Domain-Specific Formalisms and Model-Driven Development for Railway Control Systems

Jan Peleska,
Kirsten Berkenkötter, Rolf Drechsler, Daniel Große, Ulrich Hannemann, Anne E. Haxthausen, Sebastian Kinder
{jp,kirsten,drechsle,kinder,grosse,ulrichh}@tzi.de,
ah@imm.dtu.dk

University of Bremen and Technical University of Denmark

Train@SEFM2005
2005-09-05
Outline

- Introduce domain-specific formalism for requirements specification of train/tram control systems
- Show that formalism can be embedded into UML2.0 as a profile
- Describe automated transformation of requirements into fully formal low-level model and associated verification conditions
- Explain automated verification based on bounded model checking (BMC) and inductive proof strategy
- Sketch automated transformation of low-level controller model into machine code and associated equivalence/refinement proof
- Motivate where automated HW/SW integration testing is still needed and explain how full test automation is achieved

Case Study: Control system for a tram maintenance site
Background – Observations

Today, conventional development of train control systems typically proceeds along the following lines:

▶ Specification and design of **generic control system** which can be instantiated for concrete domains of control (i.e., railway nets)
▶ Manual software development in programming languages like C/C++, Pascal or domain-specific languages (Sternol)
▶ Generation of executable code using validated compilers
▶ Full semi-formal verification of generic system (**"type certification"**)  
▶ Instantiation of generic system for concrete domain of control by means of configuration data
▶ Full semi-formal verification of the configuration data
▶ Partial verification of the resulting concrete system
Background – Observations

Today’s development approach frequently encounters the following problems:

▶ Too much effort spent in manual coding phase, since re-use and utilisation of design patterns is not properly managed
▶ ⇒ Too much effort spent on code verification
▶ Exhaustive verification of configuration data is expensive and requires considerable manual effort
▶ Some errors in the generic system only come up when specific configuration data is used:
  ▶ ⇒ semi-formal verification of a generic system does not ensure correctness of all instances
  ▶ ⇒ semi-formal verification of a generic system does not ensure correct integration of HW/SW system
Domain of Control and Controller

- The **Domain of Control (Physical Model)** specifies the railway net and the behaviour of trains on the net.
- The **Controller** monitors:
  - sensors – train locations derived from sensor states
  - signal states
  - point states
and sends commands to:
- signals
- points
Domain of Control and Controller

incoming trains

Domain of Control
(Physical Model)

Railway network
+ Trains
+ Safety Condition $\Phi$

sensor-states
signal-states
point-states
route-requests
signal-ctrlcmds
point-ctrlcmds

Controller
(Control Model)

outgoing trains

Peleska, Berkenkötter, Drechsler, Große, Hannemann, Haxthausen, Kinder
V-Model for Model-Based Development and Verification

- **Step 1.** Manual requirements specification process:
  - System requirements for domain of control – static aspects: Net model + route model
  - Architectural specification of controller (= target system to be developed)
  - Physical constraints specification

Specification formalism: UML2.0 with Railway Control System Domain Profile RCSD
V-Model for Model-Based Development and Verification

- **Step 2.** Automated generation of
  - Behavioural model for domain of control
  - Behavioural model for controller
  - Verification conditions for safety properties

Specification formalism:
- Timed state-transition systems – SystemC syntax
- Verification obligations formulated as “simple” temporal logics assertions over bounded discrete time intervals
Step 3. Automated verification of controller model:
- Inductive verification strategy
- Bounded model checking

Step 4. Automated generation of executable code:
- Assembler/machine code generated directly from controller model – structured as instance of generic interpreter and configuration data
- Formal proof of equivalence between timed state-transition system model and machine code interpreter for all admissible instances of configuration data is feasible
Domain-specific description . . .

... consists of

- **Net model:** required to be correct
- **Route model:** Tables for
  - Route definition
  - Specification of conflicting routes
  - Required point positions associated with routes
  - Required signal settings associated with routes

  to be automatically verified with respect to safety properties

- **Safety model:** consists of net model + transition rules for trains, depending on point and signal states
Domain-specific requirements: concrete net model

TRAMWAY MAIN ROUTES:
1: S20–G21.1 (NORTH–SOUTH)
3: S21–G23.1 (SOUTH–NORTH)

ROUTE 1:
S21–G23.1

ROUTE 2:
S21–G23.1

ROUTE 3:
S21–G25.1

ROUTE 4:
S22–G23.1

ROUTE 5:
S22–G21.1

TRAM MAINTENANCE SITE
Domain-specific requirements: Route model

<table>
<thead>
<tr>
<th>Route</th>
<th>Route Sensor Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⟨G20.1, G20.2, G21.0, G21.1⟩</td>
</tr>
<tr>
<td>1</td>
<td>⟨G20.1, G20.3, G25.0, G25.1⟩</td>
</tr>
<tr>
<td>2</td>
<td>⟨G22.1, G22.2, G23.0, G23.1⟩</td>
</tr>
<tr>
<td>3</td>
<td>⟨G22.1, G22.3, G25.0, G25.1⟩</td>
</tr>
<tr>
<td>4</td>
<td>⟨G24.1, G24.3, G23.0, G23.1⟩</td>
</tr>
</tbody>
</table>

Table 1. Route definition table.

Peleska, Berkenkötter, Drechsler, Große, Hannemann, Haxthausen, Kinder
Domain-specific requirements: Route model

<table>
<thead>
<tr>
<th>Route</th>
<th>W100</th>
<th>W102</th>
<th>W118</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>straight</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>left</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>straight</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>right</td>
</tr>
<tr>
<td>4</td>
<td>right</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>straight</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 2. Point position table.
Domain-specific requirements: Route model

<table>
<thead>
<tr>
<th>Route</th>
<th>Signal</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S20</td>
<td>go-straight</td>
</tr>
<tr>
<td>1</td>
<td>S20</td>
<td>go-left</td>
</tr>
<tr>
<td>2</td>
<td>S21</td>
<td>go-straight</td>
</tr>
<tr>
<td>3</td>
<td>S21</td>
<td>go-right</td>
</tr>
<tr>
<td>4</td>
<td>S22</td>
<td>go-right</td>
</tr>
<tr>
<td>5</td>
<td>S22</td>
<td>go-straight</td>
</tr>
</tbody>
</table>

Table 3. Signal setting table.
Domain-specific requirements: Route model

Table 4. Route conflict table.

<table>
<thead>
<tr>
<th>Route</th>
<th>Conflicts with</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>●</td>
</tr>
<tr>
<td>2</td>
<td>○</td>
</tr>
<tr>
<td>3</td>
<td>○</td>
</tr>
<tr>
<td>4</td>
<td>○</td>
</tr>
<tr>
<td>5</td>
<td>○</td>
</tr>
</tbody>
</table>
Domain-specific description as UML2.0 profile

- **TramSegment**: `<<Segment>>`  
  - `trackId: TrackId`  
  - `crossing: TrackId`  
  - `maxNumberOfTrains: Integer`

- **TramPoint**: `<<Point>>`  
  - `pointId: PointId`  
  - `actualState: PointStateKind`  
  - `requestedState: PointStateKind`  
  - `requestTime: TimeInstant`  
  - `delta_p: TimeInterval`  
  - `delta_tram: TimeInterval`  
  - `routeDefinition: SensorId[0..*]`  
  - `signalSetting: <<SignalSetting>>`  
  - `pointPos: <<PointPosition>>[0..*]`  
  - `routeId: RouteId`  
  - `routeConflict: <<RouteConflict>>[0..*]`

- **TramSignal**: `<<Signal>>`  
  - `signalId: SignalId`  
  - `actualState: SignalStateKind`  
  - `requestedState: SignalStateKind`  
  - `requestTime: TimeInstant`  
  - `delta_s: TimeInterval`  

- **TramSensor**: `<<Sensor>>`  
  - `sensorId: SensorId`  
  - `actualState: SensorStateKind`  
  - `sentTime: TimeInstant`  
  - `counter: Integer`  
  - `delta_l: TimeInterval`  
  - `sentTime: TimeInstant`  
  - `counter: Integer`  
  - `delta_s: TimeInterval`  

Peleska, Berkenkötter, Drechsler, Große, Hannemann, Haxthausen, Kinder
UML2.0 profile construction

- **Step 1.** introduction of profile-specific primitive types and enumerations
- **Step 2.** introduction of stereotypes and their associations with elements ("meta-classes") of the meta-model
- **Step 3.** definition of properties for each stereotype by means of OCL
- **Step 4.** association of domain-specific graphical symbols with instances of each stereotype
Specification of Model Behaviour

▶ Generation of net-specific transition rules: Instantiated from generic rule patterns and concrete net model.
▶ Transition rules specify conditions for pre-state $\rightarrow$ post-state changes.
▶ Example: Domain of control transition rule for trains passing sensors:

```python
if ( (c_G221 < c_G220) && (sen_G221 == SEN_LOW) && (actsig_S21 != SIG_HALT) && (c_G221 == c_G222)) {
    sen_G221 = SEN_HIGH;
    c_G221 = c_G221 + 1;
    sentm_G221 = t;
}
```
Specification of Model Behaviour

Example: Controller transition rule for detection of train entering route 0:

```java
if ( rc_cmv(0) == ALLOCATED
    // Route 0 is safe for use
    and
    cc(G20.1) == cc(G20.2) + cc(G20.3)
    // Tram has passed both G20.1 and G20.2
) {
    reqsig(S20) = HALT;
    // Request for signal S20: switch back to HALT
    reqsigtm(S20) = t;
    rc_cmv(0) = OCCUPIED;
    // Mark route 0 as IN USE
}
```
Verification by Bounded Model Checking (BMC)

BMC checks whether properties $P$ hold over a discrete time interval $I = \{ t, t + 1, \ldots, t + c \}$.

**BMC Strategy:** check whether

$$
0 = \bigwedge_{j=0}^{c-1} T_\delta( i(t + j), s(t + j), s(t + j + 1) ) \land \\
\neg P( i(t), s(t), o(t), \ldots, i(t + c), s(t + c), o(t + c) )
$$

can be satisfied for one sequence of transitions consistent with transition relation $T_\delta$ — this falsifies property $P$ in $I$. 
Verification by Bounded Model Checking

Inductive principle:

- Specify the safety constraints
- Prove that constraints hold in initial state
- **Induction hypothesis:** Assume that constraints hold in arbitrary pre-state
- **Induction step:** Prove that all possible transitions from pre-state lead to safe post-state

Note: Detailed proof requires to argue over more than one time step – the longest interval required is \( I = t, t + 1, t + 2, t + 3, t + 4 \)

Further details: see Sebastian Kinder’s presentation tomorrow!
Verification by Bounded Model Checking – Example

SystemC proof obligation for checking assertion

- *Sensor counters managed by controller will deviate from real sensor state by at most one.*

- *The difference only occurs if physical sensor just changed from LOW to HIGH.*
Verification by Bounded Model Checking – Example

 theorem th_counter is
 assume:
during[t,t+1]: <...additional properties...>
at t+1:
 (c(g) = cc(g))
or ( sen(g) = HIGH and prev(sen(g)) = LOW
 and c(g) = cc(g) + 1 );
prove:
during [t+2,t+4]:
 (c(g) = cc(g))
or ( sen(g) = HIGH and prev(sen(g)) = LOW
 and c(g) = cc(g) + 1 );
end theorem;
Machine Code Generation – HW abstraction layer

Dual-ported RAM interface drivers ↔ safety layer:
Machine Code Generation – state/command encoding

Encoding of element states and commands as machine words (32 bits) ensures

- Interleaving semantics for all transitions – even in presence of multi threading on several CPUs
- Encoding of all conditions according to pattern
  
  \[ ((\text{operand1} \& \text{mask1}) \gg \text{shift1}) \]
  
  \[ \text{comparison}\_\text{operator} \]
  
  \[ ((\text{operand2} \& \text{mask2}) \gg \text{shift2}) \]

- Encoding of all actions as unary or binary operations:

  \[ \text{operand1} = 0; \]
  
  \[ \text{operand1}++; \]
  
  \[ \text{operand1} = \text{clock tick}; \]
  
  \[ \text{operand1} = -\text{operand1}; \]
  
  \[ \text{operand1} = \text{operand2} +/\- \text{operand3}; \]
Machine Code Generation – transition encoding

Transitions are encoded as

m1: loop over number of condition conjuncts, 
0 <= i < max 
  b = evaluation of condition i 
  according to pattern above 
  if ( not(b) ) jump m2 
  i++ 
  if ( i < max ) jump m1 
process action associated with transition 

m2: continue
Considerations above lead to the following strategy:

- Transformation from SystemC model to assembler code can be performed following a small number of very simple transformation patterns for
  - task main loop
  - transition processing
  - condition processing
  - action processing

- Conditions and actions are encoded as data – to be interpreted by instance of generic assembler code
Machine Code Generation

- Interpreter and encodings require very few CPU capabilities: Less than 10 user registers – bitwise AND – shift etc.
- $\Rightarrow$ Formal model of CPU behaviour and memory is easy to construct
- $\Rightarrow$ Abstraction mapping between SystemC model and assembler code is straightforward
- $\Rightarrow$ Behavioural equivalence between timed state transition systems and machine code/data can be verified universally, that is, for all legal models.
Conclusion

- We have presented an automated development and verification approach for executable code + configuration data of train control systems.
- The verification was based on bounded model checking (BMC), following an inductive principle for reasoning about safety properties.
- The BMC approach allows to handle verification problems of the described kind in an efficient way, because it does not require to explore complete state spaces, starting with system initialisation.
- The feasibility of machine code verification depends on the applicability of a small number of design patterns in the formal low-level model.
Ongoing research

- Final versions of generators for SystemC models, verification conditions and machine code.
- **Widening the scope of the domain:** Include
  - railway crossings
  - Railway-specific safety conditions: shunts, flank protection, ...
  - hybrid control aspects – speed, breaking curves
  \[ \Rightarrow \] a UML2.0 profile for specifying hybrid control has already been established
- **CASE Tools:** Plug-ins for checking static semantics of specifications based on profiles
- **Automated testing:** novel algorithms for model-based test case generation – can BMC help to find “relevant” test traces?