S-expressions for Actions with Logic Temporal

David McNeil

September 2019

v6

TLA$^+$ Use

- TLA$^+$ use leads to TLA$^+$ advocacy
- Network effects
  - The ability to read specifications is more valuable if there are specs for more systems
  - Specifications are of greater value if more people can read them

TLA$^+$ Adoption

- Awareness
  - Know about the tool
  - Not silver bullet
- Ease of Use
  - Know the mechanics of using the tool
- Learned skill
  - Know how to use the tool well
  - “thinking above the code”

Ease of Use

- Provide developer affordances
- Play to developers skills and expectations
  - Without distracting from the fact that specs are not a way to write “code”

SALT

- S-expressions for Actions with Logic Temporal
- Express TLA$^+$ formulas as s-expressions
- Tools to process those s-expressions / formulas
  - Evaluate
  - Simplify
  - Simulate
  - Transpile

What do developers want?

Interactivity

- Enter a TLA$^+$ formula or expression
- See what it evaluates to
- Iterate

REPL

1 Read

2 Evaluate

3 Print

4 Loop (i.e. goto 1)

REPL

1 Read

2 Evaluate

3 Print

4 Loop (i.e. goto 1)

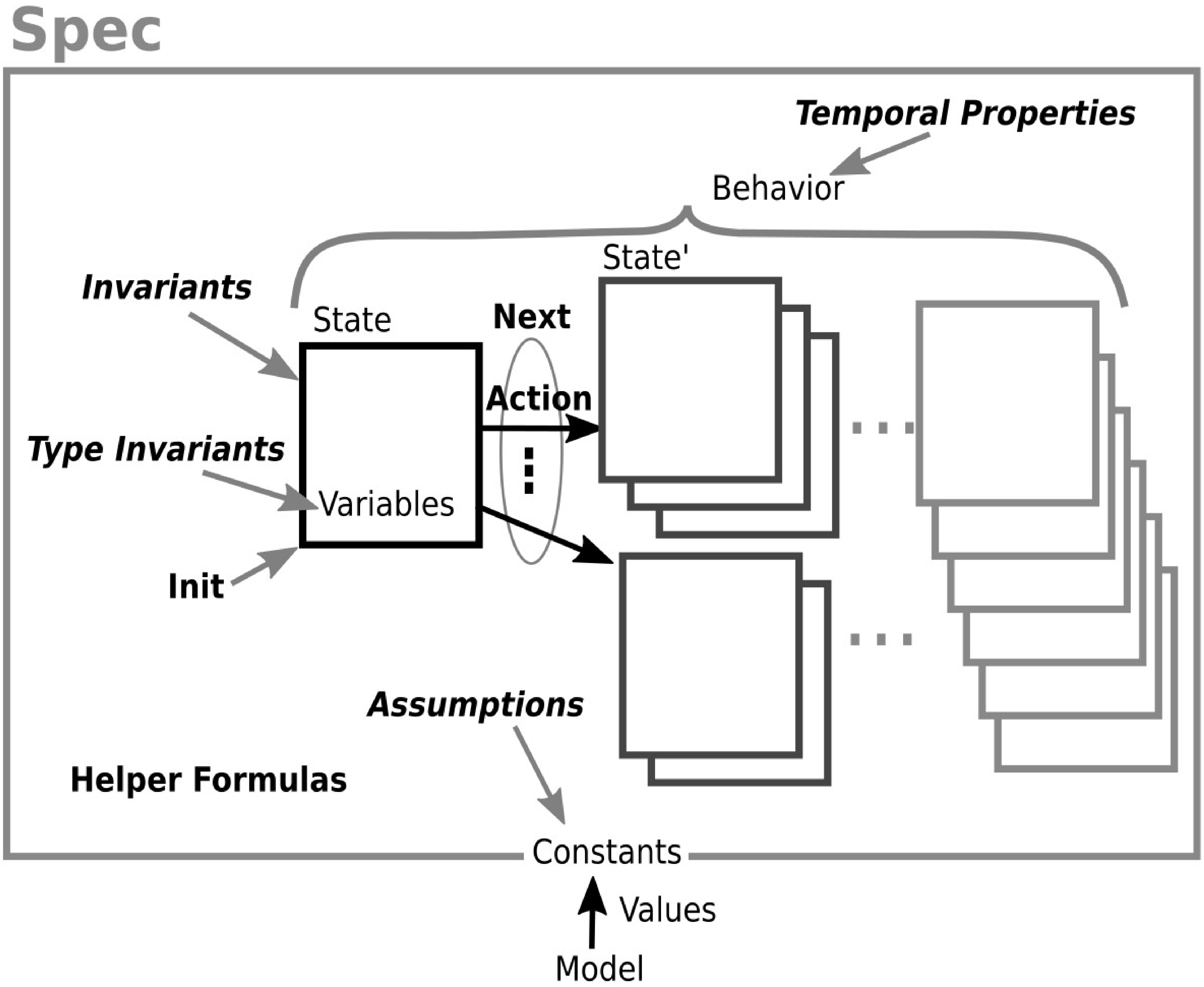
```
1 + 1
=> 2
```

REPL

- The developer is “inside” of a process running their code.
- The developer can inspect, manipulate, change the code as the process continues to run.

Incremental Coding

- Write one “layer” of a specific
- Confirm it works
- Move on to the next layer


Spec
Temporal Properties
Behavior
State'
Invariants
State
Next
Action
Type Invariants
Variables
Init
Assumptions
Helper Formulas
Constants
Values
Model

Refactoring

- Change the structure of a specification without changing its meaning
- Do this repeatedly as thinking is clarified
- Do this confidently, without fear of breaking the specification

Auto-Formatting

- Avoid need to manually layout spacing of formulas
- As a specification is iteratively refactored and incrementally created

Familiar Syntax

What problem are we trying to solve?

Facilitate Developers Use of TLA$^+$ by Providing:

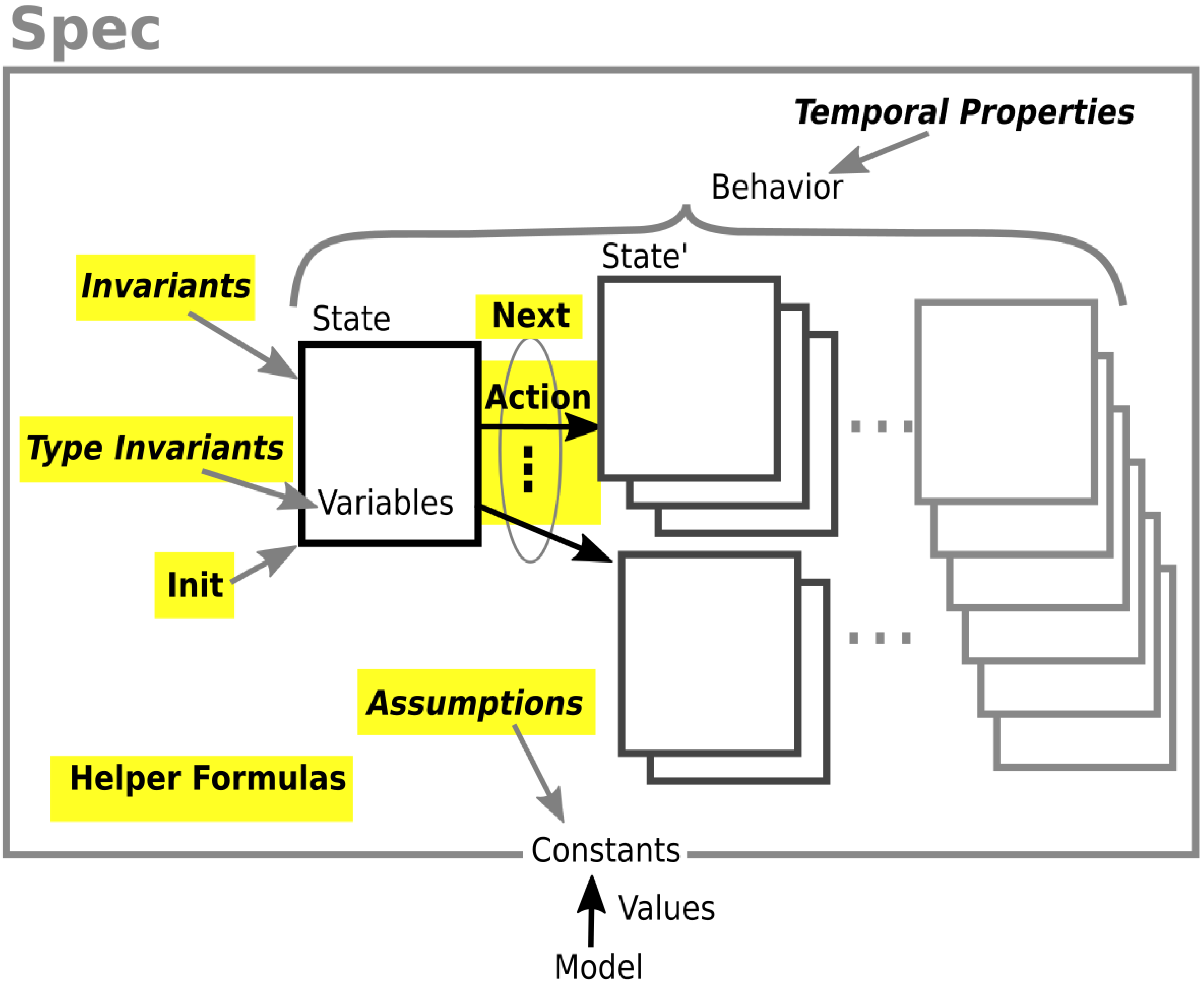
- **Interactivity**
  - Enter formulas, see what they evaluate to
- **Incremental Coding**
  - Gradual creation of spec
- **Refactoring**
  - Reorganizing formulas to improve clarity or performance
- **Auto Formatting**
  - Avoid manual formatting
- **Familiar Syntax**

Goals

- Facilitate authoring of TLA$^+$ specifications
- Lead users to the TLA$^+$ language
- Promote TLA$^+$ concepts
  - Provide one-to-one correspondence with TLA$^+$ concepts
- Leverage TLC Model Checker

Non-Goals

- Define new concepts
- Promote imperative programming
- Tooling for temporal operators
- Build a model checker
- Code generation
  - i.e. “can I generate code from the spec?”


Spec
Temporal Properties
Behavior
State'
Invariants
State
Next
Action
Type Invariants
Variables
Init
Assumptions
Helper Formulas
Constants
Values
Model

Consider a Language which:

- “takes a … mathematical view of the world”
- Eschews traditional, static typing
- Promotes sets as primary data structure
  - Literal syntax based on: { 1, 2, 3 }
- Is focused on concurrent systems:
  - Represent state with immutable data
  - Avoid side-effects
  - Carefully model state changes as atomic steps
- Provides mechanism for syntactic substitution
- Provides for mechanical manipulation of “code”

Consider **Clojure** which:

- “takes a … mathematical view of the world”
- Eschews traditional, static typing
- Promotes sets as primary data structure
  - Literal syntax based on: #{ 1, 2, 3 }
- Is focused on concurrent systems:
  - Represent state with immutable data
  - Avoid side-effects
  - Carefully model state changes as atomic steps
- Provides mechanism for syntactic substitution
- Provides for mechanical manipulation of “code”

Clojure

- A Modern Lisp
- Compiles into Java bytecode*
- Runs on Java Virtual Machine (JVM)

* and other targets

Clojure REPL

- Can work at REPL prompt
- Can work in an editor that sends expressions to the REPL to be evaluated and displays results

s-expressions

- Lists

```
(1 2 3)
```

- Nested Lists

```
((1 2) 3)
```

- Code as lists

```
(+ 1 2)
```

- Code as nested lists

```
(+ 1 (+ 2 3))
```

Primitives

Clojure	TLA$^+$
`1`	`1`
`"hello"`	`"hello"`
`true`	`TRUE`
`false`	`FALSE`
`(let [x 1] x)`	`LET x == 1 IN x`

Data Structures

Clojure	Clojure example	TLA+	TLA+ example
set	`#{1 2}`	set	`{1, 2}`
vector	`[1 2]`	tuple	`<<1, 2>>`
map	`{1 10 2 20}`	function	`(1 :> 10 @@ 2 :> 20)`
map	`{:a 10 :b 20}`	record	`[a |-> 100, b |-> 200]`
function	`(defn Add [x y] (+ 1 2))`	operator	`Add( x, y ) == x + y`
lambda	`(fn [x] (> x 2))`	lambda in SelectSeq	`LAMBDA x: (x > 2)`
lambda	`#(* 2 %)`	lambda in Except	`@ * 2`

Logic

Clojure

```
(and true false)
```

```
(A [x #{1 2 3}]
     (> x 2))
```

```
(or true false)
```

```
(E [x #{1 2 3}]
   (> x 2))
```

TLA+

```
/\  TRUE
/\  FALSE
```

```
\A x \in { 1, 3, 2 } :
       x > 2
```

```
\/  TRUE
\/  FALSE
```

```
\E x \in { 1, 3, 2 } :
    x > 2
```

Operators

Clojure

```
(defn Add [x y]
  (+ x y))
```

TLA$^+$

```
Add( x, y ) ==
    x + y
```

Recursive Operators

Clojure

```
(defn Add [x r]
  (if (> x 5)
     (Add (- x 1)
       (+ r 1))
     r))
```

TLA$^+$

```
RECURSIVE Add(_, _)


Add( x, r ) ==
    IF  (x > 5)
    THEN Add((x - 1),
             (r + 1))
    ELSE r
```

Higher Order Operators

Clojure

```
(defn Work [f a b]

      (f a b))
```

TLA$^+$

```
Work( f(_, _), a, b ) ==

    f(a, b)
```

SALT Identifiers

Identifiers from the Clojure language:

`and` `comment` `cond` `conj` `contains?` `count` `def` `defn` `difference` `expt` `first` `fn` `if` `intersection` `into` `let` `ns` `not` `or` `require` `rest` `select` `str` `subset?` `superset?` `union`

Identifiers from the Clojure language, whose semantics were modified to match TLA+:

`every?*` `get*` `map*` `mod*` `range*`

Identifiers that were added specifically to support transpiling, they are neither part of Clojure nor TLA+:

`always-` `CHANGED-` `defm-` `eventually-` `fm-` `leads-to-` `line-` `maps-` `subset-proper?` `superset-proper?` `VARS-`

Identifiers from the TLA+ language:

`==` `=>` `<=>` `=` `>` `<` `>=` `<=` `+` `-` `*` `A` `ASSUME` `Cardinality` `CHOOSE` `CONSTANT` `div` `DOMAIN` `E` `EXCEPT` `Nat` `not=` `UNCHANGED` `UNION` `SelectSeq` `Seq` `SF` `SubSeq` `SUBSET` `VARIABLE` `WF` `X`

s-expressions

- Evaluate
- Simplify
- Simulate
- Transpile

s-expressions

- Editors understand s-expression structure
  - e.g. paredit mode in Emacs
- Coding is manipulating the AST

```
(ns Clock
  (:require [salt.lang :refer :all]))

(VARIABLE clock)

(defn Init []
  (contains? #{0 1} clock))
```

salt/evaluate

```
(defmacro evaluate
  "Create a context in which constants are
defined as in 'constants' and variables are
defined as in 'state', then evaluate the 'body'
within that context. All constants and variables
referenced by the body must be defined."
  [constants state body]
  ...)
```

```
(salt/evaluate {}
               {clock 0}
               (Init))
=> true
```

```
(salt/evaluate {}
               {clock 1}
               (Init))
=> true
```

```
(salt/evaluate {}
               {clock 2}
               (Init))
=> false
```

```
(ns Clock
  (:require [salt.lang :refer :all]))

(VARIABLE clock)

(defn Init []
  (contains? #{0 1} clock))

(defn Tick []
  (or (and (= clock 0)
           (= clock' 1))
      (and (= clock 1)
           (= clock' 0))))
```

```
(salt/evaluate {}
               {clock 0
                clock' 1}
               (Tick))
=> true
```

```
(salt/evaluate {}
               {clock 0
                clock' 0}
               (Tick))

=> false
```

```
(deftest test-tick
  (is (salt/evaluate {}
                     {clock 0
                      clock' 1}
                     (Tick)))

  (is (not (salt/evaluate {}
                          {clock 0
                           clock' 0}
                          (Tick)))))
```

```
(deftest test-tick
  (is (salt/evaluate {}
                     {clock 0
                      clock' 1}
                     (Tick)))

  (is (not (salt/evaluate {}
                          {clock 0
                           clock' 0}
                          (Tick)))))

(run-tests)
=> {:test 1, :pass 2, :fail 0, :error 0,
:type :summary}
```

Coding Cycle

- Think
- Design
- Code
- Get feedback
- Iterate

Coding “Gears”

- Does the code compile?
- Does the code run?
- Does the code give the correct answer for trivial input*?
- Does the code give the correct answer for non-trivial input?
- Does the code give the correct answer for tricky edge cases?

* tests effectively used are more than guard-rails, rather they verify our design thoughts.

salt/simplify

```
(defmacro simplify
  "Read a salt source file, create a context in
which constants are defined as per the
'constants' parameter and the variables are
defined per the 'state' parameter. Then simplify
the expression, 'e' in that context. Values for
all constants referenced by the expression must
be provided. Values for a subset of the variables
referenced by the expression can be provided."
  [src-file-name constants state e]
  ...)
```

```
(salt/simplify "Clock.clj"
               {}
               {clock 0}
               (Tick))

=> (= clock' 1)
```

```
(salt/simplify "Clock.clj"
               {}
               {clock' 0}
               (Tick))

=> (= clock 1)
```

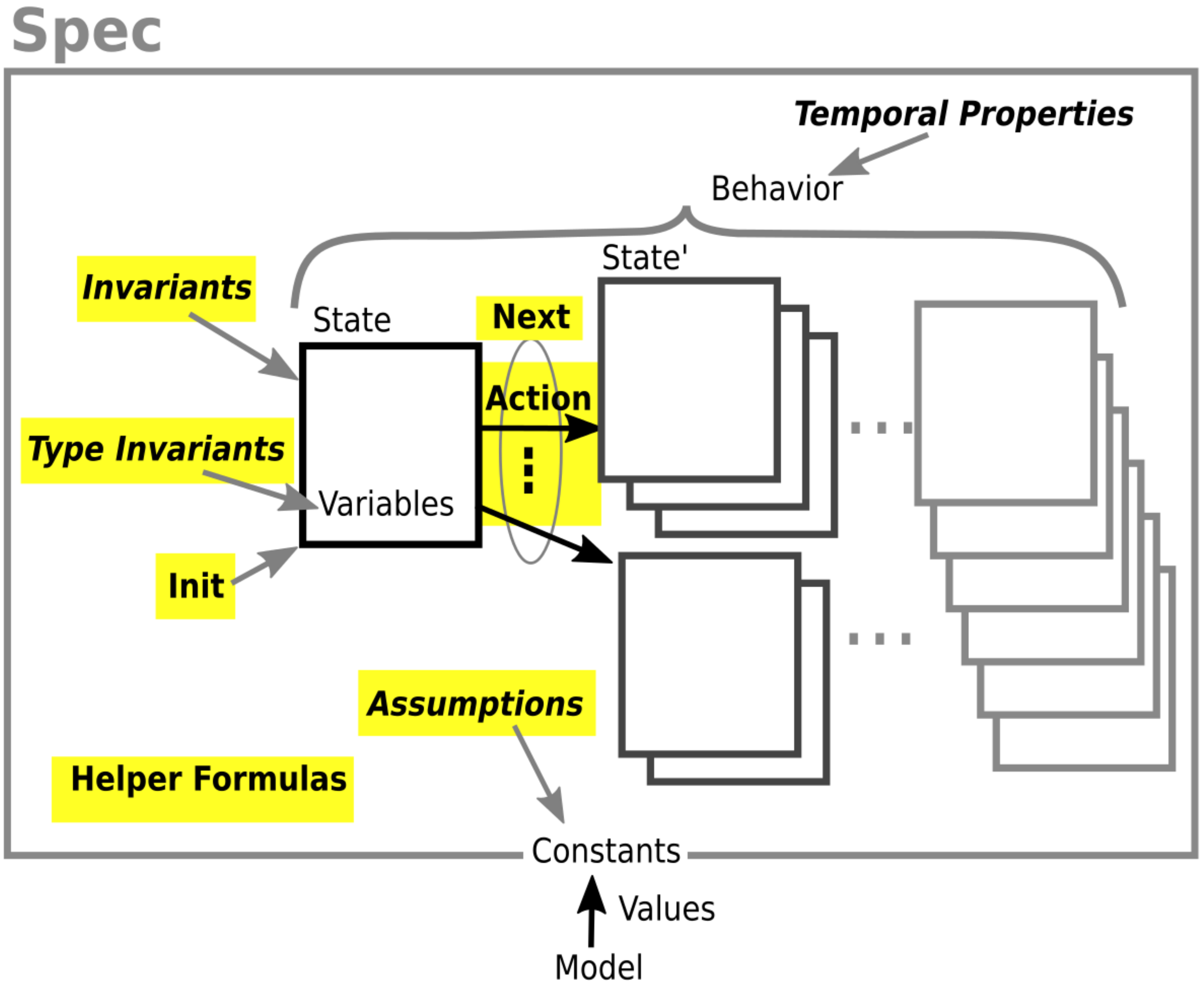
```
(salt/simplify "Clock.clj"
               {}
               {clock 1
                clock' 1}
               (Tick))
=> false
```

salt/simulate

```
(defmacro simulate
  "Same arguments as for salt/simplify with
  the addition of 'n' specifying how many
  simulations to run. Seeks to explore non-
  determinism to identify values for the
  unbound variables that make the expression
  true."
  [src-file-name constants state n e]
```

```
(salt/simulate "Clock.clj"
               {}
               {clock 0}
               100
               (Tick))

 => #{{clock' 1}}
```

- Interact with REPL
- Incrementally build spec
- Write tests as appropriate
- Refactor
- Re-run test suites

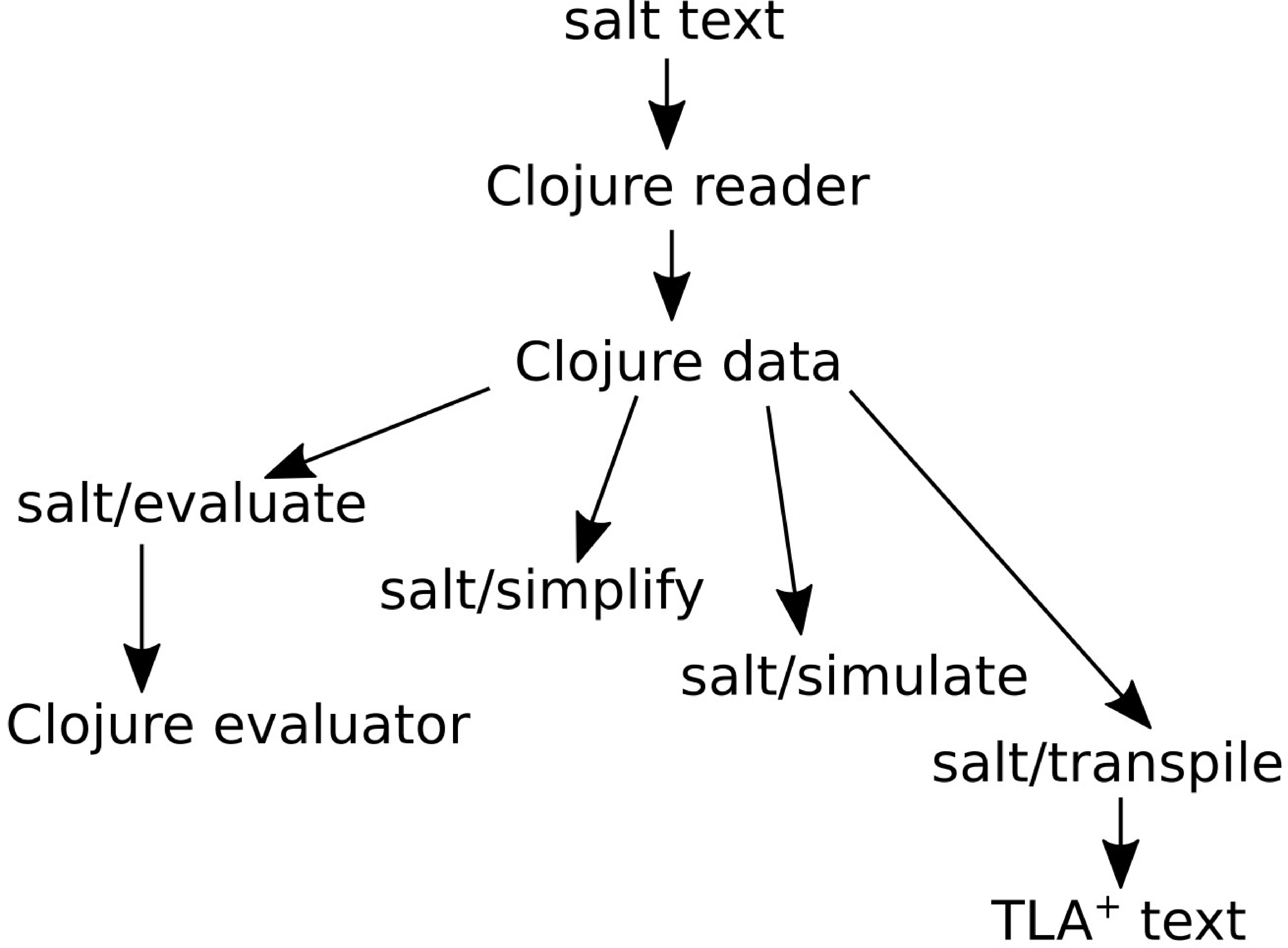


salt/transpile

```
(defn transpile
  "Read a salt source file, convert the
contents to TLA+ tokens and return the
results as a string."
  [src-file-name]
  ...)
```

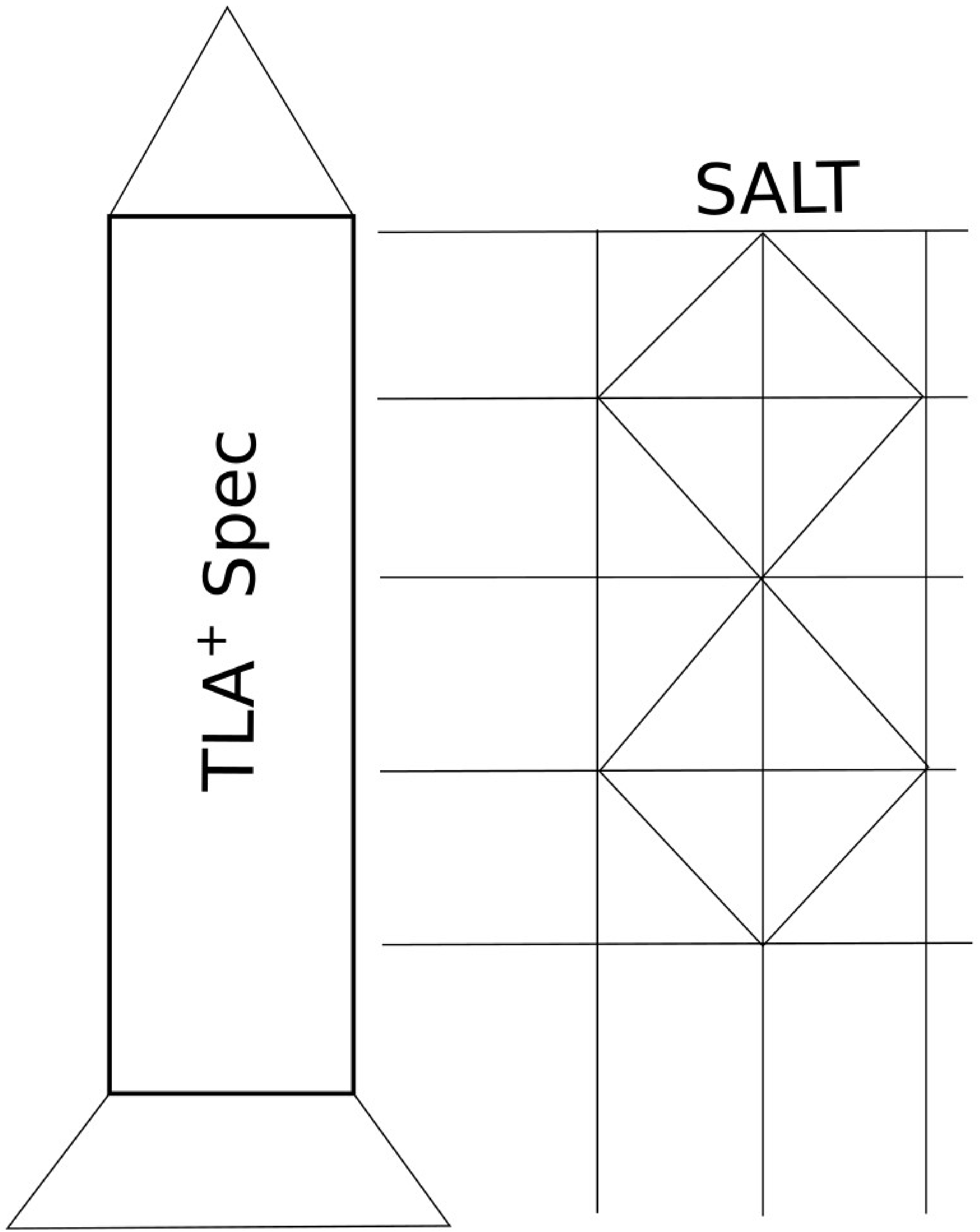
```
(ns Clock
  (:require [salt.lang :refer :all]))

(VARIABLE clock)

(defn Init []
  (contains? #{0 1} clock))

(defn Tick []
  (or (and (= clock 0)
           (= clock' 1))
      (and (= clock 1)
           (= clock' 0))))

(defn Spec []
  (and (Init)
       (always- (Tick) [clock])))
```

```
(salt/transpile "Clock.clj")

=>
"---------------------------- MODULE Clock ----------------------------
VARIABLE clock

Init == clock \\in { 0, 1 }

Tick ==
    IF  (clock = 0)
    THEN (clock' = 1)
    ELSE (clock' = 0)

Spec ==
    /\\  Init
    /\\  [][Tick]_<< clock >>


======================================================================
"
```


salt text
Clojure reader
Clojure data
salt/evaluate
salt/simplify
salt/simulate
salt/transpile
Clojure evaluator
TLA+ text

SALT Provides

- Interactivity
  - REPL
- Incremental Coding
  - Automated tests
- Refactoring
  - Regression test suite
- Auto Formatting
  - s-expressions in “paredit” mode
- Familiar Syntax
  - Clojure


SALT
TLA$^+$ Spec

TLA^+ Adoption

- Awareness
  - Know about the tool
  - Not silver bullet
- Ease of Use *<- SALT targeted at this*
  - Know the mechanics of using the tool
- Learned skill
  - Know how to use the tool well
  - “thinking above the code”

Make the mechanism more comfortable to developers so we can mature to the point of learning the skills of “thinking above the code”.

SALT In Practice

- Successfully using it to write TLA$^+$ specs
- Accomplishes its goals
- Specs are development interactively
- Specs are tested as they are written
- When TLA$^+$ specs are produced they are solid

SALT: Areas for improvement

- More refined comment handling
- Considered support for multi-module specifications
- Real numbers
- Infinite sets
- Forms such as multiple variables in \E
- Thorough pass through all of TLA$^+$ syntax and semantics (e.g. CASE semantics)
- Refine names of language identifiers
- Additional simplification rules
- More sophisticated simulator (currently just a minimal proof-of-concept)
- ...

https://github.com/Viasat/salt

```
(comment "example spec ported from

https://github.com/tlaplus/Examples/blob/master/specifications/transaction_commit/TwoPhase.tl
a")


(ns TwoPhase
  (:require [salt.lang :refer :all]))


(CONSTANT RM)


(VARIABLE rmState tmState tmPrepared msgs)


(defn Message [] (union (maps- [:type #{"Prepared"}
                                :rm RM])
                        (maps- [:type #{"Commit" "Abort"}])))


(defn TPTypeOk []
  (and (contains? (maps- RM #{"working" "prepared" "committed" "aborted"}) rmState)
       (contains? #{"init" "committed" "aborted"} tmState)
       (subset? tmPrepared RM)
       (subset? msgs Message)))


(defn TPInit []
  (and (= rmState (fm- [rm RM]
                       "working"))
       (= tmState "init")
       (= tmPrepared #{})
       (= msgs #{})))
```

```
(defn TMRcvPrepared [rm]
  (and (= tmState "init")
       (contains? msgs {:type "Prepared"
                        :rm rm})
       (= tmPrepared' (union tmPrepared #{rm}))
       (CHANGED- [tmPrepared])))

(defn TMCommit []
  (and (= tmState "init")
       (= tmPrepared RM)
       (= tmState' "committed")
       (= msgs' (union msgs #{{:type "Commit"}}))
       (CHANGED- [tmState msgs])))

(defn TMAbort []
  (and (= tmState "init")
       (= tmState' "aborted")
       (= msgs' (union msgs #{{:type "Abort"}}))
       (CHANGED- [tmState msgs])))
```

```
(defn RMPrepare [rm]
  (and (= (get* rmState rm) "working")
       (= rmState' (EXCEPT rmState [rm] "prepared"))
       (= msgs' (union msgs #{{:type "Prepared"
                               :rm rm}}))
       (CHANGED- [rmState msgs])))

(defn RMChooseToAbort [rm]
  (and (= (get* rmState rm) "working")
       (= rmState' (EXCEPT rmState [rm] "aborted"))
       (CHANGED- [rmState])))

(defn RMRcvCommitMsg [rm]
  (and (contains? msgs {:type "Commit"})
       (= rmState' (EXCEPT rmState [rm] "committed"))
       (CHANGED- [rmState])))

(defn RMRcvAbortMsg [rm]
  (and (contains? msgs {:type "Abort"})
       (= rmState' (EXCEPT rmState [rm] "aborted"))
       (CHANGED- [rmState])))
```

```
(defn TPNext []
  (or (TMCommit)
      (TMAbort)
      (E [rm RM]
         (or (TMRcvPrepared rm)
             (RMPrepare rm)
             (RMChooseToAbort rm)
             (RMRcvCommitMsg rm)
             (RMRcvAbortMsg rm)))))

(defn TPSpec []
  (and (TPInit)
       (always- (TPNext) [rmState tmState
tmPrepared msgs])))
```

```
--------------------------- MODULE TwoPhase ----------------------------


(*
example spec ported from
https://github.com/tlaplus/Examples/blob/master/specifications/transaction_commit/TwoPhas
e.tla
*)
CONSTANT RM


VARIABLE rmState, tmState, tmPrepared, msgs


Message == [type : {"Prepared"},
            rm : RM] \union [type : { "Commit", "Abort" }]


TPTypeOk ==
    /\  rmState \in [RM -> { "committed", "prepared", "aborted", "working" }]
    /\  tmState \in { "committed", "aborted", "init" }
    /\  tmPrepared \subseteq RM
    /\  msgs \subseteq Message


TPInit ==
    /\  rmState = [rm \in RM |-> "working"]
    /\  tmState = "init"
    /\  tmPrepared = {}
    /\  msgs = {}
```

```
TMRcvPrepared( rm ) ==
    /\  tmState = "init"
    /\  [type |-> "Prepared",
         rm |-> rm] \in msgs
    /\  tmPrepared' = tmPrepared \union {rm}
    /\  UNCHANGED << rmState, tmState, msgs >>

TMCommit ==
    /\  tmState = "init"
    /\  tmPrepared = RM
    /\  tmState' = "committed"
    /\  msgs' = msgs \union {[type |-> "Commit"]}
    /\  UNCHANGED << rmState, tmPrepared >>

TMAbort ==
    /\  tmState = "init"
    /\  tmState' = "aborted"
    /\  msgs' = msgs \union {[type |-> "Abort"]}
    /\  UNCHANGED << rmState, tmPrepared >>
```

```
RMPrepare( rm ) ==
    /\  rmState[rm] = "working"
    /\  rmState' = [rmState EXCEPT ![rm] = "prepared"]
    /\  msgs' = msgs \union {[type |-> "Prepared",
                              rm |-> rm]}
    /\  UNCHANGED << tmState, tmPrepared >>

RMChooseToAbort( rm ) ==
    /\  rmState[rm] = "working"
    /\  rmState' = [rmState EXCEPT ![rm] = "aborted"]
    /\  UNCHANGED << tmState, tmPrepared, msgs >>

RMRcvCommitMsg( rm ) ==
    /\  [type |-> "Commit"] \in msgs
    /\  rmState' = [rmState EXCEPT ![rm] = "committed"]
    /\  UNCHANGED << tmState, tmPrepared, msgs >>

RMRcvAbortMsg( rm ) ==
    /\  [type |-> "Abort"] \in msgs
    /\  rmState' = [rmState EXCEPT ![rm] = "aborted"]
    /\  UNCHANGED << tmState, tmPrepared, msgs >>
```

```
TPNext ==
    \/  TMCommit
    \/  TMAbort
    \/  \E rm \in RM :
            \/  TMRcvPrepared(rm)
            \/  RMPrepare(rm)
            \/  RMChooseToAbort(rm)
            \/  RMRcvCommitMsg(rm)
            \/  RMRcvAbortMsg(rm)

TPSpec ==
    /\  TPInit
    /\  [][TPNext]_<< rmState, tmState, tmPrepared, msgs >>


=========================================================
```