What is a Module?

N. Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

Sep 4, 2023 (Tetrapod workshop, Cambridge UK)

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 a number of incarnations but no precise definition
- A similar vagueness exists with respect to *process*, *class*, *object*, *method*.
- What is a module?
- 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.

What is a module?

- The dictionary definition might characterize a module as "a self-contained unit or component that is part of a larger assembly."
- For example, a lunar module, a course module, or a software module.
- A module is meant to be
  1. *Reusable* within the same system and in other systems/contexts.
  2. *Changeable*, so that it can be independently debugged/repaired/modified/adapted/improved
  3. *Interchangeable*, so that one module can be replaced with another that provides equivalent or better functionality.
- From *On-Line Data-Acquisition Systems in Nuclear Physics, 1969*, by H. W. Fulbright et al. and National Research Council (`https://www.gutenberg.org/files/42613/42613-h/42613-h.htm`)

  Debugging of that unit was exceedingly laborious because of the lack of modularity in its components.

The Art of the Module

- Defining a module as a unit of reuse, change, and exchange, does not solve the problem.
- Designing a 'good' module involves:
  1. Maximizing encapsulated functionality while minimizing the interface, e.g., a compiler.
  2. Maximizing versatility without overloading the interface or dragging along unused functionality.
  3. Maximizing locality so that the module can be independently debugged and modified while minimizing duplication.
  4. Maximizing adaptability so that the module can be reused in a range of contexts.
  5. Maximizing abstraction without compromising efficiency.

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 abstract 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.
Compositional Design: Systems can be designed to have properties by composing component and subsystem properties.

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.
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 arises from composing big components with small interfaces.
Berry Principle: Write everything (at most) once. (Predates Berry. See Wikipedia: *Abstraction Principle*.
Corollary: Prove everything (at most) once.

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
- Modules interact through side-channels

Often, 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 are 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 (for parametric theories), extended, combined, cloned, and interpreted.

Theory Interpretations

- Theories can be imported with or without explicit parameters.
- Theories can also be interpreted by assigning interpretations to uninterpreted symbols.

```
group_homomorphism[G1, G2: THEORY group]: THEORY
 BEGIN
  x, y: VAR G1.G
  f: VAR [G1.G -> G2.G]
  homomorphism?(f): bool = FORALL x, y: f(x + y) = f(x) + f(y)
  hom_exists: LEMMA EXISTS f: homomorphism?(f)
 END group_homomorphism
```

```
  IMPORTING
   group_homomorphism[group{{G := int, + := +, 0 := 0, - := -}},
                      group{{G := nzreal, + := *, 0 := 1,
                             - := LAMBDA (x: nzreal): 1/x}}]
```

Context is Everything

- A module can be seen as a unit of composition for some composition operator $\|$.
- Process algebras study the composition of processes through such composition operators.
- Such generic composition operators are semantically weak and offer very little design guidance.
- Composition frameworks or architectures that mediate between components enhance the modularity of a system.
- A good composition framework offers components an interface through which they can interoperate with other components
  1. Composability: Properties of well-behaved components are preserved in the composition
  2. Compositionality: System properties are composed from component properties
  3. Monotonicity: Components can be independently refined — better component properties yield better system properties.
- See Peter G. Neumann (2004). 'Principled Assuredly Trustworthy Composable Architectures'.

Compositional Frameworks in the Real World

- **Currency:** The exchange of goods and services is mediated by the use of currency. The alternative, barter, is highly non-compositional.
- **Stock Market:** It makes the value of stocks public without revealing the identities of the buyers and sellers.
- **Traffic:** The system of lanes and signals makes it possible for multiple vehicles to share the roads with minimal interaction between vehicles.
- **EBay:** Creates a market for the exchange of goods to the mutual satisfaction of buyer and seller.
- *The key in these systems is that each component interacts with other components through the framework.*

Compositional Frameworks in Computation

- **Schedulers:** It ensures that processes get allocated CPU time and that their state is maintained between context switches.
- **Virtual Memory:** Allows processes to share physical memory without conflict.
- **Database Concurrency Control:** Allows multiple database transactions to operate simultaneously while guaranteeing serializability.
- **Separation Kernel:** Allows multiple processes to inter-operate without covert channels.
- **Email:** Allows transmission of electronic mail independent of the physical constraints of medium and location.
- **Time-triggered Architecture:** Allows a bus to be synchronously shared by multiple nodes with real-time guarantees.
- *Each of these frameworks is scalable with respect to the number and quality of the components.*

The Radler Architecture Definition Language

- Radler is a model of computation and interaction for real-time, distributed, sense-control-actuate systems.
- A Radler architecture consists of a fixed set of nodes and channels interacting through a publish/subscribe regime.
- Each node executes at an approximately specified period (quasi-periodicity).
- In each round of execution, the node reads its subscription mailboxes, executes a step function, and writes to its published mailboxes.
- Each topic has a message type and a single publisher node.
- Each publish/subscribe channel has a upper latency bound.
- Each publish/subscribe mailbox has a buffer width bound.
- Critical safety and security properties can be ensured at the architectural level independent of the components.
- *Yields designs with efficient arguments — any flaws in the argument must be easy to find with a low amortized cost.*

Radler Architecture

Requirements:
Maintain room temperature between min and max.

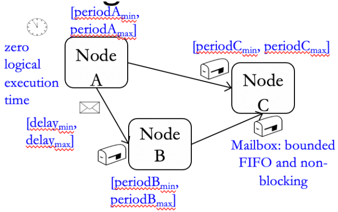
Assumptions:
Leakage rate, heater, sensor accuracy.

Logical Radler Architecture:
Sensor + Controller + Console + Safety Monitor
Channel Latencies

Radler logical architecture guarantees
- Message ordering
- Bounded/zero message loss
- End-to-end latency bounds
- Failure warnings
- No DoS attacks
- Partitioning

Physical architecture:
Machines, VMs, OS, Transport, Configuration

Code Components



- Assumptions + Architecture => Requirements
- Architecture = Nodes + Channels + Timing
- Nodes = Step function contracts
- Physical Architecture => Architecture
- Code => Step function contracts + WCET bounds

Questions for Discussion

- What exactly is a module? A namespace, a unit of specification/composition/reuse/separate compilation, a mathematical concept, an engineering convenience?
- 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.