An XSLT-Based Proposal to Ease Embedded Critical Systems Tools Implementation, Verification, Validation, Testing, and Certification Efforts

Santiago Germino, Martín N. Menéndez, and Ariel Lutenberg

***Abstract*—Implementing embedded critical system tools using general-purpose programming languages usually challenges conformance to relevant standards. This letter proposes implementing tools that manipulate a system model expressed in XML using the Extensive Stylesheet Language Transformations (XSLT) and related technologies playing specific roles in a particular sequence developed to ease verification, validation, testing, and certification efforts. XSLT is a data-centric, domain-specific language that produces a more straightforward tool implementation, thus reducing human errors. We will introduce the proposal by reviewing, as an example, the reimplementation of a tool initially written in Python that automatically generates the VHDL source code of an embedded critical system. Finally, we will compare both implementations, noting compelling results.**

***Index Terms*—Automatic code generation, certification, embedded critical systems, verification and validation, XML, XSLT.**

I. Introduction

SEVERAL standards and regulations regarding the production, commercialisation and usage of embedded systems demand a certain level of guarantee that the system will perform as expected [1]. Regulations mainly focus on functional safety: the product is reasonably harmless even on failure. Safety is strongly related to quality management through product development and manufacture. Quality is an acceptable deviation to the product specification that varies by its field of application [1]. Quality management is crucial to assure quality—and therefore safety—yet it requires detailed documentation of each step involved in the embedded system's life cycle.

The processes of verification, validation and testing (V/V/T) employ the documentation to answer the questions of "are we building it right?", "do we have what we need?" and "does it behave as expected?" and they are essential to assuring quality [2]. To commercialise the product, an independent certification institution tests the system to pass the corresponding set of minimum requirements given by relevant standards and reviews how the engineers implemented the solution by thoroughly analysing the documentation [3].

Manuscript received April 19, 2021; revised August 16, 2021. This work was supported by UBACyT under grant 20020190100321BA. S. Germino and M. N. Menéndez have doctoral scholarships from CONICET. *(Corresponding author: Santiago Germino.)*

The authors are with the Laboratorio de Sistemas Embebidos, CONICET-GICSAFe, Facultad de Ingenería, Universidad de Buenos Aires, Paseo Colón 850, Buenos Aires, Argentina (e-mail: sgermino@fi.uba.ar; mmenendez@fi.uba.ar; alutenb@fi.uba.ar).

A critical system is one where a failure can cause significant damage such as death, economic loss or environmental destruction, for example avionics or process control on a nuclear plant. An electronic critical system is inherently an embedded system. The standards that govern them require further, more robust evidence than discussed that the system will perform as expected even on failure by designing it fail-safe [4]. Reaching the stringent level of safety that the field demands requires employing the best of diverse strategies.

Formal methods allow defining the whole or part of a complex system in an unequivocal specification language, and a mathematical model that enables running inferences on system properties [5]. Although a remarkable tool, no formal method offers a single solution to every problem, and all have their drawbacks, for example, as discussed in [6] for railway systems.

Automatically implementing a critical system from its specification might eliminate human error in the implementation and is an active research area [7]. Yet, as with the critical system itself, strict safety requirements govern an automatic tool that builds the implementation.

In general, the tool gets written in general-purpose programming languages like "C" or "Python". These are complex, Turing-complete languages from which it is hard to infer input to output likenesses by analysing the code [8]. That is detrimental in assuring safety: the same solution might not be error-free on every one of its conceivable implementations and possible inputs. A way to overcome that issue is to use a formally-defined subset of a general-purpose language and its associated frameworks [9], [10], along with strict coding standards [11]. An alternative strategy is to use a simpler data-centric, domain-specific language with specific commands to perform direct operations on the data; this is the approach we will explore in this letter.

II. XSLT

The Extensive Markup Language (XML) is both a human and machine-readable format used to store and transmit structured data. It is the basic building block for mature, well-known formats such as XHTML, Scalable Vector Graphics (SVG) and emerging ones to interchange railway network designs, like railML [12]. In addition, there are closely related technologies developed to address specific needs when working with XML structured data. XML Schema Definition (XSD) formally describes the content, structure and data types allowed on a given XML dataset, and it is a means of validating the data. The Extensive Stylesheet Language Transformations (XSLT) uses a declarative, domain-specific language to transform XML structured data by defining data structure patterns and corresponding manipulations to translate it to other formats or generate new structured data. It supports arithmetic operations and "for-each", "if" and "choose" statements. XML Path Language (XPath) is a query language used to combine and select parts of the structured data. XSLT uses XPath expressions to match patterns and build a dataset to iterate with its "for-each" statement. Complete documentation of XML, XHTML, SVG, XSLT, XSD, and XPath are on the W3C website under "XML Technology" [13]. To validate an XML file through XSD or run XSLT transformations, one must use an XSLT processor. Such a facility is generally built-in in Internet browsers, exists as an external library on general-purpose languages or is available as a standalone command-line application. We will use the last option.

III. Related work

XSLT has already proved helpful in the proposed field. A code generation tool for an automotive embedded system uses XSLT to translate the XML of a state machine representation to source code whose target is an OSEK operating system [14]. The authors state that "XSLT provides the necessary functionalities to handle complex requirements regarding code generation" and "The use of schemas helps ensure that the input XML format always conforms to the requirements. Thus it can be guaranteed that the generated output meets the specification".

Another work, a requirements management tool, uses XSLT to generate requirement documentation automatically and verify quality properties and metrics, effectively "Using XSLT as a Requirements Verification Language" [15]. Moreover, the authors praise the simplicity and effectiveness of the technology: "Our approach does not use NLP but an open, simpler and lighter technology like XML/XSLT". It's worth noticing that [15] employs DTD (Document Type Definition), a predecessor of XSD, for the same purposes of document validation. These works employ XSLT to complement tools implemented in general-purpose languages. To the best of our knowledge, using XSLT to implement a tool entirely is a novel approach, as proposed in this letter.

Finally, there exists an experimental, static XSLT template validator that answers with mathematical certainty if an XSLT template can produce a valid XML given the restrictions imposed on the input and output XSD schema [16].

This letter goes a step further from the state-of-the-art in the following directions: 1) presents how and where to employ XML technologies in the context of embedded critical systems; 2) describes a proposed workflow to perform V/V/T with XML technology; and 3) examines the viability and advantages of implementing a tool programmed entirely in XSLT.

IV. Methodology

We propose entirely implementing critical embedded system tools in XSLT, using XML to store any system model and XSD to validate a precise system model structure and contents. Thanks to the domain-specific nature of XSLT and its specific way of manipulating the XML model, the resulting implementation is generally more compact, straightforward, and less prone to human errors. For the system specification, we propose to use XSD schemas to define valid data structures and contents and XPath as a formal specification language to perform deterministic relations of specific system model data by some precise attributes. XSLT can perform the following tasks on the model: 1) relational queries using XPath expressions; 2) iterate through it or the results of relational queries; 3) simple arithmetic operations; 4) store a modified model by applying custom design parameters; and 5) translate it into other programming languages [13].

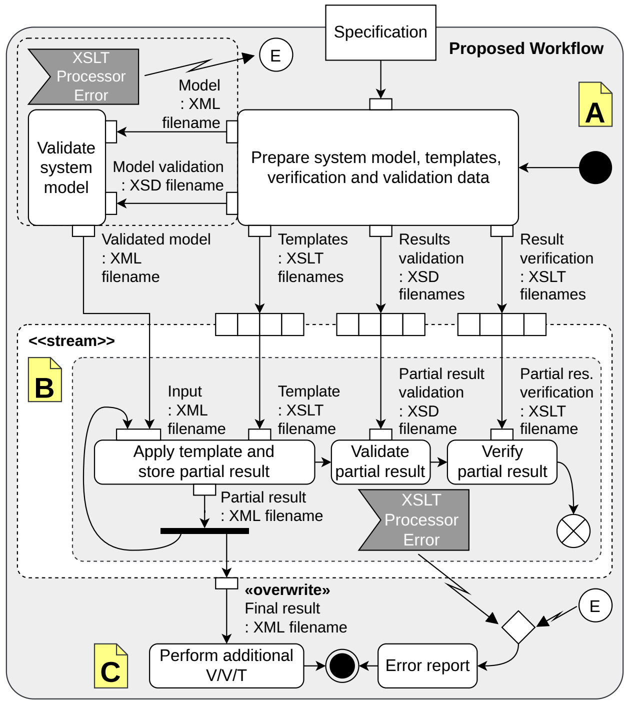
Specification
Proposed Workflow
XSLT Processor Error
E
Model : XML filename
Validate system model
Model validation : XSD filename
Prepare system model, templates, verification and validation data
A
Validated model : XML filename
Templates : XSLT filenames
Results validation : XSD filenames
Result verification : XSLT filenames
<<stream>>
B
Input : XML filename
Template : XSLT filename
Partial result validation : XSD filename
Partial res. verification : XSLT filename
Apply template and store partial result
Validate partial result
Verify partial result
Partial result : XML filename
XSLT Processor Error
«overwrite» Final result : XML filename
E
C
Perform additional V/V/T
Error report

Fig. 1. Proposed workflow UML activity diagram roughly divided into three parts (preparation, partial results, and final results) by notes A, B and C.

The XPath expressions and XSD files are also the actual method to implement the specification, helping avoid mistakes on the road between the two. Due to its inherent properties, the technologies above naturally act as a safeguard for the entire implementation, in contrast to the efforts required to achieve the same guarantees when using general-purpose languages.

The versions of each technology we found appropriate are XML 1.0, XSLT 1.0 and XPath 1.0. Their specifications are mature—have been stable for 20 years now—and there exist XSLT processors extensively tested over time [17] that successfully pass highly refined compliance test suites [18].

V. Workflow

We will explain the proposed workflow by walking through the UML diagram in Fig. 1. There are roughly three parts labelled "A", "B", and "C". The workflow starts in part "A" by preparing the system model, templates, verification and validation data.

A tool entirely implemented with XSLT splits the solution into incremental steps, iteratively calling the XSLT processor, as shown in part "B". The first iteration takes the validated system model in XML and an XSLT template as inputs and outputs a partial result in XML. After the first step, the XSLT processor takes the *n-1*th result and *n*th XSLT template to get an *n*th partial result. The last partial result is the final one. Additionally, the workflow performs V/V on each partial result by invoking the XSLT processor with the partial result and an XSD schema and again but with an XSLT template. That is why, going back to part "A", we prepare a corresponding list of XSLT template files for manipulating the system model and each partial result, the XSD files to validate them, and additional XSLT template files for verifying them. Next—still in part "A"—we validate the system model in XML by its XSD schema by calling the XSLT processor. If successful, the tool can start iterating to build the final result presented at the bottom, in part "C". That results in a new system model, possibly expressed in another language, that can be further verified, validated or tested independently.

The XSLT processor returns an error on any data manipulation, verification or validation that fails, thus immediately detecting non-conformance. That generates an error report—XSLT has extensive facilities to perform custom error reporting—and cancels the operation. The steps of repeatedly calling the XSLT processor with correct parameters and filenames can be run by hand or, as we choose to, by using GNU Make makefiles [19]. We can transparently extend the workflow to use many XSLT processors from different sources and vendors as an option. At each step, we can compare the partial results of other XSLT processors to match as a means of further validation.

The following section will illustrate how we used these technologies to simplify an embedded critical system tool implementation.

VI. Case study: Railway interlocking VHDL code generator

A railway interlocking system aims to avoid train collisions and derailments by operating associated signalling: traffic lights, barriers at level crossings, and switches. It is a critical system, so its design must follow appropriate practices [20].

We applied the discussed proposal to an automatic railway interlocking VHDL code generator tool [21] that our research team is developing at the request of Trenes Argentinos, the leading railway operator in Argentina. The tool has two well-defined parts. A Railway Network Analyzer (or RNA for short) reads railway network information stored in railML, analyses it, automatically generates railway signalling taking into account safety regulations, and finally exports the results back to railML. A second part, called Automatic Code Generator (ACG), takes the railway network nodes with modified signalling and produces an interlocking system FPGA implementation in VHDL. We initially prototyped the tool mentioned above using Python [22]. The standard that governs the software of railway interlockings is EN-50128:2012. In sections 3.1.44, 6.7.1.1, and 6.7.4, the standard categorises the

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <railML xmlns:dc="http://purl.org/dc/elements/1.1/">
3   <infrastructure id="is_01">
4    <topology>
5     <netElements>
6      <netElement id="ne16">
7       <associatedPositioningSystem id="ne16_aps01">
8        <intrinsicCoordinate id="ne16_aps01_ic01" intrinsicCoord="0">
9         <geometricCoordinate positioningSystemRef="gps01" x="-1104.347" y="-450.000"/>
10       </intrinsicCoordinate>
11       <intrinsicCoordinate id="ne16_aps01_ic02" intrinsicCoord="1">
12        <geometricCoordinate positioningSystemRef="gps01" x="1027.130" y="-450.000"/>
13       </intrinsicCoordinate>
14      </associatedPositioningSystem>
15     </netElement>
16    </netElements>
17   </topology>
18   <functionalInfrastructure>
19    <levelCrossingsIS>
20     <levelCrossingIS id="lcr120" activation="none">
21      <name name="Lc01" language="en"/>
22      <spotLocation id="lcr120_sloc01" netElementRef="ne16"
   applicationDirection="both" intrinsicCoord="0.3873"/>
23      <designator register="_Example" entry="LEVEL CROSSING Lc01"/>
24      <protection barriers="none" lights="none" acoustic="none"
   hasActiveProtection="true"/>
25     </levelCrossingIS>
26    </levelCrossingsIS>
27   </functionalInfrastructure>
28  </infrastructure>
29 </railML>
```

Fig. 2. A railML file sample cut for brevity to show only some elements.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3   <!-- find_signal_positions -->
4   <xsl:output method="xml" omit-xml-declaration="no" />
5   <xsl:variable name="settingsSpStep" select="200"/>
6   <xsl:template match="/">
7    <xsl:element name="rna">
8     <xsl:element name="settings">
9      <xsl:element name="signalPlacementStep"><xsl:value-of select="$settingsSpStep"/
   ></xsl:element>
10    </xsl:element>
11    <xsl:element name="signalPlacements">
12     <!-- levelCrossings position indexed by netElement.id -->
13     <xsl:for-each select="/railML/infrastructure/topology/netElements/-
   netElement[@id=/railML/infrastructure/functionalInfrastructure/levelCrossingsIS/-
   levelCrossingIS/spotLocation[@netElementRef=/railML/infrastructure/topology/-
   netElements/netElement/@id]/@netElementRef]/associatedPositioningSystem/-
   intrinsicCoordinate/geometricCoordinate">
14      <xsl:element name="signalPlacement">
15       <xsl:attribute name="id"><xsl:value-of select="generate-id()"/></xsl:attribute>
16       <xsl:attribute name="type">levelCrossingIS</xsl:attribute>
17       <xsl:attribute name="srcRef"><xsl:value-of select="../@id"/></xsl:attribute>
18       <xsl:attribute name="netElementRef"><xsl:value-of select="../../../@id"/></
   xsl:attribute>
19       <xsl:attribute name="actual"><xsl:value-of select="@x"/></xsl:attribute>
20       <xsl:attribute name="next"><xsl:value-of select="@x+(-$settingsSpStep)"/></
   xsl:attribute>
21       <xsl:attribute name="prev"><xsl:value-of select="@x+$settingsSpStep"/></
   xsl:attribute>
22      </xsl:element>
23     </xsl:for-each>
24    </xsl:element>
25   </xsl:element>
26  </xsl:template>
27 </xsl:stylesheet>
```

Fig. 3. XSLT template sample cut for brevity to show a matching template at line 6 and an XPath expression at line 13.

discussed tool with the highest requirements level, class T3: Tool output contributes to systems executable code, impacting its safety.

The railML format has railway infrastructure information scattered on different parts of its structured data [12]. For example, as shown in Fig. 2, lines 20 to 25 describe an element, while lines 7 to 14 locates it on the railway network. The RNA tool implemented in Python performs the following steps: create an in-memory Python object representation of the railML contents, build an in-memory dynamic lookup table to relate railway network nodes to the elements it contains, execute a set of manually implemented algorithms following their specification to determine railway signalling placement automatically, and finally put all parts together on a new, valid railML file.

When employing XSLT and related technologies, the RNA implementation itself does not have to load the data: it is already available to the matching XSLT template, as shown

TABLE I
COMPARING RNA IMPLEMENTATIONS: PYTHON AND XSLT.

Language	Type	LOC[*]	XML dataset load	Dataset manipulation	Partial results V/V	Executed by
Python	General-purpose	2000[†]	General-purpose instructions that invoke built-in modules. Data mapped to Python objects.	General-purpose instructions over Python objects.	No	Python Interpreter (Run-time < 1 s)[‡]
XSLT	Domain-specific	700	Automatic.	XPath, domain-specific instructions.	Yes (XSD, XSLT)	XSLT Processor (Run-time < 1 s)[‡]

[*] Lines of code written to implement the tool. Approximate. Fewer lines are desirable, as any of them may potentially contain human errors.
[†] Python syntax is already regarded as very compact and concise among other general-purpose programming languages.
[‡] The run-time of both implementations is negligible on a typical desktop PC (Intel i7 7700HQ, 16 GB RAM).

in Fig. 3, line 6. RNA does not need to create lookup tables by coding them with general-purpose instructions: as an example, the deterministic XPath expression at line 13 of Fig. 3 retrieves a set of elements with locations (what position, in which network node) for all level crossings in the railway network, ordered by network node id. There are no general-purpose instructions to iterate the set. Instead, the specialised "for-each" XSLT instruction iterates through the set built by the XPath expression. We generate a new XML file adding a safe distance around the track position (x-coordinate) of each element in the set, including railway platforms and rail joints. A second XSLT template receives the resulting XML, simplifies and consolidates all positions, and creates another XML file. A third XSLT template takes the file and performs signal placement on those calculated spots, creating another XML. A fourth XSLT appends the automatically generated signals, in railML format, to a copy of the original railML. A final XSLT takes the new railML, identifies railway network nodes as explained in [21] and exports them in a custom XML system model format. We perform XSD validation on output results and verify them with XSLT templates in each step. For example, we can infer if a partial result is correct by counting related railway nodes or calculating their distance to the signalling. Finally, we validate and test the automatically generated signalling inside the new railML file with RaIL-AiD, a railway layout editor with a certified railML exporter and extensive tools to analyse signalling [23].

XSLT was designed for pattern-matching and code transformations [13]. Hence, the ACG implementation in XSLT is remarkably concise. ACG runs as a sixth step where an XSLT takes the custom XML file produced in the final RNA step, transforms each railway network node to its pre-programmed VHDL code snippet and interconnects them according to the system model. Then, it creates a bitstream for the embedded critical system in FPGA with the use of a synthesiser [21]. In this letter, we will solely comment on RNA results.

VII. RESULTS

Table I summarises the main differences when comparing RNA implementations. Both output the same final results, but the implementation in XSLT is more straightforward, precise, and easier to review, removing potential points of failure common to programming in general-purpose languages. Using the workflow in Section V, the RNA implementation in XSLT executes incrementally, with each step producing XML with verified and validated partial results. This approach is consistent with essential critical systems certification efforts.

VIII. CONCLUSIONS

The XSLT-based proposal proved valuable in the presented case study to generate the VHDL code of an embedded critical system in FPGA in a way that eases certification efforts. We consider this approach may benefit embedded critical systems that store their system model in XML.

REFERENCES

[1] H. Pham, "System Software Reliability", Springer London, ISBN 978-1-85233-950-0, 2006.
[2] Engel, Avner, "Verification, validation, and testing of engineered systems", *Wiley*, ISBN 978-0-470-52751-1.
[3] TÜV Rheinland, *Product testing and test mark as proof of quality*. [Online]. Available: https://www.tuv.com/world/en/product-certification.html
[4] exida, "IEC 61508 Overview Report". [Online]. Available: https://www.win.tue.nl/~mvdbrand/courses/sse/1213/iec61508_overview.pdf
[5] G. O'Regan, "Concise Guide to Formal Methods", New York, NY, USA: Springer, doi:10.1007/978-3-319-64021-1.
[6] A. Ferrari, F. Mazzanti, D. Basile, and M. H. ter Beek, "Systematic Evaluation and Usability Analysis of Formal Tools for Railway System Design", doi:10.13140/RG.2.2.28040.96007.
[7] M. W. Whalen and M. P. E. Heimdahl, "An approach to automatic code generation for safety-critical systems", *14th IEEE Int. Conf. on Autom. Softw. Eng.*, pp. 315-318, doi:10.1109/ASE.1999.802346, 1999.
[8] J. E. Hopcroft, R. Motwani and J. D. Ullman, "Introduction to automata theory, languages, and computation". 3rd ed. Boston, MA, USA: Pearson, ISBN 0-321-45536-3.
[9] *About SPARK*. [Online]. Available: https://www.adacore.com/about-spark
[10] Verifiable C. [Online]. Available: https://vst.cs.princeton.edu
[11] MISRA C. [Online]. Available: https://www.misra.org.uk
[12] railML, *Open-source railway markup language*. [Online]. Available: https://www.railml.org/en/introduction.html
[13] W3C. [Online]. Available: https://www.w3.org/standards/xml
[14] F. Lindlar and A. Zimmermann, "A code generation tool for embedded automotive systems based on finite state machines", *6th IEEE International Conference on Industrial Informatics*, pp. 1539-1544, doi: 10.1109/INDIN.2008.4618349, 2008.
[15] A. Durán Toro, A. Ruiz-Cortés, B. Bernárdez and M. Toro, "Verifying software requirements with XSLT", *ACM SIGSOFT Software Engineering*, Notes. 27. 39-44, doi:10.1145/566493.1148028, 2002.
[16] A. Møller, M. Ø. Olesen, and M. I. Schwartzbach, "Static validation of XSL transformations", *ACM Trans. Program. Lang. Syst.*, 29, 4, Article 21, 47 pages, doi:10.1145/1255450.1255454, 2007.
[17] libxml2, *XML C parser and toolkit*. [Online]. Available: https://gitlab.gnome.org/GNOME/libxml2/-/wikis/home
[18] OASIS test suite. [Online]. Available: https://www.oasis-open.org/committees/xml-conformance/suite-v1/testsuite.htm
[19] GNU Make. [Online]. Available: https://www.gnu.org/software/make/
[20] M. Banci, A. Fantechi and S. Gnesi, "Some experiences on formal specification of railway interlocking systems using statecharts", *Train International Workshop at SEFM2005*, Koblenz, Germany, 2005.
[21] M. N. Menéndez, S. Germino, F. S. Larosa and A. Lutenberg, "Automatic generation of VHDL code for a railway interlocking system", *International Journal of Embedded Systems*, ISSN 1741-1068, Vol. 14, No. 6, pp.544–552, doi:10.1504/IJES.2021.121088, 2021.
[22] Python. [Online]. Available: https://www.python.org
[23] RaIL-AiD. [Online]. Available: https://www.rail-aid.com

All online references accessed Nov. 10, 2022.