

Beyond Design by Contract: Formal Specification of Composed Behaviour

Nele Smeets
Eric Steegmans

Report CW 398, January 2005



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Beyond Design by Contract: Formal Specification of Composed Behaviour

Nele Smeets
Eric Steegmans

Report CW 398, January 2005

Department of Computer Science, K.U.Leuven

Abstract

The basic building blocks of Design by Contract are Boolean assertions, which are used to express preconditions, postconditions and invariants. For composed commands, i.e. for commands whose effect is realized using other commands, the expressiveness of such contracts appears to be insufficient. In this paper, we propose an extension of the Design by Contract methodology that enables us to express the specification of a composed method in terms of the specification of the composing methods. Among the main advantages of this approach, we distinguish increased expressiveness of contracts, better consistency and trivial code generation.

Beyond Design by Contract: Formal Specification of Composed Behaviour

Nele Smeets*, Eric Steegmans

Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan
200A, 3001 Leuven, Belgium

{Nele.Smeets, Eric.Steegmans}@cs.kuleuven.ac.be

*Research Assistant of the Fund for Scientific Research - Flanders (Belgium)

Abstract. The basic building blocks of Design by Contract are Boolean assertions, which are used to express preconditions, postconditions and invariants. For composed commands, i.e. for commands whose effect is realized using other commands, the expressiveness of such contracts appears to be insufficient. In this paper, we propose an extension of the Design by Contract methodology that enables us to express the specification of a composed method in terms of the specification of the composing methods. Among the main advantages of this approach, we distinguish increased expressiveness of contracts, better consistency and trivial code generation.

Table of Contents

Beyond Design by Contract: Formal Specification of Composed Behaviour	1
<i>Nele Smeets*, Eric Steegmans</i>	
1 Introduction	3
2 Running Example	4
2.1 General Description	4
2.2 Principles	4
2.3 Notation	6
3 Sequential Behaviour Composition	6
3.1 Motivating Example: Composed Methods	7
3.2 Disadvantages	7
3.3 Solution	7
3.4 Design Patterns	10
4 Semantics	10
4.1 Schema Composition in Z	13
4.2 Notation	13
4.3 Schematic Representation	13
4.4 Postcondition	14
4.5 Precondition	15
4.6 Deterministic Methods	16
5 Examples	18
5.1 Independent Deterministic Methods	19
5.2 Dependent Deterministic Methods	23
5.3 Non-Deterministic Followed by Deterministic	25
5.4 Non-Deterministic Methods	29
5.5 Methods That Cannot Be Composed (Deterministic)	35
5.6 Methods That Cannot Be Composed (Non-Deterministic)	37
6 Advantages	39
7 Related Work	40

8	Future Work	41
9	Conclusion	43
10	Appendix: Some Proofs	43
	10.1 Equivalence 1	43
	10.2 Equivalence 2	47

1 Introduction

The concept of Design by Contract (DBC) was introduced by Bertrand Meyer [24] as a way to increase the reliability of object-oriented programs. The basic idea behind DBC is that there is a contract between the supplier and the client of a class, specifying rights and obligations for both parties. The basic ingredients of this contract are preconditions, postconditions and invariants. *Preconditions* of a method specify conditions that must be fulfilled each time a client invokes the method. They are obligations for the client and benefits for the supplier. *Postconditions* of a method define the conditions that must be satisfied upon returning from each invocation. They must only be satisfied if the method has been called under conditions satisfying the preconditions. Postconditions are obligations for the supplier and benefits for the client. *Invariants* are assertions that must be satisfied at all stable times, i.e. after the creation of an object, and before and after each invocation of a public method. Among the major benefits of DBC, we mention improved reliability, clearer and consistent documentation, extended opportunities for reuse and better support for debugging.

In this paper, we combine the ideas of DBC, the specification language Z [28] and Model Driven Architecture (MDA) [8]. First, we identify and illustrate a lack of expressiveness in the DBC methodology when specifying composed commands. All specification languages supporting DBC in the context of object-oriented programs, including JML, Eiffel and OCL, are facing the problems we will describe. In a next step, we will present an alternative way to specify software components, using a mechanism that we call sequential behaviour composition. The concepts supporting this technique are derived from schema composition in Z. We will show that sequential behaviour composition improves the quality of specifications. The specification becomes simpler, and more adaptable. We also claim that the new technique offers opportunities for generating larger fractions of code from declarative specifications. This is particularly interesting in relation to MDA. We will use the Java programming language to explain our ideas and to introduce our concepts. The concepts are also applicable to other object-oriented languages though.

The remainder of this article is structured in the following way: Sect. 2 introduces an example that will be used throughout this article. It also describes the principles and the notation that we will use to write method specifications. Sect. 3

explains how traditional declarative specifications give rise to problems when specifying composed commands. Using an example, it illustrates in an informal way why sequential behaviour composition leads to better specifications. Sect. 4 then describes the precise meaning of sequential behaviour composition by formally expressing the pre- and postconditions of a composed method in terms of the pre- and postconditions of the composing methods. In Sect. 5, the formulas are extensively illustrated by means of different examples. Sect. 6 describes the main advantages of the new methodology and Sect. 8 lists possibilities for further research. Finally, Sect. 7 describes related work and Sect. 9 ends this paper by formulating a conclusion.

2 Running Example

In this paper, we will use the Java programming language to explain our ideas and to introduce our concepts.

2.1 General Description

Fig. 1 introduces the example that we will use throughout this paper. The example describes a Java class, representing accounts. We assume that an account has only one characteristic, namely an integer value representing its balance. (In this example, depositing a large amount of money to an account could cause an overflow in the balance of the account. This problem is ignored, because it is of no interest in this paper.)

This Java class gives a good example of a class contract. The class invariant expresses that the balance of an account is never negative. The postcondition of the withdraw method expresses that the balance of the account is decremented by the given amount. The first precondition of that method demands that only positive amounts of money can be withdrawn. The second precondition of the withdraw method ensures that the balance of the account remains non-negative after the withdrawal; in other words, it guarantees that the class invariant is satisfied after the withdrawal. The preconditions and postcondition of the deposit method are similar.

2.2 Principles

As in Eiffel, we make a clear distinction between queries and commands. Queries return a result but do not produce any side effects. Commands may change the state of the target object as well as the state of other objects involved. Contrary to queries, commands do not return a result. Our example class contains one query, `getBalance`, returning the balance of an account and two commands,

```

/**
 * A class of accounts.
 * @invar The balance of an account is not negative.
 *       | getBalance() >= 0
 */
public class Account {
    /** The balance of this account. */
    private int $balance;
    /**
     * Return the balance of this account.
     * @basic
     */
    public int getBalance() {
        return $balance;
    }
    /**
     * Withdraw the given amount from this account.
     * @param amount The amount to be withdrawn.
     * @pre The given amount is not negative.
     *     | amount >= 0
     * @pre The given amount is smaller than or equal to
     *     the balance of this account.
     *     | amount <= getBalance()
     * @post The balance of this account is decremented by
     *     the given amount.
     *     | this.getBalance()
     *     | == old.this.getBalance() - amount
     */
    public void withdraw(final int amount) {
        $balance = $balance - amount;
    }
    /**
     * Deposit the given amount to this account.
     * @param amount The amount to be deposited.
     * @pre The given amount is not negative.
     *     | amount >= 0
     * @post The balance of this account is incremented by
     *     the given amount.
     *     | this.getBalance()
     *     | == old.this.getBalance() + amount
     */
    public void deposit(final int amount) {
        $balance = $balance + amount;
    }
}

```

Fig. 1. Running example.

`withdraw` and `deposit`, that can be used to withdraw and deposit a certain amount of money.

We further distinguish between basic queries and derived queries. Basic queries are chosen among the set of all queries in such a way that the entire state of an object can be inspected using basic queries only. The result of derived queries, and the effect of commands and constructors can then be specified in terms of the basic queries. In our example, we have only one basic query, namely `getBalance`, as indicated by the `@basic`-tag. The class of accounts contains no derived queries. The principles explained in this subsection are motivated and described in more detail in [25] and [30].

2.3 Notation

In the context of Java, it is customary to work out the contract of a class and its members in terms of documentation comments, which can be processed by Javadoc [18]. Ingredients of the documentation are preceded by tags, imposing a rigid structure on the documentation at hand. Throughout this paper, we will use the following tags:

- `@invar`: Specifies a class invariant.
- `@basic`: Is used to distinguish basic queries from derived queries.
- `@param`: Describes a method parameter.
- `@pre`: Specifies a method precondition.
- `@post`: Specifies a postcondition of a command or constructor.
- `@return`: Specifies the result of a derived query.

The preconditions, postconditions and invariants in our example are described both informally and formally. The informal description is given in natural language. For the formal description, a very simple specification language is used. Conditions are expressed using Boolean Java expressions, in which the following special constructs may be used:

- The keyword `old` is used to refer to the pre-state value of an expression. (For example, the postcondition of the `deposit` method in Fig. 1 expresses that the value of the `balance` is equal to the value of the `balance` in the pre-state, incremented by the given amount.)
- A conditional choice is expressed using the following construct: `if (Boolean assertion) then assertion else assertion`. The `else`-part is optional.

3 Sequential Behaviour Composition

The existing DBC tools and languages describe the pre- and postconditions of a command in a declarative way, using Boolean assertions. In this section, we identify situations in which traditional declarative specifications are inadequate and present a proposal to improve such specifications.

3.1 Motivating Example: Composed Methods

Consider the `transferTo` method, a method for transferring money from one account to another account. The `transferTo` method is a composed command: its effect is realized in terms of the commands `withdraw` and `deposit`. The traditional way of specifying the behaviour of the `transferTo` method is shown in Fig. 2. In the following section, we identify some disadvantages of this specification.

3.2 Disadvantages

The declarative specification of the composed method `transferTo` has some disadvantages.

First of all, the specification duplicates the pre- and postconditions of the `withdraw` and `deposit` method. In particular, the first and second precondition and the first postcondition of the composed method are almost directly taken from the specification of the composing methods. This duplication may lead to consistency problems, arising when the specification of the composing methods changes.

Further, the specification of the `transferTo` method does not express any connection between the composed method and the composing methods, losing the essential semantics of the composed method. (Notice that, in the implementation of the `transferTo` method, this bond is not lost.)

Finally, the resulting specification is rather formidable: a more succinct description would be desirable.

Similar problems emerge in the traditional specification of all composed commands, i.e. of all commands whose effect is described in terms of other commands.

3.3 Solution

To overcome the above problems, our proposal is to apply the idea of schema composition, introduced in the *Z* specification language [28], to the specification of composed commands. Schema composition in *Z* is used to specify an operation as a composition of other operations.

Applying the idea of behaviour composition to the `transferTo` method leads to the specification in Fig. 3. A specific tag (`@effect`) is used to distinguish traditional specifications from specifications of composed commands. The `;`-notation is taken from *Z*.

The specification in Fig. 3 explicitly says that the `transferTo` method is a sequence of two other methods, thereby capturing the essential meaning of the `transferTo` method. The precise semantics of sequential behaviour composition is further explained in the next section.

```

/**
 * Transfer the given amount from this account to the
 * given account, i.e. withdraw the given amount from this
 * account and deposit it to the given account.
 *
 * @param amount
 *         The amount to be transferred.
 * @param destination
 *         The target account for the transfer.
 * @pre    The given amount is not negative.
 *         | amount >= 0
 * @pre    The given amount is smaller than or equal to
 *         the balance of this account.
 *         | amount <= getBalance()
 * @pre    The given target account is effective.
 *         | destination != null
 * @post   If the given target account is different from
 *         this account, then the balance of this account
 *         is decremented by the given amount and the
 *         balance of the given target account is
 *         incremented by the given amount.
 *         | if (this != destination)
 *         |   then
 *         |     ( this.getBalance()
 *         |       == old.this.getBalance() - amount ) &&
 *         |     ( destination.getBalance()
 *         |       == old.destination.getBalance() + amount )
 * @post   If this account and the given target account
 *         are the same account, then the balance of
 *         this account is left unchanged.
 *         | if (this == destination)
 *         |   then
 *         |     this.getBalance() == old.this.getBalance()
 */
public void transferTo(final int amount, final Account
                      destination) {
    this.withdraw(amount);
    destination.deposit(amount);
}

```

Fig. 2. Traditional declarative specification of the transferTo method.

```

/**
 * Transfer the given amount from this account to the
 * given account, i.e. withdraw the given amount from this
 * account and deposit it to the given account.
 *
 * @param amount
 *         The amount to be transferred.
 * @param destination
 *         The target account for the transfer.
 * @pre    The given target account is effective.
 *         | destination != null
 * @effect First withdraw the given amount from this
 *         account, then deposit the same amount to the
 *         destination account.
 *         | this.withdraw(amount);destination.deposit(amount)
 */
public void transferTo(final int amount, final Account
                      destination) {
    this.withdraw(amount);
    destination.deposit(amount);
}

```

Fig. 3. Specification of the effect of the transferTo method using sequential behaviour composition.

Although we only gave an informal description of sequential behaviour composition, we can see that all disadvantages of the traditional specification have disappeared. Introducing sequential behaviour composition increases the expressiveness of our specification language: we are now able to explicitly express the connection between the composed method and its composing methods. The pre- and postconditions of the withdraw and deposit method are no longer duplicated and the consistency problems that were a consequence of this duplication disappear accordingly. Finally, the specification using sequential behaviour composition in Fig. 3 is much more succinct than the traditional specification in Fig. 16.¹

3.4 Design Patterns

In software engineering, design patterns are standard solutions to common problems in software design. The Gang of Four (GOF) patterns [9] are generally considered the foundation for all other patterns.

In [9], 15 of the 23 design patterns make use of composed methods. This composition can be expressed in code, but cannot be expressed in specification, creating a gap between specification and implementation. The only way to circumvent this lack of expressiveness in class contracts is to duplicate the specification of the composing methods, as illustrated in Sect. 3.1. But even this duplication is not always possible: when one of the composing methods is abstract and has an incomplete specification, duplication makes no sense. In [9], 12 of the 15 design patterns that use composed methods make use of delegation to an abstract class.

A concrete example of a design pattern that uses composed methods is the Observer pattern. The specification and implementation of this pattern is shown in Figs. 4 and 5. The question marks in the specification indicate the lack of expressiveness of the DBC methodology.

4 Semantics

In this section, we will describe the precise semantics of a sequential behaviour composition. We will do this by writing the precondition and postcondition of a sequential behaviour composition in terms of the pre- and postconditions of the composing methods. The semantics of sequential behaviour composition is based on the ideas of schema composition in Z [28]. The general expansion given in this section will be illustrated extensively in subsequent sections by means of various examples.

¹ Remark: In addition to the (implicit) preconditions implied by the sequential behaviour composition, the specification in Fig. 3 contains an extra precondition, expressing that the given target account should be effective. This precondition is introduced to guarantee that the deposit method is applied to an effective account.

```

/**
 * A class of subjects referencing a collection of
 * observers.
 * @invar The collection of observers is effective.
 *       | getObservers() != null
 * @invar The observers are effective.
 *       | for each o in getObservers():
 *       |   o != null
 */
public abstract class Subject {
    /**
     * The collection of observers.
     */
    private Collection observers = new HashSet();
    /**
     * Return the collection of observers.
     * @basic
     */
    public Collection getObservers() {
        return new HashSet(observers);
    }
    /**
     * Notify all observers.
     * @post (without sequential behaviour composition)
     *       | ???
     * @post (with sequential behaviour composition)
     *       | for each o in getObservers():
     *       |   o.update()
     */
    public void notifyObservers() {
        Iterator i = getObservers().iterator();
        while (i.hasNext()) {
            Observer o = (Observer) i.next();
            o.update();
        }
    }
    /**
     * Attach the given observer to this subject.
     *
     * @param observer
     *       An observer
     * @pre The given observer is effective.
     *     | observer != null
     * @post The given observer is added to the collection
     *     of observers.
     *     | getObservers().contains(observer)
     */
    public void attach(Observer observer) {
        observers.add(observer);
    }
}

```

Fig. 4. Subject

```

/**
 * A class of observers.
 */
public abstract class Observer {
    /**
     * Update this observer.
     * @post True
     *      | true
     */
    public abstract void update();
}
/**
 * A class of concrete observers.
 */
public class ConcreteObserver extends Observer {
    /**
     * Update this observer.
     * @post ...
     *      | ...
     */
    public void update() {
        System.out.println("updating concrete observer: "+this);
    }
}

```

Fig. 5. Observer

4.1 Schema Composition in Z

The semantics of sequential behaviour composition is based on the ideas of schema composition in Z, so we first explain the semantics of the latter in more detail. If, in Z, an operation A is defined to be the composition of operations B and C, this means that if B can cause a change of state from say S1 to S2 and C can cause a change of state from S2 to S3, then A can cause a change of state from S1 to S3. Thus any change of state described by A can be thought of as a two-stage process which passes through an intermediate state which arises as a result of operation B and serves as starting point for operation C.

4.2 Notation

Before we can explain the precise semantics of a sequential behaviour composition, we first need to introduce some additional notation.

While the specification of an ordinary command involves only two states, namely the pre-state and the post-state, a sequential behaviour composition also involves an intermediate state. An object in the pre-state is represented by the keyword `old` followed by a period and its name (e.g. `old.this`, `old.a`); an object in the intermediate state is represented by the keyword `interm` followed by a period and its name (e.g. `interm.this`, `interm.a`); an object in the post-state is represented by its name (e.g. `this`, `a`).

Suppose that A is a class containing a method `m1`, `a` is an object of type T, the number of arguments of `m1` is equal to `n`, and `p1, ..., pn` are objects whose type can be converted by method invocation conversion to the type of the corresponding formal parameter of `m1`. Then we define `post(a.m1(p1, ..., pn))` as the postcondition of `m1` in T where 'this' is substituted by `a` and the formal parameters of `m1` are substituted by `p1, ..., pn`. In a similar way, we introduce the predicate `pre(a.m1(p1, ..., pn))`.

The notation `[post/interm]` and `[old/interm]` represents a substitution. For example, `[old/interm]` means that all pre-state variables should be replaced by the corresponding intermediate state variables. So, `old.this[old/interm]` evaluates to `interm.this`.

4.3 Schematic Representation

The knowledge and notation introduced in the previous sections will help us to derive the pre- and postcondition of a sequential behaviour composition of the form `a.m1(p1, ..., pn); b.m2(q1, ..., qm)`, where `a.m1(p1, ..., pn)` and `b.m2(q1, ..., qm)` are method invocation expressions [10, 15.12] not containing the keyword `super`. A schematic representation is given in Fig. 6.

Intuitively, sequential behaviour composition expresses that two methods are executed in sequence, i.e. one after the other. In other words, we have a two-step process:

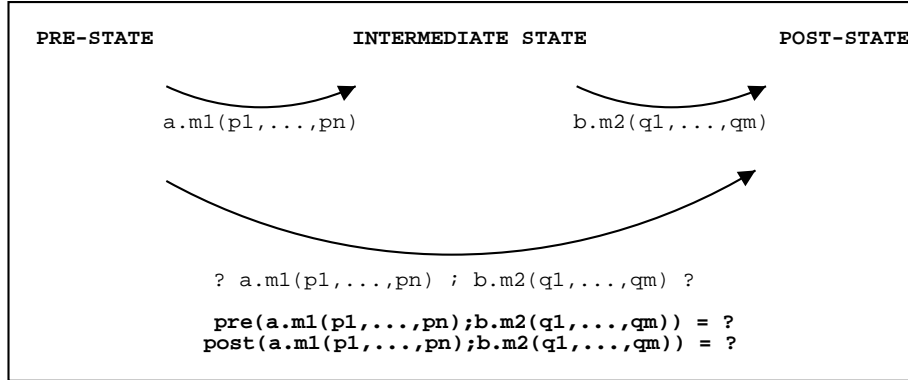


Fig. 6. A schematic representation of a sequential behaviour composition of the form $a.m1(p1, \dots, pn); b.m2(q1, \dots, qm)$.

1. We start in a pre-state. The first method is executed and gives rise to an intermediate state.
2. In this intermediate state, the second method is invoked.

In the following sections, we will derive the precondition and postcondition of the sequential behaviour composition in Fig. 6.

4.4 Postcondition

We start by deriving a formula for the postcondition of the sequential behaviour composition $a.m1(p1, \dots, pn) ; b.m2(q1, \dots, qm)$. The formula for the postcondition is straightforward: any change of state described by the composed command can be thought of as a two-stage process which passes through an intermediate state that arises as a result of method $m1$ and serves as a starting point for method $m2$. So, we arrive at the following formula:

$$\begin{aligned}
 & post(a.m1(p1, \dots, pn) ; b.m2(q1, \dots, qm)) \\
 &= \exists \text{ interm.State } | \\
 &\quad post(a.m1(p1, \dots, pn))[post/interm] \ \&\& \\
 &\quad post(b.m2(q1, \dots, qm))[old/interm]
 \end{aligned}$$

The postcondition of the composed command is computed by taking the conjunction of the postconditions of the composing commands and adding an existential quantification over the intermediate state.

In theory, it will not always be possible to simplify the quantified expression above to a simple postcondition. In practice, the expression frequently can be simplified to a much neater, but logically equivalent postcondition, as illustrated in Sect. 5..

The formula given above does not always make sense. For example, invoking $m1$ is only allowed when the precondition of $m1$ is satisfied and $m2$ can only be invoked in intermediate states satisfying the precondition of $m2$. In the next section, we will derive an appropriate precondition for the sequential behaviour composition.

4.5 Precondition

Deriving and understanding a formula for the precondition of the sequential behaviour composition $a.m1(p1, \dots, pn) ; b.m2(q1, \dots, qm)$ is more difficult. We will do this using Fig. 7.

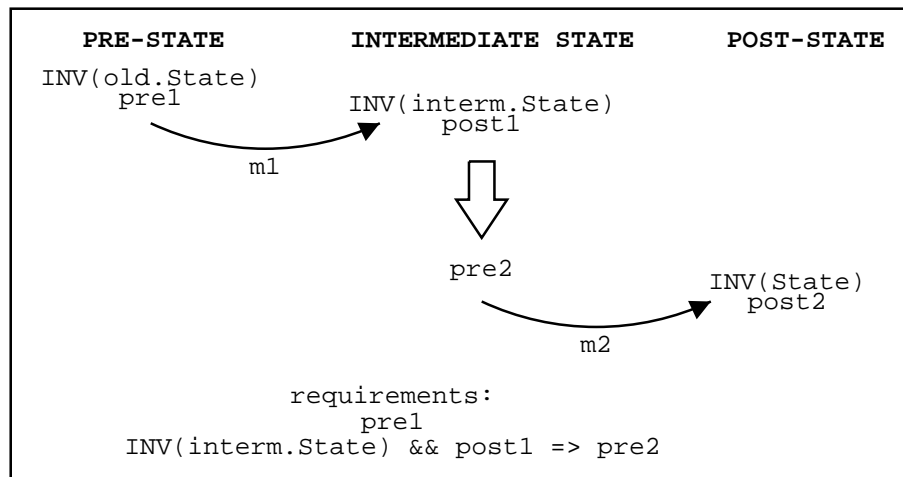


Fig. 7. A schema explaining the meaning of the precondition of a sequential behaviour composition.

Invoking the composed command $a.m1(p1, \dots, pn) ; b.m2(q1, \dots, qm)$ is only allowed when the following two conditions are satisfied:

- First of all, we can only invoke the first composing method $m1$ in the pre-state, when the precondition of $m1$ is satisfied. More correct, we are only certain that $m1$ will satisfy its postcondition when the precondition of $m1$ is true.
- Second, we can only invoke the second composing method $m2$ in the intermediate state, when the precondition of $m2$ is satisfied. So, all possible intermediate states should satisfy the precondition of $m2$. The set of all intermediate states that can be reached when we invoke $m1$ in a pre-state satisfying the precondition of $m1$ is described by the conjunction of the postcondition of

`m1` and the invariant. So, all intermediate states satisfying the postcondition of `m1` and the invariant should satisfy the precondition of `m2`.

The two conditions described above are expressed in the following formula:

```
pre(a.m1(p1, ..., pn) ; b.m2(q1, ..., qm))
= pre(a.m1(p1, ..., pn)) &&
  (∀ interm.State |
   ( post(a.m1(p1, ..., pn))[post/interm]
     &&
     inv(interm.State)
     ⇒
     pre(b.m2(q1, ..., qm))[old/interm]
   )
)
```

In the above formula, the variable `State` represents all elements of the state of all objects that are possibly influenced by the composed command. These elements are referred to as the state space for the operation.

The conditions described above are necessary conditions that have to be satisfied when the composed method is invoked. They are also sufficient conditions:

1. When we invoke the composed method in a pre-state that satisfies the precondition given above, then the precondition of `m1` is satisfied automatically. This means that `m1` is automatically invoked in a state satisfying the precondition of `m1`.
2. When we invoke the composed method in a pre-state that satisfies the precondition given above, then we know that all intermediate states that satisfy the postcondition of `m1` and the invariant also satisfy the precondition of `m2`. In other words, all intermediate states that can be reached when we invoke `m1` in a pre-state satisfying the precondition given above, will satisfy the precondition of `m2`. This means that `m2` is always invoked in an intermediate state satisfying the precondition of `m2`.

In theory, it will not always be possible to simplify the complicated universally quantified expression above to a simple precondition. In practice, the expression frequently can be simplified to a much neater, but logically equivalent precondition, as illustrated in Sect. 5.

4.6 Deterministic Methods

When the first composing method is deterministic, the formula for the precondition (introduced in Sect. 4.5) can be simplified:

```

pre(a.m1(p1,...,pn) ; b.m2(q1,...,qm))
= pre(a.m1(p1,...,pn)) &&
  ∀ interm.State |
  ( post(a.m1(p1,...,pn))[post/interm]
    &&
    inv(interm.State)
    ⇒
    pre(b.m2(q1,...,qm))[old/interm]
  )
= // since m1 is a deterministic method, its post-state
  // can be written as a function of its pre-state, because
  // there is exactly one post-state value corresponding to
  // each pre-state value
pre(a.m1(p1,...,pn)) &&
  ∀ interm.State |
  ( interm.State == f(old.State) && inv(interm.State)
    ⇒
    pre(b.m2(q1,...,qm))[old/interm]
  )
= // introduce the value for interm.State (given at the
  // left-hand side of the implication) in the right-hand
  // side
pre(a.m1(p1,...,pn)) &&
  ∀ interm.State |
  ( interm.State == f(old.State) && inv(interm.State)
    ⇒
    pre(b.m2(q1,...,qm))[old.State/f(old.State)]
  )
= // use the following law:
  // ∀ x : T | (p ⇒ q)
  // ⇔
  // ( ∃ x : T | p ) ⇒ q
  // where x is not free in q
pre(a.m1(p1,...,pn)) &&
  ( ∃ interm.State |
    interm.State == f(old.State) && inv(interm.State)
  )
⇒
pre(b.m2(q1,...,qm))[old.State/f(old.State)]
= // the existential quantification over the intermediate
  // state is removed and interm.State is replaced by an
  // expression not containing any intermediate state
  // values
pre(a.m1(p1,...,pn)) &&
  inv(f(old.State))
⇒
pre(b.m2(q1,...,qm))[old.State/f(old.State)]

```

So, when the first composing method is deterministic, the precondition of the composed method can be computed using the following, more simple, formula:

```
pre(a.m1(p1, ..., pn) ; b.m2(q1, ..., qm))
= pre(a.m1(p1, ..., pn)) &&
  inv(f(old.State))
⇒
pre(b.m2(q1, ..., qm))[old.State/f(old.State)]
```

This formula can be further simplified when we combine it with the invariant of the pre-state (resulting in the condition that should be fulfilled when invoking the composed method):

```
pre(a.m1(p1, ..., pn) ; b.m2(q1, ..., qm)) &&
inv(old.State)
= ( pre(a.m1(p1, ..., pn)) &&
  inv(f(old.State))
  ⇒
  pre(b.m2(q1, ..., qm))[old.State/f(old.State)]
)
&&
inv(old.State)
= // when the precondition of the first composing method
  // and the invariant are fulfilled, the corresponding
  // post-state value, f(old.State), will satisfy the
  // invariant.
  // in symbols: pre(a.m1(p1, ..., pn)) && inv(old.State)
  // ⇒ inv(f(old.State))
  // this formula can be used to remove the implication
pre(a.m1(p1, ..., pn)) &&
pre(b.m2(q1, ..., qm))[old.State/f(old.State)]
inv(old.State)
```

So, under the assumption that the invariant `inv(old.State)` is true and that the first composing method is deterministic, the precondition of the composed method is equal to the conjunction of the precondition of the first method and the precondition of the second method, evaluated in the corresponding post-state of the first composing method.

5 Examples

In this section, we compute the preconditions and postconditions of various composed methods, using the formulas introduced in Sect. 4.

The following examples are examined:

- Sect. 5.1: two deterministic methods; the precondition of the second method is independent of the intermediate state
- Sect. 5.2: two deterministic methods; the precondition of the second method depends on the intermediate state
- Sect. 5.3: a non-deterministic method followed by a deterministic method
- Sect. 5.4: two non-deterministic methods
- Sect. 5.5: two deterministic methods that cannot be composed
- Sect. 5.6: a deterministic and a non-deterministic method that cannot be composed

In each case, we describe the state space and we compute the precondition and postcondition of the composed method. Moreover, a schematic representation of the sequential behaviour composition is given to provide a more intuitive understanding of the composed method.

5.1 Independent Deterministic Methods

In this section, we expand the effect clause of the `transferTo` method in Fig. 3 by using the formulas introduced in Sect. 4. The `transferTo` method is a composed method; the two composing methods are deterministic, and the precondition of the second method is independent of the intermediate state.

5.1.1 State The `transferTo` method involves two objects (`this` and `destination`). The state space for the method therefore consists of `this.getBalance()` and `destination.getBalance()`.

5.1.2 Precondition The precondition of the `transferTo` method is computed using the formula for deterministic methods introduced in Sect. 4.6:

```
pre(this.transferTo(amount, destination))
= pre(this.withdraw(amount) ; destination.deposit(amount))
= pre(this.withdraw(amount)) &&
  inv(f(old.State))
⇒
pre(destination.deposit(amount))[old.State/f(old.State)]
= // fill in
  amount >= 0 && amount <= old.this.getBalance() &&
  inv(f(old.State))
⇒
  amount >= 0
= // ( amount >= 0 ) == true
  amount >= 0 && amount <= old.this.getBalance() &&
  inv(f(old.State))
```

```

=>
true
= // ( p => true ) == true
  amount >= 0 && amount <= old.this.getBalance() &&
  true
= // ( p && true ) == p
  amount >= 0 && amount <= old.this.getBalance()

```

(Since the precondition of the deposit method is independent of the state, we do not have to compute $f(\text{old.State})$ explicitly, saving time and space. We do not have to compute $\text{inv}(f(\text{old.State}))$ either, because this expression disappears during the computation.)

This precondition corresponds to the precondition given in the declarative specification of the transferTo method (see Fig. 2).

5.1.3 Postcondition The postcondition of the transferTo method is given by:

```

post(this.transferTo(amount, destination))
= post(this.withdraw(amount); destination.deposit(amount))
= ∃ interm.this.getBalance(),
  interm.destination.getBalance() |
  post(this.withdraw(amount))[post/interm] &&
  post(destination.deposit(amount))[old/interm]
= // fill in
  ∃ interm.this.getBalance(),
  interm.destination.getBalance() |
  { [ interm.this.getBalance()
    == old.this.getBalance() - amount &&
    ( ( this == destination )
      ||
      ( this != destination &&
        interm.destination.getBalance()
        == old.destination.getBalance()
      )
    )
  ] &&
  [ destination.getBalance()
    == interm.destination.getBalance() + amount &&
    ( ( this == destination )
      ||
      ( this != destination &&
        this.getBalance() == interm.this.getBalance()
      )
    )
  ]
}

```

```

= // distinguish the cases where (this == destination) and
  // (this != destination)
  ∃ interm.this.getBalance(),
    interm.destination.getBalance() |
  { ( this == destination &&
      interm.this.getBalance()
      == old.this.getBalance() - amount &&
      destination.getBalance()
      == interm.destination.getBalance() + amount
    )
    ||
    ( this != destination &&
      interm.this.getBalance()
      == old.this.getBalance() - amount &&
      interm.destination.getBalance()
      == old.destination.getBalance() &&
      this.getBalance() == interm.this.getBalance() &&
      destination.getBalance()
      == interm.destination.getBalance() + amount
    )
  }
= // - the disjunction within the existential quantification
  // is split under separate quantifications
  // - destination is substituted by this when
  // (this == destination)
  ∃ interm.this.getBalance(),
    interm.destination.getBalance() |
  ( this == destination &&
    interm.this.getBalance()
    == old.this.getBalance() - amount &&
    this.getBalance() == interm.this.getBalance() + amount
  )
  ||
  ∃ interm.this.getBalance(),
    interm.destination.getBalance() |
  ( this != destination &&
    interm.this.getBalance()
    == old.this.getBalance() - amount &&
    interm.destination.getBalance()
    == old.destination.getBalance() &&
    this.getBalance() == interm.this.getBalance() &&
    destination.getBalance()
    == interm.destination.getBalance() + amount
  )
= // the existential quantification is removed and
  // interm.this.getBalance() and
  // interm.destination.getBalance() are replaced by
  // expressions not containing any intermediate state
  // values;
  ( this == destination &&

```

```

    this.getBalance()
    == ( old.this.getBalance() - amount ) + amount
  )
  ||
  ( this != destination &&
    this.getBalance() == old.this.getBalance() - amount &&
    destination.getBalance()
    == old.destination.getBalance() + amount
  )
= // rewrite the expression using if (...) then ... else ...
  if (this != destination)
  then
    this.getBalance() == old.this.getBalance() - amount &&
    destination.getBalance()
    == old.destination.getBalance() + amount
  else
    this.getBalance() == old.this.getBalance()

```

This postcondition corresponds to the postcondition given in the declarative specification of the transferTo method (see Fig. 2).

5.1.4 Conclusion The preconditions derived in Sect. 5.1.2 correspond to the preconditions in Fig. 2. The postconditions derived in Sect. 5.1.3 correspond to the postconditions in Fig. 2. In other words, the succinct specification using sequential behaviour composition in Fig. 3 has the same meaning as the traditional declarative specification in Fig. 2.

5.1.5 Schema A schematic representation of the transferTo method is given in Fig. 8.

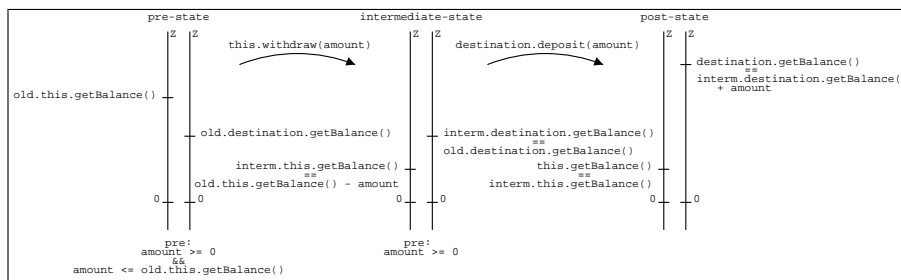


Fig. 8. Schematic representation of the transferTo method.

5.2 Dependent Deterministic Methods

The composed command `transferTo` is somewhat special, because both composing methods are deterministic and because the precondition of the second composing method (the `deposit` method) is independent of the state. As a consequence, the precondition of the composed method is simply computed by conjoining the preconditions of the composing methods.

In this section, we examine a composed method where both composing methods are deterministic and where the precondition of the second composing method is not independent of the state.

5.2.1 Definition Let us introduce a command, `doubleWithdraw`, for withdrawing the same amount of money twice from a given account. The specification of this command is given in Fig. 9.

```
/**
 * Withdraw the given amount twice from this account.
 *
 * @param amount
 *         The amount to be withdrawn twice.
 * @effect Withdraw the given amount twice from this
 *         account.
 *         | this.withdraw(amount) ; this.withdraw(amount)
 */
public void doubleWithdraw(final int amount) {
    this.withdraw(amount);
    this.withdraw(amount);
}
```

Fig. 9. Specification of the effect of the `doubleWithdraw` method using sequential behaviour composition.

Notice that, using the current semantics of the `withdraw` method, this corresponds to withdrawing an amount that is twice the given amount:

```
@effect Apply the withdraw method with as argument
        twice the given amount.
        | this.withdraw(2*amount)
```

(Notice that the effect clause is used here to refer to one command, instead of a sequence of two commands.)

However, when the postcondition of the withdraw method changes, for example by charging a fee for each withdrawal, the two effect-clauses are no longer identical:

- applying the withdraw method twice will charge the fee twice
- applying the withdraw method once with the doubled amount will only charge the fee one time

We will now compute the precondition and postcondition of the `doubleWithdraw` method described in Fig. 9.

5.2.2 State The `doubleWithdraw` method involves only one object (`this`) so the state space is given by `this.getBalance()`.

5.2.3 Precondition Since the first composing method of the `doubleWithdraw` method (the `withdraw` method) is deterministic, we can use the simplified formula derived in Sect. 4.6 to compute the precondition:

```
pre(this.doubleWithdraw(amount))
= pre(this.withdraw(amount) ; this.withdraw(amount))
= pre(this.withdraw(amount)) &&
  inv(f(old.this.getBalance()))
⇒
pre(this.withdraw(amount))
 [old.this.getBalance()/f(old.this.getBalance())]
= pre(this.withdraw(amount)) &&
  inv(old.this.getBalance()-amount)
⇒
pre(this.withdraw(amount))
 [old.this.getBalance()/old.this.getBalance()-amount]
= // fill in
  amount >= 0 && amount <= old.this.getBalance() &&
  old.this.getBalance() - amount >= 0
⇒
  amount >= 0 && amount <= old.this.getBalance() - amount
= // left-hand side of implication is true
  amount >= 0 && amount <= old.this.getBalance() &&
  true
⇒
  amount >= 0 && amount <= old.this.getBalance() - amount
= // (true ⇒ p) == p
  amount >= 0 && amount <= old.this.getBalance() &&
  amount >= 0 && amount <= old.this.getBalance() - amount
= // simplify
  amount >= 0 && 2 * amount <= old.this.getBalance()
```

The resulting precondition corresponds to what could be expected intuitively.

5.2.4 Postcondition The postcondition of the `doubleWithdraw` method is given by:

```
post(this.doubleWithdraw(amount))
= post(this.withdraw(amount) ; this.withdraw(amount))
= ∃ interm.this.getBalance() |
  post(this.withdraw(amount))[post/interm] &&
  post(this.withdraw(amount))[old/interm]
= // fill in
  ∃ interm.this.getBalance() |
  interm.this.getBalance() == old.this.getBalance() - amount
  &&
  this.getBalance() == interm.this.getBalance() - amount
= // the existential quantification is removed and
  // interm.this.getBalance() is replaced by an expression
  // not containing any intermediate state values
  this.getBalance() == (old.this.getBalance()-amount)-amount
= // rewriting
  this.getBalance() == old.this.getBalance() - 2 * amount
```

The resulting condition expresses that the `doubleWithdraw` method decreases the balance of the account by twice the given amount.

5.2.5 Conclusion The pre- and postconditions computed in the preceding sections give rise to the declarative specification shown in Fig. 10.

This example can be generalized to methods that withdraw a given amount N times, where N is a positive integer value.

5.2.6 Schema A schematic representation of the `doubleWithdraw` method is given in Fig. 11.

5.3 Non-Deterministic Followed by Deterministic

In this section and the following section, we examine the influence of non-determinism. In this section, we consider an example in which only one of the composing methods is non-deterministic. In the next section, we consider an example in which both composing methods are non-deterministic.

5.3.1 Definition Consider the non-deterministic command in Fig. 12. The postcondition of the command says that the balance of the account will be at least doubled, but it does not specify what the resulting balance will be exactly.

Now consider a composed method `nonDet`, that first increases the balance of an account, using the command in Fig. 12 and then withdraws a given amount of money using the `withdraw` method. The specification of this method is given in Fig. 13.

```

/**
 * Withdraw the given amount twice from this account.
 *
 * @param amount
 *         The amount that has to be withdrawn twice.
 * @pre    The given amount is not negative.
 *         | amount >= 0
 * @pre    Twice the given amount is smaller than or equal
 *         to the balance of this account.
 *         | 2*amount <= getBalance()
 * @post   The balance of this account is decremented by
 *         twice the given amount.
 *         | this.getBalance()
 *         | == old.this.getBalance() - 2*amount
 */
public void doubleWithdraw(final int amount) {
    this.withdraw(amount);
    this.withdraw(amount);
}

```

Fig. 10. Traditional declarative specification of the doubleWithdraw method.

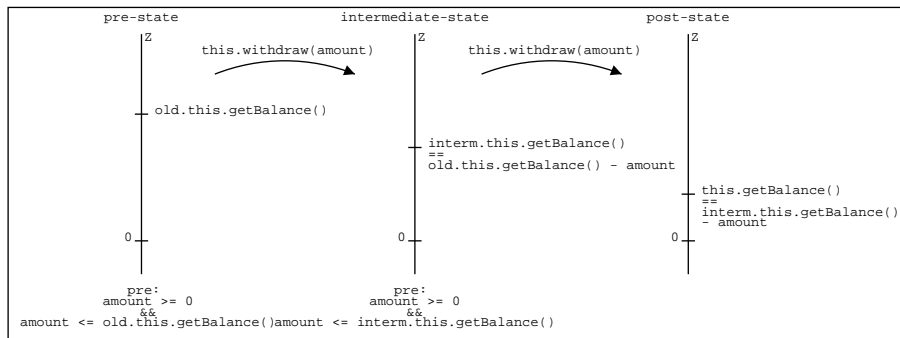


Fig. 11. Schematic representation of the doubleWithdraw method.

```

/**
 * Increase the balance of this account.
 *
 * @post   The balance of this account is at least doubled.
 *         | getBalance() > old.getBalance() * 2
 */
public void increase() {
    ...;
}

```

Fig. 12. Declarative specification of the increase method.

```

/**
 * First increase the balance of this account, and then
 * withdraw the given amount.
 *
 * @param  amount
 *         The amount to be withdrawn.
 * @effect First apply the increase method. Then apply
 *         the withdraw method, with the given amount as
 *         its argument.
 *         | this.increase() ; this.withdraw(amount)
 */
public void nonDet(final int amount) {
    this.increase();
    this.withdraw(amount);
}

```

Fig. 13. Specification of the effect of the nonDet method using sequential behaviour composition.

5.3.2 State The nonDet method involves only one object (`this`) so the state is given by `this.getBalance()`.

5.3.3 Precondition The precondition of the nonDet method is given by:

```

pre(this.nonDet(amount))
= pre(this.increase() ; this.withdraw(amount))
= pre(this.increase()) &&
  ∀ interm.this.getBalance() |
  ( post(this.increase())[post/interm]
    &&
    inv(interm.this.getBalance())
    ⇒
    pre(this.withdraw(amount))[old/interm]
  )
= // fill in
  true &&
  ∀ interm.this.getBalance() |
  ( interm.this.getBalance() > 2 * old.this.getBalance()
    &&
    interm.this.getBalance() >= 0
    ⇒
    amount >= 0 && amount <= interm.this.getBalance()
  )
= // ( true && p ) == p
  ∀ interm.this.getBalance() |
  ( interm.this.getBalance() > 2 * old.this.getBalance()
    &&
    interm.this.getBalance() >= 0
    ⇒
    amount >= 0 && amount <= interm.this.getBalance()
  )
= // see appendix
  ( amount >= 0
    &&
    2 * old.this.getBalance() + 1 >= amount
  )
  ||
  ( 2 * old.this.getBalance() + 1 < 0
    &&
    amount == 0
  )

```

²Notice that, when this precondition is combined with the invariant, saying that `old.this.getBalance()` is not negative, the right-hand side of the disjunction will never be true; so, *under the assumption that the invariant is true, the precondition simplifies to*

² The last step in the above computation is worked out in more detail in Sect. 10.1.

```

pre(this.nonDet(amount))
= amount >= 0 && 2 * old.this.getBalance() + 1 >= amount

```

This precondition expresses that the given amount should be (1) non-negative, and (2) smaller than or equal to the smallest possible balance that the increase method will set the balance to, i.e. not bigger than $2 * \text{old.this.getBalance}() + 1$.

5.3.4 Postcondition The postcondition of the nonDet method is given by:

```

post(this.nonDet(amount))
= post(this.increase() ; this.withdraw(amount))
= ∃ interm.this.getBalance() |
  post(this.increase())[post/interm] &&
  post(this.withdraw(amount))[old/interm]
= // fill in
  ∃ interm.this.getBalance() |
  interm.this.getBalance() > old.this.getBalance() * 2 &&
  this.getBalance() == interm.this.getBalance() - amount
= // rewrite the second expression
  ∃ interm.this.getBalance() |
  interm.this.getBalance() > old.this.getBalance() * 2 &&
  interm.this.getBalance() == this.getBalance() + amount
= // the existential quantification is removed and
  // interm.this.getBalance() is replaced by an expression
  // not containing any intermediate state values
  this.getBalance() + amount > old.this.getBalance() * 2
= // rewrite
  this.getBalance() > old.this.getBalance() * 2 - amount

```

The resulting balance will be bigger than twice the original balance, decremented by the given amount.

5.3.5 Conclusion The pre- and postconditions computed in the preceding sections give rise to the declarative specification shown in Fig. 14.

5.3.6 Schema A schematic representation of the nonDet method is given in Fig. 15.

5.4 Non-Deterministic Methods

In this section, we examine an example in which both composing methods are non-deterministic.

```

/**
 * First increase the balance of this account, and then
 * withdraw the given amount.
 *
 * @param amount
 *         The amount to be withdrawn.
 * @pre    The given amount is not negative.
 *         | amount >= 0
 * @pre    The given amount is smaller than or equal to
 *         twice the balance plus one.
 *         | amount <= 2 * this.getBalance() + 1
 * @post   The balance is bigger than twice the old
 *         balance minus the given amount.
 *         | this.getBalance()
 *         | > old.this.getBalance() * 2 - amount
 */
public void nonDet(final int amount) {
    this.increase();
    this.withdraw(amount);
}

```

Fig. 14. Traditional declarative specification of the nonDet method.

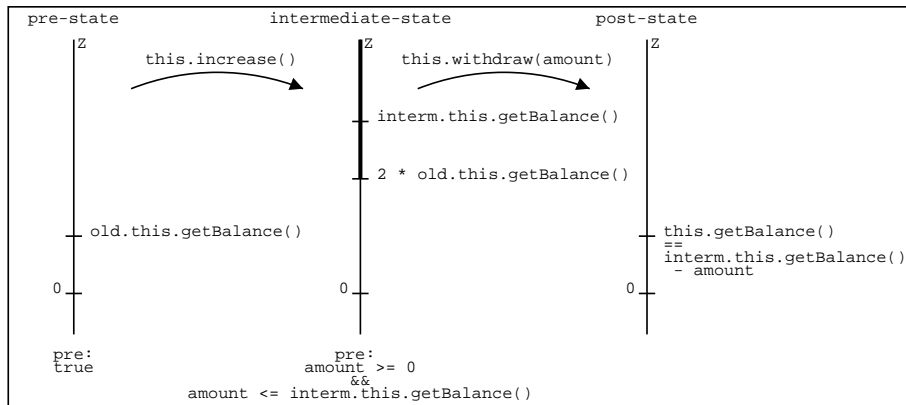


Fig. 15. Schematic representation of the nonDet method.

5.4.1 Definition Consider the following command:

```
/**
 * Withdraw at least the given amount from this account.
 *
 * @param amount
 *         The amount to be withdrawn.
 * @pre   The given amount is not negative.
 *         | amount >= 0
 * @pre   The given amount is smaller than or equal to
 *         the balance of this account.
 *         | amount <= getBalance()
 * @post  The balance of this account is at least
 *         decremented by the given amount.
 *         | this.getBalance()
 *         | <= old.this.getBalance() - amount
 */
public void withdrawAtLeast(final int amount) {
    ...
}
```

Now consider the composed command

```
doubleWithdrawAtLeast(amount1, amount2)
= withdrawAtLeast(amount1) ; withdrawAtLeast(amount2)
```

A precondition for this composed command should guarantee that the precondition of `withdrawAtLeast(amount2)` is fulfilled after invocation of `withdrawAtLeast(amount1)`. But `withdrawAtLeast(amount1)` can make the balance as small as it wants, possibly zero, so the only way to guarantee that the precondition of `withdrawAtLeast(amount2)` is fulfilled is to require that `amount2` is zero.

5.4.2 State The state space of the `doubleWithdrawAtLeast` method is given by `this.getBalance()`.

5.4.3 Precondition Let us compute the precondition of the composed method `doubleWithdrawAtLeast(amount1, amount2)` using the formula given in Sect. 4.5.

```
pre(this.doubleWithdrawAtLeast(amount1, amount2))
= pre(this.withdrawAtLeast(amount1) ;
      this.withdrawAtLeast(amount2))
= pre(this.withdrawAtLeast(amount1)) &&
  ∀ interm.this.getBalance() |
```

```

    ( post(this.withdrawAtLeast(amount1))[post/interm]
      &&
      inv(intermediate.this.getBalance())
      ⇒
      pre(this.withdrawAtLeast(amount2))[old/interm]
    )
  = // fill in
    amount1 >= 0 && amount1 <= old.this.getBalance() &&
    ∀ intermediate.this.getBalance() |
    ( intermediate.this.getBalance()
      <= old.this.getBalance() - amount1
      &&
      intermediate.this.getBalance() >= 0
      ⇒
      amount2 >= 0 && amount2 <= intermediate.this.getBalance()
    )

```

When `amount2` is strictly positive, the universal quantification will evaluate to false, because we can set `intermediate.this.getBalance()` to zero, in which case `amount2 <= intermediate.this.getBalance()` evaluates to false, while the left-hand side of the implication is true.

When `amount2` is strictly negative, the right-hand side of the implication is false. The universal quantification will evaluate to false, because we can set `intermediate.this.getBalance()` to zero, in which case the left-hand side of the implication evaluates to true.

When `amount2` is equal to zero, the universal quantification evaluates to true:

```

  ∀ intermediate.this.getBalance() |
  ( intermediate.this.getBalance()
    <= old.this.getBalance() - amount1
    &&
    intermediate.this.getBalance() >= 0
    ⇒
    0 >= 0 && 0 <= intermediate.this.getBalance()
  )
  = // simplify the right-hand side of the implication
    ∀ intermediate.this.getBalance() |
    ( intermediate.this.getBalance()
      <= old.this.getBalance() - amount1
      &&
      intermediate.this.getBalance() >= 0
      ⇒
      <= intermediate.this.getBalance()
    )
  = // ( ( p && q ) ⇒ q ) == true
    ∀ intermediate.this.getBalance() |
    true
  = true

```

So, the universal quantification over the intermediate state is true, if and only if the second amount is equal to zero. The precondition further simplifies to:

```
pre(this.doubleWithdrawAtLeast(amount1, amount2))
= // copied from above
  amount1 >= 0 && amount1 <= old.this.getBalance() &&
  ∀ interm.this.getBalance() |
  ( interm.this.getBalance()
    <= old.this.getBalance() - amount1
    &&
    interm.this.getBalance() >= 0
    ⇒
    amount2 >= 0 && amount2 <= interm.this.getBalance()
  )
= // use the reasoning given above
  amount1 >= 0 && amount1 <= old.this.getBalance() &&
  amount2 == 0
```

The precondition of the composed method expresses that the precondition of the first composing method should be true (i.e. the first given amount should be non-negative and smaller than the balance of the account) and that the second given amount should be zero. Because the first `withdrawAtLeast` method can make the balance of the account as low as it wants (zero in the most extreme case), the only way to guarantee that the precondition of the second method is satisfied is indeed to require that the second given amount is zero.

5.4.4 Postcondition The postcondition of `doubleWithdrawAtLeast` is given by:

```
post(this.doubleWithdrawAtLeast(amount1, amount2))
= post(this.withdrawAtLeast(amount1) ;
      this.withdrawAtLeast(amount2))
= ∃ interm.this.getBalance() |
  post(this.withdrawAtLeast(amount1))[post/interm] &&
  post(this.withdrawAtLeast(amount2))[old/interm]
= // fill in
  ∃ interm.this.getBalance() |
  interm.this.getBalance()
  <= old.this.getBalance() - amount1 &&
  this.getBalance() <= interm.this.getBalance() - amount2
= // rewrite the second expression
  ∃ interm.this.getBalance() |
  interm.this.getBalance()
  <= old.this.getBalance() - amount1 &&
  this.getBalance() + amount2 <= interm.this.getBalance()
= // see appendix
  this.getBalance() + amount2
```

```

    <= old.this.getBalance() - amount1
= // rewrite the expression
  this.getBalance()
    <= old.this.getBalance() - amount1 - amount2

```

³The `doubleWithdrawAtLeast` method withdraws at least the two given amounts from the account.

5.4.5 Conclusion The pre- and postconditions computed in the preceding sections give rise to the declarative specification shown in Fig. 14.

```

/**
 * Withdraw at least the given amounts from this account.
 *
 * @param amount1, amount2
 *         The amounts to be withdrawn.
 * @pre    The first given amount is not negative.
 *         | amount1 >= 0
 * @pre    The first given amount is smaller than or equal
 *         | to the balance of this account.
 *         | amount1 <= this.getBalance()
 * @pre    The second given amount is equal to zero.
 *         | amount2 == 0
 * @post   The balance of this account is at least
 *         | decremented by the given amounts.
 *         | this.getBalance()
 *         | <= old.this.getBalance() - amount1 - amount2
 */
public void doubleWithdrawAtLeast(final int amount1,
                                  final int amount2) {
    this.withdrawAtLeast(amount1);
    this.withdrawAtLeast(amount2);
}

```

Fig. 16. Traditional declarative specification of the `doubleWithdrawAtLeast` method.

5.4.6 Schema A schematic representation of the `doubleWithdrawAtLeast` method is given in Fig. 17.

³ The fifth step in the above computation is worked out in more detail in Sect. 10.2.

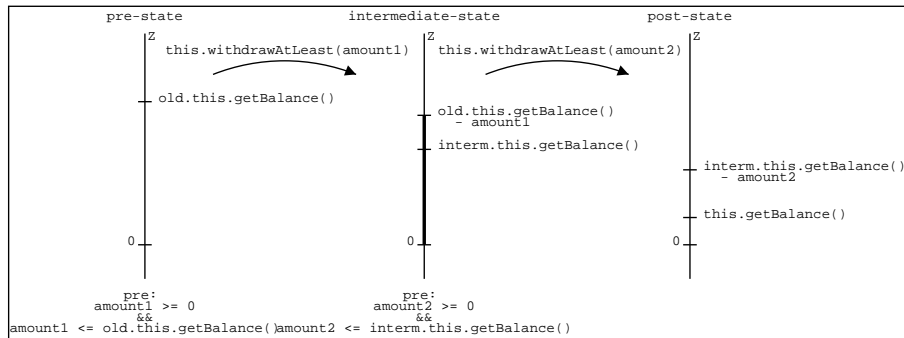


Fig. 17. Schematic representation of the doubleWithdrawAtLeast method.

5.5 Methods That Cannot Be Composed (Deterministic)

In this section and the following section, we give two examples of methods that cannot be composed. This means that some of the intermediate states realized by the first composing method do not satisfy the precondition of the second composing method. In such a situation, the precondition of the composed method evaluates to false.

In our first example, we consider two deterministic methods.

5.5.1 Definition Consider the following commands:

```

/**
 * Set the balance of this account to two.
 *
 * @post The balance of this account is set to two.
 *       | this.getBalance() == 2
 */
public void constant2() {
    $balance = 2;
}

/**
 * Set the balance of this account to one.
 *
 * @pre The balance of this account should be equal
 *      to zero.
 *      | this.getBalance() == 0
 * @post The balance of this account is set to one.
 *      | this.getBalance() == 1
 */
public void constant1() {

```

```

    $balance = 1;
}

```

In the following sections, we will compute the precondition of the composed command `this.constant2();this.constant1()`. It is clear from the specifications given above, that these methods cannot be composed. After application of the first method, the balance of the account is equal to two; as a consequence, the precondition of the second method is never fulfilled after application of the first method.

5.5.2 State The composed method introduced above involves only one object (`this`) so the state is given by `this.getBalance()`.

5.5.3 Precondition We can compute the precondition using the formula for deterministic methods introduced in Sect. 4.6:

```

pre(this.constant2() ; this.constant1())
= pre(this.constant2()) &&
  inv( f(old.State))
⇒
pre(this.constant1())[old.State/f(old.State)]
= pre(this.constant2()) &&
  inv( 2 )
⇒
pre(this.constant1())[old.this.getBalance()/2]
= // fill in
  true &&
  2 >= 0
⇒
  2 == 0
= // ( true && p ) == p
  2 >= 0
⇒
  2 == 0
= // the implication is false
  false

```

The resulting precondition is false, as we expected.

5.5.4 Schema A schematic representation of the composed method

```

this.constant2() ; this.constant1()

```

is given in Fig. 18.

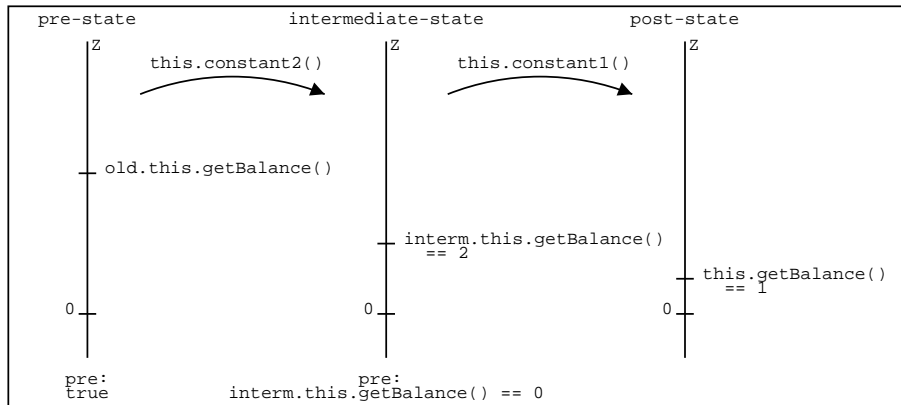


Fig. 18. Schematic representation of the constant2;constant1 method.

5.6 Methods That Cannot Be Composed (Non-Deterministic)

In this section, we examine a non-deterministic method and a deterministic method, that cannot be composed.

5.6.1 Definition Consider the following commands:

```

/**
 * Increase the balance, or set it to zero.
 *
 * @post   The balance is increased, or set to zero.
 *         | this.getBalance() >= old.this.getBalance()
 *         | ||
 *         | this.getBalance() == 0
 */
public void increaseOrReset() {
    ...
}

/**
 * Set the balance of the account to zero.
 *
 * @pre    The balance of this account should be positive.
 *         | this.getBalance() > 0
 * @post   The balance of this account is set to zero.
 *         | this.getBalance() == 0
 */
public void constant0() {
    $balance = 0;
}

```

In the following section, we will compute the precondition of the composed command `this.increaseOrReset();this.constant0()`. Again, we see that these methods cannot be composed: after application of the `increaseOrReset` method, the value of the balance is possibly equal to zero and zero does not satisfy the precondition of the `constant0` method.

5.6.2 State The composed method introduced above involves only one object (`this`) so the state is given by `this.getBalance()`.

5.6.3 Precondition We can compute the precondition using the formula introduced in Sect. 4.5.

```
pre(this.increaseOrReset() ; this.constant0())
= pre(this.increaseOrReset()) &&
  ∀ interm.this.getBalance() |
  ( post(this.increaseOrReset())[post/interm]
    &&
    inv(interm.this.getBalance())
    ⇒
    pre(this.constant0())[old/interm]
  )
= // fill in
  true &&
  ∀ interm.this.getBalance() |
  ( ( interm.this.getBalance() >= old.this.getBalance()
      ||
      interm.this.getBalance() == 0
    )
    &&
    interm.this.getBalance() >= 0
    ⇒
    interm.this.getBalance() > 0
  )
= // the universal quantification evaluates to false:
  // when interm.this.getBalance() is equal to zero
  // the left-hand side of the implication is true
  // while the right-hand side is false
  false
```

The precondition of the composed method is false, as we expected.

5.6.4 Schema A schematic representation of the composed method

```
this.increaseOrReset() ; this.constant0()
```

is given in Fig. 19.

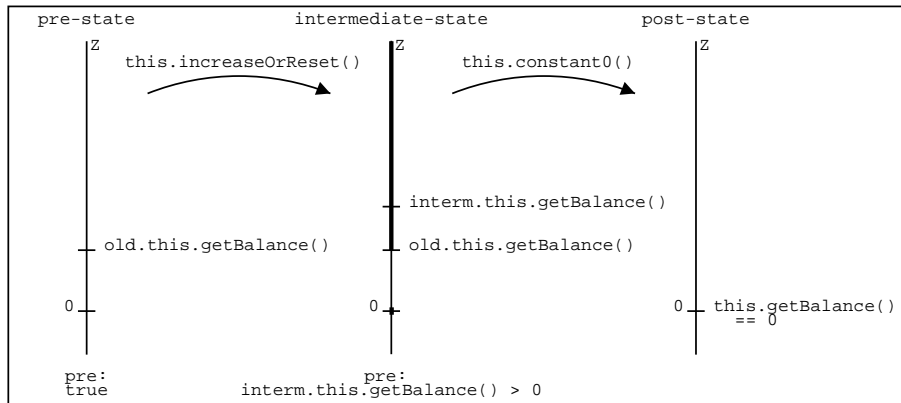


Fig. 19. Schematic representation of the increaseOrReset;constant0 method.

6 Advantages

In this section, we describe the main advantages of using sequential behaviour composition.

Specifying a composed command using sequential behaviour composition usually gives rise to more succinct specifications. (Compare the traditional specification of the transferTo method in Fig. 2 with the specification in Fig. 3 that uses sequential behaviour composition.)

None of the specification languages used in the context of DBC (see Sect. 7) allows us to specify the effect of a composed command in terms of the composing commands. Introducing sequential behaviour composition increases the expressiveness of our specification language, allowing us to explicitly express the connection between a composed method and its composing methods.

In some situations, it is possible to work around this lack of expressiveness as we illustrated in Sect. 3.1. We duplicated the pre- and postconditions of the composing methods to specify the composed method. The disadvantages of this workaround are described in Sect. 3.2. In many situations, this duplication is impossible, because the composing methods are specified only non-deterministically, and concrete specifications are given in subclasses.

Sequential behaviour composition allows us to reflect the semantics of a composed method without duplicating the pre- and postconditions of the composing methods. In this way, Parnas' principle [27] of information hiding, saying that each fact must be worked out in one, and only one, place, is supported better. As a consequence, possible consistency problems are avoided. These consistency problems can arise when duplicating the specification of the composing methods to write the specification of the composed method or they can arise when the specification of the composing methods changes. An example of such a change

for the `transferTo` method is when the bank decides to charge a fee for each withdrawal. In that case, the expanded specification of the `transferTo` method will automatically change accordingly and express that a fee is charged of the source account. Remark that, at the implementation level, the semantics of the `transferTo` method is reflected by invoking the `withdraw` and `deposit` method in the method body. Manipulating the balances of the two accounts directly in the implementation of the `transferTo` method would be considered bad object-oriented programming.

A sequential behaviour composition is a declarative specification, although it seems to be operational at first sight. Each sequential behaviour composition can be expanded into pre- and postconditions containing only basic queries.

DBC is one of the building blocks of MDA [8,20]. It is used to specify in a formal way both the structural aspects and the behavioural aspects of models. One of the ambitions of MDA is to automate the transformations between models as much as possible, in particular the transformation from PSM to code. For relatively simple operations, it is already possible to generate a prototype-implementation from the postcondition. For more complex operations, automatic generation of code from specifications is not possible (yet). Sequential behaviour composition increases the set of methods for which code can be generated: it is trivial to generate a prototype-implementation for a method that is specified in terms of a sequential behaviour composition of other methods.

7 Related Work

The ideas underlying DBC go back to the work of pioneers in computer science such as Dijkstra [4] and Hoare [11]. They already suggested documenting routines in terms of preconditions and postconditions. DBC, as we know it today, was developed by Bertrand Meyer as a part of Eiffel [24,6]. Eiffel is an object-oriented programming language that has built-in support for contracts. Another object-oriented language incorporating support for DBC is Sather [29]. Most object-oriented languages, including Java, C++ and C#, lack support for DBC though. Nevertheless, there is a growing interest in DBC and several tools have been developed that provide support for DBC in Java: Biscotti [3], Contract Java [7], Handshake [5], iContract [16,17], Jass [12,1], Jcontract [13,14], jContractor [15,19]. The Java Modeling Language (JML) [21,22] is a behavioural interface specification language that can be used to specify the behaviour of Java modules and that is supported by a wide range of tools. The Object Constraint Language (OCL) [31,32] is a formal language that can be used to specify invariants and operations for UML (Unified Modeling Language) models in a formal way.

One of the principles of DBC is that contracts for software components must be written in a declarative way, using a formal, mathematically founded notation. All the systems described above use Boolean assertions to write pre- and postconditions. JML follows a model-based specification approach, while Eiffel, OCL

and the tools listed above write specifications in terms of variables and queries directly applicable to the documented software component.

Yet another kind of specification language is Z [28]. Routines in Z are described using schemas. A schema consists of two parts: a declaration part and a predicate part. The declaration part introduces the input and output objects of the routine and the before and after components of the global state. The predicate part expresses in a declarative way the relationships between input objects, output objects, and components of the global state; it describes the semantics of the routine. One of the differences between Z and specification languages used in the context of DBC is that Z offers support for schema composition. This allows the effect of a composed routine to be specified in terms of other routines. In object-oriented programming languages, such a sequential behaviour composition mechanism is available at the level of the implementation, where methods can invoke other methods in their body. In specification languages supporting DBC, behaviour composition is limited to queries.

A last important element in our paper is Model Driven Architecture (MDA) [8,20]. MDA is a framework for software development defined by the Object Management Group (OMG) [26]. An important goal of MDA is to raise the level of abstraction at which a developer works by shifting the developer's attention from platform-specific implementation details to the development of a Platform Independent Model (PIM). By definition, a PIM is independent of any specific implementation technology. Once a PIM has been developed, it is transformed into one or more Platform Specific Models (PSM). PSMs contain technical details concerning how to implement a given PIM on a specific platform. In a final step, the PSM is transformed into code. One of the ambitions of MDA is to automate the successive transformations as much as possible. In this way, developers are able to focus on the development of PIMs, thereby raising the abstraction level. As a consequence, MDA improves productivity. PIMs are platform-independent and therefore completely portable, they simplify interoperability between platforms and provide a consistent high-level documentation for a software project.

DBC is one of the building blocks of MDA. It can be used to make structural models consistent and complete and to fully specify the result of queries. Moreover, the dynamics of the system can be expressed by pre- and postconditions imposed on operations. For relatively simple operations, the body of the corresponding operation can be generated from the postcondition, but in the current state of the art, automatic generation of code from specifications is still out of reach.

8 Future Work

In this paper, we introduced the basic ideas of sequential behaviour composition. Many opportunities for further research still exist.

In this paper, we only considered method contracts containing pre- and postconditions. We did not take exceptions into account. Many specification languages and tools supporting DBC, including the behavioural interface specification language JML [21,22], the run-time monitoring tools Jcontract [13,14] and jContractor [15,19], and the IPS [30] methodology, offer support for the specification of exceptions. The conditions under which exceptions can be thrown can be specified formally and rules are formulated about how these contracts are inherited and overridden in subclasses. Research is needed to determine the exceptions that can be thrown by a composed method and the conditions under which this can happen, given the conditions under which exceptions can be thrown by the composing methods.

Inheritance is another interesting challenge in the research about sequential behaviour composition. First, it should be examined how the expansion of a sequential behaviour composition changes when we go down an inheritance hierarchy. Second, we should analyse whether it is possible to override methods that are specified using a sequential behaviour composition. The Liskov substitution principle states that in class hierarchies, it should be possible to treat a specialized object as if it were an instance of its superclass. In terms of pre- and postconditions, this means that preconditions can only be weakened, while postconditions can only be strengthened. The rules that the Liskov substitution principle implies for methods that are specified using sequential behaviour composition are still to be determined.

Another question related to inheritance is how we should interpret the sequential behaviour composition examined in Sect. 3 when the class `Account` is at the top of a hierarchy of classes involving several kinds of accounts. Probably, we will use the dynamic type of the destination account when expanding the sequential behaviour composition. This illustrates another advantage of sequential behaviour composition, namely that it allows specifications that depend on the dynamic type of the participating objects.

The technique explained in this paper can not only be used for commands but also for the specification of constructors. The implications of this idea should be examined by means of further experiments.

Beside sequential behaviour composition, similar constructs for the specification of commands are worth examining. Possible examples are (1) an and-operator and (2) iteration. The *and-operator* can be used to express that two commands are executed in parallel. This means that the pre- and postconditions of the composing methods are simply conjoined. For independent operations, using the and-operator is equivalent to applying sequential behaviour composition. The order in which such operations are executed is arbitrary. The and-operator is not applicable for operations that have contradicting postconditions; this would lead to a method that is not implementable. *Iteration* could be used to express that the same method is applied to all objects in a given set. The syntax of such a specification could be something like

```
for each account in getAccounts():
    account.m()
```

The meaning of this specification is that method `m` should be applied to all given accounts, where the specifications are connected using the and-operator. Alternatively, iteration could also be combined with sequential behaviour composition.

9 Conclusion

The basic building blocks of DBC, i.e. preconditions, postconditions and invariants, are traditionally expressed by means of Boolean expressions. We started this paper by identifying and illustrating a lack of expressiveness in the Design by Contract methodology when specifying composed commands, i.e. of commands whose effect is realized using other commands. This lack of expressiveness causes several problems, including specification duplication, possible inconsistencies and formidable specifications.

In a next step, we presented an alternative way to specify software components, using a mechanism that we call sequential behaviour composition. Sequential behaviour composition is an extension of DBC based on the idea of schema composition in the Z specification language. It enables us to express the specification of a composed method in terms of the specification of the composing methods. The semantics of the new methodology is described formally by expressing the pre- and postconditions of a sequential behaviour composition in terms of the pre- and postconditions of the composing methods. The semantics is illustrated by means of examples, using the Java programming language.

Among the main advantages of sequential behaviour composition, we distinguish increased expressiveness of contracts, better consistency, improved adaptability and trivial code generation.

10 Appendix: Some Proofs

10.1 Equivalence 1

In this section, we proof the following equivalence, which is used in Sect. 5.3.3, when deriving the precondition of the `nonDet` method.

```
∀ interm.this.getBalance() |
( interm.this.getBalance() > 2 * old.this.getBalance()
  &&
  interm.this.getBalance() >= 0
```

```

    =>
    amount >= 0 && amount <= interm.this.getBalance()
  )
  ⇔
  ( amount >= 0
    &&
    2 * old.this.getBalance() + 1 >= amount
  )
  ||
  ( 2 * old.this.getBalance() + 1 < 0
    &&
    amount == 0
  )
)

```

We can prove that the two Boolean expressions are equivalent by proving the two implications. First we prove that

```

∀ interm.this.getBalance() |
( interm.this.getBalance() > 2 * old.this.getBalance()
  &&
  interm.this.getBalance() >= 0
  =>
  amount >= 0 && amount <= interm.this.getBalance()
)
=>
( amount >= 0
  &&
  2 * old.this.getBalance() + 1 >= amount
)
||
( 2 * old.this.getBalance() + 1 < 0
  &&
  amount == 0
)
)

```

First assume that $2 * \text{old.this.getBalance}() + 1$ is not negative. Set $\text{interm.this.getBalance}()$ to $2 * \text{old.this.getBalance}() + 1$. Then the left-hand side of the implication inside the universal quantification is true. As a consequence, the right-hand side of that implication is true, so $\text{amount} \geq 0 \ \&\& \ \text{amount} \leq \text{interm.this.getBalance}()$, or $\text{amount} \geq 0 \ \&\& \ \text{amount} \leq 2 * \text{old.this.getBalance}() + 1$. This proves the left-hand side of the disjunction. Next assume that $2 * \text{old.this.getBalance}() + 1$ is negative. Set $\text{interm.this.getBalance}()$ to zero. Then

```

interm.this.getBalance() == 0
> 2 * old.this.getBalance() + 1
> 2 * old.this.getBalance(),

```

so the left-hand side of the implication inside the universal quantification is true. The right-hand side then learns us that

```
amount >= 0 && amount <= interm.this.getBalance()
```

so `amount >= 0 && amount <= 0`. In other words `amount == 0`. This proves the right-hand side of the disjunction.

Next, we prove the other implication:

```
∀ interm.this.getBalance() |
( interm.this.getBalance() > 2 * old.this.getBalance()
  &&
  interm.this.getBalance() >= 0
  ⇒
  amount >= 0 && amount <= interm.this.getBalance()
)
⇐
( amount >= 0
  &&
  2 * old.this.getBalance() + 1 >= amount
)
||
( 2 * old.this.getBalance() + 1 < 0
  &&
  amount == 0
)
```

First assume that

```
amount >= 0 and
2 * old.this.getBalance() + 1 >= amount
```

Take an arbitrary value `interm.this.getBalance()` such that

```
interm.this.getBalance() > 2 * old.this.getBalance() and
interm.this.getBalance() >= 0
```

Then

```
interm.this.getBalance() >= 2 * old.this.getBalance() + 1
>= amount
```

Next assume that `2 * old.this.getBalance() + 1 < 0` and `amount == 0`. The right-hand side of the implication inside the universal quantification can then be rewritten as `0 >= 0 && 0 <= interm.this.getBalance()`. The implication is then of the form $(p \ \&\& \ q) \Rightarrow q$, which is always true.

Remark. A possibly easier way to understand the above equivalence, is to rewrite the implication in the following way:

```

∀ interm.this.getBalance() |
( interm.this.getBalance() > 2 * old.this.getBalance()
  &&
  interm.this.getBalance() >= 0
  ⇒
  amount >= 0 && amount <= interm.this.getBalance()
)
= // rewrite the left-hand side of the implication
  ∀ interm.this.getBalance() |
  ( interm.this.getBalance()
    >= max(2 * old.this.getBalance() + 1, 0)
    ⇒
    amount >= 0 && amount <= interm.this.getBalance()
  )
= // saying that all values bigger than or equal to p
  // should be bigger than or equal to amount, is
  // equivalent to saying that p itself should be bigger
  // than or equal to amount
  amount >= 0
  &&
  amount <= max(2 * old.this.getBalance() + 1, 0)
= // distinguish different cases
  amount >= 0
  &&
  ( ( 2 * old.this.getBalance() + 1 >= 0
    &&
    amount <= max(2 * old.this.getBalance() + 1, 0)
    == 2 * old.this.getBalance() + 1
  )
  ||
  ( 2 * old.this.getBalance() + 1 < 0
    &&
    amount <= max(2 * old.this.getBalance() + 1, 0)
    == 0
  )
)
= // distributivity
  ( ( 2 * old.this.getBalance() + 1 >= 0
    &&
    amount <= 2 * old.this.getBalance() + 1
    &&
    amount >= 0
  )
  ||
  ( 2 * old.this.getBalance() + 1 < 0
    &&

```

```

        amount <= 0
        &&
        amount >= 0
    )
)
= // simplify
( ( amount <= 2 * old.this.getBalance() + 1
  &&
  amount >= 0
)
||
( 2 * old.this.getBalance() + 1 < 0
  &&
  amount == 0
)
)
)

```

10.2 Equivalence 2

In this section, we prove the following equivalence, which is used in Sect. 5.4.4, when deriving the postcondition of the `doubleWithdrawAtLeast` method.

$$\begin{aligned} & \exists z \mid x \leq z \ \&\& \ z \leq y \\ \Leftrightarrow & \\ & x \leq y \end{aligned}$$

First, we prove the following implication:

$$\begin{aligned} & \exists z \mid x \leq z \ \&\& \ z \leq y \\ \Rightarrow & \\ & x \leq y \end{aligned}$$

This follows directly from the transitivity law.

Next, we prove the opposite implication:

$$\begin{aligned} & \exists z \mid x \leq z \ \&\& \ z \leq y \\ \Leftarrow & \\ & x \leq y \end{aligned}$$

When $x \leq y$, we can make the expression $x \leq z \ \&\& \ z \leq y$ true by setting z equal to x . So, the existential quantification is true.

References

1. Detlef Bartetzko, Clemens Fischer, Michael Möller, Heike Wehrheim. Jass - Java with Assertions. <http://csd.Informatik.Uni-Oldenburg.DE/pub/Papers/jass.pdf>
2. Blue home page. <http://www.mip.sdu.dk/~mik/blue/>.
3. Cynthia Della Torre Cicalese, Shmuel Rotenstreich. Behavioral Specification of Distributed Software Component Interfaces. IEEE Computer, July 1999, Vol. 32, No 7, p46-53.
4. Edsger Wybe Dijkstra. A Discipline of Programming. Prentice Hall, 1976, ISBN 013215871X.
5. Andrew Duncan, Urs Hölzle. Adding Contracts to Java with Handshake. <http://www.cs.ucsb.edu/labs/oocsb/papers/TRCS98-32.pdf>.
6. An Eiffel Tutorial. http://docs.eiffel.com/general/guided_tour/language/tutorial-00.html.
7. Robert Bruce Findler, Matthias Felleisen. Contract Soundness for Object-Oriented Languages. <http://www.ccs.neu.edu/scheme/pubs/oops1a01-ff.pdf>.
8. David S. Frankel. Model Driven Architecture. Applying MDA to Enterprise Computing. Wiley Publishing, 2003, ISBN 0-471-31920-1.
9. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995, ISBN 0-201-63361-2.
10. James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java Language Specification, Second Edition, 2000. <http://java.sun.com/docs/books/jls/second-edition/html/j.title.doc.html>.
11. C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, vol. 12, no. 10, pages 576-583, October 1969.
12. The Jass Page. Home Page. <http://semantik.informatik.uni-oldenburg.de/~jass/>.
13. Using Design by Contract[™] to Automate Java[™] Software and Component Testing. <http://www.parasoft.com/jsp/products/article.jsp?articleId=579&product=Jcontract>.
14. Jcontract User's Manual. <http://www.parasoft.com/jsp/products/manuals.jsp?product=Jcontract&manual=jtract/manuals/>.
15. Murat Karaorman, Parker Abercrombie. jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation. http://jcontractor.sourceforge.net/doc/jContractor_FMSD03.pdf.
16. Reto Kramer. iContract - The Java Design by Contract Tool. <http://www.reliable-systems.com/tools/iContract/documentation/iContract-tools98usa.pdf>.
17. Reto Kramer. Examples of Design by Contract in Java using iContract, the Design by Contract Tool for Java. Object World Berlin '99, Design & Components, Berlin, May 17th-20th 1999. http://www.reliable-systems.com/tools/iContract/documentation/OW_BERLIN99_WEB.PDF.
18. Javadoc Tool Home Page. <http://java.sun.com/j2se/javadoc/>.
19. Murat Karaorman, Urs Hölzle, John Bruno. jContractor: A Reflective Java Library to Support Design By Contract. <http://www.cs.ucsb.edu/labs/oocsb/papers/TRCS98-31.pdf>.
20. Anneke Kleppe, Jos Warmer, Wim Bast. MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley, 2003, ISBN 0-321-19442-X.

21. Gary T. Leavens, Albert L. Baker, Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Department of Computer Science, Iowa State University, TR #98-06x, June 1998, revised July, November 1998, January, April, June, July, August, December 1999, February, May, July, December 2000, February, April, May, August 2001, June, August, October, December 2002, April, May, September, November 2003. <ftp://ftp.cs.iastate.edu/pub/leavens/JML/prelimdesign.pdf>.
22. Gary T. Leavens, Yoonsik Cheon. Design by Contract with JML. 2003. <ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf>.
23. Barbara Liskov, Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811-1841, November 1994.
24. Bertrand Meyer. *Object-Oriented Software Construction*, Second Edition. Prentice-Hall Inc, 1997, ISBN 0-13-629155-4.
25. Richard Mitchell, Jim McKim. *Design by Contract, by Example*. Addison-Wesley, 2002, ISBN 0-201-63460-0.
26. OMG Home Page. <http://www.omg.org/>.
27. D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053 - 1058.
28. Ben Potter, Jane Sinclair, David Till. *An Introduction to Formal Specification and Z*. Prentice Hall, 1991, ISBN 0-13-478702-1.
29. Sather home page. <http://www.icsi.berkeley.edu/~sather/>.
30. Eric Steegmans, Jan Dockx. *Objectgericht programmeren met Java*. Acco, 2002, ISBN 90-334-4535-2.
31. Jos B. Warmer, Anneke G. Kleppe. *The Object Constraint Language, Precise Modeling with UML*. Addison-Wesley, 1999, ISBN 0-201-37940-6.
32. Jos Warmer, Anneke Kleppe. *The Object Constraint Language, Second Edition. Getting Your Models Ready for MDA*. Addison-Wesley, 2003, ISBN 0-321-17936-6.