Application of Model-Driven Engineering and Metaprogramming to DEVS Modeling & Simulation

Discrete EVent Specification

(PhD Award & Defense - Luc Touraille)

LIMOS

Université Blaise Pascal

cnrs

Introduction

- Context thesis sponsored by the Ministry of Higher Education and Research
- DEVS Modeling & Simulation
- Software Engineering

Outline

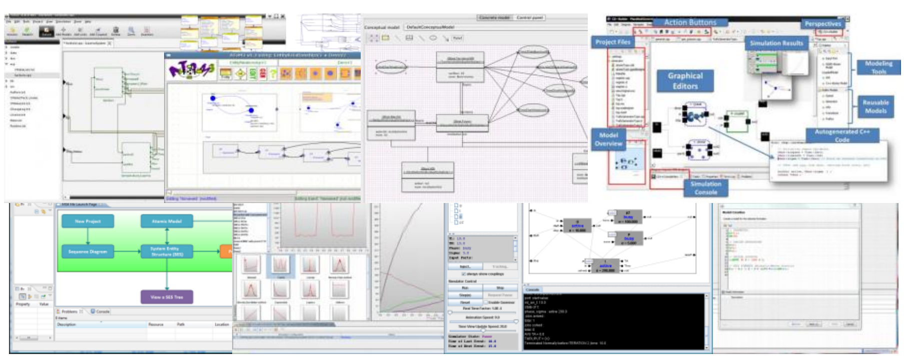
Introduction

1 Context and tackled issues

2 Model-Driven Engineering of DEVS models

3 Generating simulators with metaprogramming

Conclusion



Software tools for DEVS Modeling & Simulation

CONTEXT AND TACKLED ISSUES

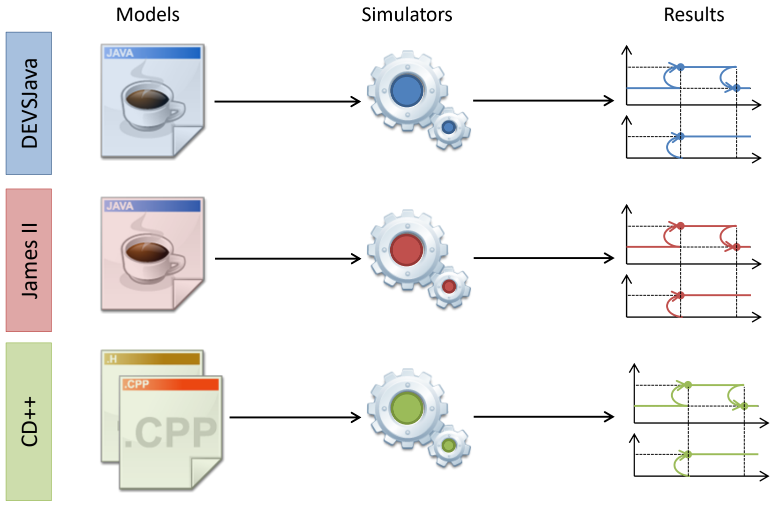
DEVS tools

- Many tools available
  - CD++
  - DEVSJava
  - James II
  - Mimosa
  - Python DEVS
  - …
- No DEVS standard

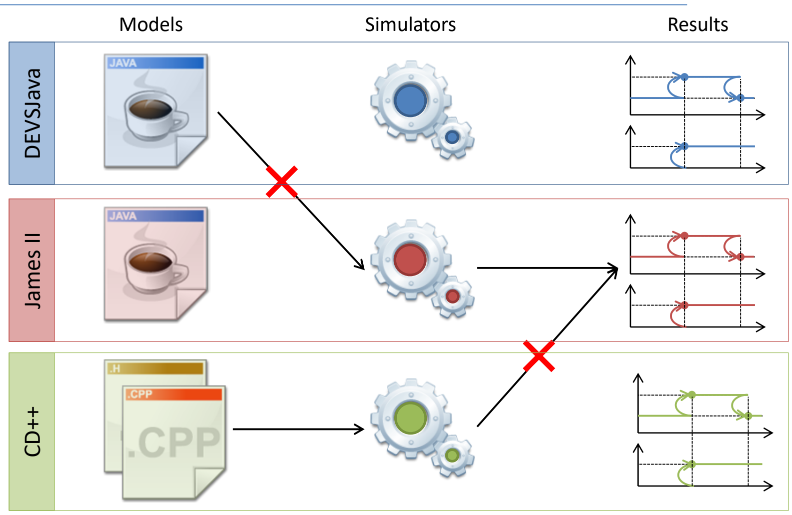
Variety of model specifications

- Code, often as classes
  - Various languages
  - Various frameworks
- Custom textual format
- Diagrams
- English description (in papers)

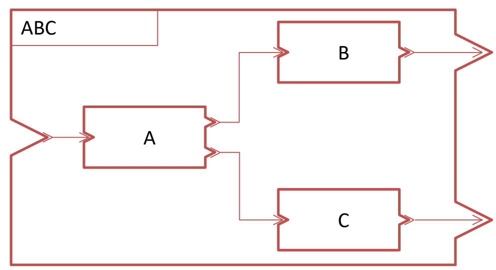
Model portability



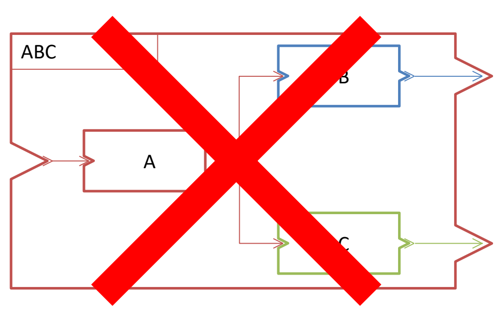
Model portability



Model compatibility



Model compatibility



No interoperability between tools

- Lack of portability
  - Compare simulators?
  - Verify results?
  - Perform simulations with custom inputs?
- Lack of compatibility
  - Reuse previous works to build complex models?
  - Collaborate, share knowledge?

Implementation - Performance

- Increase in resources consumption
- Well researched idea: parallelization
  - Not trivial
  - Not always applicable
- Another idea: shrink simulation overhead as much as possible

Implementation - Model verification

- Model verification = checking that models are correctly implemented
  - w.r.t the DEVS formalism
  - w.r.t. the developer's goal
- Several approaches
  - « Manual » debugging
  - Unit testing
  - Runtime checks by the simulation framework
  - Error-proofing API

Easing implementation and interoperability through Model-Driven Engineering

MODEL-DRIVEN ENGINEERING OF DEVS MODELS: SIMSTUDIO

Aim of SimStudio

- Leveraging MDE techniques and tools to ease development (for the toolsmith and for the practitioner)
- Platform-independent metamodel of DEVS
- Model transformations to automatically generate various artefacts

DEVS metamodel

- Represents the DEVS formalism
  - Abstract syntax for DEVS models
- Conforms to a metametamodel (EMF Ecore)
- Should be:
  - Simple
  - Platform-independent
  - Mappable to existing (implicit) metamodels
- Two facets: structure and behavior

Eclipse Modeling Framework

- Eclipse Modeling Framework (EMF) is an Eclipse-based modeling framework and code generation facility for building tools and other applications.
- From a model specification described in XML Metadata Interchange (XMI), EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.
- Models can be specified using annotated Java, UML, XML documents, or modeling tools, then imported into EMF.
- Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications.

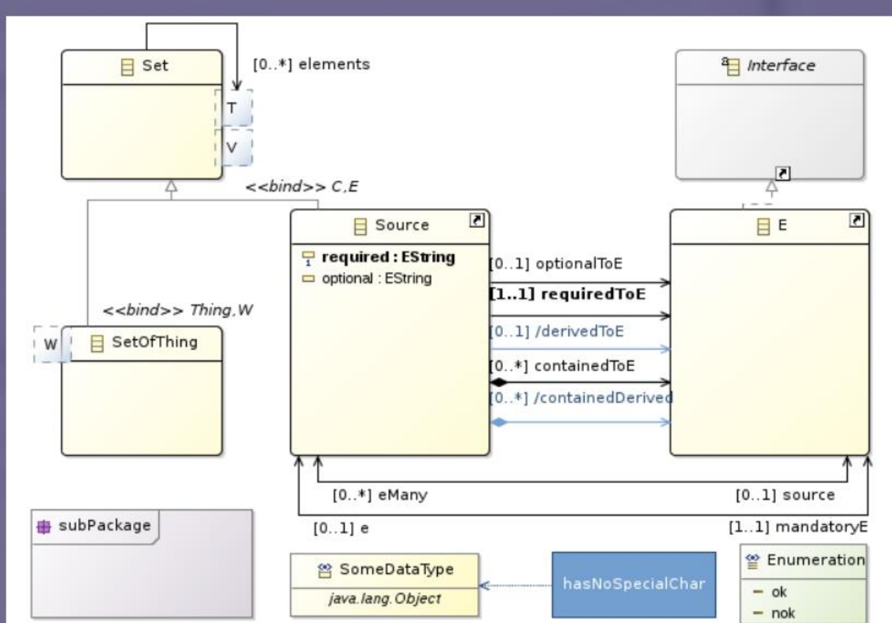
Ecore

- Ecore is the core (meta-)model at the heart of EMF. It allows expressing other models by leveraging its constructs. Ecore is also its own metamodel (i.e.: Ecore can be defined in terms of itself).
- Ecore is the defacto reference implementation of OMG's EMOF" (Essential Meta-Object Facility).
- Using Ecore as a foundational meta-model allows a modeler to take advantage of the entire EMF ecosystem and tooling
- Devlopers can defining their own metamodels based on Ecore.

Concepts in Ecore

- Ecore is an EMF model defining concepts manipulable concepts in EMF.
- These concepts, always prefixed by an "E", are as follows:
  - EAttribute, EAnnotation, EClass, EClassifier,
  - EDataType, EEnum, EEnumLiteral
  - EFactory, EModelElement, ENamedElement,
  - EObject, EOperation
  - EPackage, EParameter, EReference, EStructuralFeature
  - ETypedElement, EStringToStringMapEntry
  - EGenericType, ETypeParameter

EcoreTools - Graphical Modeling for Ecore



Structure

- Direct translation of the formalism into Ecore
  - Entities: models, ports, components, …
  - Relations: a model has ports, a coupling associate two ports, …
- Input, output and state sets represented as types
  - Reuse of Ecore itself

Verification of structural features

- Constraints embedded in the metamodel, through:
  - Cross-references
  - Invariants
- Automatically checked by the MDE framework during model design
- For instance:
  - Couplings validity, including port compatibility
  - Range of the tie-breaking function
  - Identifiers uniqueness in a given scope

Behavior (dynamics)

- Challenge: representing arbitrary computations
  - Without being tightly tied to a specific platform
  - Without restricting the expressiveness
  - Without rewriting all existing libraries
- Recurring need in the MDE community
  - Still an open question

« Semi-generic » language

- Common denominator of the most widespread programming languages
  - Usual constructs: variables, loops, conditionals, functions, arithmetic operators, …
- Extension points through platform-specific snippets
  - Abstract operations for which platform mappings need to be defined
  - ≈ macros with different definitions depending on the target platform

Model manipulation

- Model-to-model transformations
  - Transform a model conforming to a metamodel into another model conforming to a metamodel
  - Map elements of the input model(s) to elements of the output model(s)
- Model-to-text transformations
  - Generate text from a model
  - Template engine

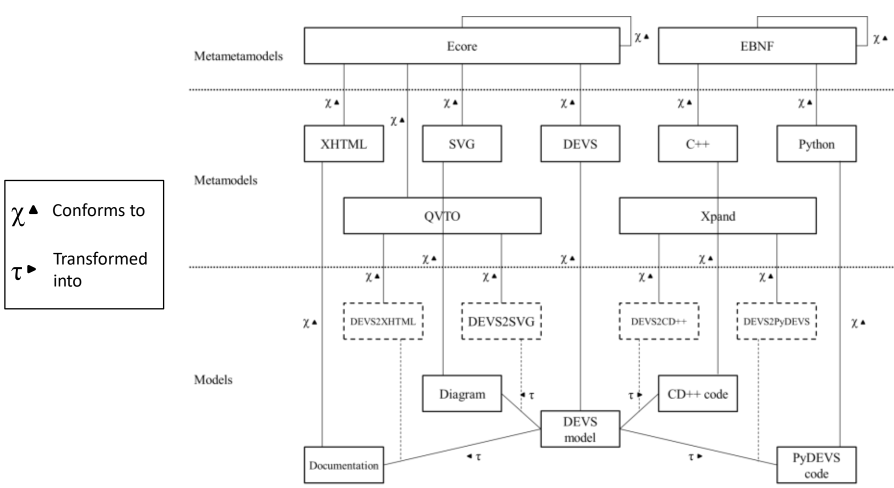
Model-to-model transformations

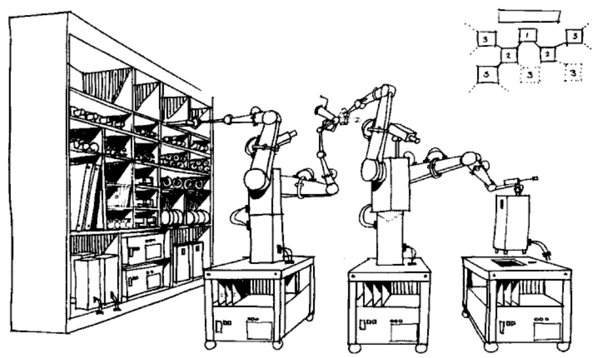
- Diagram generation
  - DEVS to SVG
- Documentation generation
  - DEVS to XHTML
- Idea: inter-formalism transformations

Model-to-text transformations

- Generation of model specifications for various platforms:
  - CD++
  - DEVS-MS
  - PyDEVS
- Solution to the interoperability issue
  - One specification usable in multiple tools

Megamodel



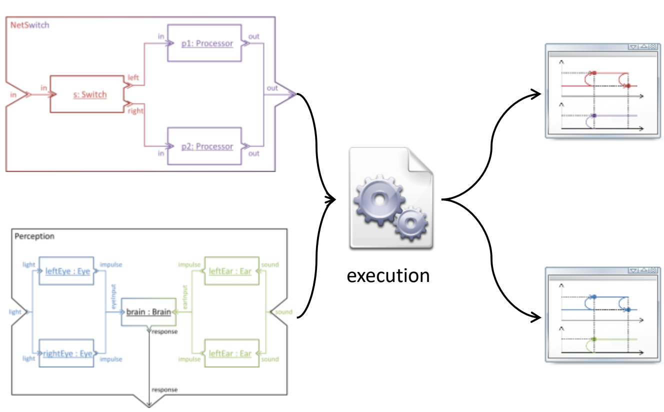


Improving model verification and simulation performance through metaprogramming

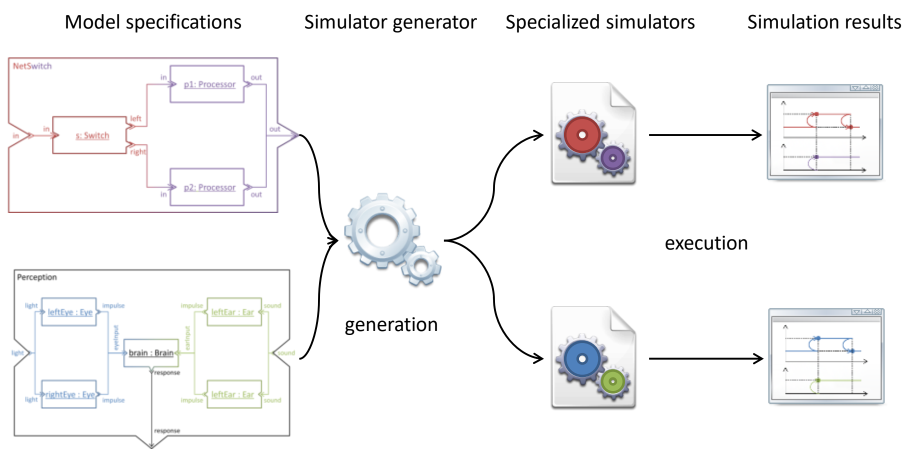
GENERATING SIMULATORS WITH METAPROGRAMMING: DEVS-METASIMULATOR

Automatic generation of specialized simulators

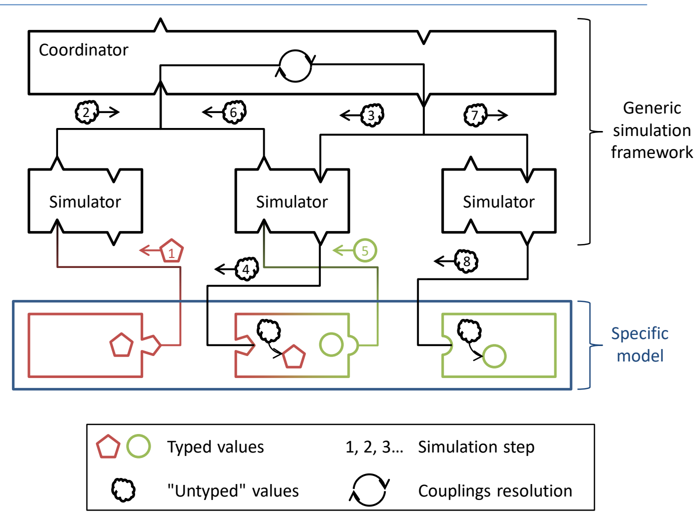
Model specifications   Generic simulator   Simulation results



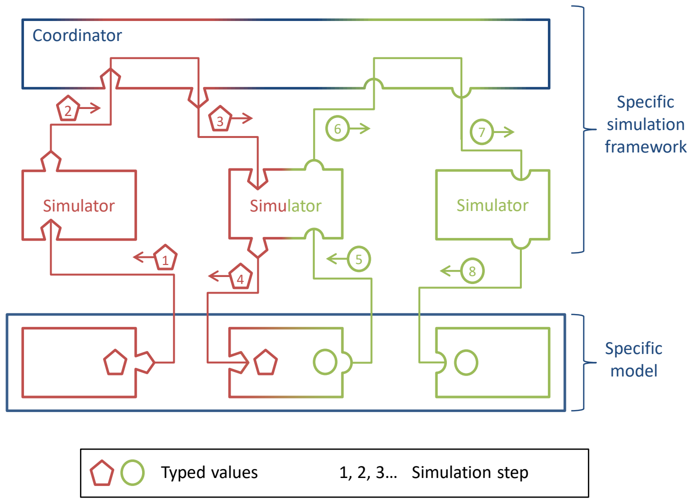
Automatic generation of specialized simulators



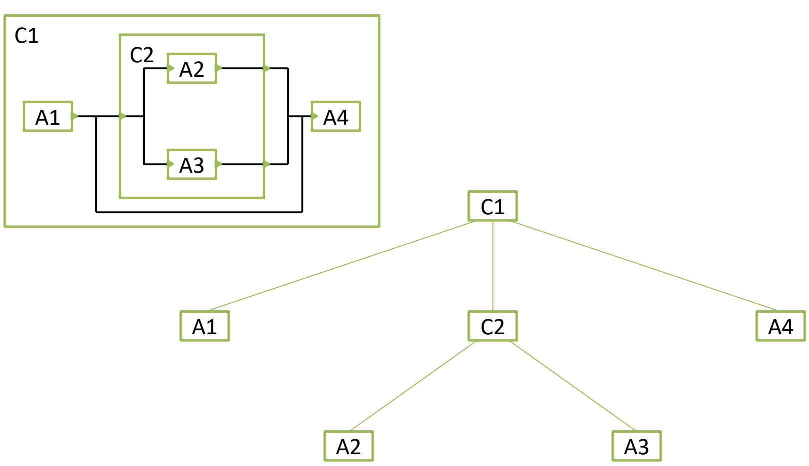
Type erasure in generic simulator



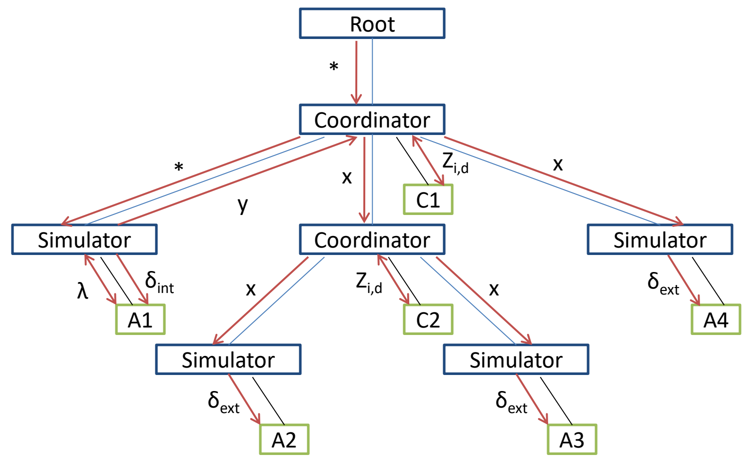
Type preservation in specialized simulator



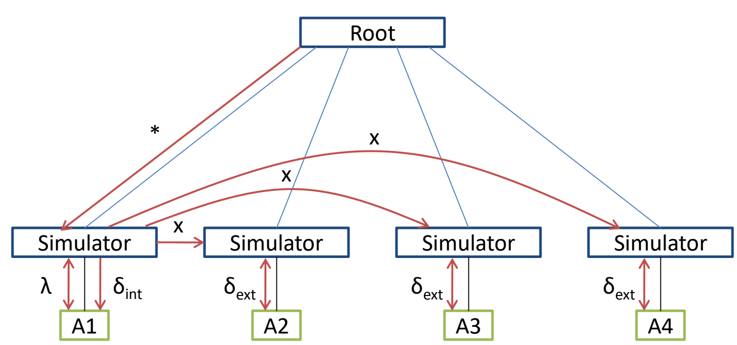
Message exchanges in generic simulator



Message exchanges in generic simulator



Message exchanges in specialized simulator



DEVS-MetaSimulator approach (1)

+ Make information about models available to the compiler
  - Port types and names
  - Couplings
  - Tie-breaking function
  - …

+ Parameterize processors with the model they handle at compile-time

DEVS-MetaSimulator approach (2)

+ Perform simulation using classic algorithms, but with many operations moved from runtime to compile-time

+ Enforce constraints at compile-time
  - Type system
  - Static assertions

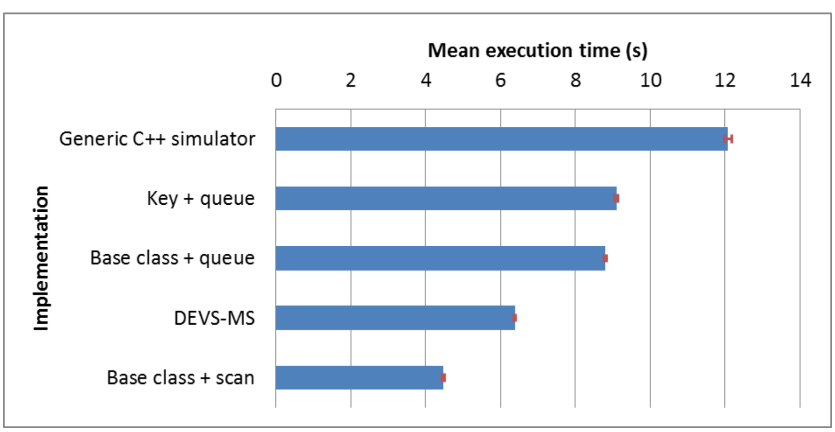
= Improved performance

= Compile-time model verification

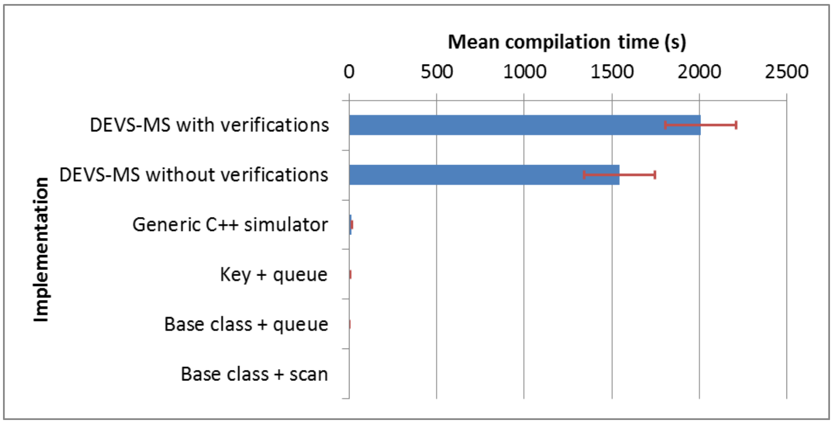
Benchmark

- Sample DEVS model from previous works
- Comparison between various implementations (all in C++)
  - Generic DEVS simulation library
  - DEVS-MetaSimulator
  - Three implementations devoted to this particular model, manually crafted

Comparison of execution times



Comparison of compile times



Model verification

- Design errors caught at compile-time
  - Impossible to miss
  - Reported early
- Many constraints implemented:
  - Couplings validity
    - Valid source and destination
    - Type checking, supporting subtyping, conversions, etc.
    - No direct feedback loop
    - …
  - Range of the tie-breaking function
  - No modification of the state outside of transition functions
  - …

CONCLUSION

SimStudio – Summary

- Proof of concept of the potential of MDE applied to M&S
- Facilitate development for toolsmiths
  - Numerous tools available
  - High-level domain-specific languages for model manipulation
- Provide interoperability to practitioners
  - Platform-independent model specifications
  - Automatic generation towards various platforms

SimStudio – Future works

- Complete the definition of the « semi-generic » language
- Integrate with the DDML editor
- Experiment with inter-formalism transformations
- Standardize DEVS?

DEVS-MS – Summary

- Metaprogramming = genericity + efficiency + static checking
- Generation of specialized simulators from a single library
  - No burden on the practitioner
  - Improved performances by precomputing part of the simulation at compile-time
  - Improved error checking by analyzing models at compile-time

DEVS-MS – Future works

- Optimize for compile-time performance
- Implement a Parallel-DEVS version
  - Possibly generating highly parallel code by taking into account the structure of the model

THANK YOU FOR YOUR ATTENTION