PiXL: Applying XML Standards to Support the Integration of Analysis Tools for Protocols

María del Mar Gallardo[1], Jesús Martínez[1], Pedro Merino[1], Pablo Nuñez[2] and Ernesto Pimentel[1]

[1] Departamento de Lenguajes y Ciencias de la Computación
University of Málaga, Spain

[2] NibiSoft S.L., Málaga, Spain

Abstract. This paper presents our experiences on using XML [1] technologies and standards for the integration of analysis tools for protocols. The core proposal consists in the design of a new XML-based language named *PiXL* (Protocol Interchange using XML Languages), responsible for interchanging the whole specification of the protocol (data and control) among different existing tools. The structure and flexibility of XML has proven to be very useful when implementing new tools such as abstract model checkers. In addition, the suitability of the proposal has been applied to achieve a new kind of analysis, where *PiXL* and new MDA [2] methodologies have been proposed to build integrated environments for reliability and performance analysis of Active Network protocols. [3]

1 Introduction

Protocols are of critical importance for the reliability of Distributed Enterprise Information Systems. The application of formal methods during the Protocol Engineering process is well-known and has proven to be very useful because the elevated cost and consequences of failures among these concurrent and distributed systems are usually unacceptable. However, there is no common standard methodology to apply formal methods in the design and implementation of protocols, in contrast to those usually applied in other engineering fields.

One of the most important drawbacks when using formal methods for the analysis of protocols is the management of different system descriptions (models) for each tool employed: one specification for reliability analysis, another one for performance analysis, templates for code generation or documentation profiles, among others.

In this sense, the use of a single common formal description having all the features required by different analysis tools would maintain all the aspects to be analyzed in a consistent way. Unfortunately, this approach presents two main difficulties. First of all, it would be quite expensive to adapt the existing algorithms and tools to that new common notation and, on the other, non-expert users would have to learn how to manage it.

[3] Work partially supported by projects TIN2004-7943-C04 and TIN2005-09405-C0201

An alternative approach to avoid the construction of new algorithms or the introduction of new (and complex) language features consists of so-called *tool integration*, which has been developed mainly in two directions: one of them relies on the construction of integrated environments to manage a group of possible analysis, using internal translators between two tools (source and destination). The ETI [3] platform follows this scheme. In ETI, the coordination among tools is obtained through the definition of functionality taxonomies that each tool exports when it is integrated into the platform. Thus, the environment is able to recognize common or compatible functionalities.

The second direction consists of the definition of *intermediate languages*, which usually constitute new formal methods grouping common features used by the tools to be integrated. This kind of intermediate format allows the interchange of models among usual code generators and verification or static analysis tools.

Some of these languages are not user-oriented, and are directly oriented to parsers and algorithms of tools like CADP [4] or Bandera [5]. By contrast, IF [6] and Veritech [7] approaches have created their own formal description technique for users, which also acts as an intermediate language.

Traditionally, one of the main disadvantages of the second approach has been the difficulty of extending the intermediate formats, because the expressivity of these languages is clearly coupled with the features of the tools to be integrated originally. Therefore, it is not trivial to incorporate a new tool in order to take advantage of its complementary new features. This paper deals with this problem. We focus on the design of a new intermediate format for analysis tools that is flexible enough to allow its manipulation and extension, so that the integration of tools can be understood as a seamless process. That is, initially there will be no limits for features to be incorporated, allowing future extensions for new tools.

In order to achieve these objectives, we have used standard tools to define the new language. Our previous experiences in integrating tools and XML [8–10] have motivated the definition and implementation of an intermediate interchange language called *PiXL* (Protocol Interchange using XML Languages), which is designed to represent protocols considering the extended communicating finite-state machine (ECFSM) model. *PiXL* constitutes a new language based on a previous XML format used to perform abstract model checking with SPIN [11]. We have also incorporated some guidelines of recent Model Driven Engineering, such as MDA [2], to manage the integration of complementary analysis tools [10]. With *PiXL* we benefit from a common set of XML elements representing the usual ECFSM features, which can be easily extended and automatically parsed with existing XML technologies. These extensions allow the representation of formalisms such as PROMELA for SPIN, StateCharts or SDL.

The paper is organized as follows. Section 2 introduces the *PiXL* language, describing its main features and extension capabilities. The language constitutes the basis for the implementation and integration of the tools described in Section 3. Finally, Section 4 presents our conclusions and lines of future work on *PiXL*.

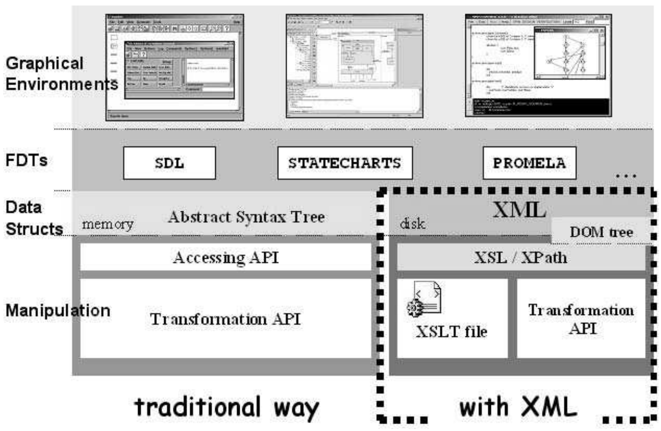

Fig. 1. Traditional APIs for extensions vs. an XML-based approach.

2 The *PiXL* Language

2.1 Justification

XML is been widely used as a language to interchange data. Nevertheless, its features have not been exploited within the domain of analysis tools. Transition-based formalisms like SDL, PROMELA, or variants of ECFSMs are commonly used to describe and analyze critical systems and communication protocols. These formalisms focus on describing the behavior of such systems, including concurrency, non-determinism, communication channels or dynamic creation of different entities, among others. Modelling languages share common characteristics; however, it is not easy to integrate tools that use such notations. This section describes the use of XML to represent all functional aspects of a complex system from the analysis point of view in order to facilitate its combined use by different (and complementary) analysis tools.

Fig. 1 presents the layered architecture of a typical analysis tool. Basically, the figure depicts the different stages through which data pass before being processed by algorithms. The upper layer consists of the graphical editor that helps users to define analysis models using some formal notation (for instance SDL or PROMELA). The intermediate layer (“data structs” in the figure) represents the most particular feature for each tool, because it defines the way in which model data are stored. Note that from this intermediate level to the bottom, fig. 1 has been divided into two columns. The left one shows the classical way of manipulating data using programming libraries (APIs). With this strategy, it is necessary to translate model data into an internal structure, like an Abstract Syntax Tree (AST). In the best case, the designer of the tool provides an open API with some limited extension capabilities. Using this API (and some technical assistance), it would be possible to access the AST to update the tool with new capabilities. However, even considering this method as successful, it is not clear whether extensions developed in this way could be partially reused with other existing or new emerging analysis tools.

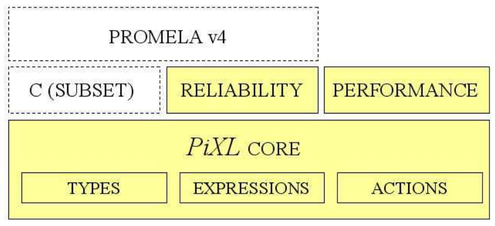

Fig. 2. Layered architecture of *PiXL* XML-Schemas.

Alternatively, we propose the use of XML to represent the original description of the model, which additionally allows us to use powerful associated tools and APIs, such as XPath and XSLT (to navigate and transform XML documents, respectively) or SAX and DOM APIs (for parsing). With this approach (shown in the right hand column of fig. 1), the AST is now represented as an XML document which can be managed by external developers in a more flexible way. The rest of the section summarizes the XML and *PiXL* features, along with extension capabilities of the language and associated tools for processing *PiXL*.

2.2 XML Technologies to Design *PiXL*

The first version of *PiXL* [12] was developed to describe PROMELA models in XML using a DTD (document type definition). Although DTDs are not as expressive as they should be to describe the complexity of a modelling language, we applied some strict semantic rules that were embedded into our XML translator to PROMELA. We also found some problems in dealing with extensions or modifications of previously declared XML elements. Alternatively, the XML Schema recommendation improves the way in which XML structures are created. It constitutes a full XML language, allowing direct XML tool support. The current version of *PiXL* is based on XML Schema layers, a feature which allows modularization and extensions, being more expressive to describe not only PROMELA models but also other transition-based formalisms.

Extension mechanisms provided with XML Schemas make it simple to redefine or add new contents to existing language elements. Fig. 2 shows the architecture of different Schemas that are part of our proposal. The *PiXL* core is composed of three basic XML Schemas (defining types, expressions and actions as XML elements). Creating new modelling components is done by reusing those provided in the core. The figure shows how upper Schemas rely on functionality previously defined in other Schemas. For instance, the C language subset Schema defines elements for C code, which reuse other elements existing in the *PiXL* core, by including its Schemas. We also provide Schemas describing common reliability or performance features useful for analysis tools. Dotted boxes constitute work in progress at the moment, oriented to building a new version of the abstract model checker αSPIN, introduced in Section 3.1, to be compatible with the new version of PROMELA [11].

Each of our *PiXL* Schemas benefits from using XML namespaces to differentiate elements; that is, in *PiXL* it is possible to define two XML elements with the same name,

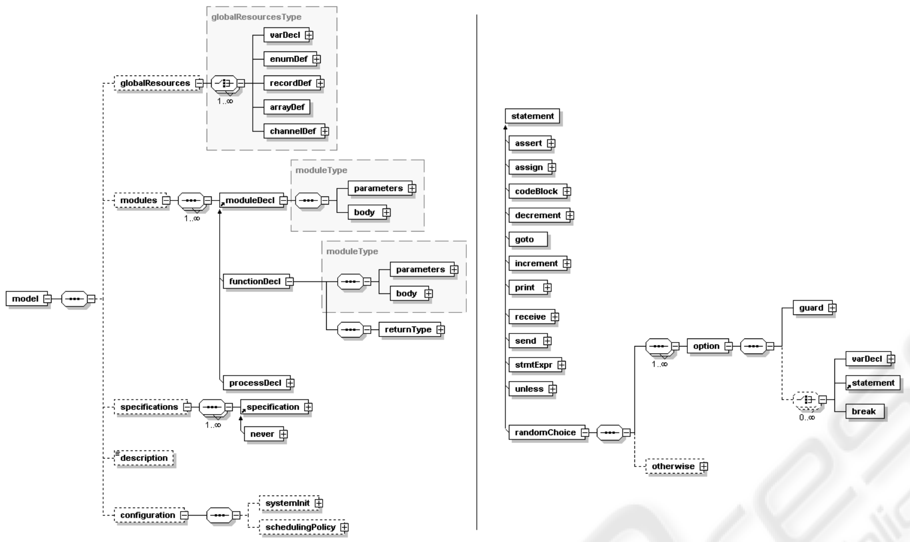

Fig. 3. Some elements of a *PiXL* model: structural elements (left) and actions (right).

if they are used within a different context (its namespace). This feature is specially interesting to build parsers that will manage only those XML elements that are interesting for a specific tool. The *PiXL* core also defines complex types which constitute the base of every XML element of the same type, as commented below.

2.3 Language Deatures

A *PiXL* model is composed of global resources and modules (fig. 3, left). Global resources describe variables, arrays, user defined types or communication channels available for all the modules in the protocol model. As shown in fig. 3, *PiXL* provides the `<moduleType>` type to define parameterized modules. Fig. 3 makes use of this complex type to create the abstract `<moduleDecl>` element. This is the base type to derive the concrete *PiXL* modules `<functionDecl>` (function declaration) and `<processDecl>` (process declaration), it being possible to include any of the concrete elements inside a `<modules>` element. Models may also contain requirements in the form of `<specifications>`. Currently, *PiXL* provides the `<never>` element to model a Büchi automaton. This kind of automaton is used by model checking algorithms [13] to describe and analyze functional properties of protocols.

PiXL language includes support for expressions, statements and data types. For example, binary and unary operations are available expressions, along with data channel queries, function calls, constants or references to variables. *PiXL* statements represent actions, as shown in fig. 3 (right). The base type `<statement>` is redefined to create typical actions to be included in the `<body>` part of the *PiXL* modules: assignments, assertions or flow control structures. Regarding flow control, *PiXL* provides the `<randomChoice>` element to define non-deterministic choices. The optional

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
<model>
...
    <configuration>
     <systemInit>
      <declaration name="myChannel">
       <channel size="1"><integer/><byte/></channel>
      </declaration>
      ...
      <codeBlock executionMode="strict-atomic">
       <stmtExpr>
        <run name="myProcess">
         <refAsExpr>
          <varReference name="myChannel"/>
         </refAsExpr>
        </run>
       </stmtExpr>
       ...
      </codeBlock>
      <send mode="fifo">
       <varReference name="myChannel"/>
       <args><const value="1"/><const value="0"/></args>
      </send>
     </systemInit>
    </configuration>
</model>
```

Fig. 4. Some excerpts of a *PiXL* model.

`<otherwise>` element shown as a dotted box in fig. 3 (right), allows executing this branch only when the other options cannot be selected (this is determined by checking the guard element). Finally, data types in *PiXL* follow a similar hierarchy of predefined types available in XML Schemas, allowing extensions. The *PiXL* core provides the primitive and complex types available in the PROMELA language.

A *PiXL* model optionally contains a system configuration to initialize global resources, to determine which processes are started at the beginning, or to define scheduling parameters such as priority semantics for process scheduling. Fig. 4 shows part of an example for a *PiXL* model that defines a system configuration: the declaration and use of the `myChannel` channel variable and the start of a process instance (`myProcess`) having `myChannel` as a parameter. For a complete reference of all the elements available in the *PiXL* language along with some examples, see [14].

2.4 Extension Capabilities

Regarding structuring capabilities of XML Schemas, they allow the introduction of upper layers to reuse previous *PiXL* core elements, creating new ones or redefining other existing ones. Therefore, the introduction of new elements is done without interfering with the way in which existing tools work, since XML technologies help to deal with unknown syntax elements for a given model. At the moment, the *PiXL* core elements allow the description of transition-based notations like PROMELA, or StateCharts diagrams, along with a subset of SDL. Some extensions to create suitable elements for usual imperative languages like C and Tcl are now in progress.

2.5 Processing *PiXL*

PiXL models benefit from using standard tools to manage their structure. Therefore, it is possible to parse such a structure in order to perform static analysis, using XML parsers and query languages like XSL. Another interesting feature of the XML parsing tools is the support to build code generators, which can be considered as one of the most important tasks in the tool integration approach.

Static analysis *PiXL* models may be analyzed with static analysis techniques, where algorithms can be coded using typical programming languages (Java, C++), XSLT templates or a combination of both of them. Abstract interpretation for model checking (described in the next section), constitutes a clear example of static analysis. Due to the complexity of the method, XML DOM is used to manipulate the representation of the *PiXL* model as a tree in Java.

Code generators Although the use of XSL to generate code from XML documents is frequent, there are cases in which the complexity of the associated XML Schema requires another kind of strategy. With *PiXL* we follow the Java architecture for XML binding (JAXB) [15], a specification that associates XML Schema components with their equivalents in form of Java objects. Therefore, using a JAXB tool we can automatically obtain Java source code (an API) corresponding to every element defined in *PiXL*. This API allows us to manage any *PiXL* model. Furthermore, it has been extended with our code in order to generate input formats for different analysis tools.

3 Applications of *PiXL*

This section summarizes our previous experiences using *PiXL* to integrate analysis tools. The first experience was the development of an abstract model checker named αSPIN, and more recently we have been working on an integrated environment to analyze active network protocols.

3.1 Abstraction and Model Checking

Model checking is effective when dealing with a useful version (a model) of the protocol to be verified. By *useful* we mean that the model should contain only those aspects necessary to analyze the critical properties of the protocol. *Abstract interpretation* [16] is one of the most successful techniques utilized to reduce the size of models [17, 18] and, therefore, to avoid the so-called state explosion problem.

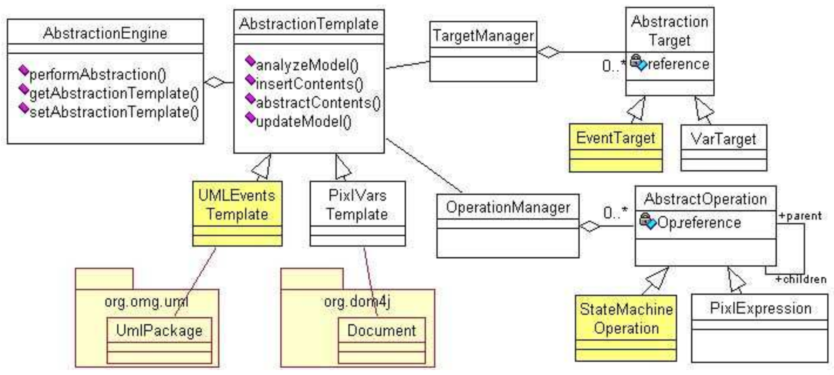

Fig. 5. A framework to perform abstraction in XML models.

For the effective application of this new technique, a complete support for the automatic abstraction of models and properties is needed. In [8] we presented the αSPIN tool, a distribution that integrates the functionality provided in the SPIN model checker with abstraction capabilities. αSPIN performs a source-to-source transformation of PROMELA models and LTL properties. This approach benefits from completely reusing the SPIN algorithms to verify the resulting abstract model.

In order to perform the syntactic transformation of the model in a flexible way, the abstraction module (a Java application) is independent of the SPIN tool [12]. Both tools interchange models using *PiXL* as the integration language which has full support to represent PROMELA. From the implementation point of view, αSPIN is the first abstraction tool based on XML standards able to perform automatic abstractions in PROMELA.

The abstraction GUI consecutively selects variables to be abstracted from a list supplied by the user. For each variable, the (concrete) operations in which it appears are analyzed and substituted by their corresponding abstract versions extracted from a predefined library. The abstraction engine makes use of a standard XML API to parse a *PiXL* model, transform it and generate its corresponding abstract version to be verified with SPIN. The abstraction engine is composed of specialized data structures (Java objects) to manage suitable references of *PiXL* variables and expressions. It has also containers to deal with an abstraction library; that is, a collection of abstract operations to replace the original ones after the abstraction process. The abstraction library is also stored in XML format, according to an extension of the *PiXL* grammar. All the syntactic transformations are made within the *PiXL* document tree in memory, using XPath queries.

In order to manage not only data abstractions but also event abstractions, the abstraction API of αSPIN was completely redefined in [9]. The new objective was to apply the same concepts to abstract UML StateCharts behavior diagrams [19], since currently commercial tools like STATEMATE are also offering model checking capabilities. The final version of the current abstraction API is partially depicted in fig. 5, where *PiXL* elements may be embedded into existing XML representations of a UML StateChart.

SCITEPRESS Science and Technology Publications

3.2 Reliability and Performance Analysis

In [10] we proposed the use of MDA and XML as a way to integrate existing tools for the analysis of new emerging telecommunication services. In the paper we presented a methodology to obtain suitable input for the tools to be integrated, avoiding the need for several hand-made specifications. We suggested the use of intermediate representation languages with XML support for the development of parsers and code generators. In order to test the viability of our approach, we chose the domain of telecommunication services, particularly the Active Network paradigm [20], which offers flexibility to develop new telecommunication services without the slow standardization process usually required by international institutions (ITU, IETF, IEEE or ANSI).

Following our proposal in [10] we are now using *PiXL* models to represent active code and considering MDA guidelines to build platform-independent models and platform-specific ones [2]. The former models are general *PiXL* models that represent the expected behavior of active code. The latter are modified *PiXL* models which incorporate those features needed by specific destination platforms (analysis tools). We are working on generating inputs for reliability and performance tools (as described in [10]).

4 Conclusions and Future Work

Intermediate languages are usually employed to integrate tools within the context of protocol analysis. However, such formats (formal methods) are prepared to deal exclusively with specific tools. Thus, it is difficult to integrate a new one without making significant modifications of the language and associated APIs.

This paper has introduced *PiXL*, a domain-specific language to connect analysis tools for protocols. *PiXL* is an XML language that exploits all the benefits of this mature technology: open tools and widely accepted standards. The main features of the language allow the extension of new expressions, statements (actions) and types, using the characteristics and possibilities available with the XML Schema. Up to now, *PiXL* has been applied to the development of abstract model checkers and integrated environments to perform different analysis of communication protocols. The novelty in using XML and MDA within the Protocol Engineering domain greatly facilitates the evolution of tools and the development of new extensions. It is also worth noting that the use of XML (and *PiXL*) technologies may support the introduction of formal techniques within the Software Engineering community.

Our future work is focused on introducing more flexible ways to define and embed properties in *PiXL* models. Currently, we have support to define Büchi automata but having other logical formalisms in XML would be of great interest. Moreover, the right combination of models, properties and analysis reports inside a single XML document will ensure consistency among tools and a way of keeping analysis as unified as possible.

References

1. W3Consortium: Extensible Markup Language (XML) 1.0 (Second Edition). Available at http://www.w3.org/XML/ (2000)
2. Object Management Group: MDA guide version 1.0.1. omg/2003-06-01 (2003)
3. ETI: The ETI Platform. Available at http://eti.cs.uni-dormund.de (2004)
4. Fernandez, J., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., Sighireanu, M.: CADP: A Protocol Validation and Verification Toolbox. In Springer, ed.: Proceedings of the 8th Conference on Computer-Aided Verification. Volume 1102. (1996) 437–440
5. Hatcliff, J., Dwyer, M., Pasareanu, C., Robby: Foundations of the bandera abstraction tools. In: The Essence of Compution. Number 2566, Springer Verlag (2003) 172–203
6. Bozga, M., Fernandez, J., Ghirvu, L., Graf, S., Krimm, J., Mounier, L.: IF: A Validation Environment for Timed Asynchronous Systems. In Springer-Verlag, ed.: Proceedings of CAV'00. Volume 1855 of Lecture Notes in Computer Science. (2000) 543–547
7. Katz, S.: Faithful Translations among Models and Specifications. In: Proc. of Formal Methods Europe. (2001)
8. Gallardo, M., Martinez, J., Merino, P., Pimentel, E.: A Tool for Abstraction in Model Checking. Software Tools for Technology Transfer **5** (2004) 165–184
9. Gallardo, M., Martinez, J., Merino, P., Pimentel, E.: Abstracting UML behavioral diagrams for verification. Chapter in In Hongji Yang. In Publishing, I.G., ed.: Software Evolution with UML and XML. (2004)
10. Gallardo, M.M., Martínez, J., Merino, P., Rodriguez, G.: Integration of Reliability and Performance Analyses for Active Network Services. Volume 133 of Electronic Notes in Theoretical Computer Science. (2005) 217–236
11. Holzmann, G.: The SPIN Model Checker. Primer and Reference Manual. Addison Wesley (2003)
12. Gallardo, M., Martinez, J., Merino, P., Rosales, E.: Using XML to implement Abstraction for Model Checking. In: Proc. of ACM Symposium on Applied Computing. (2002) 1021–1025
13. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
14. University of Málaga: The *PiXL* Project Web Page. Available at `http://www.lcc.uma.es/~gisum/fmse/pixl` (2006)
15. Sun-MicroSystems: Java Architecture for XML Binding (JAXB). Available at http://java.sun.com/webservices/jaxb/ (2006)
16. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM Symp. on Principles of Programming Languages. (1977) 238–252
17. Havelund, K., Visser, W.: Program model checking as a new trend. In: Software Tools for Technology Transfer (STTT). Volume 2. (2002) 8–20
18. Dams, D., ed.: Abstraction in Software Model Checking: Principles and Practice. Number 2318 in Lecture Notes in Computer Science, 9th Int. SPIN Workshop. Model Checking Software (2002)
19. Harel, D., Pnueli, A., Schmidt, J., Sherman, R.: On the formal semantics of statecharts. In: Proccedings of the 2nd IEEE Symposium on Logic in Computer Science, New York, IEEE Press (1987) 54–64
20. Calvert, K.L., Bhattacharjee, S., Zegura, E., Sterbenz, J.: Directions in Active Network Research. IEEE Communications Magazine **36** (1998) 72–78