PiXL: Applying XML standards to support the integration of analysis tools for protocols

María del Mar Gallardo, Jesús Martínez*, Pedro Merino, Pablo Nuñez, Ernesto Pimentel

Dpto. de Lenguajes y Ciencias de la Computación, University of Málaga, 29071 Málaga, Spain

Received 15 April 2006; received in revised form 1 July 2006; accepted 1 August 2006
Available online 18 October 2006

Abstract

This paper presents our experiences on using XML 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 methodologies have been proposed to build integrated environments for reliability and performance analysis of Active Network protocols.

© 2006 Elsevier B.V. All rights reserved.

Keywords: XML; Protocols; Analysis; Formal methods; Model checking; Performance; MDA

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.

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 hand, non-expert users would have to learn how to manage it.

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 analyses, using internal translators between two tools (source and destination). The ETI [1] and CADP [2] platforms follow this scheme.

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. Approaches such as Bandera [3], IF [4] or Veritech [5] follow this direction, allowing the interchange of models among the usual code generators, verifiers and static analysis tools.

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 new complementary 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 [6–9] have motivated the definition and implementation of an intermediate interchange language called PiXL (Protocol Interchange using XML Languages). PiXL is defined by a group of XML layered Schemas [10], each of which contains a set of the usual features of modelling languages. In particular, these Schemas adopt the features found in languages which extend CFSMs (communicating finite state machines) such as PROMELA for SPIN [11], StateCharts [12] or SDL [13]. We have also incorporated some guidelines of recent Model Driven Engineering, such as MDA [14], to manage the integration of complementary analysis tools [8]. With PiXL we benefit from a common set of XML elements which can also be easily extended and automatically parsed with existing XML technologies.

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. Section 4 summarizes existing related work. Finally, Section 5 presents our conclusions and lines of future work on PiXL.

2. The PiXL language

2.1. Motivation

XML is being widely used as a language to interchange data. Nevertheless, its features have not been exploited within the domain of analysis tools. Transition-based formalisms such as PROMELA or SDL (variants of CFSMs) 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.

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). The XML representation then constitutes an intermediate language which will represent data in a consistent way for different analysis tools. 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, also allowing the inclusion of different kinds of data needed by complementary analysis tools. 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 [15] was developed to describe PROMELA models in XML using a DTD (document type definition). PROMELA is a non-deterministic language which borrows some concepts and syntax elements from Dijkstra’s guarded command language, Hoare’s CSP language and the C programming language. A PROMELA model is composed of a finite set of processes that are executed concurrently. Processes may share global variables or channels, it being possible to represent in the language both shared-memory and distributed-memory systems. In addition, processes may have local variables storing their local state. The behavior of a process is defined by a sequence of possibly labelled sentences preceded by a declarative part. Basic sentences in PROMELA are those that modify the local state of processes (or the global state of the system) such as assignments or operations for sending/receiving messages through channels. Boolean expressions behave as guards that must be satisfied before continuing execution. Instructions If and Do in PROMELA include guards which are selected in a non-deterministic manner. There are also atomic statements which define a sequence of instructions whose execution cannot be interleaved with instructions from other processes.

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 [10] 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 and is more expressive to describe not only PROMELA models but also other transition-based formalisms.

Extension mechanisms provided by 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 the ones provided in the core. The figure shows how upper Schemas rely on functionality previously defined in other Schemas. For instance, the C 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,
Fig. 2. Layered architecture of PiXL XML-schemas.

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

oriented to building a new version of the abstract model checker αSPIN (see 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, as long as they are used in separate contexts (different namespaces). 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 features

A PiXL model is composed of global resources and modules (Fig. 3, left hand column). 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 specific PiXL modules <functionDecl> (function declaration) and <processDecl> (process declaration), it being possible to include any of the specific elements inside a <modules> element. We also consider that PiXL should allow the representation of the most popular constructions for properties. One of these constructions is the automaton to represent properties,
which takes the form of a Büchi automaton in tools like SPIN or the form of an observer in Telelogic Tau. For instance, the <buchi> specification encapsulates the description of a Büchi automaton as part of the core XML Schema.

Regarding constructs for protocol behavior, the 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 hand column). The base type <statement> is considered an extension tag which is redefined to create typical actions to be included in the <body> part of the PiXL modules. The figure shows assertions, assignments, print and goto tags, code blocks (sequences of statements), channel operations (send and receive), expressions, increments/decrements over variables, and a non-deterministic control flow construct. Regarding this kind of control flow, PiXL provides the <randomChoice> element to define non-deterministic choices. The optional <otherwise> element shown as a dotted box in Fig. 3 (right hand column), 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, also allowing extensions. The PiXL core includes all the primitive and complex types available in the PROMELA language, along with the possibility of representing any of its constructs in PiXL. For instance, in order to represent non-deterministic Do or If PROMELA constructs we use the PiXL <randomChoice> along with its optional boolean attribute loop.

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 [16].

2.4. Extension capabilities

The use of XML Schemas allows us to organize PiXL in different content layers (Schemas) which import and may then reuse previous PiXL core elements, create new ones or redefine 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 such as PROMELA or StateCharts diagrams, along with a subset of SDL. We are also extending PiXL in order to deal with C constructs. Although C is not considered one of the usual languages for protocol analysis tools, it is a complementary language now commonly used in modern model checkers such as Telelogic Tau for SDL and the current version of SPIN. These tools allow embedded C code within SDL or PROMELA designs, respectively, then reduce the gap between the usual development languages and formal description languages.
Fig. 5. A possible PiXL extension to describe state diagrams.

For instance, Fig. 5 shows part of a simple extension to define state machines, similar to the structure followed by StateCharts or SDL. The figure includes the element `<transition>` which represents the classical transition between two states. The contents of this element include: references to the states’ source and destination (declared elsewhere in the Schema), events which activate the transition, a guard which contains boolean expressions to be evaluated before the activation and, finally, the actions to be taken. It is worth noting that guards and references to events and actions are elements derived from existing PiXL expressions, types and statements, respectively.

The simple example of Fig. 5 demonstrates the expressivity of the PiXL language, allowing the reusability of element types provided in the PiXL core Schemas. Thus, actions are block types that inherit the possible sequence of previously declared PiXL statements. When references to events (`<eventRef>`) have been declared of type `<reference>`, they will be automatically available as actions, since they are expressions which act as statements (`<stmtExpr>` in Fig. 3, in the right hand column).

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 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.

2.5.1. 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. Fig. 6 shows a very simple template which detects those variables declared in a PiXL model but which are never used. Line 5 selects iteratively each declaration of a variable within the model. Given a single variable, line 7 checks for more appearances of such a variable. The predefined function `count(/varReference[@name=$varName])` returns the number of
<varReference> elements with their attributes name matching the value $varName$. If this number is equal to zero the variable appears only in its own declaration, and a corresponding warning message is thrown.

Abstract interpretation for model checking (described in Section 3.1) constitutes another example of static analysis. Due to the complexity of this method, XML DOM was preferred to manipulate the representation of the PiXL model as a tree, instead of using XSLT templates.

2.5.2. 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) [17], a specification that associates XML Schema components with their equivalents in the form of Java objects. Therefore, by using a JAXB tool we can automatically obtain Java library to manage any PiXL element. This library allows us to manage any PiXL model and we have extended it in order to generate input formats for different analysis tools.

Fig. 7 shows the class diagram of the PROMELA code generator from PiXL models. The class diagram in the figure shows the package pixl, which contains all the JAXB objects corresponding to every PiXL element. Following a variant of the Visitor pattern [18], we have also defined a generic interface called PiXLVisitor, with a method to visit every PiXL object available in the package. The JAXB library includes objects to parse an XML document, creating a tree of JAXB objects in memory. The specific generator to PROMELA is a class called PiXL2PROMELA, which implements the methods of the PiXLVisitor interface in order to obtain the resulting PROMELA model. It is worth noting that PiXL extensions are easily managed in JAXB, it being possible to reuse existing code through class inheritance. The figure shows an example of parent class Statement and its derived class Send. Thus, this hierarchy resembles the original design in the PiXL core XML Schemas (see Fig. 3).

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

One of the most employed techniques for reliability analysis is model checking [19]. Model checkers perform the analysis in an automatic way, being 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 [20] is one of the most successful techniques utilized to reduce the size of models [21,22] and, therefore, to avoid the so-called state explosion problem.

For the effective application of this new technique, a complete support for the automatic abstraction of models and properties is needed. In [6] we presented the αSPIN tool, a distribution that integrates the functionality provided by the SPIN model checker with data abstraction capabilities. With this approach the domain of a given set of variables is reduced to only a few values.

SPIN is one of the most important model checkers used in both academic and industrial fields to verify protocols. It obtained the ACM System Software Award in 2001, the same award given to well-known developments like TCP/IP,
World-Wide Web, UNIX or Java. PROMELA models for SPIN can be analyzed against general and critical reliability properties such as deadlock absence or liveness. In addition, other particular properties of the system to be analyzed can also be specified using Linear Temporal Logic formulas (LTL). SPIN also includes some powerful optimization techniques such as Partial Order Reduction, which eliminates unnecessary inter-leavings, or State Compression and Bit-state hashing to optimize the inner representation of the states in memory.

αSPIN implements data abstraction by a source-to-source transformation of PROMELA models and LTL properties. This approach benefits from completely reusing the SPIN algorithms (and optimizations) to verify the resulting abstract model. In order to perform the syntactic transformation of the model in a flexible way, the αSPIN abstraction module (a Java application) is independent of the SPIN tool [6]. 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.

In order to illustrate the way in which αSPIN works, we have described an Enterprise system in PROMELA, where different Enterprise JavaBeans (EJB) [23] request some operations with a database (getDBconnection() and commit_release() in Fig. 8, left hand box). For every thread in the system we have considered two Beans: the outer EJB uses a database connection, then makes a nested EJB call which uses a separate database connection (inner) from the same connection pool. Under normal load, the outer EJB call will get a database connection from the DBpoolManager process, and then call the nested EJB. The nested EJB will get another database connection from the pool, commit the inner transaction and return the connection to the pool. The outer EJB call will then commit its transaction, and return its connection to the pool. In this context, the size of the pool has to be large enough to avoid deadlock situations. A deadlock will occur if there are the same number of database connections as concurrent calls to the outer EJB. Each of the threads acquires a database connection, emptying the pool. Then, they each try to make the nested EJB call, which needs to acquire a second database connection. Unfortunately, none of them will proceed and no one is able to release their first connection. The verification of the PROMELA model in Fig. 8 easily demonstrated the relationship between number of threads and pool size. For instance, the absence of deadlocks with a database connection pool of size six and with five threads was found in SPIN after analyzing 590,846 states. The number of analyzed states increases if the pool size and number of threads increase too, eventually leading to a state explosion situation. Therefore, we wanted to demonstrate αSPIN abstraction manager capabilities to obtain reduced versions of the original model.

First of all, the PROMELA version with five threads and a pool of size six is exported to PiXL (some excerpts are shown in Fig. 9). Then, the abstraction manager GUI allows users to select variables to be abstracted from those available in the model. For each variable selected (connected in Fig. 8), the (concrete) operations in which it appears are analyzed and substituted by their corresponding abstract versions extracted from a predefined abstraction library (POINT in Fig. 8). The abstraction engine makes use of the standard XML DOM API to parse a PiXL model, transform it and generate its corresponding abstract version to be verified with SPIN. All the syntactic transformations are made
Fig. 8. The \( \alpha \text{SPIN} \) tool interchanging PiXL models.

within the PiXL document tree in memory. The abstraction engine is composed of specialized data structures (Java objects) to manage suitable references of PiXL variables and expressions. It also has containers to deal with the abstraction libraries; that is, collections of abstract operations to replace the original ones after the abstraction process. The abstraction libraries are also stored in XML format, according to an extension of the PiXL grammar.

The resulting abstract PiXL model is then returned to SPIN, which translates it back into PROMELA again. After the abstraction of the connected variable using the POINT library,\(^1\) the reduced state space has only 240,841 states. It would now be possible to check some interesting temporal formulas using less memory and time with this small example (see [6] for a more complex case study).

In order to manage not only data abstractions but also event abstractions, the abstraction API of \( \alpha \text{SPIN} \) was completely redefined in [7]. The new objective was to apply the same concepts to abstract UML Statechart behavior diagrams [12], 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. 10, where PiXL elements may be embedded into existing XML representations of a UML Statechart.

3.2. Integrating reliability and performance analysis

Reliability is critical in protocols in order to discard unexpected behaviors. Moreover, we may use performance analysis prior to deploying protocols in a real Enterprise, which helps to measure bandwidth usage, latency or congestion situations, among others. In [8] 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 specific inputs in order to analyze reliability and performance, avoiding the need for managing several hand-made specifications, and where the SPIN model checker was used to analyze reliability and the ns2 multi-protocol simulator

\(^1\) A very simple data abstraction scheme which maps every possible value of a variable to an UNKNOWN value, then discarding some execution paths.
Fig. 9. Excerpts of the original EJB model in PiXL.

[24] to analyze performance. Therefore, we have selected the use of the PiXL intermediate language to assist in the development of parsers and code generators.²

The viability of the approach in [8] is demonstrated in the context of the recent Active Network paradigm [26], which offers flexibility to develop new telecommunication services without the slow standardization process usually required by international institutions (ITU, IETF, IEEE or ANSI).

Active networks rely on so-called active nodes, which are programmable routers that execute code embedded into active packets or capsules. Active nodes are composed of Execution Environments (EEs) and the node operating system (NodeOS) which manages the access to network resources. The EEs provide a specific API to execute the code associated to an active packet. The NodeOS is responsible for implementing a set of abstractions that will give access to the node resources: communication channels for sending and receiving packets, and some storage facilities, usually consisting of some kind of soft-store data cache. Along with the active nodes, an Active Network topology uses active hosts which execute end-to-end active applications to send and receive active packets.

² It is worth noting that there are other approaches focused on verifying functional properties which have also developed additional specifications for performance analysis, such as [25].
It is worth noting that the SPIN tool has been successfully used to verify properties of Active Networks in [27], and the ns2 simulator with an extension for Active Networks (introduced in [28]) is used to obtain network traffic data, response times and other metrics for performance.

Fig. 11 shows the resulting models derived from a common PiXL representation which describe the behavior of part of an active protocol called Active Reservation System (ARS), introduced in [28]. This protocol is geared to travel agencies, providing mechanisms for client/server interaction in the form of data queries and responses, which are cached in the network. A sample use of an ARS application are air-flight status querying systems. The use of the Active Network paradigm overcomes the limitations existing in traditional web caching strategies for these applications, where data is dynamically changing all the time and the possible combinations of data in the requests decreases the cache hit ratio.

The solution followed by ARS is the cache with a per-item granularity, instead of caching entire web pages. This way, most popular items will yield a high hit ratio, no matter which combination is requested. The continuous change of data requires that requests specify a maximum acceptable degree of staleness, thus allowing clients to trade response time against currency of data. The code in Fig. 11 represents excerpts of the ARS request active packet. The left hand column is the representation in PROMELA for SPIN. The right hand column is the representation for ns2, in the OTcl language. Although the PiXL model includes the complete behavior of the active request, details regarding time were eliminated when generating the model for reliability analysis. Note that time features have been included in

```c
/* Promela specification for SPIN */
globals: capsule, EE

capsule.curr (not used in verification)

inline arsREQUEST()
{
short hops, /* number of (active) hops */
item, /* requested item */
timestamp;

atomic{
    hops = capsule.hops;
    item = capsule.item
    timestamp = get(item);
    /* warning: timed expression removed */
}
if ::(timestamp != NULL)
    /* warning: timed expression removed */ ->
    /*cache hit*/
    hits++;
    Capsule response = newCapsule(arsRESPONSE);
    response.hops = hops;
    response.item = item;
    /* warning: timed expression removed */
    sendTo(getSrc(),response);
    /*implicit discard and explicit forwarding */
}else ->
    /*cache miss*/
    ee.updateParam(0,misses++);
    ee.updateParam(1,hops++);
    ee.updateParam(2,item);
    /* warning: timed expression removed */
    sendTo(getDst());
}fi;
return 0:
    hops = 0; item = 0; timestamp = 0;
}
```

```c
# TCL specification for ns2
#
# hops = number of (active) hops
# item = requested item
# curr = currency
#
proc arsREQUEST { capsule ee } {
    set hops [lindex $capsule 0]
    set item [lindex $capsule 1]
    set curr [lindex $capsule 2]
    set timestamp [ee get $item]
    set now [ee getTime]
    if{($timestamp!="" && [expr($timestamp+$curr)<=$now]}{
        #
        # cache hit
        # incr (hits)
        $ee sendto [ee getSrc] arsRESPONSE \"$hops $item $timestamp\"
        $ee discard
    }else {
        #
        # cache miss
        #
        $ee updateParam 0 [incr misses]
        $ee updateParam 1 [incr hops]
        $ee updateParam 2 $item
        $ee updateParam 3 $curr
        # sendTo() is implicit
    }
}
```

Fig. 11. ARS request capsule for SPIN (left) and ns2 (right).
the performance version for OTcl, which uses variables such as $\text{now}$ (the current system time) or $\text{curr}$ (the currency of the item).

4. Related work

This section summarizes existing proposals for tool integration in the context of protocol analysis: ETI, CADP, IF, Veritech and Bandera. Only the last three proposals have decided to create a new intermediate language.

The well-known ETI platform was created to help with the integration and coordination among tools. In order to incorporate new functionality into ETI, it provides a C++ programming library used to encapsulate specific features of a tool which will be recognized as compatible by the global architecture of the platform, and available through so-called functionalty taxonomies.

CADP is another toolset used to analyze protocols described with the LOTOS [2] process algebra, it being possible to analyze models by using bisimulation and different variants of model checking. Although it was originally designed to work with LOTOS models, it now accepts other formalisms such as ECFSMs. Recent versions of CADP provide programming interfaces to facilitate the integration of new external tools.

The IF environment was designed as a flexible alternative to SDL, it being possible to express temporal concerns with asynchronous time automata, allowing the integration of static analyzers, model checkers and other exploration engines. IF provides translators for common notations such as PROMELA, SDL and UML to the transition-based IF language, along with a parsing API to aid in the integration of new tools.

The Veritech approach is a general framework to deal with translations among different verification tools, which is focused on faithful translations. It proposes a new transition-based intermediate language named core design language (CDL). The approach also proposes solutions to common translation problems between formalisms with similar semantics. Thus, given an origin format, its translation to CDL will contain details regarding the translation process, which may be used by destination tools to obtain the most suitable model from CDL in their input format.

The Bandera toolset interconnects different tools for the automatic verification of Java programs with the SPIN model checker. Its intermediate language is called BIR (Bandera Intermediate Representation), which is employed to interconnect existing modules in the toolset: slicers, abstraction modules and translators from Java and to PROMELA. This transition-based language is very close to PROMELA, but incorporates typical features available in modern programming languages, such as threads, locks and objects.

The main contribution of PiXL compared with the above approaches is the use of open standards to represent and manipulate the protocol information. These tasks can be done independently from any specific programming language. However, this is not the case of the ETI and CADP platforms, which have provided their parsing libraries in the C++ language. Note that all the approaches involving the creation of intermediate formats (IF, Veritech and Bandera) are mainly oriented to reliability and have not exploited novel concepts on integration which, again, should rely on widely adopted standards such as XML or MDA. Therefore, these (closed) intermediate languages cannot be easily extended to support other kinds of analysis, such as performance, without introducing in-depth modifications into existing tools and parsers.

5. 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 XML Schemas. 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 and its application in the domain of Enterprise Information Systems.
Following the proposal in [8] we are now exploiting some MDA guidelines [14] to transform so-called platform-independent PiXL models into different platform-specific ones. The former are general PiXL models which represent protocol behavior in a tool-independent way. The latter are modified (refined) PiXL models which incorporate those features needed by specific destination tools prior to code generation.

Our future work is also 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.

Acknowledgements

This work has been partially supported by the Spanish MEC under grants TIN2004-7943-C04-01 and TIN2005-09405-C02-01.

References