the chemical approach to typestate-oriented programming

Silvia Crafa[1] Luca Padovani[2]

[1]Dipartimento di Matematica, Università di Padova, Italy

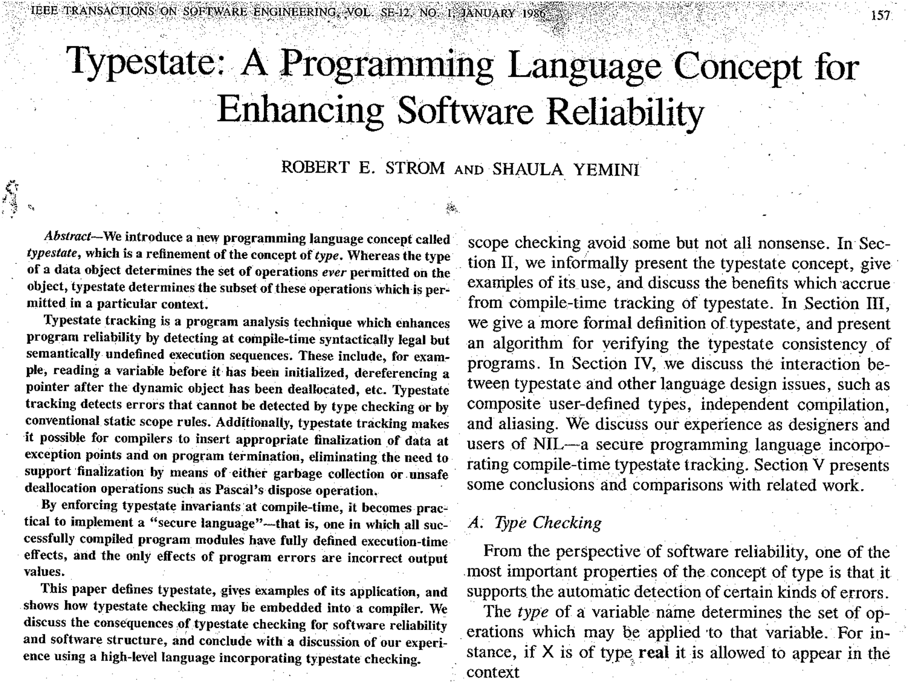
[2]Dipartimento di Informatica, Università di Torino, Italy

Novi Sad, 15 November 2016

Outline

Outline

1. The evolution of typestate
2. Chemistry and computing
3. Typestate and concurrency
4. Concurrent object protocols
5. Concluding remarks



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-12, NO. 1, JANUARY 1986 157

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

typestate = type + behavior (Strom & Yemini '86)

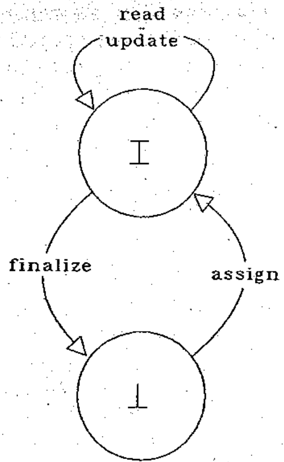


Fig. 1. Typestate transition graph for type **integer**: the scalar type integer illustrates the simplest nontrivial typestate transition graph. There are two typestates: $\perp$ (intuitively "uninitialized") and $\top$ ("intuitively initialized").

typestate for objects (DeLine & Fähndrich ’04, Microsoft)

```
[ TypeStates("Raw", "Bound", "Connected", "Closed") ]
class Socket {

  [ Post("Raw"), NotAliased ]
  Socket();

  [ Pre("Raw"), Post("Bound"), NotAliased ]
  void Bind(string endpoint);

  [ Pre("Bound"), Post("Connected"), NotAliased ]
  void Connect();

  [ Pre("Connected") ]
  void Send(string data);

  [ Pre("Connected") ]
  string Receive();

  [ Pre("Connected"), Post("Closed"), NotAliased ]
  void Close();
}
```

typestate-oriented programming (Aldrich *et al.* '09)

```
class Cell { }

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

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

typestate-oriented programming: summary

Objective

- **static** enforcement of object protocols

Mechanisms

- **abstract state** annotations in types `Empty`, `Full`
- tracking of **state transitions** `[Empty >> Full]`
- **aliasing control** **NotAliased**

Does this framework scale to **concurrent** objects?

- concurrent objects are aliased by definition !
- state transitions aren't always statically trackable !
- **. . . let's rewind time to the early 90s**

typestate-oriented programming: summary

Objective

- **static** enforcement of object protocols

Mechanisms

- **abstract state** annotations in types `Empty`, `Full`
- tracking of **state transitions** `[Empty >> Full]`
- **aliasing control** **NotAliased**

Does this framework scale to **concurrent** objects?

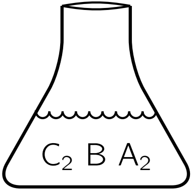
- concurrent objects are aliased by definition !
- state transitions aren't always statically trackable !
- **. . . let's rewind time to the early 90s**

Outline

the chemical abstract machine (Berry & Boudol ’92)

state change = chemical reaction

$$\mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \quad \triangleright \quad \mathsf{D} \mid \mathsf{E}$$

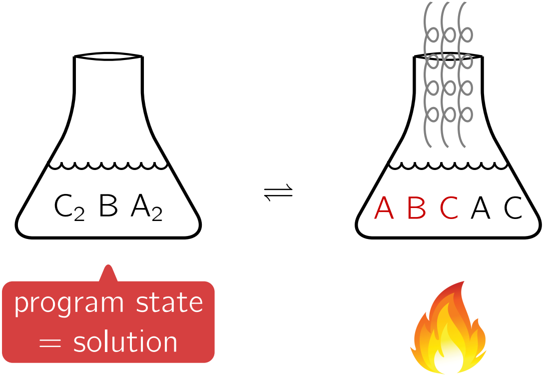


program state = solution

the chemical abstract machine (Berry & Boudol '92)

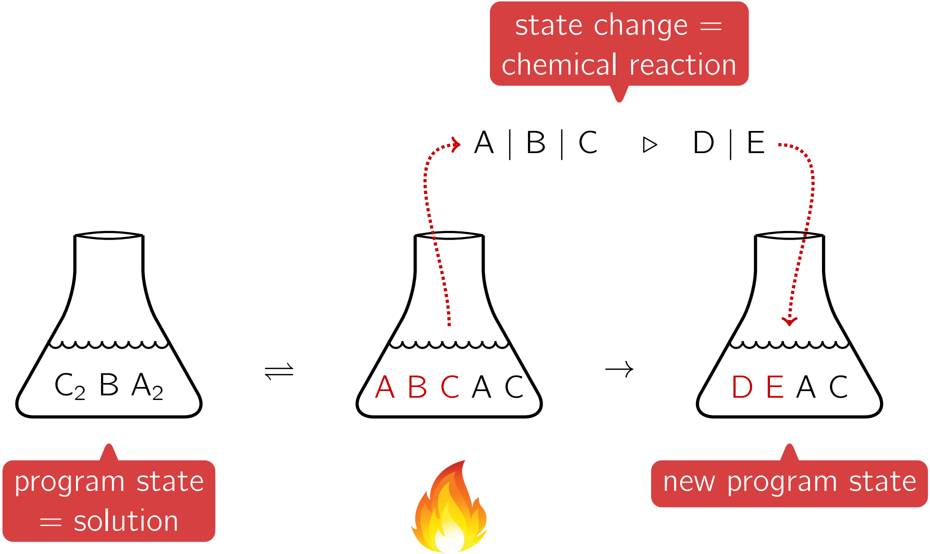
state change = chemical reaction

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



program state = solution

the chemical abstract machine (Berry & Boudol '92)



state change = chemical reaction

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

$C_2\ B\ A_2 \quad \rightleftharpoons \quad A\ B\ C\ A\ C \quad \rightarrow \quad D\ E\ A\ C$

program state = solution

new program state

join calculus = reflexive CHAM (Fournet & Gonthier '94)

```
def EMPTY() | put(x) ▷ FULL(x)
 or FULL(x) | get(c) ▷ EMPTY() | c(x)

def continue(x)      ▷ put(x + 1)

put(0) | EMPTY() | get(continue)
```

A formal model of **communicating processes**

- name $\iff$ **channel**
- **high-level**, **easy-to-implement** alternative to π-calculus

objective join calculus (Fournet, Laneve, Maranget, Rémy ’03)

```
def cell = EMPTY() | put(x) ▷ cell.FULL(x)
 or        FULL(x) | get(u) ▷ cell.EMPTY() | u.reply(x)

def user = reply(x)         ▷ cell.put(x + 1)

cell.put(0) | cell.EMPTY() | cell.get(user)
```

A formal model of **concurrent objects**

- name ⟺ **object**
- message ⟺ **method**
- interactions between **inheritance** and **concurrency**

analogies

Plaid

```
class Cell { }

state Empty of Cell {
  public void put(int x) {
    this ← Full { val = x; }
} }

state Full of Cell {
  private int val;
  public int get() {
    int x = val;
    this ← Empty {}
    return x;
} }
```

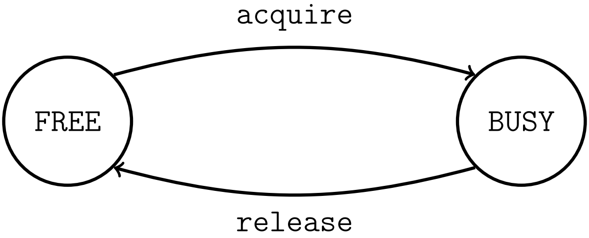
Objective Join Calculus

```
EMPTY() | put(x) ▷
  cell.FULL(x)

FULL(x) | get(r) ▷

  cell.EMPTY()
| r.reply(x)
```

Outline

1. The evolution of typestate
2. Chemistry and computing
3. **Typestate and concurrency**
4. Concurrent object protocols
5. Concluding remarks

a simple concurrent object: the **lock**



There's an asymmetry between `acquire` and `release`

- after an `acquire` **we know** that the lock is `BUSY`
- after a `release` **we don't know** when the lock is `FREE` again

competing for a lock

```
def o = FREE | acquire(u) ▷ o.BUSY | u.reply(o)
 or     BUSY | release    ▷ o.FREE
in ...
```

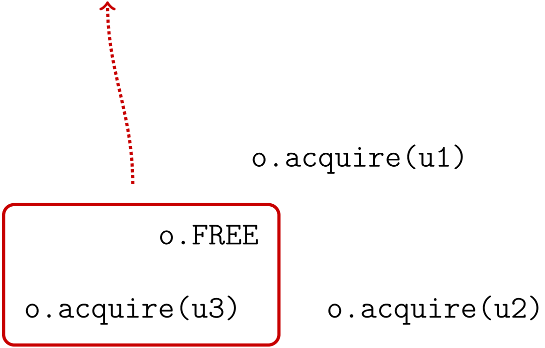
o.acquire(u1)

o.FREE

o.acquire(u3)   o.acquire(u2)

competing for a lock

```
def o = FREE | acquire(u) ▷ o.BUSY | u.reply(o)
 or     BUSY | release    ▷ o.FREE
in ...
```



o.acquire(u1)

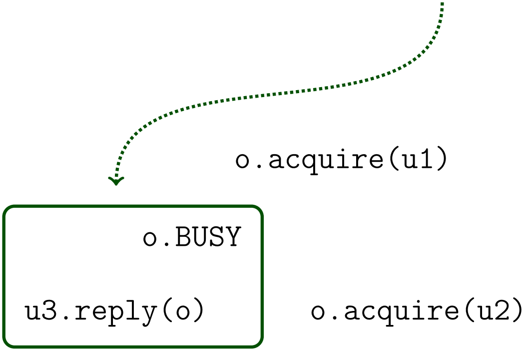
o.FREE

o.acquire(u3)

o.acquire(u2)

competing for a lock

```
def o = FREE | acquire(u) ▷ o.BUSY | u.reply(o)
 or     BUSY | release    ▷ o.FREE
in ...
```



o.acquire(u1)

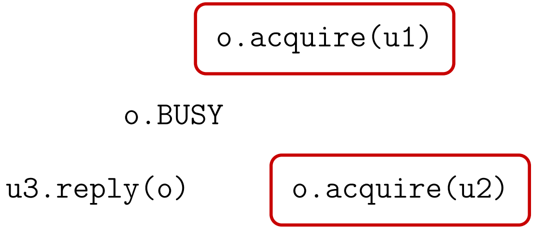
o.BUSY

u3.reply(o)

o.acquire(u2)

competing for a lock

```
def o = FREE | acquire(u) ▷ o.BUSY | u.reply(o)
 or     BUSY | release    ▷ o.FREE
in ...
```



o.acquire(u1)

o.BUSY

u3.reply(o)

o.acquire(u2)

competing for a lock

```
def o = FREE | acquire(u) ▷ o.BUSY | u.reply(o)
 or     BUSY | release    ▷ o.FREE
in ...
```

o.acquire(u1)

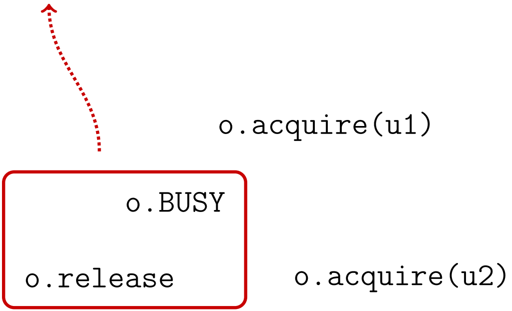
o.BUSY

u3.reply(o)

o.acquire(u2)

competing for a lock

```
def o = FREE | acquire(u) ▷ o.BUSY | u.reply(o)
 or     BUSY | release    ▷ o.FREE
in ...
```



o.acquire(u1)

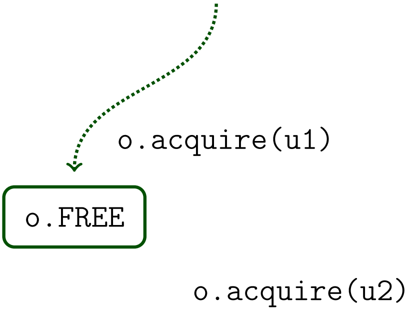
o.BUSY

o.release

o.acquire(u2)

competing for a lock

```
def o = FREE | acquire(u) ▷ o.BUSY | u.reply(o)
 or     BUSY | release    ▷ o.FREE
in ...
```



Objective Join Calculus

- formal model of concurrent objects
- explicit association of state and operations ⇒ **TSOP**
- runtime synchronization mechanism ⇒ **concurrency**
- **is this all we need?**

let's play a game

`FREE`

`acquire`

let's play a game

`FREE`

`acquire`

`acquire`

let's play a game

`acquire`

`FREE`

`acquire`

`acquire`

let's play a game

`release`

`acquire`

`acquire`

`BUSY`

`acquire`

let's play a game

`release`

`acquire`

`acquire`

`BUSY`

`release`

`acquire`

let's play a game

`acquire`

`FREE`

`acquire`

`release`

`acquire`

let's play a game

acquire

FREE

acquire

BUSY

acquire

Outline

1. The evolution of typestate
2. Chemistry and computing
3. Typestate and concurrency
4. **Concurrent object protocols**
5. Concluding remarks

lock protocol

- the lock is always either `IDLE` or `BUSY`
- there can be any number of `acquire`, regardless of state
- when the lock is `BUSY`, it must be released once

$$*\texttt{acquire} \otimes (\texttt{IDLE} \oplus (\texttt{BUSY} \otimes \texttt{release}))$$

cell protocol

- the cell is always either `EMPTY` or `FULL`
- when the cell is `EMPTY`, it is *possible* to `put` an element in it
- when the cell is `FULL`, it is *necessary* to `get` the element in it

$$(\mathtt{EMPTY} \otimes (\mathtt{put} \oplus \mathbb{1})) \oplus (\mathtt{FULL} \otimes \mathtt{get})$$

semantics of types

- commutative Kleene algebra

$$\mathtt{a} \otimes \mathtt{b} \simeq \mathtt{b} \otimes \mathtt{a} \qquad \mathtt{a} \otimes (\mathtt{b} \oplus \mathtt{c}) \simeq (\mathtt{a} \otimes \mathtt{b}) \oplus (\mathtt{a} \otimes \mathtt{c}) \qquad \cdots$$

- subtyping = inverse language inclusion

$$\mathtt{a} \oplus \mathtt{b} \leq \mathtt{a} \qquad \mathtt{a} \otimes \mathtt{b} \not\leq \mathtt{a} \qquad \mathtt{a} \not\leq \mathtt{a} \otimes \mathtt{b}$$

- valid subtyping relation $\iff$ valid Presburger formula

semantics of types

- commutative Kleene algebra

$$\mathtt{a} \otimes \mathtt{b} \simeq \mathtt{b} \otimes \mathtt{a} \qquad \mathtt{a} \otimes (\mathtt{b} \oplus \mathtt{c}) \simeq (\mathtt{a} \otimes \mathtt{b}) \oplus (\mathtt{a} \otimes \mathtt{c}) \qquad \cdots$$

- subtyping = inverse language inclusion

$$\mathtt{a} \oplus \mathtt{b} \leq \mathtt{a} \qquad \mathtt{a} \otimes \mathtt{b} \not\leq \mathtt{a} \qquad \mathtt{a} \not\leq \mathtt{a} \otimes \mathtt{b}$$

- valid subtyping relation $\iff$ valid Presburger formula

semantics of types

- commutative Kleene algebra

$$\mathtt{a} \otimes \mathtt{b} \simeq \mathtt{b} \otimes \mathtt{a} \qquad \mathtt{a} \otimes (\mathtt{b} \oplus \mathtt{c}) \simeq (\mathtt{a} \otimes \mathtt{b}) \oplus (\mathtt{a} \otimes \mathtt{c}) \qquad \cdots$$

- subtyping = inverse language inclusion

$$\mathtt{a} \oplus \mathtt{b} \leq \mathtt{a} \qquad \mathtt{a} \otimes \mathtt{b} \not\leq \mathtt{a} \qquad \mathtt{a} \not\leq \mathtt{a} \otimes \mathtt{b}$$

- valid subtyping relation $\iff$ valid Presburger formula

well-typed programs respect object protocols

Theorem (soundness)

If

- $\mathtt{o} : t \vdash P$, *and*
- P *sends* $\mathtt{m}_1 \cdots \mathtt{m}_k$ *to* $\mathtt{o}$,

then

- $\mathtt{m}_1 \cdots \mathtt{m}_k$ *is a valid message configuration according to* t.

If $\mathtt{o} : t_{\mathtt{lock}} \vdash P$

- P does not attempt to release the lock twice
- P does not attempt to release an unaquired lock
- ...

Outline

take-home messages

1. OJC is an **elegant formal model** of TSOP
    - TSOP = how you model objects in the OJC

2. OJC extends TSOP to **concurrency**
    - runtime synchronization mechanism

3. first **behavioral type theory** for OJC
    - type = set of valid message configurations

references

- Strom, Yemini, **Typestate: A Programming Language Concept for Enhancing Software Reliability**, Trans. Soft. Eng. 1986
- DeLine, Fähndrich, **Typestates for Objects**, ECOOP 2004
- Aldrich, Sunshine, Saini, Sparks, **Typestate-Oriented Programming**, OOPSLA 2009

- Berry, Boudol, **The Chemical Abstract Machine**, TCS 1992
- Fournet, Gonthier, **The Reflexive CHAM and the Join Calculus**, POPL 1994
- Fournet, Laneve, Maranget, Rémy, **Inheritance in the Join Calculus**, Journal of Log. and Algebr. Progr. 2003

- Crafa, Padovani, **The Chemical Approach to Typestate-Oriented Programming**, OOPSLA 2015