Model-Based Development of Safety-Critical Systems

Jan Peleska,
Johannes Adams, Kirsten Berkenkötter, Rolf Drechsler, Daniel Große, Ulrich Hannemann, Anne E. Haxthausen, Sebastian Kinder, Helge Löding, Waldemar Wockenfuß
jp@tzi.de

University of Bremen and Technical University of Denmark

Siemens Best Practice Sharing – RAMS
2005-11-01

Outline

- Session 1: Model-Based Development of Safety-Critical Systems – Concepts – Methodologies
- Session 2: The UML Approach to Model-Driven Development
- Session 3: UML2.0-Based Solutions to Automated Model-Based Development, Verification, Validation and Testing

Session 1: Model-Based Development of Safety-Critical Systems – Concepts – Methodologies

Session 1: Model-Based Development of Safety-Critical Systems – Concepts – Methodologies

- Model-based development – terms and definitions
- Model-based development – motivation
- Example 1: Refinement of state-machines
- Example 2: Transformational approach for state-machines
- Example 3: Data and data transformation refinement
- Model-Based Development – a survey of formalisms
- A survey of theoretic foundations

Model-based development – terms and definitions

System Model: abstract representation of a system, usually constructed by collection of sub-models reflecting different system properties:

- Functional properties:
  - Data (state) model
  - Data transformation
  - Behaviour
    - Causality
    - Synchronisation
    - Timing

Model-based development – terms and definitions

System Model (continued):

- Structural properties:
  - Components
  - Interfaces
  - Control structure of algorithms
- Non-functional properties:
  - RAMS = dependability (reliability, availability, safety, security) + maintainability,
  - Usability
  - Quality of service
  - ...

Model-based development – terms and definitions

Specification types:

- Explicit (functional) specifications are complete models describing data, transformations and behaviour
- Implicit specifications or properties are logical assertions about models – special types of implicit specifications are
  - Safety properties always hold during a model execution
  - Liveness properties hold finally for each model execution
- Algebraic specifications are models abstracting from data
- Hybrid or discrete-continuous specfcations describe both the behaviour of observables changing
  - only at discrete points in time
  - according to piecewise continuous (differentiable, analytic) functions over time

Model-based development – terms and definitions

Formalisms for models consist of

- Syntax: the visual representation of models
- Semantics: the meaning of admissible syntactic constructs
  - Denotational semantics assigns meaning by mathematical specification of the effect of specification constructs on model state and I/O sequences
  - Operational semantics assigns meaning by construction of an abstract interpreter operating on the state space in a way which is equivalent to the specification behaviour

Model-based development – terms and definitions

Formalisms may be classified according to their “closeness” to the application domain

- Domain-specific formalisms use terms and objects of the application domain – e. g. railway track sections, signals, points
- Wide-spectrum formalisms use abstract language elements which can be mapped to objects of arbitrary application domains – e. g. Statecharts, decision tables, logical formulae
- Machine-oriented formalisms use terms and objects of the target system where the solution to the problem shall be implemented – e. g. assembler code, CPU models with registers, cache, microcode

Model-based development – terms and definitions

Model-based development is a formalism together with a set of rules how to

- construct executable systems – HW and SW – from models,
- verify that an implementation conforms to the model.

Goal: Derive executable system from model in an automatic way!

Model-based development – Motivation

- Improve problem understanding by using suitable abstractions in model
- Generate executable code faster
- Apply automated model-based testing to improve HW/SW integration quality and speed up the verification process
- Automated code generation ensures
  - Unified handling of design patters
  - Code compliance with coding standards
  - Avoidance of errors during transformation from model to code

Model-based development – terms and definitions

Two alternative approaches for model-based development:

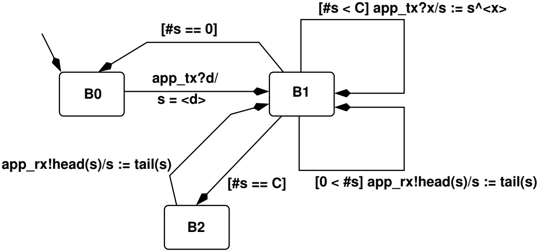
- Stepwise refinement (invent-and-verify paradigm):
  - Invent a more concrete representation S_{i+1} of the system S_i to be developed
  - Prove that S_{i+1} is equivalent or – slightly weaker – a valid refinement of S_i
  - Refine S_{i+1} ...
  - until most refined version is directly executable.
- Transformational development directly compiles specification models into executable systems.

Example 1: Refinement of state-machines

The CSP – Communicating Sequential Processes formalism for describing networks of cooperating automata with local variables

Example 1: Refinement of state-machines

FIFO buffer with capacity C



CSP representation of FIFO buffer

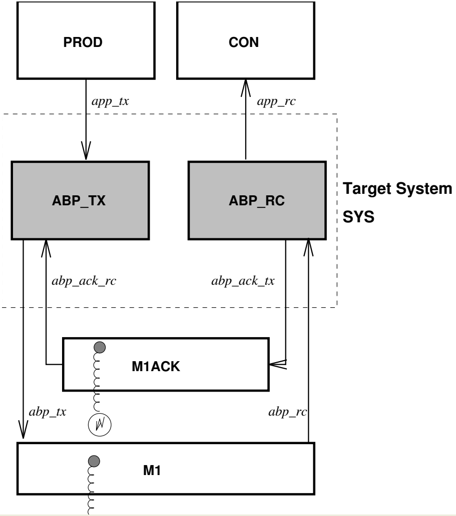
B0 = app_tx?d –> B1(<d>)

B1(s) = (#s == 0)&B0
[]
(#s < C)&app_tx? –> B1(s^<x>)
[]
(0 < #s)&app_rx!head(s) –> B1(tail(s))
[]
(#s == C)&B2(s)

B2(s) = app_rx!head(s) –> B1(tail(s))

Example 1: Refinement of state-machines

Architecure for Alternating Bit Protocol



Example 2: Transformational approach for state-machines

Operational semantics of CSP can be interpreted in hard real-time!

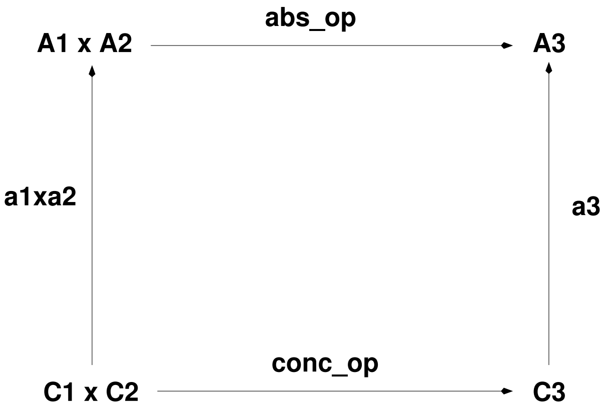
- Process states are nodes of transition graph
- Events cause state transitions between nodes
- Transition graph can be generated from CSP model
- Interpreter traverses transition graph
- Interface modules implement mapping between abstract events and concrete interfaces (refinement – abstraction)

Example 3: Data and data transformation refinement

Data and data transformation refinement is performed using the following steps:

- Construct abstraction mapping between abstract and concrete data structures
- Invent concrete operation
- Verify that – when applying the abstraction mapping – the concrete operation implements the abstract one

Example 3: Data and data transformation refinement



Correctness condition:

forall (c1,c2) in C1 x C2 . abs_op(a1(c1),a2(c2)) = a3(conc_op(c1,c2))

Session 2: The UML Approach to Model-Driven Development

Model-Driven Development

The Object Management Group¡Çs view on Model-Based Development:

- Model-Driven Architecture: A framework for transforming models, for example,
  - From UML class diagrams to relational data base schema
  - From UML class diagrams+method specifications in OCL to schema + SQL query code
  - From UML Statecharts to C++ code for embedded systems
  - From UML Statecharts to UML Sequence Diagrams
  - ...

Model-Driven Development

Standard approach for MDA utilisation:

- Elaborate Platform-Independent Model (PIM)
- Transform PIM to one or more Platform-Specific Models (PSM)
- “Simple Transformation” from PSMs to code

Session 3: UML2.0-Based Solutions to Automated Model-Based Development, Verification, Validation and Testing

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:

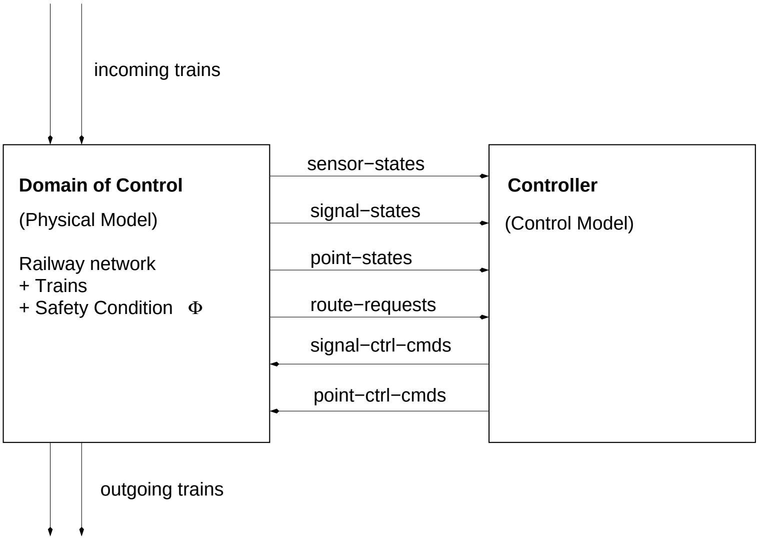
- Too much effort spent in manual coding phase, since re-use and utilisation of design patterns is not properly managed
- $\Rightarrow$ Too much effort spent on code verification
- Exhaustive verification of configuration data is expensive and requires considerable manual effort
- Some errors in the generic system only come up when specific configuration data is used:
  - $\Rightarrow$ semi-formal verification of a generic system does not ensure correctness of all instances
  - $\Rightarrow$ 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

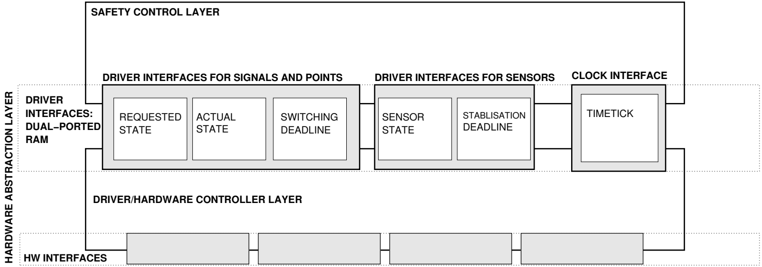
  and sends commands to
  - signals
  - points

Domain of Control and Controller



Machine Code Generation – HW abstraction layer

Dual-ported RAM interface drivers ↔ safety layer:



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

V-Model for Model-Based Development and Verification

- 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

Session 3: UML2.0-Based Solutions to Automated Model-Based Development, Verification, Validation and Testing

- UML2.0 Profile for train/tram control systems
- Automated transformation of requirements into formal SystemC low-level model and associated verification conditions
- Automated verification based on bounded model checking (BMC) and inductive proof strategy
- Automated machine code generation and verification
- Model Validation by property checking – simulation – testing
- System validation by automated HW/SW integration testing
- Motivate where automated HW/SW integration testing is still needed and explain how full test automation is achieved

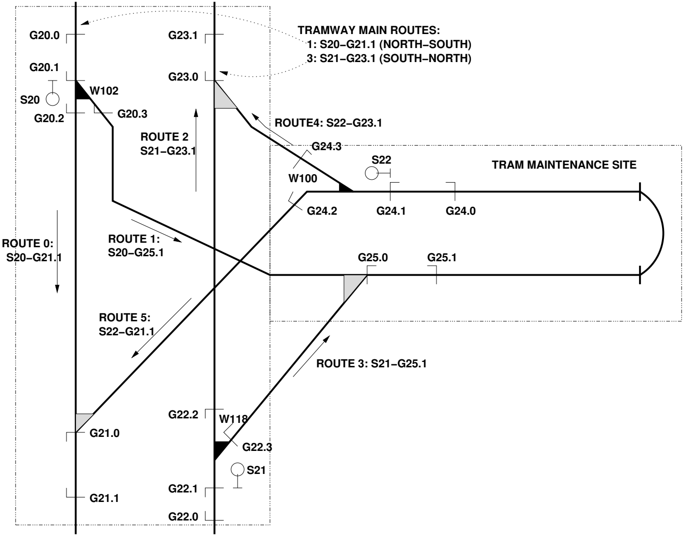
Domain-specific description . . .

. . . 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



Domain-specific requirements: Route model

Route definition table	
Route	**Route Sensor Sequence**
0	$\langle G20.1, G20.2, G21.0, G21.1\rangle$
1	$\langle G20.1, G20.3, G25.0, G25.1\rangle$
2	$\langle G22.1, G22.2, G23.0, G23.1\rangle$
3	$\langle G22.1, G22.3, G25.0, G25.1\rangle$
4	$\langle G24.1, G24.3, G23.0, G23.1\rangle$
5	$\langle G24.1, G24.2, G21.0, G21.1\rangle$

Table 1. Route definition table.

Domain-specific requirements: Route model

Point position table			
Route	**W100**	**W102**	**W118**
0	—	straight	—
1	—	left	—
2	—	—	straight
3	—	—	right
4	right	—	—
5	straight	—	—

Table 2. Point position table.

Domain-specific requirements: Route model

Signal setting table		
Route	**Signal**	**Setting**
0	S20	go-straight
1	S20	go-left
2	S21	go-straight
3	S21	go-right
4	S22	go-right
5	S22	go-straight

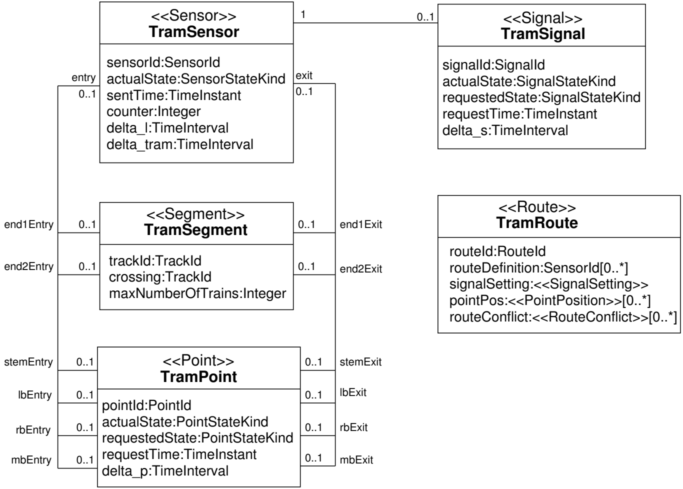
Table 3. Signal setting table.

Domain-specific requirements: Route model

Route conflict table						
Route	**Conflicts with**					
	0	1	2	3	4	5
0		●				○
1	●		○	○		○
2		○		●	○	○
3		○	●			
4			○			●
5	○	○	○		●	

Table 4. Route conflict table.

Domain-specific description as UML2.0 profile



UML2.0 profile construction

- Step 1. introduction of profile-specific primitive types and enumerations
- Step 2. introduction of stereotypes an 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 $\longrightarrow$ post-state changes.
- Example: Domain of control transition rule for trains passing sensors:

```
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:

```
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

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

can be satisfied for one sequence of transitions consistent with transition relation T_δ — 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 $I = t, t+1, t+2, t+3, t+4$

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 – 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

```
((operand1 & mask1) >> shift1)
  comparison_operator
((operand2 & mask2) >> shift2)
```

- Encoding of all actions as unary or binary operations:

```
operand1 = 0;
operand1++;
operand1 = clock tick;
operand1 = -operand1;
operand1 = operand2 +/- 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
```

Machine Code Generation

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 straight forward
- $\Rightarrow$ 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
    ⇒ 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?