Pragmatic integration of model driven engineering and formal methods for safety critical systems design

Marc Pantel and many others

IRIT - ACADIE — ENSEEIHT - INPT - Université de Toulouse

Institute of Cybernetics Institute — Tallinn University of Technology
Parrot exchange — January the 20th 2011

Plan

Plan

1. Safe MDE concerns
2. Certification and Qualification
3. Application to Code generation tools
4. Application to Static analysis tools
5. The Executable DSML metamodeling pattern

Safe MDE concerns

- Main purpose: Safety critical systems
- Main approach: formal specification and verification
- Problems: expressiveness, decidability, completeness, consistency

Safe MDE concerns II

- Proposals: Raise abstraction
  - Higher level programming languages and frameworks
  - Domain specific (modeling) languages
    - easy to access for end users
    - with a simple formal embedding
    - with automatic verification tools
    - with usefull validation and verification results
    - that are accepted by certification authorities
- Needs:
  - methods and tools to ease their development
  - algebraic and logic theoretical fondations
  - proof of transformation and verification correctness
  - links with certification/qualification

Safe MDE concerns II

- Proposals: Raise abstraction
  - Higher level programming languages and frameworks
  - Domain specific (modeling) languages
    - easy to access for end users
    - with a simple formal embedding
    - with automatic verification tools
    - with usefull validation and verification results
    - that are accepted by certification authorities
- Needs:
  - methods and tools to ease their development
  - algebraic and logic theoretical fondations
  - proof of transformation and verification correctness
  - links with certification/qualification

Related past projects

- RNTL COTRE: Transformation to verification languages
- ACI FIACRE: Intermediate verification language
- ITEA GeneAuto: Qualified Simulink/Stateflow to C code generator
  TUT and IB-Krates partners
- ITEA ES_PASS: Static analysis for Product insurance
- ITEA SPICES: AADL behavioral annex
- ANR OpenEmbedd: AADL to FIACRE verification chain (Kermeta based)
- CNES (French Space Agency) AutoJava: profiled UML to RTSJ code generator

ACADIE

Related current projects

- FUI TOPCASED: Metamodels semantics, Model animators, Verification chains based on model transformations
- ANR SPaCIFY: GeneAuto + AADL = Synoptic <-> Polychrony (Kermeta based)
- ANR iTemis: SOA/SCA verification
- FRAE quarteFt: model transformation based on Java/TOM for AADL to FIACRE
- ITEA2 OPEES: Formal methods and Certification authorities
- JTI ARTEMIS CESAR: V & V view for safety critical components.

Plan

1. Safe MDE concerns
2. Certification and Qualification
3. Application to Code generation tools
4. Application to Static analysis tools
5. The Executable DSML metamodeling pattern

A bit of wording

- Requirement: What the end user expects from a system
    - High level: focus on end users needs (user provided)
        - Translate profiled UML to RTSJ; C to PowerPC
        - Generate test inputs and expected outputs from a system specification
        - Prove the absence of runtime errors
        - Compute a precise estimation of WCET
        - Schedule activities
    - Low level: focus on technical solutions (developer provided)
        - Relies on abstract interpretation for properties estimation
        - on graph coloring for register allocation
        - on linear programming for task scheduling
        - Generates a C function for each Simulink atomic sub-system
        - a RTSJ class for each UML class
    - Traceability links between various requirements, design and implementation choices

A bit of wording

- Requirement: What the end user expects from a system
    - High level: focus on end users needs (user provided)
        - Translate profiled UML to RTSJ; C to PowerPC
        - Generate test inputs and expected outputs from a system specification
        - Prove the absence of runtime errors
        - Compute a precise estimation of WCET
        - Schedule activities
    - Low level: focus on technical solutions (developer provided)
        - Relies on abstract interpretation for properties estimation
        - on graph coloring for register allocation
        - on linear programming for task scheduling
        - Generates a C function for each Simulink atomic sub-system
        - a RTSJ class for each UML class
    - Traceability links between various requirements, design and implementation choices

A bit of wording

- Requirement: What the end user expects from a system
    - High level: focus on end users needs (user provided)
        - Translate profiled UML to RTSJ; C to PowerPC
        - Generate test inputs and expected outputs from a system specification
        - Prove the absence of runtime errors
        - Compute a precise estimation of WCET
        - Schedule activities
    - Low level: focus on technical solutions (developer provided)
        - Relies on abstract interpretation for properties estimation
        - on graph coloring for register allocation
        - on linear programming for task scheduling
        - Generates a C function for each Simulink atomic sub-system
        - a RTSJ class for each UML class
    - Traceability links between various requirements, design and implementation choices

A bit of wording II

- Verification: System fulfills its requirements explicit specification
- Validation: System fulfills its requirements implicit human needs
- Certification: System (and its development) follows standards (DO-178, IEC-61508, ISO-26262, . . .)
- Qualification: Tools for system development follows standards
- Certification and qualification: System context related

A bit of wording II

- Verification: System fulfills its requirements explicit specification
- Validation: System fulfills its requirements implicit human needs
- Certification: System (and its development) follows standards (DO-178, IEC-61508, ISO-26262, . . .)
- Qualification: Tools for system development follows standards
- Certification and qualification: System context related

A bit of wording II

- Verification: System fulfills its requirements explicit specification
- Validation: System fulfills its requirements implicit human needs
- Certification: System (and its development) follows standards (DO-178, IEC-61508, ISO-26262, . . .)
- Qualification: Tools for system development follows standards
- Certification and qualification: System context related

DO-178B/ED-12B standards: Certification

- Software in aeronautics: Design Assurance Level (A down to E)
- Most constraining standards up to now
  accepted by other standards (automotive, space, . . .)
- Main concern: Safety of passengers
- Main purpose: Provide confidence in the system and its development
- Key issue: Choose the strategy and technologies that will minimize risks (no restriction)
- Process and test-centered approach
  - Definition of a precise process (development/verification)
  - MCDC test coverage
    truth-table lines of sub-expressions in conditions
  - Asymmetry with independence argument: several implementation by different teams, with different tools, . . .

DO-178B/ED-12B standards: Certification

- Software in aeronautics: Design Assurance Level (A down to E)
- Most constraining standards up to now
  accepted by other standards (automotive, space, . . .)
- Main concern: Safety of passengers
- Main purpose: Provide confidence in the system and its development
- Key issue: Choose the strategy and technologies that will minimize risks (no restriction)
- Process and test-centered approach
  - Definition of a precise process (development/verification)
  - MCDC test coverage
    truth-table lines of sub-expressions in conditions
  - Asymmetry with independence argument: several implementation by different teams, with different tools, . . .

DO-178B/ED-12B standards: Certification

- Software in aeronautics: Design Assurance Level (A down to E)
- Most constraining standards up to now
  accepted by other standards (automotive, space, . . .)
- Main concern: Safety of passengers
- Main purpose: Provide confidence in the system and its development
- Key issue: Choose the strategy and technologies that will minimize risks (no restriction)
- Process and test-centered approach
  - Definition of a precise process (development/verification)
  - MCDC test coverage
    truth-table lines of sub-expressions in conditions
  - Asymmetry with independence argument: several implementation by different teams, with different tools, . . .

DO-178B/ED-12B standards: Qualification

- Development tools: Tools whose output is part of airborne software and thus can introduce errors (same constraints as the developed system).
- Verification tools: Tools that cannot introduce errors, but may fail to detect them (much softer constraints: black box V & V).
- No proof of error absence category

DO-178C/ED-12C standards: Qualification

- Introduce detailed Tool Qualification Level (1 downto 5)
- Criteria 1: A tool whose output is part of the resulting software and thus could insert an error (TQL-1 for DAL A).
- Criteria 2: A tool that automates verification process(es) and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of:
    - verification process(es) other than that automated by the tool (TQL-4 for DAL A),
    - or development process(es) which could have an impact on the resulting software (TQL-4 for DAL A)
- Criteria 3: A tool that, within the scope of its intended use, could fail to detect an error (TQL-5 for DAL A).
- Still no proof of error absence category (might be TQL-2 for DAL A).

DO-178C/ED-12C standards: Qualification

- Introduce detailed Tool Qualification Level (1 downto 5)
- Criteria 1: A tool whose output is part of the resulting software and thus could insert an error (TQL-1 for DAL A).
- Criteria 2: A tool that automates verification process(es) and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of:
  - verification process(es) other than that automated by the tool (TQL-4 for DAL A),
  - or development process(es) which could have an impact on the resulting software (TQL-4 for DAL A)
- Criteria 3: A tool that, within the scope of its intended use, could fail to detect an error (TQL-5 for DAL A).
- Still no proof of error absence category (might be TQL-2 for DAL A).

DO-178C/ED-12C standards: Qualification

- Introduce detailed Tool Qualification Level (1 downto 5)
- Criteria 1: A tool whose output is part of the resulting software and thus could insert an error (TQL-1 for DAL A).
- Criteria 2: A tool that automates verification process(es) and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of:
  - verification process(es) other than that automated by the tool (TQL-4 for DAL A),
  - or development process(es) which could have an impact on the resulting software (TQL-4 for DAL A)
- Criteria 3: A tool that, within the scope of its intended use, could fail to detect an error (TQL-5 for DAL A).
- Still no proof of error absence category (might be TQL-2 for DAL A).

DO-178C/ED-12C standards: Qualification

- Introduce detailed Tool Qualification Level (1 downto 5)
- Criteria 1: A tool whose output is part of the resulting software and thus could insert an error (TQL-1 for DAL A).
- Criteria 2: A tool that automates verification process(es) and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of:
    - verification process(es) other than that automated by the tool (TQL-4 for DAL A),
    - or development process(es) which could have an impact on the resulting software (TQL-4 for DAL A)
- Criteria 3: A tool that, within the scope of its intended use, could fail to detect an error (TQL-5 for DAL A).
- Still no proof of error absence category (might be TQL-2 for DAL A).

Common documents

- Phase 1: Cooperative process definition:
  - Plan for software aspects of certification (PSAC)
  - Development plan (SDP)
  - Verification plan (SVP)
  - Configuration management plan (SCMP)
  - Quality assurance plan (SQAP)
  - Tool qualification plan

Common documents (qualification case)

- Phase 2: Process application verification
  - User requirements
  - Tool architecture (elementary tools and their assembly)
  - Tool requirements: Can be refined user requirements or derived requirements (linked to technology choices, should be avoided or strongly justified)
  - Development and verification results (each elementary tools)
  - Traceability links
  - Verification results (user level)

Some comments

- Standards were designed for systems not tools:
  Adaptation required
- MCDC not mandatory for tools,
  but similar arguments might be required
- Traceability of all artefacts in the development, relate requirements, design and implementation choices
- Purpose is to provide confidence
- Both cooperative and coercive approach
- Any verification technology can be used,
  from proofreading to automatic proof
  if confidence is given
- Choose the strategy and technologies that will best reduce risks

Some comments

- Standards were designed for systems not tools:
  Adaptation required
- MCDC not mandatory for tools,
  but similar arguments might be required
- Traceability of all artefacts in the development, relate requirements, design and implementation choices
- Purpose is to provide confidence
- Both cooperative and coercive approach
- Any verification technology can be used,
  from proofreading to automatic proof
  if confidence is given
- Choose the strategy and technologies that will best reduce risks

Some comments II

- Must be applied as soon as possible (cost reduction)
- Small is beautiful (simplicity is the key)
- Certification authorities need to understand the technologies
- Cross-experiments are mandatory (classical w.r.t. formal methods)

Some comments II

- Must be applied as soon as possible (cost reduction)
- Small is beautiful (simplicity is the key)
- Certification authorities need to understand the technologies
- Cross-experiments are mandatory (classical w.r.t. formal methods)

Plan

1. Safe MDE concerns
2. Certification and Qualification
3. Application to Code generation tools
4. Application to Static analysis tools
5. The Executable DSML metamodeling pattern

Transformation verification technologies

- Verification subject:
  - Transformation: done once, no verification at use, white box, very high cost
  - Transformation application: done at each use, black box, easier, complex error management
- Classical technologies:
  - Document independant proofreading (requirements, specification, implementation)
  - Test
    - Unit, Integration, Functional, Deployment level
    - Requirement based test coverage
    - Source code test coverage
    - Structural coverage, Decision coverage, Multiple Condition Decision Coverage (MCDC)

Transformation verification technologies II

- Formal technologies (require formal specification):
  - Automated test generation
  - Model checking (abstraction of the system)
  - Static analysis (abstraction of the language)
  - Automated proof
  - Assisted (human in the loop) proof
- Transformation case
  - Transformation specification: Structural/Behavioral
  - Proof of transformation correctness
  - Links with certification/qualification

Classical development and verification process

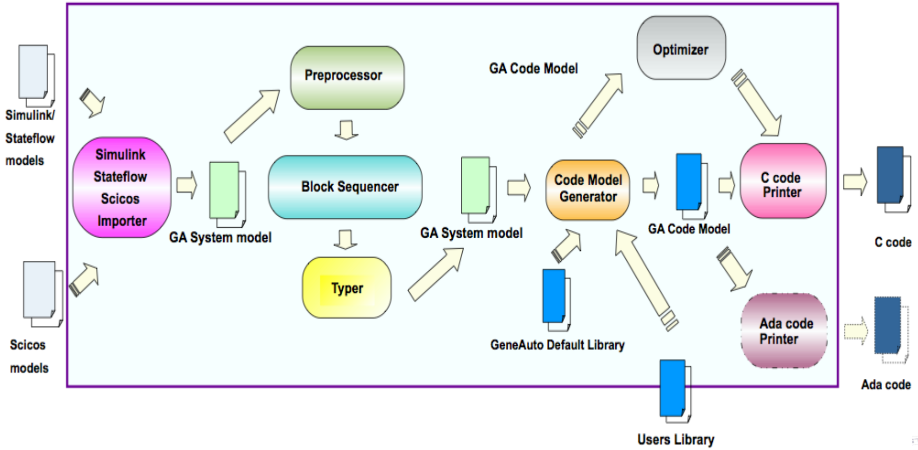
- Tool development, verification and qualification plans
- User requirements
- Tool requirements (human proofreading)
- Test plan (requirements based coverage, code coverage verification)
- Implementation and test application

GeneAuto experiment: Proof assistant based

- Derived from the classical process, validated by french certification bodies
- Formal specification using Coq of tool requirements, implementation and correctness
- Proofreading verification of requirements specification
- Automated verification of specification correctness
- Extraction of OCaML source implementation
- Proofreading verification of extracted OCaML source
- Integration of OCaML implementation with Java/XML implementation (communication through simple text files with regular grammars)
- Proofreading verification of OCaML/Java wrappers (simple regular grammar parsing)
- Test-based verification of user requirements conformance

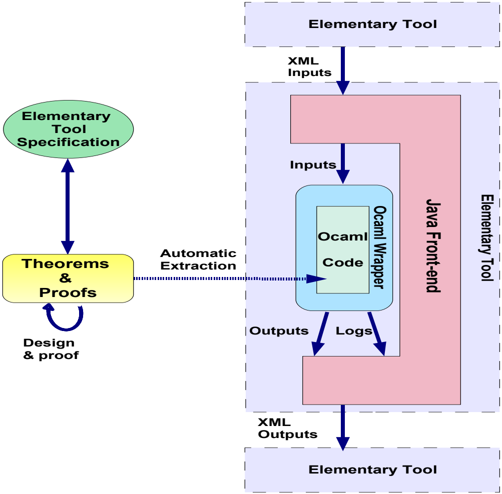
GeneAuto Code Generator Architecture

Split into independent modules (easier V & V and qualification)



ACADIE

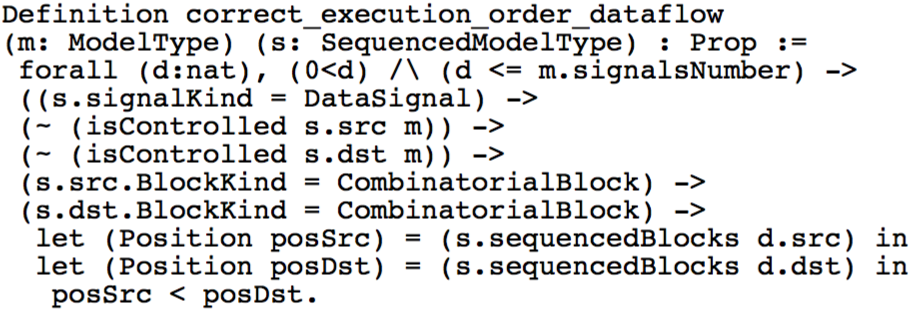
Integration



An example: User requirements R-CG-040

- F6 – Determine execution order of functional model blocks
  The execution order generated by the ACG must be as close as possible to that in Simulink and it shall be possible to visualise the execution order
  Same scheduling as Simulink is required to ensure that generated code conforms to Simulink simulations.
- Refinement
  - F6.1 Sort blocks based on data-flow constraints
  - F6.2 Refine the order according to control flow constraints
  - F6.3 Sort blocks with partial ordering according to priority from the input model.
  - F6.4 Sort blocks that are still partially ordered according to their graphical position in the input model

The same one in Coq



```
Definition correct_execution_order_dataflow
(m: ModelType) (s: SequencedModelType) : Prop :=
 forall (d:nat), (0<d) /\ (d <= m.signalsNumber) ->
 ((s.signalKind = DataSignal) ->
 (~ (isControlled s.src m)) ->
 (~ (isControlled s.dst m)) ->
 (s.src.BlockKind = CombinatorialBlock) ->
 (s.dst.BlockKind = CombinatorialBlock) ->
  let (Position posSrc) = (s.sequencedBlocks d.src) in
  let (Position posDst) = (s.sequencedBlocks d.dst) in
   posSrc < posDst.
```

Open questions ?

What are:

- User requirement for a transformation/verification ?
- Tool requirement for a transformation/verification ?
- Formal specification for a transformation/verification ?
- Test coverage for a transformation/verification ?
- Test oracle for a transformation/verification ?
- Qualification constraint for transformation/verification languages ?
- Best strategy for tool verification (once vs at each use) ?

GeneAuto feedbacks

- From the certification perspective: Very good but...
  - Still some work on qualification of the proof assistant tools
    - Proof verifier
    - Program extractor
  - Complex management of input/output
- From the developer perspective:
  - High dependence to the technologies
  - Very high cost to use the technology
  - Not easy to subcontract
  - Scalability not ensured
  - Bad separation between semantics-based verification and requirements-based specification
  - Hard to assess development time
- On the use of Java: How to provide confidence in the libraries ?

Going further: CompCert use experiment

- CompCert: C to PowerPC optimising code generator developed at INRIA by Xavier Leroy
- Ricardo Bedin-França PhD thesis with Airbus (advisor Marc Pantel): Improve certified code efficiency
  - Metrics: WCET, Code and memory size, Cache and memory accesses
  - Improvements of the various phases from models to embedded binary code
  - New verification process using formal methods
  - First CompCert experiments: -12% WCET, -25% code size, -72% cache read, -65% cadre write
  - Design of a CompCert dedicated verification process
  - Feed static analysis results (Astrée, frama-C) from C to binary through CompCert (improve WCET precision)
  - Improve SCADE block scheduling to reduce memory accesses (signal liveness)
  - Design of a whole development cycle verification process with tools qualification

Going further: CompCert use experiment

- CompCert: C to PowerPC optimising code generator developed at INRIA by Xavier Leroy
- Ricardo Bedin-França PhD thesis with Airbus (advisor Marc Pantel): Improve certified code efficiency
  - Metrics: WCET, Code and memory size, Cache and memory accesses
  - Improvements of the various phases from models to embedded binary code
  - New verification process using formal methods
  - First CompCert experiments: -12% WCET, -25% code size, -72% cache read, -65% cadre write
  - Design of a CompCert dedicated verification process
  - Feed static analysis results (Astrée, frama-C) from C to binary through CompCert (improve WCET precision)
  - Improve SCADE block scheduling to reduce memory accesses (signal liveness)
  - Design of a whole development cycle verification process with tools qualification

Proposal: Mixed approach

- Separate specification verification from implementation verification
- Define explicitly semantics link metamodel (relation between source and target)
- Specify transformation as properties of the links
- Implementation verification (mostly syntactic/static semantics)
  - Implementation must generate both target and links
  - Implementation verification checks properties on generated links links
- Specification verification: Prove the dynamic semantics equivalence between source and target in a trace link
- Rely on the specific of the operational semantics of the source and target languages
- Andres Toom PhD work (advisors : Tarmo Uustalu and Marc Pantel)

Proposal: Deductive approach

- Another kind of separation between specification and implementation verification
- Rely on Hoare logic kind of axiomatic semantics
- Specify the different construct of the language using pre/post conditions, invariants and variants
- Generate the code and the corresponding assertions
- Use deductive static analysers like frama-C to prove the correctness
- Use different kind of logics depending on the correction criteria
- Verify the correctness of the Hoare specification with respect to the operational semantics
- Might also rely on the previous links to ease the proof
- Soon to be started Arnaud Dieumegard PhD work with Airbus (advisor : Marc Pantel)

Early feedbacks

- Separation of concerns:
  - Industrial partners: Specification, Implementation, Implementation verification (mainly syntactic)
  - Academic partners: Specification verification (semantics)
- Very good subcontracting capabilities
- Almost no technology constraints on the industrial partner (classical technologies)
- Good scalability
- Easy to analyse syntactic error reports
- Enables to modify generated code and links
- Parallel work between syntactic and semantics concerns

Work in progress

- Positive first experiments on simple use cases from GeneAuto
- But requires some grayboxing (expose parts of the internals)
  - Flattening of statecharts
  - Either very complex specification (doing the flattening)
  - Or express the fixpoint nature of implementation (in the specification)
- Require full scale experiments
- Require exchange with certification authorities
- Require qualified syntactic verification tool (OCL-like, but simpler)
- Require explicit relations between syntactic and semantics work
- Require explicit description of semantics in metamodels

Plan

1. Safe MDE concerns
2. Certification and Qualification
3. Application to Code generation tools
4. **Application to Static analysis tools**
5. The Executable DSML metamodeling pattern

Static analysis tools

- Several kind of tools
  - Qualitative and quantitative properties
  - Fixed or user defined properties
  - Semantic abstraction or Proof technologies
- Common aspects: Common pre-qualification
  - Product (source of binary code) reader: fully common ?
  - Configuration (properties, . . .) reader: partly common
  - Result writer and browser: partly common ?
- Split the verification tool in a sequence of elementary activities
  - Common ones (pre-qualification could be shared)
  - Technology specific ones
  - Easier to specify, to validate and to verify
  - Can be physical or virtual (produce intermediate results even in a single tool)

Required activities

- Specify user requirements
- Specify tool architecture (elementary tools and their assembly)
- Specify tool level requirements (elementary tools and their assembly)
- Specify functional test cases and results
- Choose verification strategy:
    - Tool verification or Result verification
    - Integration and unit tests (eventually with test generators and oracles)
    - Proof reading of tool source or test results
    - Formal verification of the verification tool itself (i.e. Coq in Coq, Compcert in Coq, . . .)

Abstraction kind

- Translate to non standard semantics
- Compute recursive equations
- Compute fixpoint of equations
  - Fixpoint algorithm
  - Abstract domains and operators
  - Widening, narrowing
- Check that properties are satisfied on the abstract values
- Produce user friendly feedback (related to product and its standard semantics)

Deductive kind

- Produce proof obligations (weakest precondition, verification condition, . . .)
- Check the satisfaction of proof obligations
  - Proof term rewriting to simpler language
  - Split to different sub-languages (pure logic, arithmetic, . . .)
  - Apply heuristics to produce a proof term
  - Check the correctness of the proof term
  - Produce failure feedback or proof certificate (related to product and its standard semantics)
- Produce user friendly feedback

Potential strategy: Common parts

- Build “semantics”-related trace links during transformations
  Helps in verification of results w.r.t. parameters
- Reader and writer:
  - Cross-reading
  - Introduce dual reader/writer: check composition is identity
  - Asymmetric implementation: Several independent implementations and results comparison
- Code generation and transformation can be formally specified and verified:
  - Formal tool requirements: foreach source construct, what are the generated targets and the links with the source
  - Syntactic verification: properties of the trace links given as tool requirements
  - Semantic verification: validation of the technology
- User-friendly feedback: Code generation based on trace links

Potential strategy: Common parts

- Build “semantics”-related trace links during transformations
  Helps in verification of results w.r.t. parameters
- Reader and writer:
  - Cross-reading
  - Introduce dual reader/writer: check composition is identity
  - Asymmetric implementation: Several independent implementations and results comparison
- Code generation and transformation can be formally specified and verified:
  - Formal tool requirements: foreach source construct, what are the generated targets and the links with the source
  - Syntactic verification: properties of the trace links given as tool requirements
  - Semantic verification: validation of the technology
- User-friendly feedback: Code generation based on trace links

Potential strategy: Common parts

- Build “semantics”-related trace links during transformations
  Helps in verification of results w.r.t. parameters
- Reader and writer:
  - Cross-reading
  - Introduce dual reader/writer: check composition is identity
  - Asymmetric implementation: Several independent implementations and results comparison
- Code generation and transformation can be formally specified and verified:
  - Formal tool requirements: foreach source construct, what are the generated targets and the links with the source
  - Syntactic verification: properties of the trace links given as tool requirements
  - Semantic verification: validation of the technology
- User-friendly feedback: Code generation based on trace links

ACADIE

Potential strategy: Common parts

- Build “semantics”-related trace links during transformations
  Helps in verification of results w.r.t. parameters
- Reader and writer:
  - Cross-reading
  - Introduce dual reader/writer: check composition is identity
  - Asymmetric implementation: Several independent implementations and results comparison
- Code generation and transformation can be formally specified and verified:
  - Formal tool requirements: foreach source construct, what are the generated targets and the links with the source
  - Syntactic verification: properties of the trace links given as tool requirements
  - Semantic verification: validation of the technology
- User-friendly feedback: Code generation based on trace links

Potential strategy: Abstraction kind

- Non-standard semantics and recursive equation production are similar to code generation
  - Semantic verification: monotony at the equations-level
  - Semantic verification: soundness of the abstraction
- No verification on the fixpoint computation
  - Verification of the result (if least solution is not required)
  - A qualified (much simpler) verification tool is then required
- Verification of the properties of the abstract domains (join, meet, operators, $\alpha \circ \gamma$, widening, narrowing, monotony, . . .)
  - Proof reading
  - Automated test generation with oracles
  - Formal specification and proof
- Property checks (based on abstract property generation)
  - Related to code generation
  - Semantic verification: soundness of the abstraction

Potential strategy: Abstraction kind

- Non-standard semantics and recursive equation production are similar to code generation
    - Semantic verification: monotony at the equations-level
    - Semantic verification: soundness of the abstraction
- No verification on the fixpoint computation
    - Verification of the result (if least solution is not required)
    - A qualified (much simpler) verification tool is then required
- Verification of the properties of the abstract domains (join, meet, operators, $\alpha \circ \gamma$, widening, narrowing, monotony, ...)
    - Proof reading
    - Automated test generation with oracles
    - Formal specification and proof
- Property checks (based on abstract property generation)
    - Related to code generation
    - Semantic verification: soundness of the abstraction

ACADIE

Potential strategy: Abstraction kind

- Non-standard semantics and recursive equation production are similar to code generation
  - Semantic verification: monotony at the equations-level
  - Semantic verification: soundness of the abstraction
- No verification on the fixpoint computation
  - Verification of the result (if least solution is not required)
  - A qualified (much simpler) verification tool is then required
- Verification of the properties of the abstract domains (join, meet, operators, $\alpha \circ \gamma$, widening, narrowing, monotony, . . .)
  - Proof reading
  - Automated test generation with oracles
  - Formal specification and proof
- Property checks (based on abstract property generation)
  - Related to code generation
  - Semantic verification: soundness of the abstraction

Potential strategy: Abstraction kind

- Non-standard semantics and recursive equation production are similar to code generation
  - Semantic verification: monotony at the equations-level
  - Semantic verification: soundness of the abstraction
- No verification on the fixpoint computation
  - Verification of the result (if least solution is not required)
  - A qualified (much simpler) verification tool is then required
- Verification of the properties of the abstract domains (join, meet, operators, $\alpha \circ \gamma$, widening, narrowing, monotony, . . .)
  - Proof reading
  - Automated test generation with oracles
  - Formal specification and proof
- Property checks (based on abstract property generation)
  - Related to code generation
  - Semantic verification: soundness of the abstraction

Potential strategy: Deductive kind

- Proof obligation computation is a kind of code generation
    - Semantic verification: correctness of the axiomatic semantics
- Satisfaction of the proof obligations:
    - No verification on proof certificate generation
    - Verification of the certificate itself (much simpler than some heuristic-based automatic prover)
    - Term rewriting can be considered as code generation (endogenous)
    - Curry-Howard type checking can be verified in a similar way
    - Rely on Coq In Coq, Isabelle in Isabelle, . . .

What about validation of the technologies ?

- Mainly scientific work and a lot of publications
- Brings confidence but paperwork is not enough
- Mechanized is better but still not enough
- Functional user level tests still mandatory currently
- Mixed system verification experiments (both tests and static analysis)
- Reverse analysis of existing systems

Synthesis

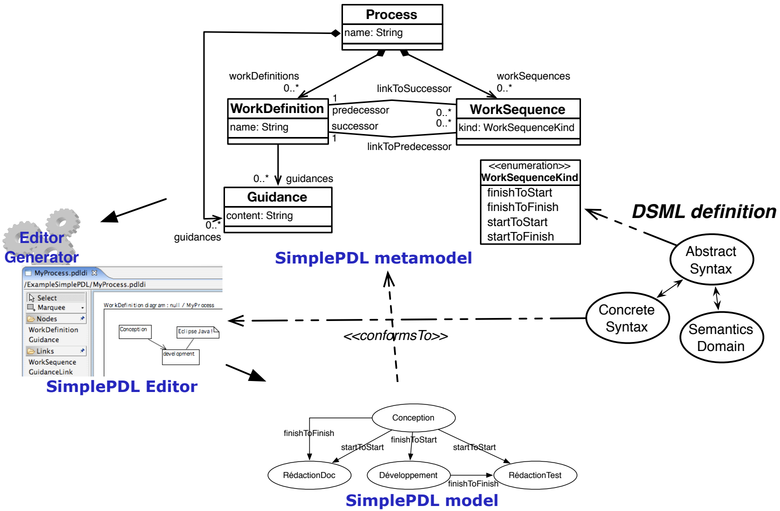
- Technical exchange with certification authorities mandatory
- Cross experiments and reverse engineering experiments mandatory
- Verification strategy must be designed early to choose the right architecture and trace information
- Semi-formal (even formal) requirements must be written as soon as possible

Plan

1. Safe MDE concerns
2. Certification and Qualification
3. Application to Code generation tools
4. Application to Static analysis tools
5. The Executable DSML metamodeling pattern

Adding a new DSML to the TOPCASED platform

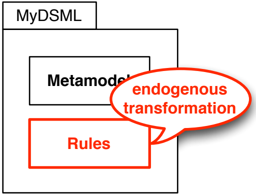
Classical MDE technologies

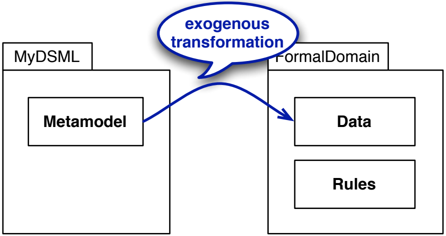


Needs for an execution semantics

- What about the dynamic semantics of a DSML?
- Needs for **model animation**
  - Does the model behave as expected?
- Needs for **model verification**
  - Does some property hold on a model?

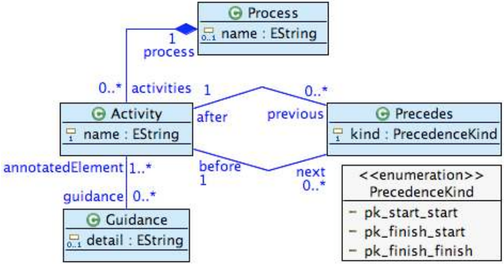
Two main techniques to express behavioral semantics:





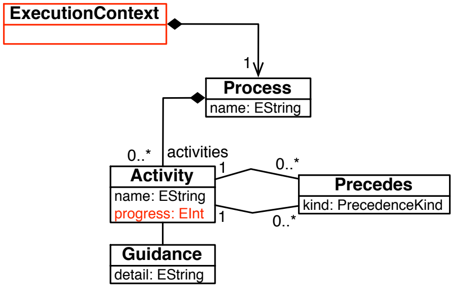
Metamodel Extensions

Basic meta-model



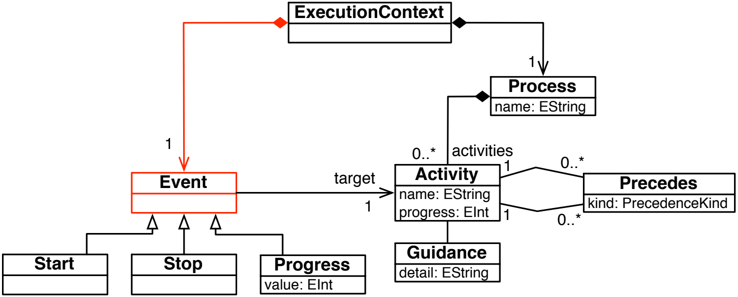
Metamodel Extensions

Capture execution state



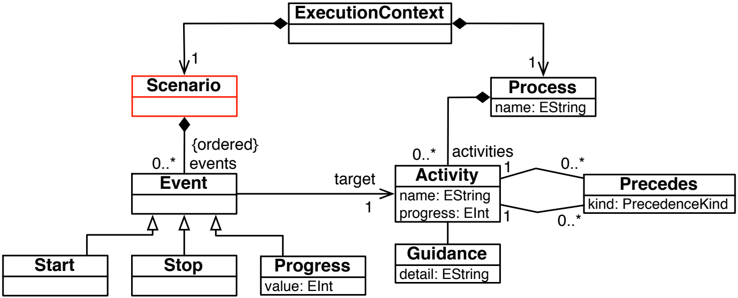
Metamodel Extensions

Scenario model definition

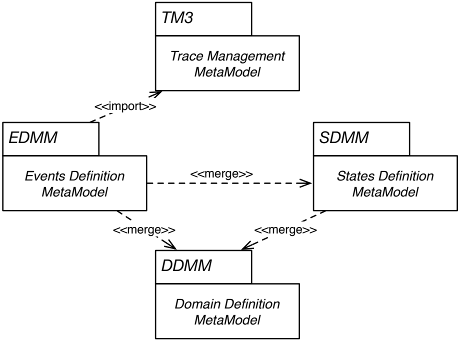


Metamodel Extensions

Scenario model definition

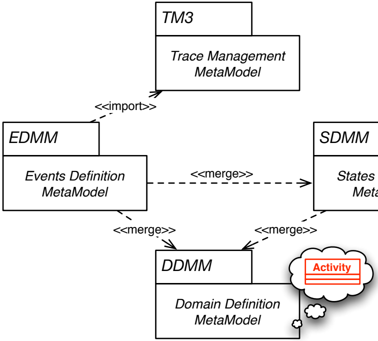


Architecture for an executable DSML



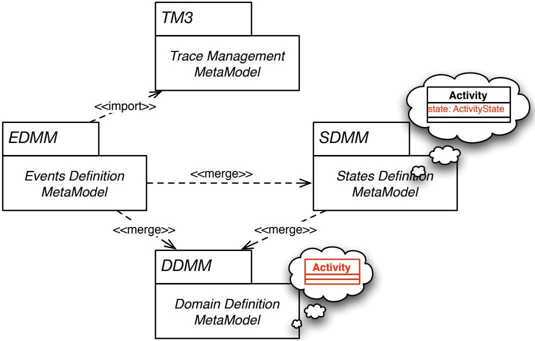
Composed of 4 metamodels

Architecture for an executable DSML



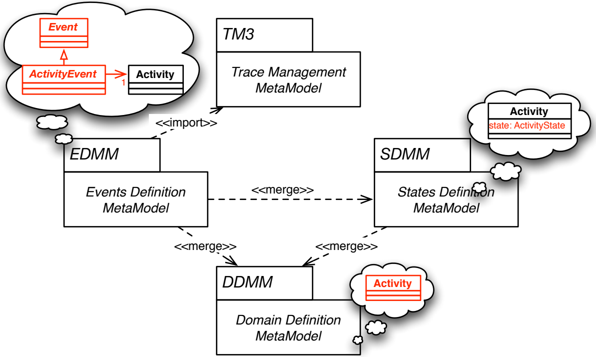
Domain Definition MetaModel:
“classical” metamodel

Architecture for an executable DSML



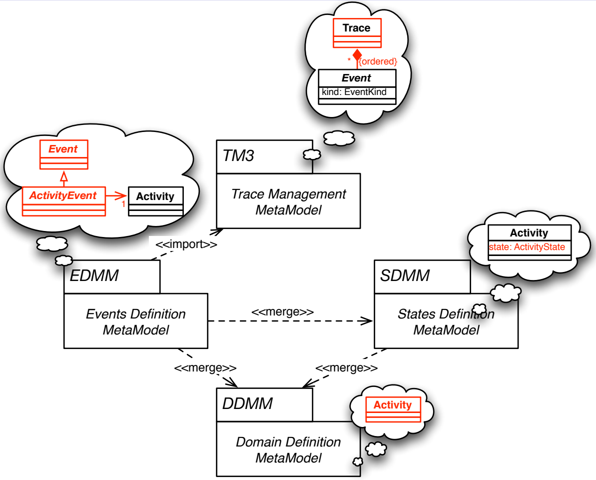
States Definition MetaModel:
runtime information

Architecture for an executable DSML



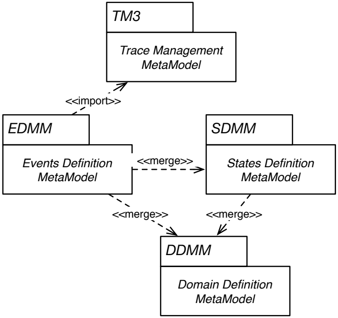
Events Definition MetaModel:
events inducing changes on SDMM (might be virtual)

Architecture for an executable DSML

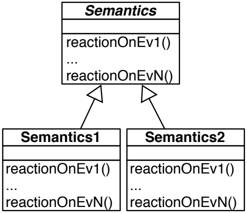


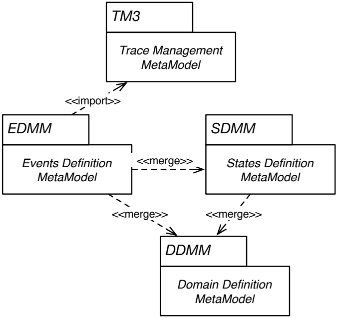
Trace Management MetaModel:
DSML independent MM for scenarios and traces

Main principles for model simulation

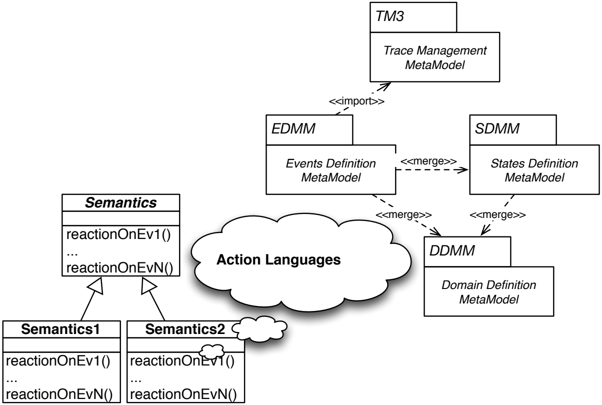


Main principles for model simulation

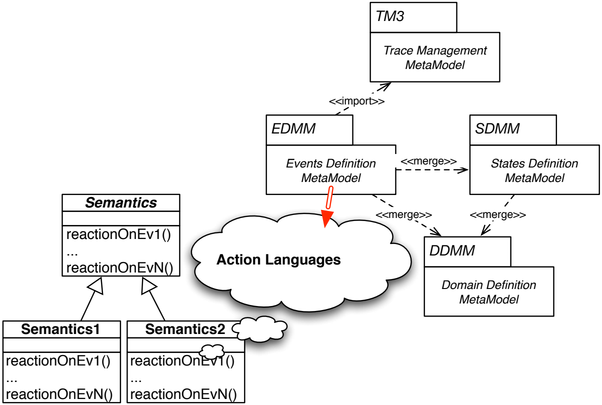




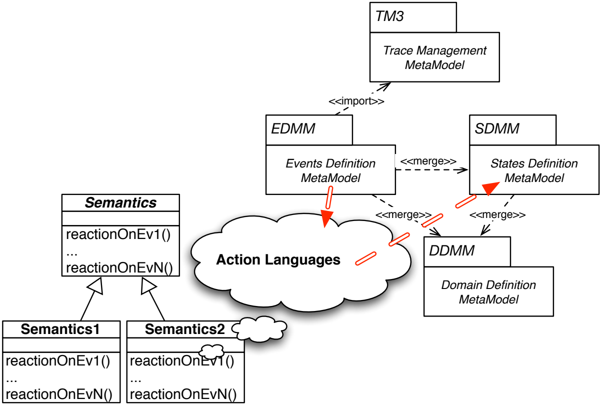
Main principles for model simulation



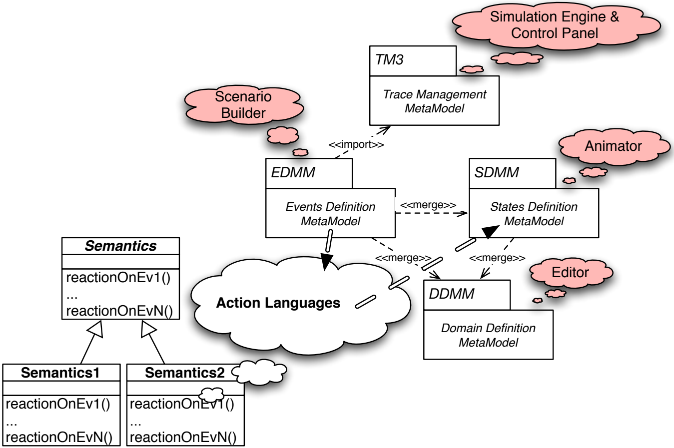
Main principles for model simulation



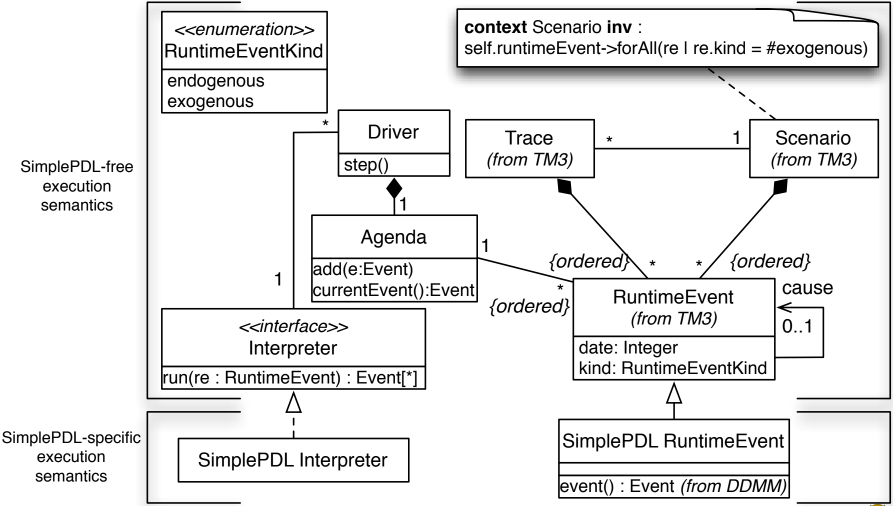
Main principles for model simulation



Main principles for model simulation

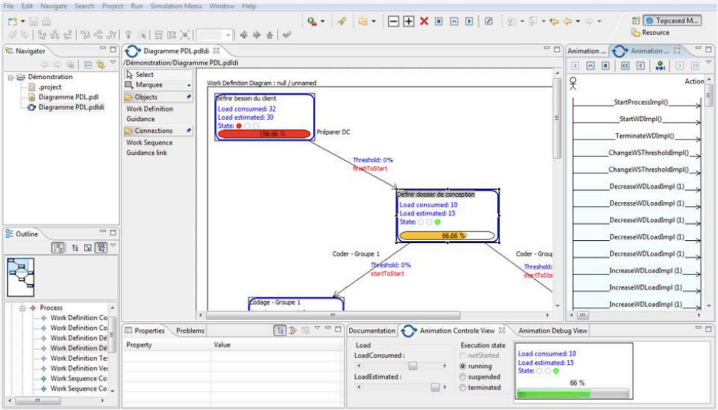


Architecture of TOPCASED Animators

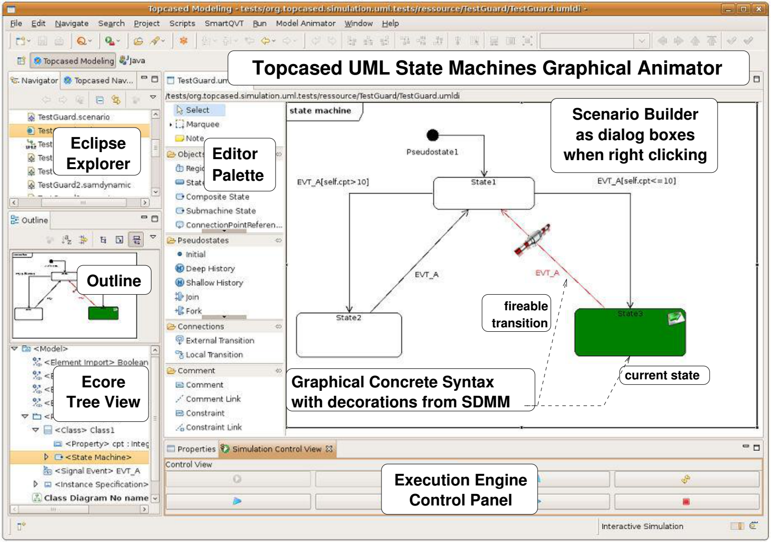


context Scenario **inv** :
self.runtimeEvent->forAll(re | re.kind = #exogenous)

SIMPLEPDL Simulator



UML2 StateChart Simulator (TOPCASED 2)



Topcased UML State Machines Graphical Animator

Eclipse Explorer

Editor Palette

Outline

Ecore Tree View

Scenario Builder as dialog boxes when right clicking

fireable transition

current state

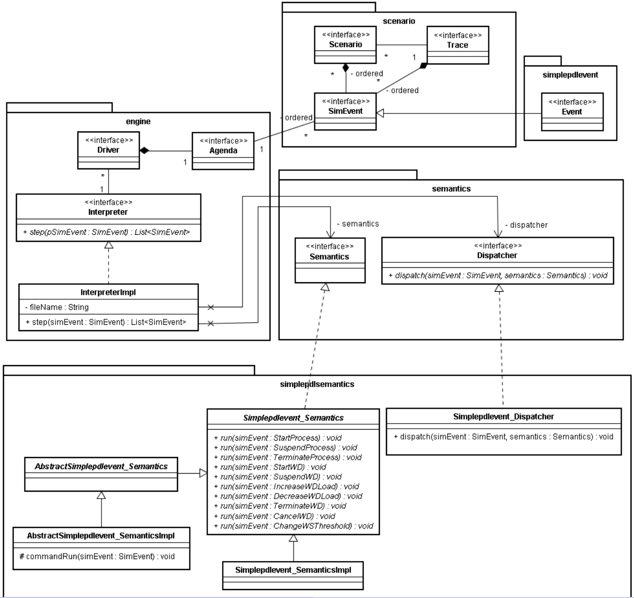
Graphical Concrete Syntax with decorations from SDMM

Execution Engine Control Panel

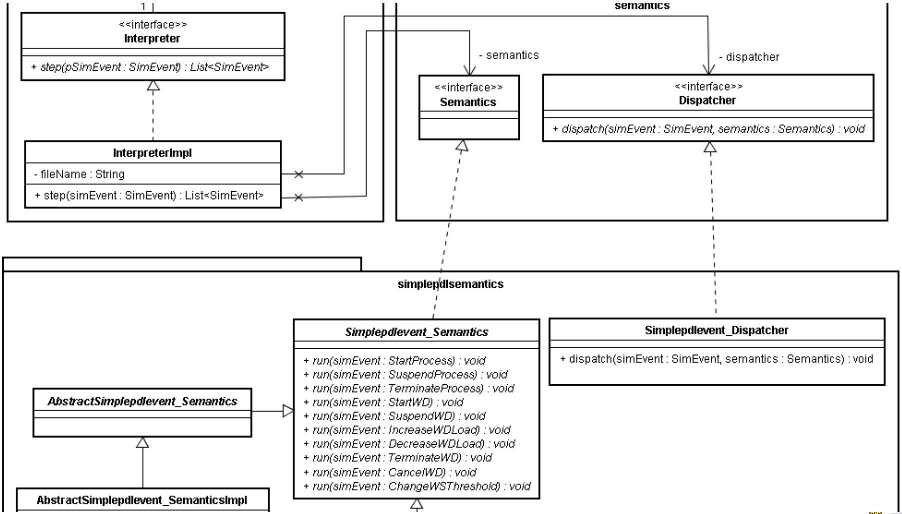
Multiple Semantics Definition

- Defining a model animator implies to:
    - implement the Interpreter interface and define the run method.
    - test the Event argument to run the right reaction
      $\Longrightarrow$ error prone (events may be missed)
- **Solution:** Apply the *Visitor* pattern
  Visitor interface and a dispatch method are generated from the EDMM
- **Benefits:** eases the definition a related semantics
    - Commonalities may be grouped in an abstract superclass.
    - A new semantics may be defined as a specialization of an existing one.
- Visitor pattern would also be useful for the SDMM.
  But transformation languages such as ATL, SmartQVT or Kermeta achieve the same purpose through aspects.

Architecture of the generated code



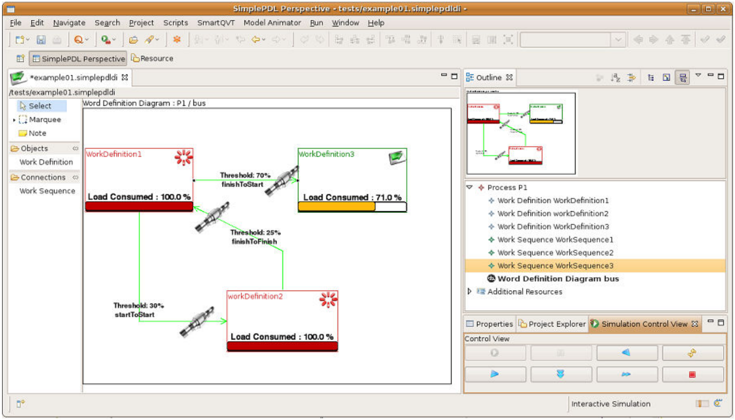
Architecture of the generated code



ACADIE

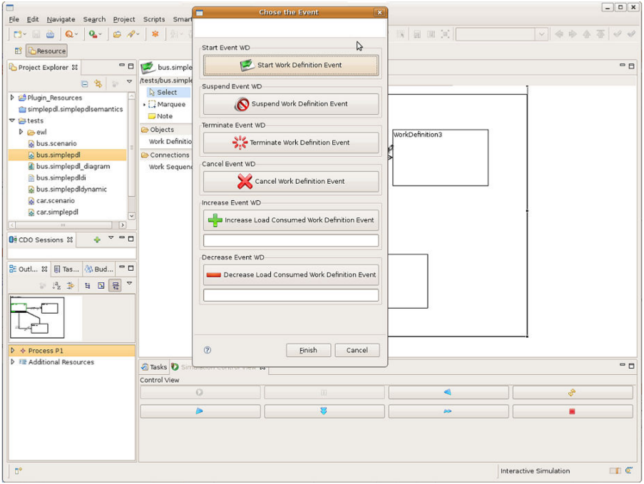
Improvement of the Model Graphical Visualization

- definition of GMF decorations on the editor graphical elements
- relying on EMF notifications to update graphical decorations



Controllers for Event Creation

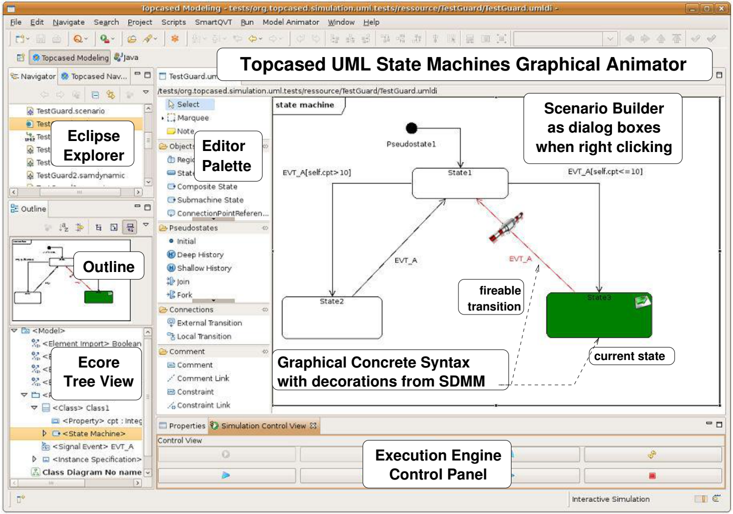
- automatic generation based on EDMM



Refactoring of existing TopCased Animators

The UML State Machines Animator

Half a day has been enough to existing TopCased animators (UML and SAM)



Topcased UML State Machines Graphical Animator

Eclipse Explorer

Editor Palette

Scenario Builder as dialog boxes when right clicking

Outline

fireable transition

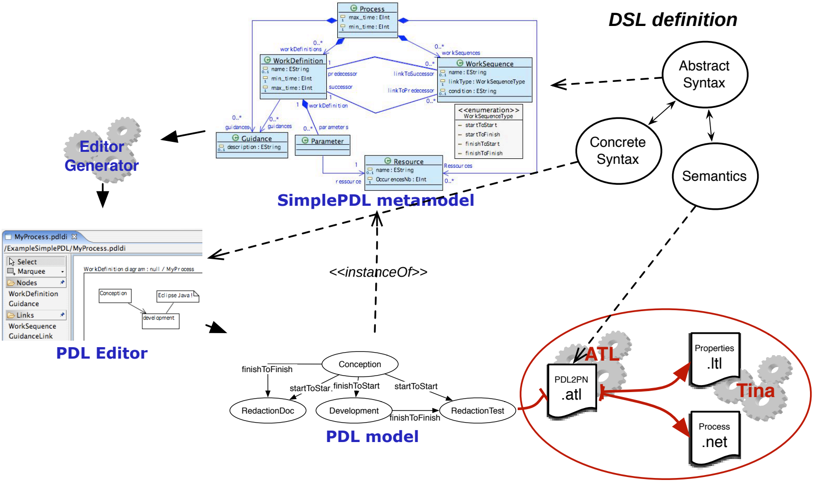
current state

Graphical Concrete Syntax with decorations from SDMM

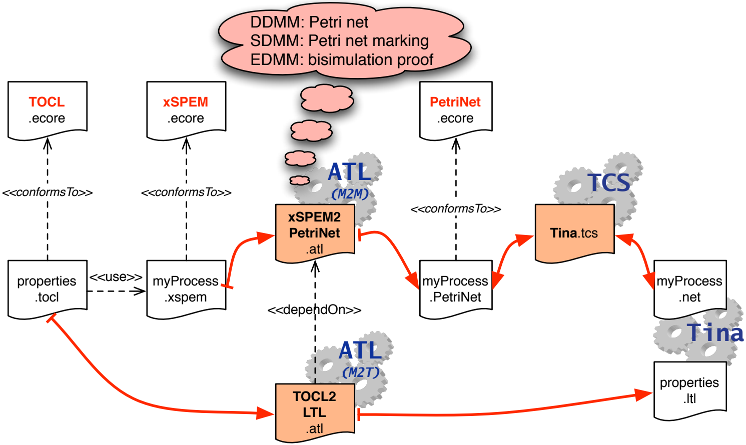
Ecore Tree View

Execution Engine Control Panel

TOPCASED proposal (through case study)



Principles applied to SimplePDL using Petri nets


DDMM: Petri net
SDMM: Petri net marking
EDMM: bisimulation proof

What do we want to check ?

- resource constraints
    - computers
    - manpower
- timing constraints
    - minimum achievement time
    - maximum achievement time
- causality constraints
    - startToStart
    - startToFinish
    - finishToStart
    - finishToFinish
- . . .
- for some execution
- or for all executions

What do we want to check ?

- resource constraints
  - computers
  - manpower
- timing constraints
  - minimum achievement time
  - maximum achievement time
- causality constraints
  - startToStart
  - startToFinish
  - finishToStart
  - finishToFinish
- . . .
- for some execution
- or for all executions

Some SimplePDL-expert properties

For all executions

- every WD must start and then finish
- once a WD is finished, it remains so
- resource and causality constraints must hold

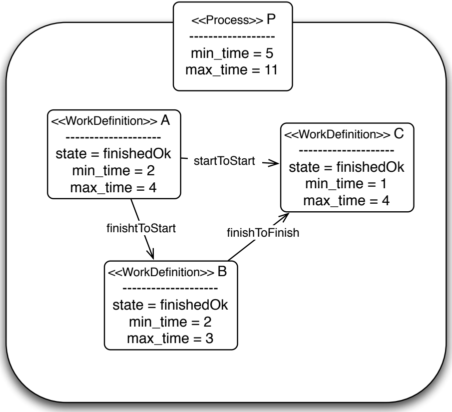
For some execution

- every WD must take between *min* and *max* time units to complete
- the overall process is able to finish

A sample run

Illustrating operational semantics

- $t = 0$: *WD*s are `notStarted`
- $t = 1$: *A* starts
- $t = 3$: *B* starts
- $t = 4$: *A* completes
- $t = 5$: *C* starts
- $t = 7$: *B* completes
- $t = 8$: *C* completes



The Temporal Object Contraint Language

TOCL (Gogolla & al., 2002) embeds

- the Object Constraint Language for spatial relations
- the Linear Temporal Logic for time relations

TOCL is used

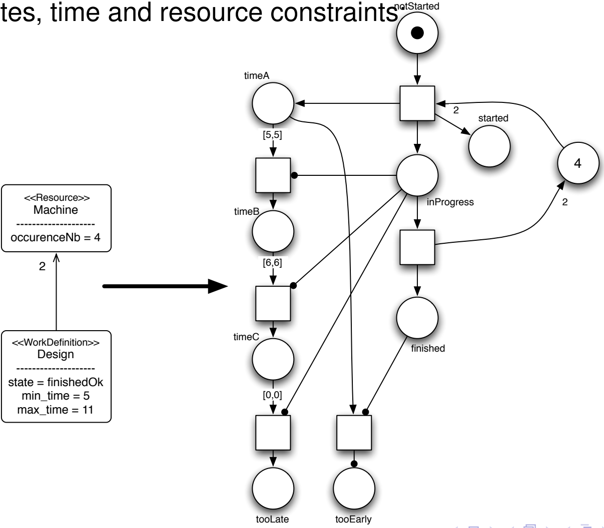
- to express fine behavioral spec (*next*, *existsNext*, *always*, *sometime*, . . .)
- about some execution or all executions

Some properties of WD alone

- $\forall w, (w.state = \texttt{notStarted} \wedge \textit{sometime}\ w.state = \texttt{inProgress})$
- $\forall w, \textit{always}\ (w.state = \texttt{inProgress} \Rightarrow \textit{sometime}\ w.state \in \{\texttt{finishedOk}, \texttt{tooEarly}, \texttt{tooLate}\})$
- $\forall w, \textit{always}\ (w.state = \texttt{finishedOk} \Rightarrow \textit{always}\ w.state = \texttt{finishedOk})$
- $\neg\exists w, \textit{always}\ w.state \neq \texttt{finishedOk}$

Expressing WorkDefinition Semantics through Petri Nets

Encoding states, time and resource constraints



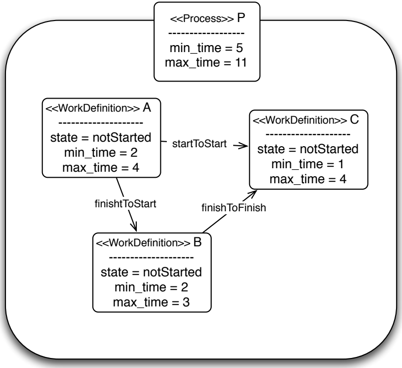
<<Resource>>
Machine
----
occurenceNb = 4

2

<<WorkDefinition>>
Design
----
state = finishedOk
min_time = 5
max_time = 11

Expressing WorkDefinition Semantics through Petri Nets

Finally, we add causality constraints:



<<Process>> P
----
min_time = 5
max_time = 11

<<WorkDefinition>> A
----
state = notStarted
min_time = 2
max_time = 4

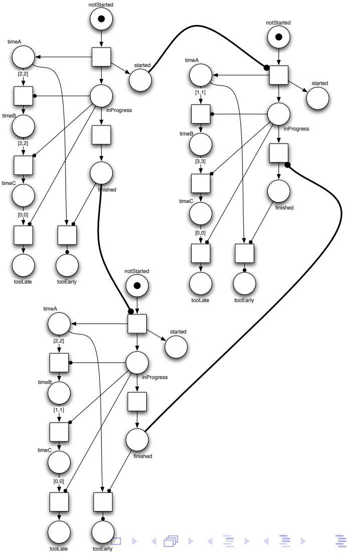
<<WorkDefinition>> C
----
state = notStarted
min_time = 1
max_time = 4

<<WorkDefinition>> B
----
state = notStarted
min_time = 2
max_time = 3

startToStart

finishtToStart

finishToFinish

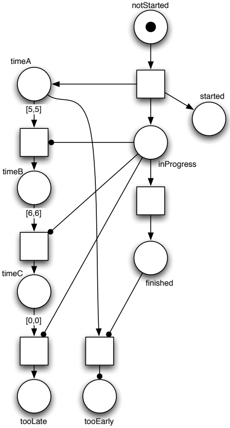


A sample run

Translation into Petri nets

A WD with *min_time* = 5 and *max_time* = 11 time units

- $t = 0$: *WD* is `notStarted`

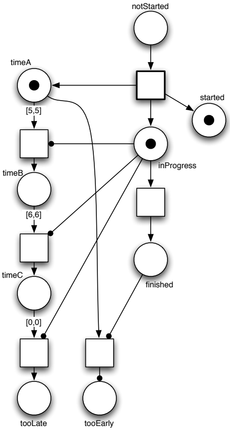


A sample run

Translation into Petri nets

A WD with *min_time* = 5 and *max_time* = 11 time units

- $t = 0$: *WD* is `notStarted`
- $t = 1$: *WD* starts

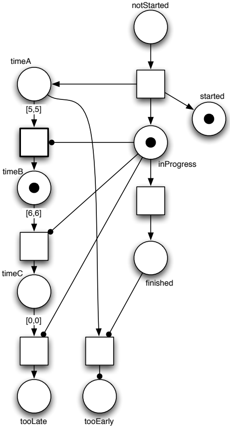


A sample run

Translation into Petri nets

A WD with *min_time* $= 5$ and *max_time* $= 11$ time units

- $t = 0$: *WD* is `notStarted`
- $t = 1$: *WD* starts
- $t = 6$: *WD* is now on time

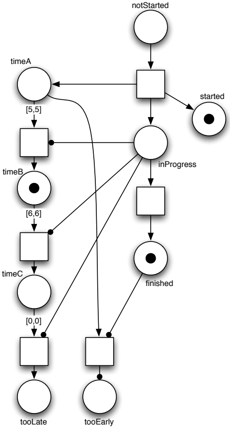


A sample run

Translation into Petri nets

A WD with *min_time* $= 5$ and *max_time* $= 11$ time units

- $t = 0$: *WD* is `notStarted`
- $t = 1$: *WD* starts
- $t = 6$: *WD* is now on time
- $t = 7$: *WD* completes on time



Some features of our translation

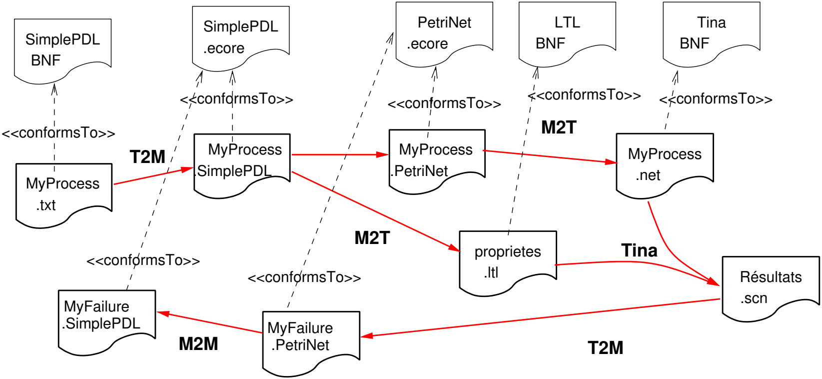
Nice properties

- functional pattern-matching ATL program
- structural (a WD is a net & a WD.state is a marking)
- modular (a constraint is also a net)
- incremental (a constraint may be plugged in & out)
- traceable

Target language comes equipped: `http://www.laas.fr/tina/`

- `nd` (*NetDraw*) : editor and simulator of temporal Petri nets
- `tina` : scanner of temporal Petri nets state spaces
- `selt`: model-checker for the temporal logic $SE-LTL$ (State/Event *LTL*), with counter-example generation

Global scheme



Formal expression with TOCL

```
context Process inv:
   sometime activities->forall(a | a.state = #finished);
```

More abstract expression

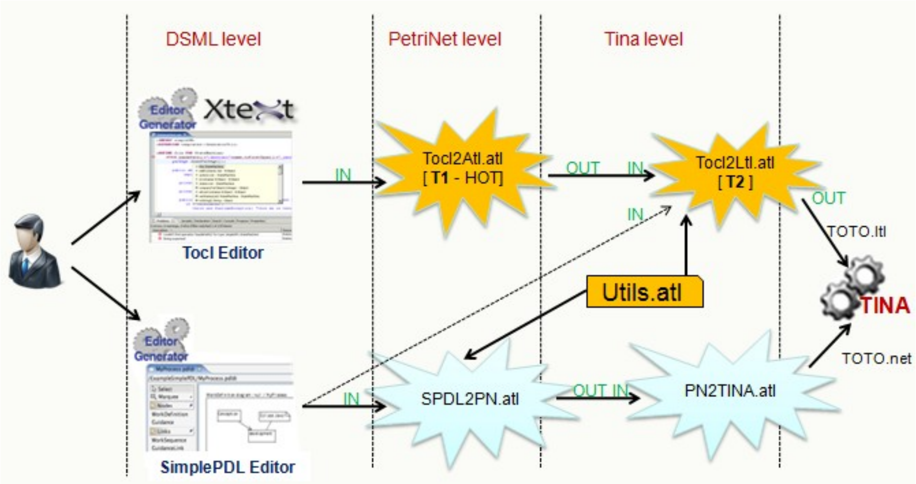
```
context Process inv:
   sometime activities->forall(a | a.isFinished());
```

Consequences

- In the semantics DSML extensions, think *query* more than *state*
- Define an ATL module gathering the methods (helpers) that defines :
  - the names given to places and transitions (∗_started, A_start, etc.)
  - the implantation of the queries related to the encoding in the formal language

ACADIE

Automatic transformation of TOCL to LTL



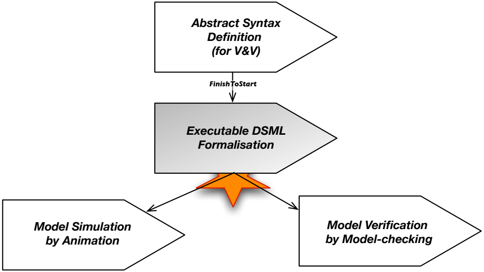
Property driven approach

1. Identify the properties of interest for the user
   (that allows to answer the questions he is asking)
2. Specify the **minimal** execution semantics
   using a translation to a formal language
3. Propose a property description language: Temporal OCL
   Properties expressed on the extended DSML (requests and events)
4. Implement a translational semantics by making concrete choices
   and provide the requests
5. Translate **automatically** the properties to the target language
6. Use the model checking tools on the target technical space
7. Bring the results back to the DSML

General method for defining an executable DSML

1. Define the Abstract Syntax (using a Property-driven approach)
    1. Define the DDMM
    2. **List the properties of interest**
    3. Define the SDMM
    4. Define the EDMM
2. **Define the reference semantics**
3. Define an operational semantics for the simulator
4. Define a translational semantics for the verification
5. Ensure the **consistency of the different** semantics (bisimulation proofs)

Formal framework for metamodeling



- **Purpose:** Qualify V&V tools to facilitate certification.
- **Principle:** Formalize the reference behavioral semantics and then
  - ⇒ generate operational semantics (animators)
  - ⇒ validate translational semantics (verification)
- **Means:**
  - Formalization of MDE concepts (a first attempt based on Coq)
  - Definition of an endogenous transformation language (not yet done)

Conclusion

- **Formal Framework**
  - formalisation of EMOF has been done using Coq
    - including promotion and conformsTo operators
  - future work: define a minimal endogenous language to define the reference semantics
  - future work: generate operational semantics
  - future work: help in proving translational semantics (bisimulation)
- **Models@runtime**: application domain for behavioral semantics definition
  - ongoing work
  - definition of DSML to describe self-* distributed systems.