dSL,
a Language and Environment
for the Design and validation of Distributed Industrial Controllers

B. De Wachter   Gabriel Kalyon   T. Massart   C. Meuter

Département d'Informatique
Université Libre de Bruxelles

20 April 2007
Outline

1. Motivations & other approaches
2. dSL
3. Distributing a dSL program
4. Implementation of dSL
5. Validation of dSL
6. Testing and Monitoring dSL programs
7. Future work
Industrial control systems

- are used to control processes like traffic lights, assembly belts, power plant regulation, ...
- Observe using sensors (pressure, speed, presence, ...)
- Decide based on a specification given by the designer
- React through actuators (physical devices allowing to force currents and voltages)
Difficulties for the programmer are:

- their distributed nature
- the need to establish functional correctness
- the need for efficiency
Goal

Offer a language with transparent distribution for the design of industrial systems
- Programmer concentrates on functional aspects
- Increases maintainability, flexibility and simplicity

Give precise semantics
- Establishes unambiguous description of the behavior
- Allows formal verification
Constraints

- Start from an existing industrial language
- Low resources on target platform
- Simplicity, monitorability and robustness
- Efficiency

Development approach taken

```plaintext
CLASS Heater
    control : INT;
    maintenance, state : BOOL;
END_CLASS

GLOBAL_VAR
    heater : Heater;
    temperature, fuel_cost : INT;
    alarm, led : BOOL;
END_VAR

SEQUENCE set_heater(new_state:BOOL)
    heater.control := heater.control + 1;
    heater.state := new_state;
    IF (heater.state == 1) THEN
        led := TRUE;
        fuel_cost := fuel_cost + 10;
    ELSE
        led := FALSE;
    END_IF
ENDSEQUENCE
```
Other approaches of design with transparent distribution

Process algebra: Correctness preserving transformation [Mas92, BL95]

- Centralized specification $B$ with actions $\Sigma$
- Distribute actions $\Sigma = \Sigma_1 \cup \Sigma_2$ ($\Sigma_1 \cap \Sigma_2 = \emptyset$)
- Find two specifications
  - $B_1$ with actions $\Sigma_1$
  - $B_2$ with actions $\Sigma_2$ such that

$$B_1 \otimes B_2 \simeq B$$

Limitations
No assignment, variables are difficult to model

Unity

- Design language and proof system for the specification of parallel programs [CM88]
- Separation of the program and physical architecture

Limitations
- The mapping is dependent on a underlying protocol
- Limits flexibility
Other approaches of design with transparent distribution

Synchronous languages

- Used for the design of reactive systems [BG92, CPHP87, LGLL91]
- Based on instants which take no time
  - Each instant computes a reaction to the environment
  - The environment is fixed during an instant
- Results in deterministic and sequential code

Limitations

- Hard to distribute
  - Instants must be preserved [Gir94]
  - Strong synchronization is needed between different sites
  - Robustness issues
  - Efficiency issues
1 Motivations & other approaches
2 dSL
3 Distributing a dSL program
4 Implementation of dSL
5 Validation of dSL
6 Testing and Monitoring dSL programs
7 Future work
**dSL in a nutshell**

**dSL**

*dSL, distributed Supervision Language [DMM03]*

**Features**

- Based upon an existing *industrial domain specific language*
- Providing **transparent distribution**
  - Allowing the designer to concentrate on the functional aspects, disregarding the physical distribution
  - Obtained by *a single program text* and a separate *localization table*
- With **formal semantics**
  - Allows to formally model the behavior of the system
  - Allows for automatic verification using model checking

**Basic Hypothesis**

- Asynchronous composition
- Static distribution
- Clear *application target*
SL

- OO Language
- Syntax inspired on Pascal
- Used on the supervisor
- Asynchronous communications with PLCs (Programmable Logic Controllers)

**Events specified by WHEN**
- Condition with raising edge
- Body specifies action to take
- Example:
  ```
  WHEN button THEN
      lamp := TRUE;
  END_WHEN
  ```

**Variables may be:**
- internal (global/local)
- external (global and input/output)
From SL to dSL

### dSL
- dSL as a **single language** for Supervisor and PLCs, called **sites**
- Each site **asynchronously communicates** with the others
- Composition of **local atomic instantaneous code** to handle events and **distributable code** for distributed actions

Localization table
Atomic code

Static distribution based on the atomicity of \textbf{WHENs}

- A \textbf{WHEN} must be able to execute without the need to synchronize with other sites
- Causes \textbf{non-distributable programs}

```
WHEN button THEN lamp := TRUE
```

where \texttt{button} and \texttt{lamp} are specified on different sites
Two syntactical additions to break atomic constraints

- The ~ operator
  
  ```
  WHEN ~button THEN lamp := TRUE
  ```

- The LAUNCH keyword
  
  ```
  WHEN button THEN LAUNCH set_lamp(TRUE); END_WHEN
  ```
Example

SEQUENCE

SEQUENCE start_pumps()
    pump1.engine := TRUE;
    WAIT pump1.pressure > 30;
    pump2.engine := TRUE;
...
END_SEQUENCE

Sequences

- Allows to perform a sequence of distributed actions (i.e. that may happen on different sites)
- Sequential code is not instantaneous (but each instruction is atomic)
- The implementation uses static thread migration
  - Predictable performance
  - Execution moves, variables do not move
  - Easy to implement
  - Needs low resources
CLASS Heater
    control :            INT;
    maintenance, state : BOOL;
END_CLASS

GLOBAL_VAR
    heater                 : Heater;
    temperature, fuel_cost : INT;
    alarm, led             : BOOL;
END_VAR

SEQUENCE set_heater(new_state:BOOL)
    heater.control :=
        heater.control+1;
    heater.state := new_state;
    IF (heater.state == 1) THEN
        led       := TRUE;
        fuel_cost := fuel_cost + 10;
    ELSE
        led := FALSE;
    END_IF
END_SEQUENCE

WHEN IN Heater (control==1000) THEN
    control := 0;
    maintenance := TRUE;
END_WHEN

WHEN heater.maintenance THEN
    alarm := TRUE;
END_WHEN

WHEN ~temperature < 0 THEN
    IF (NOT heater.maintenance) THEN
        LAUNCH set_heater(1);
    END_IF
END_WHEN

WHEN ~temperature > 20 THEN
    IF (NOT heater.maintenance) THEN
        LAUNCH set_heater(0);
    END_IF
END_WHEN

PROGRAM
    heater.control := 0;
    heater.maintenance := FALSE;
    LAUNCH set_heater(temperature<0);
END_PROGRAM
dSL’s semantics

Features

- Defined on a subset of dSL, noted dSL
  - No recursion, no METHODS
  - No local variables outside SEQUENCE
  - No LAUNCH and WAIT

- Parametrized by a distribution $D$, the set of behaviors of a program $P ([P]_D)$ is given as an LTS defined by a set of SOS rules [DGMM05]

The distribution defines

- the set of processes (one for each site)
- The processes communicate using FIFO queues
- Each process executes an infinite loop with
  1. Input sampling
  2. Processing
     - Event triggering
     - Message treatment, caused by $\sim$
  3. Output writing
dSL’s semantics - Distribution

### Definition (Distribution of a well-formed dSL program $P$)

Partition $D = \{V_1, V_2, \ldots, V_n\}$ of $\text{Var}(P)$ such that:

$\forall w \in \text{Whens}(P) \exists V \in D, \text{Var}(w) \subseteq V$

$\land \forall i \in \text{InstrSeq}(P) \exists V \in D, \text{Var}(i) \subseteq V$

Variables appearing in the same `WHEN` (or in the same `SEQUENCE` instruction) must be distributed on the same site.

### Definition (Distribution hierarchy)

Let $D = \{V_1, V_2, \ldots, V_k\}$ and $D' = \{V'_1, V'_2, \ldots, V'_l\}$ be two distributions of a dSL program $P$. We say that distribution $D'$ is a refinement of distribution $D$, noted $D \preceq D'$, if:

$\forall V' \in D' \exists V \in D \cdot V' \subseteq V$

### Theorem (Distribution Lattice)

$D$ and $\preceq$ define a lattice of distributions.

$\Rightarrow \exists D_{\text{max}}, D_{\text{min}}$
Definition (Global state of a dSL\(\blacklozenge\) program)

\[
G \equiv ((\omega_1, \nu_1, \phi_1), (\omega_2, \nu_2, \phi_2), \ldots, (\omega_n, \nu_n, \phi_n), \sigma_1, \sigma_2, \ldots, \sigma_\ell, \mu, \xi)
\]

where \((\omega_i, \nu_i, \phi_i)\) is the local state of process \(i\) with the following components:

- \(\omega_i\) is the workload
- \(\nu_i: (V_i \cup \text{Var}(P) \cup \text{OldCond}(W_i)) \mapsto \{\top, \bot, \#\}\) is a valuation function for the global variables of process \(i\)
- Let \(\Sigma_{\phi} = ((\text{Var}(P) \times \{\top, \bot, \#\} \times \mathbb{N}) \cup \{\diamond | i \in [1..n]\})^*\), then \(\phi_i \in \Sigma_{\phi}^*\) is the receiving communication channel of process \(i\)
- \(\sigma_i\) is the workload for the \text{SEQUENCE} identified by \(i\)
- \(\mu: \text{VarSeq} \mapsto \{\top, \bot, \#\}\) is the valuation function for the local variables of all \text{SEQUENCES}
- \(\xi: [1..\ell] \mapsto [1..n]\) indicates on which site a given sequence is running
**dSL’s semantics - SOS Rules**

**Rule : cycle start**

- Cycle start is a **local rule**
- Defines the cyclic Input-Process-Output behavior of a process

**[Cycle start]**

\[(E^P_D)_i \vdash (\varepsilon, \nu_i, \phi_i) \rightarrow^T (\text{Sample}(\text{Var}^\text{in}(P) \cap V_i, < \nu); \text{Treat}(W_i, < w); \text{MSG}; \text{Write}(\text{Var}^\text{out}(P) \cap V_i, < \nu), \nu_i, \phi_i')\]

\[\phi_i' \in \text{Shuffle}(\phi_i)_\pi \text{ for some } \pi\]

- If a process has nothing left to do
  - Block incoming messages
  - Sample inputs
  - Treat WHENs
  - Treat messages
  - Write outputs
Theorem (Simulation)

Given a well-formed \( d\text{SL}\) program \( P = (V, \prec_V, W, \prec_W) \), let \( D = \{V_1, V_2, \ldots, V_n\} \) and \( D' = \{V'_1, V'_2, \ldots, V'_l\} \) be two distributions of \( P \). We have that, if \( D' \) refines \( D \) then \( \llbracket P \rrbracket_D' \) simulates \( \llbracket P \rrbracket_D \):

\[
(D \preceq D') \Rightarrow \left( \llbracket P \rrbracket_D \preceq \llbracket P \rrbracket_{D'} \right)
\]

Every step that \( P \) can perform with distribution \( D \) can be simulated by \( P \) using a more refined distribution \( D' \).
Corollary (Property preservation)

Given a $\mathcal{dSL}_\Diamond$ program $P$, and two distributions $D, D'$ such that $D \preceq D'$ we have for all next-free LTL formula $\phi$:

- $[P]_{D'} \models \phi \Rightarrow [P]_D \models \phi$
- $[P]_{D\neg} \models \phi \Rightarrow [P]_{D'}\neg \models \phi$
Outline

1. Motivations & other approaches
2. dSL
3. Distributing a dSL program
4. Implementation of dSL
5. Validation of dSL
6. Testing and Monitoring dSL programs
7. Future work
Localizing instructions and variables of a dSL program is a two step process:

1. **Localizing atomic code** *(WHENS)*
   - Decide if a program is distributable
   - Assign all atomic code to the set of sites
     - Can be done in $O(n^2)$
     - Results in a complete assignment of all atomic instructions to the set of sites

2. **Localizing sequential code** *(SEQUENCES)*
   - Assign remaining sequential instructions to the set of sites
   - Is hard, equivalent to the Multiterminal Cut Problem

3. **Optimize performance by reordering instructions**
   - Is hard, equivalent to the Precedence Constrained Class Sequencing Problem
   - Not seen here.
Definition (Localization table)

A localization table $T$ for a qSL program $P$ is a total assignment from the set of external variables to the set of sites: $T \in Var^{in}(P) \cup Var^{out}(P) \rightarrow S$

Definition (Synchronous control flow)

An instruction $i'$ is reachable through synchronous control flow from $i$, noted $i \sim_a i'$ if one of the following conditions holds:

- $i$ and $i'$ are consecutive in the same WHEN
- $i'$ is the condition of a WHEN that might be immediately triggered by the execution of $i$
- $i'$ is the first instruction in a method called by $i$ without LAUNCH
Definition (Atomic Color Graph)

The atomic color graph $G_a(V, E, T)$ with

- $V = \{i | i$ is an instruction in $P\} \cup Var(P)$
- $E \subseteq \{\{v, v'\} | v, v' \in V\}$ such that:
  - $\forall x \in Var(P), \forall i \in instr(P) : x \in used(i) \Rightarrow \{x, i\} \in E$
  - $\forall i, i' \in instr(P) : i \rightsquigarrow_a i' \Rightarrow \{i, i'\} \in E$
Definition (Atomic Coloring Problem (ACP))

The atomic coloring problem consists in, given an atomic color graph $G_a$, finding a mapping $c : V \mapsto S$ compatible with $T$ such that for all connected components $C$ in $G_a$:

$$n, n' \in C \Rightarrow c(n) = c(n')$$
The ACP is easy to solve \( (O(N^2)) \)

- Building \( G_a \) can be done in \( O(N) \)
- Finding the mapping \( c \) is as easy as walking over the graph and can be done in \( O(N^2) \)

Results in a total assignment of atomic code to the set of sites
The Sequential Coloring Problem (SCP) consists in:

- **Given**: a **sequence**, with some instructions localized
  - an estimate of the number of times control flows between instructions

- **When control flows between instructions of different color**, synchronization is needed

- **Find**: a mapping from the instructions to the sites, such that the expected number of synchronizations is minimum
The Sequential Coloring Problem (SCP) consists in:

- **Given**: a sequence, with some instructions localized
  - an estimate of the number of times control flows between instructions

- When control flows between instructions of different color, synchronization is needed

- **Find**: a mapping from the instructions to the sites, such that the expected number of synchronizations is minimum
Given a weighted undirected graph $G(V, E, w): E \subseteq \{\{u, v\}|u, v \in V \land u \neq v\}, w: E \rightarrow \mathbb{N}_0 \cup \{\infty\}$ and a set of terminals $T = \{s_1, ..., s_k\} \subseteq V$, find a partition of $V$ into $V_1, ..., V_k$ such that $s_i \in V_i \ \forall \ i \in [1, k]$ and $\sum_{v \in V_i, v' \in V_j, i \neq j} w(v, v')$ is minimized.
Definition (Multiterminal Cut Problem (MCP) [DJP+94])

Given a weighted undirected graph \( G(V, E, w) \) where \( E \subseteq \{(u, v) \mid u, v \in V \land u \neq v \} \), \( w : E \rightarrow \mathbb{N}_0 \cup \{\infty\} \) and a set of terminals \( T = \{s_1, \ldots, s_k\} \subseteq V \), find a partition of \( V \) into \( V_1, \ldots, V_k \) such that \( s_i \in V_i \ \forall \ i \in [1, k] \) and \( \sum_{v \in V_i, v' \in V_j, i \neq j} w(v, v') \) is minimized.
The SCP is NP-hard, even when the program contains only a sequence of instructions.

[DGGM05] The SCP is polynomially equivalent to the multiterminal cut problem on arbitrary graphs.
New results on the Multiterminal Cut

Introduction of a new shrinkage theorem, which generalizes an existing one

Theorem (More shrinkage)

- Given a weighted graph $G(V, E, w)$ with terminals $T = \{s_1, ..., s_k\} \subseteq V$.
- Let $v \in V$ (be it in $T$ or not), and
- $G'_v$ be the graph where all terminals in $T \setminus \{v\}$ are merged into $t$, and
- $(C, \overline{C})$ a minimum $st$ cut between $v$ and $t$ in $G'_v$

then there exists an optimal multiterminal cut $(V_1, ..., V_k)$ of $G$ such that $\exists \ell : C \subseteq V_\ell$. 
(3) Reordering instructions to increase performance

Example

SEQUENCE plant_startup()
...
  engine_speed1 := 10;
  engine_speed2 := 20;
  pump1.engine := engine_speed1;
pump2.engine := engine_speed2;
...
END_SEQUENCE

SEQUENCE plant_startup()
...
  engine_speed1 := 10;
pump1.engine := engine_speed1;
  engine_speed2 := 20;
pump2.engine := engine_speed2;
...
END_SEQUENCE

- Reordering instructions can lead to more efficient programs (i.e. less synchronization points)
- There are two constraints when moving instructions
  - Def-use chains
  - Observable assignments to external variables
- To find the optimal solution
  1. Extract the partial order from the program text
  2. Find a total order with a minimum number of switchings
Reordering instructions - Example

Input:

Output:
Reordering instructions

- NP-Complete problem (Precedence Constrained class Sequencing problem (PCCS))
Outline

1. Motivations & other approaches
2. dSL
3. Distributing a dSL program
4. Implementation of dSL
5. Validation of dSL
6. Testing and Monitoring dSL programs
7. Future work
Overview of the dSL environment

- **The Frontend**
  - Parses the dSL source code
  - Produces a set of Control Flow Graphs

- **The optimizer**
  - Transforms complex instructions into 3-address code
  - Specializes METHODS

- **The distributor**
  - Performs the two-step coloring process defined earlier
  - Inserts synchronization code in SEQUENCES for thread migration

- **The backend** produces CISC code for the dSL virtual machines
Synchronization code for SEQUENCES

\[
\begin{align*}
x & := Ga; \\
y & := Ga; \\
w & := y + Gb; \\
y & := Gb + x; \\
u & := x + y + Gc; \\
v & := u + x + Ga;
\end{align*}
\]
Integration with Macq Electronique’s OBViews
1. Motivations & other approaches
2. dSL
3. Distributing a dSL program
4. Implementation of dSL
5. Validation of dSL
6. Testing and Monitoring dSL programs
7. Future work
The verification of a restricted dSL is performed with Spin. Spin is an Explicit state on-the-fly model checker with partial order reduction [Hol03]. It checks LTL properties [Eme90]. PROMELA is an input language for models of communicating finite state machines [BZ83].

A special purpose backend for the dSL compiler-distributor translates automatically to PROMELA except for:

- the environment which must be specified in PROMELA
- the property which is given in LTL

Some limitations include:

- No LAUNCH and WAIT
- No recursion (METHODS and WHENS)

Some limitations include:

- No LAUNCH and WAIT
- No recursion (METHODS and WHENS)
No two consecutive gates are opened and when opened, the water level is equal at both sides

1 to 11 site distribution, > 60 boolean variables

verified in less than 30 minutes, and 270MB on a 3Ghz processor

Showed an error in a early design
Verification Results - Conveyor belt system

- When a box is at the end of belt, the belt should be running only if the next belt is in front of it and running
- 6 site distribution, > 20 boolean variables
- verified in less than 6 minutes and 2.95GB on a 3Ghz processor
- Needed to perform breath-first search
Both trains can not be on the central track at the same time
4 site distribution, > 10 boolean variables
verified in less than 30 seconds, and 60MB of memory on a 3Ghz processor
Outline

1 Motivations & other approaches
2 dSL
3 Distributing a dSL program
4 Implementation of dSL
5 Validation of dSL
6 Testing and Monitoring dSL programs
7 Future work
Model checking vs testing

Validation techniques

Model-Checking:
- *exhaustive* verification of model
- *but*: not (yet) scalable to real-sized systems

Testing and Monitoring:
- *non-exhaustive* verification: an execution *trace* is collected
- *scalable* for real-sized systems
- instrumented to emit *relevant* events (e.g. variable assignments, message transfer)
- *widely used* in industry

Testing / Monitoring

Testing:
- Checks that the execution (or simulated execution) is *compatible* with the specification
- Generally done *offline*

Monitoring:
- Separate process which collects those events
- Checks whether a certain *property* holds
- Generally done *online*
Centralized v.s. Distributed Systems

Traces

- **partially** ordered set of events
- events are labelled with **assignments**
- **finite** number of events

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>y:=3</th>
<th>x:=5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:=0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>P2</th>
<th>w:=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>w:=4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
testing and Monitoring = Trace model checking

**LTL model-checking with Monitor**
- non-deterministic finite automaton with guards on transitions
- built from an LTL formula using tableau construction [VW86]
- some states are marked as bad

**CTL model checking**
- adapted data structure records the set of configurations
- Interval Sharing Trees (IST)
- Interval Decision Diagram (IDD)
The problem

Question

Does there exist an *interleaving* (i.e. total order) of events *compatible* with the partial order, leading the monitor to a *bad* state?

Example

<table>
<thead>
<tr>
<th>P1</th>
<th>x:=0</th>
<th>y:=3</th>
<th>x:=5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>w:=4</td>
<td>w:=0</td>
<td></td>
</tr>
</tbody>
</table>

Theorem

The trace monitoring problem is *NP-Complete*.
The problem

Question

Does there exist an interleaving (i.e. total order) of events compatible with the partial order, leading the monitor to a bad state?

Example

<table>
<thead>
<tr>
<th>P1</th>
<th>x:=0 → y:=3 → x:=5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>w:=4 → w:=0</td>
</tr>
</tbody>
</table>

Theorem

The trace monitoring problem is NP-Complete
The problem

Question

Does there exist an interleaving (i.e. total order) of events compatible with the partial order, leading the monitor to a bad state?

Example

<table>
<thead>
<tr>
<th>Event</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:=0</td>
<td>-&gt;</td>
<td></td>
</tr>
<tr>
<td>y:=3</td>
<td>-&gt;</td>
<td></td>
</tr>
<tr>
<td>x:=5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w:=4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w:=0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Theorem

The trace monitoring problem is NP-Complete
The problem

Question
Does there exist an interleaving (i.e. total order) of events compatible with the partial order, leading the monitor to a bad state?

Example

```
<table>
<thead>
<tr>
<th>P1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x:=0</td>
<td></td>
<td>y:=3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x:=5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P2</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>w:=4</td>
<td></td>
<td>w:=0</td>
<td></td>
</tr>
</tbody>
</table>
```

```
init \rightarrow x > 0 \rightarrow tmp \rightarrow w = 0 \rightarrow bad
```

```
x := 0
```

Theorem
The trace monitoring problem is NP-Complete
The problem

Question

Does there exist an interleaving (i.e. total order) of events compatible with the partial order, leading the monitor to a bad state?

Example

<table>
<thead>
<tr>
<th>P1</th>
<th>x:=0</th>
<th>y:=3</th>
<th>x:=5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>w:=4</td>
<td>w:=0</td>
<td></td>
</tr>
</tbody>
</table>

\[ x:=0; w:=4 \]

Theorem

The trace monitoring problem is NP-Complete
The problem

Question

Does there exist an interleaving (i.e. total order) of events compatible with the partial order, leading the monitor to a bad state?

Example

P1

<table>
<thead>
<tr>
<th>x:=0</th>
<th>y:=3</th>
<th>x:=5</th>
</tr>
</thead>
</table>

P2

| w:=4 | w:=0 |

\[ x:=0; w:=4; y:=3 \]

\[ \text{init} \rightarrow x > 0 \rightarrow \text{tmp} \rightarrow w = 0 \rightarrow \text{bad} \]

Theorem

The trace monitoring problem is NP-Complete
The problem

Question

Does there exist an interleaving (i.e. total order) of events compatible with the partial order, leading the monitor to a bad state?

Example

\[
\begin{align*}
P1 & \quad x:=0 \quad y:=3 \quad x:=5 \\
P2 & \quad w:=4 \quad w:=0 \\
\text{init} & \quad x > 0 \quad \text{tmp} \quad w = 0 \\
\text{bad} & \\
\end{align*}
\]

\[x:=0; w:=4; y:=3; x:=5\]

Theorem

The trace monitoring problem is NP-Complete
The problem

Question
Does there exist an interleaving (i.e. total order) of events compatible with the partial order, leading the monitor to a bad state?

Example

<table>
<thead>
<tr>
<th>P1</th>
<th>x:=0 → y:=3 → x:=5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>w:=4 → w:=0</td>
</tr>
</tbody>
</table>

init \(\xrightarrow{x > 0}\) tmp \(\xrightarrow{w = 0}\) bad

\[x:=0; w:=4; y:=3; x:=5; w:=0\]

Theorem
The trace monitoring problem is NP-Complete
## Adapted symbolic exploration (LTL with monitor)

### Some experimental results

<table>
<thead>
<tr>
<th>Model</th>
<th>Processes</th>
<th>Events</th>
<th>Property</th>
<th>Error</th>
<th>Time</th>
<th>Conf.</th>
<th>Error</th>
<th>Time</th>
<th>Conf.</th>
<th>Spin</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peterson</td>
<td>2</td>
<td>10000</td>
<td>Mutex</td>
<td>NO</td>
<td>1.39s</td>
<td>21551</td>
<td>NO</td>
<td>0.35s</td>
<td>4001</td>
<td>0.06s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10000</td>
<td>Mutex</td>
<td>NO</td>
<td>16.88s</td>
<td>215544</td>
<td>NO</td>
<td>3.45s</td>
<td>40001</td>
<td>0.06s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>100000</td>
<td>Mutex</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Peterson</td>
<td>2</td>
<td>10000</td>
<td>Mutex</td>
<td>YES</td>
<td>1.11s</td>
<td>21384</td>
<td>YES</td>
<td>0.01s</td>
<td>4</td>
<td>0.05s</td>
<td></td>
</tr>
<tr>
<td>Faulty</td>
<td>2</td>
<td>10000</td>
<td>Mutex</td>
<td>YES</td>
<td>15.95s</td>
<td>214727</td>
<td>YES</td>
<td>0.05s</td>
<td>4</td>
<td>0.05s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>100000</td>
<td>Mutex</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ABProtocol</td>
<td>2</td>
<td>10000</td>
<td>Received</td>
<td>NO</td>
<td>2.17s</td>
<td>31185</td>
<td>NO</td>
<td>0.42s</td>
<td>4654</td>
<td>0.15s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>100000</td>
<td>Received</td>
<td>NO</td>
<td>31.08s</td>
<td>316414</td>
<td>NO</td>
<td>4.25s</td>
<td>46684</td>
<td>0.15s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>100000</td>
<td>Received</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ABProtocol</td>
<td>2</td>
<td>10000</td>
<td>Received</td>
<td>YES</td>
<td>2.06s</td>
<td>31495</td>
<td>YES</td>
<td>0.01s</td>
<td>5</td>
<td>0.13s</td>
<td></td>
</tr>
<tr>
<td>Faulty</td>
<td>2</td>
<td>10000</td>
<td>Received</td>
<td>YES</td>
<td>29.70s</td>
<td>315808</td>
<td>YES</td>
<td>0.06s</td>
<td>5</td>
<td>0.13s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>100000</td>
<td>Received</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Philosopher</td>
<td>3</td>
<td>100</td>
<td>Fork</td>
<td>NO</td>
<td>1.03s</td>
<td>6190</td>
<td>NO</td>
<td>0.05s</td>
<td>299</td>
<td>0.40s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>100</td>
<td>Fork</td>
<td>NO</td>
<td>87.02s</td>
<td>60727</td>
<td>NO</td>
<td>0.21s</td>
<td>2875</td>
<td>12.01s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>100</td>
<td>Fork</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Philosopher</td>
<td>3</td>
<td>100</td>
<td>Fork</td>
<td>YES</td>
<td>0.09s</td>
<td>1187</td>
<td>YES</td>
<td>0.01s</td>
<td>63</td>
<td>0.38s</td>
<td></td>
</tr>
<tr>
<td>Faulty</td>
<td>5</td>
<td>100</td>
<td>Fork</td>
<td>YES</td>
<td>78.72s</td>
<td>55982</td>
<td>YES</td>
<td>0.01s</td>
<td>78</td>
<td>11.01s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>100</td>
<td>Fork</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
### Some experimental results

<table>
<thead>
<tr>
<th>Model</th>
<th>#proc</th>
<th>#events</th>
<th>IST (in sec.)</th>
<th>NuSMV (in sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pet</td>
<td>2</td>
<td>2000</td>
<td>0.46</td>
<td>349.57</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5000</td>
<td>7.53</td>
<td>↑↑</td>
</tr>
<tr>
<td>PetN</td>
<td>2</td>
<td>2000</td>
<td>0.20</td>
<td>294.46</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5000</td>
<td>6.44</td>
<td>↑↑</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1000</td>
<td>2.04</td>
<td>13.74</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1500</td>
<td>6.82</td>
<td>↑↑</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1500</td>
<td>7.53</td>
<td>150.23</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2000</td>
<td>27.01</td>
<td>↑↑</td>
</tr>
<tr>
<td>ABP</td>
<td>2</td>
<td>1000</td>
<td>13.60</td>
<td>297.28</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2000</td>
<td>27.56</td>
<td>↑↑</td>
</tr>
<tr>
<td>Phil</td>
<td>3</td>
<td>100</td>
<td>0.15</td>
<td>6.36</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>200</td>
<td>1.11</td>
<td>↑↑</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2000</td>
<td>366.22</td>
<td>↑↑</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>100</td>
<td>1.67</td>
<td>↑↑</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>200</td>
<td>26.94</td>
<td>↑↑</td>
</tr>
</tbody>
</table>

**Table:** Experimental results; ↑↑ indicates (> 10 min.).
Outline

1. Motivations & other approaches
2. dSL
3. Distributing a dSL program
4. Implementation of dSL
5. Validation of dSL
6. Testing and Monitoring dSL programs
7. Future work
Future work

- Missing language features (Pointers Separate compilation,...)
- Extend preliminary work on a real time semantics for dSL [Mic05]
- Verification
  - Extends the language to model the environment
  - Encodes safety properties in the dSL program by assertions
- Partial verification, testing, monitoring [ELL01]
  - Guided search
  - Distributed monitoring
  - Distributed controller synthesis
Gerard Berry and Georges Gonthier.
The esterel synchronous programming language: Design, semantics, implementation.

H. Brinksma and R. Langerak.
Functionality decomposition by compositional correctness preserving transformation.

F. Bonfatti, P.D. Monari, and U. Sampieri.

Daniel Brand and Pitro Zafiropulo.
On communicating finite-state machines.

Gruia Calinescu, Howard Karloff, and Yuval Rabani.
An improved approximation algorithm for multiway cut.

K. M. Chandy and J. Misra.
*Parallel program design: a foundation.*

P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice.
Lustre: A declarative language for programming synchronous systems.

Bram De Wachter, Alexandre Genon, and Thierry Massart.
From static code distribution to more shrinkage for the multiterminal cut.

Bram De Wachter, Alexandre Genon, Thierry Massart, and Cédric Meuter.
The formal design of distributed controllers with dsl and spin.

The complexity of multiterminal cuts.


Bram De Wachter, Thierry Massart, and Cédric Meuter.

dsl: An environment with automatic code distribution for industrial control systems.


Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue.

Directed explicit model checking with HSF–SPIN.


E. Allen Emerson.

Temporal and modal logic, 1990.

L.R. Ford and D.R. Fulkerson.

*Flows in Networks*.


A. Girault.

*Sur la Répartition de Programmes Synchrones*.


Andrew V. Goldberg and Robert E. Tarjan.

A new approach to the maximum-flow problem.


ISSN:0004-5411.

G. J. Holzmann.

*The SPIN Model Checker : Primer and Reference Manual*.

Addison-Wesley Professional, September 2003.


David R. Karger, Philip Klein, Cliff Stein, Mikkel Thorup, and Neal E. Young.

Rounding algorithms for a geometric embedding of minimum multiway cut.
Programming real time applications with signal.

T. Massart.
A calculus to define correct transformations of LOTOS specifications.

Nicolas Micheli.
Design and verification of real-time distributed industrial control systems.

Moshe Y. Vardi and Pierre Wolper.
An automata-theoretic approach to automatic program verification (preliminary report).
dSL’s semantics - Input/Output

### Input/Output

- Are the only visible actions

#### [Input]

\[
(E_D^P)_i \vdash (\text{INPUT}(x);\omega_i, \nu_i, \phi_i) \xrightarrow{x?a} (\text{BCAST}(x);\omega_i, \nu_i[x \mapsto a], \phi_i)
\]

\[
\forall a \in \{\top, \bot\}
\]

#### [Output]

\[
(E_D^P)_i \vdash (\text{OUTPUT}(x);\omega_i, \nu_i, \phi_i) \xrightarrow{x!\nu_i(x)} (\omega_i, \nu_i, \phi_i)
\]
dSL’s semantics - Assignment

**Assignmant**

[Assignment]

\[(E_D^P)_i \vdash (x := e; \omega_i, \nu_i, \phi_i) \xrightarrow{\tau} \begin{cases} \text{BCAST}(x); \text{Treat}(W_{i/x}, \prec w); \omega_i, \nu_i[x \mapsto \nu_i(e)], \phi_i & \text{if } x \in \text{Var}(P) \\ (\omega_i, \nu_i[x \mapsto \nu_i(e)], \phi_i) & \text{if } x \in \text{OldCond}(P) \end{cases}\]
dSL’s semantics - SOS-Rules

Interleaving

[Interleaving]

\[
\frac{(E_D^P)_i \vdash (\omega_i, \nu_i, \phi_i) \xrightarrow{a} (\omega'_i, \nu'_i, \phi'_i)}{E^P_D \vdash ((\omega_1, \nu_1, \phi_1), ..., (\omega_i, \nu_i, \phi_i), ..., (\omega_n, \nu_n, \phi_n), \sigma_1, ..., \sigma_\ell, \mu, \xi) \xrightarrow{a} ((\omega'_1, \nu'_1, \phi'_1), ..., (\omega'_i, \nu'_i, \phi'_i), ..., (\omega'_n, \nu'_n, \phi'_n), \sigma_1, ..., \sigma_\ell, \mu, \xi)}
\]
Known facts on the Multiterminal Cut

Algorithms and Complexity results

- **NP-Complete problem**
- **Polynomially solvable**
  - if \( k = 2 \) it reduces to ST-Cut
    - Famous max-flow/min-cut theorem by Ford and Fulkerson [FF62]
    - *Best* known algorithm: Goldberg/Tarjan \( O(|V||E| \log(\frac{|V|^2}{|E|})) \) [GT88]
  - if \( G \) is planar
- **Two approaches**
  - Dalhaus et al [DJP+94]
    - Based on min-cut/max-flow, \( \alpha = 2 - \frac{2}{k} \)
    - Isolation heuristics, find an isolating cut for each terminal, take union of \( k - 1 \) such cuts
    - \( O(knm \log(n^2/m)) \)
  - Calinescu et al [CKR00]
    - Based on Linear Programming, \( \alpha = 1.5 - \frac{1}{k} \)
    - Lowered by Karger [KKS+99] to 1.3438
PROMELA skeleton for a site

```
proctype site_X() {
    atomic { init_site_X(); }
    byte sz1, sz2, /*...*/; szN_SITES;
    do :: atomic {
        d_step {
            sz1 = len (chan_X_1);
            sz2 = len (chan_X_2);
            // ...
            sz_N_SITES = len (chan_X_N_SITES);
        }
        input_X();    // Read inputs
        when_X();     // Treat events
    }
    read_msg_X();   // Treat X’s message queue
    atomic {
        output_X();   // Write outputs
    }
    od
}
```