Verification of Sensor Network Programs

Trinh Cong Quy and Othmane Rezine,
(Parosh Aziz Abdulla and Bengt Jonsson)
Outline

1. Verification of Highly Concurrent Data Structure

2. Decidability & Complexity of Parametrized Verification of SNs

3. MPass: Message Passing Verification tool.

4. Planned cooperation: Use Dynamic POR to verify NesC programs
Verification of Highly Concurrent Data Structure
Highly Concurrent Data Structures

```c
struct cell {
    data array data;
    cell *next1, ..., nextn
}
```

Singly linked list
Highly Concurrent Data Structures

```c
struct cell {
    data array data;
    cell *next1, ..., nextn
};
```

Double linked list

Data pointers

G: Head, Tail

l: pred, curr
Highly Concurrent Data Structures

```
struct cell {
    data array data;
    cell *next1,..,nextn
}
```
Highly Concurrent Data Structures

struct cell {
    data array data;
    cell *next1,..,nextn
}

Skip-list

Data

pointers

1:pred, curr

G:Head, Tail

complex
Goals

Verification of the implementation of:

- Concurrent Sets, using
  - Skip-list
  - Singly-linked list

- Concurrent queues, using
  - Doubly-linked list
Goals

Verification of the implementation of:

- Concurrent Sets, using
  - Skip-list
  - Singly-linked list

- Concurrent queues, using
  - Doubly-linked list

with garbage collection
Goals & Properties

- Safety properties
  - No null pointer dereferencing
  - No cycle creation
- Linearization
- Data ordering properties
  - Sortedness
- Shape properties
  - Skip-list shape properties
  - Doubly-linked list properties
Summary

Challenge

First work addressing the automatic verification of:
- Concurrent skip-list
- Concurrent linked list

Status

Prototype addressing the concurrent sets based on singly-linked lists
Decidability & Complexity of Parametrized Verification of SNs
Decidability & Complexity of Parametrized Verification of SNs

- Parametrized: Same process, copied in an (unbounded) number of nodes
- Goal: Check State Reachability in a protocol, regardless of the size of the network
- Undecidable in general: Challenge is to define decidable sub-classes.
Results (1/2)

- Process: Finite State Communicating Machine
- Directed Acyclic Static Topology
- Decidable if we bound the depth of the network
Results (2/2)

- Process: Finite State Communicating Machine + Registers
- Dynamic Topology
- Decidable if:
  - We bound the underlying undirected graph
  - Allow connection loss
MPass: Message Passing Verification tool
MPass: Model

- Finite Number of Finite State Machines
- Unbounded channels
- Channel Semantics:
  - Perfect FIFO
  - Lossy (Stuttering) FIFO
  - Unordered
MPass: Under-Approximation

- Phase-Bounded
- Phase: Send-only OR Receive-Only
- Covers: Unbounded runs / Unbounded channels / Unbounded number of context switches
Results (1/2): Finite state Processes

<table>
<thead>
<tr>
<th>Finite state Processes</th>
<th>Unbounded</th>
<th>Bounded-Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect FIFO Channels</td>
<td>Undecidable</td>
<td>Undecidable</td>
</tr>
<tr>
<td>Lossy FIFO Channels</td>
<td>Non-primitive Recursive</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>Unordered Channels</td>
<td>EXPSPACE-hard</td>
<td>NP-Complete</td>
</tr>
</tbody>
</table>
# Results (2/2): PushDown Processes

<table>
<thead>
<tr>
<th>PushDown Processes</th>
<th>Unbounded</th>
<th>Bounded-Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect FIFO Channels</td>
<td>Undecidable</td>
<td>Undecidable</td>
</tr>
<tr>
<td>Lossy FIFO Channels</td>
<td>Undecidable</td>
<td>Undecidable</td>
</tr>
<tr>
<td>Unordered Channels</td>
<td>EXPSPACE-hard</td>
<td>NP-Complete</td>
</tr>
</tbody>
</table>
MPass: Idea

- Translate Protocol State Reachability to Automaton State Reachability
- Use of SMT-solvers as a Backend.
- Suitable to find bugs
MPass: Future work

- Add Shared Memory to the model
- Apply the bounded phase approach to message passing programming languages
Use Dynamic POR to verify NesC programs (TinyOS)
POR: Partial Order Reduction

- Concurrent Processes executing in parallel
- Some transitions can be swapped in order to get to the same state
- Exploit transitions indépendance
- Reduces the search space
TinyOS

- Lightweight OS
- Runs small programs on sensors
- Contains device-related libraries
- Supports event-driven concurrent model
NesC

- Dialect of C

- Component-based programming model:
  - Uses other component’s interface:
    Uses its commands and implements its events.
  - Provides an interface
  - Sync, but also Async: Tasks (FIFO)
Recent Approach: NesC@PAT

- Developed in PAT (a model-checking tool)
- Two-level Partial Order Reduction:
  - Local Independence (Inside Sensors)
  - Global Independence (Between Sensors)
- Works directly on NesC programs
- Properties: Deadlock-freedom, State Reachability, Temporal Properties
Example

```c
result_t tryNextSend()
{
    ...  
    atomic{
        if(!sendTaskBusy){
            post sendTask();
            sendTaskBusy = TRUE;
        }
    }
    ...
}
...
```

```c
task void sendTask()
{
    ...
    sendTaskBusy = FALSE;
    ...
}
```
Example

result_t  tryNextSend()
{
    ...
    atomic{
        if(!sendTaskBusy)
        {
            post sendTask();
            sendTaskBusy = TRUE;
        }
    }
    ...
}

* Taken from NesC@PAT presentation

task void sendTask()
{
    ...
    sendTaskBusy = FALSE;
    ...
}

Example

```c
result_t tryNextSend() {
    ...
    atomic{
        if (!sendTaskBusy) {
            post sendTask();
            sendTaskBusy = TRUE;
        }
    }
    ...
}
```

```c
task void sendTask() {
    ...
    sendTaskBusy = FALSE;
    ...
}
```

Possible Error
Example

Result_t tryNextSend()
{
    ...    
    atomic{
        if (!sendTaskBusy)
        {
            post sendTask();
            sendTaskBusy = TRUE;
        }
    }
    ...    
}

Task void sendTask()
{
    ...  
    sendTaskBusy = FALSE;
    ...  
}
Example

result_t tryNextSend()
{
    ...
    atomic{
        if(!sendTaskBusy){
            if(SUCCESS != post sendTask())
                sendTaskBusy = FALSE;
            else sendTaskBusy = TRUE;
        }
    }
    ...
}

task void sendTask()
{
    ...
    sendTaskBusy = FALSE;
    ...
}
Example

```c
result_t tryNextSend() {
    ...
    atomic{
        if (!sendTaskBusy) {
            if (SUCCESS != post sendTask())
                sendTaskBusy = FALSE;
            else sendTaskBusy = TRUE;
        }
    }
    ...
}
```

Possible Error

If `post sendTask()` fails, the task will never be executed, and thus `sendTaskBusy` remains TRUE forever.

If `sendTaskBusy == <> !sendTaskBusy`
Thank you for listening!