Temporal Logic Specifications for Parallel Debugging

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at

Contents

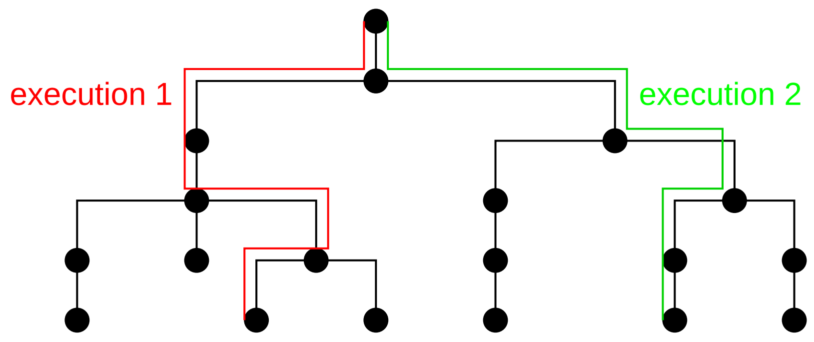
- Non-Determinism and Parallel Debugging
- Temporal Logic Specifications
- Specification Calculus
- System Architecture and Interfaces

Non-Determinism and Parallel Debugging

Debugging

- Sequential program:
  one execution per input
  (deterministic execution)

- Parallel program:
  several executions possible (non-determinism).



Handling non-determinism is a key problem in parallel debugging.

Sources of Non-Deterministim

Assume: message passing model with reliable transfer.

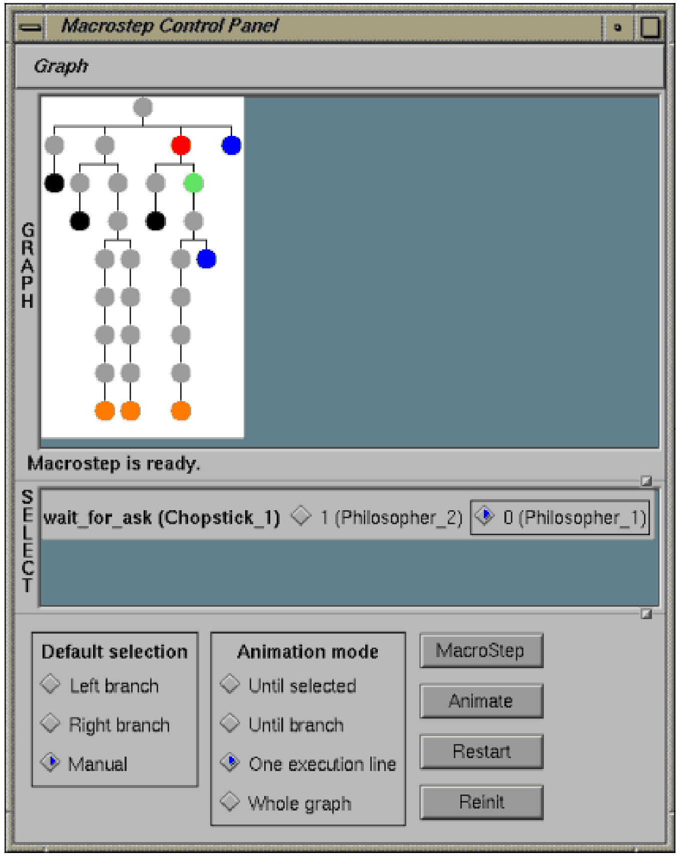
- Non-determinism arise because of
  - alternative receive operations from multiple sources,
  - non-blocking receive operations,
  - effects outside the message passing model.
- MPI Message Passing Interface:
  - `MPI_ANY_SOURCE`: message from any sender accepted.
  - `MPI_IPROBE`: non-blocking test for message availability.
  - File communication, multi-threading, etc.

Focus: non-determinism from alternative receive operations.

P-GRADE Macrostep Debugging

- MTA SZTAKI and SGI
- Controlled selection of alternative inputs
- Manual or automated traversal of state tree

http://www.lpds.sztaki.hu/projects/p-grade



Idea

How to further aid debugging of non-deterministic parallel programs?

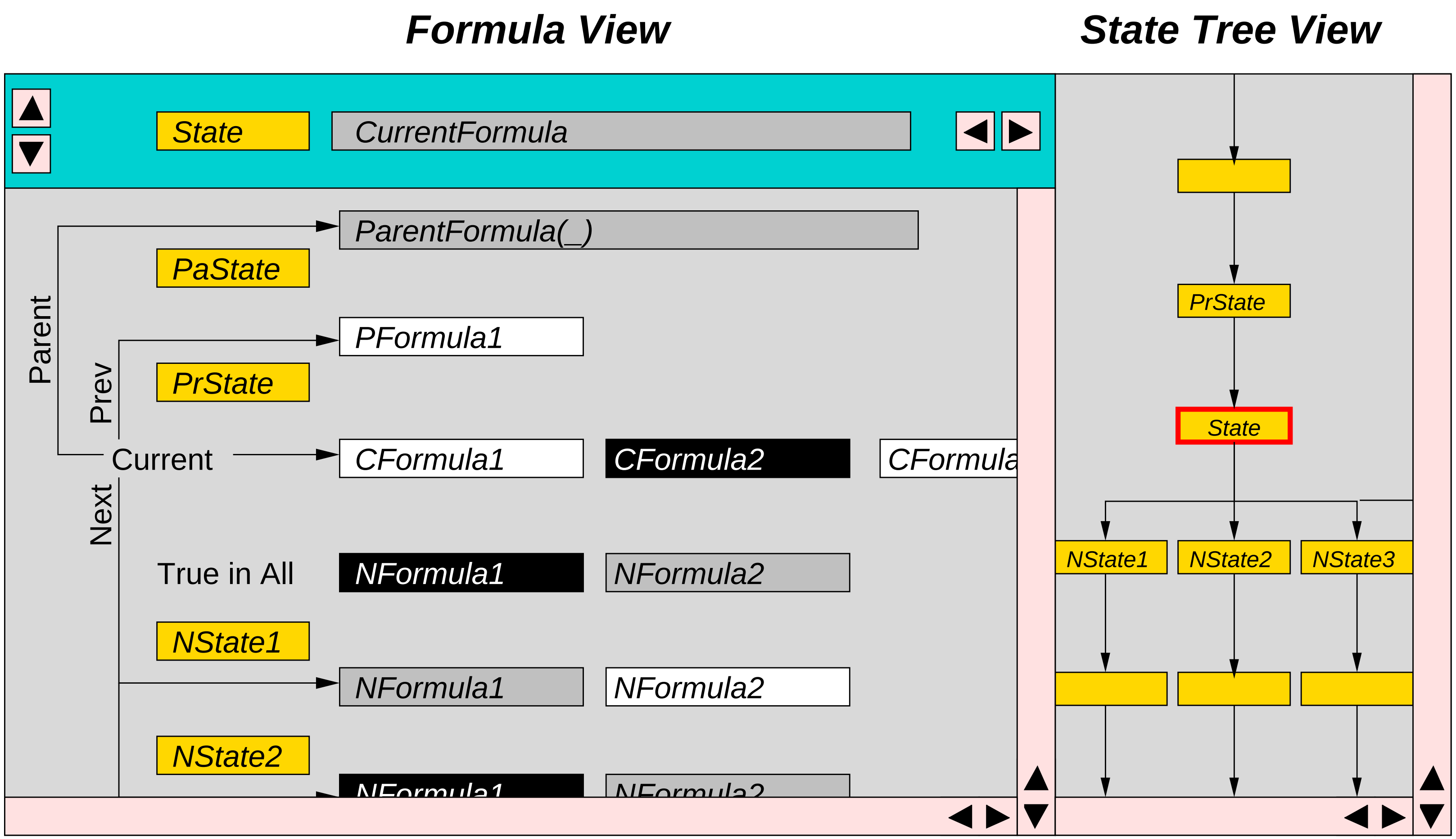
- Sequential programs: debugging crucially aided by assertions.

  Property that must be true in denoted state.

- Parallel programs: how to employ assertions?
  - State distributed among processes ⇒ need to construct consistent global state.
  - Multiple state sequences possible ⇒ assertion must be designed to hold in every sequence.
  - Useful properties involve multiple states ⇒ assertion must talk about whole state sequence.

Let assertions be generated from formal program specifications.

Macrostep Specification Checking



Temporal Logic Specifications

Example: Mutual Exclusion

$\mathbf{P_0}$:

```
        c := −1; w := ⟨⟩
        loop
          (s_0, m_0) := receive()
a:        if m_0 = "enter" then
            if c_0 = −1
              then c_0 := s_0; send(s_0, "okay")
b:            else w_0 := w_0 || ⟨s_0⟩
            end
          else if m = "exit" then
            c_0 := −1
            if w_0 ≠ ⟨⟩ then
              c_0 := head(w_0); w_0 := tail(w_0)
c:           send(c_0, "okay")
            end
          end
        end
```

$\mathbf{P_{i,i>0}}$:

```
        loop
p:        send(0, "enter")
q:        m_i := receive(0)
r:        [critical region]
          send(0, "exit")
        end
```

Crucial Properties

- Mutual exclusion:

  Always, if client i is in the critical region, then client $j \neq i$ is not in the critical region.
  $\forall i \in 1 \ldots n : \Box[\mathrm{at}_{\mathrm{r}}(i) \Rightarrow (\forall j \in 1 \ldots n : j \neq i \Rightarrow \neg \mathrm{at}_{\mathrm{r}}(j))]$

- Progress:

  Always, if client i requests access to the critical region, it eventually enters the region.
  $\forall i \in 1 \ldots n : \Box[\mathrm{at}_{\mathrm{q}}(i) \Rightarrow \Diamond \mathrm{at}_{\mathrm{r}}(i)]$

Specification of system functionality from clients' point of view.

More Properties

- No request is lost:

  Always, if the client is to send a request, the server eventually receives it:
  $\forall i \in 1..n : \Box[\mathrm{at_p}(i) \Rightarrow \Diamond(\mathrm{at_a}(i) \land s_0 = i \land m_0 = \text{"enter"})]$

- No request is forgotten:

  Always, if server i does not immediately answer the request, it will later answer it:
  $\Box[\mathrm{at_b}(0) \Rightarrow \text{ let } i = s_0 \text{ in } \Diamond(\mathrm{at_c}(0) \land c_0 = i)]$

- No request is added:

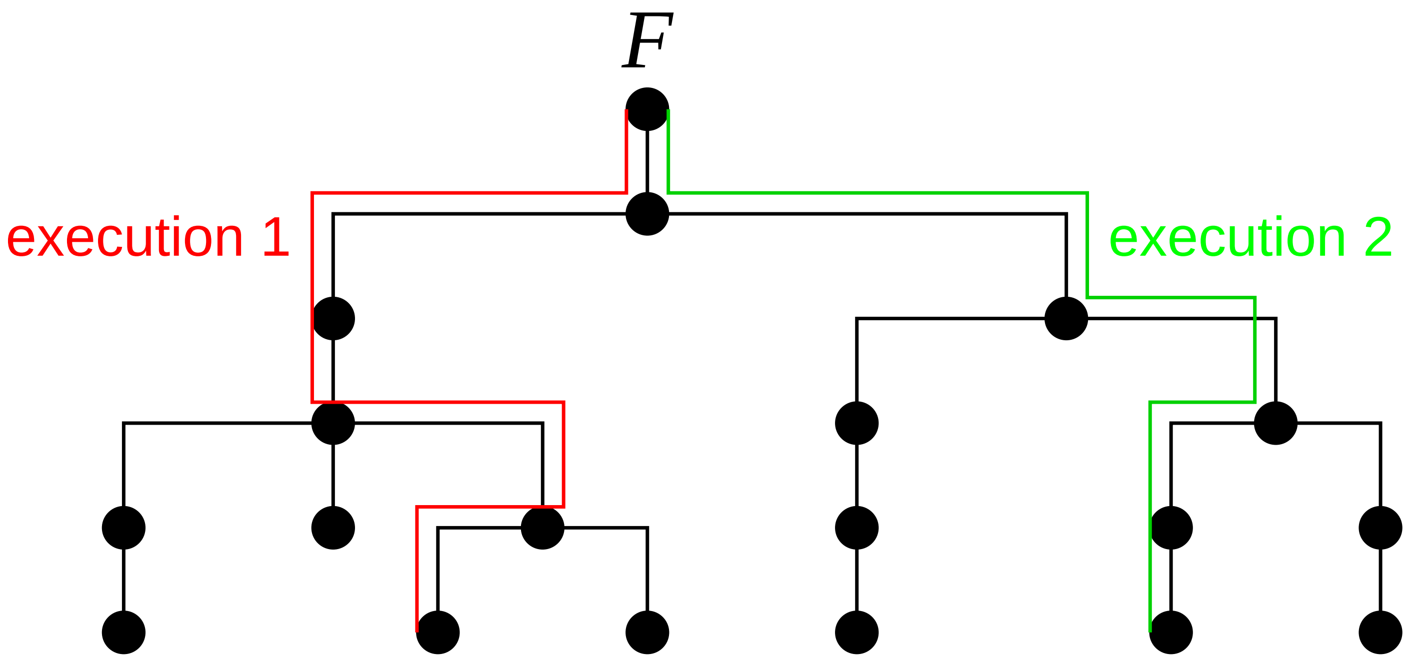
  Always, if the server grants access to client i, it has previously received a request from i:
  $\Box[\mathrm{at_c}(0) \Rightarrow \text{ let } i = c_0 \text{ in } \diamondminus(\mathrm{at_a}(0) \land s_0 = i)]$

Detailed description of system functionality possible.

Temporal Logic Formulas

- Atomic formulas $p_n(t_0, \ldots, t_{n-1})$
  - Mathematical variables x
  - Program variables v_i
  - Program counters $\mathrm{at}_{\mathrm{c}}(i)$
  - Message buffers msgbuf_i
- Connectives $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$
- Quantifiers $\forall x \in D$, $\exists x \in D$, $\mathrm{let}\ x = T$
- Temporal operators
  - Always $\Box$
  - Eventually $\Diamond$
  - Leads to $\leadsto$
  - Past variants: $\boxminus$ $\diamondminus$ $\leadsto\!\!-$

Temporal Logic Specifications



A temporal logic formula F can express a property about all possible executions of a (non-deterministic, parallel) program.

Specificaton Calculus

Specification Calculus

1. Semantics of temporal logic formulas.

   Semantics in terms of state sequences.

2. Translation to guarded temporal formulas.

   Only knowledge about previous state, current state, next state required.

3. Tree semantics of temporal formulas.

   Semantics in terms of state trees.

4. Partial tree semantics of temporal formulas.

   Next state may be unknown.

5. Extraction of (previous/current/next)-state formulas.

Temporal Logic Semantics

- Validity of temporal logic formulas:

  A temporal formula F is true for a program with initial states $is \subseteq \text{State}$ and next state relation $ns \subseteq \text{State} \times \text{State}$, if it is true for every state sequence induced by *is* and *ns*:

  $$S(is, ns) := \{s : \mathbb{N} \rightarrow \text{State} : s_0 \in is \wedge \forall i \in \mathbb{N} : (s_i, s_{i+1}) \in ns\}$$
  $$\cup \{s : \mathbb{N}_n \rightarrow \text{State} : s_0 \in is \wedge \forall i \in \mathbb{N}_{n-1} : (s_i, s_{i+1}) \in ns \wedge \neg \exists x : (s_{n-1}, x) \in ns\}$$

  $$\mathbf{T}[[F]]is\ ns :\Leftrightarrow \forall s \in S(is, ns) : \mathbf{T}[[F]]s\ 0$$

- Validity of a formula in a non-empty state sequence s:

  $$\mathbf{T}[[p_n(t_0, \ldots, t_{n-1})]]s\ i = [[p_n]]([[t_0]]s_i, \ldots, [[t_{n-1}]]s_i)$$
  $$\ldots$$
  $$\mathbf{T}[[\Box F]]s\ i \Leftrightarrow \text{true iff } \mathbf{T}[[F]]s\ j = \text{true for all } j \text{ with } i \leq j < |s|$$
  $$\mathbf{T}[[\Diamond F]]s\ i \Leftrightarrow \text{true iff } \mathbf{T}[[F]]s\ j = \text{true for some } j \text{ with } i \leq j < |s|$$
  $$\mathbf{T}[[\boxminus F]]s\ i \Leftrightarrow \text{true iff } \mathbf{T}[[F]]s\ j = \text{true for all } j \text{ with } 0 \leq j \leq i$$

Guarded Temporal Formulas

- Guard temporal operators by $\circ$ resp. $\ominus$ ("next/previous time")

$$\ldots$$
$$\mathbf{G}[[\Box F]] = \mathbf{G}[[F]] \wedge \circ_{\text{true}} \Box F$$ — "always" becomes "now and next time always"
$$\mathbf{G}[[\Diamond F]] = \mathbf{G}[[F]] \vee \circ_{\text{false}} \Box F$$ — "eventually" becomes "now or next time event."
$$\mathbf{G}[[\boxminus F]] = \mathbf{G}[[F]] \wedge \ominus_{\text{true}} \boxminus F$$ — "once" becomes "now or previous time once"
$$\ldots$$

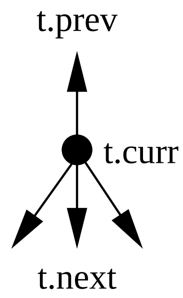
- Translation preserves semantics

$$\mathbf{T}[[F]]s\ i \Leftrightarrow \mathbf{T}(\mathbf{G}[[F]])s\ i$$

$$\mathbf{T}[[\circ_v F]]s\ i \Leftrightarrow \text{if } i+1 = |s| \text{ then } v \text{ else } \mathbf{T}[[F]]s\ (i+1)$$
$$\mathbf{T}[[\ominus_v F]]s\ i \Leftrightarrow \text{if } i = 0 \text{ then } v \text{ else } \mathbf{T}[[F]]s\ (i-1)$$

Only need current state, next state, and previous state.

Tree Semantics



- Trees of states induced by *is* and *ns*:

$$T(is, ns) := \{T(\top, i, ns) : i \in is\}$$
$$T(p, i, ns) = t \;\Leftrightarrow\; t_{\text{prev}} = p \;\wedge\; t_{\text{curr}} = i$$
$$\wedge\; t_{\text{next}} = \text{if } \exists x : (i, x) \in ns \text{ then } \{T(i, x, ns) : (i, x) \in ns\} \text{ else } \{\top\}$$

- Semantics of guarded temporal formulas based on trees

$$\mathbf{T}[[p_n(t_0, \ldots, t_{n-1})]]t = [[p_n]]([[t_0]]t_{\text{curr}}, \ldots, [[t_{n-1}]]t_{\text{curr}})$$
$$\ldots$$
$$\mathbf{T}[[\circ_v F]]t \Leftrightarrow \text{ true iff } \mathbf{T}(\mathbf{G}[[F]])x \Leftrightarrow \text{true for every } x \in t_{\text{next}}$$
$$\mathbf{T}[[\ominus_v F]]t \Leftrightarrow \text{ true iff } \mathbf{T}(\mathbf{G}[[F]])x \Leftrightarrow \text{true where } x = t_{\text{prev}}$$
$$\mathbf{T}[[\circ_v F]]\top \Leftrightarrow v$$
$$\mathbf{T}[[\ominus_v F]]\top \Leftrightarrow v$$

Translate sets of state sequences to (sets of) state trees.

Semantic Relationship

- Validity of formulas over state trees:

  $\mathbf{T}[[GF]]T \Leftrightarrow \text{true iff } \mathbf{T}[[GF]]t \Leftrightarrow \text{true for every } t \in T$

- Relationship to original semantics preserved

  $\mathbf{T}[[F]]is\ ns \Leftrightarrow \mathbf{T}(\mathbf{G}[[F]])F(is, ns)$

May operate on state trees instead of sequences.

Partial Tree Semantics

- Replace 2-valued logic in $\mathbf{T}$ by 3-valued logic ($\bot =$ unknown):

  $\neg_3\bot = \bot, \mathrm{T} \wedge_3 \bot = \bot, \mathrm{F} \wedge_3 \bot = F, \mathrm{T} \vee_3 \bot = \mathrm{T}, \mathrm{F} \vee_3 \bot = \bot$

- Assume that only part of tree is known ($\bot =$ unknown subtree).

  Tree s is a subtree of t if it equals t except for some $\bot$ subtrees:

  $s \sqsubseteq t \;\; :\Leftrightarrow \;\; s = \bot \vee (s_{\text{curr}} = t_{\text{curr}} \wedge \exists f : s_{\text{next}} \overset{\text{bij.}}{\rightarrow} t_{\text{next}} : \forall x \in s_{\text{next}} : x \sqsubseteq f(x))$

- Partial Tree Semantics

  $\mathbf{T}_3[[F]]\bot :\Leftrightarrow \bot$

- Compatibility and monotonicity:

  t does not contain $\bot \Rightarrow \mathbf{T}_3[[F]]t = \mathbf{T}_2[[F]]t$
  $s \sqsubseteq t \Rightarrow \mathbf{T}_3[[F]]s \sqsubseteq \mathbf{T}_3[[F]]t$

Extraction of State Formulas

- $\mathbf{T}(\mathbf{G}[[\Box(p_n(\ldots) \wedge q_m(\ldots))]])t$:

  $\mathbf{T}(\mathbf{G}[[\Box(p_n(\ldots) \wedge q_m(\ldots))]])t \Leftrightarrow \text{true}$
  iff $\mathbf{T}(\mathbf{G}[[p_n(\ldots) \wedge q_m(\ldots)]] \wedge \circ\Box(p_n(\ldots) \wedge q_m(\ldots)))t \Leftrightarrow \text{true}$
  iff $\mathbf{T}(\mathbf{G}[[p_n(\ldots) \wedge q_m(\ldots)]])t \Leftrightarrow \text{true and } \mathbf{T}[[\circ\Box(p_n(\ldots) \wedge q_m(\ldots))]]t \Leftrightarrow \text{true}$
  iff $\mathbf{T}[[p_n(\ldots) \wedge q_m(\ldots)]]t \Leftrightarrow \text{true and } \mathbf{T}[[\Box(p_n(\ldots) \wedge q_m(\ldots))]]x \Leftrightarrow \text{true for } x \in t_{\text{next}}$

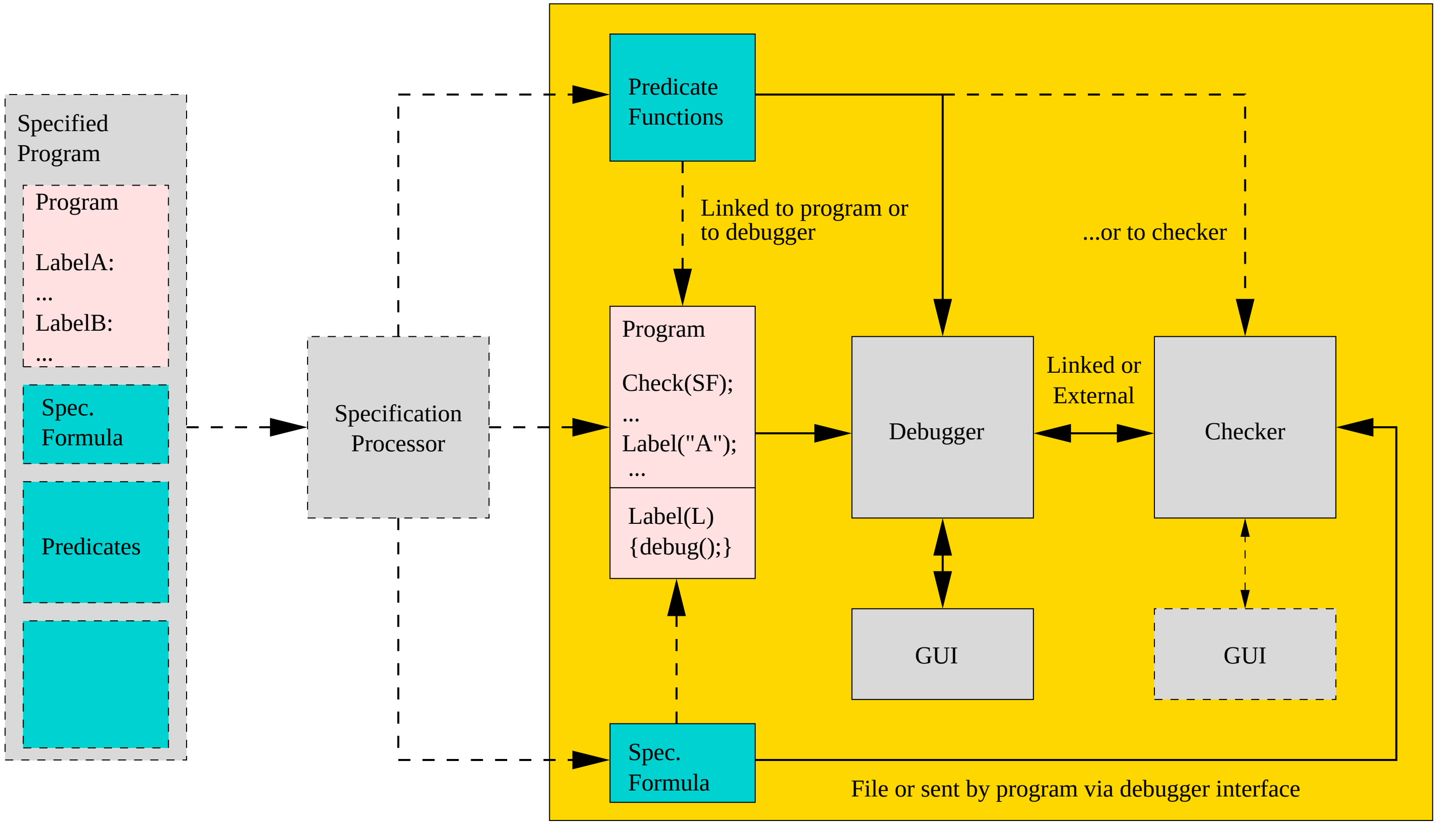
- Extracted Formulas:

Previous state:	none	(zero or more formulas)
Current state:	$p_n(\ldots), q_m(\ldots)$	(one ore more atomic formulas)
Next state:	$\Box(p_n(\ldots) \wedge q_m(\ldots))$	(zero or more formulas)

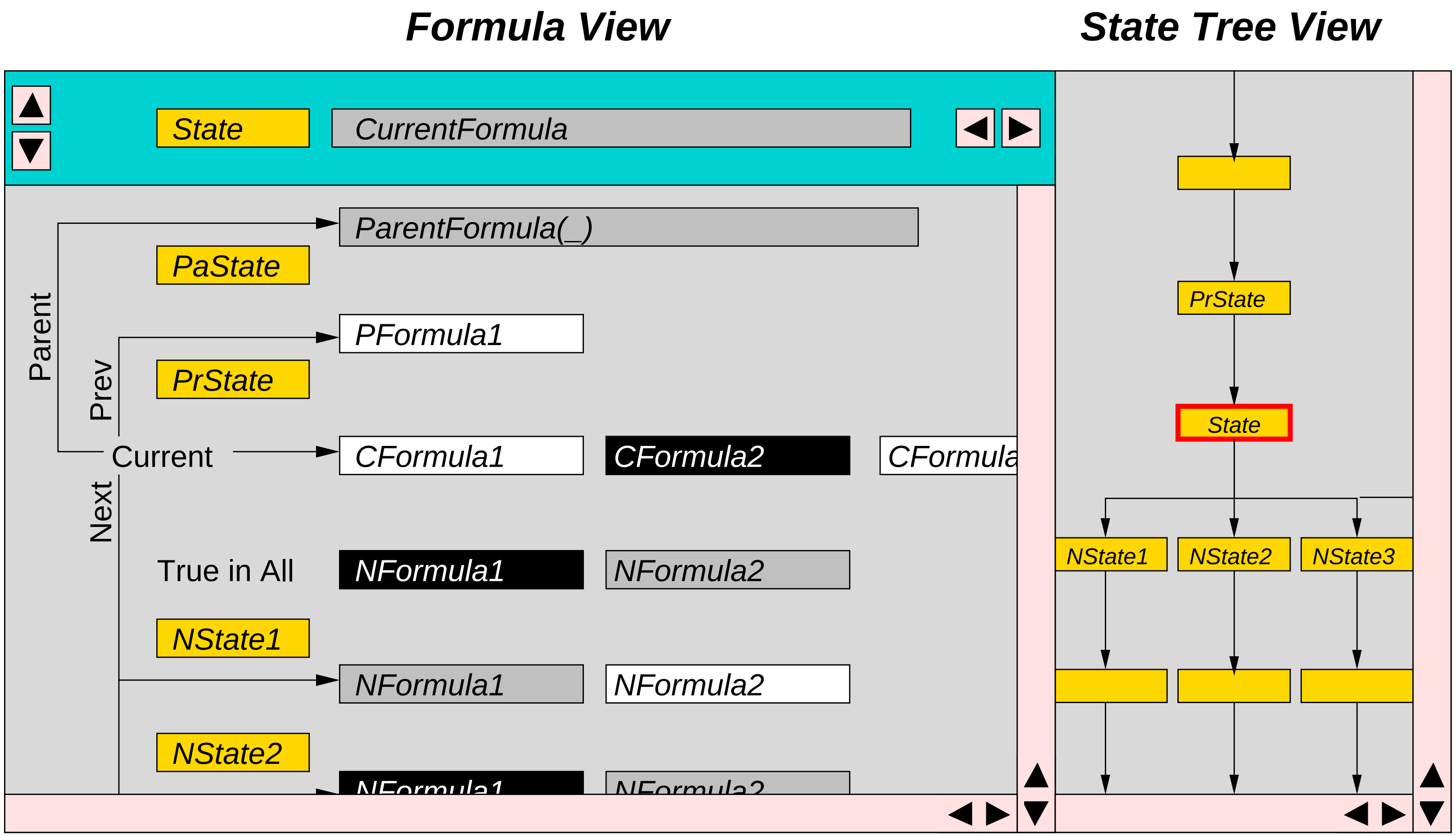
Checking formula yields formulas for previous, current, next state.

System Architecture and Interfaces

System Architecture



User Interface



Program/Debugger Interface

- Label set by program:
  - `label(`*name*`);`
  - Denotes checking state (in addition to states of receive operations?)
- Functionality of atomic predicate functions:
  - `procNumber()`
  - `getVar(`*name*`, `*procid*`)` (scope?)
  - `atLabel(`*name*`, `*procid*`)`
  - `msgNumber(`*procid*`)`, `msgSender(`*procid*`, `*i*`)`, `msgContent(`*procid*`)`
- Possibly: assert (temporal) formula in current state.
  - `tassert(formula)`
  - current state becomes root of tree for checking the formula.
  - formula as string (or possibly as object) forwarded to checker.

Debugger/Checker Interface

- General questions:
  - Is checker external program or linked to debugger?
  - If external, own graphical user interface?
- Debugger functionality (used by checker):
  - `type State = void*`
    pointer that represents program state in debugger
  - `eval(State state, String f, int x0, ...)`
    determine value of named atomic formula in denoted state with given values for the mathematical variables free in the formula.
- Checker functionality (used by debugger):
  - Determine validity of specification with partial state tree.
  - Notify debugger when truth of formula in state has changed (from unknown to true/false).

Checker Functionality

```
interface Node
{
  // register callback function for value change notification
  static void setNodeFormulaNotify(void (*f)(NodeFormula))

  // register formula to be checked with current node as root
  void setFormula(String f)

  static Node topNode()     // constructors
  Node addChild(State state)
  void noMoreChildren()

  State getState()          // selectors
  NodeFormula[] getNodeFormulas()
}
```

Checker Functionality

```
interface NodeFormula
{
  Node getNode()                  // node of formula
  Bool3 value()                   // value of formula in node
  String getString()              // string representation of formula
  String[] getString(NodeFormula s); // ...before/after subformula s

  NodeFormula getParent()         // parent formula
  NodeFormula[] getPrevious()     // previous state formulas
  NodeFormula[] getCurrent()      // current state formulas
  NodeFormulaP[] getNext()        // proxy for next state formulas
}
```

Checker Functionality

```
interface NodeFormulaP
{
  int getNumber()                     // number of next states
  NodeFormula getNodeFormula(int i) // formula for next state i
  boolean allTrue()                   // formula in all states true?
  boolean someFalse()                 // formula in some state false?
}

interface Bool3
{
  boolean isTrue();
  boolean isFalse();
  boolean isUnknown();
}
```