What is a Module?

N. Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

July 13 , 2018

Overview

- Modularity in software has been a key concern since Doug McIlroy's plea at the 1968 NATO conference on software engineering.
- The concept of a *module* appears to be fundamental to programming and specification languages.
- Examples include Ada and ML modules, C++ templates, Z schemas, PVS theories, and SAL modules.[1]
- Yet, it has no precise definition
- A similar vagueness exists with respect to *process*, *class*, *object*, *method*.
- What is modularity?
- Why do we need it?
- How can we capture modularity in language?

[1] My perspective is informed by the module systems in PVS and SAL.

Overview

- Modularity in software has been a key concern since Doug McIlroy's plea at the 1968 NATO conference on software engineering.
- The concept of a *module* appears to be fundamental to programming and specification languages.
- Examples include Ada and ML modules, C++ templates, Z schemas, PVS theories, and SAL modules.[1]
- Yet, it has no precise definition
- A similar vagueness exists with respect to *process*, *class*, *object*, *method*.
- What is modularity?
- Why do we need it?
- How can we capture modularity in language?

[1] My perspective is informed by the module systems in PVS and SAL.

Overview

- Modularity in software has been a key concern since Doug McIlroy's plea at the 1968 NATO conference on software engineering.
- The concept of a *module* appears to be fundamental to programming and specification languages.
- Examples include Ada and ML modules, C++ templates, Z schemas, PVS theories, and SAL modules.[1]
- Yet, it has no precise definition
- A similar vagueness exists with respect to *process*, *class*, *object*, *method*.
- What is modularity?
- Why do we need it?
- How can we capture modularity in language?

[1] My perspective is informed by the module systems in PVS and SAL.

Some Language Design/Modularity Principles

Frege principle (Referential Transparency): Equal expressions should be interchangeable.

Chomsky Principle: A name is merely an abbreviation for something. The denotation of a name can be used in place of the name.

Most languages violate this principle. E.g., PVS theories cannot be used in place of the theory names.

Reynolds Principle: Language features should be orthogonal.

Scott Principle: Features should be nestable.

Occam Principle: Make no irrelevant distinctions.

Parnas Principle: Localize design decisions that change together.

Dijkstra Principle: Separate concerns between different aspects of computation.

Lampson Principle: Practical modularity stems from big components with small interfaces.

Berry Principle: Write everything (at most) once.

Corollary: Prove everything (at most) once.

What is the Point of a Module?

Packaging: Entire module can be referenced instead of individual components.

Naming: Names in a module can be distinguished from those in other modules.

Reuse: Distinct copies of the module can be obtained by varying the parameters.

Testing: A module is a unit of unit testing.

Abstraction: All interaction with the module instance must be through an external interface.

Documentation: Modules capture concepts that need to be documented together.

Information Hiding: Design and implementation of the module can vary as long as the abstract interface is satisfied.

Separate Compilation: Modules are units of separate compilation.

Composition: A module calculus introduces composition operators to define new modules from existing ones.

What is the Problem with Modules?

> *The black box nature of the decision procedure is frequently destroyed by the need to integrate it.*
>
> *Boyer and Moore*

- Modules make incompatible assumptions
- Communication overhead of communicating with a module is high
- Modularity gets in the way of fine-grained interaction

Perhaps it is easier to reimplement than reuse?

Language Examples: C++ Templates

Allows type and value abstraction in the definition of classes and functions.
Example (from Shapiro):

```
template <class T, int N> class Queue
  T queueEntries[N];
  int queueDepth;
   ⋮
  ;
```

Templates used by macro-expansion.

Language Example: ML modules

Structures package a collection of declarations.

The "type" of a structure is a *signature*, i.e., the declarations without the definitions.

Functors map structures to structures.

Example (from Munoz):

```
module type OrderSig =
  sig
    type t
    val comp : t -> t -> int
  end;;

module OrderedList(Order: OrderSig) =
  struct
    type element = Order.t
    type olist = element list
    ⋮
  end;;
```

Language Example: Z Schemas

- A schema consists of a signature and some predicates.
- The signature is the visible portion of the global state space.
- Schemas can either assert invariants or transitions.
- Schemas can be imported within other schemas and can take sets as parameters.
- Compatible schemas can be combined by logical operations.
- Transition schemas can be sequenced.

Modularity Example: PVS Theories

A PVS theory is a collection of type, constant, and formula declarations.
A theory can be parametric in certain types and constants.

```
functions [D, R: TYPE]: THEORY
 BEGIN
  f, g: VAR [D -> R]
  x, x1, x2: VAR D

  extensionality: POSTULATE
     (FORALL (x: D): f(x) = g(x)) IMPLIES f = g

  congruence:
    LEMMA f = g AND x1 = x2 IMPLIES f(x1) = g(x2)
 END functions
```

Theories can be instantiated, extended, combined, and interpreted.

Compositional Verification of Concurrent Modules

- Proof techniques that decompose $P(C_1 \| C_2)$ to $P_1(C_1)$ and $P_2(C_2)$, for some generic notion of composition $\|$, e.g., parallel composition.
- Typical rule of inference would assert that $C_1 \| C_2 \models P_1 \wedge P_2$ if
  1. $C_1 \models R_2 \implies P_1$
  2. $C_2 \models R_1 \implies P_2$
  3. $P_1 \models R_1$
  4. $P_2 \models R_2$
- The rule is circular (and unsound): Consider the case when $P_1 = P_2 = R_1 = R_2 =$ **false**.
- Sound versions of the proof allow $R \stackrel{+}{\implies} P$ for *safety properties* when R fails before P does.

Compositional Model Checking

- McMillan defines a compositional proof rule for reducing $M_1 \| M_2 \models \Box(P_1 \wedge P_2)$ by showing
  1. $M_1 \models P_2 \rhd P_1$, and
  2. $M_2 \models P_1 \rhd P_2$
- Here, $P \rhd Q$ means that P fails before Q or $\neg(P \bigcup \neg q)$.
- Also, composition $M_1 \| M_2$ is just the conjunction of the transition relations.
- In general, the composition operation can depend on the model of computation: e.g., synchronous, asynchronous, dataflow, message-passing, shared-variable communication, etc.

SAL Example: Peterson

```
mutex[tval: bool; STATE turn : bool]    : MODULE
  BEGIN
   pc : STATE {sleep, try, critical}
   try: RULE pc = sleep ==>
               {pc' = try;
                turn' = tval}
   critical: RULE pc = try AND turn /= tval ==>
                {pc' = critical}
   exit: RULE pc = critical ==>
               {pc' = sleep;
                turn' = tval}
   mutexproc: PROCESS
         INITIALLY {pc = sleeping}
         NEXT try || exit || critical
END mutex
```

SAL Example: Peterson

```
peterson: MODULE
 BEGIN
  turn : STATE bool

  mutex1: MODULE = mutex[TRUE, turn]
  mutex2: MODULE = mutex[FALSE, turn]

  proc: PROCESS =
    mutex1.mutexproc || mutex2.mutexproc

  exclusive: ASSERTION = NOT (mutex1.pc = critical AND
                              mutex2.pc = critical)

  mutual_exclusion: LEMMA proc |= AG(exclusive)

END peterson
```

Questions for Discussion

- What exactly is a module?
- Are modules primarily a design time aid for reusing definitions and theorems, or do they have some first-class status in the computation itself?
- Can we usefully modularize knowledge? What language+design principles do we need?
- Can we usefully modularize in-the-small software design?
- Are the composition mechanisms for decomposing designs more critical than the modules themselves?
- *Think outside the module.*