CS 428/528 Lecture 15: The P Framework for Communicating State Machines

March 5, 2024

Based on the PLDI13 paper by Desai et al
Communicating State Machines
**P Architecture**

- **P Program**
  - Implementation
  - Specification
  - Abstraction
  - Test

- **Compiler Toolchain**
  - Autogen C Impl.
  - Wrappers

- **P deployment tool**

- **P systematic testing tool**

- **Target Platform**
  - OS, libs
  - Autogen Impl.
  - Wrappers
  - Runtime

- (Reproducible) Error Trace
Examples for events on the input buffer. The initial state of the state, the following things happen: on entry to the state, the state as the entry statement. If the state machine enters the verifier, which gets executed in both these contexts. This subroutine also implements a liveness check that prevents deferring events in this state, it can nondeterministically raise the "event" violation. There are certain circumstances under which the event can be deferred. When trying to receive an event a machine has 3 states: Environment modeling.

(c) Timer ghost machine
P Examples

(a) User ghost machine

\begin{verbatim}
start → Init
  Deferred:
  Action:
    Elevator = new Elevator();
    raise(unit)

unit → Loop
  Deferred:
  Action:
    if * then
      send(Elevator, OpenDoor);
    else if * then
      send(Elevator, CloseDoor);
    raise(unit)

unit
\end{verbatim}
The machine is for events on the input buffer. The initial state of the machine is the entry statement. If the state machine enters the state has a single action handler, and an executed when the state is entered. For instance, the state transition shows the environment machine. On the entry statement, the state of this machine in-
We provide empirical evidence to demonstrate the benefits of our new delaying scheduler for programs, in which errors may be lurking in long executions. We use the example of an elevator, which is a classical synchronization problem, to illustrate our approach.

In summary, our contributions are the following:

- We present formal operational semantics and a compiler and interpreter for a new language that allows for delay-bounded scheduling. This enables the verification of protocols governing the interaction among concurrently executing components.
- We implement this approach and find it useful for discovering errors in the code.
- We design a DSL for specifying delay-bounded programs using delay-bounded scheduling. This DSL allows for a concise and readable specification of the desired behavior.

We present our implementation and show how it can be used to verify properties of the generated code. Our study is the USB stack in Windows 8.

Figure 1: Elevator example
which are types of data arguments that are sent along with the events, a nonempty list of machines, and one machine creation is responsible for creating an instance of the Timer, this is the starting point for a model checker to perform verification.

possibilities depends on environmental factors (such as timing),

### P Language Syntax

```
program ::= evdecl machine^+ m(init*)
machine ::= optghost machine m
vrdecl* actdecl* stdecl*
spdecl* cldecl* acdecl*

optghost ::= ε | ghost
evdecl ::= event edecl^+
vrdecl ::= optghost var vdecl^+
actdecl ::= action (a, stmt)^+
stdecl ::= state (n, {e1, e2, ..., ek}, stmt, stmt)^+
spdecl ::= step (n, e, n)^+
cldecl ::= call (n, e, n)^+
acdecl ::= act (n, e, a)^+
edecl ::= e(type)
vdecl ::= x : type

stmt ::= skip | x := expr | x := new m(init*)
delete | send(expr, e, expr)
raise (e, expr) | leave | return
assert(expr) | stmt; stmt
if expr then stmt else stmt | while expr stmt
init ::= x = expr
expr ::= this | msg | arg | b | c | ⊥ | x | *
| uop expr | expr bop expr

c ∈ int
b ∈ bool
¬, − ∈ uop
+, −, ∧, ∨ ∈ bop
r ∈ expr
a, e, m, x ∈ name
```

init

expr

the core language are triples of the from

A program in the core language consists of declaration of the identifier of the executing machine, and release its resources.

Primitive statements using standard control flow constructs:

- Transition declarations describe how a state responds to events.
- An action binding does not change the state, but merely executes the payload.
- Evaluation of the statement and returns to the caller.

When

Expr sends event to another state

Deferred event s

is expected. In the examples, we use

```
un
```

is a set of deferred events, together with arguments

```
iner
```

are two transitions then

```
act
```

The initializers give the initial values of the variables in the created machine. The

```
act
```

is executed.

```
expr bop expr
```

Variables can be declared in state

```
vdecl
```

denoted by

```
vdecl
```

addition to the declared variables, can also refer to three special

```
vdecl
```

expressions also include

```
vdecl
```

the initializers give the initial values of the variables in the created machine. The

```
vdecl
```

is executed.
configuration. A machine configuration corresponding to identifier \( id \) is of the form \((\gamma, \sigma, s, q)\) with components defined as follows:

- \( \gamma \) is a sequence of pairs \((n, \alpha)\), where \( n \) is a state name, and \( \alpha \) is map from events to \( A \cup \{\top, \bot\}\), where \( A \) is the set of all actions declared in machine \( Name(id) \). This sequence functions as a call stack, to implement call and return, and the \( \alpha \) values are used to inherit deferred events and actions from caller to callee. For an event \( e \), \( \alpha(e) \) can be an action \( a \), or the value \( \top \) indicating that the event is deferred, or the value \( \bot \) which indicates that the event does not have an associated action and it is not deferred.

- \( \sigma \) is a map from variables declared in machine \( Name(id) \) to their values; this map contains an entry for the local variables \texttt{this}, \texttt{msg} and \texttt{arg}.

- \( s \) is the statement remaining to be executed in machine \( id \).

- \( q \) is a sequence of pairs of a event-argument pairs representing the input buffer of machine \( id \).
state $n$ is executed whenever control leaves $n$. Given a machine name $m$ and a state $n$ in $m$, let $Deferred(m, n)$ denote the associated set of deferred events and let $Action(m, n, e)$ be an that action $a$ is associated with event $e$ in state $n$, if such a binding exists or $\bot$ otherwise. Let $Entry(m, n)$ denote the associated entry statement, and let $Exit(m, n)$ denote the associated exit statement. The initial state of the machine $m$ is the first state in the state list and is denoted by $Init(m)$. 
M[id] = (γ, σ, S[x := r], q)  σ(r) ↓ v \quad \text{(ASSIGN)}
M \rightarrow M[id := (γ, σ[x := v], S[\text{skip}], q)]

M[id] = (γ, σ, S[x := \text{new} m'(x_1 = r_1, x_2 = r_2, \ldots, x_n = r_n)], q)
\quad id' = \text{fresh}(m') \quad n' = \text{Init}(m')
\quad \alpha_o = \lambda e. \bot \quad \sigma(r_1) \downarrow v_1 \quad \sigma(r_2) \downarrow v_2 \quad \ldots \quad \sigma(r_n) \downarrow v_n
\quad \sigma' = \lambda x. \bot [\textbf{this} := id'][x_1 := v_1][x_2 := v_2] \cdots [x_n := v_n] \quad \text{(NEW)}
M \rightarrow M[id := (γ, σ[x := id'], S[\text{skip}], q)]
\quad [id' := ((n', \alpha_o), \sigma', \text{Entry}(m', n'), \varepsilon)]

\hline
M[id] = (γ, σ, S[\text{delete}], q) \quad \text{(DELETE)}
M \rightarrow M[id := \bot]

M[id] = (γ, σ, S[\text{assert}(r)], q) \quad \sigma(r) \downarrow \text{true} \quad \text{(ASSERT-PASS)}
M \rightarrow M[id := (γ, σ, S[\text{skip}], q)]

M[id] = (γ, σ, S[\text{skip}; s], q) \quad \text{(SEQ)}
M \rightarrow M[id := (γ, σ, S[s], q)]
P Language Semantics

\[
M[id] = (\gamma, \sigma, S[\text{if } r \text{ then } s_1 \text{ else } s_2], q), \quad \sigma(r) \downarrow \text{true} \quad \text{(IF-THEN)}
\]

\[
M \rightarrow M[id := (\gamma, \sigma, S[s_1], q)]
\]

\[
M[id] = (\gamma, \sigma, S[\text{if } r \text{ then } s_1 \text{ else } s_2], q), \quad \sigma(r) \downarrow \text{false} \quad \text{(IF-ELSE)}
\]

\[
M \rightarrow M[id := (\gamma, \sigma, S[s_2], q)]
\]

\[
M[id] = (\gamma, \sigma, S[\text{while } r \text{ s}], q), \quad \sigma(r) \downarrow \text{true} \quad \text{(WHILE-ITERATE)}
\]

\[
M \rightarrow M[id := (\gamma, \sigma, S[\text{while } r \text{ s}], q)]
\]

\[
M[id] = (\gamma, \sigma, S[\text{while } r \text{ s}], q), \quad \sigma(r) \downarrow \text{false} \quad \text{(WHILE-DONE)}
\]

\[
M \rightarrow M[id := (\gamma, \sigma, S[\text{skip}], q)]
\]

\[
M[id] = (\gamma, \sigma, S[\text{send}(r_1, e, r_2)], q)
\]

\[
\sigma(r_1) \downarrow id' \quad \sigma(r_2) \downarrow v \quad M[id'] = (\gamma', \sigma', C', q')
\]

\[
M \rightarrow M[id := (\gamma, \sigma, S[\text{skip}], q)][id' := (\gamma', \sigma', C', q' \odot (e, v))]
\]

\[
\text{(SEND)}
\]
\[ M[id] = ((n, \alpha) \cdot \gamma, \sigma, S[\text{raise} (e, r)], q) \]
\[
\begin{array}{c}
\sigma(r) \downarrow v \\
\sigma' = \sigma[\text{msg} := e][\text{arg} := v] \\
m = \text{Name}(id) \\
s = \begin{cases} 
\text{if } \text{Pop}(m, n, \alpha, e) \lor \text{Step}(m, n, e) \neq \bot \\
\text{then } \text{Exit}(m, n) \\
\text{else skip}
\end{cases}
\end{array}
\]
\[
M \rightarrow M[id := ((n, \alpha) \cdot \gamma, \sigma', s; \text{raise} (e, v), q)]
\]

\[ M[id] = (\gamma, \sigma, S[\text{leave}], q) \]
\[
M \rightarrow M[id := (\gamma, \sigma, \text{skip}, q)]
\]

\[ M[id] = (\gamma, \sigma, S[\text{return}], q) \]
\[
M \rightarrow M[id := (\gamma, \sigma, \text{Exit}(m, n); \text{return}, q)]
\]
\[ M[id] = ((n, \alpha) \cdot \gamma, \sigma, \text{skip}, q_1 \cdot (e, v) \cdot q_2) \\
m = \text{Name}(id) \\
t = \{ e | \text{Trans}(m, n, e) \neq \bot \lor \text{Action}(m, n, e) \neq \bot \} \\
d = \{ e | \alpha(e) = \top \} \\
d' = (d \cup \text{Deferred}(m, n)) - t \\
|q_1| \subseteq d' \quad e \notin d' \\
\sigma' = \sigma[\text{msg} := e][\text{arg} := v] \\
s = \begin{cases} 
\text{if } \text{Pop}(m, n, \alpha, e) \lor \text{Step}(m, n, e) \neq \bot \\
\text{then } \text{Exit}(m, n) \\
\text{else skip}
\end{cases}
\]
\[
M \rightarrow M[id := ((n, \alpha) \cdot \gamma, \sigma', s; \text{raise} (e, v), q_1 \cdot q_2)]
\]
P Language Semantics

\[
M[\text{id}] = ((n, \alpha) \cdot \gamma, \sigma, \text{raise} (e, v), q) \\
m = \text{Name(id)} \\
\text{Step} (m, n, e) = n' \\
M \rightarrow M[\text{id} := ((n', \alpha) \cdot \gamma, \sigma, \text{Entry}(m, n'), q)] \quad \text{(STEP)}
\]

\[
M[\text{id}] = ((n, \alpha) \cdot \gamma, \sigma, \text{raise} (e, v), q) \\
m = \text{Name(id)} \\
\text{Trans} (m, n, e) = \bot \\
(\alpha(e) = a \land \text{Action}(m, n, e) = \bot) \lor \text{Action}(m, n, e) = a \\
\alpha \notin \{\bot, \top\} \\
M \rightarrow M[\text{id} := ((n, \alpha) \cdot \gamma, \sigma, \text{Stmt}(m, a), q)] \quad \text{(ACTION)}
\]

\[
\alpha' = \lambda e. \\
\quad \text{if} (\text{Trans}(m, n, e) \neq \bot) \text{ then } \bot \\
\quad \text{else if} (\text{Action}(m, n, e) \neq \bot) \text{ then } \text{Action}(m, n, e) \\
\quad \text{else if} (e \in \text{Deferred}(m, n)) \text{ then } \top \\
\quad \text{else } \alpha(e) \\
M \rightarrow M[\text{id} := ((n', \alpha') \cdot (n, \alpha) \cdot \gamma, \sigma, \text{Entry}(m, n'), q)] \quad \text{(CALL)}
\]

\[
M[\text{id}] = ((n, \alpha) \cdot \gamma, \sigma, \text{raise} (e, v), q) \\
m = \text{Name(id)} \\
\text{Pop} (m, n, \alpha, e) \\
M \rightarrow M[\text{id} := (\gamma, \sigma, \text{raise} (e, v), q)] \quad \text{(POP1)}
\]

\[
M[\text{id}] = ((n, \alpha) \cdot \gamma, \sigma, \text{return}, q) \\
m = \text{Name(id)} \\
M \rightarrow M[\text{id} := (\gamma, \sigma, \text{skip}, q)] \quad \text{(POP2)}
\]

\[\text{Pop}(m, n, \alpha, e) = \text{Step}(m, n, e) = \bot \land \text{Call}(m, n, e) = \bot \land \text{Action}(m, n, e) = \bot \land \alpha(e) \in \{\bot, \top\}\]
P Language Semantics

\[
M[id] = (\gamma, \sigma, S[\texttt{assert}(r)], q) \quad \sigma(r) \downarrow \texttt{false} \\
\frac{\quad}{M \rightarrow \text{error}} \quad \text{(ASSERT-FAIL)}
\]

\[
M[id] = (\gamma, \sigma, S[\texttt{send}(r_1, e, r_2)], q) \quad \sigma(r_1) \downarrow \bot \\
\frac{\quad}{M \rightarrow \text{error}} \quad \text{(SEND-FAIL1)}
\]

\[
M[id] = (\gamma, \sigma, S[\texttt{send}(r_1, e, r_2)], q) \\
\sigma(r_1) \downarrow id' \quad M[id'] = \bot \\
\frac{\quad}{M \rightarrow \text{error}} \quad \text{(SEND-FAIL2)}
\]

\[
M[id] = (\varepsilon, \sigma, s, q) \\
\frac{\quad}{M \rightarrow \text{error}} \quad \text{(POP-FAIL)}
\]

Figure 6: Operational semantics: error transitions