

Using Coq and Recurrent Neural Network to Model and Verify Connectors

Meng Sun

Department of Informatics, School of Mathematical Sciences, Peking University
<http://www.math.pku.edu.cn/teachers/sunm>

joint work with
Yi Li, Xiyue Zhang, Weijiang Hong and M. Saqib Nawaz

Departamento de Informática, Universidade do Minho
2018.2.7

Outline

- ① **Introduction**
- ② Connectors as Designs
- ③ Reasoning about Connectors in Coq
- ④ Generating Tactic Suggestions with Neural Network
- ⑤ Conclusion

Introduction

- Connector is a key concept in coordination languages like Reo, and playing an important role in complex component-based systems.
- The reliability of component-based systems highly depends on the correctness of connectors.
- Formal semantics allows us to specify and analyze the behavior of connectors precisely.
- The increasing growth in size and complexity of computing infrastructure has made the modeling and verification of connectors properties a more difficult and challenging task.

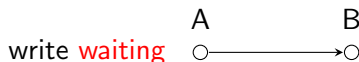
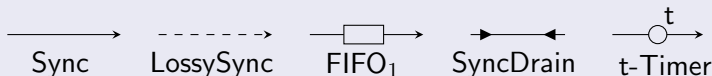
Introduction

- We provided an approach for formal modeling and reasoning about Reo connectors under the UTP semantic framework in the proof assistant Coq.
- We developed a RNN-based approach to generate suggested tactics automatically, which leads to better performance than traditional learning algorithms for proof guidance.

The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

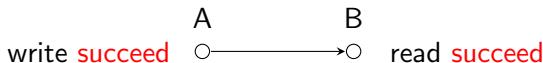
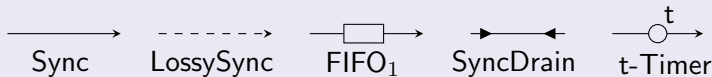
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

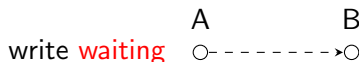
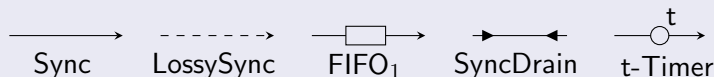
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

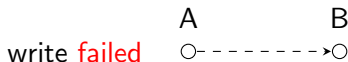
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

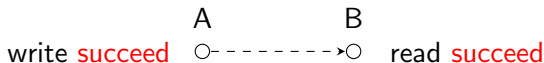
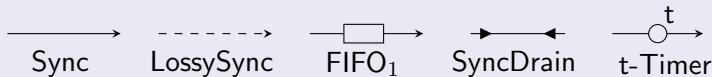
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

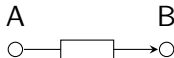
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

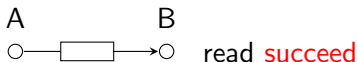
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

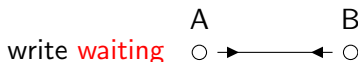
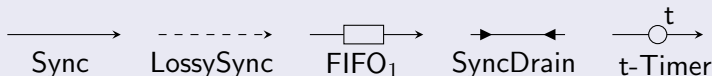
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

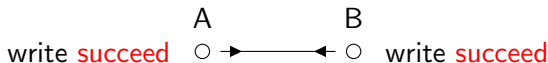
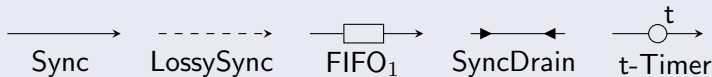
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

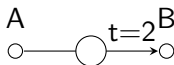
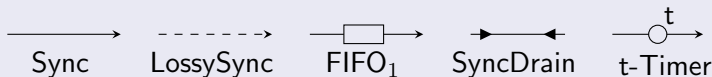
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

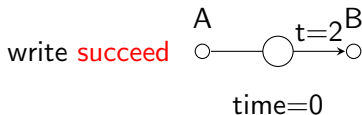
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

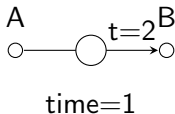
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

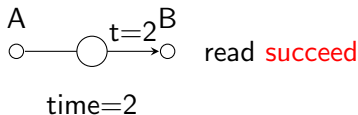
Basic Channels



The Coordination Language Reo

- Reo is a coordination language in which complex connectors are compositionally constructed from primitive **channels**.
- Components are dynamically connected to and perform I/O operations through connectors.

Basic Channels

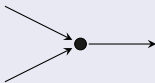


The Coordination Language Reo

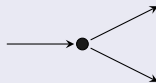
3 Types of Nodes



Flow through



Merge



Replicate

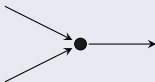
write **waiting** 

The Coordination Language Reo

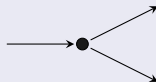
3 Types of Nodes




Flow through



Merge



Replicate

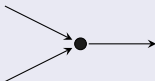
write **succeed**  read **succeed**

The Coordination Language Reo

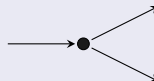
3 Types of Nodes



Flow through



Merge



Replicate

write **waiting**



write **waiting**

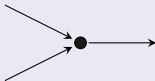


The Coordination Language Reo

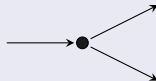
3 Types of Nodes



Flow through



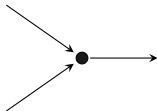
Merge



Replicate

write **succeed**

write **waiting**



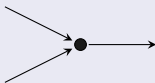
read **succeed**

The Coordination Language Reo

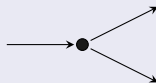
3 Types of Nodes



Flow through

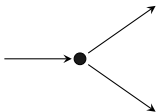


Merge



Replicate

write waiting

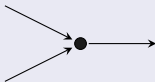


The Coordination Language Reo

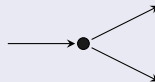
3 Types of Nodes



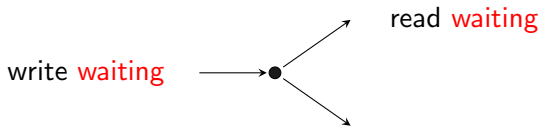
Flow through



Merge



Replicate

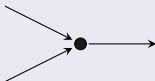


The Coordination Language Reo

3 Types of Nodes



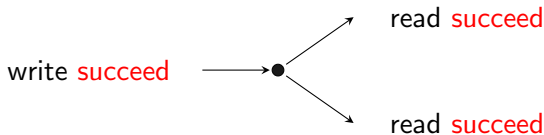
Flow through



Merge



Replicate

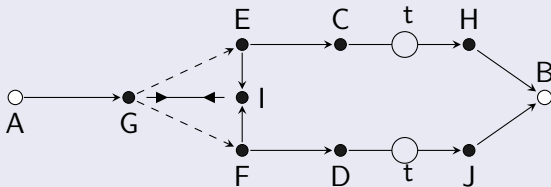


The Coordination Language Reo

The core of Reo is to build complicated connectors through simpler ones.

$2 \times t$ timer

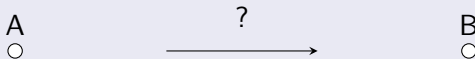
A connector $\xrightarrow{n \times t}$ can produce a timeout after a delay t for every input (for up to n such inputs), even if the inter-arrival time of the inputs is less than t . It can be built by using n timer channels \xrightarrow{t} and an exclusive router (with n sink nodes).



The Coordination Language Reo

Timed FIFO₂

The timed FIFO _{n} connector can be used to model a real-time network, which delays every input for t time units, even if the inter-arrival time of the inputs is less than t (for up to n such inputs). Such a connector can not be obtained by just composing n timed FIFO1. However, it is still easy to be constructed by using $\xrightarrow{n \times t}$.



The Coordination Language Reo

Timed FIFO₂

The timed FIFO _{n} connector can be used to model a real-time network, which delays every input for t time units, even if the inter-arrival time of the inputs is less than t (for up to n such inputs). Such a connector can not be obtained by just composing n timed FIFO1. However, it is still easy to be constructed by using $\xrightarrow{n \times t} \bullet \rightarrow$.

A
○

D
●

E
●

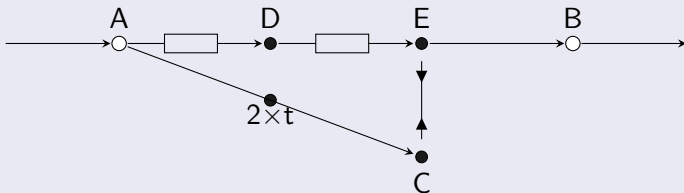
B
○

●
C

The Coordination Language Reo

Timed FIFO₂

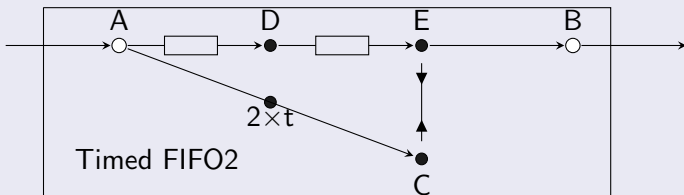
The timed FIFO _{n} connector can be used to model a real-time network, which delays every input for t time units, even if the inter-arrival time of the inputs is less than t (for up to n such inputs). Such a connector can not be obtained by just composing n timed FIFO1. However, it is still easy to be constructed by using $\xrightarrow{n \times t} \bullet \rightarrow$.



The Coordination Language Reo

Timed FIFO₂

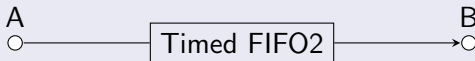
The timed FIFO_{*n*} connector can be used to model a real-time network, which delays every input for *t* time units, even if the inter-arrival time of the inputs is less than *t* (for up to *n* such inputs). Such a connector can not be obtained by just composing *n* timed FIFO1. However, it is still easy to be constructed by using $\xrightarrow{n \times t}$.



The Coordination Language Reo

Timed FIFO₂

The timed FIFO _{n} connector can be used to model a real-time network, which delays every input for t time units, even if the inter-arrival time of the inputs is less than t (for up to n such inputs). Such a connector can not be obtained by just composing n timed FIFO1. However, it is still easy to be constructed by using $\xrightarrow{n \times t}$.



Outline

- 1 Introduction
- 2 **Connectors as Designs**
- 3 Reasoning about Connectors in Coq
- 4 Generating Tactic Suggestions with Neural Network
- 5 Conclusion

Timed Data Sequences

- For an arbitrary connector \mathbf{R} , the relevant observations come in pairs, with one observation on the source nodes of \mathbf{R} , and one observation on the sink nodes of \mathbf{R} . For every node N , the corresponding observation on N is given by a timed data sequence.
- Let $TS = \{a \in \mathbb{R}_+^* \mid \forall 0 \leq n < |a|. a(n) < a(n+1)\}$ and $DS = D^*$ be the set of time sequences and data sequences, the set of timed data sequences is defined by $TDS \subseteq DS \times TS$ that contains pairs $\langle \alpha, a \rangle$ consisting of $\alpha \in DS$ and $a \in TS$ with $|\alpha| = |a|$.
- Timed data sequences can be alternatively and equivalently defined as (a subset of) $(D \times \mathbb{R}_+)^*$ because of the existence of isomorphism

$$\langle \alpha, a \rangle \mapsto (\langle \alpha(0), a(0) \rangle, \langle \alpha(1), a(1) \rangle, \langle \alpha(2), a(2) \rangle, \dots)$$

- For connectors, a design $P \vdash Q$ has the following meaning:

$$P \vdash Q =_{df} (ok \wedge P \Rightarrow ok' \wedge Q)$$

where ok and ok' are two variables being used to analyze explicitly the phenomena of communication initialization and termination.

- The variable ok stands for a successful initialization and the start of a communication. When ok is **false**, the communication has not started, so no observation can be made.
- The variable ok' denotes the observation that the communication has terminated. The communication is divergent when ok' is **false**.

Connectors as Designs

- To specify input and output explicitly, for a connector \mathbf{R} , we use $in_{\mathbf{R}}$ and $out_{\mathbf{R}}$ for the lists of timed data sequences on the input ends and output ends of \mathbf{R} respectively.
- every connector \mathbf{R} can be represented by the design $P(in_{\mathbf{R}}) \vdash Q(in_{\mathbf{R}}, out_{\mathbf{R}})$ as follows:

$\mathbf{con} : \mathbf{R}(in : in_{\mathbf{R}}; out : out_{\mathbf{R}})$

$\mathbf{in} : P(in_{\mathbf{R}})$

$\mathbf{out} : Q(in_{\mathbf{R}}, out_{\mathbf{R}})$

- $P(in_{\mathbf{R}})$ is the precondition that should be satisfied by inputs $in_{\mathbf{R}}$ on the source nodes of \mathbf{R} , and
- $Q(in_{\mathbf{R}}, out_{\mathbf{R}})$ is the postcondition that should be satisfied by outputs $out_{\mathbf{R}}$ on the sink nodes of \mathbf{R} .
- A predicate \mathcal{D} is used to denote well-defined timed data sequence types. In other words, we define the behavior only for valid sequences expressed via a predicate \mathcal{D} .

Channels

- Synchronous channel:

con : **Sync**($in : (A \mapsto \langle \alpha, a \rangle)$; $out : (B \mapsto \langle \beta, b \rangle)$)

in : $\mathcal{D}\langle \alpha, a \rangle$

out : $\mathcal{D}\langle \beta, b \rangle \wedge \beta = \alpha \wedge b = a$

- FIFO1 channel:

con : **FIFO1**($in : (A \mapsto \langle \alpha, a \rangle)$; $out : (B \mapsto \langle \beta, b \rangle)$)

in : $\mathcal{D}\langle \alpha, a \rangle$

out : $\mathcal{D}\langle \beta, b \rangle \wedge \beta = \alpha \wedge a < b \wedge b < a'$

- Synchronous drain:

con : **SyncDrain**($in : (A \mapsto \langle \alpha, a \rangle, B \mapsto \langle \beta, b \rangle)$; $out : ()$)

in : $\mathcal{D}\langle \alpha, a \rangle \wedge \mathcal{D}\langle \beta, b \rangle \wedge a = b$

out : **true**

Channels

- t -Timer:

con : **Timer** $[t](in : (A \mapsto \langle \alpha, a \rangle); out : (B \mapsto \langle \beta, b \rangle))$

in : $\mathcal{D}\langle \alpha, a \rangle \wedge a[+t] \leq a'$

out : $\mathcal{D}\langle \beta, b \rangle \wedge \beta \in \{timeout\}^* \wedge b = a[+t]$

- Synchronous channel:

con : **LossySync** $(in : (A \mapsto \langle \alpha, a \rangle); out : (B \mapsto \langle \beta, b \rangle))$

in : $\mathcal{D}\langle \alpha, a \rangle$

out : $\mathcal{D}\langle \beta, b \rangle \wedge L(\langle \alpha, a \rangle, \langle \beta, b \rangle)$

where

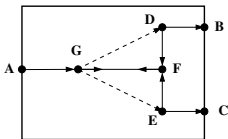
$$\begin{aligned} & L(\langle \alpha, a \rangle, \langle \beta, b \rangle) \\ \equiv & (\beta = () \wedge b = ()) \vee (a(0) < b(0) \wedge L(\langle \alpha', a' \rangle, \langle \beta, b \rangle)) \vee \\ & (a(0) = b(0) \wedge \alpha(0) = \beta(0) \wedge L(\langle \alpha', a' \rangle, \langle \beta', b' \rangle)) \end{aligned}$$

Composition

- Flow-through and replicating operations are simple.
- For merging operation on two sink nodes, we have the ternary relation M is defined as

$$M(\langle \gamma_1, c_1 \rangle, \langle \gamma_2, c_2 \rangle, \langle \gamma, c \rangle) \\ = \left\{ \begin{array}{ll} \langle \gamma, c \rangle = \langle \gamma_1, c_1 \rangle & \text{if } |\langle \gamma_2, c_2 \rangle| = 0 \\ \langle \gamma, c \rangle = \langle \gamma_2, c_2 \rangle & \text{if } |\langle \gamma_1, c_1 \rangle| = 0 \\ c_1(0) \neq c_2(0) \wedge \\ \left\{ \begin{array}{l} \gamma(0) = \gamma_1(0) \wedge c(0) = c_1(0) \wedge \\ M(\langle \gamma'_1, c'_1 \rangle, \langle \gamma_2, c_2 \rangle, \langle \gamma', c' \rangle) \text{ if } c_1(0) < c_2(0) \\ \gamma(0) = \gamma_2(0) \wedge c(0) = c_2(0) \wedge \\ M(\langle \gamma_1, c_1 \rangle, \langle \gamma'_2, c'_2 \rangle, \langle \gamma', c' \rangle) \text{ if } c_2(0) < c_1(0) \end{array} \right. \\ \text{otherwise} \end{array} \right.$$

Exclusive Router



con : **EXRouter**(*in* : ($\langle \alpha, a \rangle$); *out* : ($\langle \beta, b \rangle, \langle \gamma, c \rangle$))

in : $\mathcal{D}\langle \alpha, a \rangle$

out : $\mathcal{D}\langle \beta, b \rangle \wedge \mathcal{D}\langle \gamma, c \rangle \wedge L(\langle \alpha, a \rangle, \langle \beta, b \rangle) \wedge L(\langle \alpha, a \rangle, \langle \gamma, c \rangle) \wedge$
 $M(\langle \beta, b \rangle, \langle \gamma, c \rangle, \langle \alpha, a \rangle)$

Timed FIFO_n

con : **timedFIFO**_n(*in* : ($A \mapsto \langle \alpha, a \rangle$); *out* : $B \mapsto \langle \beta, b \rangle$)
 in : $\mathcal{D}\langle \alpha, a \rangle$
 out : $\mathcal{D}\langle \beta, b \rangle \wedge \beta = \alpha \wedge b = a[+t] \wedge b < a^{(n)}$

Outline

- 1 Introduction
- 2 Connectors as Designs
- 3 **Reasoning about Connectors in Coq**
- 4 Generating Tactic Suggestions with Neural Network
- 5 Conclusion

The Coq Proof Assistant

Coq is a formal **proof management system**. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs.

theorem represented as **type**

proof a **well-constructed term** of certain type (what we want to prove)

hypothesis an **abstract term** of certain type

tactic wizards to construct the final proof step by step

```
Parameters A B : Type.  
Hypothesis H1 : A.  
Hypothesis H2 : A -> B.  
Goal B. Proof.  
  apply H2. assumption.  
Qed.
```

Channels in Coq

- Predicates are used to describe the constraints on time and data for timed channels in Coq, and such predicates can be combined (with intersection) together to provide the complete specification of channels.
- For example, the following definitions specify the design models for the synchronous and t -timer channels in Coq:

```
Definition Sync (Input Output:Stream TD) : Prop :=  
  Teq Input Output /\ Deq Input Output.
```

```
Parameter timeout: Data
```

```
Definition Timert (Input Output: Stream TD)(t: Time): Prop:=  
  (forall n:nat,  
    PrL(Str_nth n Input) + t < PrL(Str_nth n (tl Input)))  
  /\ Teqt Input Output t  
  /\ forall n:nat, PrR (Str_nth n Output) = timeout
```

Composition Operators in Coq

- We need not to give the flow-through and replicate operators specific definitions since they can be achieved implicitly or by simple renaming.
 - For example, while we specify two channels $Sync(A, B)$ and $FIFO1(B, C)$, the flow-through operator that acts on node B for these two channels has been achieved.
- When the merge operator acts on two channels AB and CD , it leads to a choice of data items being taken from AB or CD . We define merge as the intersection of two predicates and use the recursive definition:

```
Parameter merge:Stream TD -> Stream TD -> Stream TD -> Prop.  
Axiom merge_coind:  
  forall s1 s2 s3:Stream TD,  
    merge s1 s2 s3->( ~(PrL(hd s1) = PrL(hd s2))  
  /\ (((PrL(hd s1) < PrL(hd s2)) ->  
    ((hd s3 = hd s1) /\ merge (tl s1) s2 (tl s3)))  
  /\ ((PrL(hd s1) > PrL(hd s2)) ->  
    ((hd s3 = hd s2) /\ merge s1 (tl s2) (tl s3))))).
```

Composition of Connectors

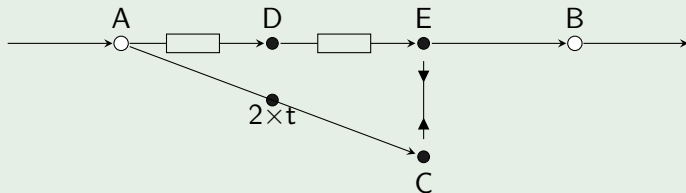
Composition of connectors is **extremely natural** in high-order logic:

$$FIFO_2(A, B) := \exists C. FIFO_1(A, C) \wedge FIFO_1(C, B)$$

$$FIFO_n(A, B) := \exists C. FIFO_{n-1}(A, C) \wedge FIFO_1(C, B)$$

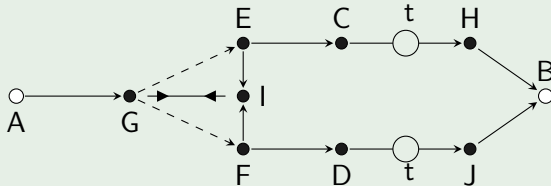
$$\begin{aligned} Timed\ FIFO_2(A, B) &:= \exists C, D, E. 2 \times t\text{-Timer}(A, C) \wedge \\ &\quad FIFO_1(A, D) \wedge FIFO_1(D, E) \wedge \\ &\quad Syncdrain(E, C) \wedge Sync(E, B) \end{aligned}$$

Example (Timed FIFO2)



Proving HOL Properties

Example (Functionality of $2 \times t$ -Timer)

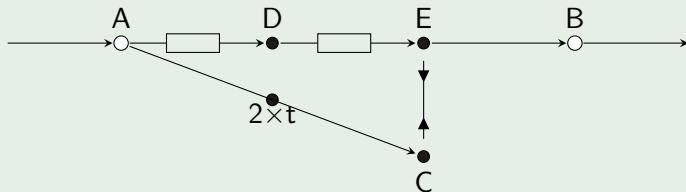


Goal : $\forall A, B, C, D, E, F, G, H, I, J \in \text{Stream } TD, t \in \text{Time}$

$$\begin{aligned} & \text{Sync}(A, G) \wedge \text{LossySync}(G, E) \wedge \text{LossySync}(G, F) \wedge \text{Sync}(E, I) \wedge \\ & \text{Sync}(F, I) \wedge \text{SyncDrain}(G, I) \wedge \text{Merge}(E, F, I) \wedge \text{Sync}(E, C) \wedge \\ & \text{Sync}(F, D) \wedge \text{Timert}(C, H, t) \wedge \text{Timert}(D, J, t) \wedge \text{Merge}(H, J, B) \\ & \rightarrow \text{Teqt}(A, B, t) \end{aligned}$$

Proving HOL Properties

Example (Functionality of Timed $FIFO_2$)

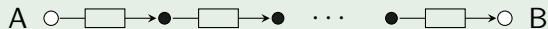


Goal : $\forall A, B, C, D, E \in Stream\ TD, t \in Time$

$$\begin{aligned} &FIFO_1(A, D) \wedge FIFO_1(D, E) \wedge Sync(E, B) \wedge \\ &SyncDrain(C, E) \wedge 2 \times t\text{-Timer}(A, C, t) \\ &\rightarrow Teqt(A, B, t) \wedge Tlt(B, tl(tlA)) \end{aligned}$$

Proving Properties on Generic Connectors

Example ($FIFO_n$)



Goal : $\forall A, B, n$

$$FIFO_n(A, B) \rightarrow Tlt(A, B) \wedge Deq(A, B)$$

- ① $FIFO_1(A, B) \rightarrow Tlt(A, B) \wedge Deq(A, B)$
- ② $(FIFO_{n-1}(A, B) \rightarrow Tlt(A, B) \wedge Deq(A, B)) \rightarrow (FIFO_n(A, B) \rightarrow Tlt(A, B) \wedge Deq(A, B))$

Outline

- 1 Introduction
- 2 Connectors as Designs
- 3 Reasoning about Connectors in Coq
- 4 **Generating Tactic Suggestions with Neural Network**
- 5 Conclusion

Motivation

In the previous slides, we omit all the proof details **because proof steps for even a trivial theorem could be very complicated.**

```
Section example2.
Theorem nt_timed : forall (A B C D E F G I C1 D1 :Stream TD) (t:Time),
  (Sync A G) /\ (LoosySync G B) /\ (LoosySync G F) /\ (Sync E I) /\
  (Sync F I) /\ (SyncDin G I) /\ (merge B F I) /\ (Sync E C1) /\
  (Sync F D1) /\ (Timert C C1 t) /\ (Timert D D1 t) /\ (merge C1 D1 B)
  -> (Teqt A B t).
Proof.
  intros.
  destruct H1destruct H2destruct H3.
  destruct H2destruct H3destruct H4.

  (**Prepare for lemma transfer_eqt**)
  assert ((Teqt A I) /\ (Teqt I B t)).
  split.
  (**Proof for Teqt A I**)
  rewrite H1.
  apply H4.
  (**Proof for Teqt I B t**)
  (**Prepare for lemma transfer_merge**)
  assert ((merge C D I) /\ (Teqt C C1 t) /\ (Teqt D D1 t) /\ (merge C1 D1 B)).
  repeat split.
  (**Proof for merge C D I**)
  destruct H5.
  destruct H6.
  rewrite <- H6.
  destruct H7.
  rewrite <- H7.
  assumption.
  (**Proof for Teqt C C1 t**)
  destruct H5.
  destruct H6.
  destruct H7.
  destruct H8.
  destruct H9.
  destruct H10.
  assumption.
  (**Proof for Teqt D D1 t**)
  destruct H5.
  destruct H6.
  destruct H7.
  destruct H8.
  destruct H9.
  destruct H11.
  assumption.
  (**Proof for merge C1 D1 B**)
  destruct H5.
  destruct H6.
  destruct H7.
  destruct H8.
  destruct H9.
  assumption.
  generalize H6.
  apply transfer_merge.
  generalize H6.
  apply transfer_eqt.
Qed.
End example2.
```

Figure on the left side shows the full proof steps of the example for $2 \times t$ timed channel. They are:

- long and including lots of repetitive work
- time consuming
- requires heavy human interaction

Question

Is it possible to automate the proof steps?

Motivation

In the previous slides, we omit all the proof details **because proof steps for even a trivial theorem could be very complicated.**

```
Section example2.
Theorem nt_timed : forall (A B C D E F G I C1 D1 :Stream TD) (t:Time),
  (Sync A G) /\ (LoosySync G B) /\ (LoosySync G F) /\ (Sync E I) /\
  (Sync F I) /\ (SyncDRAIN G I) /\ (merge B F I) /\ (Sync E C1) /\
  (Sync F D1) /\ (Timert C C1 t) /\ (Timert D D1 t) /\ (merge C1 D1 B)
  -> (Teqt A B t).
Proof.
  intros.
  destruct H1destruct H2destruct H3.
  destruct H2destruct H3destruct H4.

  (**Prepare for lemma transfer_eqt**)
  assert ((Teqt A I) /\ (Teqt I B t)).
  split.
  (**Proof for Teqt A I**)
  rewrite H1.
  apply H4.
  (**Proof for Teqt I B t**)
  (**Prepare for lemma transfer_merge**)
  assert ((merge C D I) /\ (Teqt C C1 t) /\ (Teqt D D1 t) /\ (merge C1 D1 B)).
  repeat split.
  (**Proof for merge C D I**)
  destruct H5.
  destruct H6.
  rewrite <- H6.
  destruct H7.
  rewrite <- H7.
  assumption.
  (**Proof for Teqt C C1 t**)
  destruct H5.
  destruct H6.
  destruct H7.
  destruct H8.
  destruct H9.
  destruct H10.
  destruct H10.
  assumption.
  (**Proof for Teqt D D1 t**)
  destruct H5.
  destruct H6.
  destruct H7.
  destruct H8.
  destruct H9.
  destruct H9.
  destruct H11.
  assumption.
  (**Proof for merge C1 D1 B**)
  destruct H5.
  destruct H6.
  destruct H7.
  destruct H8.
  destruct H9.
  assumption.
  generalize H6.
  apply transfer_merge.
  generalize H6.
  apply transfer_eqt.
Qed.
End example2.
```

Figure on the left side shows the full proof steps of the example for $2 \times t$ timed channel. They are:

- long and including lots of repetitive work
- time consuming
- requires heavy human interaction

Question

Is it possible to automate the proof steps?

Problem

- The complexity in proof of theorems is not only the disadvantage of our own framework, but also a very common and familiar situation for different kinds of high-order logic theorem provers.
- A generalizable method for proof automation is imperative and very useful.
- Proof automation in Coq has two steps: **generating tactic names** and **generating tactic arguments**.
 - When users tell Coq how to prove a goal, they are required to formalize their inputs with **tactic names** and **tactic arguments**.
 - Tactic names show how we simplify the goals or hypotheses.
 - Tactic arguments (also called dependencies in some literature) give more details about this tactic.

Problem

- The complexity in proof of theorems is not only the disadvantage of our own framework, but also a very common and familiar situation for different kinds of high-order logic theorem provers.
- A generalizable method for proof automation is imperative and very useful.
- Proof automation in Coq has two steps: **generating tactic names** and **generating tactic arguments**.
 - When users tell Coq how to prove a goal, they are required to formalize their inputs with **tactic names** and **tactic arguments**.
 - Tactic names show how we simplify the goals or hypotheses.
 - Tactic arguments (also called dependencies in some literature) give more details about this tactic.

Problem

- Since HOL proving is undecidable, we do not mean to provide support for fully automatic proving.
- Our approach analyzes existing Coq proofs as the training set and gives tactic suggestions according to the proof context (i.e. goals and hypotheses) to users.
- We consider only **generating tactic names** without arguments.

RNN-Based Tactic Prediction

If we consider the hypotheses and goals as input, and tactics as output, then the problem can be regarded as a variant of **sequence classification problem**, for which RNN is a popular solution.

Sequence Classification Problem

Given a sequence composed of a series of words, tell us whether it is correct under English grammar.

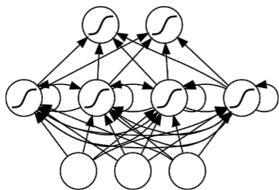
input	output
he is a good man	correct
we am fine	wrong

In RNNs, if we put a set of elements (as vectors, of course) into a hidden unit and take the last output as the result, then the network can be trained to solve sequence classification problem.

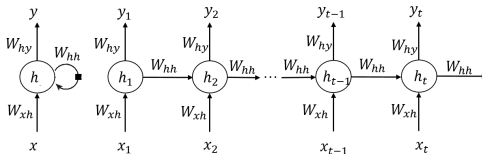
RNN-Based Tactic Prediction

Definition (Recurrent Neural Network)

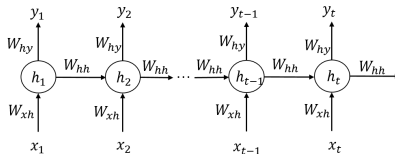
Recurrent neural networks (RNNs) are a family of artificial neural networks whose hidden units are locally connected. In such networks, output value of a hidden unit depends on not only the current input but all the history inputs.



(a)



(b)



(c)

Figure: (a) A RNN (b) A Simple RNN Cell (c) The Layer Unrolled from (b)

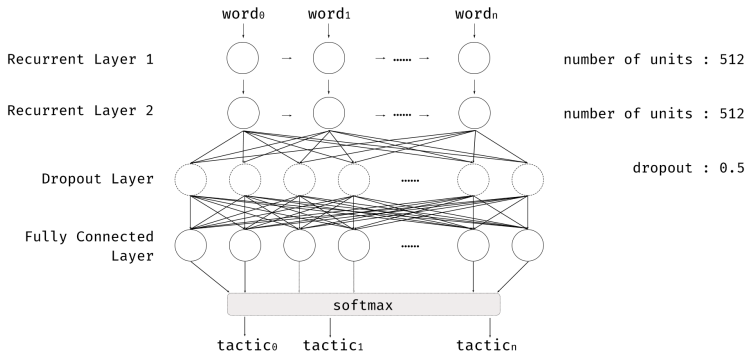
RNN-Based Tactic Prediction

Typical Coq proof steps are shown as follows:

input	proof step	tactic
H1 : $A \rightarrow B$	apply H1	apply
H2 : A		
Goal : B		
H1 : $A \rightarrow B$	assumption	assumption
H2 : A		
Goal : A		

- An input includes
 - a set of hypotheses, each in form of word sequence
 - a goal, in form of word sequence
- An output, or a proof step, includes the tactic to use and the arguments passed to the tactic. To simplify this problem, in this work we only predict the tactic itself.

RNN-Based Tactic Prediction : Build the Network



RNN-Based Tactic Prediction : Build the Network

- The network contains two recurrent layers, each composed of 512 self-connected hidden units.
- When there is a goal to be proved, usually more than one tactics are applicable. If the network concentrates too much on the “standard answer” given by the training set, it could be trapped by severe overfit. In this network, we use a dropout layer to deal with this problem.
- In the training rounds, cells in the dropout layer are randomly disconnected according to a given dropout rate.
- A fully-connected layer combines features learnt from the recurrent layers.
- A softmax layer is used to refactor the output as a probability distribution to normalize the output of the network.

RNN-Based Tactic Prediction : Build the Network

- The network contains two recurrent layers, each composed of 512 self-connected hidden units.
- When there is a goal to be proved, usually more than one tactics are applicable. If the network concentrates too much on the “standard answer” given by the training set, it could be trapped by severe overfit. In this network, we use a dropout layer to deal with this problem.
- In the training rounds, cells in the dropout layer are randomly disconnected according to a given dropout rate.
- A fully-connected layer combines features learnt from the recurrent layers.
- A softmax layer is used to refactor the output as a probability distribution to normalize the output of the network.

RNN-Based Tactic Prediction : Build the Network

- The network contains two recurrent layers, each composed of 512 self-connected hidden units.
- When there is a goal to be proved, usually more than one tactics are applicable. If the network concentrates too much on the “standard answer” given by the training set, it could be trapped by severe overfit. In this network, we use a dropout layer to deal with this problem.
- In the training rounds, cells in the dropout layer are randomly disconnected according to a given dropout rate.
- A fully-connected layer combines features learnt from the recurrent layers.
- A softmax layer is used to refactor the output as a probability distribution to normalize the output of the network.

Evaluation

- The network does not aim at full automation. It only provides tactic suggestions as reference to human users according to the context of goals and hypotheses.
- It should not be limited by “one single correct answer”.
- Instead, we take the top n possible tactics according to the probability distribution provided by the neural network.
- Correspondingly, we have **n -correctness rate** as the evaluation standard:

Definition (n -correctness)

The raw output of softmax layer can be regarded as a probability distribution. We take the top n predicted results according to this distribution, if they successfully cover the correct answer we say the output is n -correct.

Evaluation

We run the training algorithm 50 times and use the average n -correctness rates in this figure:

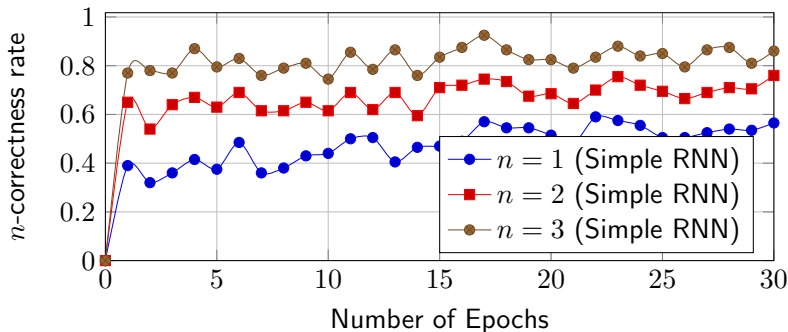


Figure: n -Correctness with Simple RNN Units

The network suffers from the **vanishing gradient problem**:

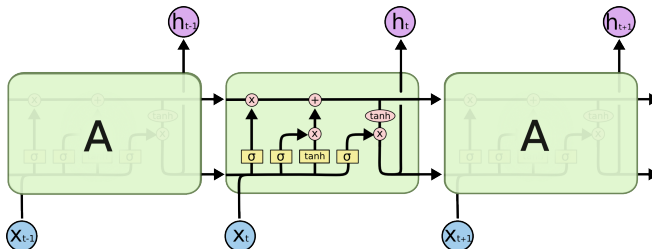
Vanishing Gradient Problem

In neural networks, when the network structure becomes deeper and deeper, the gradients of the networks output with respect to the parameters in the early layers sometimes become extremely small, which is called the vanishing gradient problem.

- A large change in the parameters of the early layers can only change the final output a little.
- The problem happens in almost all deep learning networks, such as CNN or RNN.

Optimization

- **Solution : Long-Short Term Memory Units (LSTM)**
- Compared with Simple RNN units, LSTM units use much more complex architectures to get rid of the vanishing gradient problem.
- The unrolled structure of a single self-connected LSTM hidden unit is as follows:



Optimization

- Every LSTM unit has
 - two self-loop connections h_t and C_t ,
 - four groups of trainable weight variables and biases:
 - W_f, b_f that control how much history information to forget,
 - W_i, b_i that control how much current information should be accumulated to C_t ,
 - W_C, b_C that compute current information, and
 - W_o, b_o that control how much history information should be taken as h_t .
- LSTM units satisfy the following equations:

$$C_t = \sigma(W_f \cdot [x_t, h_{t-1}] + b_f) * C_{t-1} + acc_t$$

$$acc_t = \tanh(W_C \cdot [x_t, h_{t-1}] + \sigma(W_i \cdot [x_t, h_{t-1}] + b_i) * b_C$$

$$h_t = \sigma(W_o \cdot [x_t, h_{t-1}] + b_o) * \tanh(C_t)$$

Optimized Performance

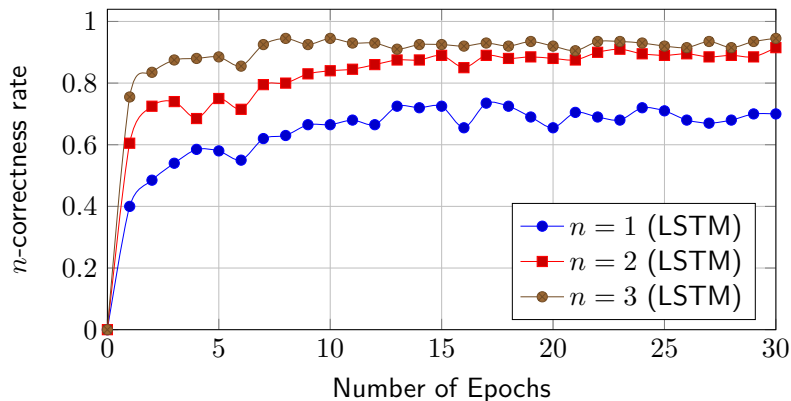


Figure: n -Correctness with LSTM Unit

Outline

- 1 Introduction
- 2 Connectors as Designs
- 3 Reasoning about Connectors in Coq
- 4 Generating Tactic Suggestions with Neural Network
- 5 **Conclusion**

Conclusion and Future Work

What we have done in this work:

- ① The UTP design semantics for timed Reo connectors
- ② A set of Coq specifications for timed Reo channels and connectors based on the design semantics
- ③ Properties of connectors being proved with help of Coq
- ④ Properties of abstract connectors being proved through induction
- ⑤ Automatic tactic prediction with help of RNN

Future work:

- ① develop the design model and specification in Coq for probabilistic and hybrid connectors
- ② extend the RNN-based approach to more Coq libraries for general problems

Conclusion and Future Work

What we have done in this work:

- ① The UTP design semantics for timed Reo connectors
- ② A set of Coq specifications for timed Reo channels and connectors based on the design semantics
- ③ Properties of connectors being proved with help of Coq
- ④ Properties of abstract connectors being proved through induction
- ⑤ Automatic tactic prediction with help of RNN

Future work:

- ① develop the design model and specification in Coq for probabilistic and hybrid connectors
- ② extend the RNN-based approach to more Coq libraries for general problems

Thanks!