Towards Integrated Model-Driven Testing of SCADA Systems Using the Eclipse Modeling Framework and Modelica

Conference Paper · March 2008

DOI: 10.1109/ASWEC.2008.4483203 · Source: DBLP

Adrian Pop
Linköping University
59 PUBLICATIONS **319** CITATIONS
SEE PROFILE

Peter Fritzson
Linköping University
365 PUBLICATIONS **3,267** CITATIONS
SEE PROFILE

Towards Integrated Model-Driven Testing of SCADA Systems Using the Eclipse Modeling Framework and Modelica

Jörn Guy Süß
The University of Queensland
jgsuess@itee.uq.edu.au

Adrian Pop
Linköpings Universitet
adrpo@ida.liu.se

Peter Fritzson
Linköpings Universitet
petfr@ida.liu.se

Luke Wildman
The University of Queensland
jgsuess@itee.uq.edu.au

Abstract

Testing SCADA (Supervisory Control And Data Acquisition) near real-time systems is challenging, as it involves complex interactions and the simulation of the supervised and controlled environment. Model-driven testing techniques can help to achieve clarity about the inner workings of the system and facilitate test construction, but these models are currently disconnected from those of the environmental simulation, leading to a paradigm break. This paper presents a strategy to remedy this situation. To this end, it leverages Modelica and the Eclipse Modeling Framework.

Modelica is an object-oriented mathematical modeling language for component-oriented modeling of complex physical systems. It is an open standard and implementation, and provides a rendering of its input language in Ecore, the meta-language of the Eclipse Modeling Framework (EMF). It also offers convenient visual editors, whose notation via the ModelicaML profile is consistent with the SysML standard, a restricted version of UML

The strategy presented here leverages EMF as a common basis for model-driven development, reusing Modelica's powerful simulation features in integration with a custom-designed testing process. With this tooling, a test engineer can model all aspects of a SCADA test within one workbench and enjoy full traceability between the proprietary test model, and its surrounding environment simulation.

1. Introduction

One of the most important current shifts in paradigm occurring in the design of products may well be the adoption of common system models, as a foundation for product/system design. This will allow for a much more effective product development process since a system can be tested in all stages of design.

For example, the development in system modeling has come to the point where complete modeling of systems is possible, e.g. the complete propulsion system, fuel system, hydraulic actuation system, etc., including embedded software could be modeled and simulated concurrently.

We envision an integrated unified model-driven development environment (Figure 1), supporting all phases including requirements, design, implementation, testing. In this paper we focus on integrated model-driven testing.

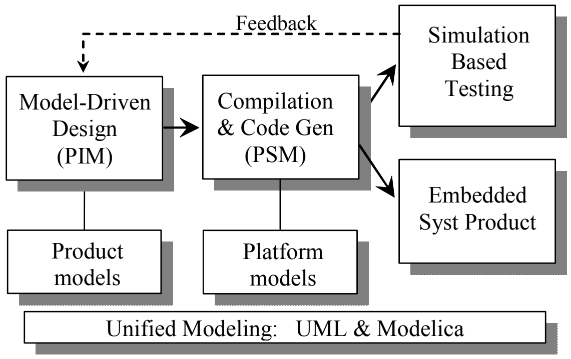

Feedback
Model-Driven Design (PIM)
Compilation & Code Gen (PSM)
Simulation Based Testing
Embedded Syst Product
Product models
Platform models
Unified Modeling: UML & Modelica

Figure 1. Vision of unified modeling, development, and testing environment.

Supervisory Control And Data Acquisition (SCADA) systems are quickly becoming the backbone of many modern industries. SCADA systems are large, distributed near real-time systems layered on top of full real-time local systems. One can imagine a SCADA system as the pan-vision and policy setter of an enterprise. Typical SCADA applications are in manufacturing, where whole production lines are controlled, in refineries, where reactors are started and stopped, in pipelines, ships, airports, subway lines.

In earlier days, SCADA applications were custom-built by system manufacturers using product specific hardware and communication standards. No two SCADA system were or needed to be, the same. In the last years, SCADA software and sensor hardware have parted ways and the TCP/IP protocol has largely replaced proprietary busses. As a result, SCADA systems for different target areas are built as software product lines from reusable components.

The impetus is shifting from individual programming to configuration. However, testing these highly central systems for safety and security properties is essential. The difficulties involved in the testing and assurance of modern SCADA systems are considerable [16]. This arises in part because of the rich number of features that SCADA products typically provide, but also because SCADA products are highly configurable. The aim of the research reported in this paper is to address these difficulties by intensive test automation. In particular we are investigating a model-driven approach.

In the following sections we will introduce SCADA software in more detail. In Section 3 we introduce a stereotypical testing model of SCADA and show its relationship with the Eclipse Modeling Framework (EMF) that we will use as a facilitator. Section 5 describes Modelica, whereas Sections 6 and 7 describes its binding to EMF, and Section 8 describes EMF as an integration platform.

2. Characteristics of SCADA

The scenarios mentioned in the introduction highlight the defining common characteristics of a SCADA system:

1. It provides traceable control and directing information about a complex distributed subject without interruption.
2. It summarizes measurements from sensors and enacts changes via local agents from a high level

These requirements prompt a stereotypical architecture for most SCADA systems. The first requirement translates into the need of spatial distribution of servers to eliminate single points of failure. This in turn prompts a bus model as an abstraction for the design of applications.

Furthermore, the first requirement makes it necessary for SCADA services to virtually run *non-stop*, which requires that the bus can hand over control on the fly from one instance of a process that is due for maintenance to a back-up. It also means that login and persistency have to be built into the underlying bus.

The second requirement means that the system has to interact with a complex environment via remote terminal units (RTUs), and account for their varying properties when working with them.

3. Testing SCADA

SCADA systems are large, specialized and entrenched core business systems. Much like host computers and operating systems, replacing a SCADA installation carries great risk. Existing SCADA platforms, i.e. bus systems may evolve, but they are unlikely to be replaced by complete rewrites in the future.

Hence, each SCADA platform carries a long legacy of development with it. Often, SCADA products survive business changes and their compacted source code pays homage to corporate history. However, this consolidation makes long-lived SCADA products hard to understand and hard to test. When tests focus on the fine-grain surface interaction with the services of a SCADA product, test coverage is hard to achieve because of lacking a birds-eye view of the subject.

Model-driven testing aims at providing such a birds-eye view. For the time being, we will not focus on what a *model* is in this context. We will first establish *what* we will have to model and then turn to the means of modeling in Section 3. The first models can be derived from the stereotypical anatomy of the SCADA system

SCADA systems typically interact with a physical environment as depicted in Figure 2; see also Figure 46. This environment can be modeled and simulated, thus greatly facilitating driving the test cases by the test client.

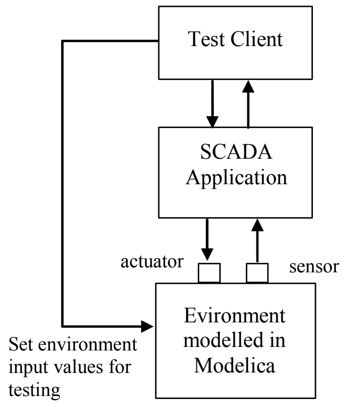

Test Client
SCADA Application
actuator
sensor
Evironment modelled in Modelica
Set environment input values for testing

Figure 2. Model-driven testing of a SCADA application.

SCADA systems are distributed, and the clients of the remote service will use it as a black box. Consequently, we will need to define the interfaces of our test subjects using a model of the technical language, the Interface Definition Language (IDL) of the platform. Very often, such higher level languages are not only used to *define* that interface, but also to *generate* skeletal implementations of the service that already fulfilled the requirements of value marshalling and other platform facilities like persistency and logging. With these additions, the *IDL model* can be managed and filled by the application architect and used by the service implementer and tester.

The service implementer can make annotations that allow the addressing of a specific *implementation* of the interface, as laid down in the company's repository. Finally, a *product* combines a number of implementations. In order to test a SCADA product, one has to have a running instance of it.

For tests to be repeatable, this configuration has to be controlled. Usually, this model will involve grounding properties, like names of computers, installation paths and addresses, and logical properties specific to the bus. For example, there may be the concept of names, by which services are addressed on the distributed bus, or the concept of domains, which guide the failover from a primary service instance to its back-up, or configuration information that tunes each service for its conditions of use.

Again, this *configuration model* has a double purpose: It can be provided and used by system operators to deploy a product using a deployment tool and it can be used by the test engineer to establish a connection to the test subject.

At this point, the test engineer can derive from the provided models a syntactically correct invocation and inspection on any service of the SCADA system.

The engineer knows the *types* that are legal for the construction of an invocation instance, and that will be returned by querying a service. The engineer knows the *address* and *configuration* of the service. Now, a test can be described as a sequence of stimuli and assertions against interfaces of services within a specific configuration of a specific product. This sequential representation augmented with instances of the data writeable to and observable in the service is the most simple *test model*. It is played out via test framework, and its performance is captured and re-imported in the model for analysis of the test run. The more the engineer knows about the interfaces and implementations of its subject, the more of the test sequences can be generated using rules that capture the testing intent.

However, the scenario so far does not cover a crucial aspect of SCADA systems. Most SCADA functionality is designed to deal with the control of an environment that exhibits complex behavior. In order to build a full test the complex environment has to be modeled and the definition of test intent has to be able to refer to parts of this model.

In theory, it is trivial to hypothesize the presence of an *environment model*, which captures the complex behavior of the mechanical, physical, electrical and mixed systems SCADA connects to and automatically produces a simulator program that represents this environment. In practice, developing such a language is so complex and costly that no SCADA company could pay for this sort of custom development.

But what are these models we have so vaguely alluded to in the previous sections? They are instances of Ecore, an object-oriented modeling notation used by the EMF [15] framework. This notation knows classes, which have value-based attributes and references that point to other classes. Classes can be bundled into packages, which may be nested. Finally, the notation allows to define enforced cardinalities on attributes and references and rules for bi-directional relations.

We will now repeat the description of the models given above with a couple of minimal examples that have been distilled out of the basis of our industry implementation. These models later translate into a Java-based persistency and editing API, a storage layer and a top level-editor. A screenshot of such an automatically generated editor is shown in Figure 1114. The models are given using the UML class diagram visual conventions to express the different Ecore models.

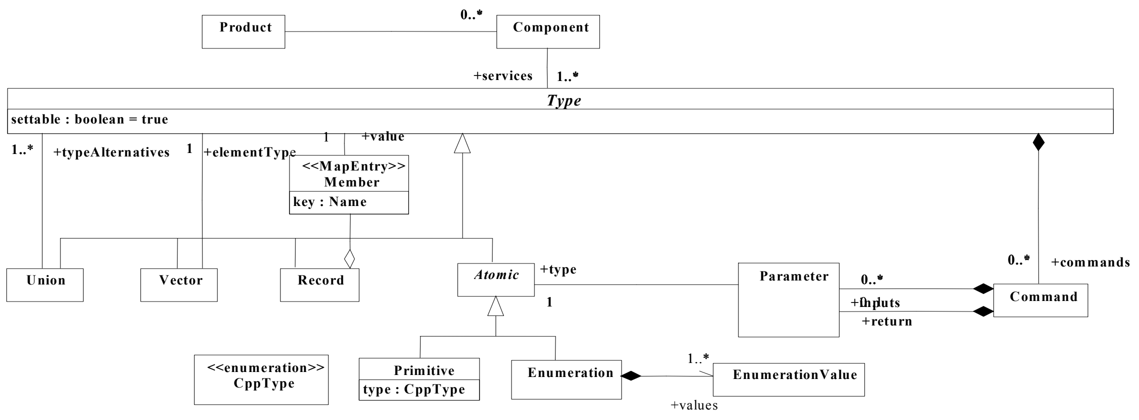


Figure 3. MODEL1: design-time model capturing an IDL and product composition.

- MODEL1 (Figure 3) shows an IDL. In it, we find a couple of typical IDL structures. We will use it to introduce the notation as we go.
- A product that references a number of components. Note the cardinality marker 0..* defines optional presence and no maximal limit (*).
- A component that references a number of types that represent the services it offers. Note that the lower cardinality marker 1 defines required presence.
- We further see a number of specialized types, see below. Note that the specialization is shown as an empty delta arrow.

The specialized types just mentioned are the following:

- Unions, which have a number of alternative types.
- Vectors, which have exactly one element type.
- Atomic types, which comprise primitive types and enumerations.
- Primitives that are of a limited set of choices. The notation `<<enumeration>>` constructs a Java 1.5 enumeration type with the listed literals.
- Enumerations are a list of choices defined at the type level. Note that the composition – the filled rhombus symbol – indicates existential dependency. The existence of an enumeration value depends on the existence of its associated enumeration.
- Records, which contain a map that maps with entries which map names to types. The `<<MapEntry>>` constructs a Java 1.5 Generics map with the defined types in the owning type.

With this introduction, the way commands are composed by a type should now be interpretable by the reader. Let us model the remote interface of a double coffee pot as an example: We create a product '`mini-kitchenette`', which contains only one component, '`a coffee-machine`'. It contains three types: An enumeration '`switch`', that can be turned '`on`' and '`off`' by the user, A read-only primitive '`low water`' indicating if we need to fill the pot. A command '`fill(amount:int_8)`', a record '`pots`' with two '`pot`' records named '`A`' and '`B`'. Each `pot` record has a read-only '`fill percentage`' and '`temperature`'.

We could now generate template source code from this model for an implementer to add the functionality and generate a build script that assembles and packages the product. Obviously, this model can be refined to include version attributes and platform facilities used in the implementation of the component. We will skip the implementation detail model in this paper to save space, instead we will turn directly to the runtime model MODEL2.

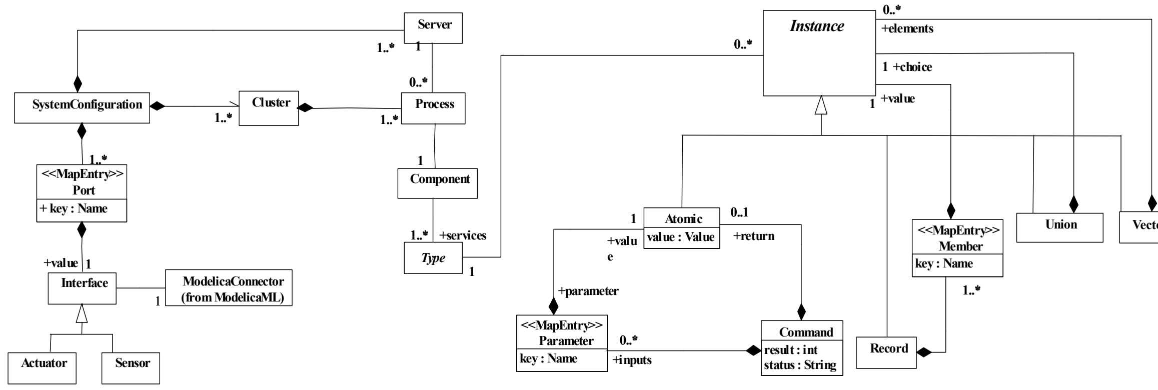


Figure 4. MODEL2: run-time model capturing system configuration and ports to sensors and actuators.

MODEL2 (Figure 4) shows three aspects in one diagram. At the left, the model contains a description of a configuration to start a system in order to test it. In this configuration model, a configuration consists of a number of logical clusters that provide fail-safety through fail-over redundancy and a number of server machines. The processes of the system are the runtime-reflections of the component. The right side of the left part of the diagram is comparable to the deployment information necessary to start a J2EE server.

The left hand side of the left part of MODEL2 introduces the first SCADA-specific part of the model: It describes how the SCADA system attaches to the outside world. In SCADA systems, the operations on the RTUs, the *collection of data* and *actuation of settings*, are on a different level than the interactions between bus components. Attachment of RTUs is a configuration issue. In our model, a SCADA system is modelled as having a number of uniquely named ports, which are either `Actuators` or `Sensors`. For the purpose of testing, they are attached to Modelica. We describe this link among the models and their code in Section 4 below.

The right hand side of this run-time model describes the construction of instances. These can be used to describe specific test situations. For example it is possible to model a coffee machine that is turned on and whose pots are half-full. A single instantiation relationship connects the abstract superclass `Instance`, and the abstract superclass `Type`. Of course, a valid model would need to enforce type correctness constraints, to ensure for example that a vector instance connects to a vector type and that the instance that form its elements conform to the element type defined for the vector. The expressive means of the class-diagrammatic Ecore, structural integrity constraints of cardinality, and containment are not sufficiently powerful to ensure the requirement of type compliance.

Consequently additional wellformedness rules need to be implemented to ensure that the instances comply. However, even this step is facilitated and made expressive through the use of the Object Constraint Language, an expression language designed specially for the purpose of writing wellformedness rules in the context of class-based models. It combines navigation with high-level logical constructs and iterators, leading to terse expressions for most constraints. In our industry case it has been used extensively.

Usually, the interest in model-driven testing approaches lies in the useful generation of instances as assertion patterns from additional information the test has about the test subject. There are numerous strategies and they are independent of the SCADA domain, so we will skip this aspect in this paper

Instead, we will show a simple linear test execution model: A test is a sequence of steps, executed in the context of a specific system configuration. In this sense, our approach is more model-driven then model-based. A is test bound to a `ModelicaModel` of the environment that is being simulated. We will describe this binding in more detail in Section 4 below.

As mentioned earlier, SCADA systems rely on sensors and thus scan intervals of these sensors are important. If a test depends on a change being signaled, that change will only become apparent after the next scan and hence the test should wait at least for this time interval before proceeding. The scan interval allows the tester to determine how 'fast' a test should run and find out at which point internal system interaction time becomes a relevant factor.

The test steps either deal with values within a component, or with a whole process. The steps that deal with values either compare instances, write the data of an instance into a service, or invoke commands on that service. The comparisons must succeed, in order for the test to succeed.

At the level of the processes, a process can be promoted to be the main instance, or demoted to a backup instance; it can be started and stopped. As SCADA systems are scanning their environment in cycles, scripts may have to wait. Finally environment steps describe direct stimulation and inspection of the simulated environment.

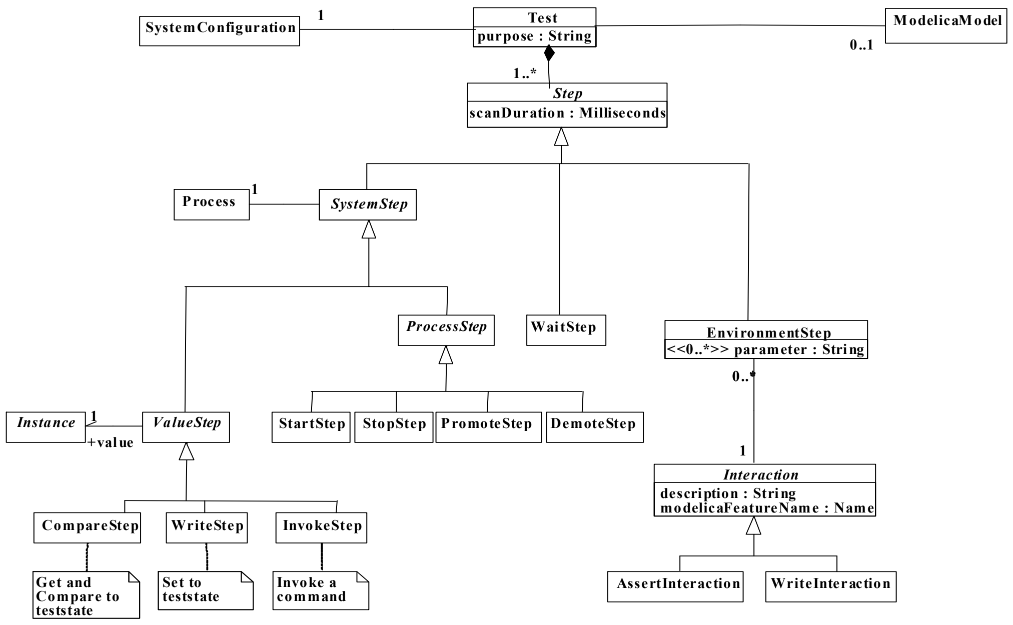


Figure 5. MODEL3: Test model capturing test steps and simulator binding.

4. Linking the Modelica Simulated Environment to the Test System

So far, we have focused on the description of the internals of the system: e.g. a coffee machine is not influenced very much by the outside world. However, SCADA systems are all about interaction with a complex outside world. While the models of the application as described above are comparatively simple, the outside world observed and controlled by SCADA systems is complex and an executable model that is able to simulate electro-mechanical and physical systems is expensive to construct.

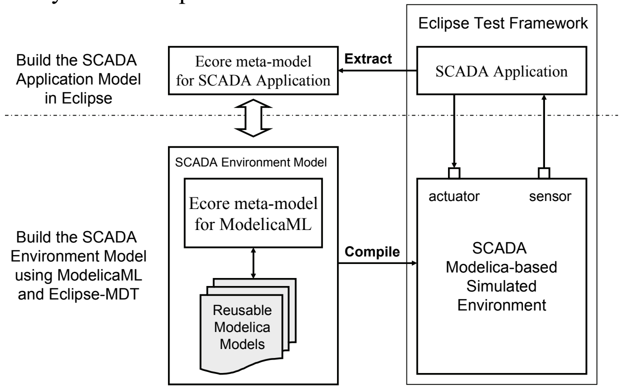

Eclipse Test Framework
Build the SCADA Application Model in Eclipse
Ecore meta-model for SCADA Application
Extract
SCADA Application
SCADA Environment Model
Ecore meta-model for ModelicaML
Build the SCADA Environment Model using ModelicaML and Eclipse-MDT
Compile
actuator
sensor
SCADA Modelica-based Simulated Environment
Reusable Modelica Models

Figure 4. Testing Framework

In this section we will explore the immediate application of Modelica as a tool for modeling and simulating the environment, and then extend it into the prospect of using Modelica as a model-driven testing tool (Figure 46). Modelica combines object-orientation with equations. A model consists of a composition of objects, with a model as the outermost compositor. This can be a physical, mechanical, control, electrical, or electronic system or any combination of these. This model is compiled into an executable that usually computes the development of the properties of objects in the system from initial values as a function of time.

However, besides this self-contained mode of simulation, Modelica also offers the definition of connectors to the outside world. A Modelica connector is a conduit for directed flow of values. In our model, the sensors and actuators are linked to connectors on the outermost simulation model, as seen in Figure 2 and below.

```
model Simulator
  ConnectorTypeA  sensor1;
  ConnectorTypeB  sensor2;
  ConnectorTypeD  actuator1;
  ConnectorTypeE  actuator2;
  ...
end Simulator
```

In addition, the test script is linked to the simulated environment via writes and assertions. Writes are signals or invocations that change the behavior on the next recomputation cycle. Assertions are invocations of functions with a Boolean result. The associated Modelica model can be used to check the correctness of parameter values provided for the invocation before it actually occurs. Effectively, this set-up puts the SCADA system in a testing harness that has the shape of a G-clamp.

For the technical execution of the test, a Modelica tool translates the model into a C source code, which compiles into an optimized binary for the target platform. This guarantees that the resulting simulator meets near-real-time requirements that could potentially arise with a SCADA system. In the execution, each step is mapped into a single invocation or a set of invocations on the system using a test framework. The information necessary to construct the invocation is drawn from the runtime model, the character of each line follows from the type of step used.

With this, we can construct the example of a SCADA system that controls a tank with valves and pumps, as shown in Section 5. Consider the following requirement: The tank control will remain effective, even in case of a power failure. We can now construct the following test: A single component, `TankMonitor`, is started in two instances on two different servers, with one dedicate primary, the other backup. The sensors of the underlying SCADA system are attached to the sensors and actuators of the Modelica Simulation.

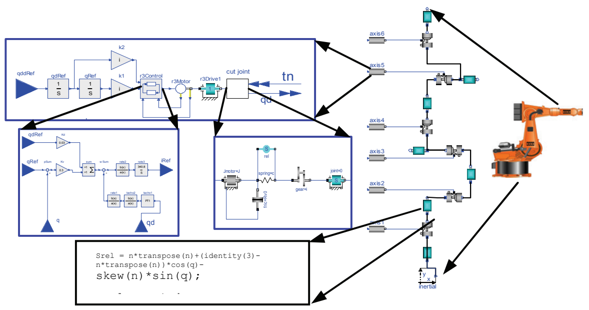
We now start a surge in the tank by triggering an event via connector to the simulator. In the next test step, we stop the primary process. We wait and invoke a test function on the tank to confirm that the level stays under control. As an aside, we can observe the water level in the simulated tank and the actions the SCADA system is enacting via its actuators and requests via its sensors by logging the interactions on the SCADA systems interface.

We can extend this approach even further by specifying the *expected behavior* of a component within the simulator. Now the simulator, in addition to the environment, will hold a model of the component. This allows us to make predictions about the behavior of the SCADA system *before* we run the test. It can also be used in prototyping before implementation of a component actually takes place. In a later test, we can check that the actual behavior of the component in the live system matches that of its simulation within the Modelica simulator.

5. Modelica

This section briefly introduces Modelica [2][3], which is a modern language for equation-based object-oriented mathematical modeling primarily of physical systems. Several tools, ranging from open-source such as OpenModelica

[1], to commercial like Dymola [11] or MathModelica [10] support the Modelica specification.



```
Srel = n*transpose(n)+(identity(3)-
n*transpose(n))*cos(q)-
skew(n)*sin(q);
```

Figure 5. Hierarchical model of an industrial robot.

The language allows defining models in a declarative manner, modularly and hierarchically and combining various formalisms expressible in the more general Modelica formalism.

A component may internally consist of other connected components, i.e., as in Figure 57 showing hierarchical modeling.

The multidomain capability of Modelica allows combining electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model. In short, Modelica has improvements in several important areas:

- *Object-oriented mathematical modelin*g. This technique makes it possible to create model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.
- *Physical modeling of multiple application domains*. Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to “signal” blocks with fixed input/output causality. In Modelica the structure of the model naturally correspond to the structure of the physical system in contrast to block-oriented modeling tools.
- *Acausal modeling. Modeling is based on equations* instead of assignment statements as in traditional input/output block abstractions. Direct use of equati*ons significantl*y increases re-usability of model components, since components adapt to the data flow context in which they are used.

As we have stated previously, in the context of this work, the environment connected to the SCADA system is modeled in Modelica. A very simple example of a SCADA system controlling an external environment (a tank system) is depicted in Figure 68.

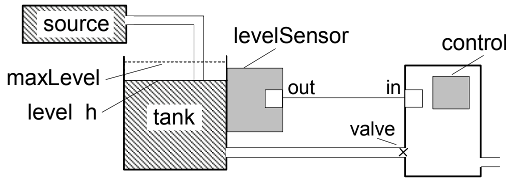


Figure 6. A small tank control application with a tank, a source for liquid, and a controller.

As an example, the Modelica code of the tank component in the simplified tank system is shown below and in Figure 912.

```
model Tank
  ReadSignal tSensor;
  ActSignal  tActuator;
  LiquidFlow qIn;
  LiquidFlow  qOut;
  parameter Real area = 0.5;
  parameter Real flowGain = 0.05;
  parameter Real minV=0, maxV=10;
  Real h(start=0.0, unit="m");
equation
  der(h) = (qIn.lflow-qOut.lflow)/area;
  qOut.lflow  = LimitValue(minV,maxV,-
flowGain*tActuator.act);
  tSensor.val = h;
end Tank;
```

5.1. SysML vs. Modelica

The System Modeling Language (SysML) [4] has also been proposed to unify different approaches and languages used by system engineers into a single standard. SysML models may span different domains, for example, electrical, mechanical and software. Even if SysML provides means to describe system behavior like Activity and State Chart Diagrams, the precise behavior cannot be described and simulated without complex transformations and additional information provided for SysML models. In that respect, SysML is rather incomplete compared to Modelica.

Modelica was also created to unify and extend object-oriented mathematical modeling languages. It has powerful means for describing precise component behavior and functionality in a declarative way. Modelica models can be graphically composed using Modelica connection diagrams which depict the structure of de-signed system. However, complex system design is more that just a component assembly. In order to build a complex system, system engineers have to gather requirements, specify system components, define system structure, define design alternatives, describe overall system behavior and perform its validation and verification.

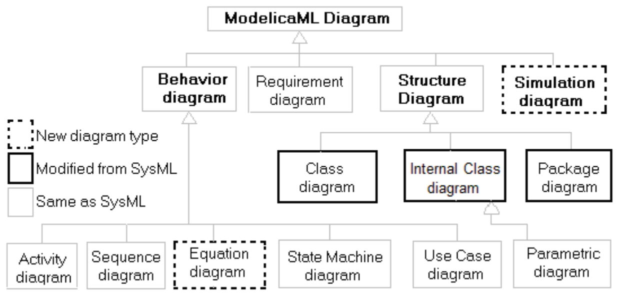


Figure 7. ModelicaML diagram overview.

6. The ModelicaML UML profile

Before we present the Eclipse integration, we briefly describe the ModelicaML profile.

With respect to SysML, ModelicaML reuses, extends and provides several new diagrams. The ModelicaML diagram overview is shown in Figure 79. Diagrams are grouped into four categories: Structure, Behavior, Simulation and Requirement. The ModelicaML profile is presented in [8], [14], and [15]. The most important properties of the ModelicaML profile are the following:

- The ModelicaML profile supports modeling with all Modelica constructs and properties i.e. restricted classes, equations, generics, variables, etc.
- Using ModelicaML diagrams it is possible to describe most of the aspects of a system being designed and thus support system development process phases such as requirements analysis, design, implementation, verification, validation and integration.
- The profile supports mathematical modeling with equations since equations specify behavior of a system. Algorithm sections are also supported.
- Simulation diagrams are introduced to model and document simulation parameters and simulation results in a consistent and usable way.
- The ModelicaML meta-model is consistent with SysML in order to provide SysML-to-ModelicaML conversion.

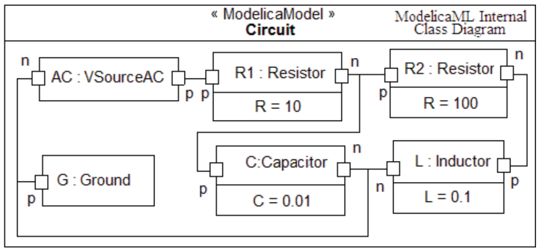
6.1. Modelica Class Diagrams in ModelicaML

Modelica uses restricted classes such as class, model, block, connector, function and record to describe a system. Modelica classes have essentially the same semantics as SysML blocks specified in [4] and provide a general-purpose capability to model systems as hierarchies of modular components. ModelicaML extends SysML blocks by defining features which are relevant or unique to Modelica.

The purpose of the Modelica Class Diagram is to show features of Modelica classes and relationships between classes. Additional kind of dependencies and associations between model elements may also be shown in a Modelica Class Diagram. For example, behavior description constructs – equations, may be associated with particular Modelica Classes. Each class definition is adorned with a stereotype name that indicates the class type it represents. The ModelicaML Class Definition has several compartments to group its features: parameters, parts, variables. Some compartments are visible by default; some are optional and may be shown on ModelicaML Class Diagram with the help of a tool. Property signatures follow the Modelica textual syntax and not the SysML original syntax, reused from UML. A ModelicaML/SysML tool may allow users to choose between UML or Modelica style textual signature presentation.

The ModelicaML Internal Class Diagram is based on the SysML Internal Block Diagram. The Modelica Class Diagram defines Modelica classes and relation-ships between classes, like generalizations, association and dependencies, whereas a ModelicaML Internal Class Diagram shows the internal structure of a class in terms of parts and connections. The ModelicaML Internal Class Diagram is similar to the Modelica connection diagram, which presents parts in a graphical (icon) form. An example Modelica model presented as an Internal Class diagram is shown in Figure 811.

Usually Modelica models are presented graphically via Modelica connection diagrams (Figure 811, bottom). Such diagrams are created by the modeler using a graphic connection editor by connecting together components from available libraries. Since both diagram types are used to compose models and serve the same purpose, we briefly compare the Modelica connection diagram to the ModelicaML Internal Class Diagram. The main advantage of the Modelica connection diagram over the Internal Class Diagram is that it has better visual comprehension as components are shown via domain-specific icons known to application modelers. Another advantage is that Modelica library developers are able to predefine connector locations on an icon, which are related to the semantics of the component.



```
model Circuit
   Resistor  R1(R=10);
   Capacitor C(C=0.01);
   Resistor  R2(R=100);
   Inductor  L(L=0.1);
   VSourceAC AC;
   Ground    G;
equation
   connect (AC.p, R1.p);
   connect (R1.n, C.p);
   connect (C.n, AC.n);
   connect (R1.n, R2.p);
   connect (R2.n, L.p);
   connect (L.n,  C.n);
   connect (AC.n, G.p);
end Circuit;
```

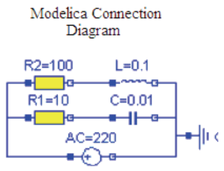


Figure 8. ModelicaML Internal Class Diagram vs. Modelica Connection Diagram.

6.2. Equation Diagram

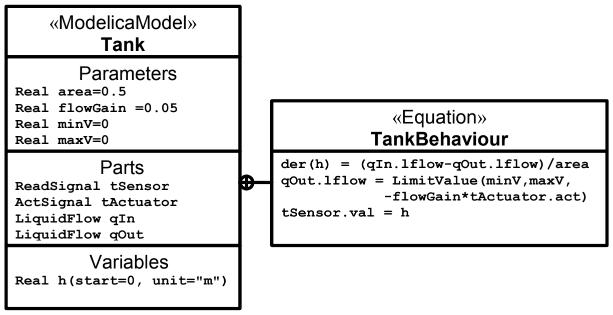


Figure 9. Equation modeling example with a ModelicaML Class Diagram.

The Modelica class behavior is usually described by equations, e.g. the `TankBehavior` in Figure 912, from the `Tank` model of Figure 68. Functions can also be used.

6.3. Simulation Diagram

ModelicaML also introduces a new diagram type, called Simulation Diagram (Figure 1013), used for simulation experiment modeling. Simulation is usually performed by a simulation tool which allows parameter setting, variable selection for output and plotting. The Simulation Diagram can be used to store any simulation experiment, thus helping to keep the history of simulations and its results. When integrated with a modeling and simulation environment, a simulation diagram may be automatically generated.

The Simulation Diagram provides facilities for simulation planning, structured presentation of parameter passing and simulation results. Simulations can be run directly from the Simulation Diagram. Association of simulation results with requirements from a domain expert and additional documentation (e.g. by: Note, Problem Rationale text boxes of SysML) which are also supported by the Simulation Diagram.

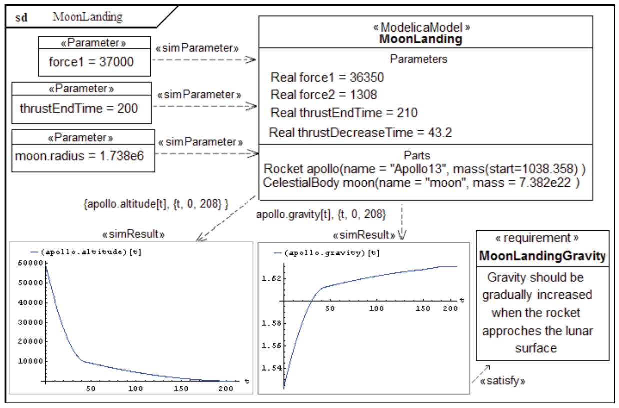
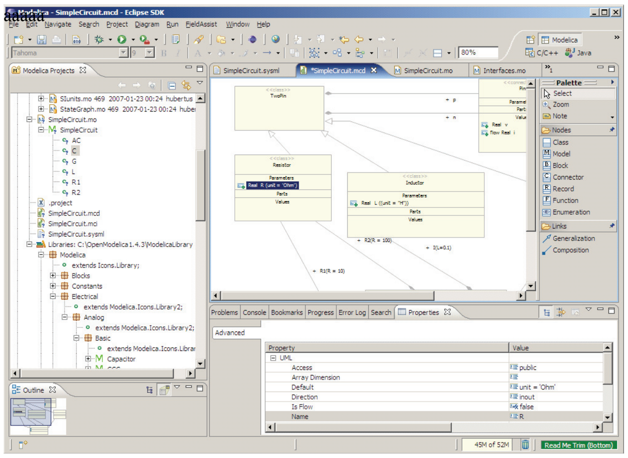


Figure 10. Simulation Diagram example.

7. ModelicaML in Eclipse

Eclipse [14] is an open source framework for creating extensible integrated development environments (IDEs). One of the goals of the Eclipse platform is to avoid duplicating common code that is needed to implement a powerful integrated environment for the development of software. By allowing third parties to easily extend the platform via the plugin concept, the amount of new code that needs to be written is decreased. For the development of our prototype we have used several Eclipse frameworks:

- EMF – Eclipse Modeling Framework is an Eclipse framework for building domain-specific model implementations, and is based on the “Essential MOF” (EMOF) portion of the Meta Object Facility (MOF) standard.
- GMF – Graphical Modeling Framework provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF. GMF consists of tooling, generative and runtime parts, depends on the EMF and GEF frameworks and also on other EMF related tools.
- The UML2 Eclipse meta-model implementation. It is an EMF based implementation of a UML2 meta-model to support development of UML modeling tools.



Property	Value
UML	
Access	public
Array Dimension	
Default	unit = 'Ohm'
Direction	inout
Is Flow	false
Name	R

Figure 11. ModelicaML Eclipse based design environment with a Class diagram.

7.1. The ModelicaML Eclipse Editor

The Modelica Development Tooling (MDT) [7] Eclipse plugin is part of the OpenModelica system [1] and provides an environment for working with Modelica projects.

We have extended the MDT plugin with a design view to facilitate ModelicaML integration. The ModelicaML integrated design environment where SysML/ ModelicaML diagrams are created is shown in Figure 1114. It consists of a diagram file browser (left), diagram editor (middle), tool palette (right), properties editor (bottom) and a diagram outline (bottom left).

8. EMF as an Integration Platform

The scenarios mentioned previously are only possible because ModelicaML and our test modeling language are both based on underlying EMF technology. In this way, the user interfaces, manipulation and test facilities are expressed as large granular binary components that can be integrated into one functional workbench by simply declaring them to be part of an aggregating feature. The test language has a dependency on ModelicaML, as it directly references parts of a Modelica Model, however, the integration goes a bit further: An instance of a test model can reference a ModelicaML model, even though the two reside in *separate resources*. Navigation can be performed from the test model into the Modelica model it depends on to check consistency and wellformedness. These properties allow for a very different approach to reuse. For example, existing Modelica models of valves, tanks, antennae, motors, spools can be reused across different domains and used with testing frameworks specifically designed for different SCADA systems. This is particularly important as Modelica is an open standard and a large reuse community already exists for it.

9. Related Work

Model-driven approaches to the development of SCADA products have been investigated previously 0. However, in that work the focus has been on the use of models of the applications for code generation, whereas our focus in this work is an integrated model-driven approach to testing. UML State charts and Modelica have previously been combined, see e.g. [9][13]. The SysML profile [4] is rather recent, but has for example been used for system on chip design [12].

Several proprietary products such as LabView, Simulink, VisSim, etc., are applicable in the context of testing SCADA products. The advantages of Modelica include its use of Open standards and the ongoing integration with SysML and UML2.0, the generality of its equation-based OO modeling language, the semantic integration with the testing framework facilitated by the use of EMF, and the portability of the generated Modelica model C-code across platforms. Finally, adoption of the Modelica standard by several vendors enables Modelica models to be portable across multiple vendors

10. Conclusion and Future Work

In this paper we have presented an integrated approach for model-driven development, with emphasis on model-driven testing of SCADA systems, as well as the use of Modelica for simplifying the testing by modeling and simulating the external environment, and the integration of these in Eclipse based on EMF.

The strategy presented here leverages EMF as a common basis for model-driven development, reusing Modelica's powerful simulation features in integration with a custom-designed testing process. With this tooling, a test engineer can model all aspects of a SCADA test within one workbench and enjoy full traceability between the proprietary test model, and its surrounding environment simulation.

The approach is being tried out at Westinghouse Rail systems Australia (WRSA) as part of an industrial collaboration. Progress thus far allows test scripts to be generated for applications built upon their SCADA product SystematICS. This involved detailed modeling of the application interface, the runtime context, the environment and the test itself. By using EMF, model editors, persistency, and constraint checking were implemented without significant difficulty. The objective of this approach is to improve test script maintainability, debugging, scalability, and automation. In future work, the relative improvement in these areas will be evaluated.

11. Acknowledgements

This work was supported by Vinnova in the Safe & Secure Modeling and Simulation project, by SSF in the RISE project, by the CUGS graduate school at Linköping University, by ARC Discovery grant, DP0557972: Enhancing MDA, at ITEE, University of Queensland.

References

[1] P Fritzson, P Aronsson, H Lundvall, K Nyström, A Pop, L Saldamli, and D Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. In Simuation News Europe, 44/45, Dec 2005. http://ww.ida.liu.se/projects/OpenModelica.

[2] Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, 940 pp., Wiley-IEEE Press, 2004. See also: http://www.mathcore.com/drmodelica/

[3] The Modelica Association. The Modelica Language Specification Version 2.2.

[4] OMG, System Modeling Language, (SysML), http://www.omgsysml.org

[5] OMG : Guide to Model Driven Architecture: The MDA Guide v1.0.1

[6] Eclipse.Org, http://www.eclipse.org/

[7] Adrian Pop, Peter Fritzson, A Remar, E Jagudin, and D Akhvlediani. OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging. In Proc of the Modelica'2006, Vienna, Austria, Sept. 4-5, 2006.

[8] David Akhvlediani. Design and implementation of a UML profile for Modelica/SysML. Final Thesis, LITH-IDA-EX--06/061—SE, April 2007.

[9] J.Ferreira, J. Estima, Modeling hybrid systems using Statechars and Modelica". 7th IEEE Intl. Conf. on Emerging Technologies and Factory Automation, October 1999, Spain.

[10] MathCore, MathModelica http://mathcore.com

[11] Dynasim, Dymola, http://dynasim.com

[12] Yves Vanderperren, Wim Dehane, SysML and Systems Engineering Applied to UML-Based SoC Design, Proc. of the 2nd UML-SoC Workshop at 42nd DAC, USA, 2005.

[13] André Nordwig. Formal Integration of Structural Dynamics into the Object-Oriented Modeling of Hybrid Systems. ESM 2002: 128-134.

[14] Adrian Pop, David Akhlevidiani, Peter Fritzson, Towards Unified System Modeling with the ModelicaML UML Profile, EOOLT'2007, www.ep.liu.se/ecp/024/, Berlin, July 30, 2007.

[15] Adrian Pop, David Akhvlediani, Peter Fritzson: Integrated UML and Modelica System Modeling with ModelicaML in Eclipse, The 11th IASTED Int. Conf on Software Eng. and Appl. (SEA 2007), Cambridge, MA, USA, Nov 19-21, 2007.

[16] Brenton Atchison and Alena Griffiths. A Product-Based Assurance Model for Mixed-Integrity Markets. In the 7th Australian Workshop on Safety Critical Systems and Software (SCS'02), Adelaide. Conf. in Research and Practice in Inf. Tech, Vol. 15. P. Lindsay, Ed., Australian Computer Society, 2002.

Anthony MacDonald, Danny Russell and Brenton Atchison. Model-driven Development within a Legacy System: An industry experience report. Proc. of the 2005 Australian Software Engineering Conf. (ASWEC'05), IEEE Computer Society, 2005.