Unfolding based model checking

Keijo Heljanko

Department of Information and Computer Science
Aalto University, School of Science and Technology
Keijo.Heljanko@tkk.fi

Joint work with: Prof. Javier Esparza
Faculty of Computer Science, Technische Universität München
esparza@in.tum.de

22.9-2010
Tutorial material

- Tutorial mainly based on the book
  Esparza, J. and Heljanko, K.: Unfoldings –
  A Partial-Order Approach to Model Checking.
  EATCS Monographs in Theoretical Computer Science,
  Final book draft available from:
  http://users.ics.tkk.fi/kepa/publications/
  Unfoldings-Esparza-Heljanko.pdf

- Please consult the book for bibliographical and historical
  information about the unfolding method

- Using material from: Unfolding-Based Model Checking
  Tutorial by Javier Esparza given in the UFO’07: Workshop
  on UnfOIding and partial order techniques
  (with permission)
Introduction to model checking - Software failures

Two very expensive software bugs:

► Intel Pentium FDIV bug (1994, approximately $500 million)
► Ariane 5 floating point overflow (1996, approximately $500 million)
Pentium FDIV - Software bug in HW

4195835 - ((4195835 / 3145727) * 3145727) = 256

The floating point division algorithm uses an array of constants with 1066 elements. However, only 1061 elements of the array were correctly initialized
Exploded 37 seconds after takeoff - the reason was an overflow in a conversion of a 64 bit floating point number into a 16 bit integer
Models and properties

- System
  - Modelling
  - System model
    - Executing the model
      - Kripke structure $M$
      - Model checking $M \models \psi$?
  - Property
    - Formalization of property
      - Formalized property $\psi$
In model checking every execution of the model of the system is simulated obtaining a Kripke structure $M$ (labelled version of the reachability graph) describing all its behaviors. $M$ is then checked against a property $\psi$:

- Yes: The system functions according to the specified property (denoted $M \models \psi$)
- No: The system is incorrect (denoted $M \not\models \psi$), a counterexample is returned: an execution of the system which does not satisfy the property
Model checking in the industry

- **Microprocessor design**: Several major microprocessor manufacturers use model checking methods as a part of their design process.

- **Design of Data-communications Protocol Software**: Model checkers have been used as rapid prototyping systems for new data-communications protocols under standardization.

- **Mission Critical Software**: NASA space program is model checking code used by the space program.

- **Operating Systems**: Microsoft is using model checking to verify the correct use of locking primitives in Windows device drivers.

- **Safety Critical Systems**: Model checking is used to find bugs in many safety critical systems.
Part I: Basics of unfolding based model checking

- This part of the tutorial will introduce the unfolding method and its use to model check reachability properties
- Basic knowledge of Petri nets is assumed
- To demonstrate the basic concepts, we limit ourselves to products of transition systems that are equivalent in expressive power to 1-safe (1-bounded) Petri nets
The state explosion problem

- The number of reachable states of a concurrent system can grow exponentially in the number of its components.
- Hinders conventional model checking even for relatively small systems.
- Many different approaches to combat the state explosion exist: e.g., partial order reduction methods (ample, stubborn and persistent sets).
- The compression approach: find compact symbolic but still manageable representations for sets of states.
  - Binary Decision Diagrams. Exploit regularities of the state space.
  - Unfoldings: Exploit concurrency.
Transition systems

A transition system is a tuple $\mathcal{A} = \langle S, T, \alpha, \beta, is \rangle$, where

- $S$ is a set of states,
- $T$ is a set of transitions,
- $\alpha : T \rightarrow S$ associates to each transition its source state,
- $\beta : T \rightarrow S$ associates to each transition its target state, and
- $is \in S$ is the initial state
Example

Transition system $\mathcal{A} = \langle S, T, \alpha, \beta, is \rangle$ where

- $S = \{s_1, s_2, s_3, s_4\}$, $T = \{t_1, t_2, t_3, t_4, t_5\}$,
- $\alpha(t_1) = s_1$, $\beta(t_1) = s_2$, ..., $\beta(t_5) = s_1$,
- $is = s_1$
Products of transition systems

A product of transition systems is a tuple \( \langle A_1, \ldots, A_n, T \rangle \) where

- \( A_1, \ldots, A_n \) are transition systems called components, and
- \( T \) is a synchronization constraint

A synchronization constraint is a set of tuples of the form \( \langle u_1, u_2, \ldots, u_n \rangle \) where \( u_i \) is either a transition of \( A_i \) or the special idling symbol \( \epsilon \)

Example: \( \langle t_1, \epsilon, \epsilon, t_2 \rangle \)

The tuples are called global transitions
Peterson’s mutex

```
UNFOLDINGS
22.9-2010
14/119

Peterson's mutex

Aalto University
School of Science
and Technology

Unfoldings
22.9-2010
14/119
```
Running example

\[ T = \{ \langle t_1, \epsilon \rangle, \langle t_2, \epsilon \rangle, \langle t_3, u_2 \rangle, \langle t_4, u_2 \rangle, \langle t_5, \epsilon \rangle, \langle \epsilon, u_1 \rangle, \langle \epsilon, u_3 \rangle \} \]
Petri net representation of products

\[
\begin{align*}
\langle t_1, \epsilon \rangle & \quad \langle t_2, \epsilon \rangle & \quad \langle \epsilon, u_1 \rangle \\
\langle t_3, u_2 \rangle & \quad \langle t_4, u_2 \rangle \\
\langle \epsilon, u_3 \rangle
\end{align*}
\]
Interleaving representation of products

\[
\langle s_1, r_1 \rangle \langle t_5, \epsilon \rangle \\
\langle t_1, \epsilon \rangle \langle s_1, r_1 \rangle \langle t_2, \epsilon \rangle \\
\langle s_2, r_1 \rangle \langle t_1, \epsilon \rangle \langle e, u_1 \rangle \langle s_3, r_1 \rangle \\
\langle s_2, r_2 \rangle \langle t_2, \epsilon \rangle \langle s_3, r_2 \rangle \langle t_4, u_2 \rangle \\
\langle t_3, u_2 \rangle \langle s_4, r_3 \rangle \langle t_5, \epsilon \rangle \\
\langle s_1, r_3 \rangle \langle t_5, \epsilon \rangle \langle s_4, r_1 \rangle \\
\langle s_2, r_3 \rangle \langle s_3, r_3 \rangle \langle s_4, r_2 \rangle \\
\langle e, u_3 \rangle \\
\langle e, u_3 \rangle \\
\langle e, u_3 \rangle \\
\langle t_5, \epsilon \rangle
\]
Unfolding transition systems: Computation tree
Unfolding as a Petri net
Unfolding products
Unfolding products

\[ \langle t_1, \epsilon \rangle \langle t_2, \epsilon \rangle \langle \epsilon, u_1 \rangle \langle t_3, u_2 \rangle \langle t_4, u_2 \rangle \langle \epsilon, u_3 \rangle \]
Unfolding products

\[ \langle t_3, u_2 \rangle \rightarrow \langle t_4, u_2 \rangle \rightarrow \langle t_5, \epsilon \rangle \]

\[ \langle t_1, \epsilon \rangle \rightarrow \langle t_2, \epsilon \rangle \rightarrow \langle \epsilon, u_1 \rangle \]

\[ \langle t_1, \epsilon \rangle \rightarrow \langle \epsilon, u_3 \rangle \rightarrow \langle \epsilon, u_1 \rangle \]
Unfolding products

Unfolding products
Unfolding products

\[
\langle t_1, \epsilon \rangle \quad s_1 \quad r_1 \\
\langle t_1, \epsilon \rangle \quad s_2 \quad \langle t_2, \epsilon \rangle \\
\langle t_3, u_2 \rangle \quad s_3 \quad r_2 \\
\langle t_4, u_2 \rangle \quad s_4 \quad r_3 \\
\langle t_5, \epsilon \rangle
\]

\[
\langle t_1, \epsilon \rangle \quad s_1 \\
\langle t_2, \epsilon \rangle \quad s_2 \\
\langle t_3, u_2 \rangle \quad s_3 \\
\langle t_4, u_2 \rangle \\
\langle t_5, \epsilon \rangle \\
\langle t_2, \epsilon \rangle \\
\langle \epsilon, u_1 \rangle
\]

\[
\langle t_1, \epsilon \rangle \quad s_1 \\
\langle t_2, \epsilon \rangle \quad s_2 \\
\langle t_3, u_2 \rangle \\
\langle t_4, u_2 \rangle \\
\langle \epsilon, u_3 \rangle \\
\langle \epsilon, u_1 \rangle \\
\langle \epsilon, u_1 \rangle
\]

\[
\langle t_1, \epsilon \rangle \quad s_1 \\
\langle t_2, \epsilon \rangle \\
\langle \epsilon, u_1 \rangle
\]

\[
\langle t_1, \epsilon \rangle \quad s_1 \\
\langle t_2, \epsilon \rangle \\
\langle \epsilon, u_1 \rangle
\]
Unfolding products

\[
\langle t_3, u_2 \rangle \quad \langle t_4, u_2 \rangle
\]

\[
\langle t_5, \epsilon \rangle \quad \langle t_1, \epsilon \rangle \quad \langle t_2, \epsilon \rangle \quad \langle \epsilon, u_1 \rangle
\]

\[
\langle t_1, \epsilon \rangle \quad \langle t_2, \epsilon \rangle \quad \langle \epsilon, u_1 \rangle \quad \langle \epsilon, u_3 \rangle
\]

\[
\langle t_3, u_2 \rangle \quad \langle t_4, u_2 \rangle \quad \langle t_1, \epsilon \rangle \quad \langle \epsilon, u_1 \rangle
\]

\[
\langle t_3, u_2 \rangle \quad \langle t_4, u_2 \rangle \quad \langle t_1, \epsilon \rangle \quad \langle \epsilon, u_1 \rangle
\]
The transitions of the unfolding are events

The places of the unfolding are conditions
Given a node $x$ (place or event) of the unfolding, we denote the label of $x$ by $\lambda(x)$. Furthermore, given a set $X$ of nodes we define $\lambda(X) = \{\lambda(x) | x \in X\}$.

**Proposition**

Let $s$ be a reachable state of product $A$, and let $M$ be a reachable marking of the unfolding of $A$ such that $\lambda(M) = s$.

(a) If $\langle M, e, M' \rangle$ is a step of the unfolding, then there is a step $\langle s, t, s' \rangle$ of $A$ such that $\lambda(e) = t$, and $\lambda(M') = s'$.

(b) If $\langle s, t, s' \rangle$ is a step of $A$, then there is a step $\langle M, e, M' \rangle$ of the unfolding such that $\lambda(e) = t$, and $\lambda(M') = s'$. 
Corollary of the fundamental property

Corollary

(a) If $\sigma$ is a (finite or infinite) occurrence sequence of the unfolding, then $\lambda(\sigma)$ is a history of $A$.

(b) If $h$ is a history of $A$, then some occurrence sequence of the unfolding satisfies $\lambda(\sigma) = h$. 
Unfoldings are synchronizations of trees

An unfolding of a transition system is a tree. Intuitively, an unfolding of a product can be seen as a synchronization of trees. We formalize this idea.

**Definition**

A place of an unfolding is an *i-place* if it is labeled by a state of the *i*th component. The *i-root* is the unique *i*-place having no input events. An event is an *i*-event if it is labeled by a global transition $\langle t_1, \ldots, t_n \rangle$ such that $t_i \neq \varepsilon$. In other words, an event is an *i*-event if the *i*th component participates in the global transition it is labeled with.

It follows that an *i*-place can only be a *j*-place if $j = i$; on the contrary, an event can be an *i*-event and a *j*-event even for $i \neq j$ if both $A_i$ and $A_j$ participate in the transition it is labeled with.
Properties of Unfoldings

Proposition

Let $\mathcal{N}$ be the unfolding of $A$. Then:

1. $\mathcal{N}$ has no cycles, i.e., no (nonempty) path of arcs leads from a node to itself.

2. For every $i \in \{1, \ldots, n\}$, every reachable marking of $\mathcal{N}$ puts a token in exactly one $i$-place.

3. The set of $i$-nodes of the unfolding $\mathcal{N}$ forms a tree with the $i$-root as root. Moreover, the tree only branches at places, i.e., if a node of the tree has more than one child, then it is a place.

4. A place of $\mathcal{N}$ can get marked at most once (i.e., if along an occurrence sequence it becomes marked and then unmarked, then it never becomes marked again), and an event of $\mathcal{N}$ can occur at most once in an occurrence sequence.
The Unfolding (recap)
Causality, conflict, and concurrency

Definition

Let \( x \) and \( y \) be two nodes of an unfolding.

- We say that \( x \) is a *causal predecessor* of \( y \), denoted by \( x < y \), if there is a (non-empty) path of arcs from \( x \) to \( y \); as usual we denote by \( x \leq y \) that either \( x < y \) or \( x = y \); two nodes \( x \) and \( y \) are *causally related* if \( x \leq y \) or \( x \geq y \)

- We say that \( x \) and \( y \) are in *conflict*, denoted by \( x \# y \), if there is a place \( z \), different from \( x \) and \( y \), from which one can reach \( x \) and \( y \), exiting \( z \) by different arcs

- We say that \( x \) and \( y \) are *concurrent*, denoted by \( x \text{ co } y \), if \( x \) and \( y \) are neither causally related nor in conflict
Concurrent conditions are simultaneously reachable

Proposition

Let $\mathcal{N}$ be an unfolding of $\mathbf{A}$ and let $P$ be a set of places (conditions) of $\mathcal{N}$. There is a reachable marking $M$ of $\mathcal{N}$ such that $P \subseteq M$ if and only if the places of $P$ are pairwise concurrent.
Proposition

(1) For every two nodes $x$, $y$ of an unfolding exactly one of the following holds: (a) $x$ and $y$ are causally related, (b) $x$ and $y$ are in conflict, (c) $x$ and $y$ are concurrent.

(2) If $x$ and $y$ are causally related and $x \neq y$, then either $x < y$ or $y < x$, but not both.
Proposition

Let $\mathcal{N}$ be an unfolding of a product $\mathbf{A}$ and let $C$ be a set of events of $\mathcal{N}$.

1. $C$ is a configuration if and only if it is causally closed, i.e., if $e \in C$ and $e' < e$ then $e' \in C$, and conflict-free, i.e., no two events of $C$ are in conflict.

Note that all the firing sequences of transitions realizing a configuration $C$ lead to the same reachable marking of $\mathcal{N}$, which is a set of conditions called the cut $\text{Cut}(C)$. 
Model checking

The model checking problem:
Does some run of the system satisfy a given property $\psi$?

Some important instances:

1. **Executability**: Does some run contain a given transition?
2. **Repeated executability**: Does some run contain a given transition infinitely often?
3. **Livelock**: Does some run contain an infinite tail of “silent” transitions?

Fact:
The model-checking problem for next-free LTL-formulas can be reduced to (2) and (3), for safety properties to (1). We use search procedures to construct prefixes of the unfolding that decide (1), (2), and (3).
Search procedures

A search procedure consists of:

▶ a search scheme:
  ▶ **Termination condition**: Determines which leaves of the current prefix are terminals, i.e., nodes whose successors need not be explored. (Terminals are also called cut-offs.)
  ▶ **Success condition**: Determines which terminals are successful, i.e., terminals proving that $\psi$ holds.
  ▶ **Search strategy**: Tells which event should be added next (nondeterministic search strategies allowed).
Search procedure pseudo-code for products

\[\text{procedure unfold(product } A) \{\]
\[\quad \mathcal{N} := \text{net containing only the initial marking from } A \text{ without events};\]
\[\quad T := \emptyset; \quad S := \emptyset; \quad X := \text{Ext}(\mathcal{N}, T); \quad /* \text{Compute possible extensions } */\]
\[\quad \text{while } (X \neq \emptyset) \{\]
\[\quad \quad \text{choose a (minimal) event } e \in X \text{ according to the search strategy};\]
\[\quad \quad \text{extend } \mathcal{N} \text{ with } e;\]
\[\quad \quad \text{if } e \text{ is a terminal according to the search scheme then } \{\]
\[\quad \quad \quad T := T \cup \{e\};\]
\[\quad \quad \quad \text{if } e \text{ is successful according to the search scheme then } \{\]
\[\quad \quad \quad \quad S := S \cup \{e\}; \quad /* \text{A successful terminal found, add early exit here!} */\]
\[\quad \quad \} ;\]
\[\quad X := \text{Ext}(\mathcal{N}, T); \quad /* \text{Compute possible extensions } */\]
\[\quad \} ;\]
\[\quad \text{return } \langle \mathcal{N}, T, S \rangle; \quad /* \text{return } \langle \text{prefix, terminals, successful terminals} \rangle */\]
\[\}\]

/* */
A search procedure for executability in transition systems

Search procedure to decide if some run executes a goal transition \( g \).

Search scheme:

- An event is a terminal if
  1. it is labeled by \( g \) or,
  2. it leads to the same state as some other event we have already seen.

- A terminal is successful if it is of type (1).

Search strategy: Any.

It is easy to prove that all these search procedures (same search scheme, different strategies e.g., DFS or BFS) are all correct (always terminate with the right outcome but maybe different set of explored nodes).
Example

\[
\begin{align*}
S_1 & \quad S_2 \\
S_2 & \quad S_3 \\
S_3 & \quad S_4 \\
S_4 & \quad S_1
\end{align*}
\]
Example

Diagram with nodes $s_1$, $s_2$, $s_3$, $s_4$ and edges labeled with $g$. Additional diagram on the right with nodes $s_1$ and $s_2$. 
Example
Example
Example
Example
Example
Second example with \( g = \{ t_5 \} \)
Second example: Two prefixes

(a)

(b)

\((a)\)

\((b)\)
We want something like this:
An event is a terminal if
(1) it is labeled by $g$ (and then it is successful) or,
(2) “it leads to the same global state as some other event we have already seen.”
But an event of an unfolding does not necessarily lead to one global state!

Solution: attach to an event $e$ the global state reached by “executing its past”. (McMillan ’92,’95)

This is the global state reached by firing the local configuration on an event.
Running Example as Petri net
Example

\[ \langle s_2, r_1 \rangle \langle s_1, r_2 \rangle \langle s_3, r_2 \rangle \langle s_4, r_4 \rangle \langle s_4, r_3 \rangle \langle s_4, r_4 \rangle \langle s_3, r_1 \rangle s_1 \]
Example

$$\langle s_2, r_1 \rangle \langle s_1, r_2 \rangle$$

$$\langle s_1, r_1 \rangle$$

$$\langle s_2, r_1 \rangle$$

$$\langle s_3, r_1 \rangle$$

$$\langle s_4, r_1 \rangle$$
Example

\[ \langle s_2, r_1 \rangle \]

\[ \langle s_3, r_1 \rangle \]

\[ s_1 \]

\[ s_2 \]

\[ r_1 \]
Example

\[ \langle s_2, r_1 \rangle \]
Example

\[ \langle s_2, r_1 \rangle \]
\[ \langle s_1, r_2 \rangle \]
Example

\[ \langle s_2, r_1 \rangle \langle s_1, r_2 \rangle \]
Example

\[ \langle s_2, r_1 \rangle \rightarrow \langle s_2, r_1 \rangle \rightarrow \langle s_4, r_3 \rangle \rightarrow \langle s_4, r_3 \rangle \rightarrow \langle s_1, r_2 \rangle \rightarrow \langle s_1, r_2 \rangle \rightarrow r_1 \rightarrow r_1 \]

\[ s_2 \rightarrow s_4 \rightarrow r_3 \rightarrow r_3 \rightarrow \]
Example

\[ \langle s_2, r_1 \rangle \langle s_1, r_2 \rangle \]

\[ \langle s_4, r_3 \rangle \]

\[ s_1 \]

\[ r_1 \]

\[ s_2 \]

\[ s_3 \]

\[ r_2 \]

\[ s_4 \]

\[ r_3 \]
Example

\[ \langle s_2, r_1 \rangle \rightarrow \langle s_3, r_1 \rangle \rightarrow \langle s_1, r_2 \rangle \]

\[ \langle s_4, r_3 \rangle \rightarrow \langle s_2, r_1 \rangle \rightarrow \langle s_3, r_1 \rangle \rightarrow \langle s_1, r_2 \rangle \]
Example

\[ \langle s_2, r_1 \rangle \quad \langle s_3, r_1 \rangle \quad \langle s_1, r_2 \rangle \]

\[ \langle s_4, r_3 \rangle \quad \langle s_4, r_3 \rangle \]

\[ s_1 \quad r_1 \quad s_2 \quad s_3 \quad r_2 \quad s_4 \quad r_3 \]
Example

\[
\langle s_2, r_1 \rangle \rightarrow s_2 \rightarrow \langle s_4, r_3 \rangle \rightarrow s_4 \leftarrow r_3 \leftarrow s_4 \leftarrow \langle s_4, r_3 \rangle
\]

\[
\langle s_3, r_1 \rangle \rightarrow s_3 \rightarrow \langle s_1, r_2 \rangle \rightarrow s_1 \rightarrow \langle s_1, r_2 \rangle
\]

\[
s_2 \rightarrow r_1 \rightarrow s_3 \rightarrow r_2 \rightarrow s_4
\]
Generalization to products: search strategies

- A search strategy determines which is the next event to be added to the current prefix. But it has to be defined before knowing the events!
- In the transition system case, an event is characterized by its past, the unique transition sequence leading to it.
- Search strategy $\rightarrow$ (partial) order $\triangleleft \subseteq T^* \times T^*$ that refines the prefix order.
Two search strategies for $w, w' \in T^*$:

1. $w \prec w'$ if $|w| < |w'|$

2. $w \prec w'$ if $w$ is alphabetically smaller than $w'$
Generalization to products: search strategies

- A search strategy determines which is the next event to be added to the current prefix. But it has to be defined before knowing the events!
- In the transition system case, an event is characterized by its past, the unique transition sequence leading to it.
- Search strategy $\rightarrow$ (partial) order $\prec \subseteq T^* \times T^*$ that refines the prefix order.
- In the product case, an event is also characterized by its past, but the past may consist of many transition sequences!
Example

The past of event labelled $\langle t_3, u_2 \rangle$ are the transition sequences:

- $w_1 = \langle t_1, \varepsilon \rangle \langle \varepsilon, u_1 \rangle \langle t_3, u_2 \rangle$
- $w_2 = \langle \varepsilon, u_1 \rangle \langle t_1, \varepsilon \rangle \langle t_3, u_2 \rangle$
Generalization to products: search strategies

- A search strategy determines which is the next event to be added to the current prefix. But it has to be defined before knowing the events!
- In the transition system case, an event is characterized by its past, the unique transition sequence leading to it.
- Search strategy $\rightarrow$ (partial) order $\prec \subseteq T^* \times T^*$ that refines the prefix order.
- In the product case, an event is also characterized by its past, but the past may consist of many transition sequences!
- Solution: these sequences are a Mazurkiewicz trace.
- Search strategy $\rightarrow$ (partial) order $\prec \subseteq [T^*] \times [T^*]$ defined on Mazurkiewicz traces that refines the prefix order.
Mazurkiewicz traces

- Two global transitions of a product are independent if no component participates in both of them.
- **Example**: \(\langle t_1, \epsilon \rangle\) and \(\langle \epsilon, u_1 \rangle\) are independent, \(\langle t_1, \epsilon \rangle\) and \(\langle t_3, u_2 \rangle\) are not.

- Two sequences of global transitions are equivalent if the one can be obtained from the other by repeatedly swapping adjacent independent transitions.

- **Example**: \(\langle t_1, \epsilon \rangle \langle \epsilon, u_1 \rangle \langle t_3, u_2 \rangle \sim \langle \epsilon, u_1 \rangle \langle t_1, \epsilon \rangle \langle t_3, u_2 \rangle\)

- Mazurkiewicz trace: equivalence classes of sequences.

- **Example**: \([\langle t_1, \epsilon \rangle \langle \epsilon, u_1 \rangle \langle t_3, u_2 \rangle] = \{ \langle t_1, \epsilon \rangle \langle \epsilon, u_1 \rangle \langle t_3, u_2 \rangle, \langle \epsilon, u_1 \rangle \langle t_1, \epsilon \rangle \langle t_3, u_2 \rangle \}\)
Search procedure for executability of $\langle t_5, \epsilon \rangle$
Are these search procedures correct?

Not for every strategy!!
(Counterexample by Esparza, Römer and Vogler)
\( \mathbf{T} = \{ a = \langle a_1, a_2, a_3, a_4 \rangle, b = \langle b_1, b_2, b_3, b_4 \rangle, c = \langle c_1, c_2, \epsilon, \epsilon \rangle, \\
\quad d = \langle \epsilon, \epsilon, d_3, d_4 \rangle, e = \langle e_1, e_2, \epsilon, \epsilon \rangle, f = \langle \epsilon, \epsilon, f_3, f_4 \rangle, \\
\quad g = \langle g_1, \epsilon, g_3, \epsilon \rangle, h = \langle \epsilon, h_2, \epsilon, h_4 \rangle, i = \langle i_1, i_2, i_3, i_4 \rangle \} \)

\( \mathbf{G} = \{ i \} \)
Which are the correct strategies?

- Sufficient condition: adequate strategies
- Mazurkiewicz traces can be concatenated in the obvious way: \([w][w'] \overset{\text{def}}{=} [ww']\)
- Definition: A strategy \(\prec\) on Mazurkiewicz traces is adequate if
  1. it is well-founded
     (there is no infinite descending chain: \(\cdots \prec [w_2] \prec [w_1] \prec [w_0]\)) and
  2. \([w'] \prec [w]\) implies \([w'][w''] \prec [w][w'']\)
     (preservation by extensions).
- (Lemma [Chatain and Khomenko]: (1) \(\rightarrow\) (2).)
- Theorem: If the strategy is adequate, then the search procedure is correct.
Proof idea:

- To prove: if \( g \) can be executed, then the search procedure explores some trace \([u \ g]\).
- If \( g \) can be executed, then the full unfolding contains some trace \([w \ g]\).
- If \([w \ g]\) is not explored by the procedure, then \( w \) it contains a terminal event \( e_1 \) with past \([w_1]\) such that there exists another trace \([w'_1]\) such that:
  - \([w \ g] = [w_1 \ w_2 \ g]\),
  - \([w_1]\) leads to the same global state as the trace \([w'_1] \prec [w_1]\).
- But then \([w'_1 \ w_2 \ g] \prec [w_1 \ w_2 \ g]\). Iterating the procedure, and by the well-foundedness of \(\prec\), we finally reach some trace \([u' \ g]\) that is explored by the procedure.
Are there total adequate strategies?

▶ **Fact 1**: Every total adequate strategy on transition sequences can be lifted to a total adequate strategy on Mazurkiewicz traces.

▶ **Fact 2**: The following strategy is adequate and total on transition sequences:

\[ w_1 \prec w_2 \iff \begin{align*}
|w_1| &< |w_2|, \text{ or;} \\
|w_1| &= |w_2| \text{ and } w_1 \text{ is lexicographically smaller than } w_2.
\end{align*} \]

▶ There are many other total adequate strategies!

(First one found by Vogler, reported in: Esparza, Römer and Vogler, An improvement of McMillan’s unfolding algorithm)
Implementing an adequate strategy

- Please see the book for additional adequate total search strategies
- One has to basically implement a comparison operator for two Mazurkiewicz traces - tricky code but needs to be only written once, see e.g., unfsmodels-0.9/eventq.cc/EventQueue::CompareERV()
- Use orders implemented by other tools to ease tool cross-comparisons for both benchmarking and debugging!
- In order to do this, aim for source-code compability, sometimes tool and a paper about it are not fully compliant with each other
Recap: Search procedure

```plaintext
procedure unfold(product A) {
  \( N := \) net containing only the initial marking from A without events;
  \( T := \emptyset; S := \emptyset; X := Ext(N, T); */ Compute possible extensions */
  while (X \( \neq \emptyset \)) {
    choose a (minimal) event \( e \in X \) according to the search strategy;
    extend \( N \) with \( e \);
    if \( e \) is a terminal according to the search scheme then {
      \( T := T \cup \{e\}; \)
      if \( e \) is successful according to the search scheme then {
        \( S := S \cup \{e\}; */ A successful terminal found, add early exit here!*/ \)
      };
    };
    \( X := Ext(N, T); */ Compute possible extensions */ \)
  }; 
  return \( \langle N, T, S \rangle \); /* return \( \langle prefix, terminals, successful terminals \rangle \) */
};
```
Computing possible extensions

Computing potential extensions uses 90%+ percent of running time in the unfolding procedures.

Let $k$ be the maximum in-degree of transitions and $n$ be the number of conditions in the prefix before calling the possible extensions subroutine.

- **Memory-intensive approach**: Maintain the co-relation between any two conditions. Takes $O(n^2)$ memory and takes $O(n^k / k^{k-2})$ time. Also updating the co-relation takes $O(n)$ time for each added condition.

- **Memory-light approach**: Enumerate all potential extension without any co-relation using $O(n)$ memory but $O(n^{k+1} / k^k)$ time.

- **More refined search approach**: Preset trees (Khomenko)

- **Solver approach**: Employ an NP solver to compute the potential extensions.
Computing possible extensions is NP-complete

A decision version of computing the possible extensions is NP-complete in the size of the prefix. Consider the 3SAT formula \( \phi = ((x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_3)) \):
Computing possible extensions is NP-complete

A partial prefix of the system. Now $t$ is in the possible extensions iff $\phi$ is satisfiable.
Optimal modelling style for the unfolding method

- Lots of concurrency is good, always try to avoid synchronizing independent subsystems “by accident” when modelling.
- Using low in-degree of transitions when modelling systems will make life easier for possible extensions subroutine, and will on average speed up unfolding.
- Try to avoid local non-determinism, as it can lead to combinatorial explosion when multiple processes synchronize.
Canonical prefixes

> All notations defined for unfoldings also carry over to prefixes. However, terminal events (cut-off events) are not to be included into any configuration of the prefix.

> If we generate a prefix with a sound (i.e., correct) search strategy we will generate a finite prefix that is identical up to isomorphism for any (total) prefix extension order compatible with the (partial order) search strategy.

> This unique prefix has been named the canonical prefix (theory by Khomenko, Koutny, and Vogler). The theory also allows several generalizations of search procedures.

> From a canonical finite prefix any simple reachability questions such as deadlock detection can be solved using an NP-solver.

> Trade-off between compactness and query complexity.
Reachability of global states

<table>
<thead>
<tr>
<th>Product/1-safe PN</th>
<th>Canonical prefix</th>
<th>Interleaving</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSPACE-complete</td>
<td>NP-complete</td>
<td>Linear</td>
</tr>
</tbody>
</table>
Reachability of local states

<table>
<thead>
<tr>
<th>Product/1-safe PN</th>
<th>Canonical prefix</th>
<th>Interleaving</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSPACE-complete</td>
<td>Linear</td>
<td>Linear</td>
</tr>
</tbody>
</table>

Reachability of local states with prefixes is even more efficient than reachability of global states. However, current NP-solvers such as SAT solvers also make NP-hard problems on prefixes quite tractable in practice.
Checking for dead transitions (product)

\[ T = \{ a = \langle a_0, a_1, \epsilon, \epsilon, \epsilon \rangle, \ b_1 = \langle \epsilon, b_1, \epsilon, \epsilon, \epsilon \rangle, \ b_2 = \langle \epsilon, \epsilon, b_2, \epsilon, \epsilon \rangle, \ b_3 = \langle \epsilon, \epsilon, \epsilon, b_3, \epsilon \rangle, \ b_4 = \langle \epsilon, \epsilon, \epsilon, \epsilon, b_4 \rangle, \ c = \langle c_0, c_1, c_2, c_3, c_4 \rangle \} \]
Checking for dead transitions (Petri net)
Checking for dead transitions (unfolding)
Canonical finite prefix sizes

Prefixes can be often smaller than the state space, and in case of total search strategies prefixes have never more (non-terminal/non-cutoff) events than reachable states.

<table>
<thead>
<tr>
<th>Problem(size)</th>
<th>S</th>
<th>T</th>
<th>B</th>
<th>E</th>
<th>#c</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPD(5)</td>
<td>45</td>
<td>45</td>
<td>1582</td>
<td>790</td>
<td>211</td>
<td>3488</td>
</tr>
<tr>
<td>DPD(6)</td>
<td>54</td>
<td>54</td>
<td>3786</td>
<td>1892</td>
<td>499</td>
<td>19860</td>
</tr>
<tr>
<td>DPD(7)</td>
<td>63</td>
<td>63</td>
<td>8630</td>
<td>4314</td>
<td>1129</td>
<td>109964</td>
</tr>
<tr>
<td>DPH(5)</td>
<td>48</td>
<td>67</td>
<td>2712</td>
<td>1351</td>
<td>547</td>
<td>3112</td>
</tr>
<tr>
<td>DPH(6)</td>
<td>57</td>
<td>92</td>
<td>14590</td>
<td>7289</td>
<td>3407</td>
<td>16896</td>
</tr>
<tr>
<td>DPH(7)</td>
<td>66</td>
<td>121</td>
<td>74558</td>
<td>37272</td>
<td>19207</td>
<td>79926</td>
</tr>
<tr>
<td>ELEVATOR(2)</td>
<td>146</td>
<td>299</td>
<td>1562</td>
<td>827</td>
<td>331</td>
<td>1061</td>
</tr>
<tr>
<td>ELEVATOR(3)</td>
<td>327</td>
<td>783</td>
<td>7398</td>
<td>3895</td>
<td>1629</td>
<td>7120</td>
</tr>
<tr>
<td>ELEVATOR(4)</td>
<td>736</td>
<td>1939</td>
<td>32354</td>
<td>16935</td>
<td>7337</td>
<td>43439</td>
</tr>
<tr>
<td>FURNACE(1)</td>
<td>27</td>
<td>37</td>
<td>535</td>
<td>326</td>
<td>189</td>
<td>343</td>
</tr>
<tr>
<td>FURNACE(2)</td>
<td>40</td>
<td>65</td>
<td>4573</td>
<td>2767</td>
<td>1750</td>
<td>3777</td>
</tr>
<tr>
<td>FURNACE(3)</td>
<td>53</td>
<td>99</td>
<td>30820</td>
<td>18563</td>
<td>12207</td>
<td>30860</td>
</tr>
<tr>
<td>RING(7)</td>
<td>91</td>
<td>77</td>
<td>813</td>
<td>403</td>
<td>79</td>
<td>16999</td>
</tr>
<tr>
<td>RING(9)</td>
<td>117</td>
<td>99</td>
<td>1599</td>
<td>795</td>
<td>137</td>
<td>211527</td>
</tr>
</tbody>
</table>
A canonical finite prefix can be very succinct

The class of Petri nets containing the following representative for $n = 4$ has a state space of size $2^n$ but a prefix of linear size in the parameter $n$:

The prefix is identical to the original net system!
A canonical finite prefix can be very large

A worst case is a system with no concurrency but lots of non-determinism.
A canonical finite prefix
Reducing reachability to SAT or ILP

\[
\begin{array}{c|c}
\rho & \phi_\rho \\
\alpha & \alpha \leftrightarrow \neg e_1 \\
\beta & \beta \leftrightarrow \neg e_1 \\
\gamma & ((e_3 \lor e_4) \rightarrow e_1) \land \neg(e_3 \land e_4) \land \\
 & (\gamma \leftrightarrow (e_1 \land \neg e_3 \land \neg e_4)) \\
\delta & ((e_2 \lor e_6) \rightarrow e_1) \land \neg(e_2 \land e_6) \land \\
 & (\delta \leftrightarrow (e_1 \land \neg e_2 \land \neg e_6)) \\
\epsilon & \epsilon \leftrightarrow e_2 \\
\zeta & \zeta \leftrightarrow e_3 \\
\eta & (e_6 \rightarrow e_4) \land (\eta \leftrightarrow (e_4 \land \neg e_6)) \\
\kappa & ((e_8 \lor e_9) \rightarrow e_6) \land \neg(e_8 \land e_9) \land \\
 & (\kappa \leftrightarrow (e_6 \land \neg e_8 \land \neg e_9)) \\
\lambda & \lambda \leftrightarrow e_6 \\
\mu & \mu \leftrightarrow e_8 \\
\nu & \nu \leftrightarrow e_9 \\
\end{array}
\]
SAT encoding

- A conjunction of all the formulas for the conditions gives a formula encoding all reachable configurations of the prefix.
- It is easy to project this on the markings of the original net by introducing variables for the original places of the net and adding to the formula a conjunction for each place of the original net:

\[
\begin{align*}
  s_1 & \iff (\alpha \lor \zeta \lor \mu) \\
  & \quad \vdots \\
  r_2 & \iff \delta
\end{align*}
\]

- Now a global state marking both \( s_1 \) and \( r_2 \) can be reached if after conjuncting formula with \((s_1 \land r_2)\) has a satisfying truth assignment.
- Deadlock detection is just another reachability property.
Deadlock checking running time

Unfolding much slower than deadlock detection (old results but the trend is still the same). Fastest tools currently are PUnf (unfolding) and CLP (reachability) by Victor Khomenko

<table>
<thead>
<tr>
<th>Problem(size)</th>
<th>DL</th>
<th>Unf</th>
<th>ERVunfold</th>
<th>DCmcsmodels -n</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPD(5)</td>
<td>N</td>
<td></td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>DPD(6)</td>
<td>N</td>
<td></td>
<td>0.5</td>
<td>0.3</td>
</tr>
<tr>
<td>DPD(7)</td>
<td>N</td>
<td></td>
<td>2.2</td>
<td>0.8</td>
</tr>
<tr>
<td>DPH(5)</td>
<td>N</td>
<td></td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>DPH(6)</td>
<td>N</td>
<td></td>
<td>4.1</td>
<td>1.3</td>
</tr>
<tr>
<td>DPH(7)</td>
<td>N</td>
<td></td>
<td>101.7</td>
<td>11.3</td>
</tr>
<tr>
<td>ELEVATOR(2)</td>
<td>Y</td>
<td></td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>ELEVATOR(3)</td>
<td>Y</td>
<td></td>
<td>1.3</td>
<td>0.2</td>
</tr>
<tr>
<td>ELEVATOR(4)</td>
<td>Y</td>
<td></td>
<td>27.4</td>
<td>1.0</td>
</tr>
<tr>
<td>FURNACE(1)</td>
<td>N</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>FURNACE(2)</td>
<td>N</td>
<td></td>
<td>0.4</td>
<td>0.1</td>
</tr>
<tr>
<td>FURNACE(3)</td>
<td>N</td>
<td></td>
<td>14.3</td>
<td>1.1</td>
</tr>
<tr>
<td>RING(7)</td>
<td>N</td>
<td></td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>RING(9)</td>
<td>N</td>
<td></td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>RW(9)</td>
<td>N</td>
<td></td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>RW(12)</td>
<td>N</td>
<td></td>
<td>25.3</td>
<td>2.2</td>
</tr>
</tbody>
</table>
This part of the tutorial will cover the advanced unfolding based model checking research issues.

This is aimed at students having a good basic knowledge of unfoldings as well as at researcher looking for open problems with unfoldings of products/1-safe Petri nets.

We discuss advanced topics on unfoldings to attract interest in open research problems in the area.

Disclaimer: This is a highly personal view of unfoldings research directions.
Minimizing canonical prefixes

Note that in the proof on slide 74 the corresponding configuration $w'_1$ did not have to be a local configuration. **Out of curiosity**, we can try and see what happens to canonical prefix size if we allow for non-local corresponding configurations, thus “minimizing prefixes” as allowed by the canonical prefix theory (Heljanko: Minimizing finite complete prefixes)

<table>
<thead>
<tr>
<th>Problem(size)</th>
<th>Original Prefix</th>
<th>Minimal Prefix</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>B</td>
<td>$</td>
</tr>
<tr>
<td>BDS(1)</td>
<td>12310</td>
<td>6330</td>
<td>3701</td>
</tr>
<tr>
<td>DPD(6)</td>
<td>3786</td>
<td>1892</td>
<td>499</td>
</tr>
<tr>
<td>DPD(7)</td>
<td>8630</td>
<td>4314</td>
<td>1129</td>
</tr>
<tr>
<td>DPH(6)</td>
<td>14590</td>
<td>7289</td>
<td>3407</td>
</tr>
<tr>
<td>DPH(7)</td>
<td>74558</td>
<td>37272</td>
<td>19207</td>
</tr>
<tr>
<td>FURNACE(2)</td>
<td>4573</td>
<td>2767</td>
<td>1750</td>
</tr>
<tr>
<td></td>
<td>30820</td>
<td>18563</td>
<td>12207</td>
</tr>
<tr>
<td>FURNACE(3)</td>
<td>529</td>
<td>302</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>704</td>
<td>402</td>
<td>1</td>
</tr>
<tr>
<td>HART(75)</td>
<td>260</td>
<td>146</td>
<td>0</td>
</tr>
<tr>
<td>HART(100)</td>
<td>371</td>
<td>206</td>
<td>0</td>
</tr>
<tr>
<td>DAC(12)</td>
<td>533</td>
<td>266</td>
<td>40</td>
</tr>
<tr>
<td>DAC(15)</td>
<td>608</td>
<td>291</td>
<td>40</td>
</tr>
<tr>
<td>SENT(75)</td>
<td>529</td>
<td>302</td>
<td>1</td>
</tr>
<tr>
<td>SENT(100)</td>
<td>704</td>
<td>402</td>
<td>1</td>
</tr>
</tbody>
</table>
Complete finite prefixes

- We can define the prefix “completeness” fully semantically as was classically done before the canonical prefixes/search procedures ways of defining prefixes. This leads to a different definition of what prefixes are!

A prefix is complete if it is both marking and transition complete:

- A prefix is marking complete if every reachable global state is represented by a configuration of the prefix
- A prefix is transition complete if every enabled global transition exists as a (non-terminal or terminal) event of the prefix

- It is unknown whether prefixes generated by search procedures have structural properties that would make algorithms working on them easier than algorithms working on complete prefixes as defined above!
Beyond reachability properties

- Intuition: Prefixes are best suited for reachability properties
- Safety properties are also easily handled by prefixes if the property only needs to observe the firing of a small number of visible transitions $V \subseteq T$
- The strategy: Compile the negation of a safety property $\psi$ to a finite state automaton $A_{\neg \psi}$
- Add the automaton $A_{\neg \psi}$ as a new component that observes the firing of all transitions in $V$ by synchronizing with all of them (sequentializing the firing of transitions in $V$)
- If a condition labelled with the final state of the observer automaton is added to the prefix, the safety property is violated
- Note: If $V = T$ the prefix is guaranteed to be as large as the state space of the system!
Model checking without observers

- Linear time temporal logic properties can also be model checked with unfoldings but with significant complications
- We first want to show that using an observer component is probably a good idea for any nested reachability questions
- To demonstrate the complications, we will use the simplest possible nested reachability temporal logic question and we will try to avoid using any observer components
Model checking nested reachability

- Let us first consider a simple question: Does the system have an execution which is a witness for the CTL formula: $EF(q_0 \land EF(q_F))$ where $q_0$ and $q_F$ are local states of a component.
- This formula holds iff the system has an execution to a global state where $q_0$ holds, and from there some global state where $q_F$ holds can be reached.
- Note: The negation of this safety property can be encoded using a 3-state observer component shown below:
Example: A binary counter for three bits

We model random initialization at asynchronous circuit power on for all bits and carries of an asynchronous three bit counter.
Some observations from the counter

- The initialization is extremely parallel → should be good for unfolding based methods. Also max in-degree is small.
- All the reachable states of the counter system can be reached using some subset of the random initialization transitions enabled in the initial state!
- Therefore, all events on the “second level of the unfolding” can be made into cut-off events (using the semantic notion of prefix completeness!) without making the prefix incomplete.
- In other words: All information about the reachable states and enabled transition of the original net remains in this compact prefix.
- Note: We are cheating a bit here, no practical prefix generation algorithm has been able to create this compact prefix allowed by the semantic definition of completeness.
Compact prefix for three bit binary counter
Here we exploit **the semantic definition of prefix completeness!**
Results for counter systems of different numbers of bits.

<table>
<thead>
<tr>
<th>System Size</th>
<th></th>
<th>Markings</th>
<th>Arcs</th>
<th>McMillan/ERV Prefix</th>
<th>Compact Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>10</td>
<td>27</td>
<td>66</td>
<td>43 23 7 16</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>15</td>
<td>108</td>
<td>351</td>
<td>105 55 16 39</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>20</td>
<td>405</td>
<td>1620</td>
<td>225 116 30 86</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>25</td>
<td>1458</td>
<td>6885</td>
<td>453 231 50 181</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>30</td>
<td>5103</td>
<td>27702</td>
<td>887 449 77 372</td>
</tr>
<tr>
<td>7</td>
<td>29</td>
<td>35</td>
<td>17496</td>
<td>107163</td>
<td>1721 867 112 755</td>
</tr>
</tbody>
</table>

Note: The table compares the sizes of prefixes for different system sizes, showing the number of markings, arcs, and the compact prefix sizes.
More observations

- The prefix is of same size as the original 1-safe net system.
- Any asynchronous circuit model with parallel random initialization will have a similar compact prefix.
- Now let us use another system in place of the binary counter.
Turing machines with linearly bounded tape

- A Turing machine with a linearly bounded tape can be simulated by a 1-safe Petri net which is only polynomially larger.
- Thus any question on such Turing machines can be reduced to a question on 1-safe Petri nets that are at most polynomially larger.
- For example, the following question is PSPACE-complete: Does Turing machine with linearly bounded tape accept on some initial tape contents and initial read head position?
Turing machine simulator net (black box)

Turing machine simulator Petri net
Adding random initialization to the Turing machine

Random initialization

Turing machine simulator Petri net
Observations

► In a complete prefix all the Turing machine simulator transitions are terminal (cut-off) events

► A fully random initialization is not what is wanted, as then the final state $qF$ of a Turing machine can always be reached, i.e., the simulation “cheats” by going directly to the accepting state of the Turing machine.

► The “non-cheating” simulations of the Turing machine are exactly those where the execution satisfies the CTL formula: $EF(q0 \land EF(qF))$, where $q0$ is the correct initial state of the Turing machine.
Model checking compact prefixes is PSPACE hard

- Because the complete prefix is also of polynomial size in the size of the Turing machine we have: Model checking of nested temporal properties is PSPACE-complete in the size of the complete prefix! (Recall, we are using the semantic definition of prefix completeness from slide 97!)

- Open research question: Does this PSPACE-completeness result also hold for prefixes created by search procedures?¹

¹As defined on slide 38 combined with any adequate order for the search strategy.
Adding an observer

If we would add the following observer component to the prefix, the problem would become a local state reachability question of the state $s_2$ of the observer, and thus the problem would become linear in the size of the prefix:

Thus sequentializing very few transitions of the original net can create exponential size changes in the compact prefix!
Second open problem

Formulation of another problem of very similar nature:

- Given a prefix generated by a search procedure\(^2\), decide if the original product/1-safe net can execute some infinite sequence of global transitions.

- Is this problem PSPACE-complete in the size of the prefix?

- My conjecture: Yes, it is. (Many colleagues disagree. . . )

\(^2\)As defined on slide 38 combined with any adequate order for the search strategy.
Model checking linear temporal logic (LTL)

A quick summary on how to do LTL model checking with unfoldings starting from a product $A$

- Restrict to LTL without the next-time operator (LTL-X): Otherwise the need to synchronize with all transitions leads to no concurrency and no savings from unfoldings

- Translate the negation of the LTL-X property $\psi$ to Büchi automaton $\mathcal{A}_{\neg\psi}$

- Find the set of visible transition $V$ that can change the value of atomic propositions, synchronize $\mathcal{A}_{\neg\psi}$ as an observer component with all transitions in $V$, call the resulting product $P$

- Detect the “bad infinite behaviours” of $A$ that are executions violating $\psi$ using $P$ as input to the unfolding procedure
Bad infinite behaviours of \( A \)

There are two classes of bad infinite behaviours of \( A \)

(1) The bad behaviour executes infinitely many visible transitions in \( V \): This reduces to the repeated executability problem for a subset of transitions \( R \) of \( P \). (Basically all transition of \( A_{\neg \psi} \) that go to an accepting Büchi state.)

(2) The bad behaviour executes only finitely many visible transitions in \( V \): This reduces to the livelock problem for a subset of transitions \( L \) of \( P \) and a set of visible transitions \( V \). (One needs to analyze the structure of \( A_{\neg \psi} \) to identify the transitions after which a livelock of invisible transitions would result in bad infinite behaviour.)

This is basically the temporal testers approach of Antti Valmari but used in combination with unfoldings. See the book for details.
DFS and unfoldings do not mix well

- The repeated executability problem is basically the generalization of the Büchi emptiness checking problem for transition systems (automata).
- The Büchi emptiness checking problem can be solved in linear time for automata. However, all linear time algorithms are depth-first-search (DFS) based.
- DFS based search strategies are incompatible with the search procedure on slide 38 (result by Esparza, Kanade, and Schwoon). Thus non-DFS based emptiness checking algorithm is needed!
- Going to non-DFS algorithms makes the size of the prefix for the repeated executability problem of size $O(n^2)$, where $n$ is the number of reachable states of the system$^3$

$^3$Doing a DFS compatible unfolding procedure is potentially possible by using a larger than $O(n^2)$ prefix. We do not consider this a viable possibility.
Third open problem

- Open problem: Come up with a unfolding based LTL model checker with better than $O(n^2)$ prefix size, where the complexity of the algorithm run on the prefix is easier than PSPACE-hard in the size of the prefix.
A search procedure for repeated executability

This procedure has a “BFS” emptiness checker flavor to it, the livelock problem has a similar algorithmic solution:

▶ Given an event \( e \), let \( \#_g e \) be the number of occurrences of \( g \) in the past of \( e \).

▶ Search scheme:
An event \( e \) is a terminal if there is an event \( e' \prec e \) leading to the same state as \( e \) and such that either

(1) \( e' \) is a causal predecessor of \( e \), or
(2) \( e \) is not a causal predecessor of \( e \), and \( \#_g e' \geq \#_g e \).

▶ A terminal is successful if it is of type (1) and some event between \( e' \) and \( e \) is labelled by \( g \).

▶ Search strategy: any adequate strategy.
Example: repeated executability of $\langle t_1, \epsilon \rangle$
Tutorial summary

- We have introduced unfoldings, a symbolic method to compactly represent the state space of the system using:
- The book bases its theory on a system model that can describe any synchronization of transition systems (automata)
- The unfolding theory of the book is built on top of the theory of Mazurkiewicz traces
- We show the algorithmic details of unfolding procedures and reachability checking with based on SAT solvers
- We discuss advanced topics on unfoldings to attract interest in open research problems in the area