UPPAAL tutorial

- What’s inside UPPAAL
- The UPPAAL input languages

UPPAAL tool

- Developed jointly by Uppsala & Aalborg University
- >>28,000 downloads since 1999
UPPAAL Tool

Architecture of UPPAAL

GUI (Java)
uppaal2k.jar

xml
xla

Server

Engine (C++)

Linux, Windows, Solaris, MacOS
What’s inside UPPAAL

OUTLINE

- Data Structures
  - DBM’s (Difference Bounds Matrices)
  - Canonical and Minimal Constraints
- Algorithms
  - Reachability analysis
  - Liveness checking
- Verification Options
All Operations on Zones
(needed for verification)

- Transformation
  - Conjunction
  - Post condition (delay)
  - Reset
- Consistency Checking
  - Inclusion
  - Emptiness

Zones = Conjuctive constraints

- A zone $Z$ is a conjunctive formula:
  $$g_1 \& g_2 \& \ldots \& g_n$$
  where $g_i$ may be $x_i \sim b_i$ or $x_i-x_j \sim b_{ij}$
- Use a zero-clock $x_0$ (constant 0), we have
  $$\{x_i-x_j \sim b_{ij} \mid \sim \text{ is } < \text{ or } \leq, \ i,j\leq n\}$$
- This can be represented as a MATRIX, DBM
  (Difference Bound Matrices)
Datastructures for Zones in UPPAAL

- Difference Bounded Matrices [Bellman58, Dill89]
- Minimal Constraint Form [RTSS97]
- Clock Difference Diagrams [CAV99]

Canonical Datastructures for Zones

### Difference Bounded Matrices

**Bellman 1958, Dill 1989**

**Inclusion**

**Z1**

\[
\begin{align*}
&x \leq 1 \\
y - x &\leq 2 \\
z - y &\leq 2 \\
z &\leq 9
\end{align*}
\]

**Graph**

```
0 1 x 2
  
  9 y z 2
```

**Z2**

\[
\begin{align*}
&x \leq 2 \\
y - x &\leq 3 \\
y &\leq 3 \\
z - y &\leq 3 \\
z &\leq 7
\end{align*}
\]

**Graph**

```
0 2 x 3
  
  7 z 3
```
Canonical Datastructures for Zones
Difference Bounded Matrices

Bellman 1958, Dill 1989

Inclusion

\[ \begin{align*}
\text{Z1} & : & x & \leq 1 \\
& & y-x & \leq 2 \\
& & z-y & \leq 2 \\
& & z & \leq 9 \\
\text{Z2} & : & x & \leq 2 \\
& & y-x & \leq 3 \\
& & y & \leq 3 \\
& & z-y & \leq 3 \\
& & z & \leq 7
\end{align*} \]

Graph

\[ \begin{align*}
0 & \rightarrow 9 & \rightarrow x & \rightarrow y \\
1 & \rightarrow 2 & \rightarrow z & \rightarrow 0 \\
2 & \rightarrow 7 & \rightarrow z & \rightarrow 0 \\
3 & \rightarrow 6 & \rightarrow z & \rightarrow 0
\end{align*} \]

Shortest Path Closure

\[ \begin{align*}
0 & \rightarrow 9 & \rightarrow x & \rightarrow y \\
1 & \rightarrow 2 & \rightarrow z & \rightarrow 0 \\
2 & \rightarrow 7 & \rightarrow z & \rightarrow 0 \\
3 & \rightarrow 6 & \rightarrow z & \rightarrow 0
\end{align*} \]

Canonical Datastructures for Zones
Difference Bounded Matrices

Bellman 1958, Dill 1989

Emptiness

\[ \begin{align*}
\text{Z} & : & x & \leq 1 \\
& & y & \geq 5 \\
& & y-x & \leq 3 \\
\end{align*} \]

Graph

\[ \begin{align*}
0 & \rightarrow -5 & \rightarrow y \\
1 & \rightarrow 3 & \rightarrow x
\end{align*} \]

Negative Cycle
iff
empty solution set
Canonical Datastructures for Zones

Difference Bounded Matrices

Conjunction

\[ 1 \leq x, 1 \leq y, -2 \leq x-y \leq 3 \]

Add new edge for \( g \)

Delay

\[ 1 \leq x \leq 4, 1 \leq y \leq 3 \]

Shortest Path Closure

Remove upper bounds on clocks

Conjunction

\[ 1 \leq x, 1 \leq y, -2 \leq x-y \leq 3, 3 \leq x \]
Canonical Datastructures for Zones

*Difference Bounded Matrices*

\[
\begin{align*}
1 \leq x, 1 \leq y \\
-2 \leq x - y \leq 3
\end{align*}
\]

Remove all bounds involving \( y \) and set \( y \) to 0

\[
\begin{align*}
1 \leq x, 1 \leq y \\
-2 \leq x - y \leq 3
\end{align*}
\]

Reset

\[
\begin{align*}
1 \leq x, 1 \leq y \\
-2 \leq x - y \leq 3
\end{align*}
\]

\[
\begin{align*}
y = 0, 1 \leq x
\end{align*}
\]

COMPLEXITY

- Computing the shortest path closure, the canonical form of a zone: \( O(n^3) \) [Dijkstra’s alg.]
- Run-time complexity, mostly in \( O(n) \) (when we keep all zones in canonical form)
Datastructures for Zones in UPPAAL

- **Difference Bounded Matrices**  
  [Bellman58, Dill89]

- **Minimal Constraint Form**  
  [RTSS97]

- **Clock Difference Diagrams**  
  [CAV99]

**Minimal Graph**

- Shortest Path Closure \(O(n^3)\)
- Shortest Path Reduction \(O(n^3)\)
- Space worst \(O(n^2)\) practice \(O(n)\)

(Minimal graph, a.k.a. compact data structure)
Graph Reduction Algorithm

1. Equivalence classes based on 0-cycles.

2. Graph based on representatives.
   Safe to remove redundant edges
Graph Reduction Algorithm

1. Equivalence classes based on 0-cycles.
2. Graph based on representatives. Safe to remove redundant edges.
3. **Shortest Path Reduction**
   - One cycle pr. class
   - Removal of redundant edges between classes

Datastructures for Zones in UPPAAL

- **Difference Bounded Matrices** [Bellman58, Dill89]
- **Minimal Constraint Form** [RTSS97]
- **Clock Difference Diagrams** [CAV99]
Other Symbolic Datastructures

- NDD’s Maler et. al.
- CDD’s UPPAAL/CAV99
- DDD’s Møller, Lichtenberg
- Polyhedra HyTech
- ......

Inside the UPPAAL tool

- Data Structures
  - DBM’s (Difference Bounds Matrices)
  - Canonical and Minimal Constraints
- Algorithms
  - Reachability analysis
  - Liveness checking
- Verification Options
Timed CTL in UPPAAL

\[ \text{EF } p \mid \text{AG } p \mid \text{EG } p \mid \text{AF } p \mid p \rightarrow q \]

\[ P ::= A.l \mid g_c \mid g_d \mid \text{not } p \mid p \text{ or } p \mid p \text{ and } p \mid p \text{ imply } p \]

Process
Location
(a location in
automaton A)

Clock
constraint

Predicate
over data variables

p leads to q
denotes
\( \text{AG } (p \implies \text{AF } q) \)

SAFETY PROPERTIES
SAFETY Properties

F ::= EF P | AG P

Reachability

Invariant = ¬ EF ¬ P
Thus, AG P is also checked by reachability analysis!

We have a search problem

Symbolic state
Symbolic transitions

Reachable?
EF ☹
Forward Reachability

\textbf{Init} -> \textbf{Final} ?

\begin{itemize}
  \item \textbf{INITIAL} \quad \textbf{Passed} := \emptyset;
  \quad \textbf{Waiting} := \{(n0,Z0)\}
  \item \textbf{REPEAT}
    \begin{itemize}
      \item pick \((n,Z)\) in \textbf{Waiting}
      \item if for some \(Z' \supseteq Z\) \((n,Z')\) in \textbf{Passed} then STOP
      \item else /\textbf{explore}/ add 
        \begin{itemize}
          \item \{(m,U) : (n,Z) => (m,U) \}
        \end{itemize}
        to \textbf{Waiting};
        Add \((n,Z)\) to \textbf{Passed}
    \end{itemize}
  \item \textbf{UNTIL} \quad \textbf{Waiting} = \emptyset
    or
    Final is in \textbf{Waiting}
\end{itemize}
Forward Reachability

Init -> Final?

INITIAL Passed := Ø;
Waiting := \{(n_0, Z_0)\}

REPEAT
- pick (n, Z) in Waiting
- if for some Z' \in Z
  (n, Z') in Passed then STOP
  else /explore/ add
  \{(m, U) : (n, Z) \Rightarrow (m, U)\} to Waiting
  Add (n, Z) to Passed
UNTIL Waiting = Ø
or Final is in Waiting

Forward Reachability

Init -> Final?

INITIAL Passed := Ø;
Waiting := \{(n_0, Z_0)\}

REPEAT
- pick (n, Z) in Waiting
- if for some Z' \in Z
  (n, Z') in Passed then STOP
  else /explore/ add
  \{(m, U) : (n, Z) \Rightarrow (m, U)\} to Waiting
  Add (n, Z) to Passed
UNTIL Waiting = Ø
or Final is in Waiting
Further question

Can we find the path with shortest delay, leading to P? (i.e. a state satisfying P)

OBSERVATION:
Many scheduling problems can be phrased naturally as reachability problems for timed automata.
Verification vs. Optimization

- **Verification Algorithms:**
  - Checks a logical property of the entire state-space of a model.
  - Efficient Blind search.

- **Optimization Algorithms:**
  - Finds (near) optimal solutions.
  - Uses techniques to avoid non-optimal parts of the state-space (e.g. Branch and Bound).

- **Goal:** solve opt. problems with verification.

**OPTIMAL REACHABILITY**

The maximal and minimal delay problem
Find the trace leading to P with \text{min} delay

There may be a lot of pathes leading to P

Which one with the shortest delay?

Idea: delay as “Cost” to reach a state, thus cost increases with time at rate 1
An Simple Algorithm for minimal-cost reachability

- State-Space Exploration + Use of global variable Cost and global clock δ
- Update Cost whenever goal state with min( C ) < Cost is found:

Terminates when entire state-space is explored.

Problem: The search may never terminate!

Example (min delay to reach G)

The minimal delay = 0 but the search may never terminate!

Problem: How to symbolically represent the zone C.
Priced-Zone

- Cost = minimal total time
- \( C \) can be represented as the zone \( Z^\delta \), where:
  - \( Z^\delta \) original (ordinary) DBM plus...
  - \( \delta \) clock keeping track of the cost/time.
- Delay, Reset, Conjunction etc. on \( Z \) are the standard DBM-operations
- Delay-Cost is incremented by Delay-operation on \( Z^\delta \).

Then: \( C_3 \subseteq C_2 \subseteq C_1 \)

But: \( C_3 \nsubseteq C_2 \nsubseteq C_1 \)
Solution: $(\cdot)^\dagger$-widening operation

- $(\cdot)^\dagger$ removes upper bound on the $\delta$-clock:
  \[
  C_3 \subseteq C_2 \subseteq C_1
  
  C_3^\dagger \subseteq C_2^\dagger \subseteq C_1^\dagger
  \]

- In the Algorithm:
  - $\text{Delay}(C^\dagger) = (\text{Delay}(C^\dagger))^\dagger$
  - $\text{Reset}(x,C^\dagger) = (\text{Reset}(x,C^\dagger))^\dagger$
  - $C_1^\dagger \land g = (C_1^\dagger \land g)^\dagger$

- It is suffices to apply $(\cdot)^\dagger$ to the initial state $(l_0,C_0)$.

Example (widening for Min)

- $Z_1 \nsubseteq Z_2$
- $Z_1 \nsubseteq Z_2$
Example (widening for Min)

\[ \delta \]

\[ Z_1 \subseteq Z_2 \]

\[ Z^+ = \text{Widen}(Z) \]

Example (widening for Min)

\[ \delta \]

\[ Z_1 \subseteq Z_2 \]

\[ Z^+ = \text{Widen}(Z) \]

\[ Z_1 \subseteq Z_2 \]

\[ Z_1 \subseteq Z_2 \]
An Algorithm (Min)

Cost:=∞, Pass := {}, Wait := {(l_0,C_0)}
while Wait ≠ {} do
    select (l,C) from Wait
    if (l,C) = P and Min(C)<Cost then Cost:= Min(C)
    if (l,C) ⊆ (l,C') for some (l,C') in Pass then skip
    otherwise add (l,C) to Pass
    and for all (m,C') such that (l,C) ⊆ (m,C'):
        add (m,C') to Wait
Return Cost

Output: Cost = the min cost of a found trace satisfying P.

Further reading: Priced Timed Automata [Larsen et al]

- Timed Automata + Costs on transitions and locations.
- Uniformly Priced = Same cost in all locations (edges may have different costs).
- Cost of performing transition: Transition cost.
- Cost of performing delay d: (d x location cost).
**Priced Timed Automata**

![Timed Automata Diagram]

**Trace:**
\[(a, x=y=0) \xrightarrow{4} (b, x=y=0) \xrightarrow{(2.5) \times 2} (b, x=y=2.5) \xrightarrow{0} (a, x=0, y=2.5)\]

**Cost of Execution Trace:**
Sum of costs: \[4 + 5 + 0 = 9\]

**Problem:** Finding the minimum cost of reaching \(c\)!

---

**Inside the UPPAAL tool**

- Data Structures
  - DBM’s (Difference Bounds Matrices)
  - Canonical and Minimal Constraints
- Algorithms
  - Reachability analysis
  - Liveness checking
- Verification Options
Timed CTL in UPPAAL

Process Location
(a location in automaton A)

Clock constraint

Predicate over data variables

SAFETY PROPERTIES

LIVENESS PROPERTIES

EF p | AG p | EG p | AF p | p → q

P ::= A.l | g_c | g_d | not p | p or p | p and p | p imply p

p leads to q
denotes
AG (p imply AF q)

LIVENESS Properties

F ::= EG p | AF p | p → q

Possibly always P
is equivalent to (AF : P)

Eventually P
is equivalent to (EG : P)

P leads to Q
is equivalent to AG (P imply AF Q)
Algorithm for checking $\text{AF } P$  

Eventually $P$  

Bouajjani, Tripakis, Yovine’97  
On-the-fly symbolic model checking of TCTL

---

**Question**

$\text{AF } P$  

"$P$ will be true for sure in future"

Does this automaton satisfy $\text{AF } P$
Note that

\[ \text{AF } P \]

"\text{P will be true for sure in future}"

\[ x \leq 5 \]

\[ m \]

\[ p \]

NO !!!! there is a path:

\( (m, x=0) \rightarrow (m, x=1) \rightarrow (m, 2) \ldots (m, x=k) \ldots \)

Idling forever in location \( m \)

Note that

\[ \text{AF } P \]

"\text{P will be true for sure in future}"

\[ x \leq 5 \]

\[ m \]

\[ p \]

This automaton satisfies \( \text{AF } P \)
Liveness Algorithm

\[
\text{proc } \text{Eventually}(S_0, \varphi) \equiv \\
\quad ST := \emptyset \\
\quad Passed := \emptyset \\
\quad \text{Search}(\text{delay}(S_0, \neg \varphi)) \\
\quad \text{exit}(\text{true}) \\
\text{end}
\]

\[
\text{proc } \text{Search}(S) \equiv \\
\quad \text{if loop}(S, ST) \text{ then } \text{exit}(\text{false}) \text{ fi} \\
\quad S := S \land \neg \varphi \\
\quad \text{push}(ST, S) \\
\quad \text{if unbounded}(S) \lor \text{deadlocked}(S) \text{ then } \\
\quad \quad \text{exit}(\text{false}) \text{ fi} \\
\quad \text{if } \forall S' \in \text{Passed} : S \not\subseteq S' \text{ then } \\
\quad \quad \text{foreach } S' : S \Rightarrow S' \text{ do} \\
\quad \quad \quad \text{Search}(\text{delay}(S', \neg \varphi)) \\
\quad \quad \text{od} \\
\quad \text{fi} \\
\quad \text{Passed := Passed } \cup \{\text{pop}(ST)\} \\
\text{end}
\]
Liveness Algorithm

\begin{verbatim}
proc Eventually((S₀, φ) ≡
ST := ∅
Passed := ∅
Search(delay(S₀, ¬φ))
exit(true)
end
proc Search(S) ≡
if loop(S, ST) then exit(false) fi
S := S ∧ ¬φ
push(ST, S)
if unbounded(S) ∨ deadlocked(S) then
exit(false) fi
if ∀S' ∈ Passed : S \nsubseteq S'
then foreach S' : S \nRightarrow S' do
Search(delay(S', ¬φ))
end
fi
Passed := Passed ∪ {pop(ST)}
end
\end{verbatim}
Liveness Algorithm

\[\text{proc } \text{Eventually}(S_0, \varphi) \equiv\]
\[ST := \emptyset\]
\[Passed := \emptyset\]
\[\text{Search}(\text{delay}(S_0, \neg\varphi))\]
\[\text{exit}(\text{true})\]
\[\text{end}\]
\[\text{proc } \text{Search}(S) \equiv\]
\[\text{if empty}(S) \text{ then } \text{exit}(\text{true}) \text{ fi}\]
\[\text{if } \text{loop}(S, ST) \text{ then } \text{exit}(\text{false}) \text{ fi}\]
\[S := S \land \neg\varphi\]
\[\text{push}(ST, S)\]
\[\text{if } \text{unbounded}(S) \lor \text{deadlocked}(S) \text{ then } \text{exit}(\text{false}) \text{ fi}\]
\[\bullet \text{ if } \forall S' \in \text{Passed} : S \not\preceq S'\]
\[\text{then foreach } S' : S \Rightarrow S' \text{ do }\]
\[\text{Search}(\text{delay}(S', \neg\varphi))\]
\[\text{od}\]
\[\text{fi}\]
\[\text{Passed} := \text{Passed} \cup \{\text{pop}(ST)\}\]
\[\text{end}\]
Liveness Algorithm

\begin{verbatim}
proc Eventually(S₀, φ) ≡
ST := ∅
Passed := ∅
Search(delay(S₀, ¬φ))
exit(true)
end
proc Search(S) ≡
if loop(S, ST) then exit(false) fi
S := S ∨ ¬φ
push(ST, S)
if unbounded(S) ∨ deadlocked(S) then
  exit(false) fi
if ∀S' ∈ Passed : S ⊊ S'
  then foreach S' : S ⊨ S' do
    Search(delay(S', ¬φ))
  od
fi
Passed := Passed ∪ {pop(ST)}
end
\end{verbatim}
**Liveness Algorithm**

```plaintext
proc Eventually(S₀, φ) ≡
  ST := ∅
  Passed := ∅
  Search(delay(S₀, ¬φ))
  exit(true)
end

diagram

proc Search(S) ≡
  if loop(S, ST) then exit(false) fi
  S := S ∧ ¬φ
  push(ST, S)
  if unbounded(S) ∨ deadlocked(S) then exit(false) fi
  if ∀S′ ∈ Passed : S ⊋ S′
    then foreach S′ : S ⇒ S′ do
      Search(delay(S′, ¬φ))
    od
  fi
  Passed := Passed ∪ {pop(ST)}
end
```

**Liveness Algorithm**

```plaintext
proc Eventually(S₀, φ) ≡
  ST := ∅
  Passed := ∅
  Search(delay(S₀, ¬φ))
  exit(true)
end

diagram

proc Search(S) ≡
  if loop(S, ST) then exit(false) fi
  S := S ∧ ¬φ
  push(ST, S)
  if unbounded(S) ∨ deadlocked(S) then exit(false) fi
  if ∀S′ ∈ Passed : S ⊋ S′
    then foreach S′ : S ⇒ S′ do
      Search(delay(S′, ¬φ))
    od
  fi
  Passed := Passed ∪ {pop(ST)}
end
```
Question: Time bound synthesis

\( \text{AF P} \) "P will be true eventually"
But no time bound is given.

Assume AF P is satisfied by an automaton A.
Can we calculate the Max time bound?

OBS: we know how to calculate the Min!

Assume AF P is satisfied

Find the trace leading to P with the max delay

Almost the same algorithm as for synthesizing Min

We need to explore the Green part
An Algorithm (Max)

\[
\text{Cost} := 0, \ \text{Pass} := \{\}, \ \text{Wait} := \{(l_0, C_0)\}
\]

while Wait \(\neq\) {} do

\[
\text{select} (l, C) \text{ from } \text{Wait}
\]

\[
\text{if } (l, C) = P \text{ and } \text{Max}(C) > \text{Cost} \text{ then } \text{Cost} := \text{Max}(C)
\]

else if \(\forall (l, C') \text{ in } \text{Pass}: \ C \not\subseteq C'\) then

\[
\text{add} (l, C) \text{ to } \text{Pass}
\]

\[
\text{forall} (m, C') \text{ such that } (l, C) \rightarrow (m, C'):
\]

\[
\text{add} (m, C') \text{ to } \text{Wait}
\]

Return Cost

\textbf{Output:} Cost = the min cost of a found trace satisfying P.

\textbf{BUT:} \(\subseteq\) is defined on zones where the lower bound of "cost" is removed

Zone-Widening operation for Max

\[\delta\]

\[\begin{array}{c}
C_1 \\
C_2
\end{array}\]

\[C_1 \not\subseteq C_2\]
Zone-Widening operation for Max

\[
\delta \\
C^1 + \delta \subseteq C^2 + \delta
\]

Inside the UPPAAL tool

- Data Structures
  - DBM’s (Difference Bounds Matrices)
  - Canonical and Minimal Constraints
- Algorithms
  - Reachability analysis
  - Liveness checking
  - Termination
- Verification Options
Verification Options

- Diagnostic Trace
- Breadth-First
- Depth-First
- Local Reduction
- Active-Clock Reduction
- Global Reduction
- Re-Use State-Space
- Over-Approximation
- Under-Approximation

Inactive (passive) Clock Reduction

- $x$ is only **active** in location $S_1$

**Definition**

$x$ is **inactive** at $S$ if on all path from $S$, $x$ is always reset before being tested.
Global Reduction
(When to store symbolic state)

No Cycles: Passed list not needed for termination

However, Passed list useful for efficiency

Global Reduction [RTSS97]
(When to store symbolic state)

Cycles:
Only symbolic states involving loop-entry points need to be saved on Passed list
To Store Or Not To Store?

- **117 states**
- **81 states** entrypoint
- **9 states**

**Time OH less than 10%**
(need to re-explore some states)

---

Reuse of State Space

- **Waiting**
  - prop1
  - prop2
  - prop3
  - prop4
  - prop5

- **Passed**
  - A[] prop1
  - A[] prop2
  - A[] prop3
  - A[] prop4
  - A[] prop5
  - .
  - .
  - A[] propn

Search in existing **Passed** list before continuing search

Which order to search?
Reuse of State Space

Search in existing Passed list before continuing search

Which order to search?

Swapped to secondary memory
Reuse of State Space

A[] prop1
A[] prop2
A[] prop3
A[] prop4
A[] prop5

Search in existing Passed list before continuing search
Which order to search?

Under-approximation

Bitstate Hashing (Holzman, SPIN)
Under-approximation

Bitstate Hashing

**Passes**

- Under-approximation
- Bitstate Hashing

- Passed
- Waiting
- Init
- n,Z
- m,U
- Final

Hashfunction

\[ F(n, Z) = 1 \]

Passed

- Passed = Bitarray

UPPAAL

- 8 Mbits

**Bit-state Hashing**

INITIAL

\[ \text{Passed} := \emptyset; \]
\[ \text{Waiting} := \{ (n0, Z0) \} \]

REPEAT

- pick \( (n, Z) \) in Waiting
- if for some \( Z' \neq Z \) \( (n, Z') \) in Passed then STOP
- else explore\:
  \[ \{ (m, U) : (n, Z) \Rightarrow (m, U) \} \]
  add to Waiting:
  add \( (n, Z) \) to Passed

UNTIL

\[ \text{Waiting} = \emptyset \]
or
Final is in Waiting

\[ \text{Passed}(F(n, Z)) = 1 \]

\[ \text{Passed}(F(n, Z)) := 1 \]
Under Approximation
(good for finding Bugs quickly, debugging)

- Possitive answer is safe (you can trust)
  - You can trust your tool if it tells: a state is reachable (it means Reachable!)
- Negative answer is Inconclusive
  - You should not trust your tool if it tells: a state is non-reachable
  - Some of the branch may be terminated by conflict (the same hashing value of two states)

Over-approximation

Convex Hull
Over-Approximation
(good for safety property-checking)

- **Possitive answer is Inconclusive**
  - a state is reachable means Nothing (you should not trust your tool when it says so)
  - Some of the transitions may be enabled by Enlarged zones

- **Negative answer is safe**
  - a state is not reachable means Non-reachable (you can trust your tool when it says so)