Deadlock-Free Typestate-Oriented Programming

Luca Padovani

University of Torino, Italy

outline

Introduction to TypeState-Oriented Programming

Typestate: A Programming Language Concept for Enhancing Software Reliability

ROBERT E. STROM AND SHAULA YEMINI

Abstract—We introduce a new programming language concept called *typestate*, which is a refinement of the concept of *type*. Whereas the type of a data object determines the set of operations *ever* permitted on the object, typestate determines the subset of these operations which is permitted in a particular context.

Typestate tracking is a program analysis technique which enhances program reliability by detecting at compile-time syntactically legal but semantically undefined execution sequences. These include, for example, reading a variable before it has been initialized, dereferencing a pointer after the dynamic object has been deallocated, etc. Typestate tracking detects errors that cannot be detected by type checking or by conventional static scope rules. Additionally, typestate tracking makes it possible for compilers to insert appropriate finalization of data at exception points and on program termination, eliminating the need to support finalization by means of either garbage collection or unsafe deallocation operations such as Pascal's dispose operation.

By enforcing typestate invariants at compile-time, it becomes practical to implement a "secure language"—that is, one in which all successfully compiled program modules have fully defined execution-time effects, and the only effects of program errors are incorrect output values.

This paper defines typestate, gives examples of its application, and shows how typestate checking may be embedded into a compiler. We discuss the consequences of typestate checking for software reliability and software structure, and conclude with a discussion of our experience using a high-level language incorporating typestate checking.

scope checking avoid some but not all nonsense. In Section II, we informally present the typestate concept, give examples of its use, and discuss the benefits which accrue from compile-time tracking of typestate. In Section III, we give a more formal definition of typestate, and present an algorithm for verifying the typestate consistency of programs. In Section IV, we discuss the interaction between typestate and other language design issues, such as composite user-defined types, independent compilation, and aliasing. We discuss our experience as designers and users of NIL—a secure programming language incorporating compile-time typestate tracking. Section V presents some conclusions and comparisons with related work.

A. Type Checking

From the perspective of software reliability, one of the most important properties of the concept of type is that it supports the automatic detection of certain kinds of errors.

The *type* of a variable name determines the set of operations which may be applied to that variable. For instance, if X is of type **real** it is allowed to appear in the context

from types to typestate [Strom and Yemini, 1986]

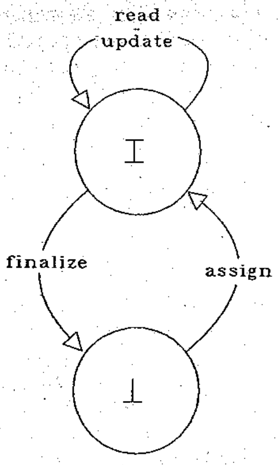


Fig. 1. Typestate transition graph for type **integer:** the scalar type integer illustrates the simplest nontrivial typestate transition graph. There are two typestates: ⊥ (intuitively "uninitialized") and I ("intuitively initialized").

typestate for objects [DeLine and Fähndrich, 2004]

- File (cannot read if closed)
- TCP socket (cannot send if disconnected)
- Stack (cannot pop if empty)
- Bounded buffer (cannot put if full)
- …

"…approximately ***7.2%*** *of all types defined protocols, while* ***13%*** *of classes were clients of types defining protocols."*
[Beckman et al., 2011]

typestate-oriented programming [Aldrich et al., 2009]

```
class Buffer { }

state Empty of Buffer {
  public void put(int x) {       [Empty >> Full]
    this ← Full { this.value = x; }
} }

state Full of Buffer {
  private int value;
  public int get() {             [Full >> Empty]
    int v = this.value;
    this ← Empty {}
    return v;
} }
```

typestate-oriented programming: wrap-up

Key mechanisms

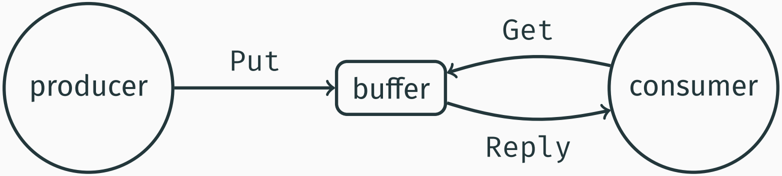
- pairing types with **states** `Empty`, `Full`
- decorating methods with **state transitions** `Empty >> Full`
- controlling **object aliasing** linearity

Well-typed programs “don’t go wrong”

- no **unavailable** method is ever invoked on any object
- well-typed programs either **reduce** or **successfully terminate**

From sequential to concurrent TSOP

a concurrent buffer



Remarks

- producer doesn't know when the buffer is empty
- consumer doesn't know when the buffer is full

Consequences

- pointless to require an order on the invocations of `Get`/`Put`
- an invocation of `Get`/`Put` may suspend the caller
- sensible to require that there's the same number of `Get`/`Put`

protocol conformance does not imply deadlock freedom

- `buffer!Put(42) | buffer!Get`

protocol conformance does not imply deadlock freedom

- `buffer!Put(42) | buffer!Get` ☺
- `buffer!Get`

protocol conformance does not imply deadlock freedom

- `buffer!Put(42) | buffer!Get` ☺
- `buffer!Get` ☺
- `buffer!Put(42) | buffer!Put(43)`

protocol conformance does not imply deadlock freedom

- `buffer!Put(42) | buffer!Get` ☺
- `buffer!Get` ☺
- `buffer!Put(42) | buffer!Put(43)` ☺
- `buffer!Put(buffer.Get)`

protocol conformance does not imply deadlock freedom

- `buffer!Put(42) | buffer!Get` ☺
- `buffer!Get` ☺
- `buffer!Put(42) | buffer!Put(43)` ☺
- `buffer!Put(buffer.Get)` ☹

Well-typed programs “don’t go wrong”* [Crafa and Padovani, 2017]

protocol conformance does not imply deadlock freedom

- `buffer!Put(42) | buffer!Get` ☺
- `buffer!Get` ☺
- `buffer!Put(42) | buffer!Put(43)` ☺
- `buffer!Put(buffer.Get)` ☹

Well-typed programs “don’t go wrong”* [Crafa and Padovani, 2017]

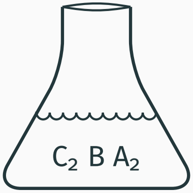
*...**but it may be the case that they don’t go at all**

A model for concurrent TSOP

the chemical abstract machine [Berry and Boudol, 1992]

state change = chemical reaction

$$A \ \& \ B \ \& \ C \quad \triangleright \quad D \ \& \ E$$

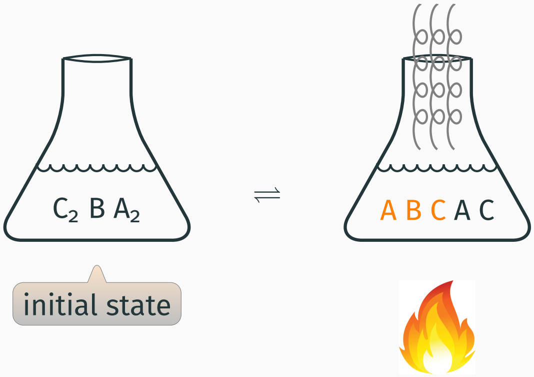


initial state

the chemical abstract machine [Berry and Boudol, 1992]

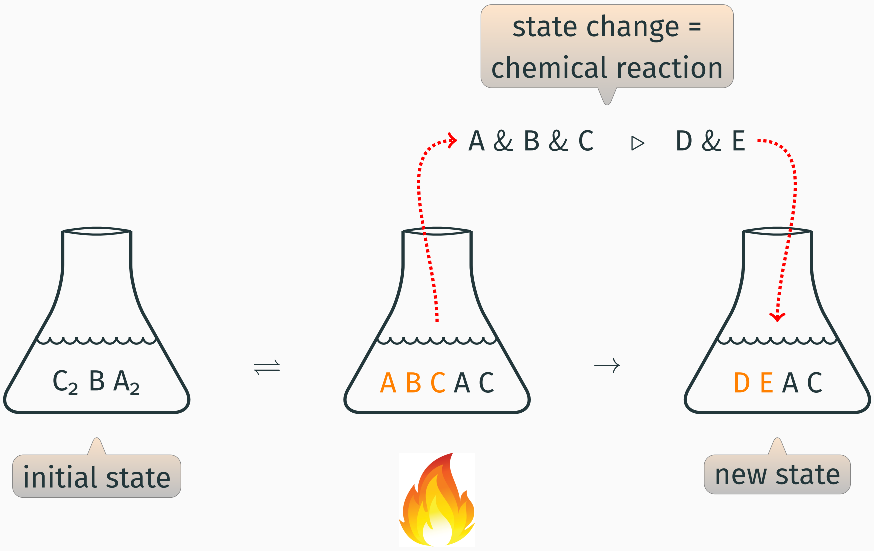
state change = chemical reaction

$A \,\&\, B \,\&\, C \quad \triangleright \quad D \,\&\, E$



initial state

the chemical abstract machine [Berry and Boudol, 1992]



the Objective Join Calculus [Fournet et al., 2003]

```
new buffer =
  EMPTY   & Put(x) ▷ buffer!FULL(x)
  FULL(x) & Get(u) ▷ buffer!EMPTY() & u!Reply(x)
```

A natural model for concurrent TSOP [Crafa and Padovani, 2017]

- concurrent objects
- patterns $\Rightarrow$ pairing of states and operations
- patterns $\Rightarrow$ synchronization

Preventing deadlocks

buffer!Put(buffer.Get) in the Objective Join Calculus

```
new buffer : (EMPTY + FULL)·*(Put·Get) =
  EMPTY    & Put(x) ▷ buffer!FULL(x)
  FULL(x) & Get(c) ▷ buffer!EMPTY() & c!Reply(x)

buffer!EMPTY &

new cont : CLOSURE·Reply =
  CLOSURE(buffer) & Reply(x) ▷ buffer!Put(x)

cont!CLOSURE(buffer) & buffer!Get(cont)
```

buffer!Put(buffer.Get) in the Objective Join Calculus

```
new buffer : (EMPTY + FULL)·*(Put·Get) =
  EMPTY    & Put(x) ▷ buffer!FULL(x)
  FULL(x) & Get(c) ▷ buffer!EMPTY() & c!Reply(x)

buffer!EMPTY &

new cont : CLOSURE·Reply =
  CLOSURE(buffer) & Reply(x) ▷ buffer!Put(x)

cont!CLOSURE(buffer) & buffer!Get(cont)
```

CLOSURE

buffer!Put(buffer.Get) in the Objective Join Calculus

```
new buffer : (EMPTY + FULL)·*(Put·Get) =
  EMPTY   & Put(x) ▷ buffer!FULL(x)
  FULL(x) & Get(c) ▷ buffer!EMPTY() & c!Reply(x)

buffer!EMPTY &

new cont : CLOSURE·Reply =
  CLOSURE(buffer) & Reply(x) ▷ buffer!Put(x)

cont!CLOSURE(buffer) & buffer!Get(cont)
```

CLOSURE

Reply

buffer!Put(buffer.Get) in the Objective Join Calculus

```
new buffer : (EMPTY + FULL)·*(Put·Get) =
  EMPTY   & Put(x) ▷ buffer!FULL(x)
  FULL(x) & Get(c) ▷ buffer!EMPTY() & c!Reply(x)
```

EMPTY

```
buffer!EMPTY &

new cont : CLOSURE·Reply =
  CLOSURE(buffer) & Reply(x) ▷ buffer!Put(x)

cont!CLOSURE(buffer) & buffer!Get(cont)
```

buffer!Put(buffer.Get) in the Objective Join Calculus

```
new buffer : (EMPTY + FULL)·*(Put·Get) =
  EMPTY   & Put(x) ▷ buffer!FULL(x)
  FULL(x) & Get(c) ▷ buffer!EMPTY() & c!Reply(x)

buffer!EMPTY &

new cont : CLOSURE·Reply =
  CLOSURE(buffer) & Reply(x) ▷ buffer!Put(x)

cont!CLOSURE(buffer) & buffer!Get(cont)
```

EMPTY

Put

buffer!Put(buffer.Get) in the Objective Join Calculus

```
new buffer : (EMPTY + FULL)·*(Put·Get) =
  EMPTY   & Put(x) ▷ buffer!FULL(x)
  FULL(x) & Get(c) ▷ buffer!EMPTY() & c!Reply(x)
EMPTY

buffer!EMPTY &

new cont : CLOSURE·Reply =
  CLOSURE(buffer) & Reply(x) ▷ buffer!Put(x)

cont!CLOSURE(buffer) & buffer!Get(cont)
                 Put          Get
```

strategy

Definition (object dependency)
A dependency between u and v is established if v is the argument of a message targeted to u (or vice versa)

Type system

1. enforce protocol conformance
2. track dependencies between objects
3. make sure the dependency graph is acyclic

a glimpse at some typing rules

A typing judgment $\Gamma \vdash P \bullet \mathfrak{D}$ reads as:

- process P **conforms** with the types (Γ) of the objects it uses
- and **establishes** the dependencies $\mathfrak{D}$ among such objects

where

- Γ maps object names to types
- $\mathfrak{D}$ is an **irreflexive** dependency relation

$$\text{[T-SEND]}\ \frac{}{u : \mathtt{m}(t), v : t \vdash u!\mathtt{m}(v) \bullet u \sim v}$$

parallel composition

$$\frac{\Gamma_1 \vdash P_1 \bullet \mathfrak{D}_1 \qquad \Gamma_2 \vdash P_2 \bullet \mathfrak{D}_2}{\Gamma_1 \cdot \Gamma_2 \vdash P_1 \mathbin{\&} P_2 \bullet (\mathfrak{D}_1 \cup \mathfrak{D}_2)^+}\ \textsc{[t-par]} \qquad \begin{array}{l} \mathfrak{D}_1 \cap \mathfrak{D}_2 = \emptyset \\ (\mathfrak{D}_1 \cup \mathfrak{D}_2)^+ \text{ irreflexive} \end{array}$$

Motivating examples

```
user!CLOSURE(buffer) & buffer!Get(user)

a!M(b) & b!M(c) & c!M(a)
```

reactions and classes

$$\frac{\overline{x : t} \vdash J \qquad \overline{x : t} \vdash P \bullet \mathfrak{D}}{\vdash J \rhd P}\ \text{[T-REACTION]}$$

reactions and classes

not in the conclusion

$$\frac{\overline{x : t} \vdash J \qquad \overline{x : t} \vdash P \bullet \mathfrak{D}}{\vdash J \triangleright P}\;[\textsc{t-reaction}]$$

Remarks

- dependencies are **confined** within classes
- classes can be type checked **independently**

properties of well-typed processes

Theorem
If $\emptyset \vdash P \bullet \mathfrak{D}$, then:

1. *P is protocol **conformant**, and*
2. *either P **reduces** (to a well-typed process) or P is **successfully terminated***

Remark

- the notion of “successfully terminated process” depends on the type of the objects it uses $\Rightarrow$ see paper for details

Conclusions

concluding remarks

This work

- closes gap between sequential and concurrent TSOP
- first type system for deadlock-freedom in OJC

In the paper

- formal definitions and proofs
- more interesting examples
  - sieve of Eratosthenes
  - Gregory-Leibniz approximation of π

Proof-of-concept implementation

- `www.di.unito.it/~padovani/Software/CobaltBlue`

References

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of OOPSLA'09*, pages 1015–1022. ACM, 2009.

Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *Proceedings of ECOOP'11*, volume LNCS 6813, pages 2–26. Springer, 2011.

Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. *Theoretical Compututer Science*, 96(1):217–248, 1992.

Silvia Crafa and Luca Padovani. The Chemical Approach to Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 39: 13:1–13:45, 2017.

Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proceedings of ECOOP'04*, LNCS 3086, pages 465–490. Springer, 2004.

Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Inheritance in the Join Calculus. *Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003.

Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12 (1):157–171, 1986.