Modular Verification Scopes via Export Sets and Translucent Exports

K. Rustan M. Leino and Daniel Matichuk

Abstract Following software engineering best practices, programs are divided into modules to facilitate information hiding. A variety of programming-language constructs provide ways to define a module and to classify which of its declarations are private to the module and which are available to public clients of the module.

Many declarations can be viewed as consisting of a signature and a body. For such a declaration, a module may want to export just the signature or both the signature and body. This translucency is particularly useful when formally verifying a program, because it lets a module decide how much of a declaration clients are allowed to depend on.

This article describes a module system that supports multiple export sets per module. Each export set indicates the translucency of its exported declarations. The module system is implemented as part of the verification-aware programming language Dafny. Experience with the module system suggests that translucency is useful.

1 Introduction

Software engineers constantly have to manage the complexity of the software they are writing. One fundamental principle to follow is to try to isolate different pieces of functionality and to limit the interfaces to the details of that functionality [9]. Programming languages provide various features for such *information hiding*, for example, procedures that separate the what from the how, and classes and modules that hide both implementations and internal state.

K. Rustan M. Leino
Amazon Web Services, e-mail: leino@amazon.com

Daniel Matichuk
Data61 CSIRO, e-mail: daniel.matichuk@data61.csiro.au

When doing verification of software, the concerns are similar. But for verification, the concerns go deeper than the concerns involved during compilation or type checking. A compiler needs to know about the existence of certain methods and other declarations in a module interface, and a type checker additionally needs to know the type signatures of such declarations. For verification, a module interface may choose to either reveal or keep hidden a part of the definition of a declaration. For example, it makes a difference if the verifier is able to use the body of a function or just the signature and specification of the function. So, there is a need for finer-granularity control of information hiding during verification.

In this paper, we motivate and explain the design of the module system in the verification-aware programming language Dafny [4, 5]. Conspicuous among various influences on the design are ML [6] and Modula-3 [8]. One of the contributions is the way our module system lets exported declarations be either "provided" or "revealed". Another contribution is the working implementation of the module system, which has now been in use for a couple of years.

While the module system provides considerable flexibility, one of our design goals was to keep simple things simple. Our presentation also follows that format, starting off with the obvious features and moving into the more advanced.

Finding ways to write modular specifications of programs, so that one can reason about the programs modularly, has been one of the research agendas of Arnd Poetzsch-Heffter (e.g., [10, 7, 11, 12]). It is therefore our privilege to contribute this article in the Festschrift honoring Arnd Poetzsch-Heffter.

2 Preliminaries

We start by explaining the names introduced in modules and how modules can refer to names declared in other modules.

2.1 Modules and Imports

A rudimentary chore that a module system helps with is organizing the names of declarations in a program. Consider the following two modules.

```
module A {
  type S = set<char>
}
module B {
  function S(x: int): int {
    x + 1
  }
}
```

Each of these modules introduces a declaration `S`, in `A` being a type synonym for sets of characters and in `B` being a successor function on the integers. To make use of these modules, a client module, say `C`, has to import them.

```
module C {
  import A
  import B
  function CardPlusOne(s: A.S): int {
    B.S(|s|)
  }
}
```

With the fully qualified names `A.S` and `B.S`, there is no ambiguity as to what they are referring to.

A module that imports another can pick a new local name to refer to the imported module. This lets modules have long, meaningful names while clients can choose abbreviations. To illustrate, the following module `D` introduces the local name `M` to refer to the imported library `MathLibrary`.

```
module MathLibrary {
  function Abs(x: int): nat {
    if x < 0 then -x else x
  }
}
module D {
  import M = MathLibrary
  function AbsDiff(m: int, n: int): nat {
    M.Abs(m - n)
  }
}
```

In fact, the `import` declarations in module `C` above are abbreviations for `import` declarations that use as the local name the same name as the imported module. That is, `import A` is simply an abbreviation for `import A = A`.

2.2 Nested Modules

Modules can be nested. The enclosing module refers to the declarations in the nested module in the same way as for an imported module. It is also possible to import a nested module, which provides the dubious pleasure of giving it an alias.

```
module Outer {
  module Inner {
    type T = int
  }
  import I = Inner
```

```
  function Double(x: I.T): Inner.T {
    2 * x
  }
}
```

The names `Inner.T` and `I.T` refer to the same declaration (and since that declaration is a type synonym, both of those qualified names refer to the type `int`).

As we have seen, a module can import "sibling modules", that is, modules declared at the same level of lexical nesting. However, it is never possible for a nested module to refer to the enclosing module itself. This makes a module blissfully unaware of how deeply nested it is, giving it a kind of contextual independence.

Just like a declaration of a function or type, an `import` declaration also introduces a new name. And like the names of other declarations, such a name has to be unique. The name of the import can be used in prefixes of qualified names, as the following example illustrates:

```
module G {
  type T = string
}
module H {
  import HereComesG = G
}
module I {
  import H
  function WelcomeString(): H.HereComesG.T {
    "greetings"
  }
}
```

The qualified name `H.HereComesG.T` gives `I` a way to refer to the type `T` declared in `G`. But since `I` does not import `G` directly, the qualified name `G.T` is not defined in `I`.

As a final example of playing around with names, note that the right-hand side of an `import` declaration can be a qualified name, provided that, as for all qualified names, each part of the prefix is defined.

```
module J {
  import H
  import GG = H.HereComesG
  function FarewellString(): GG.T {
    "this has gone on for too long"
  }
}
```

In other words, without the `import H`, the import declaration that refers to `H` in its right-hand side would not be legal.

2.3 Imports are Acyclic

We define an *import relation* on modules. A module `X` is related to a module `Y` in the import relation if `X` contains an import declaration whose right-hand side designates `Y`. This relation is in general not transitive, but it has to be acyclic, which is enforced by the compiler or linker.

2.4 Opened Imports

Since an import declaration can introduce any local name for an imported module, qualified names don't have to get too long. However, there are situations where even two extra characters (like `M.`) feel too long. In those cases, which should be kept to a minimum, it is possible to import a module as `opened`. This causes the names of the imported module to be poured into the set of names declared directly by the importing module. Almost. The names declared directly in the importing module get preference over any opened-imported names. Furthermore, names of different opened-imported declarations are allowed to overlap, but trying to refer to them generates an error.

For illustration, consider the following modules.

```
module K {
  function Fib(n: nat): nat {
    if n < 2 then n else Fib(n-2) + Fib(n-1)
  }
  const G := 9.81
  function U(x: int): int {
    x + 2
  }
}
module L {
  function U(x: int): int {
    x - 2
  }
}
module N {
  import opened LocalNameForK = K
  import opened L
  function G(n: nat): nat {
    Fib(n) + LocalNameForK.Fib(n)
  }
  function H(): nat {
    G(12)  // G refers to the local function G, not to LocalNameForK.G
  }
```

```
  function V(): int {
    L.U(50)
  }
}
```

Module `N` imports `K` and `L` under the local names `LocalNameForK` and `L`, respectively. Both of the imports are opened, so the names declared in those modules are also available in `N` without qualification. Consequently, both the unqualified and qualified mention of `Fib` in function `G` are legal and refer to the `Fib` function declared in module `K`. Using the unqualified name `G` in `N` refers to the function declared in `N`, not to the constant `G` defined in `K`, since locally declared names are preferred over opened-imported names. Finally, it is not an error to opened-import both `K` and `L`, despite the fact that both of them declare a name `U`. However, an attempt to refer to `U` without a qualification (e.g., by changing `L.U` to just `U` in function `V`) would result in an "ambiguous name" error.

By preferring local names over opened-imported names and allowing duplicate opened-imported names as long as these are not referred to without qualification, our module design remains true to the motto that "declarations are unordered" in Dafny. This is in contrast to, for example, ML-based languages, which resolve similar ambiguities by ordering imports.

3 Export Sets and Translucency

So far, the set of names introduced in a module has been available indiscriminately to any importer. This does not promote the use of information hiding. To make it possible to limit what an importing module gets to know, Dafny makes use of *export sets*. In its simplest form, an export set lets a module indicate which of its names are available to importers. This makes it easy to encode the common idiom that categorizes declarations as being either *public* (that is, available to importers) or *private* (that is, for internal use only) to a module.

3.1 Export Sets

A module uses an `export` declaration to make manifest which declarations are available to importers. For example:

```
module M {
  export
    reveals F
  function F(x: int): int { x + 1 }
  function G(x: int): int { x - 1 }
}
```

```
module Client {
  import M
  function H(x: int): int {
    M.F(x) + M.G(x)  // error: unresolved identifier: G
  }
}
```

As indicated by the `reveals` clause of `M`'s `export` declaration, only one of `M`'s two functions (`F`) is revealed to importers. In general, the `reveals` clause contains a list of names. Analogously to the use of other kinds of clauses in Dafny, an `export` declaration can have several `reveals` clauses, with the same meaning as concatenating their lists of names into just one `reveals` clause.

An export set must be *self-consistent*. The essential idea is that if you project the module's declarations onto those that are exported, then all symbols must still resolve and type check. This is most easily understood through an example.

```
module BadExportSet {
  export
    reveals T, Inc, Identity
  type U = T
  type T = int
  function Inc(x: T): T { x + 1 }
  function Identity(x: U): U { x }  // error: U undeclared in export set
}
```

This module is not self-consistent, because it attempts to reveal function `Identity` while keeping private the type `U`, which appears in the function's signature. The projection of `BadExportSet` onto its export set would look like:

```
module BadExportSet {  // projection of the module onto its export set
  type T = int
  function Inc(x: T): T { x + 1 }
  function Identity(x: U): U { x }
}
```

It is clear from this projection that the export set was not self-consistent, since `U` is nowhere declared.

3.2 Translucent Exports

Even export declarations with selectively revealed declarations can give away more information than intended to importers. This is because most declarations can be considered to have two parts. For example, a function has a signature and a body. A `reveals` clause that mentions a function reveals both the signature and body of that function to importers. Dafny's `export` declarations also have `provides` clauses.

Mentioning a function in a `provides` clause provides only the signature, not the body, of the function to importers.

For the purpose of export sets, we think of every Dafny declaration as having a *signature* part and a *body* part. Together, the signature and body parts make up the entire declaration. What is included in the signature part depends on what kind of declaration it is. For a function, the signature part includes the function's name, its type signature (that is, its type parameters, the in-parameters and their types, and the result type), and the function's specification (that is, its pre- and postcondition and its frame specification). For a type declaration, the signature part is the name of the type, its type parameters (including any variance annotations), and any type *characteristics* that can be declared (e.g., supports equality, contains no references). For a constant, the signature is the name and type of the constant, but not its defining expression. For a method or a lemma, the signature includes the name, its type signature (type parameters, in- and out-parameters and their types), and its specification.

The following example shows a mix of `provides` and `reveals` clauses.

```
module Accumulator {
  export
    provides T, Zero, Add, Get, Behavior
    reveals GetAndAdd
  datatype T = Nil | Cons(int, T)
  function Zero(): T {
    Nil
  }
  function Add(t: T, x: int): T {
    Cons(x, t)
  }
  function Get(t: T): int {
    match t
    case Nil => 0
    case Cons(x, tail) => x + Get(tail)
  }
  function GetAndAdd(t: T, x: int): (int, T) {
    (Get(t), Add(t, x))
  }
  lemma Behavior(t: T, x: int)
    ensures Get(Zero()) == 0
    ensures Get(Add(t, x)) == x + Get(t)
  {
    // proof follows trivially from the function definitions
  }
}
```

Again, the export set has to be self-consistent. The projection of module `Accumulator` onto its export set looks like this:

```
module Accumulator {  // projection of the module onto its export set
  type T
  function Zero(): T
  function Add(t: T, x: int): T
  function Get(t: T): int
  function GetAndAdd(t: T, x: int): (int, T) {
    (Get(t), Add(t, x))
  }
  lemma Behavior(t: T, x: int)
    ensures Get(Zero()) == 0
    ensures Get(Add(t, x)) == x + Get(t)
}
```

In this projection, we have indicated that `T` is exported as an opaque type—the fact that it denotes a datatype and has a certain list of constructors is kept private to the module. To an importing module, functions `Zero`, `Add`, and `Get` are just arbitrary functions. By calling lemma `Behavior`, an importer learns some properties of these functions, but the importer has no way to prove these properties directly. In this way, this lemma serves as an exported contract of the otherwise opaque functions.

By only providing, not revealing, type `T`, the `Accumulator` module retains the ability to change the representation of the type without affecting any clients. For example, the module can replace its declarations of `T`, `Zero`, `Add`, and `Get` by

```
type T = int
function Zero(): T { 0 }
function Add(t: T, x: int): T { t + x }
function Get(t: T): int { t }
```

or by

```
type T = int
function Zero(): T { 3 }
function Add(t: T, x: int): T { t + 2 * x }
function Get(t: T): int { (t - 3) / 2 }
```

without having to reverify any modules that import `Accumulator`.

Unlike the other declarations in `Accumulator`, function `GetAndAdd` is revealed in the export set. This means that the function's behavior is fully known to any exporter. For example, this lets an importer prove assertions like

```
assert Accumulator.Get(t) == Accumulator.GetAndAdd(t, 5).0;
```

In Dafny, the verifier always reasons about calls to methods and lemmas in terms of their specifications, never in terms of their bodies. Thus, there would be no difference between providing and revealing a method or lemma in an export set, so we decided to reduce confusion by disallowing methods and lemmas from being mentioned in `reveals` clauses (accompanied by a useful error message, of course).

Similarly, a name introduced by an `import` declaration can only be provided in an export set, not revealed.

If a declaration happens to be mentioned in both a `reveals` clause and a `provides` clause, it is the same as just revealing it. That is, the export set is like a set of declaration signature parts and declaration body parts, so the export set is the union of all of the parts added on behalf of `reveals` and `provides` clauses.

An export declaration can use the clause `provides *`, which is a shorthand for providing all declarations in the module. Similarly, the clause `reveals *` is a shorthand for revealing all declarations in the module that can be revealed and providing the rest (that is, `reveal *` provides methods, since methods cannot be revealed).

3.3 Effect of Import Aliases in Export Sets

Export sets are only allowed to mention names declared in the module itself. For example,

```
module P {
  type T = int
}
module Q {
  export
    provides P
    reveals U, V
  import opened P
  import R = P
  type U = P.T
  type V = T
}
```

is allowed, but adding `T` or `P.T` to either of the export clauses would not be allowed. Note that `T` cannot be exported, despite the fact that the opened import allows it to be mentioned unqualified inside `Q`. Stated differently, by marking an import as `opened`, the ability to mention the imported declarations as unqualified is not inherited by further importers.

Furthermore, neither `U` nor `V` could be revealed if the export set didn't provide `P`. More precisely, module `Q` declares three aliases to the module `P`, namely `P` and `R` and the opened-import alias. The self-consistency check is not performed textually, but uses these import aliases. Therefore, it is more correct to say that neither `U` nor `V` could be revealed if `Q`'s export set didn't provide some alias for the module where `T` (which is mentioned in the body parts of `U` and `V`) is declared. So, module `Q` would still be fine if `provides P` were replaced by `provides R`.

3.4 Default Export Set

If a module does not contain any `export` declaration, everything is revealed as an export. This makes it simple to start learning about modules and imports, like we did in Section 2 before we had mentioned the `export` declaration. In other words, without any explicit `export` declaration, Dafny acts as if the module had been declared with

```
export
  reveals *
```

As soon as a module gives some `export` declaration, only what is explicitly exported is made available to importers. This makes it easy to go from the implicit "everything is revealed" to an explicit "nothing is revealed": simply add the empty export declaration

```
export
```

That is, one keyword is all it takes.

4 Multiple Export Sets

Not all importers are alike. For instance, in some cases, a module may have some declarations that all users will want to know about, whereas some others are useful for companion modules, so-called *friends* modules. To support this, Dafny allows a module to have multiple export sets, each providing its own *view* of the module.

4.1 Named Export Sets

The following module defines two export sets, named `Public` and `Friends`.

```
module R {
  export Public
    provides Combine
  export Friends
    reveals Combine
  function Combine(x: set<int>, y: set<int>): set<int> {
    x + y - (x * y)
  }
}
```

The export set `Friends` reveals the signature and body of function `Combine`, whereas the export set `Public` only provides the signature of `Combine`. Hence, an importer of the `Public` view of the module is not able to verify an assertion like

```
assert R.Combine({2,3}, {3,5}) == {2,5};
```

whereas an importer of the `Friends` view is.

Each export set is checked separately to be self-consistent.

4.2 Importing a Named Export Set

When a module has several export sets, an importer indicates which export set to import by appending to the module name a back-tick and the name of the desired set. Here are two such client modules:

```
module R_PublicClient {
  import R = R`Public
  lemma Test() {
    assert R.Combine({2,3}, {3,5}) == {2,5};  // error: not provable here
  }
}
module R_FriendClient {
  import R`Friends
  lemma Test() {
    assert R.Combine({2,3}, {3,5}) == {2,5};
  }
}
```

Note that if the local name for the module is omitted, it defaults to the name of the module, without the name of the selected export set. That is, `import R`Friends` is the same as `import R = R`Friends`.

The back-tick notation is not new to import declarations in Dafny. Dafny uses the back-tick notation in a similar way in frame specifications, where an expression denoting one or a set of objects can be further restricted to a field of that object or objects. Dafny borrows this use of back-tick in frame specifications from Region Logic [1]

4.3 Eponymous Export Sets

When an `export` declaration omits a name, as in all of our examples before Section 4, then it defaults to the name of the module itself. This eponymous export set is also what gets imported if an `import` declaration omits the back-tick and the export-set name. While most modules tend to have an eponymous export set, there is no requirement to have one. For example, module `R` above does not have an eponymous export set.

4.4 Example

A common idiom is to use the eponymous export set to stand for the most common view of the module. This means that most importers only need to mention the imported module in their `import` statement—no need for a back-tick and a specific export-set name. Any additional export sets are then given names.

For example, module `Accumulator` in Section 3.2 defined an export set:

```
export
  provides T, Zero, Add, Get, Behavior
  reveals GetAndAdd
```

It can also define a `Friends` view that reveals how the type `T` is represented:

```
export Friends
  provides Zero, Add, Behavior
  reveals T, Get, GetAndAdd
```

Using the `Friends` view, a new module can augment the `Accumulator` functionality, like this:

```
module AccumulatorMore {
  export
    provides A, Mul, Behavior
  import A = Accumulator
  import Acc = Accumulator`Friends
  function Mul(t: Acc.T, c: int): Acc.T {
    match t
    case Nil => Acc.Nil
    case Cons(x, tail) => Acc.Cons(c * x, Mul(tail, c))
  }
  lemma Behavior(t: Acc.T, c: int)
    ensures Acc.Get(Mul(t, c)) == c * Acc.Get(t)
  {
  }
}
```

This module needs to know the full definition of `T` to implement `Mul` and needs to know the full definition of `Get` to prove the lemma about `Mul`. Therefore, it imports the `Friends` view of `Accumulator`. To be self-consistent, the module's export set must provide a way to understand the type `Acc.T`, which appears in the signatures of the provided declarations `Mul` and `Behavior`. Adding `Acc` to the `provides` clause would make the export set self-consistent. However, this would provide all `AccumulatorMore` clients with the information entailed by the `Friends` view of `Accumulator` (e.g., the full definition of `T`), which results in less information hiding than desired. Therefore, module `AccumulatorMore` also declares an import for the eponymous export set of `Accumulator`, here with the local name `A`. Providing `A` in-

stead of `Acc` in the export set makes it self-consistent without passing on any of the `Accumulator` information that only friends need.

Here is a module that uses both the original and augmented functionality of `Accumulator`:

```
module AccTest {
  import A = Accumulator
  import M = AccumulatorMore
  lemma Test() {
    var z := A.Zero();
    var e := A.Add(z, 8);
    A.Behavior(z, 8);
    M.Behavior(e, 3);
    assert A.Get(M.Mul(e, 3)) == 24;
  }
}
```

A module decides which export sets to define and which parts of which declarations to include in these export sets. However, the module does not control where these export sets can be imported. That is, unlike in C++ [2], where a class defines which other classes are its "friends", the export sets in Dafny are more like the "friends interfaces" in Modula-3 [8], where each importer gets to decide whether or not it is a friend.

4.5 Combinations of Export Sets

If a module imports several views of a module, it obtains the parts from the union of those views. Syntactically, an imported declaration can only be qualified by the local name of an import that provides or reveals that declaration. But what is known about the imported declaration (that is, only its signature part or both its signature and body parts) is derived from the union of the imported parts, not just the import used to qualify the declaration.

For example, consider the following two modules.

```
module S {
  export AB reveals a, b
  export B provides b
  export AC provides a reveals c
  const a := 10
  const b := 20
  const c := 30
}
module T {
  import S0 = S`AB
  import S1 = S`B
```

```
  import S2 = S`AC
  lemma Test() {
    assert S0.a == S2.a;
    assert S0.a + S1.b == S2.c;
  }
}
```

Module `T` can refer to `S`'s constant `a` as either `S0.a` or `S2.a`, since both the imports `S0` and `S2` include the signature part of `a`, but trying to say `S1.a` in `T` would give an "unresolved identifier" error. Because `T` imports the export set `AB`, the body part of `a` (that is, the fact that `a` is defined to be `10`) is known in `T`, regardless of if `a` is syntactically referred to as `S0.a` or as `S2.a` (see the first assertion in the example). Similarly, despite the fact that `S1` only imports the signature part of `b`, `S1.b` is known in `T` to equal `20` on account of import `S0`.

It is also possible to combine several imported views of a module into a single local name for the module. This is done by following the back-tick with a set of names. For example, the local names `S0`, `S1`, and `S2` in module `T` can be combined into the one local name `S` as follows:

```
module T' {
  import S`{AB,B,AC}
  lemma Test() {
    assert S.a == S.a;
    assert S.a + S.b == S.c;
  }
}
```

4.6 Building an Export Set as an Extension of Another

When a module defines several export sets, it is often the case that one is a superset of another. For example, export set `Friends` is a superset of the eponymous export set in module `Accumulator` in Section 4.4. To make such superset relations easier to maintain, an export set can be declared to *extend* another. This is done by following the name of the new export set by `extends` and the list of export sets that are being extended.

So, instead of spelling out all the provided and revealed declarations of `Friends` in Section 4.4, the `Friends` export set can be defined to extend the eponymous export set:

```
export Friends extends Accumulator
  reveals T, Get
```

5 The Road We Traveled

The module system in Dafny has gone through several major revisions. Ostensibly, it seems that a module system just provides a way to carve up the namespace of a program, which gives the illusion that the task of designing it would be trivial. Our struggle with the design tells a different story.

Part of our difficulty stemmed from trying to rely on Dafny's features for *refinement* [3], which are also based on modules, to be the mechanism for hiding declarations and parts of declarations. Because every new refinement module creates a copy of the module being refined (to allow several different refinements of a module), it was difficult to determine when two imported declarations were the same and when they were different copies of some previously defined declaration. After experimenting with this and finally declaring the attempt a failure, our next quagmire became trying to deal gracefully with the legacy programs that had grown out of our experiments.

Our experience shows that the need to be able to decide separately about exporting a declaration's signature and exporting the declaration's body is fundamental for verification. This rendered an otherwise convenient keyword like `public` inadequate. Would we have more than one flavor of the keyword, like `public_sig` and `public_body`? Or would we have a single keyword `public` that could designate different parts of a declaration, depending on where in the declaration the keyword was placed? Searching for a consistent way to deal with this issue for different kinds of declarations caused us to look at what subsets of parts of each declaration could make sense to make visible. We were surprised to find that essentially every declaration had a signature part and a body part, and we were delighted that no declaration needed more than two parts.

There was also a design choice of how many visibility levels (like `private` and `public`) to provide. Remembering Modula-3's interfaces, we knew we could let the program (as opposed to the programming language) make this choice if we introduced named export sets in the language. This and the "every declaration has two parts" realization eventually led us to `export` declarations with `provides` and `reveals` clauses.

With export sets, our implementation of Dafny needed to be careful about what information to use when resolving names, when type checking, and when producing logical verification conditions. The aliasing of imports with different views made this issue quite delicate, and all in all it added up to a substantial implementation effort.

The current module system has seen two years of use without major issues.

6 Concluding Remarks

A module system that supports information hiding is important for any programming language. For a verification-aware language like Dafny, a good module system

not only explicates the things clients of a module can rely on versus the things the module is free to change without affecting clients, but also enables modular verification. We have described the Dafny module system and given examples of how its features can be applied when specifying and verifying programs. The module system is characterized by its multiple export sets, its translucent `provides` exports, and its transparent `reveals` exports.

Acknowledgements This work was done in 2016 when both of us were at Microsoft Research. We are grateful to the Ironclad team at Microsoft Research, especially Chris Hawblitzel, Jay Lorch, and Bryan Parno, who went through the pains of using the (several!) previous module systems of Dafny, and offered constant feedback and valuable suggestions. Jason Koenig and Michael Lowell Roberts were instrumental in experimenting with various module-system features that influenced the current design.

References

1. Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In Jan Vitek, editor, *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, volume 5142 of *LNCS*, pages 387–411. Springer, 2008.
2. Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.
3. Jason Koenig and K. Rustan M. Leino. Programming language features for refinement. In John Derrick, Eerke A. Boiten, and Steve Reeves, editors, *Proceedings 17th International Workshop on Refinement, Refine@FM 2015*, volume 209 of *EPTCS*, pages 87–106, 2016.
4. 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 *LNCS*, pages 348–370. Springer, April 2010.
5. K. Rustan M. Leino. Accessible software verification with Dafny. *IEEE Software*, 34(6):94–97, 2017.
6. Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, May 1997.
7. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.
8. Greg Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.
9. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. Reprinted as `www.acm.org/classics/may96/`.
10. Arnd Poetzsch-Heffter and Jan Schäfer. Modular specification of encapsulated object-oriented components. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 313–341. Springer, 2005.
11. Ina Schaefer and Arnd Poetzsch-Heffter. Using abstraction in modular verification of synchronous adaptive systems. In Serge Autexier, Stephan Merz, Leendert W. N. van der Torre, Reinhard Wilhelm, and Pierre Wolper, editors, *Workshop "Trustworthy Software" 2006*, volume 3 of *OASICS*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
12. Ina Schaefer and Arnd Poetzsch-Heffter. Compositional reasoning in model-based verification of adaptive embedded systems. In Antonio Cerone and Stefan Gruner, editors, *Sixth*

IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, pages 95–104. IEEE Computer Society, 2008.