Ideas for Security Assurance in Security Critical Software using Modelica

David Broman, Peter Fritzson
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{davbr,petfr}@ida.liu.se

Abstract

Due to the increasing number of vulnerabilities in software systems and customers' need to trust the producers' development process, third party security evaluations, such as Common Criteria (CC), are today commonly used to provide assurance of security critical software. Modelica is a modern, strongly typed, declarative, and object-oriented language for modeling and simulation of complex systems. In this paper we sketch two ideas for improving security assurance, by expanding the scope of Modelica into also becoming a declarative modeling language for other application areas than simulation.

Keywords: security assurance; secure software; Modelica; serialization; stream; automated testing, unit testing;

1 Introduction

Software development is a complex process and since the number of software systems increase in our society, so do the number of programming flaws that result in vulnerabilities and security threats. The Coordination Center at Carnegie Mellon University (CERT) [2] collection of reported vulnerabilities shows in Figure 1 that the number of vulnerabilities has dramatically increased during the last 5 years. The number of reported incidents has increased to such proportions, that CERT decided in the year of 2004 to stop publishing the statistics.

Customers in the commercial, government, and military sector are totally dependent on the software vendors and how they address quality and security of their products.

The single most important technology choice a vendor makes is, according to [25], the choice of programming language. There are many different factors that impact the choice of language, where performance, expressiveness, and familiarity are some key factors. Many performance critical applications tend to select C as the language of choice, which leads to more security risks compared to other safer languages such as Java^{TM}.

There exist security auditing tools, such as RATS [9], which scan source code for vulnerabilities. A number of such publicly available tools for preventing buffer overflows are tested in [26], where it was shown that only 50% of the attack scenarios were successfully prevented.

In the short term, such tools and techniques are necessary to mitigate the number of vulnerabilities in existing software, but in the long term, new software must, from our point of view, be developed using languages that help the programmer avoid making mistakes that lead to security vulnerabilities.

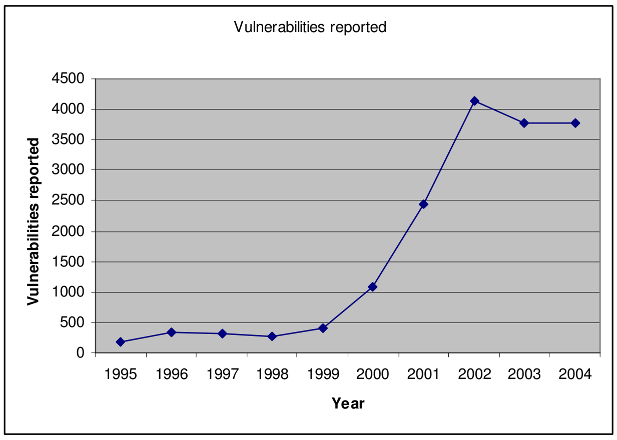


Figure 1. CERT/CC Statistics - Vulnerabilities reported.

1.1 Modeling and Simulation for Safe Engineering Practices

Recent years have witnessed a significant growth of interest in modeling and simulation of engineering application systems. A key factor in this growth has been the development of efficient equation-based simulation languages, with Modelica as one of the prime examples. Such languages have been designed to allow automatic generation of efficient simulation code from declarative specifications. A major objective is to facilitate reuse and exchange of models, model libraries, and simulation specifications.

The Modelica language and its associated support technologies have achieved considerable success through the development of domain libraries in a number of technical areas. By using domain-libraries complex simulation models can be built by aggregating and combining sub-models and components from various physical domains.

The concept of *safe engineering practices* has been one of the most important guidelines when designing Modelica. This made it natural to make Modelica a statically strongly typed language, which allows the compiler to check the consistency of a design before it is executed, in contrast to dynamically typed languages such as Matlab.

The ability of static checking has also influenced the design of conditional equations and the ongoing design of variant handling features in Modelica. Moreover, the language allows support for standardized physical units, thus enabling tools for unit checking of relationships and connections between interfaces. A third possible level of checking is through design rules within application-specific libraries, which can be enforced via assert statements. These properties taken together give a good foundation for safe engineering practices, even though more work is needed to further increase the safety quality level.

1.2 Assurance of Security Critical Software - the Problem Background

Due to the fact that customers in areas such as commercial industry, governments, and the military sector are dependent on the software vendors and the quality of their software and software development process, the need for third party evaluation of security critical software has increased.

In the past, there existed various certification programs for evaluation of IT products, where ITSEC and TCSEC (often called the orange book) have been the international standards. These programs have recently been replaced by a new standard called Common Criteria, ISO Standard 15408, which is accepted by most governments in Europe and the United States [17].

The security evaluation of Common Criteria is divided into two main areas: (1) Security functions that the IT system is claimed to support and (2) the methods used to assure that the product is capable of what it claims to handle.

The second area concerning security assurance requirements is defined in [18] and handles assurance in areas such as configuration management, delivery and operation, development, guidance documents, life cycle support, tests, and vulnerability assessment.

In this paper, we are concerned with the assurance of the development process, especially issues that are related to the programming language. We define here the concept of *assurance* as the way for a vendor or a third party evaluator to show that a software product is indeed secure. By the term *secure software*, we share the same view as described in [12], where security has a strong connection to reliable software, i.e. if reliability is important, so is security.

From the authors experience by working in the commercial industry with both Common Criteria evaluations and FIPS 140-2 validation of cryptographic modules [19], the following two problems and challenges exists:

- How can the vendor guarantee and an evaluator assure that possible runtime errors, such as trapdoors (a hidden backdoor implemented in the software), buffer overflows (buffers are filled with more data than it can hold) or memory reuse (secrets are exposed, since reallocated memory was not cleaned) etc. are limited or eliminated [10].
- How can the vendor guarantee and an evaluator assure that the formal architecture design and specification of the product is actually what was implemented?

1.3 Scope

In the rest of this paper we first give a brief overview of the Modelica language, how it is designed today, and its current main applications. We will discuss why Modelica as a language has interesting concepts and properties, which have the potential to mitigate the problematic issues described in the last subsection. We will suggest two conceptual ideas of language improvements, which can enable the language to be used in new application domains, especially where secure software development is in focus. The purpose of the work is not to describe or prove the correctness of an implementation of the ideas. Instead, our aim is to explain the concepts, to discuss potential strengths and weaknesses, and to suggest further research in the area.

1.4 Paper Overview

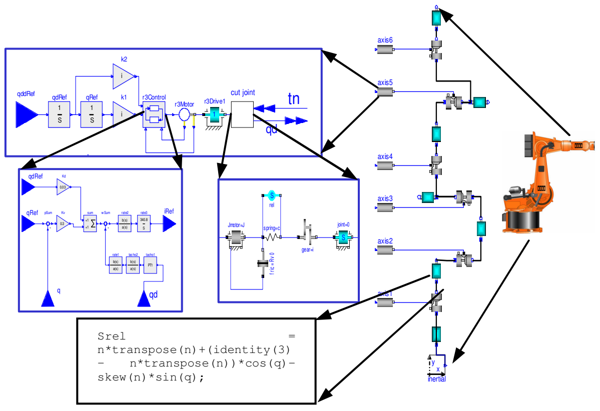
The paper is structured as follows. Section 2 gives a brief overview of the Modelica language and concepts available today for modeling and simulation. In section 3 we conceptually describe and discuss our first language improvement idea regarding models for protocols and safe streams. Section 4 briefly describes our second suggested language enhancement: language support for automated testing. Finally, section 5 states our conclusions and gives recommendations for further research.

2 Overview of Modelica

In the fall of 1996, the Modelica Design Group was created. The group started working towards standardization and unification of object oriented mathematical modeling languages by defining a model description language named *Modelica* for modeling dynamic behavior of engineering systems, intended to become a de facto standard. This language has been continually developed through a series of design meetings and participation of many partners. A formal organization named *Modelica Association* [16], was formed in 2000 in Linköping, Sweden, in order to promote and support this work. In March 2005, the most recent version 2.2 of the Modelica language was released. Modelica is superior to most current technology in modeling and simulation mainly for the following reasons:

- *Acausal modeling*. Modeling is based on equations instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases re-usability of model components, since components adapt to the data flow context in connection structure in which they are used.
- *Object-oriented physical modeling of multiple domains*. This technique makes it possible to create model components that correspond to physical objects in the real world, in contrast to established techniques that require conversion to signal blocks. For application engineers, such ”physical” components are particularly easy to combine into simulation models using a graphical editor. The object-oriented methodology is employed to support hierarchical structuring, reuse, and evolution of large and complex models.
- The Modelica approach to multi-physics simulation enables *real-time simulation* with short deadlines not possible using loosely coupled approaches to multi-physics through connection of present simulation applications for different application domains.

The following figure shows hierarchical component-based modeling using the Modelica technology, with hierarchical decomposition of a design, and strongly typed checkable connections between system components.



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

Figure 2. Hierarchical model of an industry robot, including components such as motors, bearings, control software, etc. At the lowest (class) level, equations are typically found.

2.1 Safe Engineering Practices and Checkable Models

What do we mean by the term safe engineering practices? When constructing a system such as a car, a train, or a nuclear power plant, we would like the design to be safe in the sense that it should exhibit unexpected behavior that can cause accidents. One way to increase the safety of a design is to have well specified components, interfaces, and system architecture, and to be able to verify the properties of the design against formal requirements. We have two groups of verification techniques:

- *Formal verification* techniques allow the consistency of the design to be checked before it is used. Here we include static checking of type constraints for strongly typed languages, unit checking of relationships in physical systems models, and checking domain-specific design guidelines by analyzing the application model code, as in the model checking approach mentioned briefly below. If an inconsistency is found, static model debugging can be used to find the probable causes of the problem.
- *Dynamic verification* techniques, i.e. a kind of systematic testing, will systematically test the design by executing it for many combinations of design parameters that hopefully well represents the design space of the implemented system.

A third technique is to make multiple independent and redundant designs, let them execute in parallel, and use a majority vote mechanism if there are differences.

In practice, both formal verification and dynamic verification techniques are typically employed to ensure maximum safety for critical systems.

2.2 Graphic Model Configuration

Another aspect of safe engineering practices is reusing well-tested simulation model components through an easy-to-use graphical user interface, as depicted in Figure 3, where the tool checks that connected ports are type compatible.

The MathModelica graphic model editor allows picking components from the library windows to the left, dragging these components icons into the drawing area in the middle, and connecting these by lines that represent communication or attachment between the components.

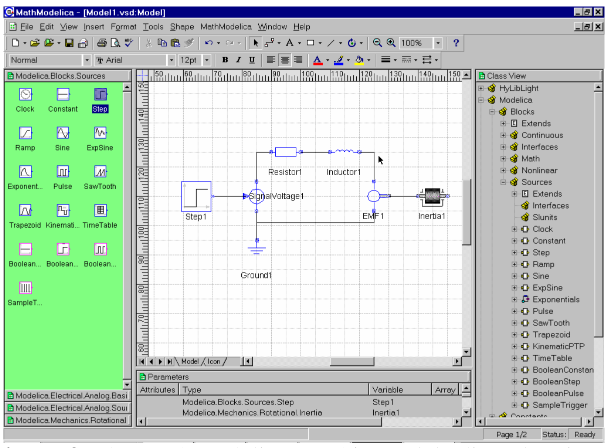


Figure 3. The MathModelica graphic model editor showing a simple electro-mechanical DC-motor model.

2.3 The Future of the Modelica Language

The development of the Modelica language has until now been focusing on applications concerning modeling and simulation of systems. Due to the strong foundation of the language it has the potential to become a general purpose modeling language suitable for development of application in other areas such as *data communication services*. Further, because of fundamental language design features of Modelica, the steps to mitigate security issues described in previous section seem within reach.

Modelica currently has a strong static type checking mechanism, which eliminates many runtime errors at an early stage. Further, the concept of components and visual representation makes it very suitable for representing the formal architecture design with a direct connection to the concrete runtime implementation. Thus, the Modelica language itself has already several properties that handle the above described problem issues.

Our suggestion is that Modelica should be extended in two main areas to further enhance the ability of the language to provide support in the area of specification and development of security critical software. The idea concept of models for protocols and safe streams is described in the following section and the idea of language support for automated testing is discussed in section 4.

3 Models for Protocols and Safe Streams

In this section will we give a brief overview of our idea to add data streams to Modelica and also describe the concept of models that can operate on these streams.

3.1 Protocol, Model and Executable Representation

The concept of protocol may have different interpretations in different contexts. In this text we define a *protocol specification* as the entity defining *data communication*, such as Internet protocols listed in [21]. We use the same interpretation for a protocol as described in [6], where a protocol for data communication is defined as a set of rules that covers all aspects of information communication. A protocol consists in this interpretation of three key elements:

- *Syntax* – the structure of the data sent between the two peers. This could for example be the types of data and the order of the data.
- *Semantics* – indicates the meaning of the data and how the pattern of data should be interpreted. For example, a four bytes field can mean both that it is a length value and that it is a date.
- *Timing* – basically refers to when data should be sent and how fast it should be sent.

In the description of our intention in this paper, we will only consider the first two elements of the protocol; the syntax and the semantics. Timing issues are outside the scope of this description.

Figure 4 shows our definition of the relation between the protocol specification, protocol model and executable program.

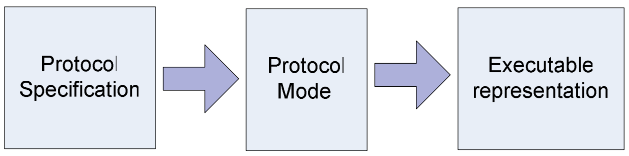


Figure 4. Relation between Protocol specification, Protocol Model and Executable representation.

Here the *protocol model* can be seen as a precise description of the syntax and semantics of the protocol specification. It is this model that should be possible to implement as source code in the Modelica language. The protocol model here states *what* the goal is, without expressing *how* to achieve this. This follows the declarative view of Modelica described in [8] and results in that the *protocol model* can be compiled into an *executable representation,* i.e., a representation where the *how* of the model is given.

The purpose for the protocol model is not to be a specification language for proof engineering, but to enable a protocol to be implemented in a simple, expressive, and still efficient way. The process of implementing protocol models can also be seen as *meta-programming*, i.e., a program that is used to create a program. If, for example a model is implemented in Modelica and then C code is generated and then later used by a C application, the model and Modelica are used as a program generator.

3.2 Streams, Producer, Consumer and Stream Objects

The problem of transferring abstract data types (ADTs) has been investigated since the concept of distributed systems appeared. According to [20], one of the first detailed descriptions of such a design was stated in [11], where it is described how primitive data types can be transferred between distributed systems. Further, in [1], the authors describe a marshaling algorithm of network objects, i.e., an algorithm for transforming values of data into a stream of bytes.

These works are similar to the serialization concept available in Java^{TM} and described in [24]. Here, *object serialization* is defined as the process of creating a serialized representation of objects or graph of objects. Both object values and types are serialized so that they can be recreated. The opposite process is called *deserialization*, i.e. to recreate the object from a serialized stream.

In Figure 5, a schematic view is given of how we define the protocol model's correlation to a *stream*.

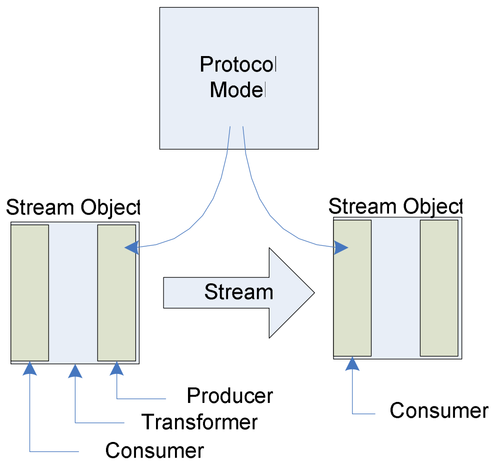


Figure 5. Protocol model that describes both the producer and the consumer of a stream.

We use a similar concept of stream as the one given in [23], where a stream is defined as a potentially unbound list of tokens. A *token* is here defined as a data item of arbitrary type. The entity that is creating the stream is called the *producer* and the entity which is reading messages from the stream is called the *consumer*.

In Figure 5, we can see a data stream as defined between two entities called *stream objects*. These objects can read or write data from or to the stream. If a stream object is both a consumer and a producer, it can be viewed as a *transformer*, which transforms one stream to another.

At the top of Figure 5, we see the protocol model and two arrows pointing to both the producer and the consumer of the stream objects connected to the stream. This means that the same protocol model is used for both executable representations; the producer and the consumer. This property is possible since the data flow in the model is not needed to be defined in the protocol model, i.e. the protocol model should have the *acausal modeling* property.

This concept has similarities to the concepts of Java^{TM} serialization and streams [24], with two design differences:

- Focus in our concept is on the expressiveness of describing external representation, i.e., the way the stream can be created and interpreted. Java serialization do have APIs for customizing the external representation, but the design goal of the Java approach is, among other things, to implement a structured and efficient way to serialize whole objects, including class information.
- Our suggestion is that the model should represent relations between elements in the model in a declarative way by using equations and in this way keep the acausal modeling property.

3.3 Data Communication Examples

In this subsection we will give small examples on how protocol models might be expressed in Modelica syntax. Note that this is not an implemented or working syntax and even does not have the purpose to become so. The goal of the code fragments given in this section is only to illustrate and simplify the explanation about the idea concept. If it is possible to create a compiler implementation that actually executes these models is out of the scope of this paper and is suggested for further research.

To illustrate our example, we introduce a new class section called a *stream section*, starting with the keyword `stream`. Figure 6 shows the syntax of a stream section.

```
stream <type_name> using
   ...
   <variable declarations>
   ...
<some keyword>
```
Figure 6. Syntax of a stream section.

The section can be read as: A `stream` of tokens of type `<type_name>` can be created by `using` the `<variable declarations>`. The stream section contains one or more variable declarations and is terminated by the appearance of one of the five keywords `equation`, `public`, `protected`, `algorithm`, `initial`, or `end`. The *type name* declared between the keyword `stream` and `using` is the type of tokens that the section should be able to serialize to, or be deserialized from, i.e., all variable declarations in the section should be able to be transformed to and from a stream of tokens of type `<type_name>`.

Our first example illustrates a protocol model of a subset of the Remote Authentication Dial In User Service (RADIUS) defined in request for comments 2865 [22]. The main packet format is expressed as a Modelica class, listed in Figure 7. If the reader is familiar with Modelica syntax, it is obvious that some new syntactic elements are added to the syntax.

In the example, the section `stream` states that the section on line 4 to 8 should be able to be serialized to and deserialized from a stream of `Byte` tokens. Note that variable `attribLength` is outside the section, and is therefore not part of the stream.

The type `Byte` should be interpreted as an abbreviation of an `Integer` with min value 0 and max value 255. A class that contains at least one `stream` section should be viewed as being a protocol model, i.e., it is possible to serialize and deserialize according to the model's syntax and semantics.

The serialization of `code`, `identifier`, and `authenticator` to a stream of bytes should be straight forward to see, but how can `length` with type `UInt16` be serialized to a stream of tokens of type `Byte`?

```
01: class RADIUS
02:   Integer attribLength; //Help variable
03: stream Byte using
04:   Byte    code;
05:   Byte    identifier;
06:   UInt16  length;
07:   Byte    authenticator[16];
08:   Byte    attributes[attribLength];
09: equation
10:   attribLength = length -
11:                  sizeof(code) -
12:                  sizeof(identifier) -
13:                  sizeof(length) -
14:                  sizeof(authenticator);
15: end RADIUS;
```
Figure 7. RADIUS model.

The key is that the type `UInt16` must also be serializable to the same token stream, in this case a stream of type `Byte`. Generally, all variables declared in a `stream` section must be recursively serializable to the token type specified in the header of the `stream` section. Figure 8 shows a listing of the `UInt16` model. The first notable difference compared to Figure 7 is that we have multiple `stream` sections. This should be interpreted as that the model is serializable to and deserializable from many different types of token streams; in this case both streams of type `Byte` and streams of type `Integer`. If we recall Figure 5, the `UInt16` model can be seen as a *stream object* which can transform from a byte stream to an integer stream. It can also be seen as the class `UInt16` can be implicitly type converted from and to both an `Integer` and an array of type `Byte` with two elements.

This is actually what happens if we look at the equation section of Figure 7. On line 10 to 14, an equation relation is stated between the size of the array `attributes` using variable `attribLength`, the variable `length` declared on line 6 and the size in `Byte` of all attributes before `attributes`. As we can see, we have in a declarative way expressed the size of the array `attributes`, with information that is only available in attribute `length`.

In the RADIUS protocol, the `length` field indicates the number of bytes in the whole packet. With the information stated in the RADIUS class, it is possible for an executable representation to deserialize a stream and at the same time dynamically bound check all attributes in the `stream` section. The process is reversible since the variable `length` can be calculated through the equation and then serialized to a stream of element of type `Byte` using the semantics of class `UInt16`.

```
01: class UInt16
02:   String  strValue;
03: stream Byte using
04:   Byte   data[2];
05: stream Integer using
06:   Integer value;
07: equation
08:   strValue = String(value);
09:   if from(Byte) then
10:     value = leftshift(data[0], 8) or
11:              data[1];
11:   else if from(Integer) then
12:      data[0] = rightshift(value, 8);
13:      data[1] = value and 0xff;
14:   end if;
15: end UInt16;
```
Figure 8. Unsigned 16-bit Integer class.

If we take a deeper look into Figure 8, the `if` statement of the `equation` section, starting on line 9, describes the conversion between an array of bytes and an integer. This section uses many constructions which are not available in Modelica, but they should be fairly self-documenting. We have introduced a keyword `from`, which returns the value `true` if the class is deserialized from a stream of the type given as the argument. The functions `leftshift` and `rightshift` perform bit-wise shift on the first argument.

The last observation we should make in the `UInt16` model, is that we have defined a `String` attribute called `strValue` on line 2. This has no direct effect on the model defined in Figure 7, but it illustrates that a model can, for example be used to define explicit type conversions. In this example, it would be possible to access the string representation of `UInt16` by using a dot notation, e.g., use the syntax `val.strValue`, to convert the variable `val` of type `UInt16` to a `String`.

We have in this subsection described an uncomplicated case where only a portion of the fairly simple RADIUS protocol was modeled. We have also showed that the direction of the data flow in the model is not fixed, i.e., acausal modeling is possible. The idea would in our option be powerful if complicated protocols, such as the Transport Layer Security (TLS) [3], are modeled. TLS has many complicated nested constructions with several semantic relations between the elements. In this case, our intention is that even operations such as symmetric and asymmetric encryptions should be possible to describe in a declarative approach, where the data flow is not stated in the model.

3.4 Safe Data Types and Streams

Sensitive data, such as passwords and cryptographic keys allocated in the computers memory must be especially carefully handled, to avoid security threats such as *memory reuse.* In the Federal Information Processing Standard Publication (FIPS 140-2), which specifies requirement for cryptographic modules, there are specific requirements for destruction of sensitive data, called *zeroization*, after usage, i.e., the memory cells holding the data must be explicitly overwritten [19].

Most programming languages do not have any method of avoiding sensitive data to be swapped down to disk, with the exception of C where a library function mlock() can be used for this purpose. Further, many high level languages support data types which are immutable, i.e. the memory area cannot be overwritten by the programmer [25], which makes it impossible for the programmer to implement *zeroization.*

We therefore propose that there should be a keyword, for example `safe`, which indicates that a variable will be safe in the context that it will not be swapped to disk and that it will be automatically *zeroized* when it is not used anymore. The following code fragment illustrates how a text string containing a secret password can be declared as a safe attribute in the example record *Secret*, shown in Figure 9.

```
record Secret
         String username;
  safe   String password;
end Secret;
```

Figure 9. Example of using the *safe* keyword.

3.5 Discussion

The above conceptual demonstration of the idea of protocol models and streams should by itself state the flexibility and power of what such a declarative solution could give.

It would in the same sense as for Java eliminate all threats regarding buffer overflow attacks [25], since array bound checking is done dynamically automatically according to the protocol model. However, the risk is that implementing a similar solution as described above, can either be very hard to design and implement or result in low performance. Since we are only theoretically discussing the ideas, the concepts must be investigated further by designing and implementing a practical test solution.

As mentioned above, the concept of protocol models can have direct usage as a meta-programming language for generating safe C-code, which then can be used by other applications. In such a scenario, the Modelica language would act as meta-language for generating safe library functions.

The concept of data streams may also have other applications, even for simulation cases. The concept can be used for feeding the simulation with test data from for example a file.

A possible drawback and a potential risk is that the expressive language above is hard to implement as expressive as shown, which can result in that it is not simpler or safer to implement the model declarative compared to an imperative solution.

However, besides the potential risk of not being able to implement this idea, we think it improves the chances of giving better assurance for software by the following reasons:

- It reduces or potentially eliminates vulnerabilities such as buffer overflows and memory reuse.
- The declarative syntax with acausal data flow makes the code cleaner, and therefore easier to find implemented security flaws such as trapdoors.
- Together with the Modelica concept with visual modeling of components, it has a potential to make the correspondence between high level design and implementation details easy to follow.

4 Language Support for Automated Testing

All programming errors can of course not be discovered at compile time using techniques such as static type checking. Many human errors are hard or impossible for the compiler to distinguish from behaviors implemented by design.

The usual approach to control such a behavior is by setting up automated tests in a unit test system as part of the development environment. In this section, we will give a brief description of our view of automated tests and suggest a approach connected to the Modelica language.

4.1 Concept of Automated Unit Testing

Unit testing is a central part of the software development method *Extreme Programming* (XP). A gentle introduction of rules and practice of XP can be found in [5].

Basically, unit testing is a way to automate the testing by implementing tests for different parts of the source code, call units. A *unit* in this context is defined as a part of the system that can be tested and also is deterministic, i.e. it will for the same input return the same output every time the test is executed, assuming that the program or test has not been changed. A unit can for example be a class in Java or a function in C.

One of the key concepts of XP is to make the developer to program the tests first; before implementing the source code for the unit. According to this method, it helps the programmer to concentrate on what really needs to be done. Further, it is stated in [5] that it actually helps other developers to understand someone else's code, since it can be easier to browse the tests instead of the actual implementation.

Another central part in XP is that the unit test code is released into the source repository together with the source code, i.e. all code must have tests associated with it before the code can be released.

It is often argued that writing the tests costs too much both regarding time and money, especially when it is close to deadline. However, there are practical studies made, such as [4], which claims that the cost for writing the tests may be exaggerated.

Automated unit tests have another clear advantage when it comes to development of code by large teams, which is the ability to guard functionality from being accidentally broken. This method of protecting working code is sometimes described as regression testing and has for example in the success story report [14] been shown to improved code quality, especially in environments where the source code is ported to many platforms (both hardware and operating systems).

4.2 Language Support

There are many testing frameworks available for various languages, such as unit test programs related to Extreme Programming [27].

One of the most popular unit testing frameworks available for Java is JUnit [13], which has become the de facto standard for Java. In other languages such as C and C++, there are many available frameworks, such as CppUnit, Boost.Test, Unit++, CxxTest etc. A comparison of the frameworks is given in the article "Exploring the C++ Unit Testing Framework Jungle" [15].

Today, there is no publicly available unit testing framework available for Modelica, but there might exist proprietary implementations used by different organizations. Without being specific about how unit testing should or can be implemented for Modelica, or which constructions that are suitable for testing (i.e. functions, classes etc.), we suggest that a specification about how the test should be expressed should be standardized. For example, the Modelica *annotation* mechanism can be used to introduce standardized testing annotations. The Modelica *assertion* facility is also useful, e.g. to introduce certain checks.

4.3 Discussion

We have so far briefly described why automated unit testing is generally considered a good idea, but why do we want to connect the unit tests with the language specification?

Basically, our philosophy is that if the encoding syntax is standardized in an early stage in a programming language's history, it is more likely that developers will take advantage of the feature and use it as part of the development concept.

Further, if the syntax of the tests is standardized, different software vendors that implement environments for the language will be compatible even with test suits, not just the source code. A comparable successful concept is the graphical annotations of Modelica, where today at least three vendors have implemented support for the graphical annotation [8].

It is of course a risk that a standardized test specification syntax can be useless, if it is not carefully designed from the beginning. Today, there are many successful environments for unit testing, especially JUnit for Java^{TM}, which should enable the ability to design a future-proof standardized testing syntax for the Modelica language.

5 Conclusions and Further Research

We have in this paper described two language ideas using an abstract conceptual approach, without any practical research regarding implementation complications. Our goal has been to explain the importance of language design in order to improve the security quality of software systems. Further, by improving the quality of the software and by using more powerful abstraction mechanisms in the language, we claim that this will improve the possibility to give better security assurance for software.

We have previously noted that there are many question marks regarding the possibility of actually designing and implementing the ideas described in this paper, and we therefore suggest further research, particular in the area of stream concept connected to equation based modeling. Besides practical effort to implement and design a subset of the protocol model concepts, a survey in the area of streams, serialization and equation based modeling would be a suggested step forward.

6 Acknowledgements

We are grateful to the readers, especially John Wilander and Levon Saldamli, for helpful comments on the paper.

References

[1] A. Birrell et al. (1994). "Network Objects", Digital Equipment Corporation Systems Research Center Technical Report 115, February 1994.

[2] Cert Coordination Center. (2005). "Cert/CC Statistics 1988-2005", http://www.cert.org/stats/ Accessed: 2005-05-05.

[3] T. Dierkes et al. (1999). "The TLS Protocol Version 1.0", Request for Comments: 2246.

[4] Michael Ellims et al. (2004). Unit Testing in Practice". In Proceedings of the 15th International Symposium on Software Reliability Engineering (02-05 Nov, 2004).

[5] www.extremeprogramming.org (2005). "The Rules and Practices of Extreme programming", http://www.extremeprogramming.org/rules.html.

[6] Behrouz Forouzan. (2001). "Data communication and networking", ISBN 0-07-232204-7, 2nd edition, McGraw-Hill Higher Education.

[7] Peter Fritzson, et al. (2002). The Open Source Modelica Project. In Proceedings of The 2nd International Modelica Conference, 18-19 March, 2002. Munich, Germany See also: http://www.ida.liu.se/~pelab/modelica/OpenModelica.html.

[8] Peter Fritzson. (2004). "Principles of Object-Oriented Modeling and Simulation with Modelica 2.1", 940 pp., ISBN 0-471-471631, Wiley-IEEE Press.

[9] FSF/UNESCO Free Software Directory. (2005). "RATS- Security auditing tool for source code" http://directory.fsf.org/devel/build/RATS.html, Accessed: 2005-05-09.

[10] Susan Hansche, John Berti, Chris Hare. (2003). "Official (ISC)$^{2®}$ guide to the CISSP$^{®}$ exam", Auerbach Publications.

[11] H. Herlihy and B. Liskov. (1982). "A Value Transmission Method for Abstract Data types", ACM Transactions on Programming Languages and Systems, vol. 4, no. 4, October 1982.

[12] Michael Howard and David LeBlanc. (2003). "Writing Secure Code", 2nd edition, Microsoft Press.

[13] JUnit.org. (2005). "JUnit, Testing Resources for Extreme Programming", http://www.junit.org/index.htm Accessed 2005-05-05.

[14] Michael Long. (1993). "Software Retression Testing Success Story". In Proceeding of the International Test Conference (17-21 October, 1993).

[15] Noel Llopis. (2004). "Exploring the C++ Unit Testing Framework Jungle". http://www.gamesfrom within.com/articles/0412/000061.html Accessed: 2005-05-05.

[16] The Modelica Association. (2005). "The Modelica Language Specification Version 2.2", March 2005. http://www.modelica.org.

[17] NIAP. (2004). "Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model", version 2.2.

[18] NIAP. (2004). "Common Criteria for Information Technology Security Evaluation, Part 3: Security Assurance Requirements", version 2.2.

[19] NIST. (2001). "Security Requirements for Cryptographic Modules", FIPS 140-2, http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.

[20] Lucasz Opyrchal and Atul Prakash. (1999). "Efficient object serialization in Java". In Proceedings of 19th IEEE International Conference on Distributed Computing Systems Workshops (31 May-4 June, 1999).

[21] J. Reynolds and S. Ginoza. (2004). "Internet Official Protocol Standards", Request for Comments: 3700.

[22]C. Rigney et al. (2000). “Remote Authentication Dial In User Service (RADIUS)”, Request for Comments: 2865.

[23]Peter Van Roy and Seif Haridi. (2004). “Concepts, Techniques, and Models of Computer Programming”, ISBN 0-262-22069, The MIT Press.

[24]Sun Microsystems. (2003). “Java^{TM} Object Serialization Specification”, http://java.sun.com/j2se/1.5/pdf/serial-1.5.0.pdf, Accessed: 2005-05-05.

[25]John Viega and Gary McGraw. (2002). “Building Secure Software – How to avoid Security Problems the Right Way”, Addison-Wesley.

[26]John Wilander and Mariam Kamkar. (2003). "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention" In Proceedings of 10th Network and Distributed System Security Symposium (NDSS'03). (San Diego, California, USA, February 5-7, 2003).

[27]XProgramming.com Software Downloads. (2005). Unit testing frameworks http://www.xprogramming.com/software.htm, Accessed: 2005-05-06.