

Tamarinハンズオン

CSS2026 形式検証とセキュリティワークショップ (FWS)

FWS運営委員会

※本資料に掲載されている商品，機能等の名称はそれぞれ各社が商標として使用している場合があります

はじめに

ぜひTamarin Proverの実行環境をご用意の上ご参加ください

- 方法1：ハンズオン用Docker環境を用いてインストール (Windows, Mac) **推奨**
- 方法2：Tamarin公式推奨の方法でインストール (Mac, Linux)



詳しくはFWSのハンズオンページをご覧ください

<https://www.iwsec.org/fws/2026/hands-on.html>

Dockerファイルのbuild・起動の仕方

※ buildは事前に済ませておくことをお勧めします

- Docker Desktopを用いる場合（その他の場合はREADMEを参照ください）
 - tamarin-demo.zipをダウンロード・解凍
 - Docker Desktopを起動
 - ターミナルで該当のディレクトリに移動し、次を実行

はじめに

ご不明な点があればお気軽にどうぞ！

- slackのFWSチャンネル
- 挙手による質問

Special Thanks !

本ハンズオン資料は招待講演者の田中 篤様よりご提供いただいた内容をベースに作成しています。

ありがとうございます！

(参考) このハンズオンのここがすごい！

- Tamarinには公式のチュートリアルや例が多数あるが、結局「状態遷移」や「マルチセット書換ルール」の直感的な理解が難しい・・・
- 入門者向け資料はあっても超入門者向け資料はない・・・ましてや日本語資料なんて・・・

そんな方に向けて、本ハンズオンでは超入門者向けにTamarinの基礎を日本語で直感的にわかりやすく説明します！

基礎を理解すれば、プロトコル記述の応用にも繋がられます。

もくじ

- Tamarinにおけるモデリングの基本
- Tamarinで検証できること
- Tamarinの証明ルート
- Tamarinでプロトコルをモデリングしてみよう

もくじ

- Tamarinにおけるモデリングの基本
- Tamarinで検証できること
- Tamarinの証明ルート
- Tamarinでプロトコルをモデリングしてみよう

- Tamarinの状態遷移は以下のように表記

[Premise()] --[ActionFact()]-> [Conclusion()]

前提

(存在しなければならない状態)

ラベル

(状態遷移されたことを記録)

結論

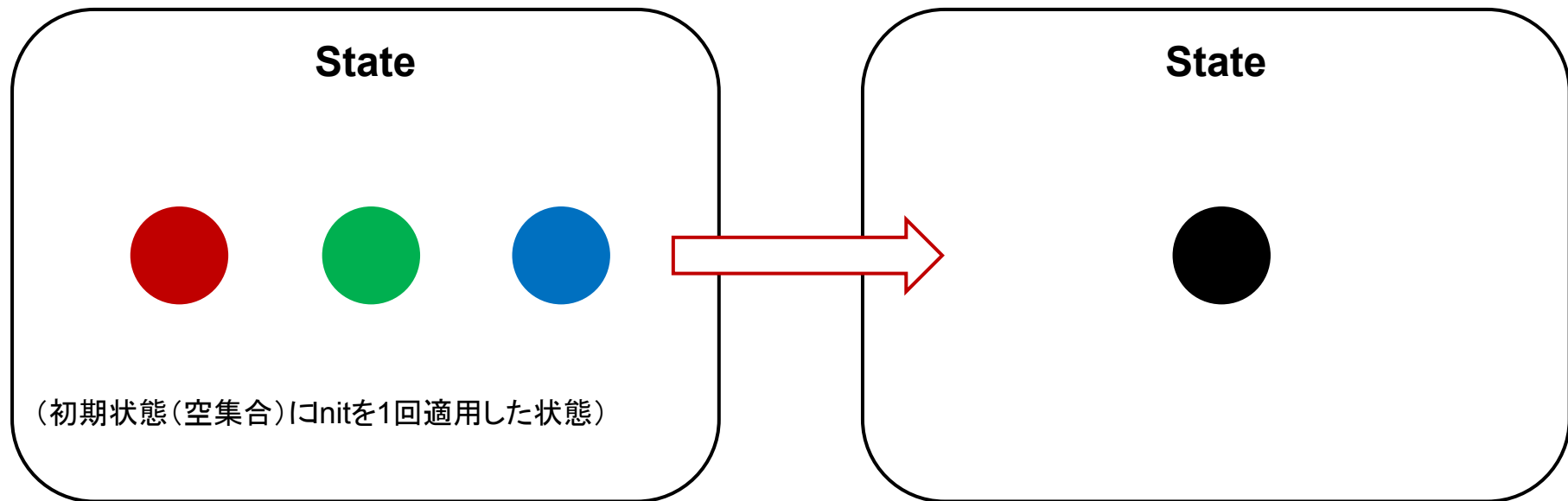
(新しい状態)

Point

状態遷移が発火した時、前提条件は (基本的には) 消える

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身はActionFact名)
 - (初期状態は空集合)
 - **Red, Green, Blue** (“Init”) 現在の状態にRed, Green, Blueを加える (任意のタイミングで適用可能)
 - **Red + Blue => Purple** (“GenPurple”) RedとBlueを消費してPurpleを作る
 - **Purple + Green => Black** (“GenBlackFromPurple”) PurpleとGreenを消費してBlackを作る
 - **Red + Green + Blue => Black** (“GenBlackFromRGB”) RedとGreenとBlueを消費してBlackを作る

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身はActionFact名)
 - Red, Green, Blue (“Init”)
 - Red + Blue => Purple (“GenPurple”)
 - Purple + Green => Black (“GenBlackFromPurple”)
 - Red + Green + Blue => Black (“GenBlackFromRGB”)



- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]
```

```
rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]
```

```
rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]
```

```
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

trace 履歴

Init()

State

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

```
rule Init:  
  [ ]  
  --[ Init() ]->  
  [ Red(), Blue(), Green() ]
```

```
rule GenPurple:  
  [ Red(), Blue() ]  
  --[ GenPurple() ]->  
  [ Purple() ]
```

```
rule GenBlack:  
  [ Red(), Blue(), Green() ]  
  --[ GenBlackFromRGB() ]->  
  [ Black() ]
```

```
rule GenBlack_From_Purple:  
  [ Purple(), Green() ]  
  --[ GenBlackFromPurple() ]->  
  [ Black() ]
```

trace 履歴

Init()

State

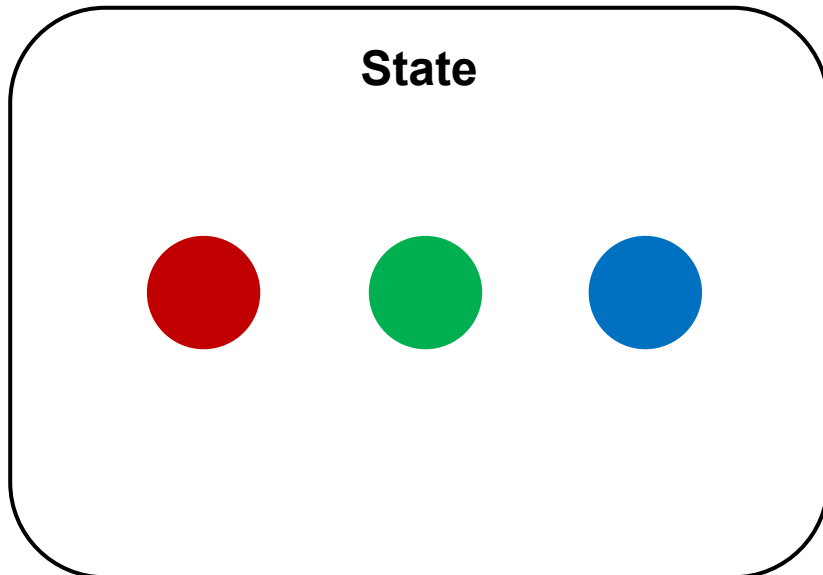
何もない状態

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]

rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]

rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]

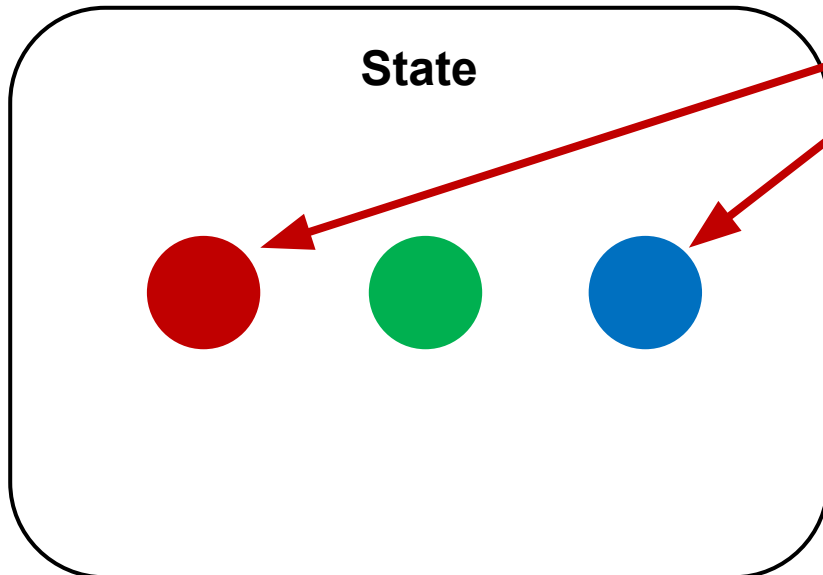
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenPurple()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]

rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]

rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]

rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

例: 色の混ぜ合わせ

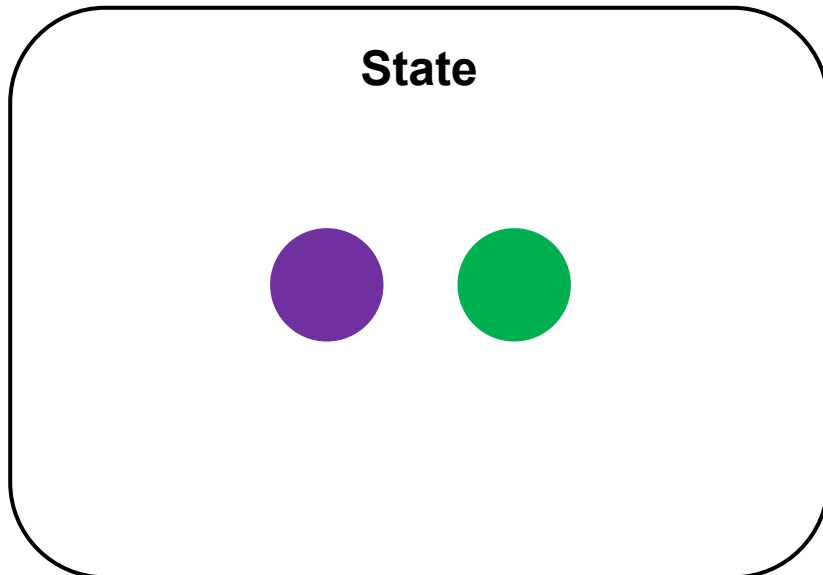
16

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenPurple()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]
```

```
rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]
```

```
rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]
```

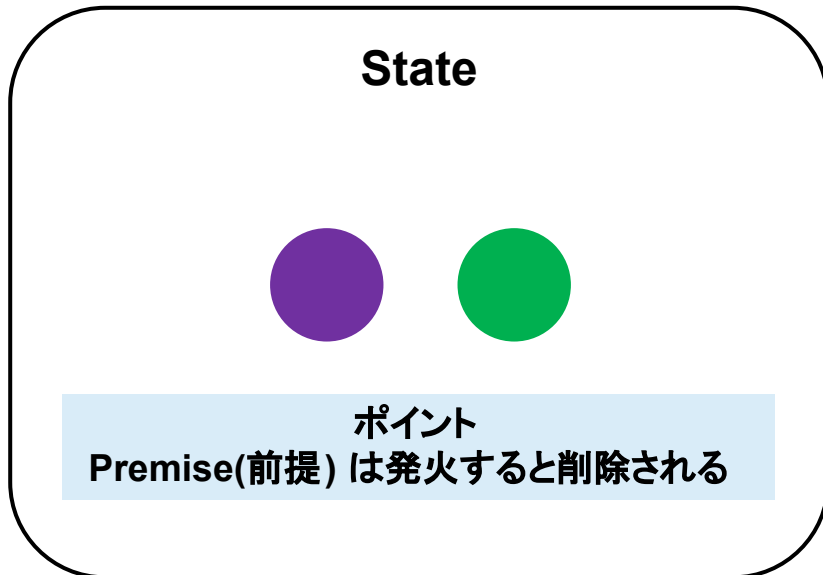
```
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenPurple()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]
```

```
rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]
```

```
rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]
```

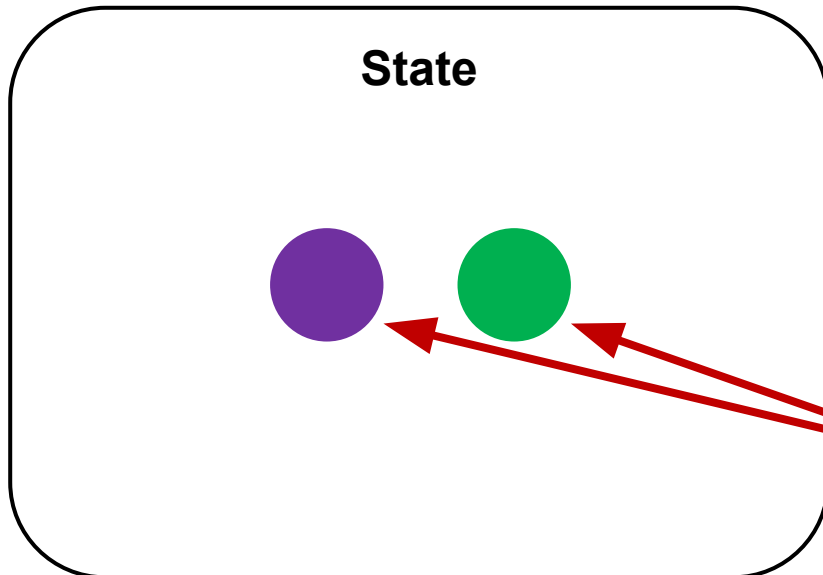
```
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenPurple()
GenBlackFromPurple()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]

rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]

rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]

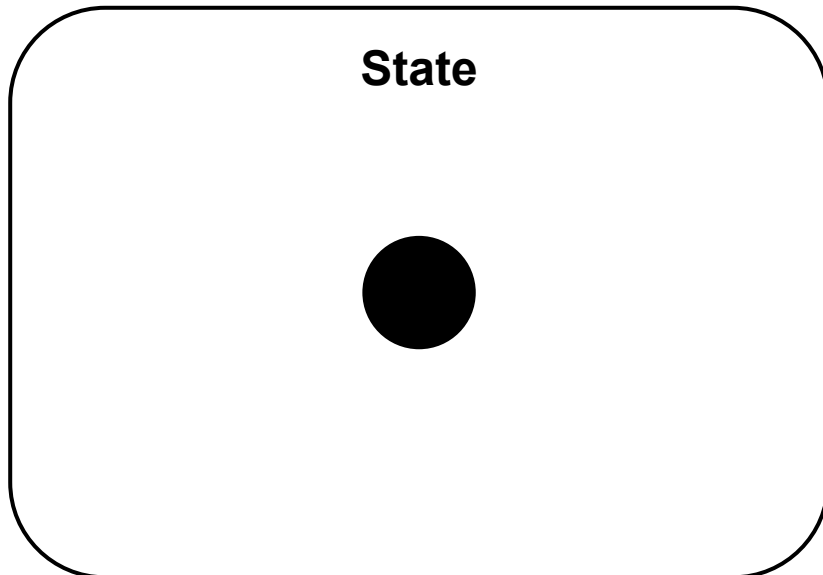
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

`code1.spthy`

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenPurple()
GenBlackFromPurple()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]

rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]

rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]

rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

`code1.spthy`

- Red, Green, Blue (“Init”)

- Red + Blue => Purple (“GenPurple”)

- Purple + Green => Black (“GenBlackFromPurple”)

```
rule Init:  
  [ ]  
  --[ Init() ]->
```

Blackになるパターンはこれだけではない
別のパターンも考えてみる

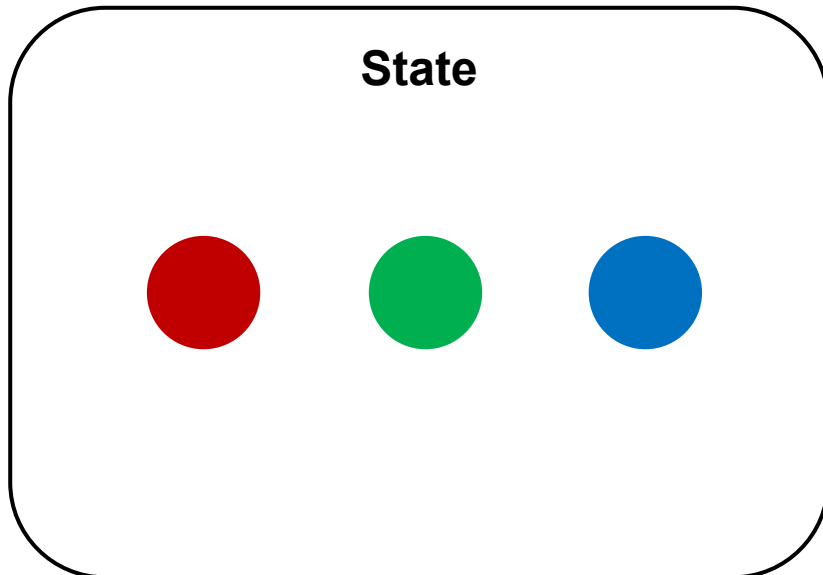
```
rule GenBlack_From_Purple:  
  [ Purple(), Green() ]  
  --[ GenBlackFromPurple() ]->  
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]

rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]

rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]

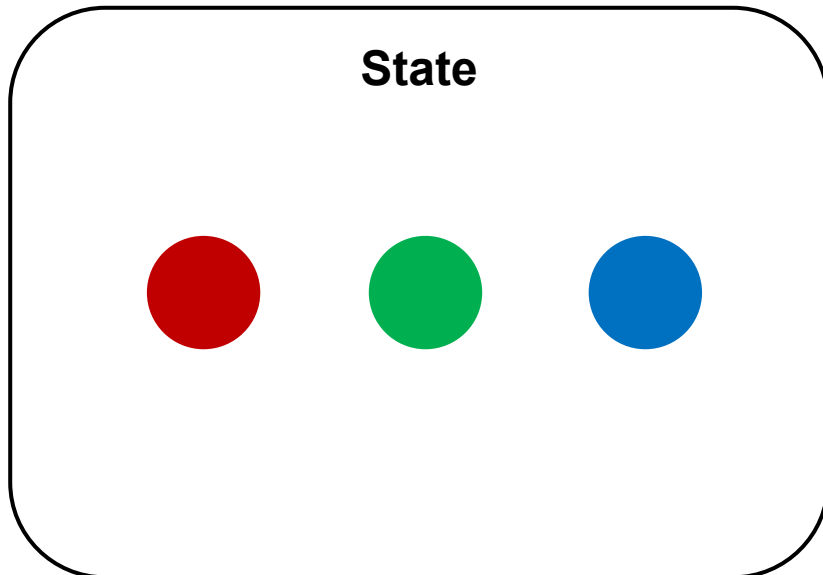
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenBlackFromRGB()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]
```

```
rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]
```

```
rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]
```

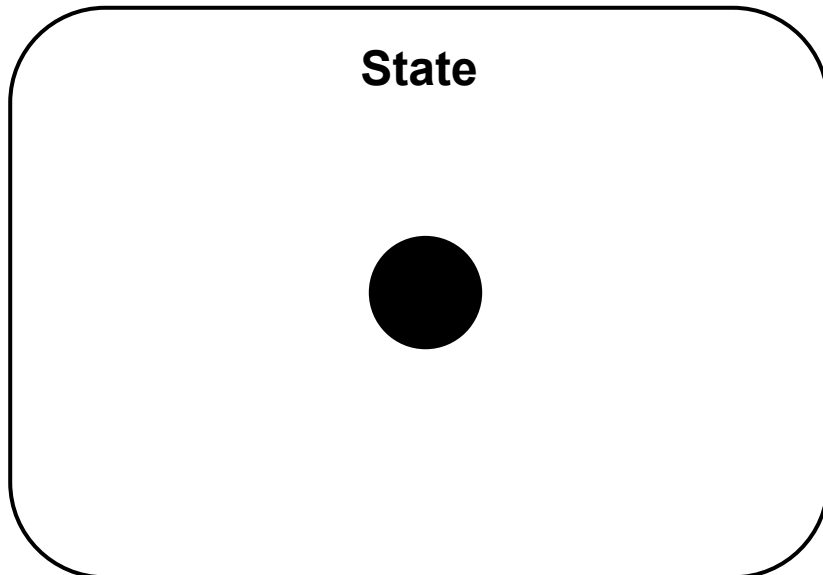
```
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenBlackFromRGB()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]
```

```
rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]
```

```
rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]
```

```
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

`code1.spthy`

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)

```
rule Init:  
  [ ]  
  --[ Init() ]->
```

これらを組み合わせればいろんな状態に遷移可能

例題

「PurpleとBlackが同時に存在することは可能？」

```
rule GenBlack_From_Purple:  
  [ Purple(), Green() ]  
  --[ GenBlackFromPurple() ]->  
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴

State



ゴールの状態

```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]

rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]

rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]

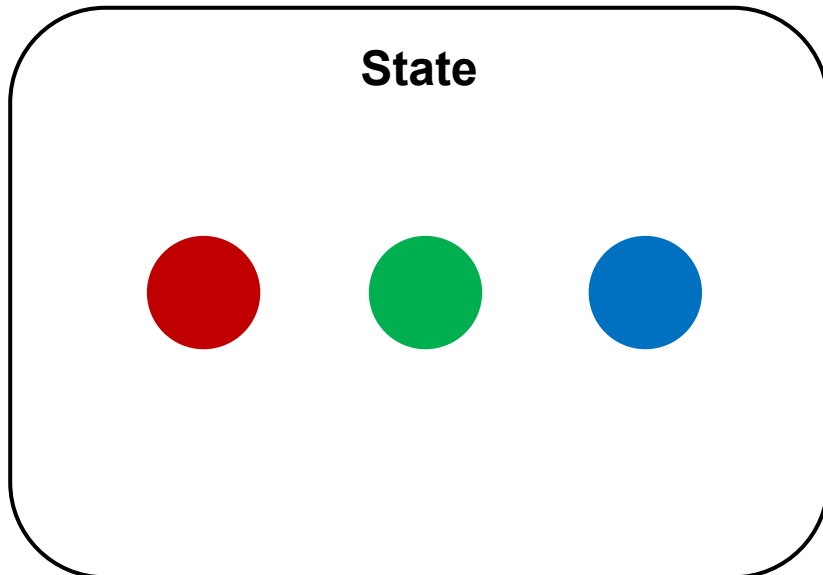
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]

rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]

rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]

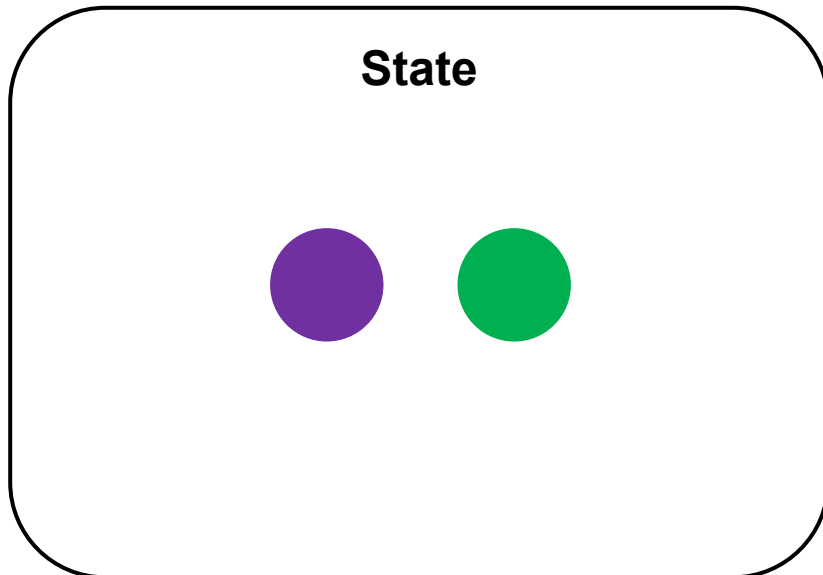
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

`code1.spthy`

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenPurple()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]
```

```
rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]
```

```
rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]
```

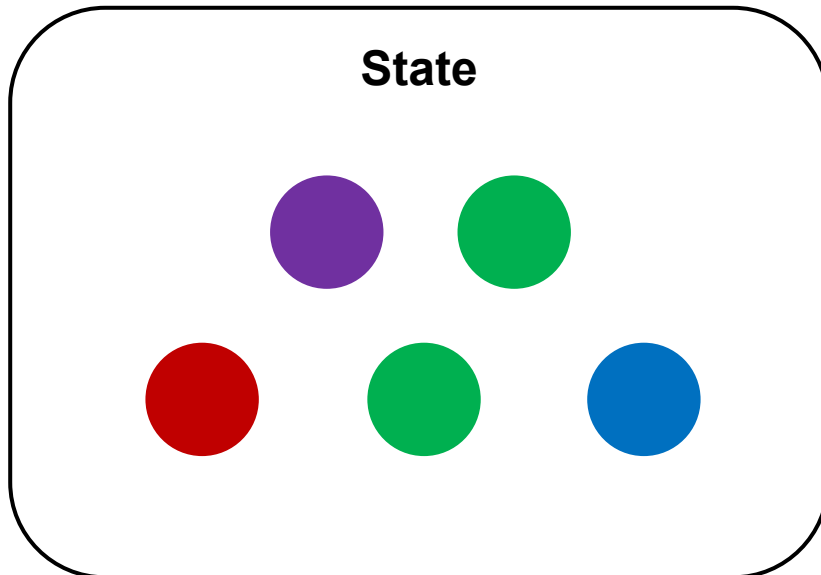
```
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenPurple()
Init()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]

rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]

rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]

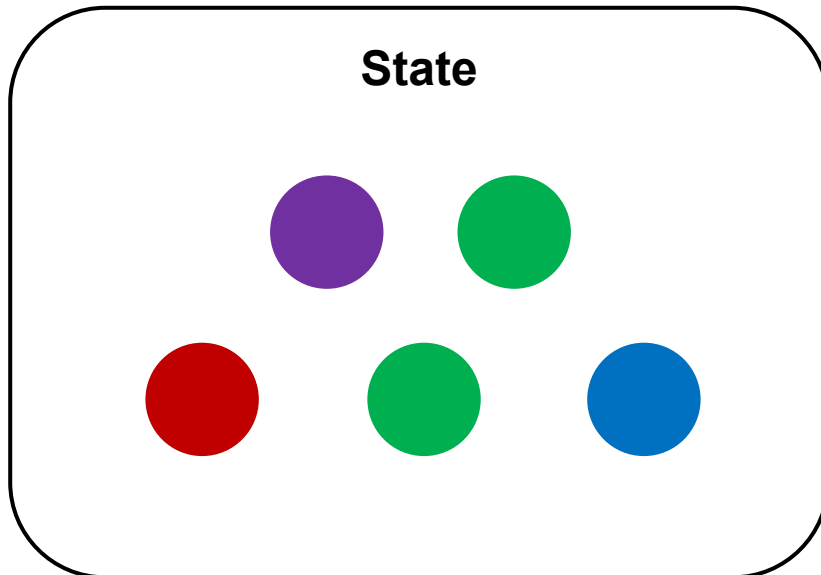
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenPurple()
Init()
GenBlackFromRGB()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]
```

```
rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]
```

```
rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]
```

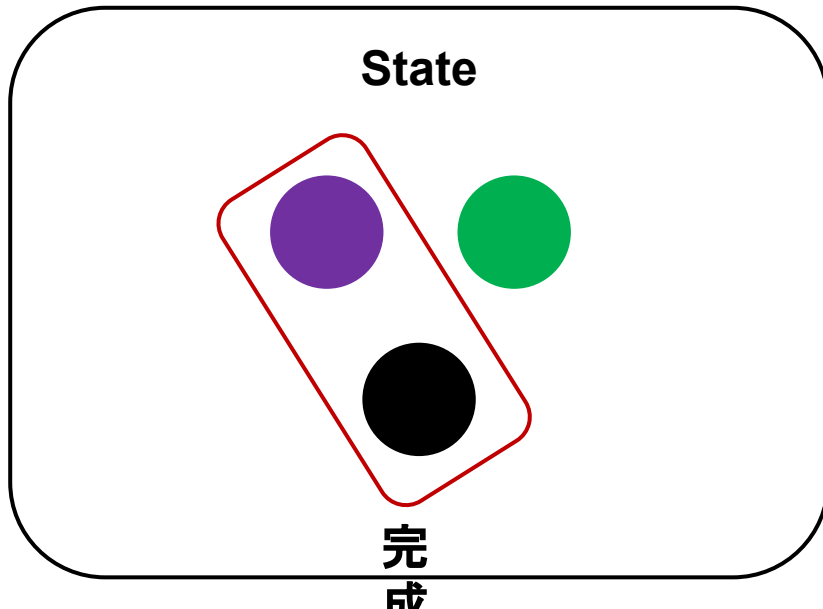
```
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Red, Blue, Greenを用いて、以下の状態遷移を表現 (カッコの中身は ActionFact名)

code1.spthy

- Red, Green, Blue (“Init”)
- Red + Blue => Purple (“GenPurple”)
- Purple + Green => Black (“GenBlackFromPurple”)
- Red + Green + Blue => Black (“GenBlackFromRGB”)

trace 履歴
Init()
GenPurple()
Init()
GenBlackFromRGB()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]
```

```
rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]
```

```
rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]
```

```
rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- これができると、だいたいのモデリングは自分でイメージ可能
- あとはこの考え方の応用で、プロトコルもモデリング可能

```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]

rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]

rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]

rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

• まとめ

[Premise()] --[ActionFact()]-> [Conclusion()]

前提

(存在しなければならない状態)

ラベル

(状態遷移されたことを記録)

結論

(新しい状態)

Point

状態遷移が発火した時、前提条件は (基本的には) 消える

もくじ

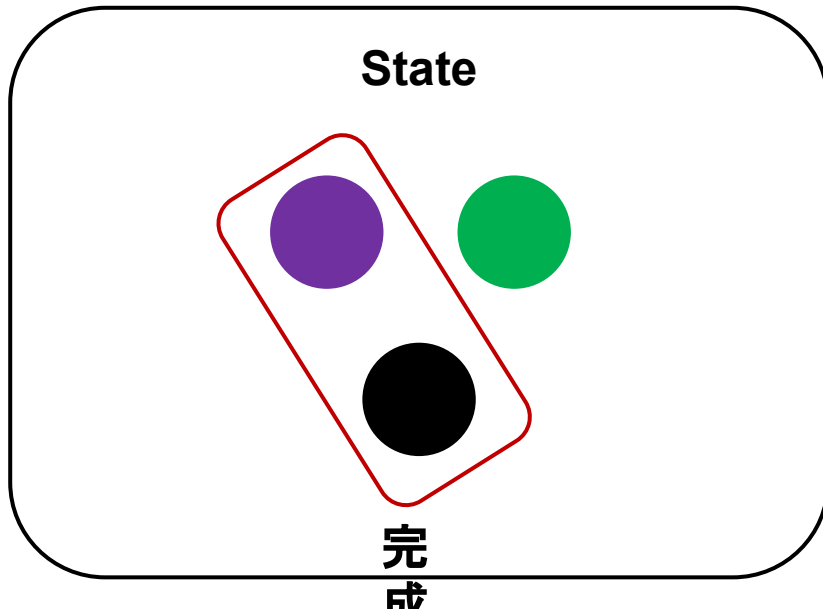
- Tamarinにおけるモデリングの基本
- **Tamarinで検証できること**
- Tamarinの証明ルート
- Tamarinでプロトコルをモデリングしてみよう

- Tamarinでは、ラベルについての一階述語論理式を用いて検証要件を表現できる

`code1.spthy`

- 例
 - 紫が存在する状態に遷移可能か？
 - 紫が生成された後に黒が生成される traceは存在するか？

trace 履歴
Init()
GenPurple()
Init()
GenBlackFromRGB()



```
rule Init:
  [ ]
  --[ Init() ]->
  [ Red(), Blue(), Green() ]

rule GenPurple:
  [ Red(), Blue() ]
  --[ GenPurple() ]->
  [ Purple() ]

rule GenBlack:
  [ Red(), Blue(), Green() ]
  --[ GenBlackFromRGB() ]->
  [ Black() ]

rule GenBlack_From_Purple:
  [ Purple(), Green() ]
  --[ GenBlackFromPurple() ]->
  [ Black() ]
```

- Tamarinでは、ラベルについての一階述語論理式を用いて検証要件を表現できる

`code1.spthy`

- 例
 - 紫が存在する状態に遷移可能か？

```
lemma reach_Purple:
exists-trace
"Ex #i. GenPurple() @ #i"
```

- 紫が生成された後に黒が生成される traceは存在するか？

Point

時点(#iや#j)を用いて柔軟な要件を記述可能

all-traces(全称) 命題と
exists-trace(存在) 命題
を記述可能

| : OR
& : AND
⇒ : ならば

```
lemma Purple_after_Black:
exists-trace
"(Ex #i #j.
  GenPurple() @ #i
  & GenBlackFromRGB() @ #j
  & #i < #j)
|
(Ex #i #j.
  GenPurple() @ #i
  & GenBlack_From_Purple() @ #j
  & #i < #j)"
```

```
rule Init:
[ ]
--[ Init() ]->
[ Red(), Blue(), Green() ]
```

```
rule GenPurple:
[ Red(), Blue() ]
--[ GenPurple() ]->
[ Purple() ]
```

```
rule GenBlack:
[ Red(), Blue(), Green() ]
--[ GenBlackFromRGB() ]->
[ Black() ]
```

```
rule GenBlack_From_Purple:
[ Purple(), Green() ]
--[ GenBlackFromPurple() ]->
[ Black() ]
```

演習1 code1を実行して検証結果を見てみよう

検証結果を予想しつつ、[code1.spthy](#) をTamarinで実行し、
検証結果を見てみましょう

コードを選択/入力して
「検証する」をクリック



Tamarin Prover ローカルデモ tamarin-prover.com · FWS運営委員会

— サンプルを選択 — ▶ 検証する

入力 (.spthy)	結果 (tamarin-prover --prove)
<pre>1 theory CODE1 2 begin 3 4 rule Init: 5 □ 6 --[Init()]-></pre>	<pre>完了 (0.1s) qcu qed /* All wellformedness checks were successful. */</pre>

(Docker以外の人向け) 実行コマンドはこちら

```
tamarin-prover --prove code1.spthy
```

--proveは「証明を完了させてください」というオプションコマンドです
これを付けない場合は、証明プロセスが1stepずつ実行されます

演習1 code1を実行して検証結果を見てみよう

解説

次のような結果が出力されたと思います

```
=====
summary of summaries:

analyzed: /tmp/tmpiol6rluq/input.spthy

processing time: 0.05s

reach_Purple (exists-trace): verified (4 steps)
Purple_after_Black (exists-trace): verified (8 steps)
=====
```

```
reach_Purple (exists-trace): verified (4 steps)
Purple_after_Black (exists-trace): verified (8 steps)
```

これは,

- 紫が存在する状態に遷移可能である
- 紫が生成された後に黒が生成されるtraceは存在する

ことを意味します

演習2 検証要件を追加してみよう

[code2.spthy](#) の***部分を埋めて、次の検証要件を追加してみましょう

- 紫と黒が同時に存在する状態に遷移可能か？

```
(追加ルール)
rule CheckPurpleAndBlack:
  [ Purple(), Black() ]
  --[ PurpleAndBlack() ]->
  [ Purple(), Black() ]
```

```
lemma reach_PurpleAndBlack:
  exists-trace
  "***"
```

- 紫が存在する状態に到達可能でないか？
(「紫が存在する状態に到達可能」の否定)

```
lemma not_reach_Purple:
  " not (***)"
```

notは, all-traces not...
(任意のトレースに対して...ではない) の略

演習2 検証要件を追加してみよう

解説

- 紫と黒が同時に存在する状態に遷移可能か？

```
(追加ルール)
rule CheckPurpleAndBlack:
  [ Purple(), Black() ]
  --[ PurpleAndBlack() ]->
  [ Purple(), Black() ]
```

```
lemma reach_PurpleAndBlack:
  exists-trace
    "Ex #i. PurpleAndBlack() @ #i"
```

- 紫が存在する状態に到達可能でないか？
(「紫が存在する状態に到達可能」の否定)

```
lemma not_reach_Purple:
  " not (Ex #i. GenPurple() @ #i)"
```

演習2 検証要件を追加してみよう

解説

- 次の結果が表示されたと思います

```
reach_PurpleAndBlack (exists-trace): verified (4 steps)
not_reach_Purple (all-traces): falsified - found trace (4 steps)
```

- これは、Tamarinが次を証明したことを意味します：
 - **紫と黒が同時に存在する状態に遷移可能であること**
 - **紫が存在する状態に到達可能ではないこと**
(「紫が存在する状態に到達可能」の否定を否定する結果)

もくじ

- Tamarinにおけるモデリングの基本
- Tamarinで検証できること
- **Tamarinの証明ルート**
- Tamarinでプロトコルをモデリングしてみよう

- Tamarinは深さ優先探索 → **ゴールから逆に探索する**
- ゴールから逆算している様子をインタラクティブモードで観測してみる

インタラクティブモード:

ユーザが証明の進め方を視覚的に確認・選択しながら検証を進められるモード
攻撃が存在する場合, 攻撃に至る実行過程も確認できる

(時間の都合上本ハンズオンでは扱いません)

- Tamarinは深さ優先探索 → **ゴールから逆に探索する**
- ゴールから逆算している様子をインタラクティブモードで観測してみる
- さきほどの色混ぜの例をもとに
 - 「Black()になるようなパスは存在するか？」を1つずつ追って証明する

```
lemma Black_exists:  
  exists-trace  
  "  
  (Ex #t1. GenBlackFromRGB() @ t1)  
  |  
  (Ex #t1. GenBlackFromPurple() @ t1)  
  "
```

“Black()が存在するためには
trace履歴に

- GenBlackFromRGB
 - GenBlackFromPurple
- のどちらから存在している ”

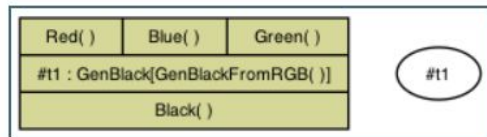
を証明する lemma

- ゴールは何か？ → `Black()`が生成されていること
- 以下の2つのパターンが考えられる
 - `GenBlackFromRGB` (3色から生成)
 - `GenBlackFromPurple` (紫と緑から生成)
- Tamarinの証明木の初期状態も以下の 2パターンを想定していることが確認できる

```
1. solve( Red( ) ▶0 #t1 ) // nr. 3 (from rule GenBlack)
2. solve( Blue( ) ▶1 #t1 ) // nr. 4 (from rule GenBlack)
3. solve( Green( ) ▶2 #t1 ) // nr. 5 (from rule GenBlack)

a. autoprove (A. for all solutions)
b. autoprove (B. for all solutions) with proof-depth bound 5
s. autoprove (S. for all solutions) for all lemmas
```

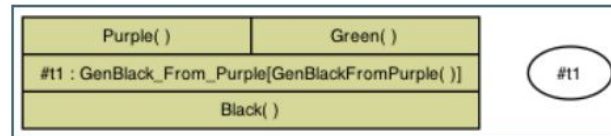
Constraint system



```
1. solve( Purple(
    ) ▶0 #t1 ) // nr. 3 (from rule GenBlack_From_Purple)
2. solve( Green(
    ) ▶1 #t1 ) // nr. 4 (from rule GenBlack_From_Purple)

a. autoprove (A. for all solutions)
b. autoprove (B. for all solutions) with proof-depth bound 5
s. autoprove (S. for all solutions) for all lemmas
```

Constraint system



- ゴールは何か？ → `Black()`が生成されていること
- 以下の2つのパターンが考えられる
 - `GenBlackFromRGB` (3色から生成)
 - `GenBlackFromPurple` (紫と緑から生成)
- Tamarinの証明木の初期状態も以下の 2パターンを想定していることが確認できる

```
1. solve( Red( ) ▶0 #t1 ) // nr. 3 (from rule GenBlack)
2. solve( Blue( ) ▶1 #t1 ) // nr. 4 (from rule GenBlack)
3. solve( Green( ) ▶2 #t1 ) // nr. 5 (from rule GenBlack)

a. autoprove (A. for all solution
b. autoprove (B. for all solution
s. autoprove (S. for all solution

Constraint system
```

Red()	Blue()	Green()
#t1 : GenBlack[GenBlackFromRGB()]		
Black()		

#t1

```
1. solve( Purple(
    ) ▶0 #t1 ) // nr. 3 (from rule GenBlack_From_Purple)
2. solve( Green(
    ) ▶1 #t1 ) // nr. 4 (from rule GenBlack_From_Purple)

with proof-depth bound 5
for all lemmas

Constraint system
```

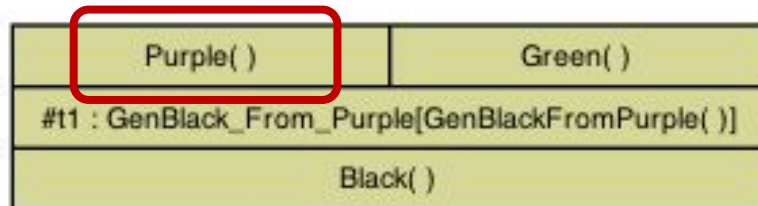
Purple()	Green()
#t1 : GenBlack_From_Purple[GenBlackFromPurple()]	
Black()	

#t1

ここから上に木が追加されていく過程を
見ていきましょう

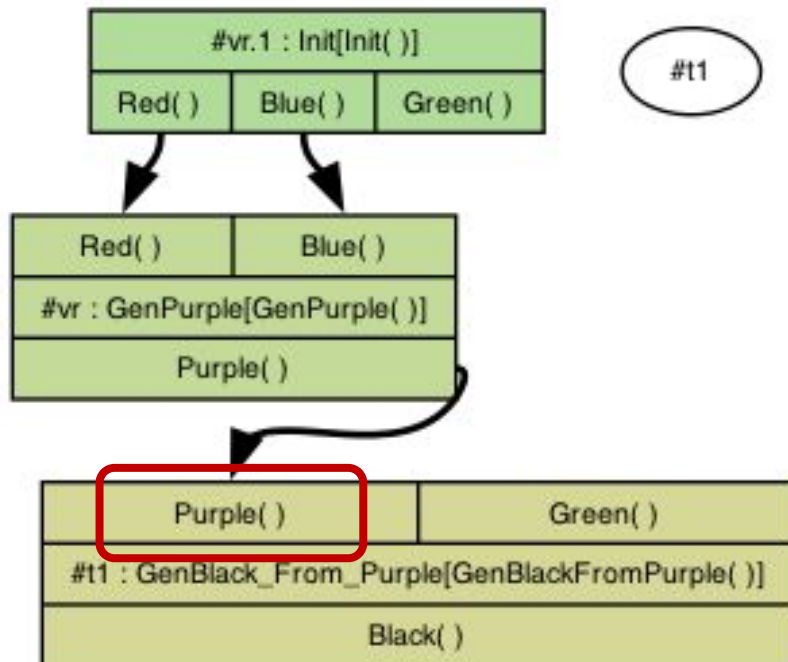
- まとめ
 - Tamarinは深さ優先探索
 - 「Premise(前提条件) はどこからきたのか？」をさかのぼっていく

Purple()はどこから？



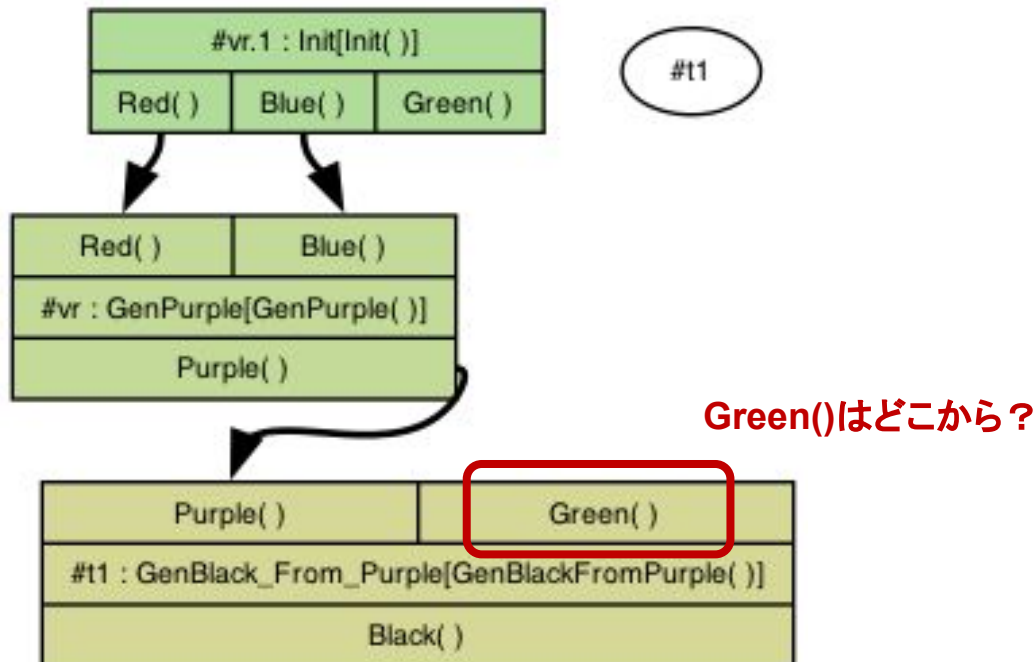
- まとめ

- Tamarinは深さ優先探索
- 「Premise(前提条件) はどこからきたのか？」をさかのぼっていく



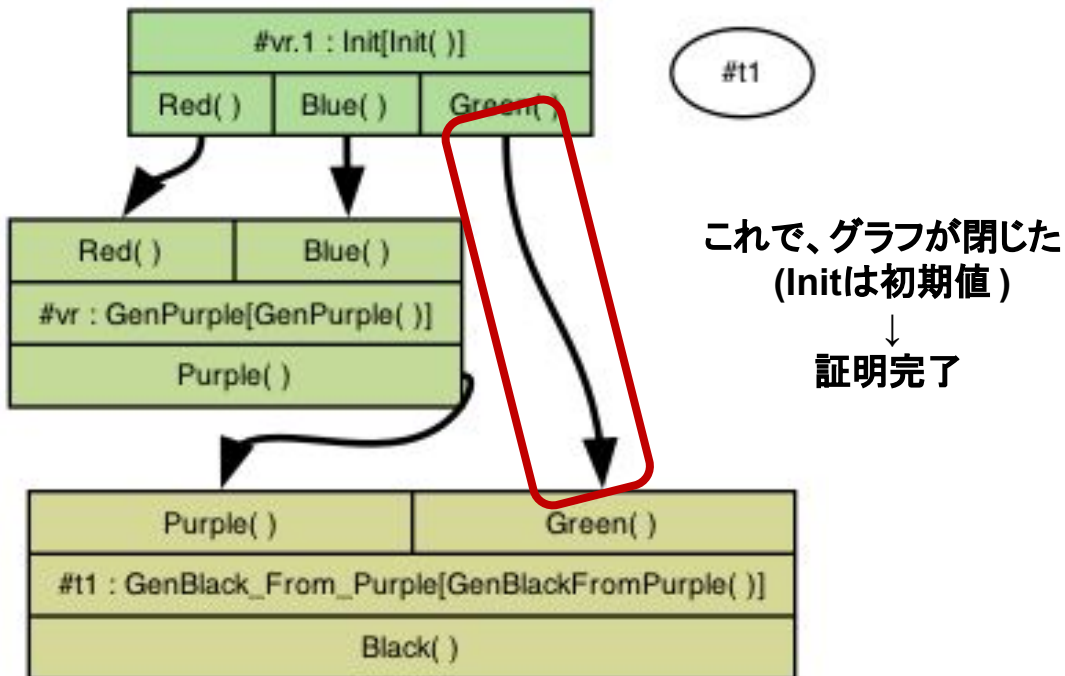
- まとめ

- Tamarinは深さ優先探索
- 「Premise(前提条件) はどこからきたのか？」をさかのぼっていく



- まとめ

- Tamarinは深さ優先探索
- 「Premise(前提条件) はどこからきたのか？」をさかのぼっていく



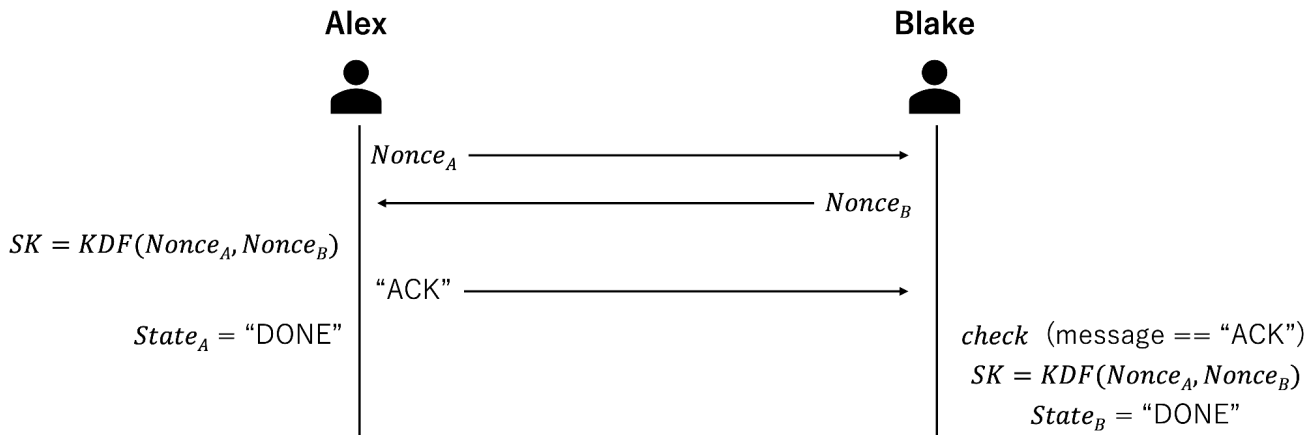
- Tamarinの自動証明は、「状態遷移の無限ループ」や「間違った探索ルート」に陥ることがある
- これを防ぐために、“interactive mode”がある
 - 「人間の手で補助的に探索空間を選択して、正しい証明ルートを辿らせる」 ことが可能
 - すなわち、“探索空間の枝刈り”が可能

時間の都合上、本ハンズオンでは省略します・・・

もくじ

- Tamarinにおけるモデリングの基本
- Tamarinで検証できること
- Tamarinの証明ルート
- Tamarinでプロトコルをモデリングしてみよう

- Tamarin Toy Protocol* を例に, 演習ベースで Tamarinでプロトコルをモデリングする方法を学ぶ
- 次を検証
 - SKの機密性
 - 明示的な(一方向)認証性
(BlakeがDONE状態になっているならば, Alexも同じ鍵について DONE状態になっている)



*: https://github.com/benjaminkiesl/tamarin_toy_protocol より

- WPA2の4wayハンドシェイクを簡略化したバージョン
- Toy Protocol の流れ
 - 欠陥のある単純なプロトコルをモデル化
 - 段階的にプロトコルを改良して欠陥に対処(全 3段階)

【演習課題の流れ】

1. 初期プロトコルの実装: 簡単なセッション鍵の確立 脆弱
2. 拡張: 事前共有マスター鍵の導入 機密性を満たすようにしよう
3. 拡張: メッセージ認証コード(MAC)の導入 明示的な認証性を持たせよう

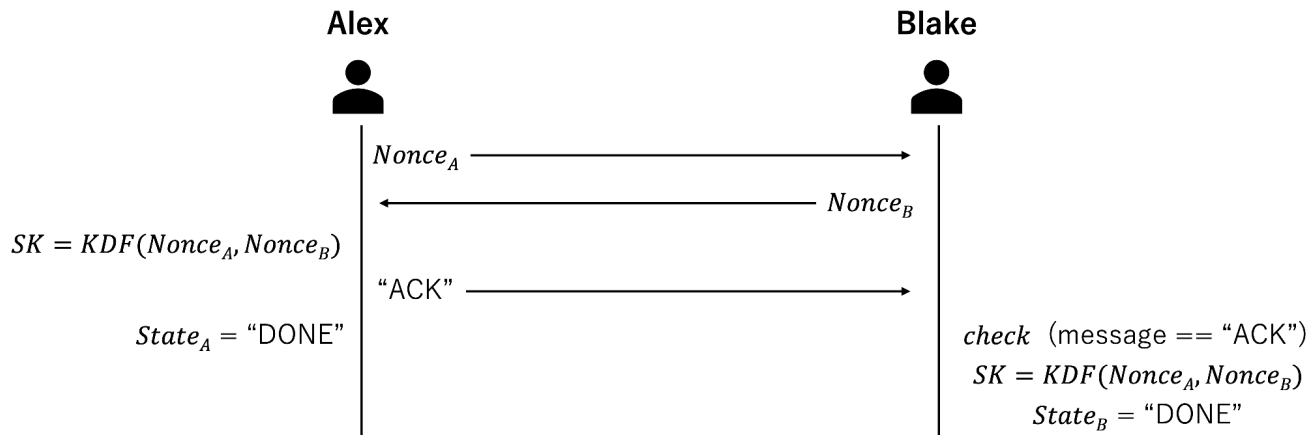
初期プロトコル

1. AlexがNonceを生成し、Blakeへ送信： $\text{Alex} \rightarrow \text{Blake} (\text{Nonce}_A)$
2. BlakeがNonceを生成し、Alexへ送信： $\text{Blake} \rightarrow \text{Alex} (\text{Nonce}_B)$
3. AlexはBlakeから Nonce_B を受け取ると、以下の3つの手順を実行
 - i. Alexは Nonce_A と Nonce_B を KDF に入力 $\rightarrow SK$ を導出： $SK = KDF(\text{Nonce}_A, \text{Nonce}_B)$
 - ii. “ACK”という文字列を含むメッセージをBlakeに送信： $A \rightarrow B$ (“ACK”)
 - iii. 自身を「DONE」状態に遷移： $\text{State}_A = \text{“DONE”}$
4. Blakeは“ACK”メッセージを受信すると、以下の2つの手順を実行
 - i. Blakeが SK を計算： $SK = KDF(\text{Nonce}_A, \text{Nonce}_B)$
 - ii. 自身も「DONE」状態に遷移： $\text{State}_B = \text{“DONE”}$

(※)

SK ：セッション鍵

KDF ：鍵導出関数



4.2 主な記号（変数や定数に対する接頭辞）

記号	発音	意味・用法	例
!	パン	永続Factを示す。ルールが発火しても削除されない読み取り専用トークン	!Pk(A, pkA)
~	チルダ	フレッシュ変数 (通信毎に都度生成される) を示す。Fr(~x) で新しい乱数や鍵として生成され、ルール外では通常の変数 x として扱われる	~k, ~ltk
\$	ダラー	公開名変数を示す。プロトコル内の定数としてインスタンス化される	

チルダが付く変数と
そうでない変数に注意

時刻を表すタイムスタンプ変数

【使い方】タイムスタンプの変数宣言時に、先頭に # を付与

```
/* 5-2 認証 (weak agreement) */
/* 通信を完了した主体 A が「相手 B と話した」と判断したとき、B が確かに直前に A と話していたことが保証される */
lemma Client_auth:
  " /* For all sessions keys 'k' setup by clients with a server 'S' */
    ( All S k #i. SessKeyC(S, k) @ #i
      ==>
        /* there is a server that answered the request */
        ( (Ex #a. AnswerRequest(S, k) @ a)
          /* or the adversary performed a long-term key reveal on 'S'
            before the key was setup. */
          | (Ex #r. LtkReveal(S) @ r & r < i)
        )
    )
  "
```

1回宣言すれば、
以降の使用では #をつける必要なし
(どちらでも可)

よく使われるタイムスタンプの変数名

(Tamarinに組み込まれてるわけではなく、ただの慣習)

変数名	意味
#i	セッション鍵生成の時刻
#j	比較対象となる他の時刻
#a	AnswerRequestイベントの時刻
#r	LtkRevealイベントの時刻

演習3 コードとプロトコルの対応関係を確認しよう

[code3.spthy](#) は前ページの初期プロトコルのTamarinモデルです
コードとプロトコルを見比べて、対応関係を確認してみましょう
また、code3.spthy を実行して検証結果を見てみましょう

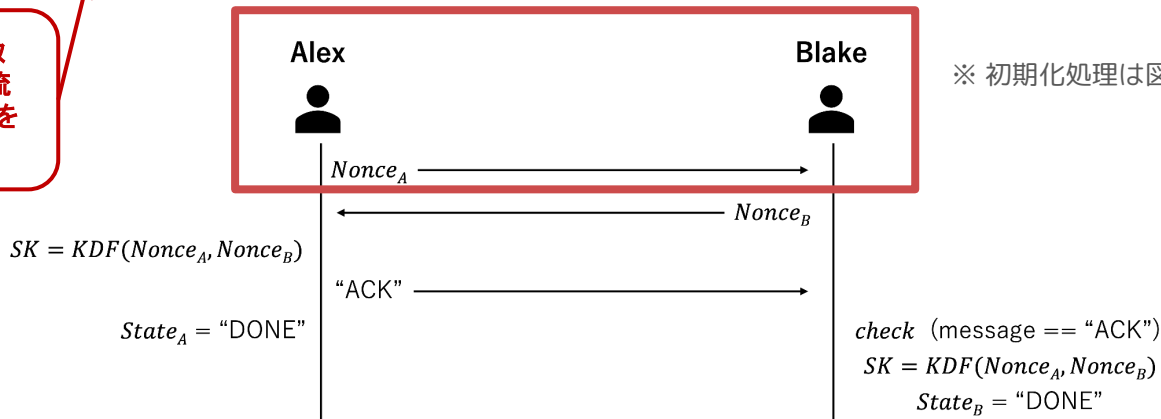
演習3 コードとプロトコルの対応関係を確認しよう

解説

```
9 rule Init: 初期化処理 // 初期化ルール
10   [ Fr(~aID), Fr(~bID) ] // 乱数aIDと乱数bIDを新しく生成
11   --[ Init(~aID, ~bID) ]->
12   [ AState(~aID, 'INIT', 'EMPTY_STATE') // Aの状態をINITとして定義
13     , BState(~bID, 'INIT', 'EMPTY_STATE') ] // Bの状態をINITとして定義
14
15 rule ASendNonce: Aのメッセージ送信
16   [ AState(~aID, 'INIT', 'EMPTY_STATE') // Aが初期状態ならばこのルールを適用
17     , Fr(~ANonce) ] // 乱数ANonceを新しく生成
18   --[ ASendNonce(~aID, ~ANonce) ]->
19   [ AState(~aID, 'SENT_NONCE', ~ANonce) // Aの状態をSENT_NONCE更新
20     , Out(~ANonce) ] // 通信路にANonceを送信
```

Tamarinではフレッシュな値
(乱数等)を生成するルール
が予め組み込まれており、
そのファクトを消費すること
で新しい値を生成できる

In(通信路から値を受け取る)とOut(通信路に値を流す)でメッセージの送受信を表現

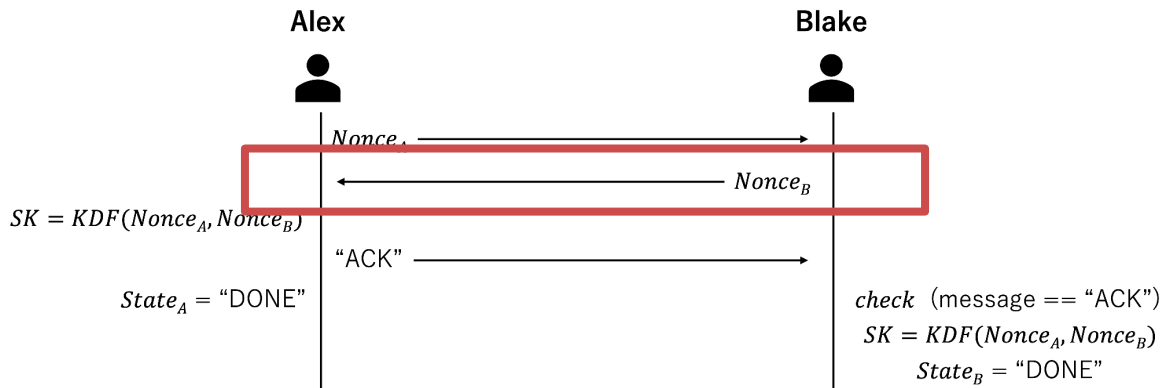


演習3 コードとプロトコルの対応関係を確認しよう

解説

Bのメッセージ受取・Bのメッセージ送信

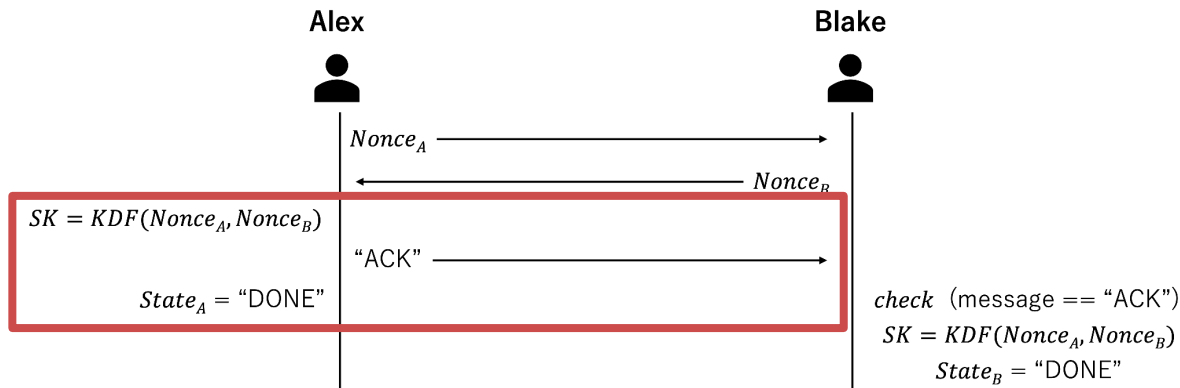
```
32 rule BReceiveNonceSendNonce:
33   [ BState(~bID, 'INIT', 'EMPTY_STATE') // Bの状態がINITかつ通信路から何かを受け取れる (35行目) ならばこのルールを適用
34     , Fr(~BNonce) // 乱数BNonceを新しく生成
35     , In(ANonce) ]
36   --[ BReceivesNonceSendsNonce(~bID, ANonce, ~BNonce) ]->
37   [ BState(~bID, 'SENT_NONCE', <ANonce, ~BNonce>) // Bの状態をSENT_NONCEに更新
38     , Out(~BNonce) ] // 通信路にBNonceを送信
```



演習3 コードとプロトコルの対応関係を確認しよう

解説

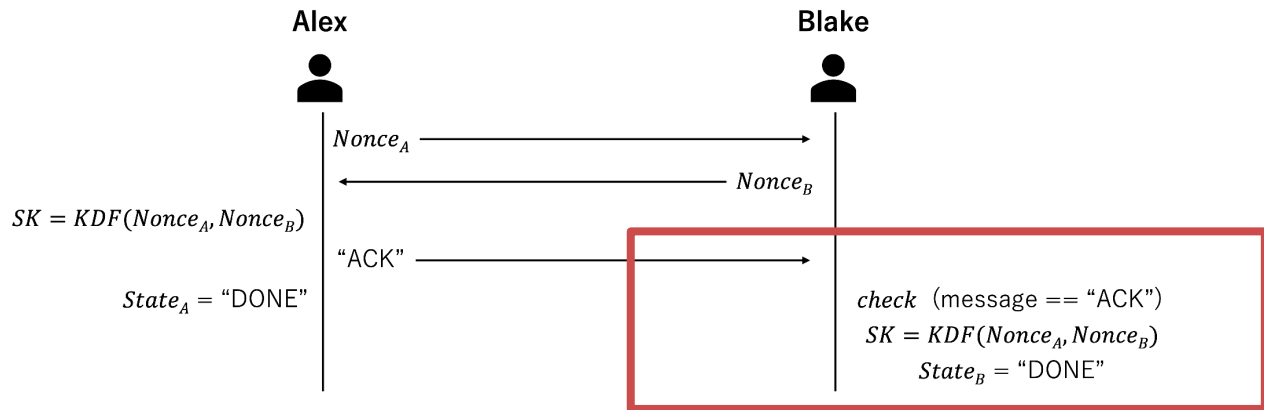
```
22 rule AReceiveNonceInstallKey: Aのメッセージ受取・AのSK導出・Aのメッセージ送信（2回目）
23   let
24     SK = kdf(~ANonce, BNonce) // SKを定義
25   in
26     [ AState(~aID, 'SENT_NONCE', ~ANonce) // Aの状態がSENT_NONCE かつ
27       , In(BNonce) ] // 通信路から何かを受け取れるならばこのルールを適用
28     --[ AInstallsKey(~aID, ~ANonce, BNonce, SK) ]->
29     [ AState(~aID, 'INSTALLED_KEY', <~ANonce, BNonce, SK>) // Aの状態をINSTALLED_KEYに更新
30       , Out('ACK') ] // 通信路にACKを送信
```



演習3 コードとプロトコルの対応関係を確認しよう

解説

```
40 rule BReceiveAckInstallKey: Bのメッセージ受取(2回目)・BのSK導出
41   let
42     SK = kdf(ANonce, ~BNonce)           // SKを定義
43   in
44     [ BState(~bID, 'SENT_NONCE', <ANonce, ~BNonce>) // Bの状態がSENT_NONCE かつ
45       , In('ACK') ]                       // 通信路から'ACK'を受け取れるならばこのルールを適用
46   --[ BInstallsKey(~bID, ANonce, ~BNonce, SK) ]->
47   [ BState(~bID, 'INSTALLED_KEY', <ANonce, ~BNonce, SK>) ] // Bの状態をINSTALLED_KEYに更新
```



演習3 コードとプロトコルの対応関係を確認しよう

解説 安全性要件

```
53 lemma successful_run: exists-trace // プロトコルが最後まで通ることの確認（エラー対策用）
54   "Ex bID aID ANonce BNonce SK #i #j.
55     AInstallsKey(bID, ANonce, BNonce, SK) @ i &
56     BInstallsKey(aID, ANonce, BNonce, SK) @ j"
57
58 lemma sk_secret_a: // Aが確立したSKの機密性の検証
59   "All aID ANonce BNonce SK #i. // 任意のaID ANonce BNonce SK #iに対して,
60     AInstallsKey(aID, ANonce, BNonce, SK) @ i // Aが(aID, ANonce, BNonce, SK)に対して鍵を導出した,
61     ==> not(Ex #j. K(SK) @ j)" // ならば, 攻撃者はSKを手に入れることはできない
62
63 lemma sk_secret_b: // Bが確立したSKの機密性の検証
64   "All bID ANonce BNonce SK #i. // 任意のbID ANonce BNonce SK #iに対して,
65     BInstallsKey(bID, ANonce, BNonce, SK) @ i // Bが(bID, ANonce, BNonce, SK)に対して鍵を導出した,
66     ==> not(Ex #j. K(SK) @ j)" // ならば, 攻撃者はSKを手に入れることはできない
67
68 lemma if_b_finishes_a_has_finished_too: // 明示的な一方認証性の検証
69   "All bID ANonce BNonce SK #i. // 任意のbID ANonce BNonce SK #iに対して,
70     BInstallsKey(bID, ANonce, BNonce, SK) @ i // Bが(bID, ANonce, BNonce, SK)に対して鍵を導出した,
71     ==> (Ex aID #j. AInstallsKey(aID, ANonce, BNonce, SK) @ j // ならば, それよりも前の時点で必ずAも(aID, ANonce, BNonce, SK)に対して鍵を導出している
72       & j < i)"
```

機密性は「攻撃者が秘密を知っている状態に到達しない」という要件として検証する

$K(SK)$ …攻撃者がSKを知っている状態

認証性は「一方がある鍵を導出した状態に到達可能であるならば, その状態よりも前にもう一方が同じ鍵を導出する状態が存在する」という要件として検証する

ラベルに変数を追加することで, 「同じ鍵」を表現可能

(参考) Tamarinにおける攻撃者の能力

いわゆるDolev-Yaoモデル

- **公開通信路の情報を全て盗聴可能**
- **自分の知り得る情報**（公開通信路からの情報，公開情報，それらから公開関数の適用によって作ることができる情報）**を用いて，任意のタイミングで公開通信路にメッセージを挿入可能**

組み込まれているルール

```
rule (modulo AC) isend:  
  [ !KU( x ) ] --[ K( x ) ]-> [ In( x ) ]  
攻撃者がxを知っている      ならば      通信路にxを流せる
```

```
rule (modulo AC) irecv:  
  [ Out( x ) ] --> [ !KD( x ) ]  
通信路にxが流れている      ならば      攻撃者はxを知っている
```

演習3 コードとプロトコルの対応関係を確認しよう

解説 検証結果

```
successful_run (exists-trace): verified (6 steps)
sk_secret_a (all-traces): falsified - found trace (5 steps)
sk_secret_b (all-traces): falsified - found trace (5 steps)
if_b_finishes_a_has_finished_too (all-traces): falsified - found trace
(3 steps)
```

上から,

- (プロトコルが正しく最後まで実行できること,)
- Aが導出したSKの機密性が成り立たないこと,
- Bが導出したSKの機密性が成り立たないこと,
- 明示的な認証性が成り立たないこと

を意味します

初期プロトコルでは、中間者攻撃者が $Nonce_A$ および $Nonce_B$ を取得し、

それらからセッション鍵を計算が可能

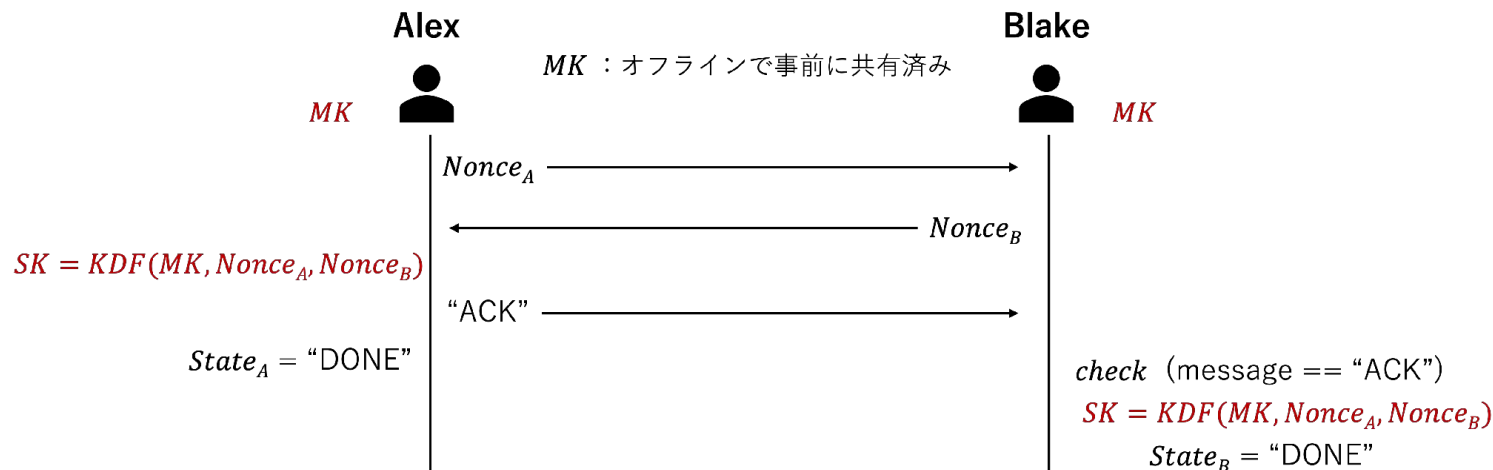
→ Alex と Blake がプロトコル実行前にあらかじめマスター鍵 MK を共有

(※) ここでは、マスター鍵が事前にオフラインで共有済みであると仮定

実際には、Alex と Blake が事前に合意した単純なパスワード等

セッション鍵の計算時には、このマスター鍵 MK を組み込むものとする：

$$SK = KDF(MK, Nonce_A, Nonce_B)$$



演習4 事前共有マスター鍵を組み込もう

[code4.spthy](#) の***部分を埋めて、事前共有マスター鍵を組み込み、検証結果がどのようになるか見てみましょう

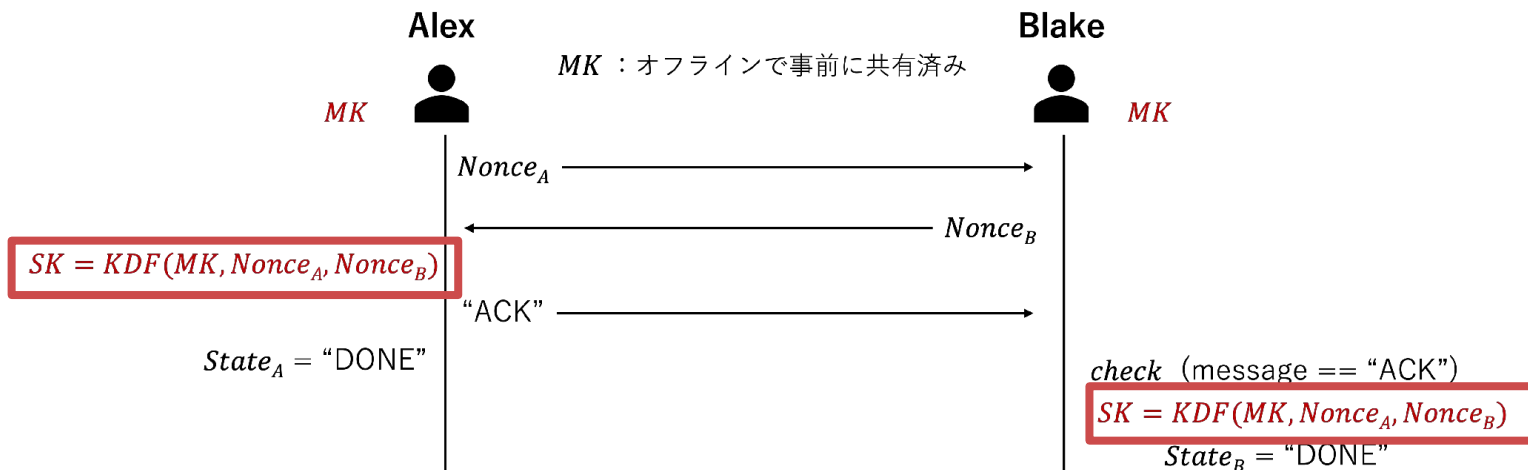
事前共有マスター鍵は ~MK としてモデリングされています

演習4 事前共有マスター鍵を組み込もう

解説

24 $SK = \text{kdf}(\sim MK, \sim A\text{Nonce}, B\text{Nonce})$

42 $SK = \text{kdf}(\sim MK, A\text{Nonce}, \sim B\text{Nonce})$



演習5 検証要件を追加してみよう

[code5.spthy](#) に，事前共有マスター鍵を組み込んだプロトコル（code4）に対する明示的な認証性に対応するlemmaを記述してみましょう

- 明示的な認証性：
Blake が “DONE” 状態に到達した場合，Alex もすでに（Blakeより先に） “DONE” 状態に到達している

初期プロトコルのlemma（↓）を参考に書いてみてください

```
lemma if_b_finishes_a_has_finished_too:  
  "All bID ANonce BNonce SK #i.  
    BInstallsKey(bID, ANonce, BNonce, SK) @ i  
    ==> (Ex aID #j. AInstallsKey(aID, ANonce, BNonce, SK) @ j  
      & j < i)"
```

検証結果を確認し，なぜそのような結果が出るのかを考えてみてください

演習5 検証要件を追加してみよう

解説

```
lemma if_b_finishes_a_has_finished_too:  
  "All bID MK ANonce BNonce SK #i.  
    BInstallsKey(bID, MK, ANonce, BNonce, SK) @ i  
    ==> (Ex aID #j. AInstallsKey(aID, MK, ANonce, BNonce, SK) @ j  
          & j < i)"
```

code4でのlemmaと比べて、MKについての記載が追加されます

検証結果

SKの機密性は満たしますが、明示的な認証性はまだ満たされません

```
successful_run (exists-trace): verified (6 steps)  
sk_secret_a (all-traces): verified (4 steps)  
sk_secret_b (all-traces): verified (4 steps)  
if_b_finishes_a_has_finished_too (all-traces): falsified - found trace (3 steps)
```

拡張:メッセージ認証コード (MAC)

前回のプロトコルでは、攻撃者が“ACK”メッセージを送信することで、

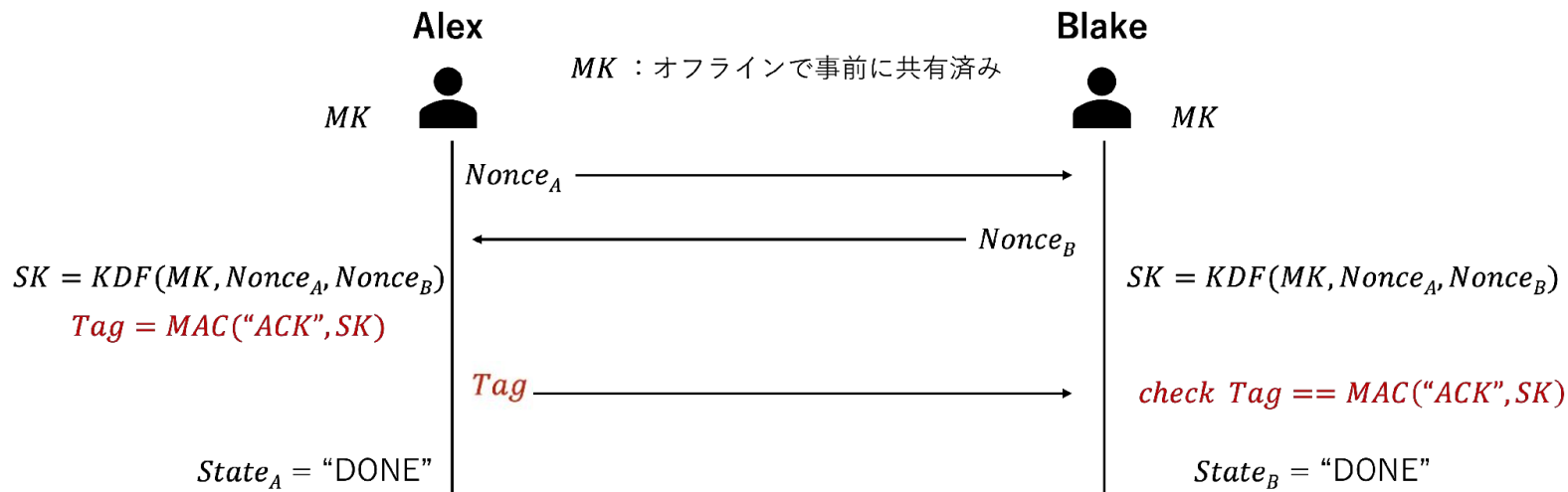
Blake に Alex がセッション鍵をインストール済みだと誤認させることが可能

これを防ぐため、Alex は“ACK”メッセージに対して SK を用いてMACを計算 → Blake に送信

Blake はメッセージとMACを受信した後、まずMACを検証し、成功したら“DONE”状態に遷移

これにより、攻撃者は“ACK”メッセージの偽造が困難

ここまでは implicit な認証の話でした
ここからは明示的な (explicit) 認証にしていきます



演習6 MACを組み込もう

[code6.spthy](#) の***部分を埋めてMACを組み込んでみましょう

MACの関数は `mac(X,Y)` と記述することで、定数はシングルコーテーションで囲うことで使うことができます

"ACK" → 'ACK'

組み込み後、実行して機密性と明示的な認証性についての検証結果を見てみましょう

演習6 MACを組み込もう

解説

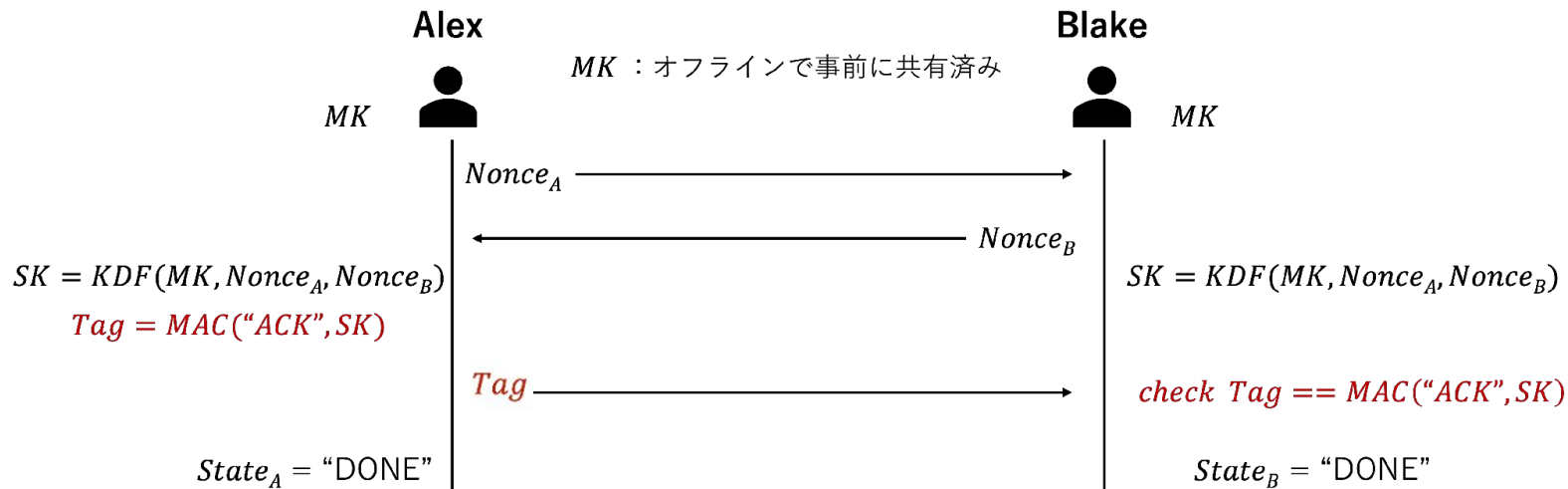
```
rule AReceiveNonceInstallKey:
  let
    SK = kdf(~MK, ~ANonce, BNonce)
  in
  [ AState(~aID, 'SENT_NONCE', <~MK, ~ANonce>)
    , In(BNonce) ]
  --[ AInstallsKey(~aID, ~MK, ~ANonce, BNonce, SK) ]->
  [ AState(~aID, 'INSTALLED_KEY', <~MK, ~ANonce, BNonce, SK>)
    , Out(mac('ACK', SK)) ]
```

明示的な認証性も満たすようになりました！

```
successful_run (exists-trace): verified (7 steps)
sk_secret_a (all-traces): verified (4 steps)
sk_secret_b (all-traces): verified (4 steps)
if_b_finishes_a_has_finished_too (all-traces): verified (6 steps)
```

※コードはハンズオンのページにあります **72**

SKの機密性, 明示的な認証性の両方を満たす



応用演習 1 MACの中身を変えてみたら

MACの中身をSKからMKにしてみましょう

どういう結果になるか確認してみましょう
また、なぜこうなるのか考えてみましょう

応用演習 1 MACの中身を変えてみたら

解説

また明示的な認証性を満たさなくなりました

```
successful_run (exists-trace): verified (7 steps)
sk_secret_a (all-traces): verified (4 steps)
sk_secret_b (all-traces): verified (4 steps)
if_b_finishes_a_has_finished_too (all-traces): falsified - found trace (4 steps)
```

MKを使ったTagは複数のセッションで使い回すことができるため、このような結果になっています

応用演習 2 双方向認証にしてみよう

今までに考えたプロトコルは一方向（片側）認証のプロトコルでした

このプロトコルを修正し，双方向認証にしてみましょう

また，双方向認証に対応するlemmaを追加し，検証結果を確認しましょう

おわりに

このハンズオンでは、暗号プロトコルの形式検証ツールTamarin Proverの基本について紹介しました

興味を持っていたただけた方は、公式マニュアルや関連研究のコードを見てもらえればと思います！

皆さんの研究に活用してもらえれば幸いです

参考：Tamarin公式ページ <https://tamarin-prover.com/>

参加者アンケート

FWSにご参加いただきありがとうございました！

より良いワークショップにしていくため、参加者アンケートへのご協力をお願いいたします

