Algorithmic Verification

Model Checking

Aidan Farrell

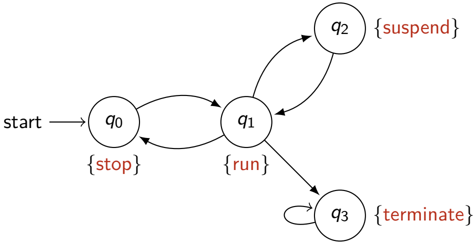
UNSW

August 4, 2020

Outline

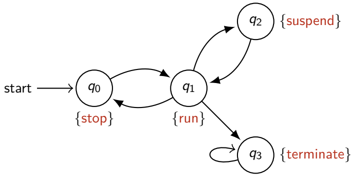
- Review of Kripke Structures
- Structure & Parsing of CTL
- CTL checking example
- CTL checking algorithm

Kripke Structures



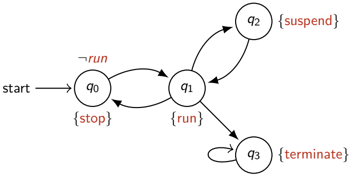
- Represent Processes as states and transitions:
- Each state has an associated set atomic propositions
- stop, run, terminate, suspend

CTL Verification



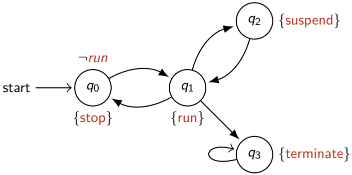
- To verify a property, we express said property as CTL.

CTL Verification



- To verify a property, we express said property as CTL.
- E.g. CTL $\neg run$ is true in at q_0 because q_0 is not in the set of states satisfying run.

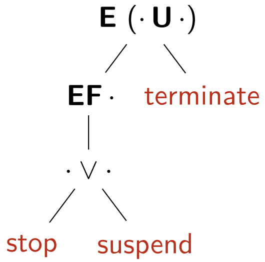
CTL Verification



- To verify a property, we express said property as CTL.
- E.g. CTL $\neg run$ is true in at q_0 because q_0 is not in the set of states satisfying run.
- Similarly for any CTL without timing operators.

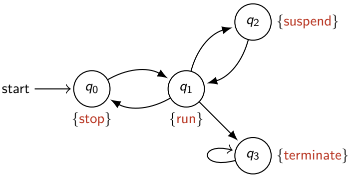
CTL Parsing

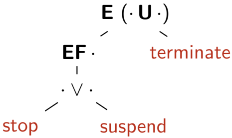
- Atomic propositions as leaf nodes
- Branch nodes are operators.
- E.g. $\mathbf{E}\ (\,\mathbf{EF}\ (\textit{suspend} \vee \textit{stop})\ \mathbf{U}\ \textit{terminate})$ becomes



- Branches are true/false at states
- States with child node true determe states where parent is true.

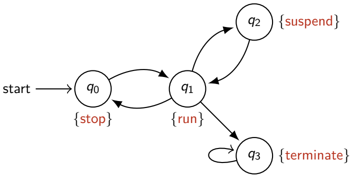
Step by step Verification

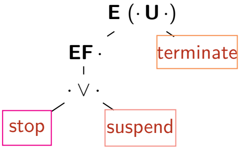




- How do we apply this CTL parse tree to verification?

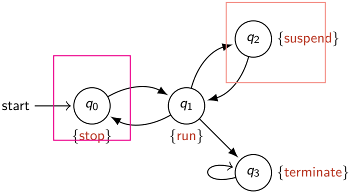
Step by step Verification

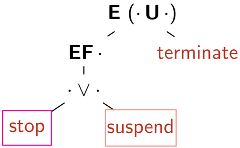




- How do we apply this CTL parse tree to verification?
- Our parse tree makes it easy to verify incrementally.

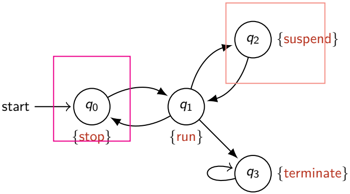
Step by step Verification

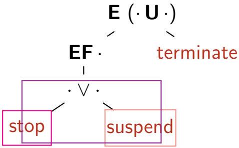




- How do we apply this CTL parse tree to verification?
- Our parse tree makes it easy to verify incrementally.
- Already, states that are *suspend* or *stop* are atomic propositions

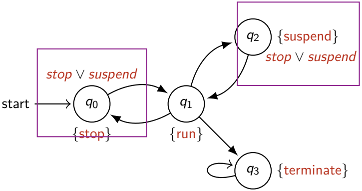
Step by step Verification

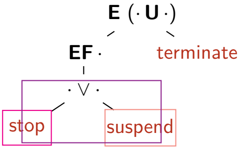




- How do we apply this CTL parse tree to verification?
- Our parse tree makes it easy to verify incrementally.
- Already, states that are *suspend* or *stop* are atomic propositions
- Going up the parse tree to more complicated CTL, can we determine states that are *stop* $\vee$ *suspend*?

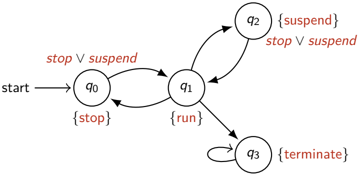
Step by step Verification

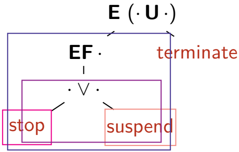




- How do we apply this CTL parse tree to verification?
- Our parse tree makes it easy to verify incrementally.
- Already, states that are *suspend* or *stop* are atomic propositions
- Going up the parse tree to more complicated CTL, can we determine states that are $stop \vee suspend$?
- Union of *stop* states and *suspend* states.

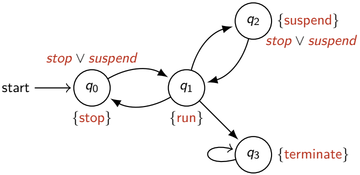
Step by step Verification

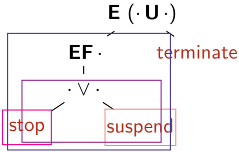




- What about Timing operators? **EF** *stop* ∨ *suspend*

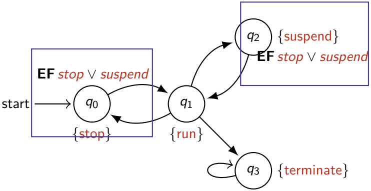
Step by step Verification

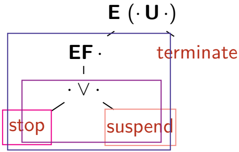




- What about Timing operators? **EF** *stop* $\vee$ *suspend*
- States that have are *stop* $\vee$ *suspend*, obviously eventually get there.

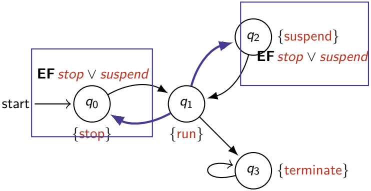
Step by step Verification

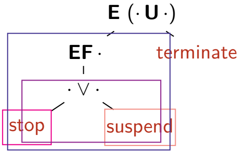




- What about Timing operators? **EF** *stop* ∨ *suspend*
- States that have are *stop* ∨ *suspend*, obviously eventually get there.

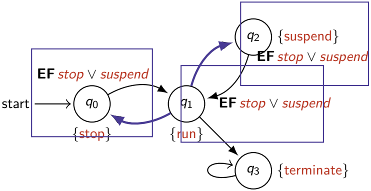
Step by step Verification

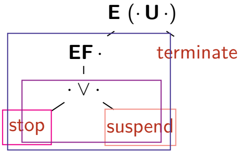




- What about Timing operators? **EF** *stop* ∨ *suspend*
- States that have are *stop* ∨ *suspend*, obviously eventually get there.
- States with any transition to a **EF** *stop* ∨ *suspend* states.

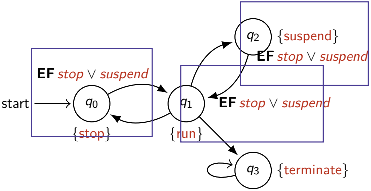
Step by step Verification

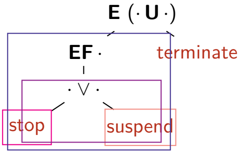




- What about Timing operators? **EF** *stop* ∨ *suspend*
- States that have are *stop* ∨ *suspend*, obviously eventually get there.
- States with any transition to a **EF** *stop* ∨ *suspend* states.

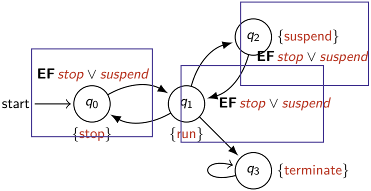
Step by step Verification

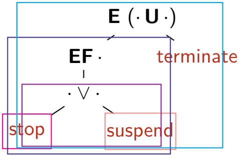




- What about Timing operators? **EF** *stop* ∨ *suspend*
- States that have are *stop* ∨ *suspend*, obviously eventually get there.
- States with any transition to a **EF** *stop* ∨ *suspend* states.
- No more states can transition to those states.

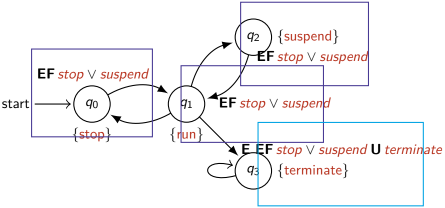
Step by step Verification

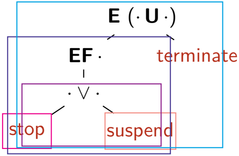




- q_3 satisfies because it is marked *terminate*.

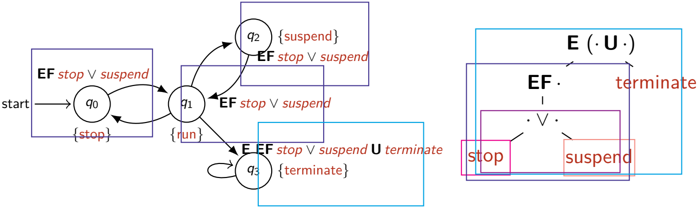
Step by step Verification





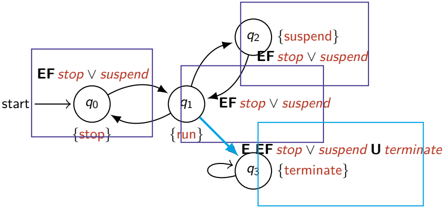
- q_3 satisfies because it is marked *terminate*.

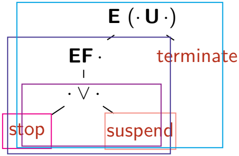
Step by step Verification



- q_3 satisfies because it is marked *terminate*.
- q_2 is ok because
  1. Marked with LHS: **EF** *stop* ∨ *suspend*

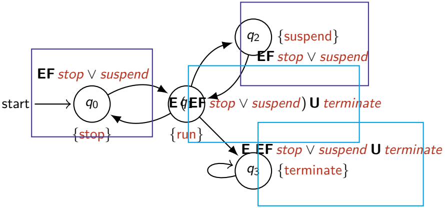
Step by step Verification

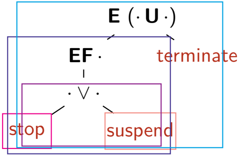




- q_3 satisfies because it is marked *terminate*.
- q_2 is ok because
  1. Marked with LHS: **EF** *stop* ∨ *suspend*
  2. Transitions to an already **E** (· **U** ·) state

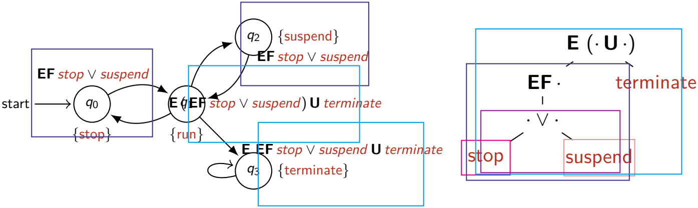
Step by step Verification





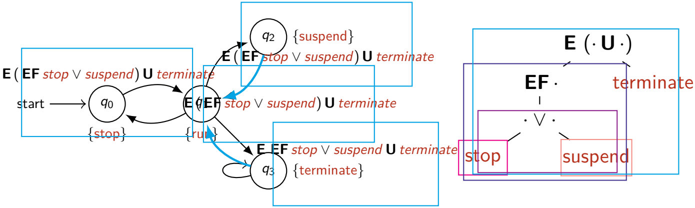
- q_3 satisfies because it is marked *terminate*.
- q_2 is ok because
  1. Marked with LHS: **EF** *stop* ∨ *suspend*
  2. Transitions to an already **E** (· **U** ·) state

Step by step Verification



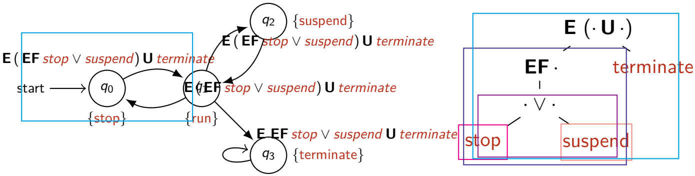
- q_3 satisfies because it is marked *terminate*.
- q_2 is ok because
  1. Marked with LHS: **EF** *stop* ∨ *suspend*
  2. Transitions to an already **E** (· **U** ·) state
- Same for q_0 and q_2

Step by step Verification



- q_3 satisfies because it is marked *terminate*.
- q_2 is ok because
  1. Marked with LHS: **EF** *stop* ∨ *suspend*
  2. Transitions to an already **E** (· **U** ·) state
- Same for q_0 and q_2

Step by step Verification



- q_3 satisfies because it is marked *terminate*.
- q_2 is ok because
  1. Marked with LHS: **EF** *stop* ∨ *suspend*
  2. Transitions to an already **E** (· **U** ·) state
- Same for q_0 and q_2
- q_0 now marked with the CTL we wanted to prove!

Marking algorithm

- Recursively process the CTL Parse tree.
- Input: CTL Parse tree φ, Kripke Structure $Q = \{q_0, q_1, \ldots\}$.
- Output: Set Sat_φ of states where φ is true.

Marking algorithm

- Recursively process the CTL Parse tree.
- Input: CTL Parse tree φ, Kripke Structure $Q = \{q_0, q_1, \ldots\}$.
- Output: Set Sat_φ of states where φ is true.
- $L(q) =$ set of propositions at q. $(q, q') \in edges$ set of edges.

Marking algorithm

- Recursively process the CTL Parse tree.
- Input: CTL Parse tree φ, Kripke Structure $Q = \{q_0, q_1, \ldots\}$.
- Output: Set Sat_φ of states where φ is true.
- $L(q)$ = set of propositions at q. $(q, q') \in edges$ set of edges.

Mark(φ) where φ is an atomic proposition

foreach $q \in Q$ **do**
  if $\varphi \in L(q)$ **then**
    $Sat_\varphi := Sat_\varphi \cup \{q\}$

- Mark depending if in the set of propositions

Marking algorithm

- Recursively process the CTL Parse tree.
- Input: CTL Parse tree φ, Kripke Structure $Q = \{q_0, q_1, \ldots\}$.
- Output: Set Sat_φ of states where φ is true.
- $L(q)$ = set of propositions at q. $(q, q') \in$ *edges* set of edges.

Mark($\varphi = \neg\psi$) boolean negation

$Sat_\psi = Mark(\psi)$;
foreach $q \in Q$ **do**
  if $q \notin Sat_\psi$ **then**
    $Sat_\varphi := Sat_\varphi \cup \{q\}$;

- Run algorithm for un-negated ψ.
- For every state, φ marking is opposite of ψ marking.

Marking algorithm

- Recursively process the CTL Parse tree.
- Input: CTL Parse tree φ, Kripke Structure $Q = \{q_0, q_1, \ldots\}$.
- Output: Set Sat_φ of states where φ is true.
- $L(q)$ = set of propositions at q. $(q, q') \in edges$ set of edges.

Mark($\varphi = \psi_1 \wedge \psi_2$) boolean and/or

$Sat_{\psi_1} := Mark(\psi_1)$;
$Sat_{\psi_2} := Mark(\psi_2)$;
foreach $q \in Q$ **do**
  if $(q \in Sat_{\psi_1}) \wedge (q \in Sat_{\psi_2})$ **then**
    $Sat_\varphi := Sat_\varphi \cup \{q\}$;

- Run algorithm for both ψ_1 and ψ_2.
- Only add to set if state is marked both ψ_1 and ψ_2
- Disjunction $\vee$ is similar.

Marking algorithm

Mark exists until: $\varphi = \mathbf{E}\, \psi_1 \, \mathbf{U} \, \psi_2$

$Sat_{\psi_1} := Mark(\psi_1); Sat_{\psi_2} := Mark(\psi_2)$
foreach $q \in Q$ **do**
  if $q \in Sat_{\psi_2}$ **then**
    $Sat_\varphi := Sat_\varphi \cup \{q\};$
    $V := V \cup \{q\};$
    $W := W \cup \{q\};$

while $W \neq \emptyset$ **do**
  $q :=$ remove from W;
  foreach q', where edge $q' \to q$ **do**
    if $q' \notin V$ **then**
      $V := V \cup \{q'\};$
      if $q' \in Sat_{\psi_1}$ **then**
        $Sat_\varphi := Sat_\varphi \cup \{q'\}; W := W \cup \{q'\};$

- *visit* every node, from ψ_2 back through edges.
- W is the 'frontier' of the φ marked nodes.

Marking algorithm

Mark always until: $\varphi = \mathbf{A}\, \psi_1\, \mathbf{U}\, \psi_2$

$Sat_{\psi_1} := Mark(\psi_1); Sat_{\psi_2} := Mark(\psi_2)$
foreach $q \in Q$ **do**
  counts[q] := # of edges $q \rightarrow$;
  if $q \in Sat_{\psi_2}$ **then**
    $Sat_{\varphi} := Sat_{\varphi} \cup \{q\}$;
    $W := W \cup \{q\}$;

while $W \neq \emptyset$ **do**
  q := remove from W;
  foreach q', where edge $q' \rightarrow q$ **do**
    counts[q'] := counts[q] − 1;
    if counts[q] $= 0 \wedge q' \in Sat_{\psi_1} \wedge q' \notin Sat_{\varphi}$ **then**
      $Sat_{\varphi} := Sat_{\varphi} \cup \{q'\}$;
      $W := W \cup \{q'\}$;

- counts[q] is the # edges not visited yet.
- Instead of *visited*, check *every* outgoing edge, by counting down *numC*.

Marking algorithm

Mark exists globally: $\varphi = \mathbf{EG}\,\psi$

$Sat_\psi := Mark(\psi)$;
$SCC := \{C | C$ is a nontrivial SCC of $Sat_\psi\}$;
$T := \bigcup_{c \in SCC} \{s | s \in C\}$;
$Sat_\varphi := Sat_\varphi \cup T$;
while $T \neq \emptyset$ **do**
  $s :=$ remove from T;
  foreach $q \in Sat_\psi \wedge$ edge $q \rightarrow s$ **do**;
    $T := T \cup \{q\}$;
    $Sat_\varphi := Sat_\varphi \cup \{q\}$;

- Allows for fairness assumption
- (SCC)C (strongly connected components)
- maximal subgraph where every node reachable from every other, in C.
- nontrivial = either > 1 node or self-loop.

Fairness

- Most fairness can be expressed as LTL, but not CTL, because they are path-based.
- Modify the definition of $\forall$ and $\exists$
- Such that interpreted on fair paths

References

[1] [2]

- Christel Baier. *Principles of model checking*. eng. Cambridge, Mass.: MIT Press, 2008. ISBN: 026226756X.
- (Edmund Melson) Clarke Edmund M. *Model checking*. eng. Cambridge, Mass.: MIT Press, 1999. ISBN: 0585385580.