Experiments on Processing and Linking Semantically Augmented Requirement Specifications [1]

Daniel Conte de Leon and Jim Alves-Foss
Center for Secure and Dependable Systems
University of Idaho
[jimaf , danielc]@csds.uidaho.edu

Abstract

Today's critical systems increasingly rely on computers and software. However, market pressure, problems in the application of formal methods, and ineffective traceability techniques may all exacerbate the difficulty of applying adequate assurance techniques to the design and development of safe and trustworthy systems. Necessity dictates that engineers target critical sections that require formal verification and high reliability. To achieve this objective, we need to implement and maintain linking relationships among system work products and be able to propagate criticality of requirements to subsequent work products. We successfully implemented traceability between an informal requirements document and its formal specification using two new XML-derived markup languages. We addressed the issues of specifying and propagating criticality of requirements and consistency of this determination within and between work products.

1. Introduction

Computer systems are being used in increasingly more applications every day. The nature of some of these systems makes them intrinsically dangerous to the environment and the people who use them. Consider systems such as an aircraft, a power generation plant, a power distribution grid, a tanker, a medical machine, or a database with credit or medical records -all are critical since a malfunction could lead to unacceptable consequences- [20]. As Nancy Leveson states so accurately in [27] -"Software is a human product"- and computerized critical systems rely on software. However, the consequences of malfunction are unacceptable.

[1] Portions of this work were funded by grant #60NANB1D0116 from the National Institute of Standards and Technology, United States Department of Commerce.

Therefore, we must allocate all of our resources to minimize the occurrence of errors and their consequences.

The rest of the paper is organized as follows: Section 2 expands on critical systems, their faults, and approaches to prevent them. Section 3 defines system work products and explains XML as our implementation vehicle for representation and linking. Section 4 describes the Steam Boiler case study. Section 5 explains how we semantically augmented informal requirement specifications by using XML markup. Section 6 briefly describes how we are using our framework and linking techniques to navigate system work products. Section 7 describes how we are propagating criticality properties in order to help engineers discover sources of faults. Sections 8 and 9 state Related Work and Conclusions respectively.

2. Critical Systems

A critical system is one that, if operating incorrectly, could cause loss of life, damage to life or the environment, or other great losses (financial, organizational, national, or global) (adapted from [9, 20]). Critical systems may be classified, among others, as safety-critical, security-critical, and mission-critical, depending on their purpose and the risks associated with their incorrect functioning. For example, an electric power generation plant is a safety-critical system, since its incorrect operation could cause loss of life or harm to the environment. An organizational database with sensitive information is a security-critical system, as a malfunction could generate financial losses. An aircraft is both a safety-critical and a mission-critical system; the power distribution grid is a mission-critical system.

A safety-critical system must protect life and the environment from damage while achieving its functional objectives; if it does, we say the system is safe. A security-critical system must protect assets and itself from unauthorized access while achieving its functional objectives; if it does, we say the system is secure. A mission-critical system must conduct its mission even under adverse or unplanned conditions; if it does, we say

the system is survivable. These concepts can be combined into *system dependability*. In [5, 24, 25, 34] the authors describe dependability as a property of a system, which integrates the following attributes: reliability, availability, safety, security, survivability, and maintainability; they define dependability as the ability of a system to deliver justifiably trusted service; in other words the system is trustworthy.

2.1 Errors in Critical Systems

Lamentably, there have been cases where safety-critical systems operating in an unsafe state have caused loss of life [23, 26, 39]. In addition, innumerable articles describe losses associated with unauthorized access to and/or use of security-critical systems.

A failure is the consequence of an error affecting the service delivered by a system. An error occurs when a system or a portion of it enters an unplanned state. An error is the consequence of a fault. A fault can be dormant or active [5]. In [28] the author identifies two main causes for the occurrence of safety-related errors in critical systems: 1) a discrepancy between requirement specifications and the requirements needed for correct functioning and 2) a misunderstanding of interfaces between the system and its controller software. In other words, errors that compromise the correct functioning of the system are mainly the result of faults introduced due to mismatches between the real system under certain operating conditions and the system models. Our methodology aims to help system practitioners discover these types of mismatches.

2.2 Approaches for Preventing Faults in Critical Systems

We currently identify four strategies to deal with faults in critical systems usually named *Fault Avoidance*, *Fault Elimination*, *Fault Tolerance*, and *Fault Evasion* [5, 9, 20, 25]. These approaches have been partially successful in removing faults or preventing errors in their own domains. However, today's critical systems are complex combinations of physical and control devices, and controller software; some are highly networked and distributed. We need a combined approach to successfully handle these more complex systems [34, 36].

Current requirement and traceability tools provide for coverage analysis (e.g. modules that implement a particular requirement). However, in dependable systems development we need a mechanism to assure that applied verification techniques are appropriate to the criticality of the components being used or created. Usually systems are modeled using a set of specification languages and tools. Leveson points out several characteristics that specification languages and development tools should have in order to help engineers build better systems. We recall two of those characteristics here: great flexibility to adapt to different mental models and problem solving approaches, and ability to "clearly and perceptually" display those attributes that are key to solving the problem [27]. Our work aims to support these strategies as well as different development techniques and languages. Our objective is to identify critical sections of a system and discover sources of conflict in the system model by propagating requirements criticality to model components using traceability relationships.

3. System Work Products and the Extensible Markup Language

3.1 System Work Products

The process of developing critical systems requires a myriad of techniques and disciplines working together. Practitioners from different backgrounds and with different ways of solving problems must successfully communicate in order to design and develop a safe, secure, reliable, and/or dependable system. Each discipline has its own methods and languages to represent and communicate information and views of the system. The model of the system is a conglomeration of documents expressed in different modeling and description languages for hardware and software components. Those disparate documents include among others: system intentions [27], system definition and architecture, informal and formal requirement specifications, design specifications (including graphical representations), safety, security, reliability, and hazard analyses, and component descriptions. We call all of these documents *system work products.*

3.2 The Extensible Markup Language (XML)

The Extensible Markup Language (XML) [10] was developed by the World Wide Web Consortium (W3C). XML and its associated technologies are open international standards and non-vendor, non-platform dependent. Among those XML-based technologies we have XML Linking Language (Xlink) [15], XPath [7], Extensible Stylesheet Language and Transformations (XSL and XSLT) [2, 13], Scalable Vector Graphics Language (SVG) [17], Extensible Metadata Interchange (XMI) [33], and Math Markup Language (MathML) [12]. These technologies can greatly improve the way we represent and process system work products. For more information on XML and related technologies, we encourage the interested reader to see the references and some of the abundant literature on the topic.

3.3 System Work Products and XML

Our approach to improve system development is to have a common format for all system work products in order to be able to automatically interpret semantic

information and establish meaningful cross-section linking among system work products. Then, we can use that information to create navigable views of system models and identify sources of conflict. XML and its related technologies offer a flexible implementation platform in where we can achieve our objectives. First, we can have enhanced information, such as requirement or component type or criticality level. Second, we can automatically process this information and generate different views of our system model. Third, we can add and maintain traceability and navigate system work products. Fourth, we can use system work products linking and criticality properties to discover model conflicts that could lead to system faults. In our framework, it is necessary to represent every system work product as an XML derivative. Fortunately, some research work has been done on this track, and we briefly describe it in the Related Work Section of this article.

3.4 TraceML: Linking System Work Products

TraceML is an Xlink-based language for the specification of relationships among sections of system work products. In [3] we described some experiments on using XML, Xlink, and XSLT to associate design specifications with their implementing source code. In [14] we described our framework along with the definition of TraceML. TraceML allows us to define and implement any type of dependency among requirements or any other system work product. Using this framework, we can establish for example, dependencies between a class method, its implemented requirement, and a formal proof of correctness for that method, or a hardware component and its reliability analysis, and be able to navigate the links backwards and forward.

4. The Steam Boiler Case Study

The Steam Boiler case study is one of a few well-published case studies on the formal specification of safety-critical systems. We will use this case study as a reference example to discuss our work. The initial problem was based on an informal requirement document by J.R. Abrial [1] presented to the participants of the Dagstuhl Meeting held in June 1995. Currently, there are more than 20 published papers using different formal techniques to specify the Steam Boiler controller. The Steam Boiler case study offers a wide range of solutions and approaches based on a single original document and it is a real-life critical application tractable in an academic environment. Experiments described in this paper refer to this case study.

One of the objectives of our work is to keep views of system work products as close to their original format [38] while, at the same time, having markup information, which will be mechanically processed. With that purpose, we transcribed and adapted three Steam Boiler documents to our newly created XML-based representations that we will explain throughout this article. Those documents and their correspondents on our experiment are: 1) *"Steam-Boiler Control Specification Problem"* by J.R. Abrial, [1], which we call *Control System Definition Document*, (CSDD). 2) Segments of *"A Constraint-Oriented Specification of a Steam Boiler Controller in Higher Order Logic"* by J. Alves-Foss [4], which we call *Control System Informal Requirements Document* (CSIRD). 3) Segments of *"Specifying and Verifying the Steam Boiler Control System with Time Extended LOTOS"* by A. Willig and I. Schieferdecker [40], which we call *Control System Formal Specification,* (CSFS).

Note that we pre-appended "Control System" to these titles to distinguish them from documents or specifications of the whole system, which have not been explicitly created for this case study.

5. Semantically Augmented Specifications

The starting point of any system development process is a customer intention expressed in an informal requirements specification document (usually written in a natural language). Between such a document and a formal specification of a system, there is inevitably a semantic gap. Even with the use of tools such as artificial intelligence and knowledge representation, extracting and formalizing information stated using natural languages is a human task. Computer systems are currently of little help in transforming informal system definitions and requirements into formal requirement specifications and design specifications. This fact is a cause of today's great difficulty in developing error-free systems. Recall that most errors in critical-systems are consequences of mismatches between the modeled system and the actual system (including the model of the control system).

In order to improve this situation, our approach reduces the semantic gap by augmenting system definitions and requirements specifications with known data about the system components. By using XML markup to semantically augment the information in an informal requirements specification, we allow computer systems to extract and process such information. In this way, software practitioners can express, piece-by-piece, large amounts of information, while automated systems can extract and process that information stated so far by the markup.

In order to test our approach, we designed two new XML-based markup languages and applied them to the steam boiler case study. Those languages are: Requirements Markup Language (RML) and Extended Lotos Markup Language (ELotosML). The first is one is the key to narrowing the semantic gap, and it will be explained in this section. The second one adds a new

piece to the puzzle of XML-represented system work products [3, 14] and it will be explained in Section 6.

5.1 RML: A Markup Language for Informal System Requirements

We developed an XML-derivate language for representing informal requirements called RML. The requirements for our new language were as follows:

- Represent System and sub-system Definitions and Informal Requirements Documents.
- The viewable format of the document should be as close as possible to currently used documents.
- The document should contain information about the system requirements and components, which can be processed by a computer system.
- Allow for incremental refinement of specifications.

Our approach is based on the idea that information contained in an informal requirements specification can be classified in two main categories: *statements* and *objects.*

5.1.1. RML Statements. A statement is a piece of information that describes something we know about the system, such as the definition of the system or a requirement. It defines information--such as an assumption or a constraint --that applies to the whole system, or any set of components.

Statements can be nested to express parent-child relationships; these are used by an XSLT stylesheet to automatically generate an HTML [35] view of the informal requirements, as shown in Figure 1. The XSLT transform translates statements into HTML and automatically inserts the numeration. Cascading Stylesheets (CSS) [41] are used to achieve indentation and text formatting. Figure 1 shows a view of nested informal requirements as stated in our CSIRD and transformed to RML. The RML section that originated this view is shown in Figure 2. A statement can have the following attributes: *name, title, domain, verification-state, criticality-level, type, and belongs-to*. Figure 3 shows a section of the RML's

2.1.2.3 Comparison Of Calculated Values

The comparison of the calculated values to the reported information.

2.1.2.3.1 Reported Values Out Of Expected Range

If reported values fall out of expected range for previously operational components, that component is now considered to have failed.

Figure 1. Sample Parent-Child Informal Requirements View

Document Type Definition (DTD) --a language used to define the syntax of XML documents based on the Extended Backus Naur Form (EBNF) [10]—, which defines the syntax of a statement.

The ***name*** attribute is used to address the statement and to be able to add links from other system work product sections to this statement using XPath. Linking techniques are explained in Section 6. The ***title*** attribute is used in any view referring to the statement as we describe in Section 6. The ***domain*** attribute indicates the kind of statement such as *assumption*, *requirement*, or *constraint*. The ***verification-state*** attribute indicates whether or not the statement has been verified according to the process and criteria associated with the project.

The ***criticality-level*** attribute can have one of three possible values: *True*, *False*, or *Inherited*. *True* indicates that we affirmatively know that this statement is critical--for example, the root requirement. We expect formal verification for this kind of statement. *False* indicates that we know this statement is not critical; consequently, we expect rigorous/formal verification of this statement. *Inherited,* indicates that this statement inherits its criticality from its parent statement. We explain the semantics of criticality propagation in Section 7.

The ***belongs-to*** attribute defines a logic requirement hierarchy and it is used to propagate the criticality attributes. The ***type*** attribute gives more information about the declared statement. For example, we could have a *Functional Requirement* statement or an *Environmental Constraint* statement using a combination of the values indicated by the ***domain*** and ***type*** attributes.

```
<statement
    name ="ComparisonOfCalculatedValues"
    title ="Comparison Of Calculated Values"
    domain ="Requirement"
    verification-state ="Pending"
    criticality-level ="Inherited"
    type ="Functional"
    belongs-to ="UpdateSystemState">
        The comparison of the calculated values to
        the reported information.
  <statement
    name="ReportedValuesOutOfExpected..."
    title ="Reported Values Out Of Expected
    Range">
        If reported values fall out of expected range
        for previously operational  components, that
        component is now considered to have failed.
  </statement>
</statement>
```

Figure 2. Excerpt of the Steam Boiler Informal Requirements Document

```
<!ELEMENT statement
    (#PCDATA | object | statement ) * >
<!ATTLIST statement
    name NMTOKEN #REQUIRED
    title CDATA #IMPLIED
    domain ( Assumption | Requirement |
        Goal | Constraint | Definition | Property
        | Information | Unknown ) #IMPLIED
    type ( Functional | Environmental |
    Unknown ) #IMPLIED
    criticality-level (True | False | Inherited)
    #IMPLIED
    verification-state ( Correct | Conflict |
    Pending ) #IMPLIED
    belongs-to NMTOKEN #IMPLIED >
```

Figure 3. RML-Statement Formal Syntax Rule

A statement contains objects (Section 5.1.2) as well as other statements. Objects contained in a statement refer to components involved in that statement. For example, the requirement shown in Figure 4 (extracted from our CSIRD) contains three objects marked in italics and bold: 1) the *controller* object, which in our internal references is called control system. 2) the *SteamBoilerWaiting message* object, an internal data structure of the control system. 3) the *physical units* object, composed by all devices in the physical system, We describe the structure for the system and its sub-systems in Section 6 along with the generated views of the system model. We describe the object element markup immediately following.

1.1 Boiler Initialization

The ***controller*** will not begin operation until it receives a ***SteamBoilerWaiting message*** from the ***physical units***.

Figure 4. Objects Contained in a Statement

5.1.2. RML Objects. We use object elements to indicate the presence of a system component. Objects can also describe states, variables, or data. Figure 5 describes the formal syntax for an object as stated in the RML DTD. Object elements cannot be nested. An object element has attributes similar to those of a statement.

Attributes of an object hold a similar purpose as their statement counterparts but with a difference in their possible values. The ***name*** and ***title*** attributes have the same purpose of reference to the object and viewable name of an object respectively. The ***domain*** and ***type*** attributes describe the type of object. For example, the controller object of Figure 4 would be qualified as *Component-Software* for domain and type respectively. The *SteamBoilerWaiting* message would be qualified as *Data-Record*. The ***criticality-level*** attribute indicates if the object is critical or not, or if it inherits its criticality from its parent object (indicated by the ***belongs-to*** attribute). This relationship describes the structure of the system respect to its components.

In this experiment, we implemented single criticality inheritance. Multiple inheritance could be achieved by adding multiple ***belongs-to*** attributes.

5.2 Views of RML Documents

We used our newly created RML to state our CSDD and our CSIRD for the Steam Boiler. We opted to use the term *document* for those system work products that are originally stated in a natural language. We will use the term *specification* for those system work products that are originally stated in a formal specification language.

```
<!ELEMENT object ( #PCDATA ) >
<!ATTLIST object
    name NMTOKEN #REQUIRED
    title CDATA #IMPLIED
    domain ( Component | State | Variable |
    Data | Unknown ) #IMPLIED
    type ( Physical | Software | Variable |
    Record ) #IMPLIED
    criticality-level ( True | False | Inherited )
    #IMPLIED
    verification-state ( Correct | Pending )
    #IMPLIED
    belongs-to NMTOKEN #IMPLIED>
```

Figure 5. RML-Object Formal Syntax Rule

Both types of work products can be stated using RML but their viewable formats are different. We created two XSLT stylesheets which transform RML into HTML: SystemDefinitionToHtml and InformalRequirements-ToHtml. Figure 6 shows an excerpt of the first stylesheet that creates the corresponding HTML section for a statement of a CSDD. After applying this stylesheet to our CSDD the output is a document very similar to the original Steam Boiler Control Specification Problem [1]. Figure 7 shows an excerpt of a template from our InformalRequirementsToHtml stylesheet that processes a statement element.

```
<xsl:template   match="statement"
                name="statement">
    <xsl:element name="span">
        <xsl:attribute name="id">
            <xsl:value-of select="@name"/>
        </xsl:attribute>
        <xsl:value-of select="."/>
    </xsl:element>
</xsl:template>
```

Figure 6. System Definition (RML) To Html XSL Stylesheet - Template for Statement

6. Viewing and Navigating System Work Products

As mentioned before, we developed an XML derivative called ELotosML, which is used to represent E-LOTOS formal specifications [21]. Using JavaCC [22], we developed a parser for E-LOTOS that transforms an E-LOTOS specification into its ELotosML representation. We are also developing an XSLT stylesheet which transforms an ELotosML document into HTML. The XSL stylesheet does not process every defined ELotosML element yet, but it does enough to be able to show a Steam Boiler formal specification section adapted from [40]. We describe in this section and the following how we use a combination of ELotosML, ELotosMLToHtml, and TraceML to navigate from a formal specification to its corresponding informal requirements document.

```
<xsl:template match="statement"
    name="statement">
    <xsl:param name="indent-number"/>
    <xsl:element name="span">
        <xsl:attribute name="id">
            <xsl:value-of select="@name"/>
        </xsl:attribute>
        <xsl:element name="div">...
            <xsl:value-of select="$indent-n..."/>
            <xsl:text> </xsl:text>
            <xsl:choose>...
<xsl:when test=" @verification-state ='Pending'
    and @criticality-level = 'Inherited' ">
    <span class="inherited">
        <xsl:value-of select=
        "concat( @title , ' (Inherited-Pending)' )"/>
    </span>
</xsl:when>...
        </xsl:choose>
        </xsl:element>
        <xsl:element name="div">...
            <xsl:value-of select="text()"/>
            <xsl:for-each select="statement">
                <xsl:call-template
                    name="statement">
                    <xsl:with-param
                    name="indent-number"
                    select="concat( $indent-
                number , string( '.' ) , string(
                position() ) )"/>
                </xsl:call-template>
            </xsl:for-each>
        </xsl:element>
    </xsl:element>
</xsl:template>
```

Figure 7. Informal Requirements (RML) To Html XSL Stylesheet Excerpt - Template for Statement

Figure 8 shows a view of Microsoft® Internet Explorer rendering the HTML result of the ELotosMLToHtml XSL stylesheet. In the picture, we can see a box titled *TransmissionMedium* containing a link named *implements: Transmission Medium Constraints.* The box appears whenever we locate the mouse pointer over the link titled *TransmissionMedium*, in the formal specification. After clicking on the link inside the box another document opens showing the Control System Informal Requirements document at the point where the requirement *Transmission Medium* was defined; Figure 9, right-hand pane. The pop-up boxes are created by JavaScript code automatically generated by the mentioned stylesheet upon parsing the ELotosML files along with a TraceML linkbase stating how sections of the CSIRD and the CSFS are to be linked. The Javascript uses the AJPopUp [30] library. Figure 10 shows an excerpt of the linkbase. These links, defined using TraceML and being rendered by this application, are Xlink extended links [15]. They can have several entry and salient points. Our rendering application is currently one of a few Xlink extended link implementations.

The left pane on Figure 9 shows a tree structure of the informal requirements, which is also automatically populated by another XSL stylesheet. This tree is rendered by the Tigra Tree Menu library from SoftComplex [37]. By clicking on any of the requirements in the tree the document on the right pane will show the corresponding description. These links are automatically generated and do not have to be explicitly stated in a linkbase.

7. Propagating Criticality and Identifying Sources of Faults

Navigating system work products is not the only use we are giving to our XML-based framework and linking endeavors. Our goal is to be able to help identify sources of faults and consequently minimize the occurrence of errors in critical systems. By propagating criticality as defined in the requirements to the system architecture and subsequent system work products, we can discover mismatches between different system models.

7.1 Pieces of Information

In order to propagate requirements criticality, we need two pieces of information: the criticality of the requirements and how those requirements relate to subsequent system work products such as system architecture and formal specifications. Information about requirements criticality as well as propagation paths is stated in the requirement documents using the *criticality-level* and *belongs-to* attributes. Currently, this information must be manually inserted into our CSDD and CSIRD.

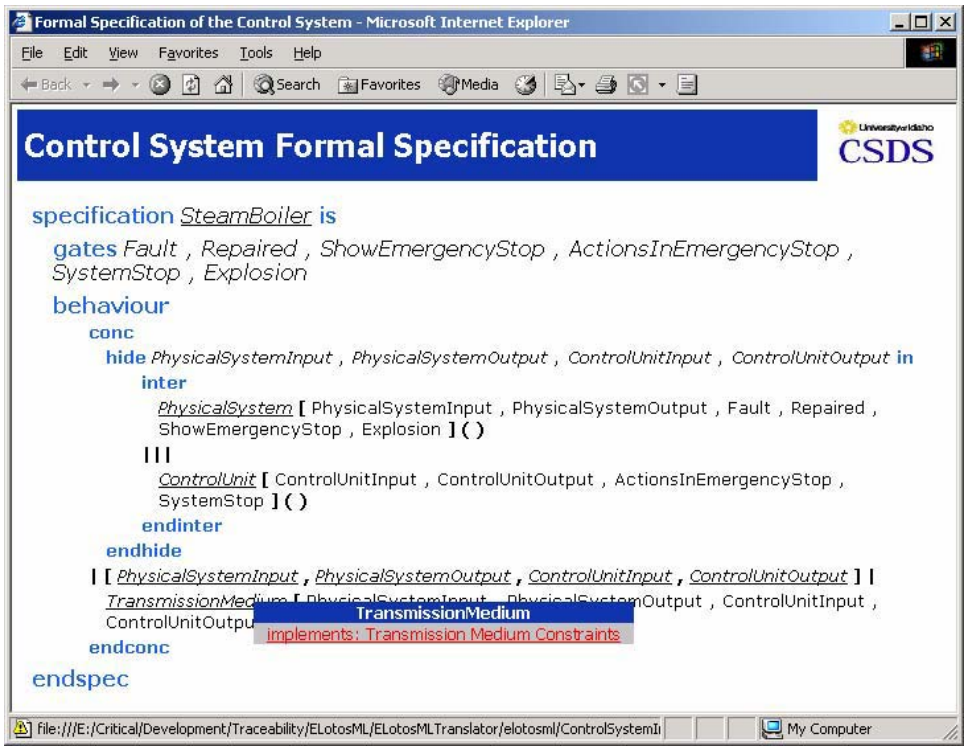

Figure 8. View of an E-LOTOS Formal Specification of the Steam Boiler with a Link to its Corresponding Informal Requirements Document

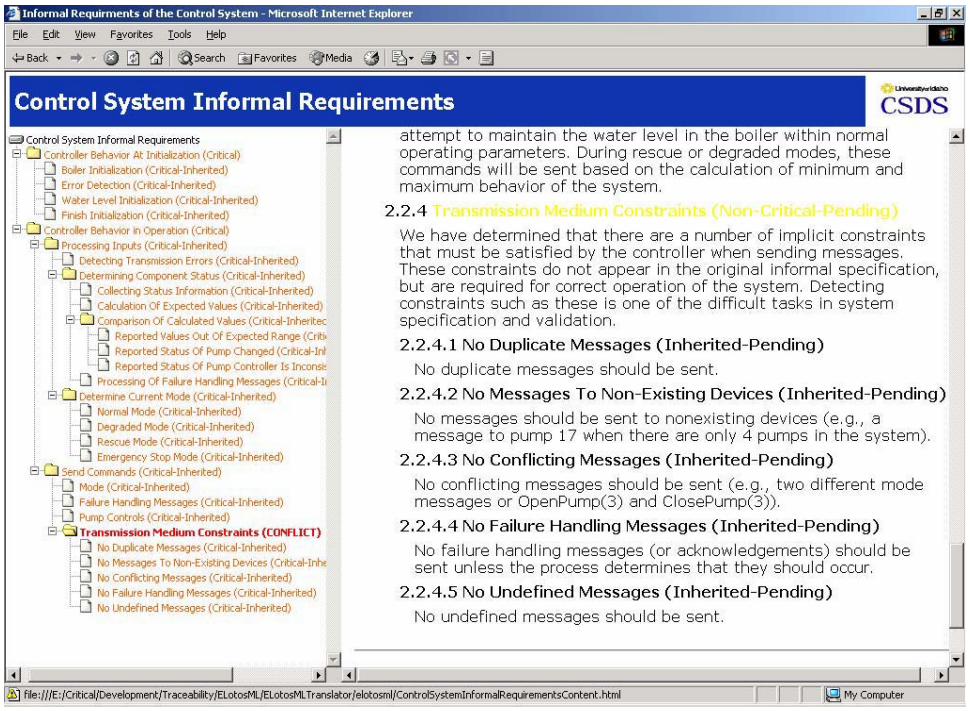

Figure 9. View of the Steam Boiler Informal Requirements Document

```
<resource xlink:type = "resource"
    xlink:label = "TransmissionMedium"
    xlink:title  = "Control System Formal Specif..."
    xlink:href ="ControlSystemFormalSpecification"
    traceml:level ="identifier"
    traceml:expr ="TransmissionMedium">
</resource>...
<link
    xlink:type = "arc"
    xlink:from = "TransmissionMedium"
    xlink:to  = "TransmissionMediumConstraints"
    xlink:title ="Transmission Medium Constraints"
    traceml:relation = "implements">
</link>
```

Figure 10. TraceML Linkbase Excerpt

The RML language described in Section 5 explains how this information is kept by the markup. The second set of information is the relationships among system work products, which are explicitly stated in a linkbase using our TraceML language. Figure 10 shows an excerpt of such a linkbase, for the Steam Boiler case study stating that the identifier *TransmissionMedium* of the CSFS implements the section *TransmissionMediumConstraints* of the CSIRD. A detailed explanation of TraceML can be found in [14]. This link is the source of the generated popup box appearing in Figure 8.

Now that we have all necessary pieces of information, we will explain how criticality propagation can be achieved. Propagation of criticality encompasses two purposes: to identify critical system components and to help discover model conflicts.

7.2 Critical Component Identification

As stated in Section 5 and revealed by Figures 3 and 5, statements and objects hold a criticality-level attribute. Possible values of this attribute are *True*, *False*, and *Inherited*. The criticality propagation semantics for an object or a statement are as follows: if the criticality-level of an element is true then the element is critical, if the criticality level is false then the element is not critical, if the criticality level is inherited then the criticality of that element is the criticality of its parent element. Figure 11 states this criticality propagation rule.

```
[[ criticality ( E ) ]] =
    ( equal ( criticality-level (E),'Inherited' ),
    [[ criticality ( parent(E) ) ]] ),
    (equal ( criticality-level (E),'True' ),Critical),
    (equal ( criticality-level (E),'False' ),Not-Critical ).
    (Where E is a statement S or an object O)
```

Figure 11. Criticality Propagation Semantics

This rule allows us to identify which system work sections are critical depending upon the criticality of the requirements which those sections are implementing. We implemented this semantic rule into the XSL stylesheet, which creates the tree presented in our Control System Informal Requirements (Figure 9). In that tree, we can observe two root requirements called *Controller Behavior At Initialization* and *Controller Behavior In Operation*. Those two requirements are considered critical, and their titles are followed by a *(Critical)* label. Other requirements on the tree have been marked as *(Critical-Inherited)* indicating that their criticality has been inherited from their parent requirements. Note that the root statement criticalities are *True,* since we are dealing with critical systems. The same rule has been implemented for propagation of criticality in the system architecture, which is not exposed in this paper due to space constraints.

7.3 Discovering Model Conflicts

Now that we have established attributes and links we are in a position to discover model conflicts that could lead to design faults and therefore to errors in the system. Table 1 defines the basic rule that we apply to discover conflicts among system work products applied to an element, which can be a statement or object. This rule can be applied, for example, to a statement and its parent, to an object and its parent statement, or to the same object as stated in different system work products. The first two cases are Intra-Model Conflicts and the third one is an Inter-Model Conflicts.

Table 1. Inter-Element Conflict Discovery Rule

	Element Criticality	
Element Criticality	Critical	Not-Critical
Critical	**OK**	**OK**
Not-Critical	**CONFLICT**	**OK**

7.3.1. Intra-Model Conflicts. Intra-Model conflicts arise when a statement or object is explicitly marked as non-critical and its propagated criticality tells us that it should be critical. This conflict detection rule has been implemented into our XSLT processing. Figure 9 shows the requirement *2.2.4 Transmission Medium Constraints* in red color and labeled as *(CONFLICT)* to indicate the presence of such a conflict among the propagated and assigned criticalities.

7.3.2. Inter-Model Conflicts. Inter-Model conflicts arise when an object or statement's criticality are in conflict with criticalities assigned or propagated from a different model. We are currently working towards

implementing inter-model criticality propagation semantics into our XSLT processing. As an example, assume a component such as the Message Transmission System is a child, within the CSIRD, of a critical requirement. By propagating the criticality of the statement to all its child objects, those objects will be declared as critical, including the mentioned component. Now, if we go to our system architecture, we see that the Message Transmission System component has been marked as non-critical because the message transmission system is assumed an external hardware component. Hence, a conflict arises between the criticalities of the Message Transmission System component in the system architecture model and the informal requirements. To solve this conflict, we need to either change the architecture of the system or prove that the component will operate correctly to adequate levels of reliability [11].

8. Related Work

Given the enormous number of specification and modeling languages in existence, developing XML derivates to represent system work products is a long-haul task. Fortunately, we are not alone in this task. The following is a list of some of those XML-based languages: Extensible Metadata Interchange (XMI) from the OMG [33] can be used to represent object-oriented detailed-design specifications and Unified Modeling Language (UML) models [32]; JavaML versions from G. Badros [6] and E. Mamas [29] can be used to represent Java source code; An XML-based representation of the formal specification language Z from Bompani, et. al., [8]. With the exception of XMI, all these languages are still in early research stages.

In addition, some experimental tools have been developed. IBM Alphaworks developed the Reengineering Toolkit for Java using JavaML; a tool used to calculate code metrics and help with source code refactoring [19]. REM, a tool to create edit and view software requirements [16]. Columbus, a tool to parse C++ source code and create its CppML representation. Finally, the Xlinkit tool from Finkelsetein and colleagues [18, 28]. The REM tool allows traceability among requirements using a traceability matrix approach based on identifiers inserted in the markup. Our approach also uses identifiers, but as a means of establishing complex relationships among sections of system work products at any desired level of granularity not just requirements. Xlinkit uses a rule-based approach to automatically create links among XML document sections. This tool could be used in our framework to automatically create links based on process or organizational development rules. Other links, though, might need to be manually added by engineers since not all relationships can be automatically derived. Our framework is open to both approaches.

9. Conclusions

The development of critical systems requires a solid software engineering framework. Unfortunately, complex systems necessarily require large development teams with diverse work-styles and backgrounds. There is a need for strong traceability between the work products generated by these teams. In our research, we are exploring technology for this strong traceability. In this paper, we have reported a portion of this work. With this technique, we can identify system components implementing critical requirements and assure that they will receive appropriate levels of verification.

We think there are two main lines where future work is needed: 1) furthering of our techniques of system work product sections linking, visualization and navigation, and criticality propagation, and 2) integrating these techniques into a complete system development environment for critical systems.

System work products linking achieved in this work allows us to navigate from informal requirements specifications to their corresponding formal system specifications. In [3], we explained how we linked object-oriented software specifications with their implementing source code. We would like to be able to close the circle and link and navigate the whole spectrum of system work products. In this work we used binary criticality and single criticality inheritance, we need to analyze the possibility of using other criticality domains such as enumerates or numbers, and multiple criticality inheritance. We also would like to discover more ways of finding model inconsistencies and conflicts using our framework.

Further work is necessary in order to determine how to embed these techniques into a fully integrated system development environment for critical systems. Note, though, that the flexibility of our approach is an advantage on this line. In our experiment, we manually entered the markup into the requirement documents. We need to analyze possibilities on how to make the task of adding the markup more intuitive to the developer, such as highlighting of statements and objects in a document and posterior selection of attributes from a selection list. Further work is needed to test coloring schemes that facilitate model understanding and error discovery.

10. Acknowledgments

We would like to thank the National Institute of Standards and Technology for funding this work under grant #60NANB1D0116 and the U.S. Army Research Laboratory for funding prior efforts [3, 14] that laid the foundations for this work. We would also like to thank SoftComplex and NavSurf for making their Java Script applications available for academic trial.

11. References

[1] J. R. Abrial, “Steam-Boiler Control Specification Problem,” Dagsthul Meeting, June 4 to 9, 1995, 10 August 1994.

[2] S. Adler, A. Berglund, J. Caruso, S. Deach, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, S. Zilles: Editors, “Extensible Stylesheet Language (XSL) Version 1.0,” W3C Recommendation. 15 October 2001.

[3] J. Alves-Foss, D. Conte de Leon, and P. Oman, “Experiments in the Use of XML to Enhance Traceability Between Object-Oriented Design Specs. and Source Code,” Proc. of the 35th Hawaii Intl. Conf. on System Sciences, 7-10 Jan. 2002.

[4] J. Alves-Foss, “A Constraint-Oriented Specification of a Steam-Boiler Controller in Higher Order Logic,” Technical Report, Laboratory of Applied Logic, University of Idaho, 1995.

[5] A. Avisienis, J. C. Laprie, and B. Randell, “Fundamental Concepts of Computer Systems Dependability,” Proc. of the Workshop on Robot Dep. , Seoul, Korea, 21-22 May 2001.

[6] G. Badros, “JavaML: A Markup Language for Java Source Code,” University of Washington, Seattle, WA, 2000.

[7] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Soemon, “XML Path Language (XPath) Version 2.0,” W3C Working Draft, 30 April 2002.

[8] L. Bompani, P. Ciancarini, F. Vitali, “XML-Based Hypertext Functionalities for Software Engineering,” Annals of Software Engineering, June 2002, Kluwer Academic Publ. Pp 231-247.

[9] J. Bowen and V. Stavridou, “Safety-Critical Systems, Formal Methods and Standards,” Technical Report PRG-TR-5-92, Oxford University Computing Laboratory, Oxford, May 1992.

[10] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, “Extensible Markup Language (XML) Version 1.0 Second Edition”, W3C Recommendation, 06 October 2000.

[11] R. Butler and G. Finelli, “The infeasibility of quantifying the Reliability of Life-Critical Real-Time Software”, IEEE Trans. on Soft. Engineering, Vol. 19, Nr. 1, Pp 3-12, Jan. 1993.

[12] D. Carlisle, P. Ion, R. Miner, and N. Poppelier, “Mathematical Markup Language (MathML) Version 2.0,” W3C Recommendation, 21 February 2001.

[13] J. Clark: Editor, “XSL Transformations Language (XSLT) Version 1.0,” W3C Recommendation, 16 November 1999.

[14] D. Conte de Leon, “Formalizing Traceability among Software Work Products,” Thesis, Univ. of Idaho, 2002.

[15] S. DeRose, E. Maler, D. Orchard: Editors, “XML Linking Language (XLink) Version 1.0,” W3C Rec., 27 June 2001.

[16] A. Durán, A. Ruiz, R. Corchuelo, and M. Toro, “Applying XML technologies in Requirements Verification,” Depto. de Leng. y Sist. Informáticos, Univ. de Sevilla, España, Unpub.

[17] J. Ferraiolo: Editor, “Scalable Vector Graphics (SVG) 1.0 Specification,” W3C Recommendation, 04 September 2001.

[18] A. Finkelstein, C. Nentwich, W. Emmerich, “Static Consistency Checking for Distributed Specifications,” Proc. of the 16th International Conference on Automated Software Engineering (ASE), San Diego, California, November 2001.

[19] IBM Alphaworks, “The Reengineering Toolkit for Java,” Web Site: http://www.alphaworks.ibm.com/tech/ret4j, 2002.

[20] U. Isaksen, J.P. Bowen, and N. Nissanke, “System and Software Safety in Critical Systems,” The University of Reading, Whiteknights, United Kingdom, December 1996.

[21] International Organization for Standardization, “ISO 15437: Information Technology – E-LOTOS,” Intl. Standard, 2001.

[22] WebGain, “JavaCC: Java Parser Generator,” http://www.webgain.com/products/java_cc, 2003.

[23] P. Ladkin, “Abstracts of References and Incidents,” Faculty of Technology, University of Bielefeld, http://www.rvs.uni-ielefeld.de/publications/Incidents/DOCS/FBW.html, 1998.

[24] J.C. Laprie, “Dependable Computing and Fault Tolerance: Concepts and Terminology,” Proc. of the 15th International Symposium on Fault Tolerant Computing (FTCS-15), 1985.

[25] J.C. Laprie: Editor, “Dependability: Basic Concepts and Terminology,” Springer-Verlag, Berlin, 1992.

[26] N.G. Leveson, “Safeware: System Safety and Computers,” Addison-Wesley 1995.

[27] N.G. Leveson, “Intent Specifications: An Approach to Building Human-Centered Specifications,” IEEE Transactions on Software Engineering, Vol. 26, Nr. 1, January 2000.

[28] R.R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical Embedded Systems," Proceedings of the IEEE International Symposium on Requirements Engineering, San Diego, CA, Pages 126-133, January 1993.

[29] E. Mamas and K. Kontogiannis, “Towards Portable Source Code Representations Using XML,” Proc. of the Seventh Working Conference on Reverse Engineering, 2000, Pages 1-12.

[30] NavSurf, “AJPopUp: A DHTML Javascript application for popup display,” http://navsurf.com/dhtml/ajpopup.asp, 2003.

[31] C. Nentwich, L. Capra, W. Emmerich and A. Finkelstein “Xlinkit: A Consistency Checking and Smart Link Generation Service,” ACM Transactions on Internet Technology 2001.

[32] The Object Management Group, “Unified Modeling Language Specification (UML),” Version 1.4, 07 Sep. 2001.

[33] The Object Management Group, “XML Metadata Interchange (XMI) Version 1.2,” Specification, 01 Jan. 2002.

[34] J. Peleska, “Formal Methods and the Development of Dependable Systems,” Thesis, University of Kiel, 1995.

[35] D. Raggett, A. Le Hors, and I. Jacobs: Editors, “HTML 4.01 Specification,” W3C Recommendation, 24 December 1999.

[36] J. Rushby, “Critical System Properties: Survey and Taxonomy,” Tech. Report CSL-93-01, SRI Intlernational, Menlo Park, CA, May 1993, Revised February 1994.

[37] SoftComplex, “Tigra Tree Menu: A DHTML Javascript,” http://navsurf.com/dhtml/ajpopup.asp, 2003.

[38] M. Weber and J. Weisbrod, “Requirements Engineering in Automotive Development: Experiences and Challenges,” IEEE Software, January-February 2003, Pages 16-24.

[39] L.R.. Wiener, “Digital Woes: Why We Should Not Depend Upon Software,” Addison-Wesley Co., Reading, MA, 1993.

[40] A. Willig and I. Schieferdecker, “Specifying and Verifying the Steam Boiler Control System with Time Extended LOTOS,” in J.R. Abrial: Ed., LNCS 1165, Springer-Verlag, October 1996.

[41] H. Wium Lie and B. Bos, “Cascading Style Sheets, Level 1,” W3C Rec., 17 December 1996, Revised 11 January 1999.