., allow the user to assume `false`); Viper places the burden on the user to construct these assertions appropriately for their use case. In contrast, the `holds()` expressions in our methodology can be used directly in pre- and postconditions, and cannot directly introduce unsoundness.

8.3 Verification of Smart Contracts

There is substantial prior work focused on verification of smart contracts themselves [1][7][19]: while not the specific focus of our work, these are clearly resource-manipulating programs (used to handle cryptocurrency assets). The verification tool 2Vyper [7] is closest to our work: it provides its own resource reasoning via effect clauses for smart contracts in the Vyper language: these can specify possible resource transfers, and users can also define invariants that connect the resource state to the runtime state. However, there are several technical differences with our work: unlike our system, 2Vyper's resource specifications describe approximations (in the style of effect systems) of a function's behaviour, and over representations of the entire resource state; there is no way to partition the resource state into a only a local part that a function is concerned with, which is what eliminates heavyweight frame conditions in our work.

2Vyper's effect-clauses consist of a multiset of the operations that will occur in their execution, but these only refer to the resource operations performed directly by the function itself: because 2Vyper considers interactions with unverified external code, it is not possible to reason in general about external resource operations. In contrast, because we do not allow untrusted external calls, the specifications in our methodology effectively summarise the resource operations that occur within a method call. We have not yet applied our technique to verify interactions with untrusted code; doing so could be interesting future work.

Ahrendt and Bubel verify Solidity contracts with a proof technique centred around two-state invariants [1]. They show that *differences* between *wei* amounts (the built-in currency) are powerful for expressing such invariants, similar to the encoding of our `holds` feature into Viper. They address untrusted code and security properties while we do not; on the other hand, their work does

not support custom notions of resource, build in resource-like properties, or address data structure framing; their technique is concerned specifically with currency in Solidity.

Other prior work has focused on verification of smart contracts by modelling them as extended finite state machines [19]. Our approach is more general, as it is not limited to the domain of smart contracts and does not assume any particular program architecture.

The specification language Chainmail [10] enables user-defined invariants using holistic specifications, which can be used to enforce a wide range of security-related properties, including some related to resources in the program. For example, it is possible to define an invariant that ensures any change in the balance of a particular account is associated with a call to `deposit()` referencing that account. Specifications in Chainmail must still ultimately be phrased in terms of program states; leading to less direct and more-complex specifications. In contrast, our methodology, which treats resources as first class, allows specifications concerning resource operations directly.

8.4 Resource Support in Smart Contract Languages

Various smart contract languages provide first-class support for resources. The Move language [4], originally developed by Facebook for the Diem blockchain, supports first-class resources that are implemented with linear types. Obsidian [9] uses both linear types and typestate to prevent bugs related to improper handling of resources. Flint [23] supports asset types that encapsulate unsafe operations and provide a safe interface.

Neither Move, Obsidian, nor Flint support static reasoning about the resource quantity. In contrast to our approach, they cannot ensure that resource quantities are preserved, or any other quantitative properties about resources.

Chapter 9

Conclusion

In this thesis, we examined the challenges encountered when writing specifications of resource manipulating programs in terms of program state. When using a modular verifier, this approach requires users to explicitly write frame conditions in specifications, and these frame conditions make specifications lengthier and harder to interpret. Furthermore, such specifications do not easily compose, and are not expressive enough to rule out certain kinds of resource-related bugs.

The root cause of these issues is the semantic gap between our high-level, intuitive expectations of how resources behave, and the language used to write specifications in the code. Therefore, we developed a new methodology to support reasoning about resources directly inside specifications, thereby narrowing the semantic gap.

The methodology we developed extends a program verifier with a first-class notion of resources, without requiring support for resources in the source language. Instead, we allow users to define coupling invariants to connect resource operations to program state. These invariants are checked by the verifier, allowing users to describe the behaviour of their program in terms of resource operations rather than as relations on program states.

We implemented our methodology as an extension to the program verifier Prusti, and evaluated our design by using our extended version of Prusti to verify a real-world resource-manipulating program. Our evaluation shows that, compared to a standard Prusti specification written in terms of program states, specifications written using our methodology are more concise and syntactically simpler.

For future work, we would like to extend our methodology to facilitate interactions with untrusted or external code, as such interactions are typical in dealing with smart contracts. In particular, enforcing coupling invariants for functions that reborrow could increase the applicability of our technique. Finally, we could consider applying our methodology to reason about resources within a program, such as locks, database connections, or file handles.

Bibliography

[1] W. Ahrendt and R. Bubel. Functional verification of smart contracts via strong data integrity. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, volume 12478 of *Lecture Notes in Computer Science*, pages 9–24. Springer, 2020. doi:10.1007/978-3-030-61467-6_2. URL https://doi.org/10.1007/978-3-030-61467-6_2. → page 49

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust Types for Modular Specification and Verification. *Proceedings of the ACM on Programming Languages*, 3 (OOPSLA):1–30, 2019. → pages 3, 5, 6

[3] V. Astrauskas, A. Bílỳ, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*, pages 88–108. Springer, 2022. → pages 2, 30, 33

[4] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. R. Rain, S. Sezer, et al. Move: A Language with Programmable Resources. *Libra Assoc*, page 1, 2019. → page 50

[5] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011. → page 25

[6] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission Accounting in Separation Logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, 2005. → page 48

[7] C. Bräm, M. Eilers, P. Müller, R. Sierra, and A. J. Summers. Rich Specifications for Ethereum Smart Contract Verification. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021. → page 49

[8] C. Calcagno, P. W. O'Hearn, and H. Yang. Local Action and Abstract Separation Logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378. IEEE, 2007. → page 21

[9] M. Coblenz, R. Oei, T. Etzel, P. Koronkevich, M. Baker, Y. Bloem, B. A. Myers, J. Sunshine, and J. Aldrich. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(3):1–82, 2020. → page 50

[10] S. Drossopoulou, J. Noble, J. Mackay, and S. Eisenbach. Holistic Specifications for Robust Programs. In *FASE*, pages 420–440, 2020. → page 50

[11] C. Goes. The Interblockchain Communication Protocol: An Overview. *arXiv preprint arXiv:2006.15918*, 2020. → pages 8, 34

[12] C. Goes. Ics-20 Fungible Token Transfer Specification, 2022. URL https://github.com/cosmos/ibc/tree/main/spec/app/ics-020-fungible-token-transfer. → pages 34, 36

[13] Z. Grannan and A. J. Summers. Resource Specifications for Resource-Manipulating Programs, 2023. URL https://s3.us-west-1.wasabisys.com/zg-public/oopsla23.pdf. → page v

[14] A. M. Hein. Identification and Bridging of Semantic Gaps in the Context of Multi-domain Engineering. In *Forum on Philosophy, Engineering & Technology*, pages 58–57, 2010. → page 1

[15] Informal Systems Inc. and ibc-rs authors. ibc-rs, 2022. URL https://github.com/cosmos/ibc-rs. → page 34

[16] K. R. M. Leino. This is Boogie 2. *manuscript KRML*, 178(131):9, 2008. → page 25

[17] J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 47–57, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912527. doi:10.1145/73560.73564. URL https://doi.org/10.1145/73560.73564. → page 48

[18] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Readings in artificial intelligence*, pages 431–450. Elsevier, 1981. → pages 2, 4

[19] S. Mohajerani, W. Ahrendt, and M. Fabian. Modeling and Security Verification of State-Based Smart Contracts. *IFAC-PapersOnLine*, 55(28):356–362, 2022. → pages 49, 50

[20] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A Verification Infrastructure for Permission-based Reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, pages 41–62. Springer, 2016. → pages 5, 25, 49

[21] M. Parkinson and G. Bierman. Separation Logic and Abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 247–258, New York, NY, USA, 2005. Association for Computing Machinery. ISBN

158113830X. doi:10.1145/1040305.1040326. URL https://doi.org/10.1145/1040305.1040326. → page 48

[22] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002. → page 48

[23] F. Schrans, S. Eisenbach, and S. Drossopoulou. Writing Safe Smart Contracts in flint. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, pages 218–219, 2018. → page 50

[24] J. Smans, B. Jacobs, and F. Piessens. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*, pages 148–172. Springer, 2009. → page 25

[25] H. Xi. Ics-721 Non-Fungible Token Transfer Specification, 2022. URL https://github.com/cosmos/ibc/tree/main/spec/app/ics-721-nft-transfer. → page 42

Appendix A

Supporting Materials

Node Type	Meaning
Add	Addition expression
And	Logical AND expression
BoolLit	Boolean literal
Call	Function call
Cast	Type cast
Eq	Equality comparison
FieldAccess	Struct field access
Forall	Universal quantification
Ge	Greater-than or equal to comparison
Holds	`holds()` clause
Ident	Identifier
If	Conditional expression
Implies	Logical implication (=>)
IntLit	Integer literal
Ne	Not equal comparison
Not	Logical negation
Old	`old()` expression
Reference	Reference (i.e `&x`)
Resource	`resource()` expression
ResourceConstructor	Resource constructor application
Sub	Subtraction expression

Table A.1: Classification of different node types for determining the number of unique nodes in specifications.