ExoDPOR: Exogenous Stateless Model Checking

Lars Tveito, Einar Broch Johnsen and Rudolf Schlatte
August 26, 2021

Department of Informatics, University of Oslo
{lars,tvei,einarj,rudi}@ifi.uio.no
We present ExoDPOR, a framework for exogenous stateless model checking.
We present ExoDPOR, a framework for exogenous stateless model checking. Given a program and an initial state, it explores all non-equivalent executions.
We present ExoDPOR, a framework for exogenous stateless model checking
- Given a program and an initial state, it explores all non-equivalent executions
- ExoDPOR explores executions by invoking and controlling an external runtime using executable traces
We present ExoDPOR, a framework for exogenous stateless model checking.

- Given a program and an initial state, it explores all non-equivalent executions.
- ExoDPOR explores executions by invoking and controlling an external runtime using executable traces.
- We formalize requirements for the traces and relations that are communicated between ExoDPOR and the external runtime.
We present ExoDPOR, a framework for exogenous stateless model checking.

- Given a program and an initial state, it explores all non-equivalent executions.

- ExoDPOR explores executions by invoking and controlling an external runtime using executable traces.

- We formalize requirements for the traces and relations that are communicated between ExoDPOR and the external runtime.

- ExoDPOR can be instantiated for programming languages or systems supporting deterministic record and replay.
We present ExoDPOR, a framework for exogenous stateless model checking. Given a program and an initial state, it explores all non-equivalent executions. ExoDPOR explores executions by invoking and controlling an external runtime using executable traces. We formalize requirements for the traces and relations that are communicated between ExoDPOR and the external runtime. ExoDPOR can be instantiated for programming languages or systems supporting deterministic record and replay. The search orchestrated by ExoDPOR can be parallelized in a straightforward manner and scales over multiple machines.
We present ExoDPOR, a framework for exogenous stateless model checking.

- Given a program and an initial state, it explores all non-equivalent executions.
- ExoDPOR explores executions by invoking and controlling an external runtime using executable traces.
- We formalize requirements for the traces and relations that are communicated between ExoDPOR and the external runtime.
- ExoDPOR can be instantiated for programming languages or systems supporting deterministic record and replay.
- The search orchestrated by ExoDPOR can be parallelized in a straightforward manner and scales over multiple machines.
- We have instantiated ExoDPOR for Real-Time ABS.
Testing sequential programs is nice and reliable

Testing concurrent programs is hard and unreliable

Testing all executions of a concurrent program would be ideal

But is rarely feasible
Context

- Testing sequential programs is nice and reliable
  - Low risk of false positives

- Testing concurrent programs is hard and unreliable
  - High risk of false positives

- Testing all executions of a concurrent program would be ideal
  - But is rarely feasible
Testing sequential programs is nice and reliable
  - Low risk of false positives
Testing concurrent programs is hard and unreliable
Context

- Testing sequential programs is nice and reliable
  - Low risk of false positives
- Testing concurrent programs is hard and unreliable
  - High risk of false positives
Context

- Testing sequential programs is nice and reliable
  - Low risk of false positives
- Testing concurrent programs is hard and unreliable
  - High risk of false positives
- Testing all executions of a concurrent program would be ideal
Testing sequential programs is nice and reliable
  - Low risk of false positives
Testing concurrent programs is hard and unreliable
  - High risk of false positives
Testing all executions of a concurrent program would be ideal
  - But is rarely feasible
Related Work

- Stateless model checkers are tools for systematically exploring the execution paths of a concurrent program.

- Partial order reduction is crucial for avoiding many redundant executions.

- Dynamic partial order reduction is a state-of-the-art algorithm for stateless model checking.

- Requires a runtime with backtracking.

- DPOR algorithms are generally sequential and challenging to parallelize.
Related Work

- Stateless model checkers are tools for systematically exploring the execution paths of a concurrent program.
- *Partial order reduction* is crucial for avoiding many redundant executions.
Related Work

- Stateless model checkers are tools for systematically exploring the execution paths of a concurrent program
- *Partial order reduction* is crucial for avoiding many redundant executions
- *Dynamic partial order reduction* is a state-of-the-art algorithm for stateless model checking
Related Work

- Stateless model checkers are tools for systematically exploring the execution paths of a concurrent program.
- *Partial order reduction* is crucial for avoiding many redundant executions.
- *Dynamic partial order reduction* is a state-of-the-art algorithm for stateless model checking.
  - Requires a runtime with backtracking.
Related Work

- Stateless model checkers are tools for systematically exploring the execution paths of a concurrent program.
- *Partial order reduction* is crucial for avoiding many redundant executions.
- *Dynamic partial order reduction* is a state-of-the-art algorithm for stateless model checking.
  - Requires a runtime with backtracking.
  - DPOR algorithms are generally sequential and challenging to parallelize.
The main feature of ExoDPOR is that it is decoupled from the runtime.
The main feature of ExoDPOR is that it is decoupled from the runtime.

ExoDPOR allows us to benefit from the Erlang backend of Real-Time ABS.
The main feature of ExoDPOR is that it is decoupled from the runtime

ExoDPOR allows us to benefit from the Erlang backend of Real-Time ABS

A change in the Erlang backend does not imply that a change is needed in the ExoDPOR instantiation
ExoDPOR for Real-Time ABS

- The main feature of ExoDPOR is that it is decoupled from the runtime
- ExoDPOR allows us to benefit from the Erlang backend of Real-Time ABS
- A change in the Erlang backend does not imply that a change is needed in the ExoDPOR instantiation
- Much easier to support the full language
We assume a *labeled transition system* (LTS), \((S, \mathcal{E}, \rightarrow)\), where \(S\) is a set of states, \(\mathcal{E}\) a set of events, and \(\rightarrow \subseteq S \times \mathcal{E} \times S\) the transition relation.
LTS and Traces

- We assume a *labeled transition system* (LTS), $(\mathcal{S}, \mathcal{E}, \rightarrow)$, where $\mathcal{S}$ is a set of states, $\mathcal{E}$ a set of events, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$ the transition relation.

- We allow *unlabeled transitions*.

An execution is a sequence of transitions $\sigma_1 \xrightarrow{e_1} \sigma_2 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \sigma_{n+1}$ such that all events are distinct and such that $\sigma_{n+1}$ is a final state.

An execution trace is a sequence of events, denoted $\tau = e_1 \cdot e_2 \cdot \ldots \cdot e_n$. 

We assume a labeled transition system (LTS), \((\mathcal{S}, \mathcal{E}, \rightarrow)\), where \(\mathcal{S}\) is a set of states, \(\mathcal{E}\) a set of events, and \(\rightarrow \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}\) the transition relation.

- We allow unlabeled transitions.
- An execution is a sequence of transitions

\[
\sigma_1 \xrightarrow{e_1} \sigma_2 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \sigma_{n+1}
\]

such that all events are distinct and such that \(\sigma_{n+1}\) is a final state.
LTS and Traces

- We assume a labeled transition system (LTS), $(S, \mathcal{E}, \rightarrow)$, where $S$ is a set of states, $\mathcal{E}$ a set of events, and $\rightarrow \subseteq S \times \mathcal{E} \times S$ the transition relation.
- We allow unlabeled transitions.
- An execution is a sequence of transitions
  $$\sigma_1 \xrightarrow{e_1} \sigma_2 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \sigma_{n+1}$$
  such that all events are distinct and such that $\sigma_{n+1}$ is a final state.
- We conventionally denote a final state by $\sigma_\varepsilon$. 
LTS and Traces

- We assume a *labeled transition system* (LTS), \((\mathcal{S}, \mathcal{E}, \rightarrow)\), where \(\mathcal{S}\) is a set of states, \(\mathcal{E}\) a set of events, and \(\rightarrow \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}\) the transition relation.

- We allow *unlabeled transitions*.

- An *execution* is a sequence of transitions:

\[
\sigma_1 \xrightarrow{e_1} \sigma_2 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \sigma_{n+1}
\]

such that all events are distinct and such that \(\sigma_{n+1}\) is a final state.

- We conventionally denote a final state by \(\sigma_\varepsilon\).

- An *execution trace* is a sequence of events, denoted \(\tau = e_1 \cdot e_2 \cdot \ldots \cdot e_n\).
Record & Replay semantics

\[
\begin{align*}
&\text{(Unlabeled Record & Replay)} \\
&\sigma \rightarrow \sigma' \\
&\langle \tau \bullet | \tau \triangleright \rangle \triangleright \sigma \xrightarrow{\bullet/\triangleright} \langle \tau \bullet | \tau \triangleright \rangle \triangleright \sigma'
\end{align*}
\]
Record & Replay semantics

(Unlabeled Record & Replay)

\[
\sigma \rightarrow \sigma'
\]

\[
\langle \tau_0 \mid \tau \rangle \triangleright \sigma \xrightarrow{\bullet/\triangleright} \langle \tau_0 \mid \tau \rangle \triangleright \sigma'
\]

(Labeled Record)

\[
\sigma \xrightarrow{e} \sigma'
\]

\[
\langle \tau_0 \mid e \rangle \triangleright \sigma \xrightarrow{\bullet/\triangleright} \langle \tau_0 \cdot e \mid e \rangle \triangleright \sigma'
\]
Record & Replay semantics

(Unlabeled Record & Replay)

$$\sigma \rightarrow \sigma'$$

$$\langle \tau_\bullet \mid \tau_\triangledown \rangle \triangleright \sigma \quad \xrightarrow{\bullet/\triangledown} \quad \langle \tau_\bullet \mid \tau_\triangledown \rangle \triangleright \sigma'$$

(Labeled Record)

$$\sigma \xrightarrow{e} \sigma'$$

$$\langle \tau_\bullet \mid \varepsilon \rangle \triangleright \sigma \quad \xrightarrow{\bullet/\triangledown} \quad \langle \tau_\bullet \cdot e \mid \varepsilon \rangle \triangleright \sigma'$$

(Labeled Record & Replay)

$$\sigma \xrightarrow{e} \sigma'$$

$$\langle \tau_\bullet \mid e \cdot \tau_\triangledown \rangle \triangleright \sigma \quad \xrightarrow{\bullet/\triangledown} \quad \langle \tau_\bullet \cdot e \mid \tau_\triangledown \rangle \triangleright \sigma'$$
We have the following relations over $\mathcal{E}$:

- $e_i \text{MHB} \Rightarrow e_j$ if the event $e_i$ must happen before $e_j$ in all feasible executions;
- $e_i \text{⋆} e_j$ if the order of $e_i$ and $e_j$ may affect the result of an execution;
- $e_i \text{HB} \Rightarrow e_j$ if $e_i$ occurs before $e_j$ in the trace $\tau$ and $e_i \text{MHB} \Rightarrow e_j$ or $e_i \text{⋆} e_j$.

We can derive $\text{HB} \Rightarrow \tau$ from $\text{MHB} \Rightarrow \tau$ and ⋆ for a given $\tau$.

Two traces $\tau_1$ and $\tau_2$ are equivalent, denoted $\tau_1 \simeq \tau_2$, if $\text{HB} \Rightarrow \tau_1 = \text{HB} \Rightarrow \tau_2$. 
We have the following relations over $\mathcal{E}$:

- $e_i \xrightarrow{\text{MHB}} e_j$ if the event $e_i$ must happen before $e_j$ in all feasible executions;
- $e_i \star e_j$ if the order of $e_i$ and $e_j$ may affect the result of an execution;
- $e_i \xrightarrow{\text{HB}} \tau e_j$ if $e_i$ occurs before $e_j$ in the trace $\tau$ and $e_i \xrightarrow{\text{MHB}} e_j \text{ or } e_i \star e_j$.

We can derive $\text{HB} \xrightarrow{\tau}$ from $\text{MHB} \xrightarrow{\tau}$ and $\star$ for a given $\tau$.

Two traces $\tau_1$ and $\tau_2$ are equivalent, denoted $\tau_1 \equiv \tau_2$, if $\text{HB} \xrightarrow{\tau_1} \text{HB} \xrightarrow{\tau_2}$.
We have the following relations over $\mathcal{E}$:

- $e_i \xrightarrow{\text{MHB}} e_j$ if the event $e_i$ must happen before $e_j$ in all feasible executions;
- $e_i \star e_j$ if the order of $e_i$ and $e_j$ may affect the result of an execution;
Relations over events

- We have the following relations over $E$:
  - $e_i \xrightarrow{\text{MHB}} e_j$ if the event $e_i$ must happen before $e_j$ in all feasible executions;
  - $e_i \star e_j$ if the order of $e_i$ and $e_j$ may affect the result of an execution;
  - $e_i \xrightarrow{\text{HB}}_{\tau} e_j$ if $e_i$ occurs before $e_j$ in the trace $\tau$ and $e_i \xrightarrow{\text{MHB}} e_j$ or $e_i \star e_j$.

We can derive $\xrightarrow{\text{HB}}_{\tau}$ from $\xrightarrow{\text{MHB}}$ and $\star$ for a given $\tau$. 

Two traces $\tau_1$ and $\tau_2$ are equivalent, denoted $\tau_1 \simeq \tau_2$, if $\xrightarrow{\text{HB}}_{\tau_1} = \xrightarrow{\text{HB}}_{\tau_2}$. 

Page 7
Relations over events

- We have the following relations over $\mathcal{E}$:
  
  $e_i \xrightarrow{\text{MHB}} e_j$ if the event $e_i$ must happen before $e_j$ in all feasible executions;
  
  $e_i \star e_j$ if the order of $e_i$ and $e_j$ may affect the result of an execution;
  
  $e_i \xrightarrow{\text{HB}}_{\tau} e_j$ if $e_i$ occurs before $e_j$ in the trace $\tau$ and $e_i \xrightarrow{\text{MHB}} e_j$ or $e_i \star e_j$.

We can derive $\xrightarrow{\text{HB}}_{\tau}$ from $\xrightarrow{\text{MHB}}$ and $\star$ for a given $\tau$. 

Two traces $\tau_1$ and $\tau_2$ are equivalent, denoted $\tau_1 \equiv \tau_2$, if $\xrightarrow{\text{HB}}_{\tau_1} = \xrightarrow{\text{HB}}_{\tau_2}$.
We have the following relations over $\mathcal{E}$:

- $e_i \xrightarrow{\text{MHB}} e_j$ if the event $e_i$ must happen before $e_j$ in all feasible executions;
- $e_i \xrightarrow{\text{HB}} e_j$ if the event $e_i$ occurs before $e_j$ in the trace $\tau$ and $e_i \xrightarrow{\text{MHB}} e_j$ or $e_i \xrightarrow{\text{HB}} e_j$.
- $e_i \xrightarrow{\star} e_j$ if the order of $e_i$ and $e_j$ may affect the result of an execution;

We can derive $\xrightarrow{\text{HB}}$, $\xrightarrow{\text{MHB}}$, and $\xrightarrow{\star}$ for a given $\tau$.

Two traces $\tau_1$ and $\tau_2$ are equivalent, denoted $\tau_1 \simeq \tau_2$, if $\xrightarrow{\text{HB}}_{\tau_1} = \xrightarrow{\text{HB}}_{\tau_2}$.
Exogenous exploration

\[(\text{EXPLORE-SINGLE-TRACE})\]

\[\langle \text{exo} : \{\tau\} \cup \text{Seeds}, \{w\} \cup W, Ss \rangle \rightarrow \langle \text{exo} : \text{Seeds}, W, Ss \rangle \langle w : \langle \varepsilon \mid \tau \rangle \triangleright \sigma_0 \rangle\]
Exogenous exploration

((EXPLORE-SINGLE-TRACE))
\[ \langle \text{exo} : \{ \tau \} \cup \text{Seeds}, \{ w \} \cup W, S_s \rangle \rightarrow \langle \text{exo} : \text{Seeds}, W, S_s \rangle \langle w : \langle \varepsilon | \tau \rangle \triangleright \sigma_0 \rangle \]

((WORKER-PROGRESS))
\[
\begin{align*}
\langle \tau \bullet | \tau' \rangle \triangleright \sigma & \rightarrow \langle \tau \bullet | \tau' \rangle \triangleright \sigma' \\
\langle w : \langle \tau \bullet | \tau' \rangle \triangleright \sigma \rangle & \rightarrow \langle w : \langle \tau' \bullet | \tau' \rangle \triangleright \sigma' \rangle
\end{align*}
\]
Exogenous exploration

(Explore-single-trace)
\[
\langle \text{exo} : \{\tau\} \cup \text{Seeds}, \{w\} \cup W, Ss \rangle \rightarrow \langle \text{exo} : \text{Seeds}, W, Ss \rangle \langle w : \langle \varepsilon | \tau \rangle \triangleright \sigma_0 \rangle
\]

(Worker-progress)
\[
\frac{\langle \tau_\bullet | \tau_\triangleright \rangle \triangleright \sigma}{\langle w : \langle \tau_\bullet | \tau_\triangleright \rangle \triangleright \sigma \rangle \rightarrow \langle w : \langle \tau'_\bullet | \tau'_\triangleright \rangle \triangleright \sigma' \rangle}
\]

(Update-with-explored-trace)
\[
Ss' = \text{addTrace}(Ss, \tau) \quad \text{Seeds'} = \text{Seeds} \cup \text{newSeeds}(Ss', \tau)
\]
\[
\langle \text{exo} : \text{Seeds}, W, Ss \rangle \langle w : \langle \tau | \varepsilon \rangle \triangleright \sigma_{\varepsilon} \rangle \rightarrow \langle \text{exo} : \text{Seeds'}, \{w\} \cup W, Ss' \rangle
\]
Framework

Runtime

Translator
Framework

$t = t_s \cdot t'$

Runtime \[→\] $t_s$ \[→\] Translator
Framework

- Runtime
- Translator
- Communicator
- DPOR

\[
\tau_s = \tau_s \cdot \tau_t' \cdot \tau
\]
Framework

ExoDPOR

Runtime
Translator
Communicator
DPOR
Framework

ExoDPOR

\[ t = t_s \cdot t' \]

Runtime \[ t_s \] Translator \[ \tau_s \] Communicator \[ \text{DPOR} \]
$t = t_s \cdot t'$

$\tau_s$

ExoDPOR

Runtime

Translator

Communicator

DPOR
$t = t_s \cdot t'$

Framework

ExoDPOR

Runtime

Translator

Communicator

DPOR

$t = t_s \cdot t'$
Current state

- We have a sound implementation of Source-DPOR
Current state

- We have a sound implementation of Source-DPOR
  - Major disadvantage with not being able to dynamically avoid redundant executions

```
<table>
<thead>
<tr>
<th>workers</th>
<th>time-ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td>21</td>
</tr>
<tr>
<td>16</td>
<td>22</td>
</tr>
<tr>
<td>18</td>
<td>23</td>
</tr>
<tr>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

program
- sleep 1000ms
- sleep 100ms
- sleep 10ms
- sleep 0ms
```
Current state

- We have a sound implementation of Source-DPOR
  - Major disadvantage with not being able to dynamically avoid redundant executions
- We are working on an implementation of Optimal-DPOR (still containing bugs)

```
program
sleep 1000ms
sleep 100ms
sleep 10ms
sleep 0ms
```

<table>
<thead>
<tr>
<th>workers</th>
<th>time-ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
</tbody>
</table>

10
Current state

- We have a sound implementation of Source-DPOR
  - Major disadvantage with not being able to dynamically avoid redundant executions
- We are working on an implementation of Optimal-DPOR (still containing bugs)
  - Every seed trace is guaranteed to reach a non-equivalent execution
Current state

- We have a sound implementation of Source-DPOR
  - Major disadvantage with not being able to dynamically avoid redundant executions
- We are working on an implementation of Optimal-DPOR (still containing bugs)
  - Every seed trace is guaranteed to reach a non-equivalent execution
- We have instantiations for two toy languages and for Real Time ABS
Current state

- We have a sound implementation of Source-DPOR
  - Major disadvantage with not being able to dynamically avoid redundant executions
- We are working on an implementation of Optimal-DPOR (still containing bugs)
  - Every seed trace is guaranteed to reach a non-equivalent execution
- We have instantiations for two toy languages and for Real Time ABS
- Parallelization is simple and gives a linear speedup for long-running programs
We have introduced ExoDPOR, a general framework for stateless model checking with state of the art DPOR-algorithms.
Conclusion

- We have introduced ExoDPOR, a general framework for stateless model checking with state of the art DPOR-algorithms
- It has been instantiated for Real-Time ABS
Conclusion

- We have introduced **ExoDPOR**, a general framework for stateless model checking with state of the art DPOR-algorithms
- It has been instantiated for Real-Time ABS
- We believe an instantiation for **ExoDPOR** is significantly easier than to implement a model checker from scratch
Conclusion

- We have introduced ExoDPOR, a general framework for stateless model checking with state of the art DPOR-algorithms
- It has been instantiated for Real-Time ABS
- We believe an instantiation for ExoDPOR is significantly easier than to implement a model checker from scratch
- Experiments indicate that parallelization gives a linear speed-up for long-running programs