electronics

MDPI

Article

Open Source Languages and Methods for Cyber-Physical System Development: Overview and Case Studies

Lena Buffoni [1,*], Lennart Ochel [1], Adrian Pop [1], Peter Fritzson [1], Niklas Fors [2], Görel Hedin [2], Walid Taha [3] and Martin Sjölund [1]

[1] Department of Computer and Information Science, Linköping University, IDA, 581 83 Linköping, Sweden; lennart.ochel@liu.se (L.O.); adrian.pop@liu.se (A.P.); peter.fritzson@liu.se (P.F.); martin.sjolund@liu.se (M.S.)
[2] Department of Computer Science, Lund University, 221 00 Lund, Sweden; niklas.fors@cs.lth.se (N.F.); gorel.hedin@cs.lth.se (G.H.)
[3] Department of Computing and Electronics for Real-Time and Embedded Systems, Halmstad University, 301 18 Halmstad, Sweden; walid.taha@hh.se
* Correspondence: lena.buffoni@liu.se

Abstract: Industrial cyber-physical system products interleave hardware, software, and communication components. System complexity is increasing simultaneously with increased demands on quality and shortened time-to-market. To effectively support the development of such systems, we present languages and tools for comprehensive integrated model-based development that cover major phases such as requirement analysis, design, implementation, and maintenance. The model-based approach raises the level of abstraction and allows to perform virtual prototyping by simulating and optimizing system models before building physical products. Moreover, open standards and open source implementations enable model portability, tool reuse and a broader deployment. In this paper we present a general overview of the available solutions with focus on Modelica/OpenModelica, Bloqqi, and Acumen. The paper presents contributions to these languages and environments, including symbolic-numeric modeling, requirement verification, code generation, model debugging, design optimization, graphical modeling, and variant handling with traceability, as well a general discussion and conclusions.

Keywords: cyber-physical systems; modeling languages; equation-based; simulation; open source

Citation: Buffoni, L.; Ochel, L.; Pop, A.; Fritzson, P.; Fors, N.; Hedin, G.; Taha W.; Sjölund, M. Open Source Languages and Methods for Cyber-Physical System Development: Overview and Case Studies. *Electronics* **2021**, *10*, 902. https://doi.org/10.3390/electronics10080902

Academic Editor: Juan M. Corchado, Luis Gomes and Tony Givargis

Received: 31 January 2021
Accepted: 28 March 2021
Published: 10 April 2021

1. Introduction

General purpose software development has made a lot of progress over the past twenty years, with the adoption of powerful programming languages, agile methodology, software tool pipelines, standardization, and open source development. The same development has not yet fully taken place in the area of cyber-physical systems (CPS). Instead, many CPS languages and tools are proprietary, often restricted to particular application domains, and lack precise language definitions and APIs, making interoperability with other tools and languages difficult and error prone.

There is still a lot of work to be done regarding the integration of the modeling and model-based development tools into a full product development cycle (such as the one in Figure 1), with research ongoing in areas such as requirement verification, testing and distributed compilation.

Several industry standards are partly informal, based on text and diagrams, lacking unambiguous definition of syntax and semantics, like IEC 61131-3 [1] for programmable logic controllers, and OMG SysML [2] for systems modeling. As a result, different vendors interpret the standards differently, use their own serialization formats, and implement vendor-specific extensions. This hinders collaboration and portability between tools, and leads to vendor lock-in and an increase in development costs. On the other hand, there are also a few industrial standards with more precisely defined semantics, like Modelica [3],

Functional Mock-up Interface (FMI) [4], and Very high speed integrated circuit Hardware Description Language—Analog and Mixed-Signal extensions (VHDL-AMS) [5], which enable model portability and reuse between tools.

In this paper, we primarily discuss open approaches and open source implementations. Several of these are based on open formal language definitions. We outline how this can take advantage of advances in modern language technology and software development methodology. In the modeling community, such approaches have, over the past two decades, led to the emergence of open formal modeling languages and standards, like Modelica [3], FMI [4], Distributed Co-Simulation Protocol (DCP), and Structure and System Parameterization (SSP) [6]. This has led to an increasing shift towards open source and collaborative tool development from the industry. While this approach is very fruitful, the CPS development methodologies and tools based on these languages have not yet reached the maturity levels of several general purpose software development tools.

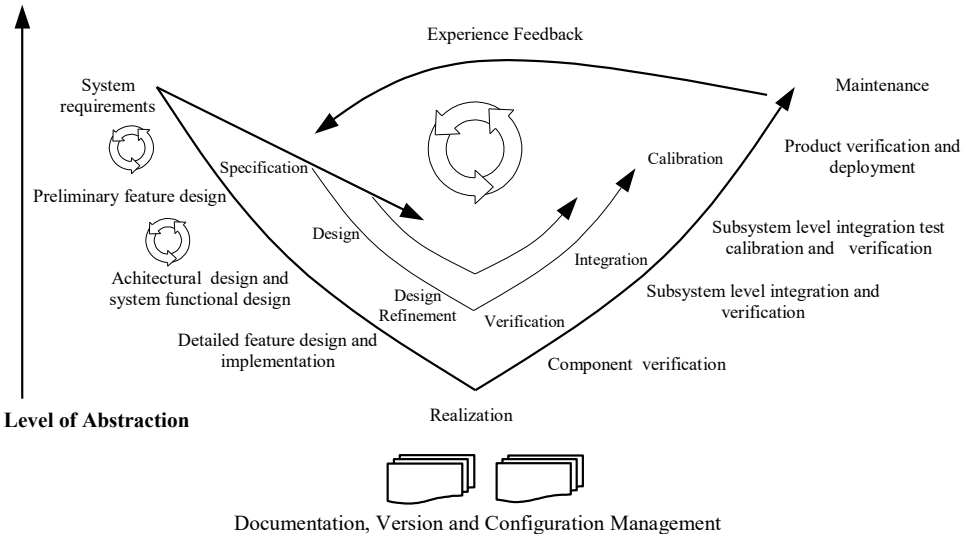


Figure 1. The so-called product design-V includes all phases of product development from requirement specification, to modeling, implementation, deployment, and maintenance.

In the following we provide an overview of some current state-of-the-art and currently ongoing work on CPS languages and tools using open formal language definitions. We mention key features that are essential in languages for cyber-physical system modeling and development, based on our experience and research in that domain. The following aspects are discussed:

- CPS languages,
- Textual vs. graphical programming,
- Formal semantics—language and simulations,
- 3D visualization,
- Debugging, testing, requirement verification,
- Traceability,
- Control applications.

In Section 2 we discuss different CPS languages based on open formal language definitions, including Modelica, Acumen, and Bloqqi. Section 3 discusses tool support, including debuggers, graphical editors, and web-based interfaces. Section 4 presents tool implementation techniques, supporting extensible tooling through high-level declarative implementation languages (Feature Programming, MetaModelica, Julia, Reference Attribute Grammars (RAGs)). Section 5 discusses testing and verification of CPS systems. Section 7 concludes the paper.

2. CPS Languages and Tools

A Cyber-Physical System (CPS) integrates physical components with software. Control systems coupled to physical systems is one common example of CPS. A model-based approach to CPS development is based on describing both the physical and software parts through models, allowing the whole system to be simulated before it is deployed. This way, virtual engineering can be supported, allowing much of testing, verification, and optimization to be performed through simulations. Simulation is often used in conjunction with physical systems, such as in hardware-in-the-loop simulation, or to completely replicate a physical entity, such as digital twins [7].

CPS languages can be used to describe either the physical components, the control software, or both. Models written in one language can be combined with models in other languages, for example, by using FMI. FMI [4] and SSP [6] are open standards that define a model format using the C language (behavior) and XML (interface) for representing precompiled models that can be exchanged between tools and/or combined to be co-simulated (further discussed in Section 2.4).

Modelica [3] is a mature industrial modeling language with multiple implementations, both proprietary and open source. Modelica is aimed at modeling and simulation of cyber-physical systems, but it has also been used for automatically generating deployable (embedded) control software (C code) from models [8].

Acumen [9] is a research language for modeling and simulation of physical systems, in particular aimed at exploring rigorous simulation.

Bloqqi [10] is a research language and tool for control software, supporting generation of C code for deployment. The language is aimed in particular at exploring reusability of control models.

Common to these languages is that they have well-defined syntax and semantics, and employ types to support user-defined abstractions. The serialized format is readable text, similar to ordinary general-purpose programming languages. However, interactive tools can, in several cases, visualize and support direct editing of visual views. The use of a well-defined textual representation has many advantages, similar to the case for general-purpose languages: it allows new tools to be built independently of other tools, it allows the rich set of general text-based tools to be used, and it works well with ordinary version control systems. Furthermore, textual representations are often used by more advanced users for more efficient development.

In the following subsections we will discuss these languages and formats in some more detail.

2.1. The Modelica Language and the OpenModelica Environment

Modelica is an acausal equation-based object-oriented modeling language for cyber-physical system modeling [11]. In Modelica, behavior is described declaratively using mathematical equations and functions. Object-oriented concepts are used to encapsulate behavior and facilitate reuse of model components. The acausal and object-oriented aspects of Modelica make it particularly well suited for code reuse through libraries. Modelica is superior to most other modeling formalisms due to the following important properties:

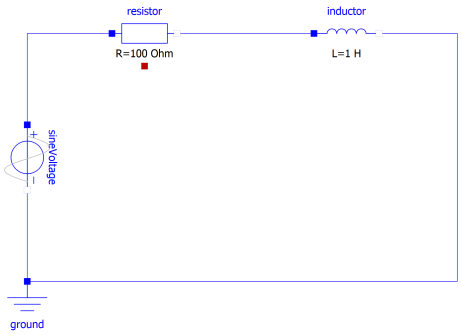
- Object-oriented modeling. This technique makes it possible to create physically relevant and easy-to-use model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.
- Acausal modeling. Modeling is based on equations instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases reusability of model components, since components adapt to the data flow context in which they are used. This generalization enables both simpler models and more efficient simulation.
- Physical modeling of multiple domains. Model components can correspond to physical objects in the real world, in contrast to established techniques that require con-

version to signal blocks. For application engineers, such "physical" components are particularly easy to combine into simulation models using a graphical editor.

- Hybrid modeling. Modeling of both continuous-time and discrete-time aspects of systems is supported in an integrated way. From Modelica 3.3, clocked discrete-time modeling is also supported for increased modeling precision and simulation performance.

A large set of Modelica libraries is available, under both free and commercial licenses, with the most important being the Modelica Standard Library (MSL) [12]. MSL version 4.0.0 released in 2020 contains about 1400 model components and 1200 functions from many domains.

The Modelica language support of both visual and textual views of the same model is shown by the example in Figure 2. Since the visual view is defined by standardized graphical annotations, both the visual view and the textual view are preserved if the model is moved between Modelica tools, as well as allowing both visual and textual model editing.




```
model RLCircuit
  SineVoltage sineVoltage;
  Ground ground;
  Resistor resistor(R = 100);
  Inductor inductor(L = 1);
equation
  connect(sineVoltage.n, ground.p);
  connect(sineVoltage.p, resistor.p);
  connect(resistor.n, inductor.p);
  connect(inductor.n, ground.p);
end RLCircuit;
```

Figure 2. Graphical vs. textual view of the same Modelica model. (**left**) A simple RL-circuit is modeled using Modelica graphical connection diagrams. (**right**) Textual view of the RL-circuit Modelica model.

Cyber-physical modeling including multiple domains is illustrated in Figure 3. The model contains parts from three domains, two physical and one cyber: an electric part using components from the `Modelica.Electrical` library, a mechanical part using components from the `Modelica.Mechanical` library, and a control (cyber) part using the `Modelica.Blocks` library.

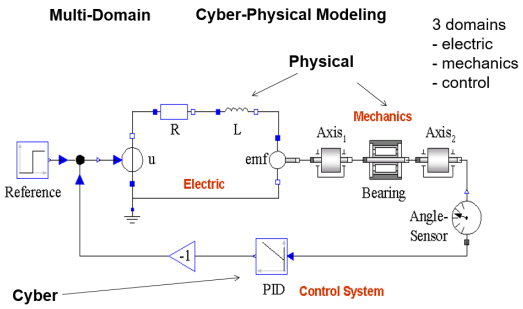


Figure 3. A simple example of a multi-domain cyber-physical model in Modelica. Two physical domains are present: electrical and mechanical, and one software (cyber) domain for the control system part.

As already mentioned, modeling in Modelica of both continuous-time and discrete-time aspects of systems is possible in an integrated way. In the Modelica 3.3 and later versions of the language, clocked discrete-time modeling is also supported. This is illustrated in Figure 4.

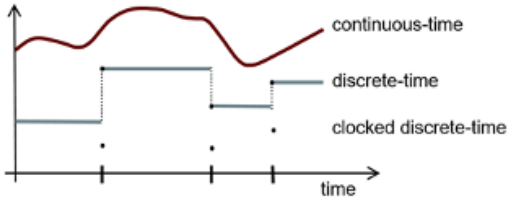


Figure 4. Illustration of hybid modeling in Modelica, allowing combinations of continuous-time, discrete-time, and clocked discrete-time variables.

A slightly larger textual Modelica model example is displayed in Figure 5, with the bat-and-ball model. A ball is hit by a racquet causing it to reverse direction via a bounce. This is another example of a hybrid system where the `reinit` construct causes a discontinuous change of ball movement at the bounce event. A 3D visualization of an advanced ping pong model in Acumen is shown in Figure 8. The results of simulation with the different approaches are compared in Section 3.

```
model BatAndBall
  Real ball_h(fixed=true, start=1.0) "height of ball";
  Real ball_v(fixed=true, start=0.0) "velocity of ball";

  Real bat_h(start=0.0, fixed=true) "height of bat";
  Real bat_v(start=0.0, fixed=true) "velocity of bat";

  Real e "Error";

  parameter Real r=0.9 "coefficient of restitution";
  parameter Real g=10.0 "gravitational acceleration";

equation
  // Dynamics for ball bouncing on bat
  der(ball_v) = -g;
  der(ball_h) = ball_v;

  when ball_h < bat_h then
    reinit(ball_h, bat_h);
    reinit(ball_v, -r*(ball_v-bat_v) + bat_v);
  end when;

  // Dynamics for bat (controlled by a force)
  der(bat_h) = bat_v;
  der(bat_v) = hold(e);

  // Sampling and calculation of error
  e = sample(-2*bat_h - 0.4*(ball_v-bat_v), Clock(1, 10));
end BatAndBall;
```

Figure 5. A textual Modelica model example for playing ping pong that combines continuous-time, discrete-time, and clocked discrete-time equations.

The OpenModelica Environment

OpenModelica [13] is an open source Modelica- and FMI-based modeling, simulation, optimization, model-based analysis, and development environment. It includes a number of facilities such as textual and graphical model editing, simulation, optimization, debugging, visualization and 3D animation, requirement verification, web-based model editing and simulation, scripting from Python, Julia, Matlab, Modelica; efficient simulation

and co-simulation of FMI-based models using its OMSimulator subsystem. The most important subsystems are the OpenModelica Compiler (OMC) and the OMEdit graphical connection editor and user interface for simulation, plotting, and debugging. OMC is implemented in MetaModelica, an extended version of Modelica (Section 3). Models are compiled to efficient C or C++ code. Experimental Java and C# code generators have also been developed.

Modelica models can be created and edited graphically, by dragging and dropping and connecting together existing model components from libraries, or textually using ordinary text editing. Figure 6 illustrates the graphical user interface. To the left is the library browser, in the center is the model, shown graphically or textually. The upper right pane shows model documentation, and lower right pane displays the plot variable browser, to select which variables should be plotted.

Figure 7 shows OpenModelica simulating the Chua Circuit and plotting two variables, the `C1.v` and `C2.v`, which are selected in the plot variable browser to the right.

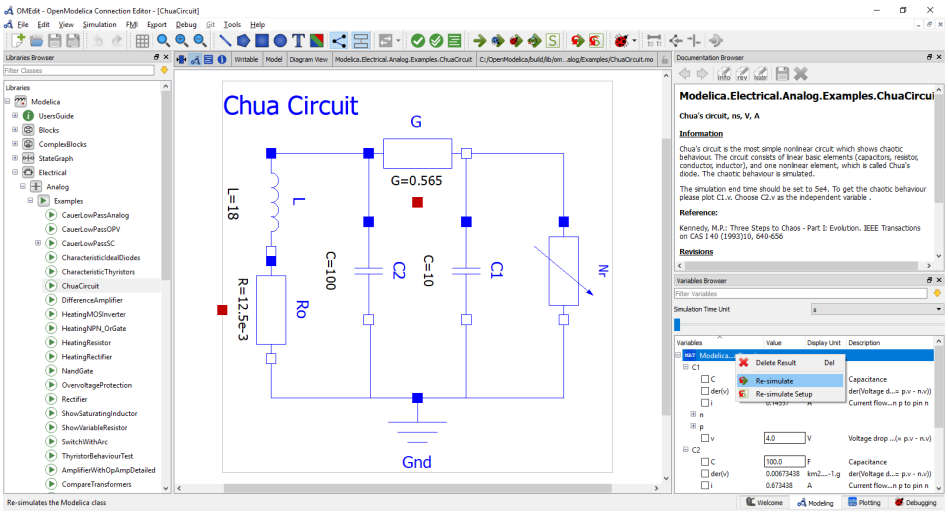


Figure 6. OpenModelica graphical editor OMEdit on a Chua Circuit Modelica model. Upper right: model information pane. Lower right: plot variable control pane.

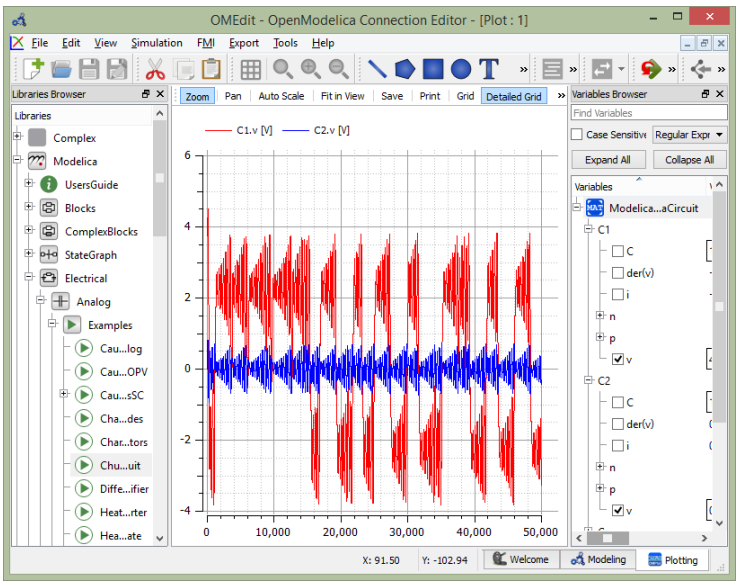


Figure 7. OpenModelica graphical editor OMEdit displaying the simulation results of the Chua Circuit model.

2.2. Acumen Language and Environment

Acumen [9] is an experimental modeling environment for cyber-physical systems aimed at better understanding of usability and foundational challenges. To address usability, the current implementation features a textual modeling language that can express hybrid (continuous/discrete) dynamics and an interactive development environment (IDE) that supports animated 3D visualization. The core language is minimal, consisting of continuous formulae, discrete formulae, and conditional statements that enable or disable formulae. Variables include state variables and derivatives. Extensions of the core language include support for features that increase convenience but can be eliminated statically, such as equations and partial derivatives. Both these features have proven necessary and effective for expressing classical mechanics problems using the Euler-Lagrange equation, which significantly simplifies modeling of the rigid body dynamics of a robotics problem. This implementation has been used for developing and teaching a course on cyber-physical systems at Halmstad University since 2013 [14].

The Acumen environment includes an Acumen language interpreter implemented in Scala and a graphical user interface that has three main panes: model, results, and console. The environment shows the model with keyword highlighting, and supports size change. The results pane has three modes: table, plot, and 3D (Figure 8). The table and plot modes are automatically populated for any model run with information about the trace of the system simulation.

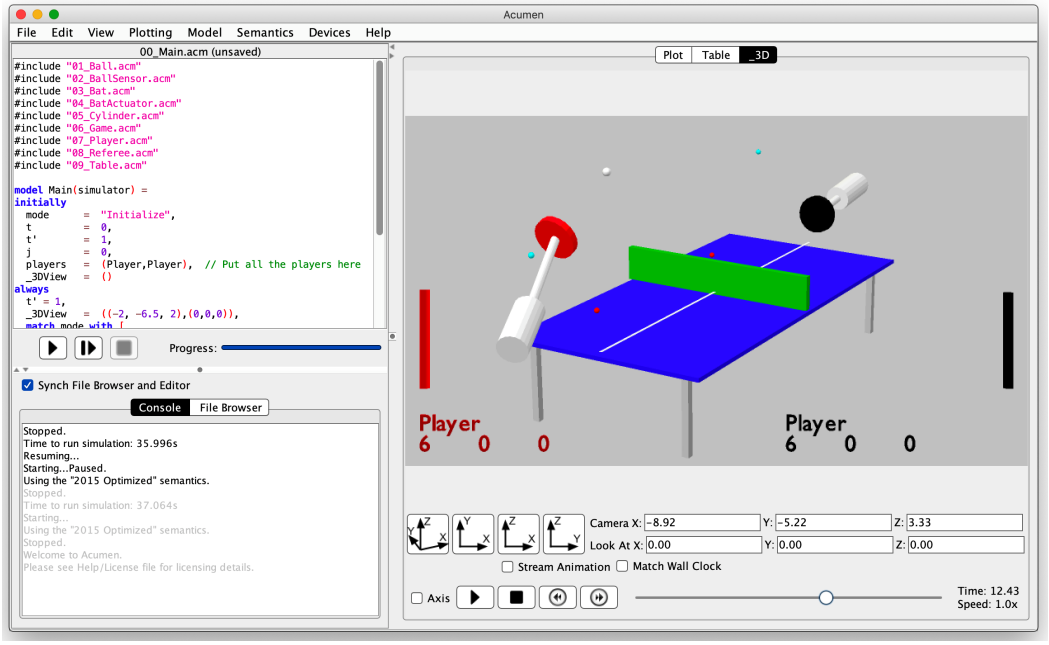


Figure 8. The Acumen environment including 3D visualization of a ping pong playing model.

To address foundational aspects, the Acumen implementation supports both "traditional" and "enclosure" semantics for simulation, the key distinction being that traditional simulation produces a collection of discrete points at discrete times, whereas exact solutions exist for all real-valued times and are in general real-value rather than belonging to a finite set like floats. As such, enclosures address correctness and point to the need for using sets rather than individual points if we are to produce simulation results with any correctness guarantees. Foundational issues in relation to the development of Acumen include showing that there are methods for correctly simulating Zeno systems [15] without diverging or sacrificing correctness. More recent work has focused on developing categorical semantics [16] for hybrid systems modeling languages.

2.3. Bloqqi Language and Environment

Bloqqi [10,17] is a research language and tool for building control systems, with the main goal of experimenting with language constructs facilitating code reuse. In particular, it supports a kind of feature-oriented development [18], where variants of a system can be constructed by selecting features from a library.

Bloqqi programs have state and directed data-flow, and execute periodically. In each periodic scan, input sensor signals and local state are used for computing output actuator signals and the next local state. Bloqqi is aimed for control programs rather than simulation, and does not include a numerical solver. (Bloqqi data-flow may not contain cycles. However, it is possible to specify algebraic loops through the use of explicit state variables: when a state variable is read, the value from the previous period is used.) The Bloqqi compiler supports generation of C code for deployment on suitable hardware.

Figure 9 shows a simple example of a Bloqqi program regulating the liquid level of a tank. The program periodically reads the input tank level, and outputs two control signals to open/close the upper/lower valves on the tank. The details of the regulator, a simple on/off controller, are shown in Figure 10.

Similar to Modelica, Bloqqi is a textual language with a visual representation. Programs can be edited either visually or as text. Both Figures 9 and 10 are screenshots from the Bloqqi interactive editor. Figure 11 shows the corresponding textual representation of the tank controller program.

To test a Bloqqi program, physical components (e.g., the tank) can be replaced by simulations implemented in other languages, such as Modelica. For the ping pong playing model, we implemented the controller part in Bloqqi, see Figure 12, and ran it together with the simulation part of the Modelica model in Figure 5. Bloqqi also supports distributed execution over MQTT [19] (a publish/subscribe middleware).

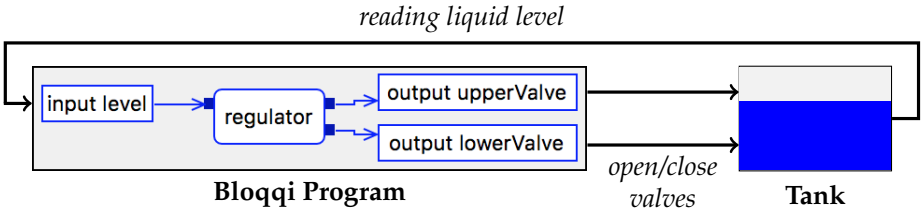


Figure 9. A simple Bloqqi program that controls a tank. Regulator details are shown in Figure 10.

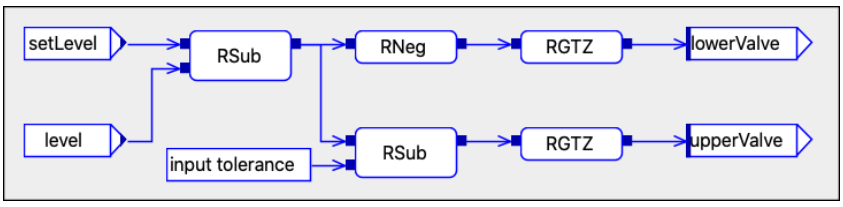


Figure 10. Bloqqi details for the on/off regulator in Figure 9.

```
diagramtype Main {
  output upperValve: Bool;
  output lowerValve: Bool;
  input level: Real;
  input setLevel: Real;
  regulator: Regulator;
  connect(regulator.lowerValve, upperValve);
  connect(regulator.upperValve, lowerValve);
  connect(level, regulator.level);
  connect(setLevel, regulator.setLevel);
}
diagramtype Regulator(level: Real, setLevel: Real
                      => upperValve: Bool, lowerValve: Bool) {
  input tolerance: Real;
  RSub_1;
  RSub_2;
  RNeg_1;
  RGTZ_1;
  RGTZ_2;
  connect(setLevel, RSub_1.in1);
  connect(RSub_1.out, RSub_2.in1);
  connect(RSub_2.out, RGTZ_1.in);
  connect(RSub_1.out, RNeg_1.in);
  connect(RNeg_1.out, RGTZ_2.in);
  connect(RGTZ_2.out, lowerValve);
  connect(RGTZ_1.out, upperValve);
  connect(level, RSub_1.in2);
  connect(tolerance, RSub_2.in2);
}
diagramtype RGTZ(in: Real => out: Bool) {
  gt: RGt;
  connect(in, gt.in1);
  connect(0.0, gt.in2);
  connect(gt.out, out);
}
```

Figure 11. Bloqqi textual representation of the tank controller in Figures 9 and 10.

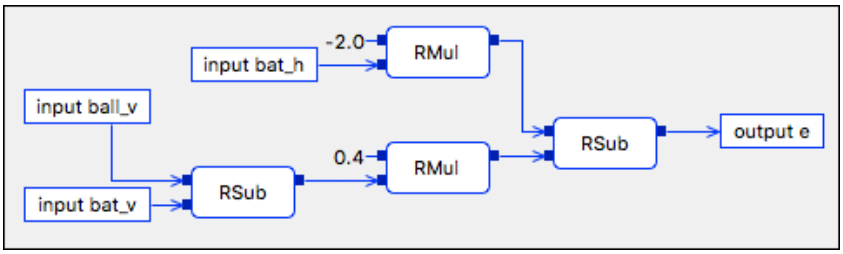


Figure 12. Bloqqi controller for the Modelica Ping Pong model in Figure 5.

Feature-Oriented Programming in Bloqqi

Feature-oriented development [18] supports the construction of variants of a system, by selecting what features to include. Bloqqi supports a kind of feature-oriented development based on object-orientation combined with new language constructs.

Like Modelica, Bloqqi is object-oriented, and supports diagram inheritance and block redeclaration in a similar way. Furthermore, Bloqqi supports another specialization mechanism called *connection interception* that allows directed connections to be specialized in

subtype diagrams. Thus, a connection defined in a supertype diagram can be specialized in a subtype to go via another block.

Figure 13 shows a screenshot from the Bloqqi programming environment where a supertype (grey parts) is extended to a subtype (white parts). In this case, the supertype is a proportional controller, and the subtype adds a derivative part. To accomplish this, the original connection between `PPart` and `Out` in the supertype has been intercepted in the subtype to go via a new `Add` block.

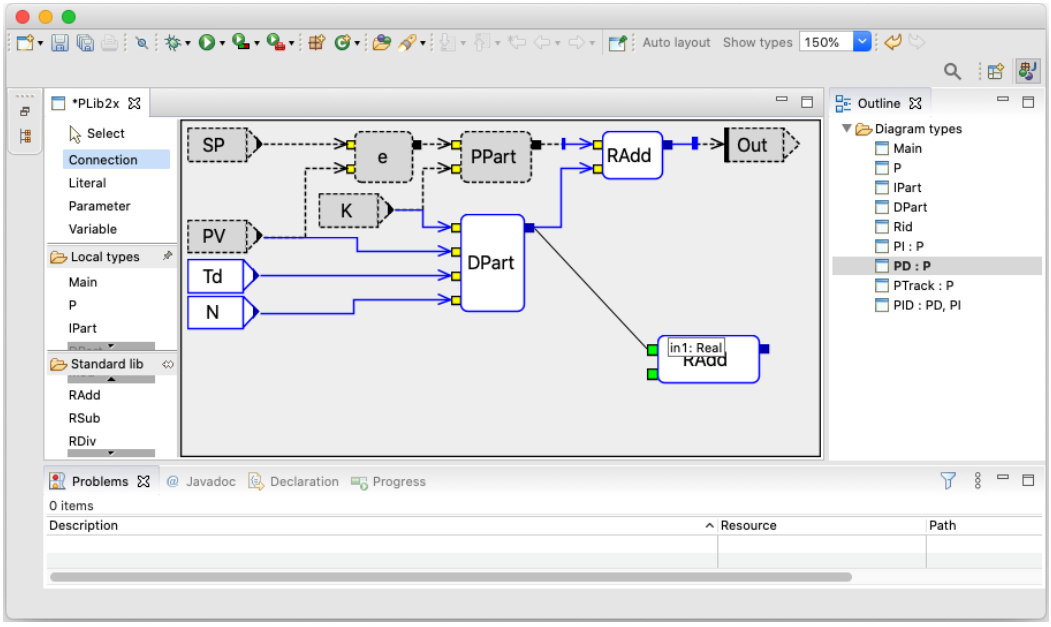


Figure 13. The Bloqqi programming environment. A proportional (P) controller is extended with a derivative (D) part.

A subtype defined in this way can be seen as an optional *feature* of the base diagram. A variant of the base diagram can be constructed by selecting one or more such features, and by specializing existing (mandatory) blocks. The Bloqqi language contains constructs for defining libraries of features that specify what subtypes correspond to optional features, what mandatory blocks can be specialized, and how to resolve possible feature interaction, for example, if two features intercept the same port. Based on these feature specifications, the Bloqqi environment can automatically generate a wizard for any Bloqqi base diagram, and present the possible feature choices to the user. The user can thus easily construct a specific variant of a control program by creating a base diagram and using the wizard to select which optional features to add and which mandatory features to specialize.

Figure 14 shows an example wizard for a simple library for Proportional-Integral-Derivative (PID) controllers. The controller always contains a proportional part (a mandatory feature), and the user can, for example, select if also an integral part or a derivative part (or both) should be added.

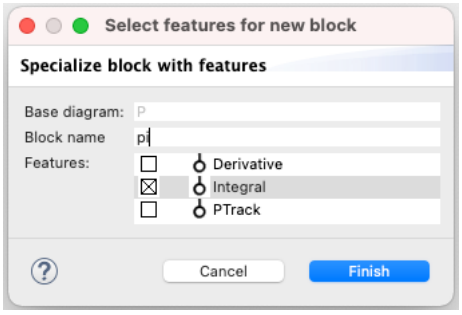


Figure 14. Example of Bloqqi wizard, automatically generated from the simple Proportional-Integral-Derivative (PID) library in Figure 13.

2.4. FMI/SSP

The Functional Mock-up Interface (FMI) standard [4] specifies a way of describing and packaging causal models in either compiled binary form or source-code (C code and XML descriptors) form. Many tools (including Modelica tools) support exporting models from their native modeling representation into FMI form. The standard is widely used in industry, especially the automotive industry, which initially pushed the development in order to be able to simulate system models consisting of models from multiple tools and modeling formalisms, as depicted in Figure 15. Today, the Modelica Association is maintaining the standard and continuously developing it further. A model or simulation unit is called a Functional Mock-up Unit (FMU) according to the standard. Regarding export from Modelica tools, compared to a Modelica model, which is usually acausal, an exported model in FMU form is less general since it is causal—the causality of ports has to be fixed to input or output.

SSP [6] is a complementary standard to FMI, which specifies how FMUs can be connected to create composite models and how they can be parameterized. Both FMI and SSP are standardized by the Modelica Association.

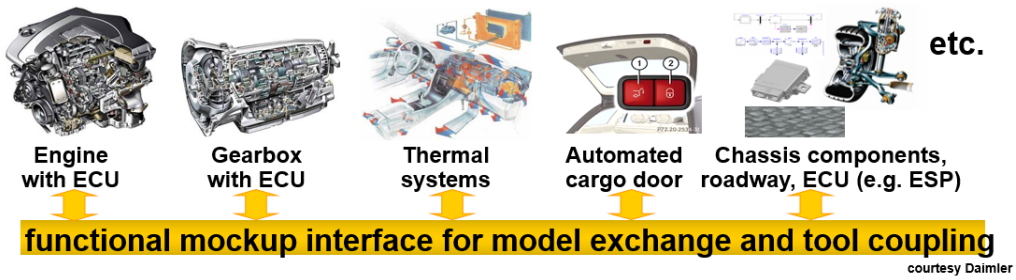


Figure 15. Automotive industry applications of Functional Mock-up Interface (FMI), allowing models from several domains to be simulated together.

The OpenModelica environment can be used to both export any given Modelica model as an FMU and import FMUs to create a composite model using its OMSimulator subsystem (Figure 16). Bloqqi can import FMU blocks and connect them to Bloqqi blocks.

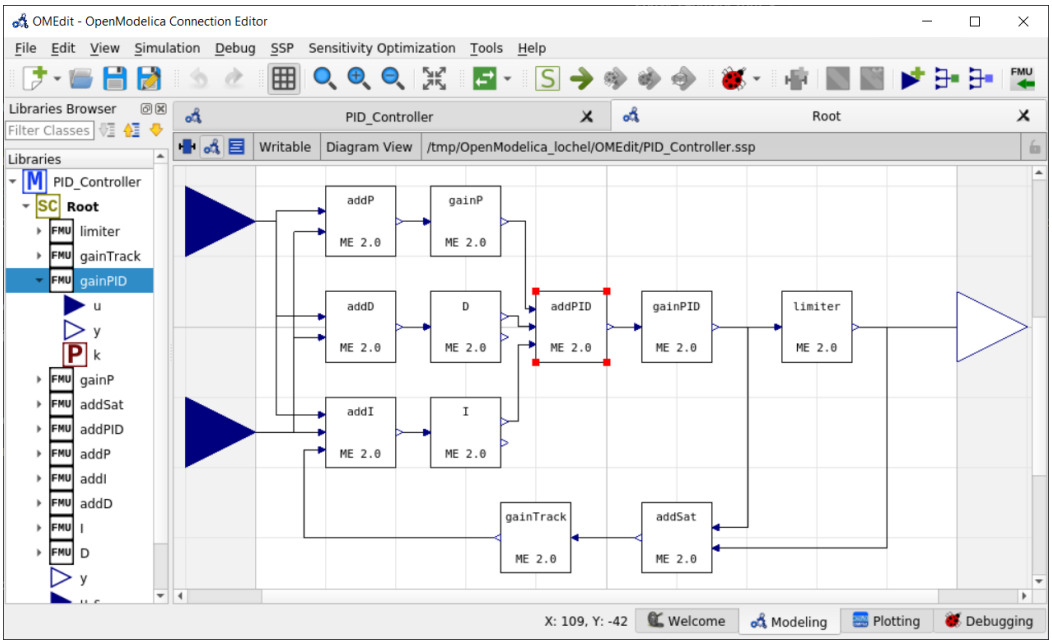


Figure 16. Composite model of a PID controller composed of 11 Functional Mock-up Units (FMUs) connected according to the Structure and System Parameterization (SSP) standard in the OpenModelica OMEdit tool using the OMSimulator subsystem.

2.5. Some Other Languages and Tools

In addition to the OpenModelica tool presented here, there are a number of proprietary Modelica tools, including Dymola [20] from Dassault Systems, Impact [21] from Modelon, solidThinking Activate [22] from Altair, Wolfram SystemModeler [23] from Wolfram, MWorks [24] from Suzhou Tongyuan, LMS Imagine.Lab Amesim [25] from Siemens PLM Software, Simplorer [26] from ANSYS, SimulationX [27] from ESI ITI GmbH and MapleSim [28] from Maplesoft.

OpenModelica contains a package manager to quickly install and update Modelica libraries; it is similar to the `impact` [29] package manager, which has a web frontend for searching for libraries.

Simulink [30], is an extension of MATLAB [31] and provides a graphical block-based approach to model-based design that does not support acausal modeling. Simscape, an extension of Simulink for physical system modeling, focuses on component models with connections based on physical connections and also allows equation-based modeling.

Ptolemy II [32] is an open source software framework supporting experimentation with actor-oriented design. Actors are software components that execute concurrently and communicate through messages sent via interconnected ports. A model is a hierarchical interconnection of actors. In Ptolemy II, the semantics of a model is not determined by the framework, but rather by a software component in the model called a director, which implements a model of computation. The Ptolemy Project has developed directors supporting process networks (PN), discrete-events (DE), dataflow (SDF), synchronous/reactive (SR), rendezvous-based models, 3-D visualization, and continuous-time models. Each level of the hierarchy in a model can have its own director, and distinct directors can be composed hierarchically.

A prototype integrating OpenModelica in the Ptolemy framework has been implemented [33]. An integration of any Modelica tool that supports FMI export is possible with Ptolemy II through co-simulation with FMUs.

The Very high speed integrated circuit Hardware Description Language—Analog and Mixed-Signal extensions (VHDL-AMS) [5], 1999, is a hardware description language that has been developed to support the modeling of analog, digital, mixed-signal and mixed-technology systems. The language extends the IEEE standard digital design lan-

guage VHDL (IEEE Std 1076.1-1999) to cover mixed analog and digital systems. It is a large and rich modeling language targeted primarily at the application domain of electronics hardware.

The general purpose software modeling language, Unified Modeling Language (UML), has also been used for describing system models, in particular the profiles MARTE and xtUML can be mentioned. Marte [34] is a profile targeting real-time and embedded systems specifically and xtUML [35] defines semantics to make UML blocks executable. System dynamics can be described using state machines in UML. OMG SysML [2] is another extension of UML for systems modeling.

3. Tool Support for Cyber-Physical Modeling

The industry is under pressure to shorten product development cycles and time-to-market, while optimizing performance and providing safety. In order to meet these constraints, the availability of tools that support the developers in the workflow, helping them track, debug, and optimize their models is crucial. In this section we review the state-of-the-art of different aspects of tool support for model based development.

3.1. Debugging

Debugging support when modeling cyber-physical systems is particularly important since it is usually difficult to get the correct result with the first attempt to model a system. Commonly implemented features include inspection and plotting of result-files: the ability to view the inputs and outputs of a model and how the outputs change over time. This is often sufficient, but if the model does not simulate, for example, due to numerical errors, additional tool support is needed. Such tool support provides traceability of information from a high-level model all the way through the transformation stages down to the generated code so that error messages from the executing compiled program can be linked from numerical run-time errors to specific equations in the original source model.

All three environments, OpenModelica, Acumen, and Bloqqi, provide access to the result file simulation data. Plotting of variables shows the simulation as a function of time.

The Acumen IDE provides access to the entire state at all times during the simulation in the form of a table. In addition, plotting support is available. Finally, the easy-to-use support for 3D animation facilitates visualization of complex model aspects in the simulation. The 3D visualization feature is also a powerful tool for debugging.

Bloqqi programs can be debugged in a web user interface, where inputs, outputs and state variables can be examined. If the user wants to debug parts of a computation, he or she can make these parts visible in the interface by introducing state variables. It is also possible to debug Bloqqi programs in an ordinary C debugger, since the Bloqqi compiler generates C code for program execution, but there might be difficulties in recognizing the low level generated code.

OpenModelica provides both an equation model debugger and an algorithmic code debugger, a performance analyzer, as well as 3D visualization (Section 3.3).

The equation model debugger [36], (Figure 17), provides capabilities for debugging equation-based models, such as showing and explaining the symbolic transformations performed on selected equations on the way to executable simulation code. It can locate the source code position of an equation causing a problem such as a run-time error, traced backwards via the symbolic transformations.

The algorithmic code debugger (Figure 18) provides traditional debugging of the algorithmic part of Modelica, such as setting breakpoints, starting and stopping execution, single-stepping, inspecting and changing variables, inspecting all kinds of standard Modelica data structures as well as MetaModelica data structures such as trees and lists.

By using performance profiling analysis it is possible to detect which equations or functions cause low performance during a simulation. The OpenModelica profiler [37] depicted in Figure 19 uses compiler-assisted source code instrumentation to improve a certain aspect of the clock before and after executing each equation block or function call.

Associated with each call is a counter that keeps track of how many times this function was triggered for the given time step. Similarly, each call is associated with clock data—one variable for the total time spent in the block for all time steps and one variable for the total time spent in the block for the current time step.

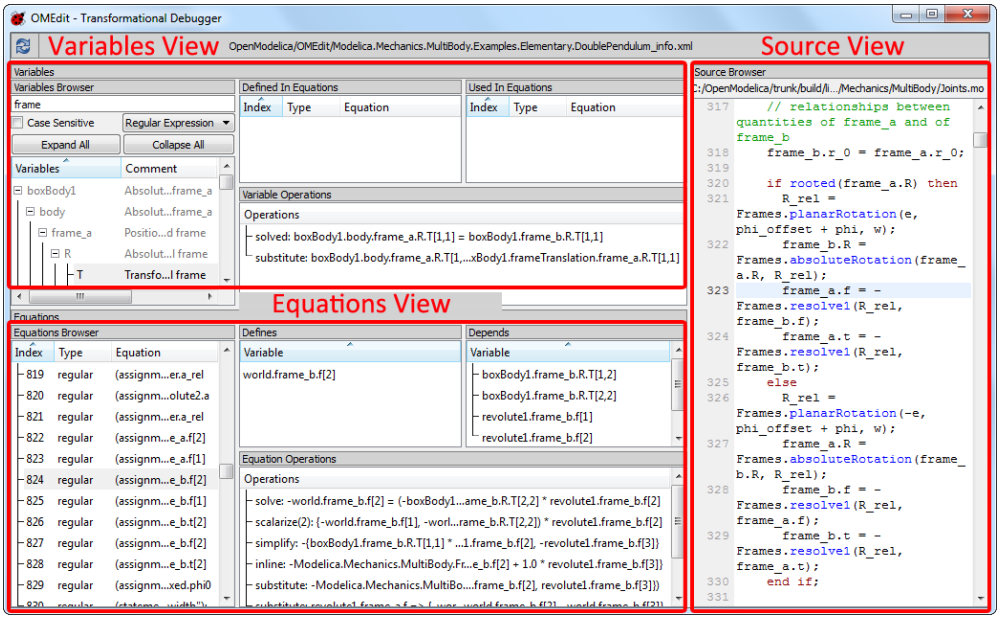


Figure 17. OpenModelica equation model debugger.

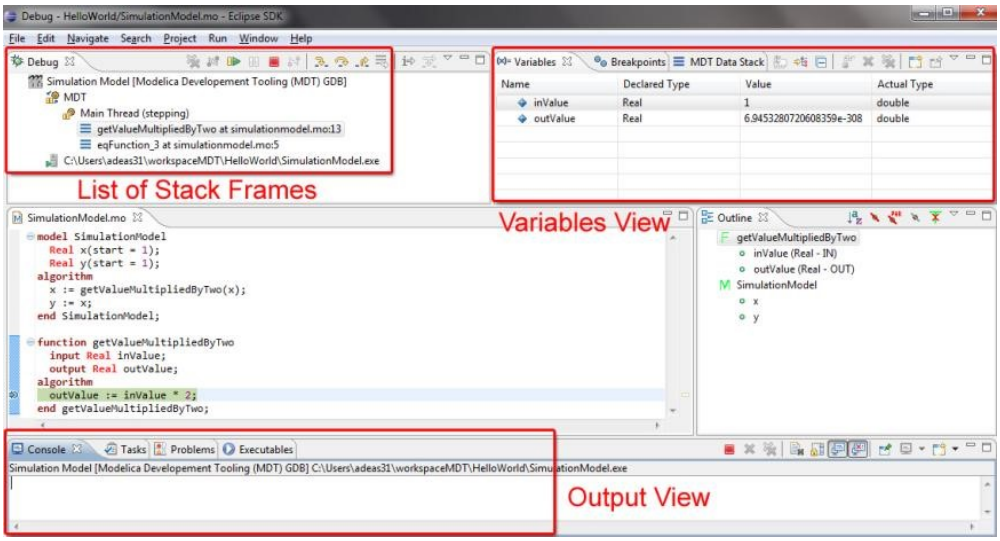
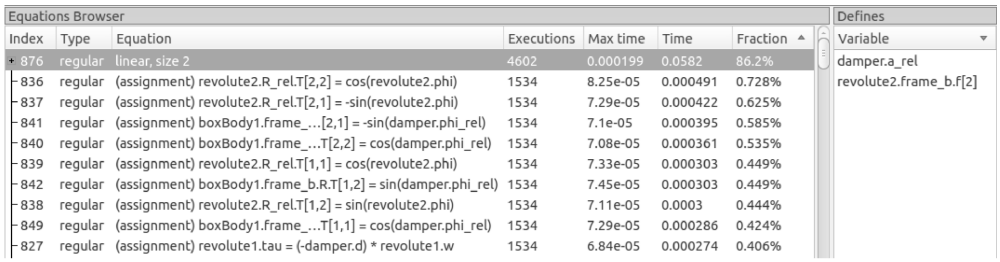


Figure 18. The OpenModelica algorithmic code debugger viewed from the MDT Eclipse plug-in. A breakpoint has been set in the function, which is called from the small model SimulationModel.



Equations Browser						
Index	Type	Equation	Executions	Max time	Time	Fraction
876	regular	linear, size 2	4602	0.000199	0.0582	86.2%
836	regular	(assignment) revolute2.R_rel.T[2,2] = cos(revolute2.phi)	1534	8.25e-05	0.000491	0.728%
837	regular	(assignment) revolute2.R_rel.T[2,1] = -sin(revolute2.phi)	1534	7.29e-05	0.000422	0.625%
841	regular	(assignment) boxBody1.frame_...[2,1] = -sin(damper.phi_rel)	1534	7.1e-05	0.000395	0.585%
840	regular	(assignment) boxBody1.frame_...T[2,2] = cos(damper.phi_rel)	1534	7.08e-05	0.000361	0.535%
839	regular	(assignment) revolute2.R_rel.T[1,1] = cos(revolute2.phi)	1534	7.33e-05	0.000303	0.449%
842	regular	(assignment) boxBody1.frame_b.R.T[1,2] = sin(damper.phi_rel)	1534	7.45e-05	0.000303	0.449%
838	regular	(assignment) revolute2.R_rel.T[1,2] = sin(revolute2.phi)	1534	7.11e-05	0.0003	0.444%
849	regular	(assignment) boxBody1.frame_...T[1,1] = cos(damper.phi_rel)	1534	7.29e-05	0.000286	0.424%
827	regular	(assignment) revolute1.tau = (-damper.d) * revolute1.w	1534	6.84e-05	0.000274	0.406%

Defines
Variable
damper.a_rel
revolute2.frame_b.f[2]

Figure 19. The OpenModelica performance profiler showing which sets of equations use the biggest fraction of the simulation time (model: DoublePendulum).

3.2. Variant Handling and Traceability

Modeling and simulation tools have become increasingly used for industrial applications. Such tools support different activities in the modeling and simulation lifecycle, like specifying requirements, model creation, model simulation, FMU export, model checking, and code generation. However, the heterogeneity and complexity of modern industrial products often require special purpose modeling and simulation tools for different phases of the development life cycle. Seamless exchange of models between different modeling tools is needed in order to integrate all the parts of a complex product model throughout the development life cycle.

During system development several model versions might be created for different system components.

As these models evolve in time, new variants are created to:

- Improve a certain aspect of a model,
- Provide less (faster simulation but less accurate) or more detail (slower but more accurate).

It is important that tools support variant management to help the user analyze and compare variants. Once a model under development is good enough to satisfy the requirements, a tool should be able to ensure that new variants of the same model also satisfy these requirements.

Traceability is another important aspect of any system development. Traceability can occur at different system levels:

- Tracing system requirements and system models to simulation traces that are used to check requirement validity,
- Tracing model elements to low-level representation for debugging,
- Tracing model variant evolution during the development cycle.

Dependency analysis is one form of traceability, used for instance in the JModelica compiler [38]. In [36], the OMC compiler supports traceability in terms of tracing generated C code back to the originating Modelica source code; thus, is mostly used for debugging.

During the past decade, the Open Services for Lifecycle Collaboration (OSLC) specifications [39] have emerged for integrating development lifecycle tools using Linked Data [40–42]. OSLC is an open source initiative for creating a set of specifications that enables integration of development life cycle tools (e.g., modeling tools, change management tools, requirements management tools, quality management tools, configuration management tools) [39]. The goal of OSLC is to make it easier for tools to work together by specifying a minimal protocol without standardizing the behavior of a specific tool.

The most recent work in [43,44] has successfully demonstrated traceability support within the INTO-CPS tool-chain [45,46] or OpenModelica, based on OSLC. This includes mostly tracing of the requirements and connecting them with the models, FMUs, the simulation results, the produced code and test results.

The primary scope for traceability in the INTO-CPS project was the demonstration of the basic traceability tasks across the tool-chain with the following goals [47]:

- Checking the realization of requirements in models.
- Enabling collaborative work by connecting artifacts and knowledge from different users.
- Decreasing redundancy by connecting different tools to a single source for requirements and allowing a system-wide view that is not solely limited to single tools.

The Provenance (PROV) [48] and OSLC standards presented in [49]) are used to support traceability activities. PROV is a set of documents built on the notation and relation of entities, activities, and agents. The design and architecture of traceability-related tools has been developed in [47] and is shown in Figure 20. Any modeling tool written in any programming language can use these traceability standards to support the traceability of activities performed within the tool and interact with other tools.

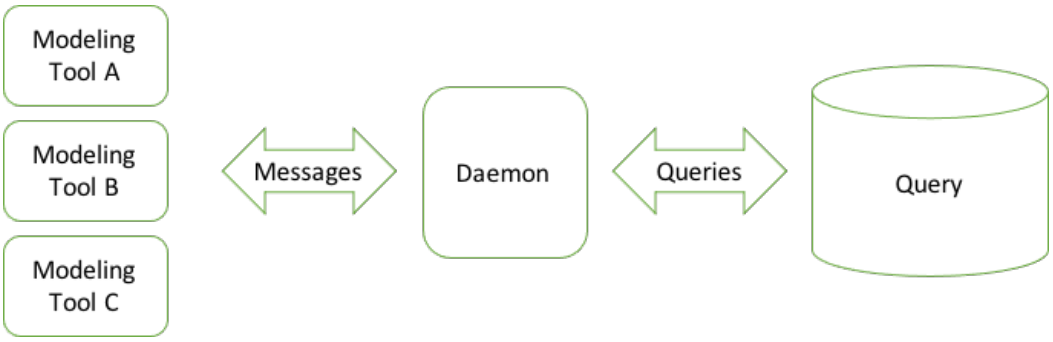


Figure 20. Schematic architecture of traceability-related tools.

All artifacts and actions are versioned and traced, which means that it is possible to use the traceability information and the Git repository to provide impact analysis from different perspectives, i.e., what is affected (with regards to verification) by a change in different parts of the model or in the requirements placed on the model. One could even start from two different verification reports and highlight their differences with respect to all involved artifacts.

3.3. Graphical Interfaces and 3D Visualization

3.3.1. Web-Based User Interfaces

Analogous to portability of models, the portability of tools themselves is also an important issue. Providing web-interfaces to simulation tools is a way to provide consistent, platform independent tool access to the user.

OMWebBook (part of OpenModelica) is a web-based executable extensible Modelica book that allows textual editing of models and running of simulations in an interactive electronic book from a web page [50,51]. All the code is executed on the server, therefore no installation is necessary on the user's computer.

There is currently ongoing work on a more general OpenModelica web-based interface also including graphical model editing. It consists of two major parts. One is the OMWebService that provides an interface for querying the model information and performing the simulation. The other part is OMWebEdit, which is a new OpenModelica web-based editor. The implementation is in its early phases with plans for a collaborative approach with shared work-spaces.

Regarding Bloqqi, a running Bloqqi program can be accessed through a web interface, where input values can be set and output values can be read real-time. These values are exposed as API endpoints (in JSON format), from which a standard user interface is derived automatically using Javascript. This interface shows all values in a list, where input values have HTML input fields that can be used to change values. The user can also easily specialize the user interface by specifying custom Javascript code that gets notified when values change. This makes it very easy to use arbitrary Javascript libraries for custom rendering.

3.3.2. 3D Visualization

The Modelica language standard includes definitions of standardized graphical annotations. Some of these annotations can be used to define 3D shapes of physical objects. There are standard annotations for a number of shapes such as cylinders, rods, etc. These can be combined to create more complex visualizations as shown in Figure 21. Arbitrary shapes can also be defined in files and referred to in the annotations. The library Modelica.Mechanics.Multibody has such annotations defined for all model components. This allows a Modelica tool to automatically generate 3D animations, for example, as in Figure 21.

The Acumen environment includes a graphical user interface that has three main panes: model, results, and console (Figure 22). The results pane has three modes: table, plot, and 3D. Support for 3D is built into the Acumen IDE. Both static and dynamic 3D

(animations) are done through special variables such as `_3D` and `_3DView`. The first one is followed by a list of graphical objects, such as `Box` (center, orientation, color, transparency), `Cylinder` (center, orientation, ...). Graphical objects include both basic shapes and a way to include OBJ files. Arguments to parameters such as "center" and "orientation" can include arbitrary expressions, including variables. When the values of these parameters change dynamically, the result is an animation. Other parameters such as `_3DView` enables controlling the view point both statically and dynamically in the same manner as for _3D.

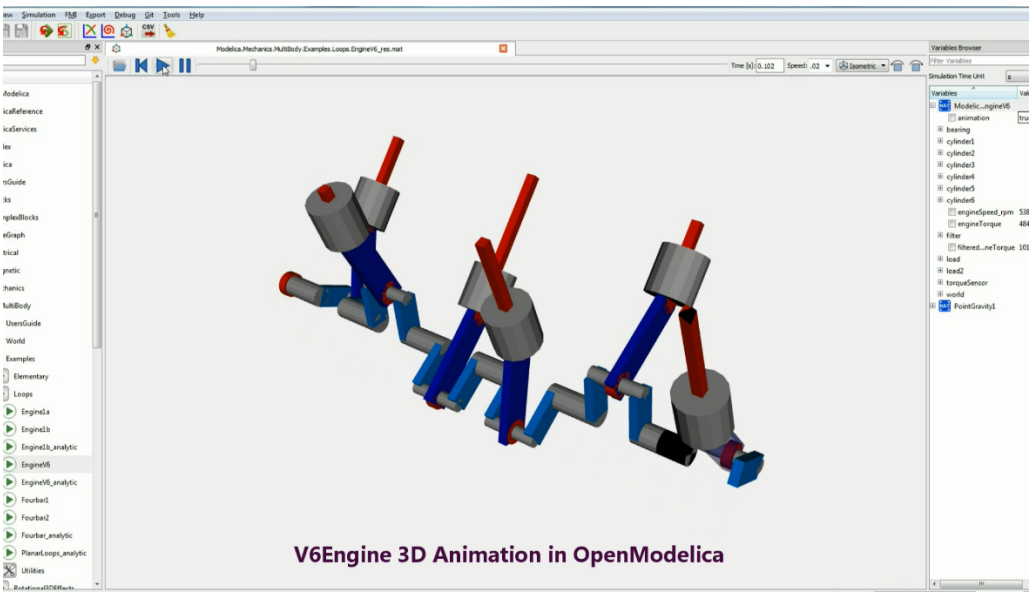


Figure 21. EngineV6 3D animation in OpenModelica.

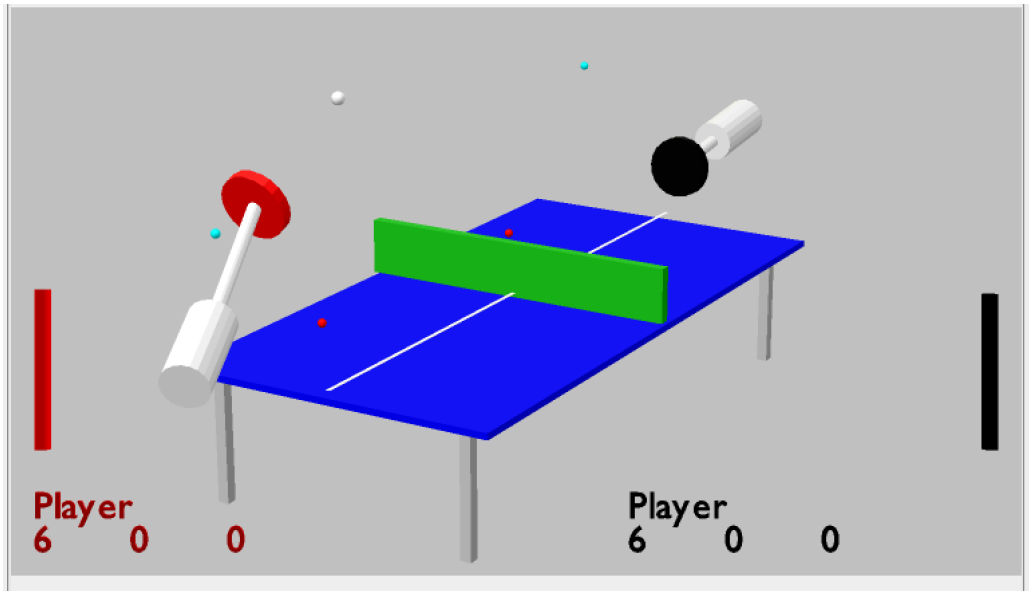


Figure 22. A ping pong playing model using balls and bats. Acumen 3D visualization.

3.4. Interoperability

A frequent problem in large industrial projects is that although component level models are available, it is a big hurdle to integrate them into large system simulations. This is because different development groups and disciplines, for example, electrical, mechanical, hydraulic, as well as software, often use their own approaches and special purpose tools for modeling and simulation.

To improve the interoperability of behavioral models, the MODELISAR project developed the FMI as a standardized exchange format for simulation units. Components can be exported as FMUs with a standardized interface for (co-)simulation. The FMI standard describes two different variations of mock-up units: ME-FMUs (model exchange) expose the ODE structure, which allows tight integration and CS-FMUs (co-simulation) expose a white-box interface including its own potentially special solver methods. A master algorithm can then integrate these FMUs into a functional mock-up integrated simulation model using a suitable master algorithm for coupling the individual simulation units.

Acumen provides support for connecting to other simulation tools via a socket interface. This interface has been used, for example, for connecting with external visualization tools.

OMSimulator is an FMI-based co-simulation tool and recent addition to the OpenModelica tool suite. It supports large-scale simulation and virtual prototyping using models from multiple sources utilizing the FMI standard. It is integrated into OpenModelica but also available stand-alone, i.e., without dependencies to Modelica models or technology.

OMSimulator provides an industrial-strength open source FMI-based modeling and simulation tool. Input/output ports of FMUs can be connected, ports can be grouped to buses, FMUs can be parameterized and composed, and composite models can be exported according to the SSP standard. Efficient FMI-based simulation is provided for both model-exchange and co-simulation.

Bloqqi programs can be exported as co-simulation FMUs [52], which allows them to be tested with simulations of the processes they control. This enables Bloqqi programs to be tested with models specified in any tool that supports co-simulation FMUs. For example, we have successfully exported Bloqqi programs and Modelica models as FMUs, composed them with FMU and SSP tools, and simulated the composition.

Results of simulating the ball and bat example that was presented in Section 2 using Modelica, Acumen, Bloqqi and co-simulation via FMI are illustrated in Figure 23. The Modelica simulation provides the most accurate results, because it uses a sophisticated approach to integrate the discrete model simulation with the continuous physics simulation. The Bloqqi results are identical because only the control part was replaced with the Bloqqi model illustrated in Figure 12.

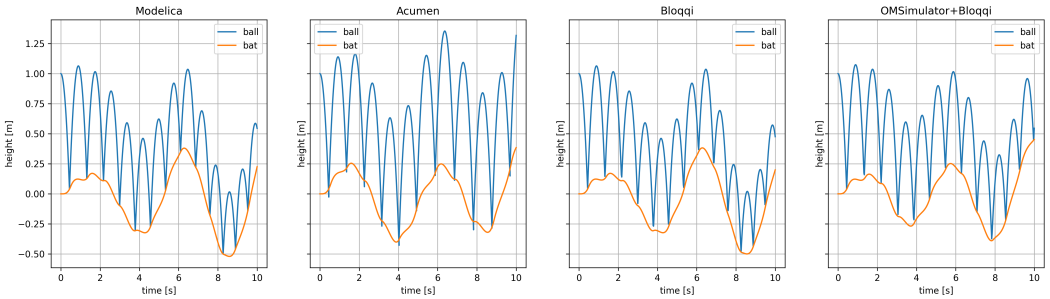


Figure 23. The simulation of the ball and bat example in different tools.

The differences in the graphs between Modelica/Bloqqi and Acumen are due to a different approaches in event handling. Acumen does not try to solve for the exact time point when events occur as Modelica does. Instead, it relies on the first integration step that triggered the event. Those differences are usually small, but especially for models with unstable solutions, the results can quickly vary significantly. It also explains why the ball falls through the bat before bouncing back.

The OMSimulator+Bloqqi simulation is using the co-simulation approach to simulate composite models originating from different sources—Modelica and Bloqqi in this case. The different sub-models are simulated independently and synchronized at communication time-points. This typically introduces small errors. Controlling these errors is a major challenge in co-simulation applications.

4. Model Compiler Implementation from Formal Specifications

Modeling languages such as Modelica, Acumen, Bloqqi, and others usually need model compilers to translate models to a form suitable for final compilation to machine code or for interpretation. Figure 24 depicts typical translation and execution for Modelica tools, including OpenModelica. The Bloqqi compiler has a similar architecture, but with a simpler backend since there are no equations and equation sorting. For Acumen, interpretable code in an intermediate form is generated instead of C code.

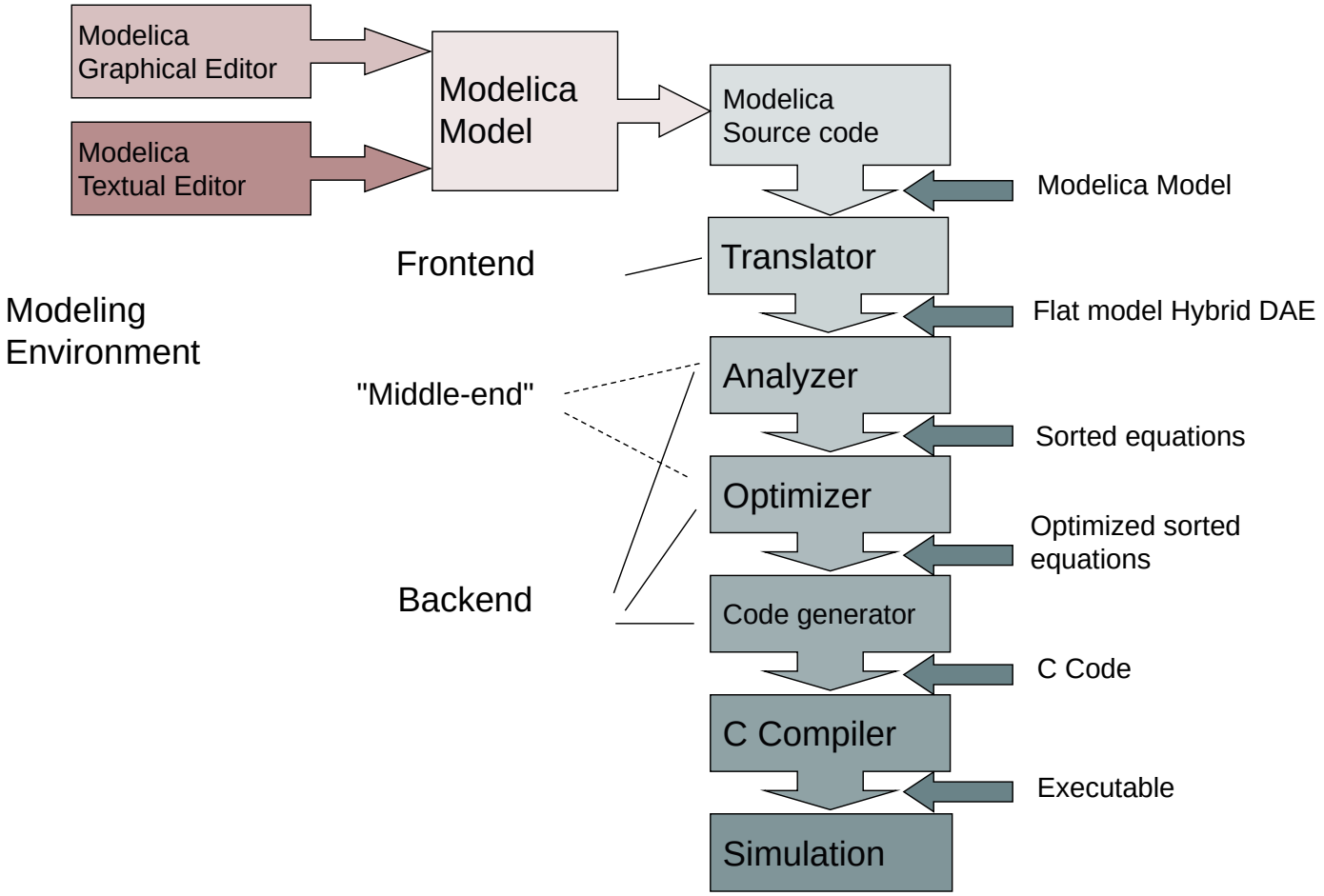


Figure 24. Typical translation stages for a Modelica model to executable form and simulation.

A Modelica compiler translates high-level object-oriented Modelica models to a flat set of hybrid differential-algebraic equations from which an executable simulation program is generated. This translation is highly complex, involving many phases and analyses [53]. Furthermore, some of the analyses are interdependent. For example, full name resolution depends on performing instantiation of compile-time objects, but instantiation also depends on parts of the name resolution. A Modelica model is defined in terms of classes containing equations and definitions. The semantics, i.e., the meaning of such a model, is defined via translation of classes, instances, connections, and functions into a flat set of constants, variables and equations. Equations are sorted and converted to assignment statements when possible. Strongly connected sets of equations are solved by calling a symbolic and/or numeric solver.

It is typically very costly and error-prone to implement compilers and other tooling for programming and modeling languages, in particular if done by hand in standard imperative implementation languages. By employing higher-level implementation languages and techniques, this cost can be substantially reduced, and a higher code quality obtained.

Declarative compilation technology has proven highly successful in order to deal with these complexities [38,54]. With declarative programming, computation results can be expressed using side-effect-free functions and equations, rather than by imperative side-effectful code. This allows programming to be done at a higher level, closer to the problem at hand, and in a less tangled way, promoting extensibility of the tools. Furthermore, declarative languages have the potential for supporting automatic incrementalization and parallelization, which is useful for improving performance of the tools.

Major parts of the Bloqqi compiler and editor are generated from reference attribute grammars [55]. The syntactic analysis parts of both systems have been generated from BNF grammar specifications. The Acumen interpretive implementation has been implemented in the Scala language, which also allows a high level of abstraction.

4.1. Reference Attribute Grammars

RAGs [55] is a declarative compilation technology where a compiler is implemented in the form of attributes and equations associated with the node types of an abstract grammar. The attributes are properties of the nodes in a syntax tree, for example name bindings, types, and generated code. The equations define the values of the attributes, and are solved automatically by an attribute evaluation engine. RAGs extend plain attribute grammars [56] by allowing attributes to be references to nodes, thereby supporting the declarative construction of graphs over the syntax tree; for example, name binding graphs and call graphs. This makes RAGs especially suited for developing compilers, as many compilation subproblems work on graphs. RAG evaluation engines work using on-demand evaluation and memoization for efficiency, and algorithms are available also for incremental and concurrent evaluation [57,58].

The Bloqqi compiler and large parts of the Bloqqi editor are implemented in RAGs, using the JastAdd metacompiler [59]. Because of the declarative programming using attributes and equations, it is easy to extend a RAG-based compiler with both new language constructs and new computations. For Bloqqi, this extensibility is used for separating different parts of the compiler and the editing tooling into separate modules. Each module has access to the syntax tree, and can extend it with attributes and equations, as well as accessing attributes defined by other modules. Figure 25 shows the main RAG modules of the Bloqqi tooling. The frontend contains the core compiler functionality with abstract grammar, name analysis, and type checking. The backend extends the frontend with modules for generating C code, FMUs, and communication code for the MQTT middleware. For editor support, the frontend is extended with attributes used by the editor. For example, the visualization module computes a visual representation of a block diagram, taking inheritance and connection interception into account. The interaction feedback module computes attributes for giving type-based feedback in the editor, for example, what ports will match when adding a connection between two blocks (as was shown in Figure 13). The wizard generator computes a feature selection wizard for a selected block in the edited diagram (like the one shown in Figure 14). The attribution engine of JastAdd performs all attribute computations automatically, and attributes are updated consistently as the model is edited.

RAGs are also used for implementing large parts of the JModelica.org Modelica compiler [54].

4.2. Operational Semantics

The OpenModelica effort started in 1997 by developing a formal specification [60] of an early version of the Modelica language. The syntactic part of this specification was expressed in BNF and the semantics part was specified in Operational Semantics, which still is the most popular and widely used semantics specification formalism in the programming language community. It was initially used as input for automatic generation of the Modelica translator implementations, which are part of the OpenModelica environment. The RML compiler generation tool (our implementation of Operational Semantics) [61] was used for

this task to generate compilers in C with a performance close to hand-written compilers. The RML system has also been used to generate a Java 1.2 compiler, a C large subset compiler, and a Pascal subset compiler.

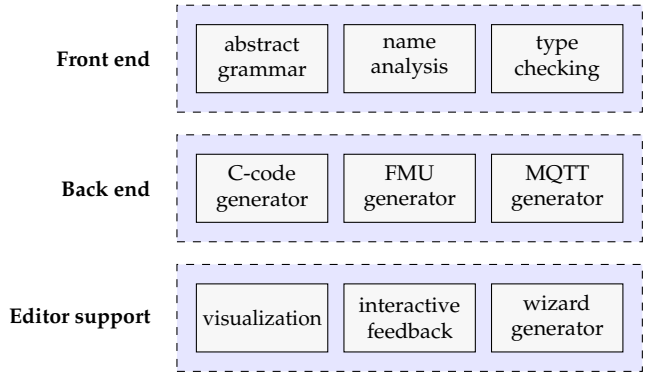


Figure 25. Key RAG modules in the Bloqqi compiler and editor.

However, inspired by our vision of integrated interactive environments with self-specification of programs and data, and integrated modeling and simulation environments, in 2005 we designed and implemented an extension to Modelica called MetaModelica [62–64]. This was done in order to support language modeling and specification (including modeling the language itself), in addition to the usual physical systems modeling applications, as well as applications requiring combined symbolic-numeric capabilities. Modeling the semantics in Modelica itself was also inspired by functional languages such as Standard ML [65], and OCaml [66]. Moreover, it was an investment into a future Modelica becoming a combined symbolic-numeric language such as Mathematica, but more efficient and statically strongly typed.

This language extension has a backwards compatible Modelica-style syntax but was initially implemented on top of the RML compiler kernel. The declarative specification language primitives in RML with single-assignment pattern equations, potentially recursive union types of records and match expressions, fit well into Modelica since it is a declarative equation-based language. In 2006 our whole formal specification of Modelica static and translational semantics, at that time about 50,000 lines, was automatically translated into MetaModelica syntax. After that, all further development of the symbolic processing parts of the OpenModelica compiler (the run-time parts were mainly written in C), has been done in MetaModelica. Later, the compiler was made to compile itself, so-called bootstrapping [67]. More recently, the semantics parts of the frontend analysis of the compiler has been re-structured and re-written, to achieve better modularity, adapted to recent semantic changes of the Modelica language. This also gave a significantly higher compilation speed of the produced compiler.

4.3. Standardization

To increase interoperability, to ensure that the tools and languages can be maintained over time, and to encourage cooperation between tool developers and the industry, it is important to rely on open standards as much as possible. This section covers the standardization status of the languages discussed in this paper.

The Modelica language, Functional Mock-up Interface (FMI) and Structure and System Parameterization (SSP) are open standard specifications developed by the Modelica Association that support the development of CPS. Implementations of these open standards in tools help the industry to lower costs and prevent vendor lock-in. OpenModelica is an open source implementation whereas proprietary commercial implementations are listed in Section 2.5.

OpenModelica also supports interfacing with the Open Services for Lifecycle Collaboration (OSLC) standard, which facilitates collaboration with other tools that support this standard [43].

VHDL-AMS, introduced in Section 2.5, is based on the IEEE standard 1076-1993 and is used in the industry with a large number of libraries targeted particularly at the electronic domain.

The Bloqqi language was developed in collaboration with ABB. ABB's software is proprietary, whereas Bloqqi is described in papers and the software is open source.

Acumen was developed initially in close collaboration with Schlumberger to enable virtual testing of embedded microcontrollers running on horizontal drilling tractors. Such drilling tractors are essentially robotic devices. More recently, case studies focused on safety in the automative vehicle domain. As the core primitives of the language are intended to correspond directly to mathematical notions, the primary focus has been ensuring the connection between implementations and mathematical meaning (i.e., semantics) rather than on standardization. The implementation includes a reference implementation that uses traditional simulation methods as well rigorous simulation [68]. Rigorous simulation builds on top of interval arithmetics to create more specialized set representations called enclosures. While the use of set representations seems critical for ensuring correctness, it also points out the unique challenges of correct simulation, such as the need to handle Zeno behaviors correctly. Acumen's implementation builds on an efficient (third party) interval arithmetic library, but this by itself is far from sufficiently from remaining within a constant execution time factor of traditional simulation methods. In particular, the runtime cost comes not just from interval computation (small part), but rather from the set representations (enclosures) and the algorithms needed to calculate on sets in terms of one another. Especially due to the hybrid systems nature of models, which means that the computations involve conditionals (such as zero crossing), these calculations often require branch and bound computations. Such calculations can quickly generate a huge number of threads (possibilities) that can only be controlled/reduced using what is so far ad hoc methods.

5. Testing, Verification, and Fault Analysis

5.1. Test-Based Requirement Verification in Modelica

In recent years the need for a more formal requirement verification process and for language and tool support for this has been increasingly recognized by the cyber-physical modeling community. Several efforts on language and tool support have been made and are ongoing in this area [69,70].

Having both system requirements and system models in the same language reduces the semantic gap in the terminology used between the requirement verification engineers and the system modelers, simplifies the modeling effort and allows for an automated combination of the requirement models. The resulting models can be simulate to check whether the model violates any requirements (Figure 26).

The ongoing project EMBRACE [71] is focusing on developing a requirement specification language, partly based on Modelica and the Extended Temporal Language [72], which can be used in conjunction with Modelica to verify system specifications. This language will be compiled to standard Modelica, which will also enable co-simulation with models in other languages through FMI.

Modelica can also be used for verification of models via Monte Carlo simulation either in Modelica itself, (Figure 27), modeling the failure modes through state machines with stochastic transitions [73], or through interfacing with Julia to control the simulation [74].

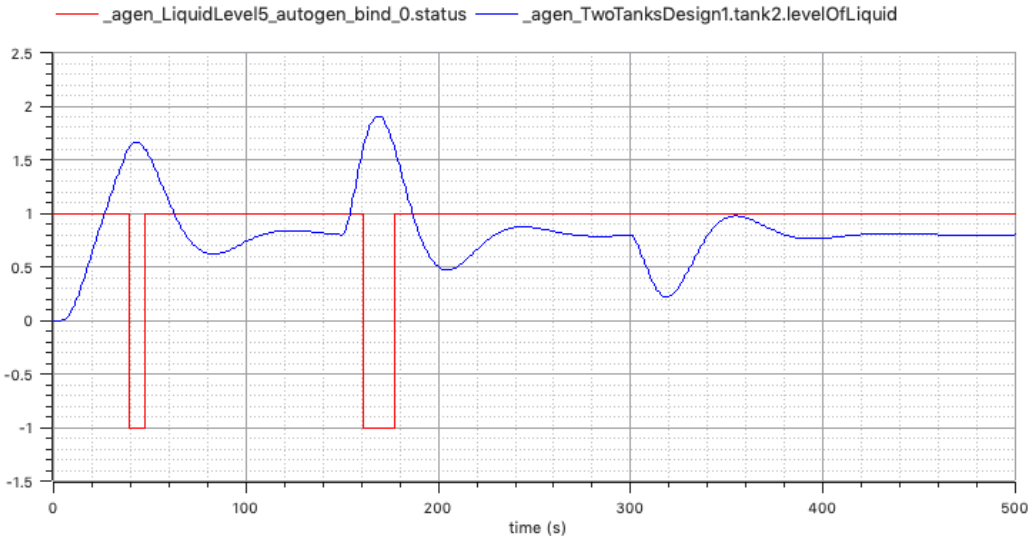


Figure 26. Simulation-based requirement verification for the two-tanks example in OpenModelica. When the volume of water in the tank rises above 80% of the tank volume the requirement is marked as violated.

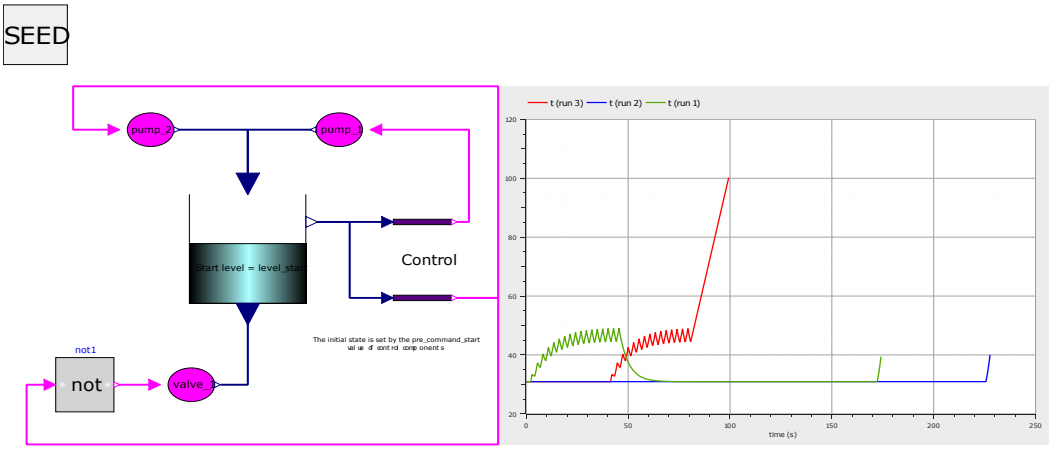


Figure 27. A model of a temperature controlled room, which includes different types of failures that occur with specified probabilities. The graph on the left shows examples of different faults that occurred during simulation in green and blue and a successful temperature control scenario in red.

5.2. Fault and Dependability Analysis

The purpose of reliability, and more generally, of dependability studies is to evaluate non-functional behavior, i.e., to calculate probabilities of undesirable events such as the failure of the mission of a system, or to estimate the probability distribution of quantities like: total production on a given time interval, maintenance cost, number of repairs, etc. Usually, dependability studies are performed with dedicated methods and tools, based on discrete (and often even Boolean) models of systems: fault trees, Markov chains, Petri nets, etc. EDF (Electricité du France) designed the Figaro modeling language in 1990 [75]. This language generalizes all the above-mentioned dependability-related models, and allows casting knowledge into categories of systems in libraries.

In order to benefit from this type of analysis a prototype in OpenModelica for coupling Modelica models with their Figaro counterpart has been developed [76].

The reliability analysis performed on the Figaro model coupled to a Modelica model can then be used to identify potential issues (e.g., critical components). This information can be fed back into the Modelica simulation (e.g., investigate in more detail the effect of the failure of a critical component).

Starting from a pure Modelica model designed for physical simulation, annotated with some Figaro-specific information, one can obtain a Figaro model by extracting the objects relevant to the dependability analysis (not all the objects in the physical representation are necessarily used in the dependability analysis) and their inter-relations from the model and associating them to a well suited library in Figaro.

5.3. Test Selection for Regression Testing

Regression testing is an important activity when developing software, by running tests when the software is changed to detect if previous working functionality still works. Running tests for Modelica usually takes a long time, involving long compilation and simulation times. We have developed a technique for reducing testing times by analyzing the Modelica source code to conservatively detect which tests are actually affected by a change [77], and to only run the estimated affected tests. This analysis is based on dependencies between classes, which forms a dependency graph. The selection of tests is then computed using reachability analysis over this graph.

5.4. Correctness of Simulation

Modeling and simulation technologies in general, including those for cyber-physical systems, can exhibit several different problems:

- A tool may produce results that are misleading or inaccurate, given the models they are supposed to be simulating.
- A tool may enter an infinite loop or produce an error on a given model when an unexceptional result is expected.
- Different tools (or versions of the same tool) may produce different results or behave differently on the same model.

All of these types of problems can be viewed as issues of correctness, thereby allowing us to apply the principles and methods of formal methods and semantics to the development of such tools. Formal methods focus on establishing the correctness of given artifacts (such as a tool). Semantics techniques focus on specifying the correctness of language processing tools, whereby using a language we can create sets of strings as simple as sequences of zeros and ones or more structured entities such a programming or modeling languages.

A common source of difficulty for simulation tools is that the traditional numerical methods generate discrete approximations to trajectories (computed variables as functions of time). This, by itself, would have been acceptable, if it were not for two facts. The first is that there is generally no relation between this discrete approximation and a formally specified ideal result. The second is that most methods store this discrete approximation and use it for the rest of the computation with an implicit assumption that this discrete approximation is exact. The result is that deviations from an ideal solution can quickly grow. This is seen in practice in the form of different implementations of the same tool producing radically different results, corresponding to the first and third problems listed above. On the other hand, there are error-controlling numerical solution methods that are typically used, but they are not completely fail-safe.

Intuitively, formal semantics for such methods take the form of set-based computations where the answer sets can be made increasingly more precise (smaller). Ideally, in cases where the idealized answer should be a single real number, the limit of such a sequence of sets would be a singleton set. In practice, there are several well-justified situations where the limit may not be a singleton element, including non-determinism and artifacts introduced by the choice of semantics. A drawback is that such methods may be several orders of magnitude slower than floating-point numerical computations. In general, we expect that such formal semantics will play an increasingly important role in the design of tools for cyber-physical systems.

5.5. Control Systems Applications

For a long time Modelica has been used to model control systems using the Blocks library in the Modelica Standard Library. It contains several controller models including the well-known P, PI, and PID controllers.

For increased performance and more precise control and compile-time checking, clocked synchronous language features have been introduced in the Modelica 3.3 and later language specification, and a library for clocked synchronous control systems has been developed [78].

The Modelica Device Drivers library (MDD) provides an interface between Modelica and hardware devices using external C functions. This library can be used to write controller code for embedded devices in Modelica and then generate minimal footprint C code that can be executed on these devices [79].

Interactive simulation has been identified as a desirable feature by several industrial users of OpenModelica. It is implemented via the OPC UA protocol and allows to modify variable values during the simulation [80].

Bloqqi has been developed in collaboration with ABB, which has its own language called ControlBuilder that their customers use to create control systems in the process industry. ControlBuilder is based on the language Function Block Diagram in the IEC-61131 standard. Bloqqi was created to experiment with language constructs for code reuse in this context. Periodic sampling and directed data-flow thus come from ControlBuilder and Function Block Diagram. It is also possible to have external functions in Bloqqi programs that are implemented as C functions.

6. Discussion and Future Work

In this section we highlight the trends that impact future development directions of the tools presented in the paper and discuss necessary improvements.

One of the common factors impacting the development of open-source tools presented in this paper is the lack of stable funding. This makes it hard to plan long term development and research. Development is often funded through master or doctoral theses, which means that the long-term goals have to be balanced with academic requirements. One solution to this is to secure more funding from industrial users.

In the remainder of this section we discuss the trends we have identified that impact CPS development.

6.1. Digital Twins

The concept of "digital twin" is closely related to modeling and simulation. In [11] we have the following definitions

- A model of a system is anything an "experiment" can be applied to in order to answer questions about that system.
- A simulation is an experiment performed on a model.

Artifacts represented by mathematical models in a computer are often called virtual prototypes. The process of constructing and investigating such models is virtual prototyping. Many people view a digital twin as a virtual prototype represented digitally. It is a virtual model that can be created in a computer, simulated, analyzed, and tuned before building a physical counterpart. This is typical for model-based development of industrial products, and probably the most common application of the concept. Another interpretation is that the digital twin should interact in real-time with the physical world, as in the following definition of virtual/digital twin from [81]:

> *"A virtual twin is a real time virtual representation of a product, process, or a whole system that is used to model, visualize, predict and provide feedback on properties and performance, and is based on an underlying digital thread."*

Yet another interpretation is more AI-inspired [82], viewing a digital twin as a living, intelligent, and evolving model, being a virtual counterpart of a physical entity, and

following the lifecycle of its physical twin. There should be continuous synchronization and communication between the two twins. This view is, for example, relevant for long-running autonomous systems such as robots.

6.2. Dynamic System Modeling

The adoption of IoT technology means that CPS applications have become highly variable with entities connecting and disconnecting continuously and with a dynamically changing structure. To be able to simulate this type of system, tools need support for variable data structures as well as dynamic recompilation. Research on these topics is ongoing in the EMBRACE project [71].

Modeling complex and large systems is an error-prone activity, and as discussed in Section 3, good tool support is very important. In recent years the OpenModelica tool suite has gained in stability, performance, and inter-operability. However, progress is needed in the ease of use and debugging support for modern CPS modeling tools.

6.3. Cloud-Based Computing

Another big trend is distributed and cloud-based architectures. The importance of this trend is two-fold. As systems are becoming increasingly complex, with hundreds of thousands of equations, integrating support for cloud-based computations will enable efficient simulation. At the same time, providing support for cloud-based development tools such as OMEdit will increase collaboration in the current context where many teams are split across different geographical locations. Constructing low-latency interactive cloud-based development tools for CPS is costly, however, and generation of such tools from high-level specifications is an important research area.

7. Conclusions

CPS modeling is a relatively new domain, compared to traditional software development. Although there has been progress in both methodology (such as efforts to formalize language and model format specifications, graphical model editing) and in tooling (such as scripting support, code-completion), there is still a lot of work to be done on the integration of the modeling tools into a full product development cycle, with research ongoing in areas such as requirement verification, testing, and distributed compilation.

In this paper we have identified the following general requirements for modeling environments:

Language specification In order to have true tool interoperability, it is important that all the tools behave in the same manner. Therefore, an unambiguous (preferably an open standard) textual language definition is necessary.

Extensible tools As complexity increases, or as modeling applications become more diverse, additional features need to be added to the tools, for example, debugging and interactive simulation. In order to achieve this it is important to design the tools to be extensible. Generation from high-level specifications can help to achieve this goal.

Open standards In order to enhance the interoperability between different formalisms it is important to have common standards for model exchange and co-simulation, such as FMI.

In this paper we identify the main enablers to interoperability: building tools based on standards and relying on textual language specification. This is key to combining complementary tools and building a comprehensive CPS modeling environment. Openness also facilitates dissemination of research results and thus encourages collaboration and further tool development.

We have presented, with the help of examples, the main characteristics of three different modeling languages: Modelica, Bloqqi, and Acumen and have covered their state-of-the-art in the areas outlined above.

Author Contributions: All the authors participated in the following stages: Conceptualization, methodology, software, writing of the original draft, review and editing. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the Strategic Research Area ELLIIT, funded by the Swedish government, and by Vinnova in the ITEA OPENPROD, MODRIO, OPENCPS, EMPHYSIS and EMBRACE projects. Support has also been received from the Swedish Strategic Research Foundation (SSF) in the LargeDyn project. The OpenModelica development is supported by the Open Source Modelica Consortium.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. John, K.H.; Tiegelkamp, M. *IEC 61131-3: Programming Industrial Automation Systems Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*, 2nd ed.; Springer: Berlin, Germany, 2010.
2. OM Group. System Modeling Language Specification v1.3. Available online: https://www.omg.org/spec/SysML/1.3/PDF (accessed on 20 January 2020).
3. Modelica Association. Modelica-A Unified Object-Oriented Language for Physical Systems Modeling-Language Specification Version 3.4. Available online: https://www.modelica.org/documents/ModelicaSpec34.pdf (accessed on 20 January 2020).
4. MODELISAR. Functional Mock-Up Interface, Version 2.0. Interface Specification. 2017. Available online: https://fmi-standard.org/downloads/ (accessed on 20 January 2020).
5. Christen, E.; Bakalar, K. VHDL-AMS—A hardware description language for analog and mixed-signal applications. *IEEE Trans. Circuits Syst. II Analog Digit. Signal Process.* **1999**, *46*, 1263–1272. [CrossRef]
6. Modelica Association. System Structure and Parameterization, Version 1.0. Available online: https://ssp-standard.org (accessed on 20 January 2020).
7. Chen, Y. Integrated and Intelligent Manufacturing: Perspectives and Enablers. *Engineering* **2017**, *3*, 588–595. [CrossRef]
8. Thiele, B.; Beutlich, T.; Waurich, V.; Sjölund, M.; Bellmann, T. Towards a Standard-Conform, Platform-Generic and Feature-Rich Modelica Device Drivers Library. In Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, 15–17 May 2017. [CrossRef]
9. Zeng, Y.; Chad, R.; Taha, W.; Duracz, A.; Atkinson, K.; Philippsen, R.; Cartwright, R.; O'Malley, M. Modeling electromechanical aspects of cyber-physical systems. *J. Softw. Eng. Robot.* **2016**, *7*, 100–119.
10. Fors, N.; Hedin, G. Bloqqi: Modular feature-based block diagram programming. In Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! Amsterdam, The Netherlands, 2–4 November 2016; pp. 57–73.
11. Fritzson, P. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*, 2nd ed.; Wiley-IEEE Press: Hoboken, NJ, USA, 2015.
12. Modelica Association. Modelica Standard Library GitHub Page. 2017. Available online: https://github.com/modelica/Modelica/ (accessed on 20 January 2020).
13. Fritzson, P.; Aronsson, P.; Pop, A.; Lundvall, H.; Nyström, K.; Saldamli, L.; Broman, D.; Sandholm, A. OpenModelica—A Free Open-Source Environment for System Modeling, Simulation, and Teaching. In Proceedings of the 2006 IEEE Conference on Computer Aided Control System Design, Munich, Germany, 4–6 October 2006.
14. Taha, W.; Zeng, Y.; Duracz, A.; Fei, X.; Atkinson, K.; Brauner, P.; Cartwright, R.; Philippsen, R. Developing a first course on cyber-physical systems. *ACM SIGBED Rev.* **2017**, *14*, 44–52. [CrossRef]
15. Moggi, E.; Farjudian, A.; Duracz, A.; Taha, W. Safe & robust reachability analysis of hybrid systems. *Theor. Comput. Sci.* **2018**, *747*, 75–99.
16. Duracz, A.; Moggi, E.; Taha, W.; Lin, Z. A Semantic Account of Rigorous Simulation. In *Principles of Modeling*; Springer: Berlin, Germany, 2018; pp. 223–239.
17. Fors, N.; Theorin, A.; Robertz, S.G.; Hedin, G. Feature-Oriented Control Programming. In Proceedings of the 25th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2020, Vienna, Austria, 8–11 September 2020; pp. 1043–1046.
18. Apel, S.; Kästner, C. An Overview of Feature-Oriented Software Development. *J. Object Technol.* **2009**, *8*, 49–84. [CrossRef]
19. MQ Telemetry Transport. Available online: https://mqtt.org (accessed on 20 January 2012).
20. Brück, D.; Elmqvist, H.; Mattsson, S.E.; Olsson, H. Dymola for Multi-Engineering Modeling and Simulation. In Proceedings of the 2nd International Modelica Conference, Oberpfaffenhofen, Germay, 18–19 March 2002.
21. Modelon. ModelonImpact: System Simulation Software. Available online: https://www.modelon.com/modelon-impact (accessed on 20 January 2020).
22. Altair. solidThinking Activate. Available online: https://www.altair.com/activate/ww.simulationx.com (accessed on 20 January 2020).

23. Wolfram Research. Wolfram System Modeler Documentation and Overview. 2018. Available online: http://www.wolfram.com/system-modeler/ (accessed on 20 January 2020) .
24. Tongyuan, S. MWorks: A Modelica-Based Visual Modeling and Simulation Platform. Available online: https://www.tongyuan.cc (accessed on 20 January 2020).
25. Software, S.P. LMS Imagine.Lab Amesim. Available online: https://www.plm.automation.siemens.com/global/en/products/simcenter/simcenter-amesim.html (accessed on 20 January 2020).
26. ANSYS. Ansys Twin Builder: Digital Predictive Maintainance Software. Available online: https://www.ansys.com/products/systems/ansys-twin-builder (accessed on 20 January 2020).
27. GmbH, I. SimulationX Modeling and Simulation Tool. Available online: https://www.simulationx.com/ (accessed on 20 January 2020)
28. Maplesoft. MapleSim—Advanced System Level Modeling. Available online: https://www.maplesoft.com/products/maplesim/ (accessed on 20 January 2020).
29. Michael Tiller, D.W. Impact—A Modelica Package Manager. In Proceedings of the 10th International Modelica Conference, Lund, Sweden, 10–12 March 2014.
30. MathWorks. MathWorks. Simulink-Simulation and Model-Based Design. 2019. Available online: https://se.mathworks.com/products/simulink.html (accessed on 1 October 2019).
31. Moler, C.B. Cleve's Corner: The Growth of MATLAB and The MathWorks over Two Decades. The MathWorks News&Notes. 2006. Available online: https://webspace.clarkson.edu/~ebollt/Classes/EE520DataAnalysisComplexSystemsF19/DrSVD.pdf (accessed on 20 January 2020).
32. Ptolemaeus, C. (Ed.) *System Design, Modeling, and Simulation Using Ptolemy II*; Ptolemy.org: Berkeley, CA, USA, 2014.
33. Mirzaei, M.; Buffoni, L.; Fritzson, P. Integration of OpenModelica in Ptolemy II. In Proceedings of the 10th International Modelica Conference, Lund, Sweden, 10–12 March 2014; Tummescheit, H.; Årzén, K.E., Eds.; Modelica Association and Linköping University Electronic Press: Linköping, Sweden, 2014. [CrossRef]
34. Selić, B.; Gérard, S. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*; Morgan Kaufmann: Boston, MA, USA, 2014. [CrossRef]
35. Raistrick, C.; Francis, P.; Wright, J. *Model Driven Architecture with Executable UML*; Cambridge University Press: Cambridge, UK, 2004.
36. Pop, A.; Sjölund, M.; Ashgar, A.; Fritzson, P.; Casella, F. Integrated Debugging of Modelica Models. *Model. Identif. Control* **2014**, *35*, 93–107. [CrossRef]
37. Sjölund, M. Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models. Ph.D. Thesis, Linköping University, Department of Computer and Information Science, Linköping, Sweden, 2015.
38. Broman, D.; Fritzson, P.; Hedin, G.; Åkesson, J. A comparison of two metacompilation approaches to implementing a complex domain-specific language. In Proceedings of the 27th Annual ACM Symposium on Applied Computing-SAC '12, Trento, Italy, 26–30 March 2012. [CrossRef]
39. Open-Services.net. Open Services for Lifecycle Collaboration-Lifecycle Integration Inspired by the Web. 2008. Available online: http://open-services.net/ (accessed on 20 Janyary 2020).
40. Heath, T.; Bizer, C. Linked Data: Evolving the Web into a Global Data Space. *Synth. Lect. Semant. Web Theory Technol.* **2011**, *1*, 1–136. [CrossRef]
41. linkeddata.org. Linked Data-Connect Distributed Data across the Web. 2006. Available online: http://linkeddata.org/ (accessed on 20 January 2020).
42. Berners-Lee, T. Linked Data. W3C. 2006. Available online: https://www.w3.org/DesignIssues/LinkedData.html (accessed on 20 January 2020).
43. Mengist, A.; Pop, A.; Asghar, A.; Fritzson, P. Traceability Support in OpenModelica Using Open Services for Lifecycle Collaboration (OSLC). In Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, 15–17 May 2017; Modelica Association and Linköping University Electronic Press: Linköping, Sweden, 2017. [CrossRef]
44. König, C.; Mengist, A.; Gamble, C.; Höll, J.; Lausdahl, K.; Bokhove, T.; Brosse, E.; Möller, O.; Pop, A. Traceability in the Model-based Design of Cyber-Physical Systems. In Proceedings of the 2nd American Modelica Conference, Boulder, CO, USA, 23–25 March 2020. [CrossRef]
45. Gorm-Larsen, P.; Thule, C.; Kenneth Lausdahl, V.B.; Gamble, C.; Brosse, E.; Sadovykh, A.; Bagnato, A.; Couto, L.D. Integrated Tool Chain for Model-Based Design of Cyber-Physical Systems: The INTO-CPS project. In Proceedings of the 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data), Vienna, Austria, 11 April 2016. [CrossRef]
46. Gorm-Larsen, P.; Thule, C.; Kenneth Lausdahl, V.B.; Gamble, C.; Brosse, E.; Sadovykh, A.; Bagnato, A.; Couto, L.D. Integrated Tool Chain for Model-Based Design of Cyber-Physical Systems. 2015. Available online: https://www.overturetool.org/publications/papers/05LarsenEtAl.pdf (accessed on 20 January 2020).
47. Lausdahl, K.; Niermann, P.; Höll, J.; Gamble, C.; Möller, O.; Brosse, E.; Bokhove, T.; Couto, L.D.; Pop, A.; König, C. NTOCPS Traceability Design. In *Technical Report, INTO-CPS Deliverable, D4.2d*; Linköping University Electronic Press: Linköping, Sweden, 2016.
48. Moreau, L.; Missier, P.; Cheney, J.; Soiland-Reyes, S. An Overview of the PROV Family of Documents. Project Report W3C. 2013. Avaliable online: https://eprints.soton.ac.uk/356854/ (accessed on 20 January 2020).

49. Fitzgerald, J.; Gamble, C.; Payne, R.; Pierce, K. Methods Progress Report 1. In *Technical Report, INTO-CPS Deliverable, D3.1b*; Linköping University Electronic Press: Linköping, Sweden, 2015.
50. Tari, M.T.; Zoheb, M.H.; Fritzson, P.; Thomas, R. OMWeb-Virtual Web-based Remote Laboratory for Modelica in Engineering Courses. In Proceedings of the 8th International Modelica Conference, Dresden, Germany, 20–22 March 2011. [CrossRef]
51. Open Modelica. Omwebbook. Available online: http://omwebbook.openmodelica.org (accessed on 11 March 2019).
52. Fors, N.; Petersson, J.; Henningsson, M. Generating FMUs for the Feature-Based Language Bloqqi. In Proceedings of the 2nd Japanese Modelica Conference, Tokyo, Japan, 17–18 May 2018; pp. 245–254.
53. Sjölund, M.; Fritzson, P.; Pop, A. Bootstrapping a Modelica Compiler aiming at Modelica 4. In Proceedings of the 8th International Modelica Conference, Dresden, Germany, 20–22 March 2011. [CrossRef]
54. Åkesson, J.; Ekman, T.; Hedin, G. Implementation of a Modelica compiler using JastAdd attribute grammars. *Sci. Comput. Program.* **2010**, *75*, 21–38. [CrossRef]
55. Hedin, G. Reference Attributed Grammars. *Informatica* **2000**, *24*, 301–317.
56. Knuth, D.E. Semantics of Context-Free Languages. *Math. Syst. Theory* **1968**, *2*, 127–145. [CrossRef]
57. Söderberg, E.; Hedin, G. Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking. In *Technical Report LU-CS-TR:2012-249*; Department of Computer Science, Lund University: Lund, Sweden, 2012.
58. Öqvist, J.; Hedin, G. Concurrent circular reference attribute grammars. In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, 23–24 October 2017; pp. 151–162.
59. Ekman, T.; Hedin, G. The JastAdd system-modular extensible compiler construction. *Sci. Comput. Program.* **2007**, *69*, 14–26. [CrossRef]
60. Kågedal, D.; Fritzson, P. Generating a Modelica Compiler from Natural Semantics Specifications. In Proceedings of the Summer Computer Simulation Conference, Society for Computer Simulation, ETC, Reno, NV, USA, 19–22 July 1998.
61. Fritzson, P.; Pop, A.; Broman, D.; Aronsson, P. Formal Semantics Based Translator Generation and Tool Development in Practice. In Proceedings of the 20th Australian Software Engineering Conference (ASWEC 2009), Gold Coast, Australia, 14–17 April 2009.
62. Pop, A.; Fritzson, P. MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language. In *Modular Programming Languages*; Lightfoot, D., Szyperski, C., Eds.; Modular Programming Languages; JMLC 2006; Lecture Notes in Computer Science; Springer: Berlin/Heidlberg, Germany, 2006; Volome 4228, pp. 211–229. [CrossRef]
63. Fritzson, P.; Pop, A.; Sjölund, M. Towards Modelica 4 Meta-Programming and Language Modeling with MetaModelica 2.0. In *Technical Report 2011:10*; Linköping University, PELAB-Programming Environment Laboratory: Linköping, Sweden, 2011.
64. Fritzson, P.; Pop, A.; Sjölund, M.; Asghar, A. MetaModelica—A Symbolic-Numeric Modelica Language and Comparison to Julia. In Proceedings of the 13th International Modelica Conference, Regensburg, Germany, 4–6 March 2019. [CrossRef]
65. Milner, R.; Tofte, M.; Harper, R.; MacQueen, D. *The Definition of Standard ML (Revised)*; MIT Press: Cambridge, MA, USA, 1997.
66. OCaml. OCaml Web Site. 2020. Available online: https://ocaml.org (accessed on 30 January 2020).
67. Sjölund, M.; Fritzson, P.; Pop, A. Bootstrapping a Compiler for an Equation-Based Object-Oriented Language. *Model. Identif. Control* **2014**, *35*, 1–19. [CrossRef]
68. Duracz, A. Rigorous Simulation: Its Theory and Applications. Ph.D. Thesis, Halmstad University Press, Halmstad, Sweden, 2016.
69. Schamai, W.; Buffoni, L.; Albarello, N.; Miranda, P.F.D.; Fritzson, P. An Aeronautic Case Study for Requirement Formalization and Automated Model Composition in Modelica. In Proceedings of the 11th International Modelica Conference, Versailles, France, 21–23 September 2015. [CrossRef]
70. Garro, A.; Tundis, A.; Bouskela, D.; Jardin, A.; Thuy, N.; Otter, M.; Buffoni, L.; Fritzson, P.; Sjölund, M.; Schamai, W.; et al. On formal cyber physical system properties modeling: A new temporal logic language and a Modelica-based solution. In Proceedings of the 2016 IEEE International Symposium on Systems Engineering (ISSE), Edinburgh, UK, 4–5 October 2016; pp. 1–8. [CrossRef]
71. ITEA3. Environment for Model-Based Rigorous Adaptive Co-design and Operation of CPS, 2020–2022. Available online: https://itea3.org/project/embrace.html (accessed on 20 January 2020).
72. Bouskela, D.; Jardin, A. ETL: A new temporal language for the verification of cyber-physical systems. In Proceedings of the 2018 Annual IEEE International Systems Conference (SysCon), Vancouver, BC, Canada, 23–26 April 2018; pp. 1–8. [CrossRef]
73. Bouissou, M.; Elmqvist, H.; Otter, M.; Benveniste, A. Efficient Monte Carlo simulation of stochastic hybrid systems. In Proceedings of the 10th International Modelica Conference, Lund, Sweden, 10–12 March 2014. [CrossRef]
74. Lie, B.; Palanisamy, A.; Mengist, A.; Buffoni, L.; Sjölund, M.; Asghar, A.; Pop, A.; Fritzson, P. OMJulia: An OpenModelica API for Julia-Modelica Interaction. In Proceedings of the 13th International Modelica Conference, Regensburg, Germany, 4–6 March 2019; Linköping University Electronic Press, Linköpings Universitet: Linköpings, Sweden, 2019; p. 10.
75. Bouissou, M.; Bouhadana, H.; Bannelier, M.; Villatte, N. Knowledge Modelling and Reliability Processing: Presentation of the Figaro Language and Associated Tools. In Proceedings of the IFAC Symposium on Safety of Computer Control Systems 1991 (SAFECOMP'91), Trondheim, Norway, 30 October–1 November 1991. [CrossRef]
76. Bouissou, M.; Buffoni, L.; Thiele, B. From design to dependability: A Bridge Between Physical Simulation and Risk Analysis. In Proceedings of the Congrès Lambda Mu 20 de Maîtrise des Risques et de Sûreté de Fonctionnement, Saint Malo, France, 11–13 October 2016. [CrossRef]
77. Fors, N.; Sten, J.; Olsson, M.; Stenström, F. A Safe Regression Test Selection Technique for Modelica. In Proceedings of the American Modelica Conference, Cambridge, MA, USA, 9–10 October 2018; pp. 131–137.

78. Otter, M. *Modelica: A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification Version 3.3*; Modelica Association: Linköping, Swenden, 2012. Avaliable online: https://www.modelica.org/documents/ModelicaSpec33.pdf (accessed on 20 January 2020).
79. Berger, L.; Sjölund, M.; Thiele, B. Code Generation for STM32F4 Boards with Modelica Device Drivers. In Proceedings of the 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, Weßling, Germany, 1 December 2017. [CrossRef]
80. Fors Johansson, C. OpenModelica Interactive Simulation Using an OPC UA Client. Ph.D Thesis, Dissertação Linkoping University, Linkoping, Sweden, 2017.
81. Verzelen, F.; Peter Lacy, N.S. Designing Disruption: The Critical Role of Virtual Twins in Accelerating Sustainability. 2021. Available online: https://www.3ds.com/sites/default/files/2021-01/dassault-systemes-and-accenture-virtual-twin-and-sustainability.pdf (accessed on 20 January 2020).
82. Barricelli, B.R.; Casiraghi, E.; Fogli, D. A Survey on Digital Twin: Definitions, Characteristics, Applications, and Design Implications. *IEEE Access* **2019**, *7*, 167653–167671. [CrossRef]