Specifying the boundary between unverified and verified code

David R. Cok[0] and K. Rustan M. Leino[1]

[0] Safer Software, LLC
david.r.cok@gmail.com
[1] Amazon Web Services
leino@amazon.com

Abstract. This paper introduces a specification construct that is fitting when combining verified code with unverified code. The specification is a form of precondition that imposes proof obligations for both callers and callees. This precondition, `recommends`, blends in well with the parameter-validation conventions in Java and with the syntax and semantics of specification languages such as JML, ACSL, and Dafny.

0. Introduction

In a Utopian world, every line of software we run would be formally verified. That's not the world we live in. Even with the increasing industrial demand for and use of verified software, there will always be components of our software systems that are not verified. This may be an intentional business decision, just like a manufacturing company today makes a decision about the strength of plastic to use in a product, or it may reflect a transient situation where unverified software components are gradually being replaced by verified components.

Whatever the situation, there are issues to consider on the boundary between unverified and verified software. Indeed, by hardening this boundary, we may also be able to ease the transition to software with more verified components.

In this paper, we focus on the interface to a verified component. In particular, we want to address the question of how to allow calls into a verified component.

As usual in specification and verification, a method has a *precondition* that must hold at the time of a call. For example (the code examples are written in a blend of Dafny [11, 12], Java [8], and the Java Modeling Language (JML) [5], but are intended to be language-agnostic), the method

```
method Example(x: int)
  requires 0 <= x < 10
```

declares as a precondition that parameter `x` must be in the range `0` to `10`. It is the *caller's* responsibility to establish the precondition (see e.g., [15, 0, 9]). In the situation we're focusing on in this paper, the code of the caller may not be verified, which means that the callee really has no guarantee that the precondition will hold on entry. Instead, we propose an alternative precondition, which we write as

```
method Example(x: int)
  recommends 0 <= x < 10
```

This form of precondition still documents the *intention* that the given condition hold on entry to the method. The difference from the standard `requires` precondition is that `recommends` shifts some responsibility from the caller to the callee. It is important to allow the callee to meet this responsibility using the natural patterns of programming that are used today, for example in Java. It is also important to let the construct pave the way to more verified code. Our proposal addresses both of these points.

In the next three sections, we review some points about parameter validation, method specifications, and the strictness of a static verifier. In §4, we give the precise details of the basic `recommends` construct, and in the subsequent three sections we extend the basic construct to some variations. In §8, we report on a prototype of `recommends` implemented in OpenJML for Java/JML and on possible extensions that incorporate `recommends` into Dafny.

1. Parameter validation

Embracing the *fail-fast* philosophy, a common programming pattern in Java and other languages is for a method to validate its parameters on entry. This validation typically consists of checking some simple property that is expected to hold of a parameter and throwing an exception if the property does not hold.

For example, a method that expects a non-`null` reference to a `Cell` and an integer in the range `0` to `10` is written

```
method FailFastExample(c: Cell, x: int) {
  if c == null {
    throw ArgumentNullException();
  }
  if ¬(0 <= x < 10) {
    throw BadParameterException();
  }
  // ...
}
```

On entry, this method immediately rejects unsupported parameter values.

Any exception thrown by a parameter-validation check indicates a program error, a failure on behalf of the caller to use the method correctly. They are what Goodenough calls *client failures* [7]. Even in languages that allow such exceptions to be caught, there's usually not much of a recourse, other than letting a backstop clean up the program state before exiting the program gracefully, or in some cases obtaining new input to the method and trying again.

The common documentation style in Java and .NET is to be obsessively specific about which exception is thrown in response to which unsupported parameter values. While looking at the information associated with such an exception is useful when debugging a program crash, we maintain that the particular exception thrown is not so important to the program itself. Indeed, it is not a good idea for a program to read

too much into the parameter-validation exceptions. For example, consider a caller that tries to use the callee's parameter validation to check the value of the parameters being passed in:

```
try {
  FailFastExample(c, 0);
} catch ArgumentNullException {
  // silly me, I must have called FailFastExample with c == null
  // ...
}
```

If `c` is `null` in this call to `FailFastExample`, then the exception handler will be invoked. But it's a mistaken view to think that this is the only situation under which the exception handler is invoked. It may be that the implementation of `FailFastExample` is buggy and passes in `null` to another method it calls, which would also result in the exception handler above being invoked. So, it is safer to consider all parameter-validation exceptions as catastrophic errors rather than trying to handle them in overly specific ways.

The explicit `if` statements in the implementation of `FailFastExample` above are not the only way to perform parameter validation. Let us describe two alternatives that are common in practice.

One alternative is for the code to perform an operation that is sure to result in an exception when a parameter has an unsupported value. For example, in a language like Java, which throws a `NullPointerException` if a `null` pointer is dereferenced at run time, the example method above can be written

```
method FailFastExample(c: Cell, x: int) {
  var y := c.data; // this throws a NullPointerException if c is null
  if ¬(0 <= x < 10) {
    throw BadParameterException();
  }
  // ...
}
```

By writing the code in this way, one saves the explicit check. But since the new check is implicit in the language semantics, it is easier to forget to include it at the right time in the method. Without a code comment like the one above, it may also be difficult for a human to determine which pointer dereferences are in effect parameter validations and which ones are expected to work. Finally, the exception thrown by the language semantics may be slightly different from one chosen in an explicit check (as the examples above show), but, as we argued, the particular exception thrown is not so important.

The second alternative to an explicit parameter-validation check is to delegate the check to another method. For example, if the `FailFastExample` method calls a companion method that is going to validate the parameters, then `FailFastExample` can leave the check for the companion method.

```
method FailFastExample(c: Cell, x: int) {
  // the following method starts by validating the parameters
  // c and x, so we delegate the checking to it
```

```
    ASimilarFailFastExample(c, x, 100, true);
    // ...
  }
```

To stick with the fail-fast philosophy, these alternatives are advisable only if they can be placed near the entry to the method, in order to avoid any state changes before the parameters have been validated. For example, if the first dereference of `c` does not happen until later in the method when the method has already started its intended job or if the call to the companion method doesn't always take place, then parameter validation must be done explicitly.

2. Method specifications

In this paper, we write method specifications following the method signature. We have already seen a glimpse of the two kinds of preconditions we'll consider, `requires` and `recommends`. These say what is expected to hold on entry to the method. In addition, a method specification can relate the method's post-state to its pre-state, which is done using a combination of three *post-specification* clauses. These have the form

```
modifies Frame
ensures Post
throws Exc ensures EPost
```

A `modifies` clause describes which parts of the program state are allowed to be changed by the method. The details of how `Frame` is specified are not important to this paper. The only thing we need to know is that the absence of a `modifies` clause means the method is not allowed any visible change to the program state.[0]

An `ensures` clause describes what is expected to hold on *normal* exit from the method. In other words, if the method terminates normally (that is, terminates without throwing an exception), then `Post` relates the method's pre- and post-states. Since `Post` is a two-state predicate, we use the expression `old(E)` to denote the expression `E` evaluated in the method's pre-state.

A `throws Exc ensures` clause describes what is expected to hold on exit from the method, in the event that the method exits by throwing an `Exc` exception. The specification can include several `throws ensures` clauses, each possibly mentioning a different exception. For the purposes of this paper, we ignore Java's `throws` clauses. That is, we allow methods to throw any exception, as long as all `throws ensures` clauses are satisfied.

We also allow a clause

```
throws * ensures EPost
```

which describes what is expected to hold on exit in the event that the method exits by throwing an exception not covered by any other `throws ensures` clause in the method post-specifications.

[0] This coincides with the interpretation of `modifies` clauses in Dafny. In JML, frames are defined using `assignable` clauses, and an absent `assignable` clause means `assignable \everything`. For our discussion here, `assignable \nothing` is more convenient as a default.

For example,

```
method PostSpecifications(c: Cell, x: int)
  modifies c
  ensures c.data == old(c.data) + 1
  throws ArithmeticException ensures x == 0 && c.data == old(c.data)
```

declares that method `PostSpecifications` does not have an effect on any object except possibly `c` and that upon normal termination, `c.data` is `1` more than it had been in the method's pre-state. The specification also says that if the method happens to terminate with an `ArithmeticException`, then the parameter `x` was `0` (in other words, this specification clause says that `x == 0` is the only circumstance in which the method is allowed to throw an `ArithmeticException`) and the value of `c.data` is unchanged.

In this paper, it will be convenient to group post-specification clauses according to what holds in the pre-state. For a pre-state condition `Guard` and a list of post-specification clauses `PS`, we will write

```
when Guard
  PS
```

to express that post-specifications `PS` apply only if the condition `Guard` holds in the pre-state. If `Guard` does not hold in the pre-state, then `PS` is ignored.[1] The `*` in any `throws * ensures` clause in `PS` denotes those exceptions not covered by other `throws ensures` clauses in `PS`.

3. Language discipline versus verifier discipline

When designing a static verifier for an existing programming language, it is common (and desirable) to enforce a stricter discipline than the language requires. As an example, the C and Java languages implement modulo semantics for operations on fixed-bit-width integers; for instance, addition of 32-bit `int`s silently overflows to negative values. However, as overflows frequently are unintended and mask bugs, the verification discipline might be stricter, at least by default, warning the implementer by reporting potential over- and underflows in fixed-bit-width operations as errors.

In a language like Java, a choice about what discipline to enforce also arises in those places where the Java language definition prescribes run-time exceptions to be thrown. For example, the effect of an assignment `y := c.data;` at run time is really

```
if c == null {
  throw NullPointerException();
}
y := c.data;
```

[1] JML uses an operator called `also` to combine specifications. It gives a nice way to express what we write with `when` in this paper. Nevertheless, we chose not to use the `also` operator in this paper, because `also` in essence uses `requires` clauses for the guards. By using the `when` notation in this paper, we avoid any confusion regarding `requires` and `recommends` in this context.

A static verifier that uses this loose language discipline of pointer dereference would have to deal with all of the control paths that the `throw` operation gives rise to. A respectable alternative is to instead enforce the stricter discipline of banning `null` from ever being dereferenced. So, if a program possibly dereferences `null`, then the verifier reports an error rather than analyzing any exceptional control paths. (You may think of this stricter discipline as the verifier's fail-fast alternative to the loose language discipline.)

For flexibility, a static verifier can let the user choose between the language discipline and the stricter discipline. For this purpose, the KeY tool [0] supports both a `ban` mode and an `allow` mode. In this paper, we will use the stricter discipline unless explicitly directed by the user to use the language discipline. The directive we propose is to write `//@ allow` at the end of a line, which will cause the verifier to use the language discipline for any construct on that source line that has a choice between the two.

For example, consider the following program snippet in a context where `z` may be `0` and `c` may be `null`:

```
try {
  x := 100 / z;
} catch ArithmeticException {
  y := c.data;
}
```

Under the stricter discipline, the verifier will detect and report the potential division by zero. No `ArithmeticException` is thrown, so the `catch` block and its dereference of `c` are not reachable. In contrast, by including `//@ allow`, the program snippet

```
try {
  x := 100 / z; //@ allow
} catch ArithmeticException {
  y := c.data;
}
```

directs the verifier to use the language discipline for the statement that assigns to `x`. Under this discipline, the attempt to evaluate `100 / z` when `z` is zero turns into a jump to the exception handler. No error is reported for the attempt to divide by zero, but the verifier detects and reports the potential `null`-dereference error in the `catch` block.

In its form above, `allow` uses the language discipline for all constructs where a choice exists on the given line. We also permit `allow` to take a list of exception names, with the effect of using the language discipline on that line only for the indicated exceptions.

We'll come back to `allow` directives in §5, where we show how the new `recommends` clauses benefit from the directives.

4. Recommends clauses

With those introductory sections out of the way, we now define `recommends` clauses.

Just like the standard precondition declaration `requires P`, the declaration `recommends P` says that `P` is a precondition for the method. That is, the intention is that `P` hold

on entry to the method. For this reason, the `recommends` condition is checked by the verifier at call sites. Of course, this only refers to the call sites that the verifier is aware of, so in the presence of unverified code, it can still happen that the method is called from a state where the precondition does not hold. But whenever the verifier encounters a call to a method with a `recommends` clause, it will check that the given condition holds at the call site.

For method *implementations*, there is a big difference between `requires` and `recommends`. To guard a method implementation from breaches of the precondition at call sites where the verifier has not been applied, one could imagine automatically compiling the `recommends` clause into a run-time check. For example, a method declaration

```
method M(u: U)
  recommends P(u)
{
  Body
}
```

could be compiled in the same way as

```
method M(u: U) {
  if ¬P(u) {
    throw PreconditionFailure();
  }
  Body
}
```

Although simple, this strawman solution falls short in at least a couple of ways.

One shortcoming is that this strawman can give rise to unnecessarily repeated run-time checks. For example, if (like we saw in §1) `Body` calls another method that also performs parameter validation, then some parameter-validation checks would be performed once in `M` and once in the other method.

A second shortcoming of this strawman is that the condition `P(u)` may not be compilable. Specifications (like those in JML and in Dafny) can contain specification-only features or ghost variables that are not available at run time. In these cases, it would be nice to allow the more abstract condition `P(u)` in the `recommends` clause as part of the method's specification, and to allow the implementation of the method to prescribe an alternate way to check `P(u)` in terms of compilable constructs.

For these reasons, we abandon the idea of automatically compiling `recommends` clauses. Instead, we let the programmer write the code for testing the `recommends` conditions manually, throwing parameter-validation exceptions if the conditions do not hold, and generate verification conditions that check the correctness of that manually authored code. That is, suppose method `M` is declared by

```
method M(u: U)
  recommends Pre
  modifies Frame
  ensures Post
  throws Exc ensures EPost
```

From the point of view of verifying the body of `M`, it is as if its specification had been written

```
method M(u: U)
  when Pre
    modifies Frame
    ensures Post
    throws Exc ensures EPost
  when ¬Pre
    ensures false
    throws PreconditionFailure ensures true
    throws * ensures false
```

This specification says that, if `Pre` holds on entry to the method, then the method is bound by the given post-specification. If `Pre` does not hold on entry to the method, then the method body must arrange to throw a `PreconditionFailure` exception—it is not allowed to terminate any other way. Furthermore, if `Pre` does not hold on entry, then (since the second post-specification group does not have a `modifies` clause) the method is not allowed to modify anything.

Note that in this rewritten specification—which, remember, is from the point of view of the method implementation—there is no precondition. In particular, the method body is *not* allowed to assume the condition `Pre` on entry.

Here is an example with a method body:

```
method Increment(c: Cell, x: int)
  recommends c != null
  recommends 0 <= x < 10
  modifies c
  ensures c.data == old(c.data) + x
{
  if c == null || x < 0 || 10 <= x {
    throw PreconditionFailure();
  }
  c.data := c.data + x;
}
```

This method implementation meets its specification. The method body itself checks that the conditions stated in the `recommends` clauses hold. If they do not hold, the method modifies nothing and throws a `PreconditionFailure` exception. If the conditions do hold, the method body increments `c.data` as described by the post-specification clauses.

Two more points are worthy of attention in this example.

One point is to remember that the `recommends` conditions are not assumed on entry to the method body. If they were assumed, the "then" branch of the `if` statement would be unreachable code, and then the verifier would not detect if the method failed to throw the desired exception or if the "then" branch contained any other errors. That would be most regrettable, because the control flow through such a "then" branch is less likely to be thoroughly exercised by testing, so it is especially important that the verifier scrutinize such code.

The other point has to do with the well-definedness of conditions. To be meaningful, the conditions mentioned in specifications must themselves be free of errors. For example, what would a specification `c.data == 100 / x` mean if `x` could be `0` or `c` `null`? Although `recommends` clauses are not assumed on entry to the method's body, they can be assumed when checking the well-definedness of post-specifications. This is achieved by our transformation of `recommends` clauses into post-specifications grouped by `when` guards. For the `Increment` method above, this means that `c != null` can be assumed when checking the well-definedness of the postcondition `c.data == old(c.data) + x`, which avoids any `null`-dereference errors in this postcondition.

5. Using recommends clauses with standard patterns

Let us now explain how to use `allow` directives to achieve the programming patterns mentioned in §1. As we saw in that section, fail-fast parameter validation can be achieved by performing an operation that the language defines as triggering a run-time exception. For example, instead of testing the condition `c == null`, a program can attempt to read a field of `c`. In Java, that causes a `NullPointerException` to be thrown if `c == null` holds. As we discussed in §3, the verifier ordinarily reports an error if the program attempts to dereference a pointer that may be `null`. But by using an `allow` directive on a source line that performs such a dereference, the verifier will instead treat that operation as possibly throwing an exception. This is just what we need to encode this fail-fast pattern.

For example, the `Increment` method from the previous section can be written

```
method Increment(c: Cell, x: int)
  recommends c != null
  recommends 0 <= x < 10
  modifies c
  ensures c.data == old(c.data) + x
{
  var y := c.data; //@ allow
  if x < 0 || 10 <= x {
    throw PreconditionFailure();
  }
  c.data := y + x;
}
```

By including the `allow` directive, the verifier is satisfied that the first `recommends` clause is tested and properly handled. Actually, there is a difference between this `Increment` method and the one in the previous section, namely that if `c` is `null`, this one will at run time throw a `NullPointerException`, whereas, the way we wrote it, `Increment` in the previous section throws a `PreconditionFailure` exception. As we mentioned in §1, the particular exception thrown as part of parameter validation is not so important. But if you're worried about this, just think of `PreconditionFailure` as a supertype of all parameter-validation exceptions for now; we'll return to this issue in the next section.

In the other fail-fast pattern from §1, parameter validation is delegated to another method. To illustrate, suppose there is another method

```
method IncrementAndDecrement(c: Cell, x: int, z: int)
  recommends c != null
  recommends 0 <= x < 10 && 0 <= z < 10
  modifies c
  ensures c.data == old(c.data) + x - z
```

Since `Increment` is a special case of this more general method, it can be implemented by a call to `IncrementAndDecrement`. As we glean from its `recommends` clauses, `IncrementAndDecrement` has to do (a superset of) the parameter-validation checks that `Increment` has to do, so it seems everything `Increment` has to do can be done by a single call to `IncrementAndDecrement`.

If we implement `Increment` as we just discussed, then the verifier would complain. It would complain that the call to `IncrementAndDecrement` does not live up to the precondition of the callee—remember that `recommends` clauses are enforced as preconditions at call sites. It would also complain that `Increment` is not doing the parameter validation that its `recommends` clauses demand. We solve both of these problems by marking the call with the `allow` directive:

```
method Increment(c: Cell, x: int)
  recommends c != null
  recommends 0 <= x < 10
  modifies c
  ensures c.data == old(c.data) + x
{
  IncrementAndDecrement(c, x, 0); //@ allow
}
```

In our design of the `recommends` specifications, placing an `allow` directive on a call causes the verifier to consider the callee's `recommends` clauses as control flow that may throw exceptions. That is, the call above with the `allow` directive is treated as

```
if ¬(c != null && 0 <= x < 10 && 0 <= z < 10) {
  throw PreconditionFailure();
}
IncrementAndDecrement(c, x, 0);
```

This will satisfy the verifier as a correct implementation of `Increment`.

A detail remains. As for the first pattern above, we need to be careful with any code that pays specific attention to which parameter-validation exception is thrown. In the `allow`-rewrite of the call above, we wrote `throw PreconditionFailure()`, but at run time, `IncrementAndDecrement` may actually throw some other exception. To make this more exact, our semantics is not exactly the `throw PreconditionFailure()` we showed above; rather, it is to throw *some* parameter-validation exception. If you think of `PreconditionFailure` as the supertype of all parameter-validation exceptions, then it would be accurate to replace the `throw` above with a call

```
_ThrowPreconditionFailure();
```

where

```
method _ThrowPreconditionFailure()
  ensures false
  throws PreconditionFailure ensures true
  throws * ensures false
```

is a method invented by the verifier. Appropriately, this method is not too specific about which `PreconditionFailure` subtype the thrown exception has or how that exception is created.

6. Recommends else

As we have defined the basic `recommends` clause, a method implementation is obligated to throw a `PreconditionFailure` if a recommended condition does not hold on entry. This is the *specification*, but an implementation is, as usual, allowed to take a more specific action. In particular, the exception thrown by the implementation may be an exception of any subtype of `PreconditionFailure`. This can be useful when debugging the stack trace from a parameter-validation crash.

Though our contention is that one should not put too much emphasis on the actual exception thrown, we are sympathetic to the practice that the documentation style in Java and .NET prescribes particular exceptions to be thrown in response to invalid parameters. To let a verifier check that method implementations follow the documented behavior, we extend the basic `recommends` clause with the ability to specify the use of more specific exceptions.

6.0. Exception designations

To specify that violations of a `recommends` clause are to be countered by a designated exception, we allow an optional `else` suffix. For a condition `Pre` on the method's pre-state and an exception type `Exc`, the precondition

```
recommends Pre else Exc
```

expresses the intention that `Pre` hold on entry to the method, and obliges the implementation to throw an `Exc` exception if `Pre` does not hold. A basic `recommends` clause defaults to having the suffix `else PreconditionFailure`.

For example, the `Increment` method can be specified and implemented as follows:

```
method Increment(c: Cell, x: int)
  recommends c != null else NullPointerException
  recommends 0 <= x < 10 else IllegalArgumentException
  modifies c
  ensures c.data == old(c.data) + x
{
  var y := c.data; //@ allow
  if x < 0 || 10 <= x {
    throw IllegalArgumentException();
  }
```

```
    c.data := y + x;
}
```

6.1. Multiple recommends clauses

The `recommends else` constructs bring up a question: What do these specifications mean if multiple recommended conditions do not hold? For example, what does the specification above say for a call `Increment(null, 12)`? One could adopt the design that `recommends` clauses are ordered and that the first condition to fail is the one whose `else` exception must be thrown. However, this would force specifications to be overly specific, because they would not give implementations a choice about the order. Indeed, insisting on such an order would even be stricter than the natural-language descriptions of methods in the Java and .NET standard libraries, which often do not state an order.[2]

So, we instead adopt the design that `recommends` clauses are unordered. If any recommended condition does not hold, then the implementation is obligated to throw *some* exception and must throw an exception that corresponds to one of the failing conditions.

For example, consider a specification

```
recommends A else X
recommends B else Y
ensures Post
```

where `X` and `Y` denote disjoint exception types. Its meaning for a caller is, as before, that both `A` and `B` are required to hold. For an implementation, the meaning of the specification is[3]

```
when A && B
  ensures Post
when ¬A || ¬B
  ensures false
  throws X ensures ¬old(A)
  throws Y ensures ¬old(B)
  throws * ensures false
```

This says that if `A` and `B` are both true, then the implementation must live up to the given post-specification. If `A` and `B` are both false, then one of the exceptions `X` and `Y` must be thrown, but the choice between these two is up to the implementation. If only one of `A` and `B` is false, then the exception for that one must be thrown. (Recall that the `throws ensures` clause is read as "if the given exception is thrown, then the given condition holds".)

---

[2] Some methods in Java 11, like `java.lang.System.arraycopy` (https://docs.oracle.com/javase/7/docs/api/java/lang/System.html), spell out the order, whereas many others, like `javax.crypto.Cipher.getInstance` (https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html), do not.

[3] Since the second `when` group has an empty `modifies` clause, `old(A)` is the same as `A`. Nevertheless, we will continue to write `old(A)` to emphasize that we're referring to the value of `A` on entry to the method.

Note that it is possible to write a specification that says recommended conditions must be checked in some particular order. For example, we can modify the previous example to specify that `X` should be thrown if `A` doesn't hold, regardless of whether or not `B` holds:

```
recommends A else X
recommends ¬A || B else Y
ensures Post
```

(With an implication operator, the second recommended condition can be equivalently written as `A ==> B`.) To see that this says what is intended, it is perhaps easiest to write out the implementation-side view of this specification, which is

```
when A && (¬A || B)
  ensures Post
when ¬A || ¬(¬A || B)
  ensures false
  throws X ensures ¬old(A)
  throws Y ensures ¬old(¬A || B)
  throws * ensures false
```

After some logical simplifications and distributing `old` over connectives, this is equivalent to

```
when A && B
  ensures Post
when ¬A || ¬B
  ensures false
  throws X ensures ¬old(A)
  throws Y ensures old(A) && ¬old(B)
  throws * ensures false
```

Here, we see that, in the event that either `A` or `B` is false, only `X` and `Y` are allowable outcomes of the implementation. And in that situation, exception `Y` is allowed to be thrown only if the recommended condition `A` did in fact hold and `B` did not.

In general, different `recommends else` clauses may mention the same exception. And if, like in Java, exception types form a subtyping hierarchy, then the exceptions in different `recommends else` clauses may overlap. To illustrate the interpretation of these, suppose `X` and `Y` are disjoint exception types and that `X0` is a proper subtype of `X`. Then, the implementation-side view of

```
recommends A else X
recommends A0 else X0
recommends B0 else Y
recommends B1 else Y
requires C
modifies Frame
ensures Post
```

is

```
when A && A0 && B0 && B1
  requires C
  modifies Frame
  ensures Post
when ¬A || ¬A0 || ¬B0 || ¬B1
  ensures false
  throws X \ X0 ensures ¬old(A)
  throws X0 ensures ¬old(A) || ¬old(A0)
  throws Y ensures ¬old(B0) || ¬old(B1)
  throws * ensures false
```

where we have used the notation `X \ X0` to denote any exception type that is a subtype of `X` but not a subtype of `X0`. Formulaically, for every exception `Z` mentioned in some `recommends else` clause, the interpretation has a clause

```
throws Z \ Z0 \ Z1 \ ... ensures ¬old(P) || ¬old(Q) || ¬old(R) || ...
```

where `Z0`, `Z1`, ... are proper subtypes of `Z` mentioned in `else` designations and `P`, `Q`, `R`, ... are the conditions in `recommends` clauses whose `else` designation is a subtype of `Z` but not a subtype of any of `Z0`, `Z1`, ...

6.2. Well-definedness

As we touched on in §4, specifications have to be well-defined. Since we consider `recommends` clauses to be unordered, our design is also to check their well-definedness independently. In particular, this means that each `recommends` condition must be well-defined whether or not the other `recommends` conditions hold.

To illustrate, suppose we want to create an array containing the first `n` elements of a given array `a`. We would then write the specification

```
method CopyN(a: array<int>, n: int) returns (r: array<int>)
  recommends a != null else NullPointerException
  recommends 0 <= n else NegativeArraySize
  recommends a != null ==> n <= a.length else ArrayIndexOutOfBounds
  // ...
```

Note that the last `recommends` clause must include the "`a != null ==>`", as it may not presume the first `recommends` to hold. Also, an easy mistake to make is to instead write "`a != null &&`", but that would allow `ArrayIndexOutOfBounds` to be thrown in response to `a` being passed in as `null`.

7. Combining recommends and requires

We have motivated our work by the need to write specifications at the boundary between unverified and verified code. Our general guideline is to use `recommends` preconditions for methods that may be called by unverified code and to use standard `requires` preconditions for methods that are called only from call sites where those conditions are

enforced statically by a verifier. But there are also some situations where a method may be specified by a combination of `recommends` and `requires`.

One example of such a situation is an auxiliary method, perhaps like `ASimilarFail-FastExample` in §1, that is not callable from outside the verified module. In addition to the conditions for which this method promises to throw parameter-validation exceptions, there may be additional conditions that the auxiliary method wants to assume that callers establish. Since the auxiliary method is private to the verified module, all of its callers are verified, so these additional conditions can indeed be enforced at all call sites.

Another example of such a situation is when a precondition is complicated, which means the condition wouldn't be checked at run time anyway. In such a case, the caller is expected to establish the condition and there is no telling what might happen if the caller doesn't live up to this obligation. For example, a method that searches an array for a given element may expect the array to be sorted and may expect a given comparison operator to be transitive. Here, it makes sense to document the transitivity precondition and to let the method implementation be verified under the assumption that the precondition does hold.

To support combinations of `recommends` and `requires`, we need to be precise about the semantics. As before, for callers, there's no difference between the two kinds of preconditions, except that any `allow` directives on a call apply only to `recommends` clauses, never to `requires` clauses. For implementations, we consider `requires` conditions to be contingent on the `recommends` conditions being true. That is, an implementation is not allowed to assume `requires` conditions until after the `recommends` conditions have been checked.

To illustrate, for disjoint exception types `X` and `Y`, consider the following specification:

```
recommends A else X
recommends B else Y
requires C
requires D
ensures Post
```

For the method implementation, the expanded view of this specification is

```
when A && B
  requires C
  requires D
  ensures Post
when ¬A || ¬B
  ensures false
  throws X ensures ¬old(A)
  throws Y ensures ¬old(B)
  throws * ensures false
```

That is, the `requires` conditions `C` and `D` apply only if `A` and `B` hold on entry. So, effectively, the condition that an implementation may assume on entry is

```
A && B  ==>  C && D
```

where ==> denotes implication.

As follows from the expansion above, the well-definedness checks of the `requires` conditions are allowed to assume both `A` and `B`. For example, the array dereferences in the following specification are well-defined because of the `recommends` clauses:

```
method Search(a: array<int>, lo: int, hi: int, key: int)
     returns (r: Option<int>)
  recommends a != null
  recommends a != null ==> 0 <= lo <= hi <= a.Length
  requires forall i, j :: lo <= i < j < hi ==> a[i] <= a[j]
  // ...
```

8. Prototype implementations

In this section, we discuss a prototype of `recommends` that we have built in the OpenJML tool for Java annotated with JML specifications. We also discuss possibilities of adding `recommends` specifications to the verification-aware language Dafny.

8.0. JML and OpenJML

8.0.0. recommends clauses The Java Modeling Language (JML) [5] and its supporting tool OpenJML [4, 3] implement a prototype version of the `recommends else` clause described in this paper. Here is an example[4]

```
//@   recommends v != null else NullPointerException;
//@   ensures \result == v.value;
//@ pure
public int get(/*@ nullable */Value v) {
  return v.value; // implicit allow because of the recommends
}
```

We discuss defaults for `allow` behavior in §8.0.1.

JML already has the capability to specify the behavior of exceptional control paths. The following code is the equivalent of the above example, but in the older style:

```
//@ public normal_behavior
//@   requires v != null;
//@   ensures \result == v.value;
//@ also public exceptional_behavior
//@   requires v == null;
//@   signals_only NullPointerException;
//@ pure
```

[4] The `nullable` keyword avoids the default situation in JML in which `Value` is implicitly a non-null type. We use it in this section, since the focus of our discussion is on using specifications for parameter validation.

```
public int get(/*@ nullable */Value v) {
  return v.value; // implicit allow because of
                  // the exceptional_behavior
}
```

Compared to using `recommends else`, this specification is verbose and repetitive; it is worse when there are many exception conditions to consider. Furthermore, it is easy during code maintenance to have the two preconditions become out of sync. The `recommends else` syntax is a more readable and understandable equivalent. Though the normal-termination effect of this program is simple, the exceptional-behavior specification is typical even of much more complex methods.

The method in the example above is without side-effects (is `pure`) in both the normal and exceptional behaviors. That is not always the case; when it is not, then separate frame conditions (notated in JML by `assignable` clauses) must be written:

```
//@ public normal_behavior
//@   requires v != null;
//@   assignable v.value;
//@   ensures v.value == i;
//@ also public exceptional_behavior
//@   requires v == null;
//@   assignable \nothing;
//@   signals_only NullPointerException;
public void set(/*@ nullable */Value v , int i) {
  v.value = i; // implicit allow
}
```

and correspondingly

```
//@   recommends v != null else NullPointerException;
//@   modifies v.value; // does not apply to the else case
//@   ensures v.value == i;
public void set(/*@ nullable */Value v, int i) {
  return v.value; // implicit allow
}
```

Our design of `recommends` clauses does not let an implementation have any side-effects until after the `recommends` conditions have been tested. In our experience in specifying Java programs, exceptional control flow almost never has any side-effects. Therefore, we expect that many current specifications of exceptional behavior can be simplified using `recommends else` clauses. Even in the simple examples in this section, we see that `recommends` clauses add readability and save several lines of specification. If the method being specified does allow side-effects in exceptional behavior, then the more lengthy `normal_behavior`/`exceptional_behavior` form must be used.

8.0.1. allow designations OpenJML also implements the `allow` designation as described above, with a bit of enhancement. If a language construct might throw a particular subtype of `RuntimeException` then:

- if there is no `allow` designation that includes a supertype of that exception type and the verifier cannot establish that the offending condition never happens, then a verification warning is issued for that language construct on that line;
- otherwise, the verifier treats the language construct as if the exceptional control flow might happen, and continues on to verify any internal exception catches within the method or the method's `throws` clauses.

But OpenJML has these enhancements:

- If the language construct in question is nested inside a `try-catch` block for the relevant exception or if the method specification itself declares (in an explicit JML `signals_only` clause) that it might throw that exception, then that construct has an implicit `allow`.
- In order to turn off the implicit `allow` where needed, OpenJML has a corresponding `forbid` designation, which will cause the verifier to check that no exception can be thrown from that construct.
- The default exception if an `allow` or `forbid` has no exception stated is `RuntimeException`; a comma-separated list of exception names is also permitted.

This language feature solves a usability problem with JML that confused users. A user trying out JML might write this simple method and specification:

```
//@ ensures \result == c[i];
public int value(int[] c, int i) {
  return c[i];
}
```

Without a precondition on the value of `i`, OpenJML will issue a verification error that `i` might not be in the range of allowed indices for the array `c`. No problem for the user there. But then the user would go on to try some exception handling:

```
//@ ensures \result == c[i];
public int value(int[] c, int i) {
  try {
    return c[i];
  } catch (ArrayIndexOutOfBoundsException e) {
    // ...
  }
}
```

Now the user does not expect to see verification errors on `c[i]`, but originally OpenJML would. With these proposed features, OpenJML takes the presence of the `catch` clause in this example as evidence of the user's intent that an exception should be `allow`ed at `c[i]`. If that is not the desired behavior, an explicit `forbid` can be written. Even without the `catch` clause, the user can use an explicit `allow` to clearly document that an exception is allowed.

8.1. Dafny

Dafny is a programming language designed for verification [11, 12]. Failures in Dafny are different in two major ways from what we have discussed for Java. Still, the situation of calling verified Dafny methods from unverified code in other languages can benefit from `recommends` specifications.

One major difference is that what in a language like Java or C# would be a run-time exception is typically a verification failure in Dafny. For example, writing `c.v` or `a[j]` in Dafny gives rise to proof obligations that `c` is not `null` and `j` is in range. The program is not valid—and no compiled code is emitted—unless such proof obligations can be validated by the static verifier. In other words, Dafny's language semantics already follows a strict discipline.

Another major difference is that Dafny does not have exceptions. In languages like Java and C#, exceptions are used for both recoverable and unrecoverable errors, whereas in Dafny, there is a different mechanism for each of these. In the rest of this section, we show how our `recommends` declarations could be added to Dafny.

8.1.0. Recommendation failures as unrecoverable errors The most natural way to guard verified Dafny code from unverified callers is to test preconditions and use an unrecoverable error in response to any detected violation. Unrecoverable errors are produced using Dafny's `expect` statement, which performs a run-time check of its given condition and halts program execution if the condition evaluates to `false`.[5]

For example, here is a method with the envisioned `recommends` clauses for Dafny:

```
method Double(a: array?<int>, i: int)
  recommends a != null
  recommends a != null ==> 0 <= i < a.Length
  modifies a
  ensures a[i] == 2 * old(a[i])
{
  expect a != null;
  expect 0 <= i < a.Length;
  var x := a[i];
  a[i] := x + x;
}
```

There are several things to note in this simple example.

The first is a reminder that callers written in Dafny are verified, so such callers would be verified to satisfy the `recommends` preconditions.

The second is that a method like this is more commonly written with Dafny's non-null type `array<int>`. However, if this method is at the boundary from unverified code written in a different language that does not support non-null reference types, then it's

[5] In other words, the `expect` statement trades a verification assumption for a run-time check. The run-time behavior of an `expect` statement is similar to an always-enabled `assert` statement in Java, but from the verification perspective, the `assert` condition is checked by the verifier whereas the `expect` condition is assumed by the verifier.

safer to declare the parameter to be of the nullable type `array?<int>` and to list the non-nullness as a precondition.

The third thing to note is that, without `else` suffixes, the `recommends` clauses may as well be ordered, in which case the `a != null` antecedent can be dropped in the second `recommends` clause.

The fourth is that the `expect` statements for this simple example are, except for the ordering, identical to the recommended conditions, which makes it tempting to design the `recommends` clauses to compile into `expect` statements automatically. However, the condition in an `expect` statement must be compilable, whereas preconditions in Dafny often use specification-only ("ghost") features. Therefore, imposing on method implementations to add tests for recommended conditions (as we have done everywhere else in this paper) gives more flexibility.

The fifth is that it's not possible in Dafny to replace the body of `Double` with

```
{
  var x := a[i]; //@ allow
  a[i] := x + x;
}
```

where the `allow` directive is intended to rely on the language's run-time checks to test and handle `a != null` and `0 <= i < a.Length`. This is because there are no run-time checks for these operations in Dafny, so turning off verification for the indicated source line would not properly implement the fail-fast parameter validation we're after.

The sixth and final thing to note is that `allow` directives still make sense on calls to methods with `recommends` clauses. This would allow delegating `recommends` checking to another method, as we have discussed earlier in the paper.

In the method implementation's view, a specification

```
recommends A
requires C
modifies Frame
ensures Post
```

expands to

```
when A
  requires C
  modifies Frame
  ensures Post
when ¬A
  ensures false
```

where, as earlier in the paper, we have used `when` groups as a way of explaining when various specification clauses apply. In the case where `A` does not hold on entry, the expanded specification shows that the method implementation is obliged to establish `false` upon normal termination, and the only way to do this is via the `expect` statement, which avoids normal termination by generating an unrecoverable error.

8.1.1. Recommendation failures as recoverable errors Another possible way of adding `recommends` clauses to the Dafny language is to tie them to the standard mechanism for reporting recoverable errors. For this, Dafny uses a return value of a *failure-compatible type*. For illustration, we will use the typical type

```
datatype Result<T> = Success(value: T) | Failure(error: string)
{
  function method IsFailure(): bool // ...
  function method PropagateFailure(): Result<T> // ...
  function method Extract(): T // ...
}
```

whose three shown members make it a failure-compatible type [13].

For a method that returns a failure-compatible type, we can oblige the implementation to return a failure if a recommended condition does not hold. For example, the specification of

```
method GetAny<T>(a: array?<T>) returns (r: Result<T>)
  recommends a != null && a.Length != 0
  ensures r.Success? ==>
          exists i :: 0 <= i < a.Length && r.value == a[i]
{
  if a == null {
    return Failure("array is null");
  }
  if a.Length == 0 {
    return Failure("array has no elements");
  }
  return Success(a[a.Length / 2]);
}
```

promises, in the event of success, to return an element of the array `a`. The `recommends` clause says that the implementation must return a failure if the recommended condition does not hold on entry. As written, the specification allows the method to return a failure in other situations, too.

In the method implementation's view, the specification

```
recommends A
requires C
modifies Frame
ensures Post
```

expands to

```
when A
  requires C
  modifies Frame
  ensures Post
when ¬A
  ensures r.IsFailure()
```

Dafny has special syntax that makes it easy to propagate failures and easy to use successful values. For a method `M` that returns a failure-compatible type, the statement

```
x :- M();
```

(note the operator `:-` instead of the usual assignment operator `:=`) immediately propagates any failure that `M` returns. If `M` returns success, then the successful value is extracted into `x`. The context using the `:-` statement must itself allow a failure-compatible type to be returned. (Dafny's `:-` operator and failure-compatible types are closely related to the `?` operator and built-in `Result` type in the Rust language [10].)

The use of the `allow` directive to delegate parameter validation is conveniently combined with the `:-` operator. For example, consider a method

```
method GetElement<T>(a: array<T>, i: nat) returns (r: Result<T>)
  recommends i < a.Length
  ensures r.Success? ==> r.value == a[i]
{
  if i < a.Length {
    return Success(a[i]);
  } else {
    return Failure("index out of bounds");
  }
}
```

We're supposing this method is called only from within Dafny code, since it takes a non-null array and a non-negative integer as parameters (and these types may not be available in the foreign language that otherwise may call the method). Even for this Dafny-only method, the `recommends` clause usefully allows the implementation of `GetAny` to delegate its responsibility to check the emptiness of the array:

```
method GetAny<T>(a: array?<T>) returns (r: Result<T>)
  recommends a != null && a.Length != 0
  ensures r.Success? ==>
          exists i :: 0 <= i < a.Length && r.value == a[i]
{
  if a == null {
    return Failure("array is null");
  }
  r :- GetElement(a, a.Length / 2); //@ allow
}
```

9. Related work

There are several contract languages that provide run-time checking of preconditions. Among these are the programming languages Eiffel [15, 6] and Spec# [1] and the modeling languages JML for Java [5, 0, 4] and ACSL for C [2]. The compiled preconditions provide fail-fast checking at run time, while still allowing the static verifiers for these languages to check the preconditions at call sites. The `requires otherwise` clauses in

Spec#, which inspired our `else` suffixes, also allow customization of which run-time exception is thrown in response to violated preconditions [14].

In these languages, the condition that is compiled into a run-time check is the condition given in the `requires` clause. This forgoes the flexibility of having ghost expressions in specifications and compilable counterparts in method implementations. It also misses the opportunity to save on some explicit checks using the common patterns we showed in §1.

These contract languages generally allow the expanded implementation-side view of our `recommends` clauses to be specified explicitly. This supports programmer-defined testing of the preconditions. Unfortunately, this long form of specifications is error-prone to write and hard to read. Moreover, with just the implementation-side view of the specification, there is nothing that enforces the intended preconditions at call sites.

The only way we see to reap the benefits of the two kinds of specifications is to provide different precondition interpretations for callers and callees. Our proposed `recommends` clauses achieve that. Used in conjunction with `allow` directives, it is also possible to choose between the interpretations at call sites.

The `recommends` capability helps address the case of unverified callers calling verified callees. A related problem, which we have not considered in this paper, is that of verified callers calling unverified methods (e.g., unverified libraries). We consider this to be a significantly different problem worth separate attention. Even with a mechanism for writing out-of-band specifications for unverified library methods, it's not easy to use run-time monitoring to detect whether or not callees follow the specifications. For example, if the precondition in such a specification is too weak, then the callee may cause irreparable damage before a run-time monitor has a chance to react. As another example, it is at best expensive to detect whether or not the callee modifies only what the specification says.

10. Concluding remarks

We have proposed a new specification-language feature, `recommends`, that (like common `requires` preconditions) states what conditions are *intended* to hold on method entry, but (unlike `requires` preconditions) does not trust that these conditions actually hold. Not only is a method implementation not allowed to assume these preconditions on entry, but the `recommends` feature forces the method implementation to test them on entry. This makes `recommends` suitable in situations where the method may be called from unverified code. The additional `allow` directive equips the software developer with flexibility and precision to say which locations in a program allow exceptional behavior and which locations outright forbid it.

We illustrated the naturalness of our new notation by example, and our prototype in JML and OpenJML illustrates its feasibility. Supporting the claim that our proposal is language agnostic, we described two ways in which `recommends` clauses could fit into Dafny, a verification-aware language that treats failures in a significantly different way from Java.

We hope that the clarity of `recommends` declarations will help move the unverified-verified boundary in the direction of more verified software.

Acknowledgments We thank Serdar Tasiran for suggesting the keyword `recommends` as a good description of the alternative precondition. We are grateful to the reviewers and Mattias Ulbrich for useful comments on a draft of this paper. Happy birthday, Reiner Hähnle, and thank you for your groundbreaking contributions to verifying software and the tooling that is taking us there.

References

0. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification — The KeY Book — From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.
1. Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: The Spec# experience. *Comm. ACM*, 54(6):81–91, June 2011.
2. Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Comm. ACM*, 64(8):56–68, 2021.
3. David R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer, 2011.
4. David R. Cok. JML and OpenJML for Java 16. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, FTfJP 2021, page 6567. ACM, 2021.
5. David R. Cok, Gary T. Leavens, and Mattias Ulbrich. *JML Reference Manual (2nd edition)*, 2021. https://www.openjml.org/documentation/JML_Reference_Manual.pdf.
6. ECMA International. *Eiffel: Analysis, Design and Programming Language*, 2nd edition edition, June 2006. Standard ECMA-367.
7. John B. Goodenough. Structured exception handling. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 204–224. ACM, January 1975.
8. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
9. John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys*, 44(3):16:1–16:58, June 2012. Article 16.
10. Steve Klabnik and Carol Nichols. *The Rust Programming Language*, 2018. https://doc.rust-lang.org/book/.
11. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, April 2010.
12. K. Rustan M. Leino. Accessible software verification with Dafny. *IEEE Software*, 34(6):94–97, 2017.
13. K. Rustan M. Leino, Richard L. Ford, and David R. Cok. *Dafny Reference Manual*, 2021. https://dafny-lang.github.io/dafny/DafnyRef/DafnyRef.
14. K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In Jorge R. Cuellar and Zhiming Liu, editors, *SEFM 2004—Second International Conference on Software Engineering and Formal Methods*, pages 218–227. IEEE, September 2004.
15. Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.