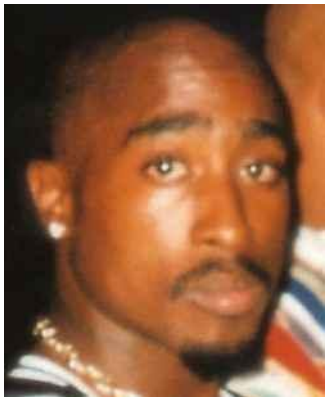
http://psychopath.sourceforge.net

PsychoPath
XPath 2.0 Processor

Fatima Ahmed Andrea Bittau Siwa Hemwarapornchai
Amir Hossein Fanian Shivraj Singh Kalyan
Oscar Kozlowski James Singer Ivan Sit

February 7, 2005

to Tupac Shakur...



Contents

Chapter 1

Introduction

XML Path Language version 2.0 is an expression language that allows the processing of values conforming to the data model defined in XQuery 1.0 and XPath 2.0 Data Model. The data model [13] provides a tree representation of XML documents as well as atomic values such as integers, strings, and booleans, and sequences that may contain both references to nodes in an XML document and atomic values. The result of an XPath expression may be a selection of nodes from the input documents, an atomic value, or more generally, any sequence allowed by the data model. The name of the language derives from its most distinctive feature, the path expression, which provides a means of hierarchic addressing of the nodes in an XML tree. XPath 2.0 is a superset of XPath 1.0, with the added capability to support a richer set of data types and to take advantage of the type information which becomes available when documents are validated using XML Schema. A backward compatibility mode is provided to ensure that nearly all XPath 1.0 expressions continue to deliver the same result with XPath 2.0.

1.1 New features in XPath 2.0

XPath 2.0 contains many new expressions types compared to its predecessor. This allows greater freedom and effectiveness in processing data. Most notable is the much wider support for different data types. XPath 1.0 could only address strings, numbers and booleans. In the new version, support for more complex types are available, such as dateTime, duration, binary, token, QName and entity types. With the increased support for data types comes the increased function and operator support. Version 2.0 can perform operations such as KindTest and AxisStep operations when version 1.0 could do relatively little in terms of data processing. Version 2.0 also supports XML Schema which adds support for custom rules and data types.

1.2 Existing Solutions

At the start of this project there was only one free and open source XPath 2.0 capable processor: Saxon [5]. However, Saxon itself has 2 versions, namely an open source and a commercial version. The Basic version does not support XML

Schema and try/catch capability for catching dynamic errors. A single user license for the Commercial version of Saxon costs 250 and 60 for each subsequence user license. This is costly and since Saxon was the only solution available, it had a complete monopoly. There is also another popular implementation called Jaxen [7], but it currently only is XPath 1.0 capable, and no development for XPath 2.0 seems active.

1.3 The Proposal

The goal of the PsychoPath project is developing a fully capable XPath 2.0 processor that is XML Schema aware, easily extensible and available as an open source project at no cost.

There were three paths we could have undertaken to accomplish our goals:

1. Extend the XPath1.0 processor, Jaxen, to support XPath2.0 and XML Schema.
2. Come up with our own implementation from the grounds up.
3. Extend the free version of Saxon to support XML Schema and complete Saxon's implementation of XPath 2.0.

After some discussions amongst us and our supervisor, we decided to pick the second choice and to come up with our own implementation. We rejected the other options because of the following main reasons:

1. Extending Jaxen or Saxon meant that we would have to understand how they work, thus increasing the learning curve of the project.
2. We wanted to have an easily extensible implementation, after looking at the available solutions, it did not seem too easy and elegant to extend the implementations, for example, by adding support for more XPath functions.

1.4 Superiority

Our implementation aimed to be easily extensible and with support for XML Schema, for free. We believe that by making our implementation open source and freely available to the community, those who are interested and/or looking to use a XPath 2.0 processor will pick us from other available solutions. By making it open source we will be able to benefit from contributions by interested developers, and possibly encourage them, to help make our implementation better and faster.

1.5 Limitations

At times, a general and extensible design will come at the cost of performance—at this stage, our goal is not speed. Currently, not all functions and operators in XPath 2.0 are currently supported by our implementation, mainly due to the short time available for this project. However, the most critical components of

the specification have been implemented and are mostly fully working. On the other hand, due to the design of the processor most bugs can be easily located and solved as they emerge.

1.6 The Impact

Since this is an open source project many users and developers who are looking for an XML Schema aware XPath 2.0 processor will be interested in our implementation. It will also be a very attractive solution from the commercial point of view as well, as this would save users the licensing fees, and also cut the maintenance delay which exists in commercial products as the developers within the organization can easily work on the implementation themselves and modify the processor to suit their needs. Because of the relatively clean design and documentation accompanying PsychoPath, the learning curve for new developers will be less steep giving them a chance to quickly start working with the processor and adding additional components.

1.7 Team Organization and Operations

For operations between the team to progress smoothly we needed to share code and make sure files would not clash and get lost. In order to achieve this we used Concurrent Versions System (CVS) as our versions control system, in order to maintain and share files that the team would need. Weekly meetings between all team members were held to ensure that everyone understood their tasks and in which problems could be conveyed to the rest of the group were hopefully answers could be obtained. Also a key factor in maintaining a daily communication channel to which the whole group could keep in contact was the use of an Internet chat system. A perfect way to keep in constant contact during implementation of work, so that any problems could be solved in real time rather than having to e-mail or bring it up during a meeting and therefore wait for the same response.

Diversifying the team roles gave each member a possibility to learn and become more skilled in specific areas of expertise. Many roles relied on other ones and without each member accomplishing their task the project would suffer and progress slower, possibly stalling. We decided to give each head role a deputy team member that would allow a certain degree of safety such that in an event where a team member in an important head role could not accomplish their tasks, another worker could be called upon to takeover. By a unaminous decision the team's Technical Director was elected to oversee the project and he had the respect of the other members and could effectively distribute the workload evenly. He also had the knowledge and the skill to manage the technical structure of the project and help members with technical queries. The various team members and their main goals are summarized below:

Andrea Bittau Technical Director, Core Coder.

Siwa Hemwarapornchai Project manager, Core Coder, Project Report.

Shivraj Singh Kalyan Tester, Editor of User Manual.

James Singer Tester, Editor of Project Report.

Ivan Sit Presentations/Documenter, Editor of Project Report and User Manual, Coder.

Fatima Ahmed Documenter of code and weekly log.

Oscar Kozlowski XPath Expert, Coder, Deputy Technical Director.

Amir Hossein Fanian XPath Expert Deputy, Deputy Documenter of code.

During the course of the project implementation roles began to merge and because of tight deadlines and other external work commitments, team members began to undertake other people's jobs in order to keep the project running as smoothly as possible. In the end everyone became responsible for each other—if one failed then we all failed unless we helped or did the job ourselves. It became clear as the project progressed that some team member's abilities were better matched to other specific tasks and therefore in a natural step, members began working on other tasks outside of their own allocated area of expertise, sometimes effectively switching roles with others.

Testing was a task to be carried out by all team members. Each member was assigned specific XPath expressions to test on PsychoPath. The group report was also split into sections for members to write up. Whoever took a major role in a certain job aspect (e.g. design, testing etc) would write up that particular section. To meet the deadline of the final deliverable on time, members that tested a particular XPath expression would be in charge of documenting the Java files in question with the use of Javadoc. This became a large part of the team member's workload as the end of the project implementation moved closer.

Chapter 2

Contract

This section publishes the contract drawn up for the client in order to maintain an understanding of the objectives and milestones that are required to be achieved between the project team and the client in question. A preliminary draft was drawn up and was signed by the team and the client. The contract was not changed or updated during the continuation of the project. It is included in this section.

2.1 XPath 2.0 Processor Contract

Group Supervisor:	Prof. Anthony Finkelstein
Group Members:	Ahmed, Fatima Bittau, Andrea Hemwarapornchai, Siwa Fanian, Amir Hossein Kalyan, Shivraj Singh Kozlowski, Oscar Singer, James Sit, Ivan

Overview

XPath 2.0 is an expression language which provides a means of hierarchical addressing for the nodes of trees that represent XML documents. XPath 2.0 is a backwards compatible superset of XPath 1.0 with the added capability of supporting a richer set of data types including integers, strings, booleans and links to other XML documents.

The purpose of this project is to design and implement an XPath 2.0 processor. This will allow a user to extract and address specific information from an XML document. No XPath 2.0 processor has yet been implemented, as this is a relatively new standard.

This document defines the contractual agreement between the developers and the client regarding this project.

Project Development

An extreme programming approach will be used by developing the code in small increments. Each increment will fulfill a subset of the requirements, which will involve designing, implementing and testing. Upon completion of each increment, we expect client feedback in order to make corrections and changes. When enough satisfaction is reached, the next subset of requirements will be implemented.

For a successful result, the components of the software need to be fragmented and prioritized appropriately. A schedule of when components must be completed by will be provided in order to make sure that enough progress is made.

Each iteration of code development will involve:

Requirement Analysis The aim is to produce a complete description of the problem and of the requirements according to the specification of creating an XPath 2.0 processor. It will prioritize the functionality which should be implemented, whilst including possible future extensions within the project time scale and constraints. A conceptual interface of the XPath 2.0 processor will also be developed.

Design A model of the entire system will be developed and decomposed into definitions of components functionalities and interface. Issues on how the requirements should be implemented will be explored and analyzed. Decisions on modularity and extensibility will be addressed. The result will be a UML specification of the major classes making up the various components and their interfaces.

Implementation The implementation should resemble the design as much as possible. Some design decisions may have to be re-evaluated to ease implementation. Performance issues will be addressed which may involve reducing code flexibility and increase coupling. The main aim will however be making the code robust and possibly scalable to ease its extension in the case a new XPath specification becomes available.

Testing and Evaluation A test suite will be developed throughout code implementation, after each increment of implementation testing will be used for validation on the user requirements. Once a beta version is reached, there will be extensive final testing in order to see whether the product meets its requirements. An evaluation of the product's performance may be conducted. As there are currently no other implementations only comparisons with XPath 1.0 processors may be made, which will only give a rough estimate of how good our solution is. Most importantly however, a relatively high confidence in the code's stability will be guaranteed by the test suite and its results.

Communication between the developers and the client is essential for a successful implementation. Developers must know the desires of a client before they may proceed, and the client will have a chance to see whether the project being created matches his ideal.

Deliverables

23 November 2004	Interim report. This will stipulate, in depth, the system requirements (eg. MoSCoW), use cases and analysis model.
18 January 2005	Beta release. Usable but unstable version of software.
8 February 2005	Final and individual reports, executive summary, group diary and a working prototype.
23 February 2005	Presentation.

Further Agreements

- Final dates for each of the milestones in the project development are to be supplied within three weeks.
- Brief presentation outlining the architecture of the software to be developed.
- Allocation of Intellectual Property to University College London.

2.2 Summary of Contract

Due to the extensive research required in understanding the problem in hand, we omitted the specific milestones of the project and their due dates. In the final contract however, we included a date in which we would deliver a document 'Milestones and Dates' containing the milestones, their due dates and an explanation of what each deliverable would entail. We did however include in the contract dates regarding deliverables of the end product, interim document, beta release and final release.

2.3 Success of Contract Obligations

This section details the team's success (and failures) in reaching and completing the milestones according to the milestone document submitted. Any milestones which the team was unable to complete on due date are explained and detailed with reasons why we think failure occurred and how we were able to remedy the situation.

The milestones are as follows:

15 November 2004 XPath 2.0 expression parser. We were able to complete this milestone ahead of schedule on the 11 November 2004. The system as it was constructed was able to parse XPath expression and load XML documents as DOM.

29 November 2004 [1] Static checking. We failed to deliver this milestone on time. During the course of this milestone we encountered several fatal issues which lead to our failure. Firstly, the estimated time for our 'XPath experts' to become proficient in XPath was slightly too early, however,

[1] In this day, one of our most loved group members turned 21.

even with extended time this was still not achieved. This is very important as the core coders and testers needed to be able to refer to the 'XPath experts' in order to complete their assignments.

Secondly, there was a significant drop in our communication leading team members not knowing what was completed and what still needed to be finished. These issues could partly be blamed for the timing of this milestone to clash with other course assignments, however, this could not be used as a significant excuse. It also taught us that we had to be more aggressive in meeting the deadlines and much more communication between the team was needed for the future success of the project.

Technically we failed for many reasons: firstly we did not consider XML schema and did not understand that "expression normalization" was such a large task to undertake (a lot of time was spent only on deciphering formal specifications).

The remedy was to try and do the opposite of what caused our failure. Upon failing to meet the deadline for our client, the team communicated on a more regular basis and used more means of conversing i.e. with the help or phone calls, e-mails, Internet chat and weekly meetings. Meetings ensured everyone had a chance to discuss problems, find help and make sure that they understood and knew what they had to accomplish by specific dates. The team also began to spend more time on the project as other course assignments were completed. We adopted a new method of keeping track of what each team member was doing and how they were moving along. This was a simple weekly status report sent by email to the project manager. All that was needed was a brief summary to explain our own situation and therefore the project manager could get an overview of how the entire project was progressing and make sure that everyone was meeting their deadlines.

The team decided to push all future deadlines a week back in order to accommodate the failure to meet milestone 2. Milestone 2 was completed on the 17^{th} of December 2004.[2]

17 December 2004 XPath execution. The team reached milestone 3 on January 9^{th} 2005 which was just before milestone 4's original deadline. We were therefore back on schedule but pushing for the last milestone and deliverable. The system as it stood was a complete structure that contained basic functionality. It was a full XPath processor with the restriction that it only supported part of the complete specification (such as functions and types).

18 January 2005 Additional XPath implementation. Finalized on January 16^{th} 2005, milestone 4 was reached earlier than the set deadline and was an almost complete XPath 2.0 processor compliant to the specification. With a fully tested and working beta version of PsychoPath, we would then release it on SourceForge.net.

[2]Friday 17^{th} is the most unlucky day in Italian culture. Fortunately our Italian group member is not superstitious and did not protest by trying to push the deadline earlier.

Chapter 3

Requirements Specification

The XPath 2.0 specification [11] defines a large and complex language. A full implementation of it is a demanding task. The time available for developing PsychoPath was not sufficient to expect a full implementation of the standard upon completion of the project. What was certainly expected however was a well tested and documented product. Although the implementation may not be complete, whatever has been covered is accompanied with appropriate tests and documentation.

The strategy used in developing PsychoPath was Extreme Programming. The development process would proceed in small increments where a complete software engineering cycle would be performed in each single iteration. Upon completion of each increment a requirements analysis was sent to the client from which we then received feedback.

The client would not receive a version of the code on each increment as the results from early milestones were rather intangible such as parsing an expression into a tree. Most of the early code was a preparation for the core engine of the project. It was only toward the end of the project that it actually became usable. The result of this is that the client could not get a direct feel of the use and performance of our product throughout its early development and thus the feedback we received was almost always positive and not too in depth. In a way, the client had to trust we would manage to cross the final line before he could enjoy our product. Hopefully, we did so.

The project was split into four main problems. These defined the four core milestones in the development process. A brief description of the milestones follows:

Milestone 1 XPath 2.0 expression parsing and DOM [10] loading. This milestone consisted in implementing a parser and internal representation for an XPath expression. Also, it ensured the capability of loading XML files as DOM.

Milestone 2 Static analysis of XPath expressions. This involved static name checking which ensures the syntactic validity of an expression. It also involved implementing an XPath expression normalizer which reduces an XPath expression from the full grammar to an equivalent expression in the core grammar.

Milestone 3 Dynamic analysis of XPath expressions. The implementation of an expression evaluator had to be completed. At this point, although limited, our project was suable by the client. An XPath could be evaluated against an XML document.

Milestone 4 Implementation of XPath functions. This final milestone added capability to the expression evaluator by the implementation of functions defined by XPath. This enabled more useful and powerful expressions to recognized.

Upon completion of the fourth milestone a final iteration has been conducted in order to resolve outstanding issues and package the software appropriately. This finally iteration was highly important for testing as most of the test cases were developed during it.

3.1 Overview of the Requirements

The main requirement for this project was extensibility and good design. Knowing the full specification could not be implemented in the given time, there was a need to insure the architecture would easily be understandable and extended by a future team which may have an interest in continuing our work.

The most practically important requirement is perhaps the performance of our solution. This was never a requirement of the project, at least during its main development phase. It was necessary to build the system before looking at how it may be improved for speed. However, the client showed interest in performance toward the end of the project. Although not enough time was available to dedicate a milestone on optimizations, an attempt has been made to study performance aspects of our project and its results will be presented in section 6.2.

The goal of the project was to implement as much of the specification as possible. It was early recognized that it was impossible, or at least very undesirable, to focus only on certain parts of the XPath 2.0 grammar and implement them fully. The reason being the highly recursive definition of the language. Many production rules toward the end of the definition point back to the first ones. An example is the production rule 40 (predicates) out of 80, which points to the production rule 2 (expressions). Thus, it became a core requirement to implement all of the XPath 2.0 grammar which would provide the backbone for the entire project. On the other hand, implementing all functions and aspects defined by XPath was not a key requirement although the effort involved to add such functionality needed to be minimal, which enforces the main requirement concerning design.

The requirements for each milestone, in general, contain two main parts. The first part are the core requirements which must be implemented. The second part consists of optional requirements which mainly indicate what may be implemented in the future to enhance the product. In both cases, requirements are prioritized. The following sections describe the requirements for each milestone and are listed according to their priority in descending order.

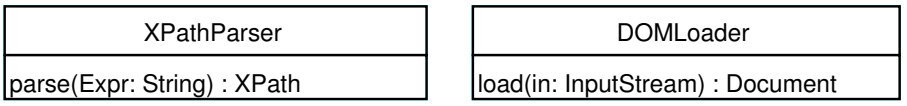


Figure 3.1: Interfaces which need to be implemented as a result of milestone 1. The XPathParser encapsulates all parsing details and returns an XPath tree for a given expression. The DOMLoader will perform a similar task by returning a DOM object corresponding to an input stream.

3.2 Milestone 1 Requirements

The key objective for this milestone was implementing an XPath 2.0 expression parser. The important aspect of this is choosing an appropriate structure into which the expression will be parsed. The result of the parsing will be a main data structure used throughout the entire project. A less complex objective included in this milestone was the need for a DOM loader. Figure 3.1 summarized the interfaces which had to be implemented.

This was a high risk milestone. Pushing the parsing of the *whole* XPath 2.0 grammar in the first milestone was a good move as it ensured the foundation of the project being present early in the development process. In case of failure, there was enough time to understand why and plan a new strategy.

Core Requirements

- Isolate the parsing engine from the internal representation used for an XPath expression.

  This will allow different parsing strategies to be implemented and switching between the implementations should be achievable almost automatically. The concern was not to couple the rest of the project with the specific parsing scheme used. By doing so, an optimized parsing engine may be developed in the future and replaced as the default mechanism with minimal effort.

- Reliable and scalable method for testing the parsing of expressions.

  An obvious way to check whether the parser executes properly is to assert it raises an error upon an invalid input expression. However more needs to be done in order to ensure the expression is parsed correctly upon success. Such method needs to be devised. It must also be efficient to add test cases using this approach.

- Encapsulate mechanism for obtaining a DOM and allow for optional XML validation.

  As with XPath parsing, the DOM interaction with the rest of the project must not be tied to a specific implementation. This will enable a simple way to switch the DOM implementation used if more powerful ones arise in the future. XML validity checking is an important requisite, and more specifically, XML Schema support is highly desired.

- Uniform and effective error reporting scheme.

The previous requirements suggest an architecture which promotes the ease of switching between implementations which performing the same specific task possibly in a different way. What also needs to be common is the way errors are generated an translated to the user. It is best if the user is not aware of the different implementations.

A large proportion of errors will occur during the XPath parsing and DOM loading phase which give a further incentive to concentrate more on error reporting issues. An important aspect is to give as much information possible on an error and its locality. Not only it is important to indicate the presence of an error, but its location must be clearly revealed—XPath expressions may be very long.

- Effective mechanism for testing DOM loading.

  Although the providers of the DOM implementation are responsible for testing their products, it is in our interest to ensure we are using their solutions correctly. This aspect of the project should not be thoroughly tested but at least a minimal attempt should be made in order to give more confidence that all the components behave properly.

Optional Requirements

- Provide different parsing implementations.

  This aspect will tackle performance requirements. In the case of long expressions, a fast parser will result in high speed gains. This is also true when multiple different expressions need to be executed sequentially.

- Provide different DOM loading mechanisms.

  Loading DOM is a performance bottleneck. Frequently it may be the case that multiple different expressions need to be evaluated on the same document so the DOM loading overhead will only be dealt with once. Other than speed, an important aspect of different DOM implementations is what features they provide. An example is XML Schema validation which is supported only by particular implementations. In our case, the highest benefit obtained from using a different DOM implementation was the extra features provided rather than its performance.

Implementation Status

The PsychoPath implementation satisfies all the core requirements. There is a slight issue however about not tying a particular DOM implementation to the project. Currently Xerces [1] is being used as it is the only implementation we are aware of which supports XML Schema. Most of the schema code is dependent on Xerces although a common wrapper should be possible to implement. Xerces seems to be the default DOM implementation in Java 1.5 so it should not be a problem if PsychoPath is tied to it.

Initially our implementation used the default Java 1.4 DOM implementation. Later it was modified to use Xerces, and the process of switching was very simple. This proved we definitely met one of our requirement. Also, by doing so, the user now has a choice of using Java's default DOM implementation or

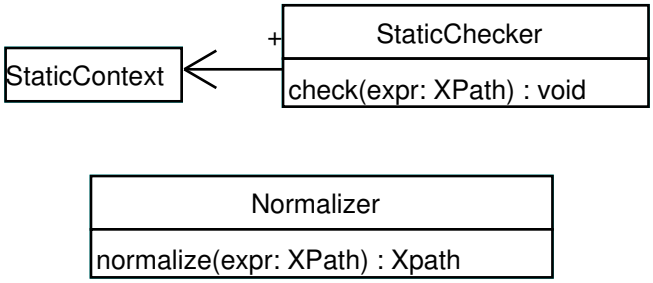


Figure 3.2: Interfaces implemented during milestone 2. The static checker uses the static context to resolve names. The normalizer will convert an input XPath representation into a normalized version.

Xerces thus satisfying one of the optional requirements. On the other hand, no alternate XPath parsing mechanism has been implemented other than the default one.

3.3 Milestone 2 Requirements

The whole of this milestone dealt with static analysis. Anything which may be processed on an XPath expression without the use of the XML source (except for its possible schema) is referred to as static analysis. This mainly involves checking for error which may not be detected at parse time. Another aspect of static analysis is normalization which transforms an XPath expression in a "simpler" version.

Normalization is partly what made us fail the delivery of this milestone on time. There was an underestimation of the problem. Ironically, our project currently does not support normalization anymore as it is seen as a performance overhead, thus not implementing it at all would have been fine. Figure 3.2 illustrates the interfaces to be implemented.

This milestone was of high risk because it would be the first time the data structures realized in the previous milestone be used to produce some results. If the choices previously made were inadequate major redesigning would have to be carried out. Fortunately they were not. Although we failed to be on time for this milestone, there was plenty of time to replan and get back on schedule.

Core Requirements

- Implement a static context conformant to the XPath 2.0 specification.

  The static context is the core component used during static analysis. It contains information about prefixes and namespaces, variables, functions, etc. The addition and retrieval of information from it must be performed via an elegant and simple interface.

- Static checking of all names in expressions.

  This phase of static checking ensures the user has not misspelled names or referenced unresolvable namespace prefixed.

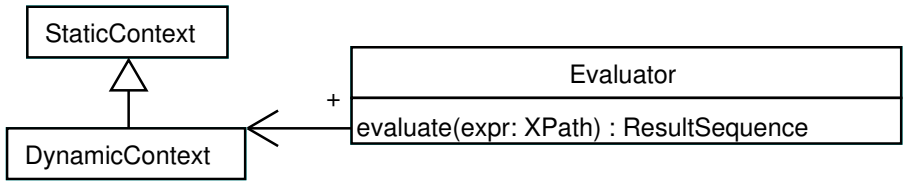


Figure 3.3: Milestone 3 interfaces. The evaluator will execute an XPath with the aid of the dynamic context and return a sequence. Note that the XPath class may not be evaluated by itself. It needs an external evaluator, thus making it possible to implement various versions.

- Normalization of expressions.

  It seems that normalization is a requirement for conformance to the XPath specification. It was therefore necessary to implement it. In practice, it performs syntactic changes to an expression without changing its semantics.

Optional Requirements

- Static type checking.

  The XPath 2.0 specification defines an algorithm for inferring the types of expression. These types may then be checked to ensure the correctness of an expression. This is an optional feature in XPath which when implemented may detect type errors during static analysis instead of during run time evaluation.

Implementation Status

All core requirements have been implemented. However normalization is no longer maintained as it is seen only as a performance overhead. For the same reason the optional requirement of static type checking has never been implemented.

3.4 Milestone 3 Requirements

This was perhaps the most important milestone. All the preparation and code developed so far was necessary to support the execution of an XPath expression. The goal of this milestone was to implement the ability of evaluating any XPath expression. The only limitation would be the number of XPath functions and types available to the user. The types must be conformant to the XML Schema specification [12].Figure 3.3 identifies the main interfaces this milestone will deal with.

This milestone had the highest risk factor. It would be the revelation whether our design choices and predictions were suitable. Considering this milestone was delayed due to the previous one, failure at this point could have meant quite negative consequences. However, luck was on our side. There was no way to

push this milestone earlier as the foundations deployed by the previous two iterations were crucial in order to start tackling this milestone.

Core Requirements

- Implement a dynamic context as required by the XPath specification.

  The dynamic context keeps all the run-time state associated with the XPath expression execution. It is an extension of the static context. For example it has the ability to store values in variables and dispatch function calls.

- Have the ability to evaluate all types of XPath 2.0 expressions defined in the grammar.

  This requirement regards the evaluation of all *forms* of XPath expressions. Although only a minimal subset of types, operators and functions will be supported at this stage, all the different ways of expressing concepts with them should be implemented.

- Design an architecture where different XPath evaluation implementations may be used.

  The XPath evaluation engine will be the performance bottleneck in the general case. Having an architecture which promotes the ease of switching implementations may have benefits. The concept may be extended further by using specific evaluators which are optimized for particular expressions when such expressions arise and may be detected.

Optional Requirements

- Support for XPath 1.0 backward compatibility.

  The static context has a flag indicating whether the operation mode should be backward compatible with XPath 1.0. In various cases, the specification refers to this flag and an expression may have different semantics depending on the setting of the flag.

Implementation Status

The core requirements have all been implemented. An interesting aspect to be examined in the future would be how much expressions evaluators may be optimized. Also, if classes of common XPath expressions exist for which a very specific and highly optimized evaluator may be implemented.

Backward compatibility with XPath 1.0 not implemented mainly due to time constraints and being busy on implementing more important aspects of the project.

3.5 Milestone 4 Requirements

In this final core milestone the responsibility was to implement as many functions, types and operators as possible. However, more importantly, an elegant architecture for these components had to be designed. Once this was

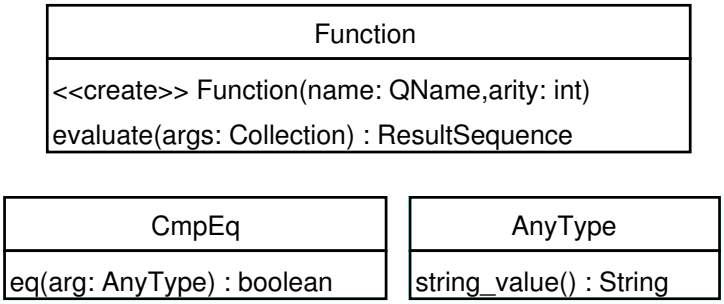


Figure 3.4: Interfaces in milestone 4. A function has a requirement of having a name and an arity—its function signature. It must also implement an evaluate method which will return a sequence based on the input arguments. Types must have a string value as defined by the specification. If they wish to support an operator they need to implement the appropriate interface. The interface for the equality operator, CmpEq, is shown.

accomplished, the addition of functions would be trivial. Figure 3.4 displays an overview of the desired architecture.

There was very little risk involved in this milestone. The main design of the project was complete and functionality had to be added in a very systematic way, almost without thinking. In case of failure, the end user could not be as expressive as he/she desired by not being able to use the whole set of functions, operators and types defined by the XPath specification.

Core Requirements

- Define a simple mechanism for adding a function, operator and type.

  The effort of defining a function should be minimal for the project to be scalable. The project should not depend on "knowing" which functions are present and which are not. The same is true for defining a new type and operators associated with it. For example, each new type has a default constructor associated with it. The developer of a new type should not be involved with defining such a function, but it should be handled automatically by the rest of the implementation.

Optional Requirements

- Implement as many functions, operators and types as possible, giving priority to the most relevant.

  The more functions, operators and types are present, the more useful and expressive the language is. The architecture of the project however, will not change at all. By adding further functions we are helping future users, rather than future developers which we have been aiding in design up to this milestone.

Implementation Status

The core requirement has been fully satisfied. Many functions, operators and types have been implemented as suggested by the optional requirement. A reference on the full implementation status of the XPath 2.0 specification is available in appendix B.

Chapter 4

Analysis and Design

The XPath specification proposes a processing model as shown in figure 4.1. This model was the base for PsychoPath's design.

By following the diagram, the parsing of an XPath expression yields an *Op-Tree*. The variety of nodes this tree may contain is reflected by the different types of XPath expressions which exist. For example nodes representing expressions such as and, or, plus, etc. need to be implemented. Having a grammar of 80 production rules, and almost each one of them representing a different type of expression, it became clear that a separate package should be dedicated entirely to this Op-Tree. This lead to the definition of the *ast* package which contained the Abstract Syntax Tree nodes representing the various XPath expressions.

The next interesting component in the processing model is the execution engine. Much of the detail is hidden for clarity in the diagram. After some research on what exactly is needed for a complete execution of an expression, two main components were revealed: types and functions. These aspects of XPath are treated in a separate document [15]. Currently, over 90 functions are defined and we realized an extensible design for functions needed to be devised. They were all grouped in the *function* package.

A large number of types is also defined. Also, each type implicitly defines a constructor function and has specific interactions with operators defined on it. All matters regarding types and the type system are grouped in the *types* package.

The link between all these components is the main *xpath* package. Here the main interface to the client is defined, where components such as the execution engine are present.

Finally an additional package containing all the test code, called *xpathtest*, was created. The primary reason for separating the main code from the tests was to facilitate the distribution of the product in case some users were merely interested in the actual processor implementation without the accompanying test suite. Also we believe there is more order in the code's structure this way, possibly aiding development.

Figure 4.2 summarizes the packages present in PsychoPath and their relationships. The design and rational for each package will now be presented in turn, except for the xpathtest package which is discussed in section 6.1.

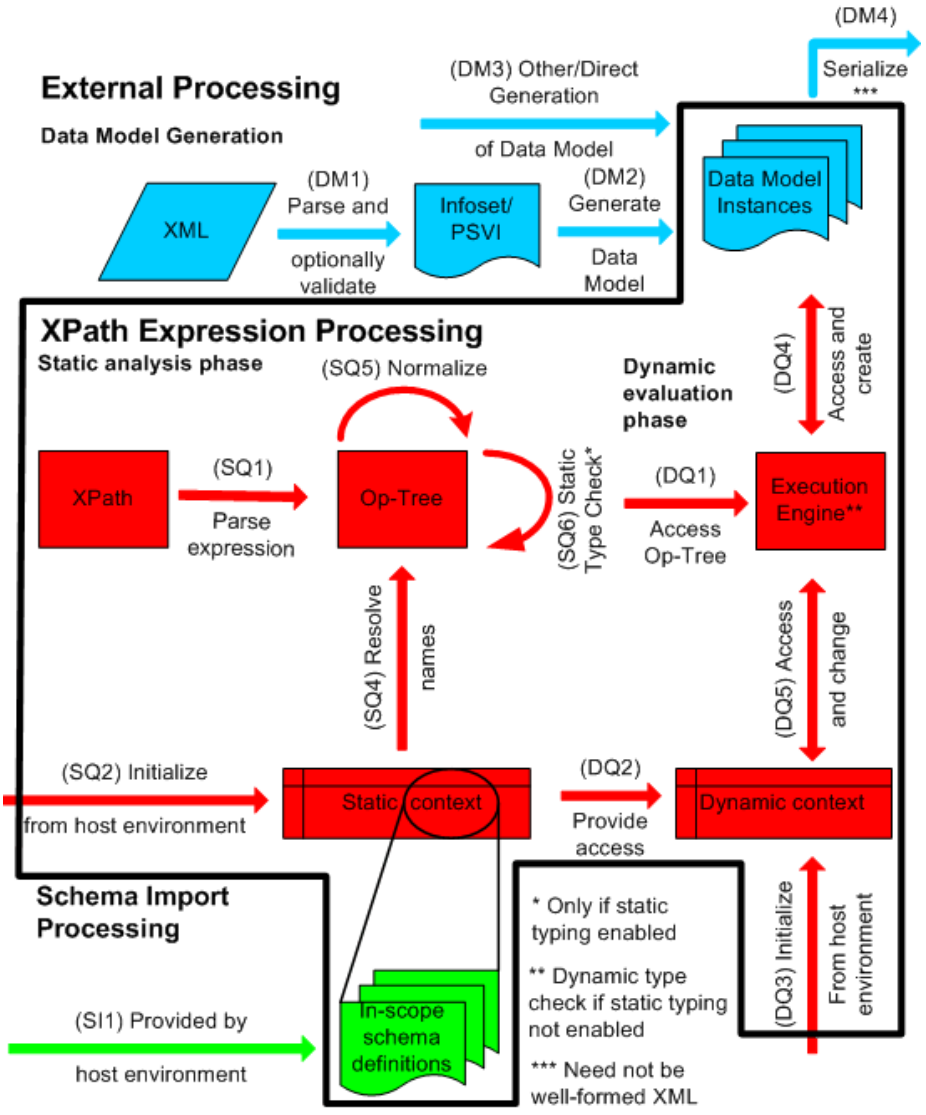


Figure 4.1: Processing model taken from the XPath 2.0 specification.

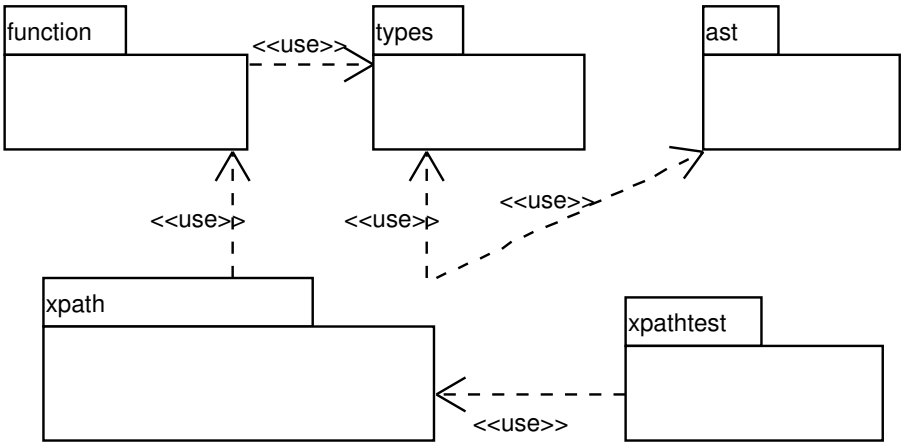


Figure 4.2: Packages present in PsychoPath. The main xpath package makes extensive use of its function, types and ast sub packages. However, the xpathtest package will only make use of the main user interface as explained in section 6.1.

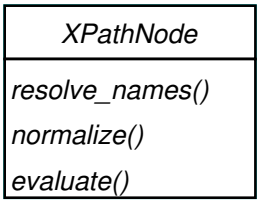


Figure 4.3: A way of defining the base class for all ast nodes. All operations are supported directly by the nodes themselves.

4.1 ast Package

The ast package represents the Op-Tree defined in figure 4.1. From the diagram, it is evident that various operations need to be performed on the ast such as resolving names, normalizing, executing and so on. These operations will lead to a definition of a base class for a node in the ast.

A first attempt in defining such a base class may be as shown in figure 4.3. There are several reasons why this design did not seem suitable. Firstly, if a new operation had to be added, all the nodes in the ast would have to be modified (currently over 60 classes). By by examining the details more closely, it is not yet clear what should the parameters and results be of these operations. For example, what arguments should the evaluate method take? What is necessary for full evaluation of an XPath? Another problem of this approach is that it gives more functionality to the ast than desired. The ast should represent expressions and not the way in which they are evaluated or statically checked, since the xpath package should deal with that responsibility. Furthermore, the code for these operations would be spread across all the ast classes instead of being present in one single location reflecting the single logical operation. Various implementations of the same operation would be impossible to be present in an

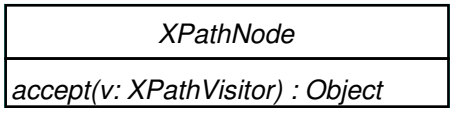


Figure 4.4: Current design of the base class for all ast nodes with support for the visitor pattern.

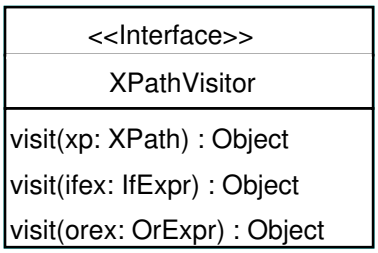


Figure 4.5: XPathVisitor interface. Only two three ast nodes are shown for clarity—all of the nodes must be implemented.

effective manner using this design, which defeats one of our main requirements.

The solution to these problems was to implement the visitor pattern [3]. One of the disadvantages of the visitor pattern is that if the ast changes, all the visitors would need to be updated. This is not at all a limitation for two main reasons. Firstly, a change in the ast would mean a change in the XPath language, which is not likely unless version 3 of XPath will be developed. Secondly, in the case of a change, if the previously described design was used new code would have to be implemented to support each novel expression. The amount of effort to update the visitors will be the same. The real downside of the visitor approach is performance. Instead of performing a single virtual call, the overhead of double dispatch will incur—the ast node will invoke the visitor, and then the operation will be executed.

The current design of the ast base class is shown in figure 4.4. No longer there is a question what each operation should return and what arguments it needs for performing correctly. All this information is defined by the specific implementation of the `XPathVisitor`. Also, all code for an operation will be contained in a single location. Implementing a different version for an operation is achievable by writing a new visitor—no changes to the ast are necessary.

In order to write a visitor the interface in figure 4.5 must be implemented. The visitor is then used by passing itself as the argument to the accept method of an `XPathNode`. The result will be whatever the visitor defines it to be. A wrapper to the visitor may be implemented which returns a specific type rather than `Object`. The following fragment of code depicts how a visitor may be developed:

```
public class Printer implements XPathVisitor {
    public Object visit(XPath xp) {
        System.out.println("Node is XPath");
        return null;
```

```
    }
    public Object visit(IfExpr ifexpr) {
        System.out.println("Node is IfExpr");
        return null;
    }
    ...
}
```

Finally, the visitor may be used in the following way:

```
XPathNode node;
XPathVisitor visitor;
...
Object result = visitor.accept(node);
```

Now visitors performing name resolving, normalization and evaluation may be implemented without any changes to the ast.

4.2 types Package

This package faces a similar problem as the ast one. In both cases there is a relatively complex class hierarchy and the single base class is used throughout the code at most times. The developer does not know the concrete types of the objects being manipulated although the semantics of the operations highly depend on them. Polymorphism is the main tool which needs to be used effectively to solve this problem.

Types not only need to represent values, but in most cases they must also support operators. Unary operators may be implemented very effectively with standard polymorphism by simply applying a method representing the operation to the type such as:

```
AnyType t;
...
t.negate(); // unary negation
```

The problem is more complex with binary operations. In this case, the semantics of the operation depends on two types and not only a single one. Consider the following example which attempts to solve the problem using standard polymorphism:

```
AnyType left, right;
...
left.plus(right);
```

Although the problem of the plus operator may seem solved, it is not. How is will the plus method actually be implemented? Using the solution described above, only the type of the left argument will be revealed but not the right one. A naive implementation would be:

```
public Object plus(AnyType right) {
    if(right instanceof AnyNumber) {
        ...
```

```
    } else if(right instanceof AnyString) {
        ...
    }
    ...
    else
        return null; // ???
}
```

An interesting solution to this problem are multi methods as described in [8]. Briefly, it consists in registering all permutations of operations and arguments to a hash table. When an operation needs to be performed on two unknown types, a key consisting of the operation and concrete types (obtainable via standard polymorphism) is computed. The key is then used to lookup the hash table and retrieve a function pointer to the specific operation required.

PsychoPath's implementation adopts the standard polymorphism mechanism. The main reason for not using multi methods was that we were not confident enough on how to implement it with Java (no direct support for function pointers). It also turns out that many operators require specific right arguments for specific left arguments, i.e. the types of the left and right arguments need to be always the same. Thus the previous implementation described for plus would simply be, in pseudo code:

```
if(right instanceof typeof(this))
   // do operation
else
   // error
```

4.2.1 Type hierarchy

XPath defines a large type hierarchy. Figure 4.6 summarizes a portion of the current implementation of the type hierarchy. Classes of special interest are the base `AnyType` and `CtrType`. Their definition is illustrated in figure 4.7.

The `AnyType` class will define the requirements for implementing a new type. Currently these are to provide a string representation of the value which they hold. This is a requirement in the XPath specification. Additionally, types must provide a string representation of the type name. This requirement is purely for PsychoPath and its main reason is for testing. It makes it much easier to write test cases by specifying the expected type as a string rather than having to use some Java operators or other mechanisms to determine the type of the result. Also, these two methods are very useful for debugging.

The methods to be implemented for a `CtrType` are both required by the XPath specification. The obvious method is `constructor` which will construct the type from the argument supplied. The implementor is required to sanity check the arguments and must not alter them. The `type_name` method will return the name of the implicit constructor function defined in order to create this type. An example would be the type name of "string" which will define a function available in XPath to construct string elements by invoking an expression such as `string("w00t")`. The developer of a type will not need to create the code for a function constructing the type but rather, as later explained, he/she simply needs to add the newly developed type to a special function library.

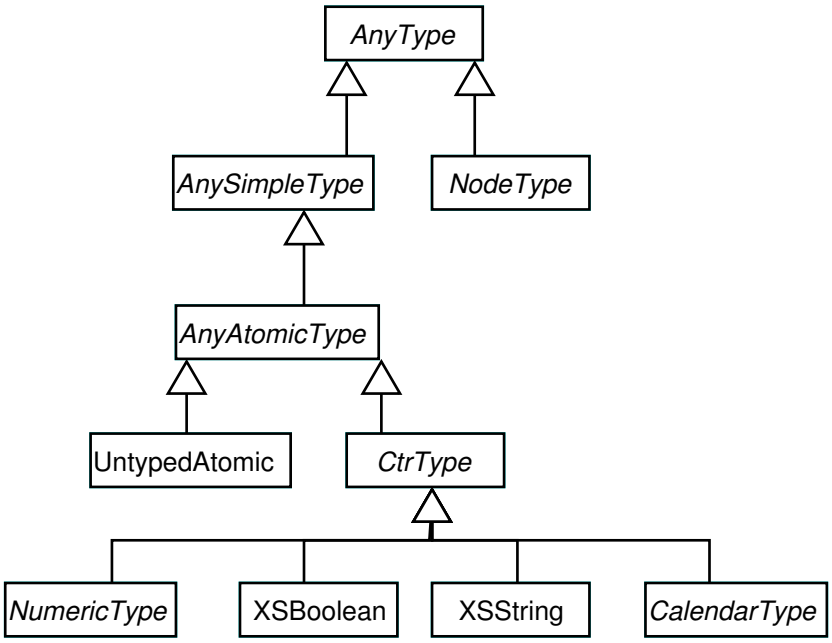


Figure 4.6: Part of the type hierarchy currently implemented.

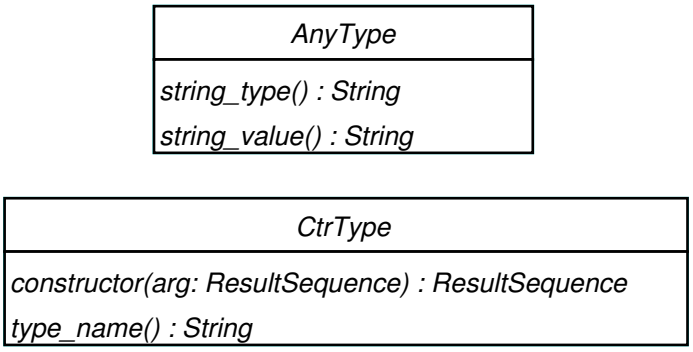


Figure 4.7: The design of the base class for all types AnyType. CtrType represents the base class for all types which may be created using constructors in XPath.

4.2.2 Operators on Types

The way operators are treated in the main implementation is purely syntactic. If a type is to support a certain operator, a specific interface needs to be implemented. It is up to the implementation of the type to provide the semantics of the operator. There is a interface to be implemented for each symbol representing an operator. For example if a type requires support for the plus and minus symbol a class such as this may be defined:

```
public class MyType extends AnyType
                    implements MathPlus,
                               MathMinus {
  ...
  public ResultSequence plus(ResultSequence arg)
                              throws DynamicError {
     ...
  }
  public ResultSequence minus(ResultSequence arg)
                               throws DynamicError {
     ...
  }
}
```

The implementor is responsible for sanity checking arguments. Since the result is a sequence, the implementor has total control over what the result is, and thus the semantics of the operator is. For example, the plus may be used for string concatenation and not necessarily addition.

This flexibility has been reduced toward the end of the project as explained in section 6.2 mainly due to performance issues. Comparison operators no longer use sequences as arguments and return values, but rather they are restricted in returning a primitive boolean and taking in a single type for an argument. This is always the case in XPath so it is not a limitation. However it shows that design rational and generality sometimes needs to be given up for the benefit of performance. Figure 4.8 shows a subset of the operator interfaces some of which use the original style and others the revised design.

4.3 function Package

The XPath specification contains many functions, thus the ease with which a single function may be implemented had to be maximized. Another problem is locating the relevant function and dispatching it in an effective way. A function is uniquely identified by its signature consisting of its name and arity (number of parameters). These requirements already define a minimal interface a function needs to implement as depicted by the summarized version of the current `Function` base class in figure 4.9.

The implementor is required to sanity check arguments and must not alter them. He/she also has total control over the semantics of the function. Some functions call other ones in order to aid their evaluation. Also, the main XPath execution engine maps operators to specific function calls to obtain a result. For these reasons, all of the functions have been implemented by using a static

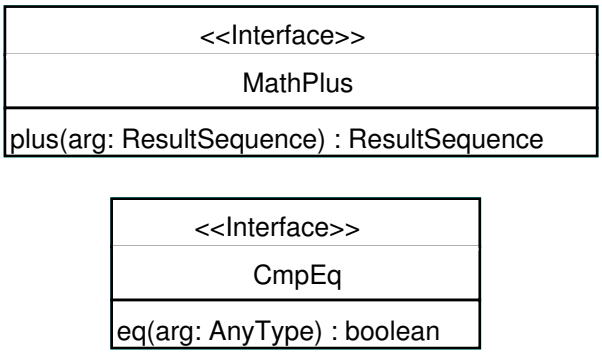


Figure 4.8: The MathPlus interface shows the original design where the implementor had total control over the semantics of the operator. The CmpEq interface resembles the new design where less power is given to the implementor but efficiency is gained by not having to (un)wrap arguments and results from sequences.

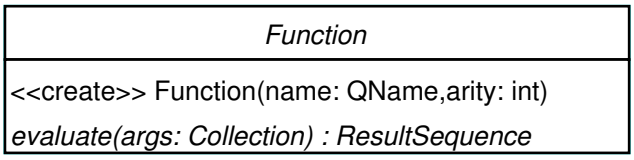


Figure 4.9: Simplified diagram of the current Function base class. A developer is only required to implement the evaluate method and provide the function's name and arity in the constructor.

evaluate method. The non-static version will call the static method directly to perform the computation. This does add extra overhead. On the other hand however, if a function calls another one, it may do so via the static method directly rather than going through the overhead of locating the function (the same is true for operators in the evaluator). Another interesting aspect of this is that unit testing may be done on the individual functions directly without having to create the necessary environment for execution.[1] An example of how a function may be implemented follows:

```
public class MyFunction extends Function {
    public MyFunction() {
        // provide function name and arity
        super(new QName("MyFunction"), 2);
    }
    public ResultSequence evaluate(Collection args)
                                    throws DynamicError {
        // call static version
        return MyFunction(args);
    }
    public static ResultSequence MyFunction(Collection args)
                                            throws DynamicError {
       ...
    }
}
```

In order to locate and dispatch functions, they are organized in libraries. It is important to note that the developer of new function does not need to know anything about how functions are located and dispatched. He/she only needs to implement the minimum required for a function to work fully.

4.3.1 Function Libraries

A function library contains a logical collection of functions. The obvious example is the "default" function library which contains all XPath function specified in [15]. Other examples include functions defined in [14], which not directly available to the user, and constructor functions. Function libraries are responsible for adding, locating and retrieving functions within them.

The two main function libraries implemented are the `FnFunctionLibrary` and the `ConstructorFL`. These map respectively to the default library and constructor functions available to the user. If a developer desires to make a new function, the addition of it is necessary in the appropriate function library. For example to add MyFunction the the default function library, a developer would add a line resembling this in `FnFunctionLibrary`'s constructor: `add_function(new MyFunction())`. Constructible types are added in a similar way.

[1] Testing is not performed in this manner for the reasons explained in section 6.1.

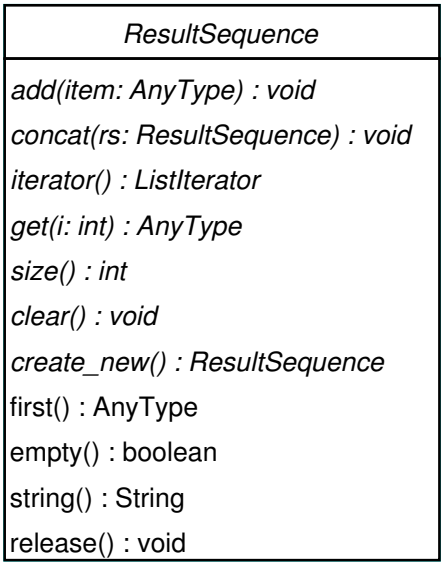


Figure 4.10: Interface of the ResultSequence class. The class plays a major role as everything in XPath is treated as a sequence of items.

4.4 xpath Package

This package links together all of PsychoPath's functionality and provides a portal to the user. The user interface is described in detail in chapter 5 and these sections will focus mainly on design decisions made throughout the development process rather than explaining how the user should interact with PsychoPath.

One of the main classes in this package is `ResultSequence` and is fully described in figure 4.10. In XPath everything is a sequence—arguments to functions, results of expressions, etc. This class is highly used in functions and the evaluator visitor, and in several cases, many temporary instances of it are created. For this reason, sequences may be constructed via a factory and later released to try and minimize memory utilization.

This package also contains two fundamental classes to XPath processing: the static and dynamic context. These are used by various visitors in order to retrieve and insert state and other information into them. Their functionality is equivalent to the one explained in the XPath specification.

Other functionality in this package includes the parsing of an XPath expression to an AST. This is currently achieved via JFlex [6] and CUP [4]. Implementing a new, and possible more efficient, parser in the future will be simple as long as the result of the parse operation is an AST as described earlier. DOM Loading is also accomplished by this package with a well defined interface. Currently Xerces [1] is used as the main DOM implementation.

The visitors which perform actual computations on an XPath will now be described.

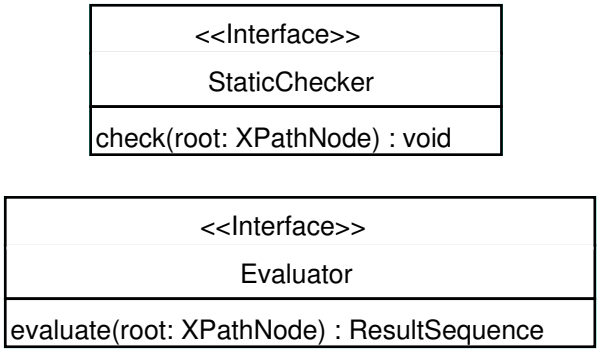


Figure 4.11: Interface for the two phases of execution. First the StaticCheck will perform all necessary static checks on the expression. Then, the Evaluator will execute the expression and return its result.

4.4.1 XPath Visitors

There are two main phases in evaluating an XPath expression: the static and the dynamic phase. The static phase consists in name checking, possibly static type checking, and normalization. The dynamic phase relates to the actual execution of the expression. Currently two interfaces have been designed which reflect these two phases and are shown in figure 4.11.

Both of these interfaces use the visitor pattern inside and are merely wrappers to hide implementation details to the user. Also, since they take a generic XPathNode as an argument, it should be possible to evaluate or check only a specific portion of the whole XPath expression. Both of the interfaces throw exceptions on errors.

The visitors currently implemented in PsychoPath are `StaticNameResolver` which will check names and expand all QNames, `Normalizer` which performs normalization and is not supported any longer and the `DefaultEvaluator` which will evaluate an XPath expression even if it is not normalized. No static type checking has been implemented, and the specification marks it as an option.

The main rational behind the visitors is that they should keep all their state internally. Also, they should not modify objects with which they are initialized. If they do, they must bring the object to its initial state at the end. In the current implementation this is not quite true. An example, although currently not supported, is the normalizer which will modify the AST it is provided. Each visitor may in turn decide its own rules regarding what should be result of a visit operation and how state should be maintained.

4.4.2 Exceptions and the Error Handling System

All exceptions in PsychoPath derive from a common base: `XPathException`. The only requirement for exceptions is to have a human readable error message. This facilitates error reporting and assures there is a specific reason to each error.

Furthermore, two other main classes of exceptions exist: `StaticError` and `DynamicError`. These relate to errors generated during a specific phase of eval-

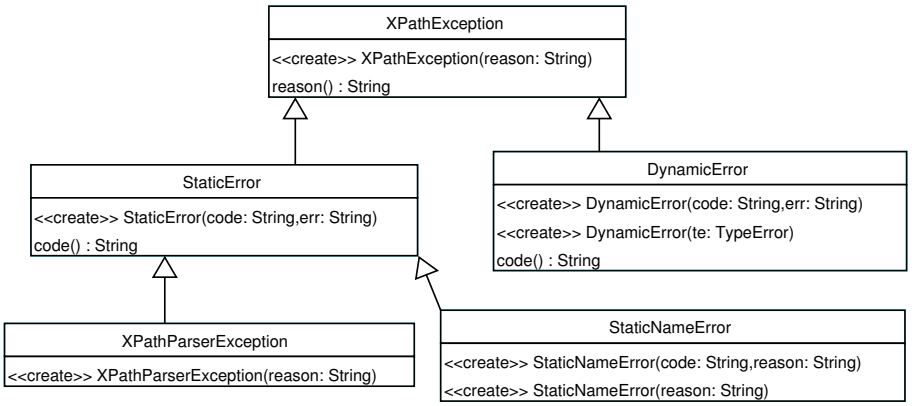


Figure 4.12: Part of the error hierarchy in PsychoPath.

uation. An additional requirement of providing an error code is enforced. These error codes match the ones in the XPath 2.0 specification. In the future, they may be used to check whether failures of a test occurred because of an expected reason or not.

Figure 4.12 summarizes the key portions of the current exception hierarchy in PsychoPath.

Chapter 5

User Interface

The meaning of a user interface is generally taken as a methodology for the user to interact with the program, usually via a graphical user interface (GUI) or a command line interface. However, as PsychoPath has been designed as a library, in this context, the user interface is defined as the public methods exposed within the library which may be invoked by the user in a defined manner so that a result may be obtained in a specific way. This user interface should be competently and extensively designed for two primary reasons:

1. The user interface is the only section of the library visible to the user under normal circumstances and should be designed in a logical grouped manner for ease of use.

2. In a library, the user interface usually follows and reflects the architectural design of the underlying implementation. Thus, a poorly designed user interface is indicative of a poor underlying structure.

5.1 High level Overview

Processing an XPath 2.0 expression can be decomposed into the following set of sequential and largely uncoupled operations:

1. Load the XML document.

2. Optionally validate the XML document.

3. Initialize static and dynamic context in respect to the document root.

4. Parse the XPath 2.0 expression.

5. Statically verify the XPath 2.0 expression.

6. Evaluate the XPath 2.0 expression in respect to the XML document.

This decomposability has allowed PsychoPath to be designed in a highly modular manner with almost no coupling between the packages. This has the potential for future extendibility and even for the re-implementation of existing packages with few modification needed in other packages.

We have chosen to use the external DOM package, `Xerces`, to handle the first two tasks, namely, loading the XML document and performing the optional XML Schema verification. PsychoPath makes use of Xerces in a largely package independent manner; only the XML Schema extraction and manipulation has been based off Xerces' implementation.
As Xerces is used as the default DOM package within Java 1.5, we believe this minimal coupling is acceptable, but if for some reason, the user desires a different DOM package to be used, a common wrapper can be easily implemented so as to not break functionality in the other packages.

The initialization of static and dynamic contexts is performed internally, entirely within the `DynamicContext` class. This subpackage fully initializes the relevant properties to the default values, adds Post Schema Validation Information (PSVI) according to the XML Schema of the document and additionally, handles the registration of data-type namespaces and function libraries.

We have used the external packages `JFlex` and `CUP` to generate a XPath 2.0 parser which PsychoPath uses to parse the XPath expression and represent it as an Abstract Syntax Tree (AST).
PsychoPath's usage of this parser is entirely package independent so a replacement, specifically designed for XPath 2.0, parser can be used instead with no break in functionality elsewhere.

Static verification of the AST represented XPath expression is performed internally using a Visitor pattern to traverse the AST. Other Visitors (e.g. an Optimizer visitor to simplify and optimize the XPath expression) can be implemented for additional functionality.

Finally, the evaluation of the XPath expression is also performed internally.

5.2 Example Usage

5.2.1 Loading the XML Document

The very first step in using PsychoPath is to load the relevant XML document. PsychoPath has been designed for use with the DOM package Xerces and although the XML Schema usage and manipulation is coupled to Xerces, a common wrapper can be easily implemented allowing use of any specification adhering external DOM package.

If using Xerces, this entire step is achieved by initially creating an InputStream from the XML document and initializing the Xerces DOM loader in the following manner:

```
InputStream is = new InputStream(XMLdocument);
DOMLoader domloader = new XercesLoader();
```

Now is the time to perform optional XML Schema checking to verify the structure and integrity of the XML document. This is done by setting a flag within the DOMLoader object:

```
domloader.set_validating(true);
```

Finally the XML document needs to be loaded its Document Object Model (DOM) root is stored:

```
Document doc = domloader.load(is);
```

5.2.2 Initializing static and dynamic contexts

The static context in PsychoPath is initialized automatically, so the user is required only to set the dynamic contexts in respect to the schema information of the document (may be null for schema-less documents). If Xerces was used to load the XML document, the schema must first be extracted from the DOM root of the XML document. This extraction and initialization is shown below:

```
ElementPSVI rootPSVI = (ElementPSVI)doc.getDocumentElement();
XSModel schema = rootPSVI.getSchemaInformation();
DynamicContext dc = new DefaultDynamicContext(schema, doc);
```

There are two other essential initializations within this step. The first is the registration of the namespaces of the XPath 2.0 predefined data-types (the 'xs' and 'xdt' namespaces) as shown below. Any user defined namespaces should also be registered at this point in the same manner.[1]

```
dc.add_namespace("xs","http://www.w3.org/2001/XMLSchema");
dc.add_namespace("xdt",
                 "http://www.w3.org/2004/10/xpath-datatypes");
```

The second essential initialization is the registration of the predefined XPath 2.0 functions. PsychoPath groups these functions into a standard library to simplify this registration step. Any user defined functions should also be registered at this point in a similar manner.

```
// The default fn library
dc.add_function_library(new FnFunctionLibrary());
// Constructor functions for Schema types
dc.add_function_library(new XSCtrLibrary());
// Constructor functions for XPath types
dc.add_function_library(new XDTCtrLibrary());
```

5.2.3 Parsing the XPath expression

The XPath 2.0 expression must now be parsed and represented as an Abstract Syntax Tree (AST) which is the internal format that PsychoPath uses. PsychoPath includes a parser created using JFlex and CUP that performs this step and its usage is as follows:

```
XPathParser xpp = new JFlexCupParser();
XPath path = xpp.parse(StringPath);
```

5.2.4 Static type checking

The XPath 2.0 expression obtained must be statically type checked to verify its structural validity, and check for possibly undefined names. PsychoPath uses a class implementing the Visitor pattern for traversing and checking the AST and is used as follows:

```
StaticChecker name_check = new StaticNameResolver(dc);
name_check.check(path);
```

[1] Automatic registration of namespaces defined in the document and its Schema has never been implemented, but should be in the future. All code is namespace aware, so the addition of a routine to extract namespace information should be very easy.

5.2.5 Evaluating the XPath expression

Finally, the time for evaluating the XPath 2.0 expression has arrived. This is shown below and the result is of the evaluation is stored in the ResultSequence:

```
Evaluator eval = new DefaultEvaluator(dc, doc);
ResultSequence rs = eval.evaluate(path);
```

5.2.6 Extracting the results

XPath 2.0 defines everything to be a sequence of items, including the arguments to expressions and the result of operations. Thus, the overall result of a XPath expression evaluation is also a sequence of items. PsychoPath uses the class `ResultSequence` as a Collection wrapper to store these sequences and therefore, the result of an evaluation is of this type also.

Extraction of certain or next items from the ResultSequence class is fully described in figure 4.10. However, all the items extracted will have the type of the base class `AnyType`. They will then need to be cast into the correct concrete type in order to be used.
Certain operations always return a particular type and using this knowledge, the extracted item may immediately be casted. An example is the "if expression" which always returns a boolean type and can safely be cast as such:

```
XSBoolean xsbool = (XSBoolean)(rs.first());
```

The actual result can now be extracted from this XSBoolean in the following manner:

```
boolean bool = xsbool.value();
```

Alternatively, a String representation of the value can also be extracted from the XSBoolean as shown below:

```
String sbool = xsbool.string_value();
```

However, if the expected return type is unknown or multiple types are possible, the types hierarchy depicted in figure 4.6 may be traversed in a breadth first manner making use of the Java `instanceof` operator to ascertain the actual type.
The first query would be to determine if the type is derived from `NodeType` or `AnySimpleType`:

```
AnyType at = rs.first();
if(at instanceof NodeType)
    checkNodeTypes(at);
else if(at instanceof AnySimpleType)
    checkSimpleTypes(at);
```

The result of this query would determine which subsequence queries should be performed, eventually reaching a leaf type satisfying these two criteria:

1. The type is not abstract
2. No other types are derived from this type

At this point, the actual type has been determined and can be safely casted to in order to extract the final result. For example, if the type has been progressively narrowed down to `NumericType`, the next query will determine the type to be a leaf node at which point, it can be safely casted:

```
if(at instanceof XSInteger) \\ leaf type
    XSInteger result = (XSInteger)at;
else if(at instanceof XSDecimal) \\ leaf type
    XSDecimal result = (XSDecimal)at;
...
```

5.3 User Interface Evaluation

PsychoPath has a well defined and logically assembled user interface. This arises due both due to the easily decomposable and sequential nature of the act of XPath 2.0 processing and also due to the extensive time we spent on iteratively designing and evaluating our system architecture.

The final cast of the result extracted from the ResultSequence is the weakest aspect of the user interface but is unavoidable due to Java's lack of support for templates in versions 1.4.x.

Due to the sequential nature of XPath 2.0 processing, Saxon has a similar user interface also decomposed into minimally uncoupled packages. Saxon also has the same flaw regarding casting of the result into a unknown type but also reduces the number of possible types by converting all numeric types into longs during evaluation. This decreases the number of `if(at instanceof type` constructs required by the user but at a trade-off of lesser adherence to the XPath 2.0 specification.

Chapter 6

Testing and Evaluation

This chapter will present the two most important aspects of our product. Extensive testing is what gives faith in a software project. Performance is what makes a product preferable against another one. A combination of both, is what makes the ideal software package.

Our project is too immature to be a winner in performance, but we believe it has something to offer when effective testing and design is the question. The first sections will discuss these two aspects. The last section will focus on the evaluation of the final product mentioning what has been accomplished and what still needs improvement.

6.1 Testing

Testing is perhaps the most important aspect in software development. In order to increase the confidence in the reliability of our product, two main goals have been established:

1. Test corner cases and non trivial aspects of each expression.
2. Cover most, if not all, the code produced.

Most of the testing has been performed after milestone 4. This is not ideal in extreme programming but it turned out to be a good solution in our case. There were also technical issues as to why thorough testing could not be performed effectively throughout the initial phase of the project as it will be explained in the next section.

6.1.1 Methodology of Testing

All of the testing in PsychoPath is performed via JUnit [2] and all the test code is in a separate package. The first question which had to be answered was: What should the granularity of the testing be? Should every single class be tested independently or should only the main interfaces be tested?

The first method, of testing each single class, had the advantage of giving certainty that everything is being tested and there is no untested code hiding somewhere. The main disadvantage however is that although we have the certainty that the single pieces work, we have less evidence that the project works

as a whole. Also the test cases would be highly coupled with the internals of the project (as they test all classes). If internal components change, many test cases would have to be updated although the results of the actual tests would be the same. This will create a major overhead and will possibly make the developers resistant to change. Having used an iterative development strategy, the probability of changing design and internal structure between milestones is high.

The second method seemed to be more suitable for our circumstances. The high level interfaces will change very infrequently. An example would be testing an XPath operator. No matter what the design is and how the code is structured, a user always expects to get a specific output on a certain input: 1 + 1 will always equal 2 no matter how classes are organized. Another advantage from this is that test cases from previous milestones may be re-used no matter how heavy the refactoring was. Indeed, no time is spent on re-designing the test suite itself, but rather, tests are simply added as appropriate. Using this approach also gives us confidence that the project works as a whole. However, there are two main disadvantages with this testing method.

One of them is that there is no longer certainty that all code has been tested. The execution flow of a particular input is highly complex and being able to predict it is error prone. Although a tester might think he/she is testing something specific, it may be that the result is obtained via other means (optimizations may be an example). This problem has been tackled as later explained by using JProbe [9].

The other disadvantage is that in order to test the high level interfaces, they must work or at least be implemented. To complete the project in an iterative fashion, a bottom up approach had to be used. In this case, the high level interfaces would be the last to be completed, which explains why the major bulk of testing was left to the end. Although the pieces of the puzzle were available half way through the project, they were put together only at the end, finally enabling more complex test cases to be performed. The only "solution" to this, was testing those few paths present in the puzzle as the project was being developed. The main purpose for these tests was to make sure that refactoring between milestones was carried out properly (i.e. all tests pass after changing the code).

6.1.2 PsychoPath Test Suite

Up to the first two milestones, a lot of code was present although it did nothing useful—It would load an XML document, parse an XPath 2.0 expression and sanity check it. However, an important observation regarding testing was made. The best way to perform test cases was to supply an input and an expected answer. The input would be fed into the high level interface which would perform the needed computation, and the answer would be compared to the expected answer. If the computation was expected to fail, the expected answer would represent a failure. Currently there is a limitation when testing failures as there is no way to check if the failure occurred due to a specific reason or not. Each test class would contain a method similar to this:

```
private void check_input(String input, String answer);
```

The testers of the software would then add their tests by calling the `check_input` method. For example:

```
public void testInputs() {
    check_input("1+1","2");
    // null represents an expected failure
    check_input("1/0",null);
    ...
}
```

This turned out to be a good strategy as the number of test cases was low during the initial phase of development. At the end of milestone 4 however, when the volume of testing was becoming high, a new strategy had to be devised as the current technique was becoming tedious. Interestingly, the conceptual model of testing was correct, the real limitation to scalability was where the test cases were being put. Inserting them into the source files had several disadvantages. First, the test source would have to be recompiled. Eventually, the source file would become unreadable with many test cases. There are also some impracticalities such as using \n for newlines, escaping quotes and so on.

The solution was very simple: read the inputs and answer from an external test file. This made the test suite very strong and extensible mainly because there was no need to modify the source of the test programs anymore, thus reducing the possibility of introducing new bugs in the test code itself. A single test program may load test cases from multiple files to logically split up tests making the management and organization easier. The format of the text file is very simple and illustrated by the following example:

```
% this is a comment
%
% **** is treated as a delimiter between input/answer
% ****xmlfile will load a specific XML.
% The file will be used for tests which follow.
%
% If FAIL is put as an answer,
% the expression is expected to fail.
****xmlfile test_data/test_schema.xml
% this is the input
1 + 1
****
% this is the answer
1) xs:integer: 2

****
1 / 0
****
FAIL
****
% put more test cases here...
```

In order for test programs to support this solution, they must implement the following interface:

```
public void check_input(String input, String expected,
                        String error);
public void load_xml(String fname);
```

`check_input` is as described previously. The only difference is that it takes an error string which, if not null, will be displayed if a test fails. It is normally used to indicate on which line of the test cases file the fail occurred. The `load_xml` method maps the `****xmlfile` command for which the implementation is required to use that particular XML file for all test cases which follow.

Following is a brief description of which components are tested and how.

Milestone 1 Testing

Upon completion of this milestone, two main components arose:

1. XPath 2.0 expression parser.
2. DOM loader.

In order to test the first item, a visitor which prints the AST structure has been implemented. The test cases would contain an XPath expression and the expected AST structure. These would then be compared to assert whether the test passed or not. A pseudo-example[1] is:

```
1
****
XPATH:[
 XPATHXPR:[
  FILTER:[

   PRIMARY:[
    INTLIT:[ 1 ]:INTLIT
   ]:PRIMARY

   PREDICATES:[
   ]:PREDICATES
  ]:FILTER
 ]:XPATHXPR
]:XPATH
```

DOM loading was tested in the same way. The expected structure of an XML file was compared with the structure returned by the loader. A DOM printer which serves a similar purpose as the XPath printer was developed.

Milestone 2 Testing

This milestone implemented these aspects:

1. Expression static checking.
2. XPath expression normalization.

[1] To make it a full example, the expected answer should all be in a single line

The first item was tested mainly by using `FAIL` as an expected answer. This involved tests such as referencing unexisting variables.

The next component was tested similarly to the expression parser. The expression was first parsed, then normalized, and its new AST structure was compared with the expected one by using the XPath printer.

Milestone 3 Testing

Upon completion of this milestone, "real" aspects of XPath may be tested. All of the core XPath had been implemented except for functions. Each result from an XPath expression had the capability of being converted to a string. This fact was mainly used in testing. A tester would input a valid XPath expression and its expected outcome in the same form as results are converted to strings. The way results are converted to strings is very simple and may be obtained by using the SimpleXPath tool provided in the examples. It is important that results obtained via this tool are not blindly put in test cases—their correctness must by manually checked first. This is an example of how a resulting sequence is converted to a string:

```
(1,2)
****
1) xs:integer: 1
2) xs:integer: 2

****
```

Be aware of the newline which is always inserted at the end.

Milestone 4 Testing

The project is now complete, so anything may be tested. And this is when the testers went full throttle. At this point however, XPath functions had to be tested mainly.

6.1.3 Accuracy and Complexity of tests

One of the main goals was to test corner cases. If a computation returns a correct result under extreme conditions, the like hood of it succeeding in normal circumstances is relatively high. The opposite is not usually true. A simple example would be testing division by 0 instead of division by 1.

A more interesting example was a test case written the 16^{th} of January 2005. Life was going on beautifully when one day, specifically on midnight of the 28^{th} of the same month, the test started failing after having succeed up to then. Even two minutes to midnight[2] the test was passing and no code has been modified since then. Something evil was in the air. The test was (using the old interface):

```
check_input("xs:date(\"1983-02-29\")", null);
```

It depicted the fact that 1983 was not a leap year, thus the 29^{th} of February did not exist that year. The test was complaining about expecting a failure but actually getting a result of the date 1983-03-29.

[2] Tribute to Iron Maiden.

At 01:46 in the morning of the same day, or more precisely the 29^{th}, the bug was isolated and fixed with the following CVS log:[3]

> fixed the millinium bug, or rather, the month bug
>
> when i create a calendar, by default its initted to the current day... then i set in this order: year, month, day
>
> suppose it is 29/01/2005 and we want to set 29/02/1983 [and expect a fail]
>
> when i set year we get
>
> 29/01/1983
>
> when i set month we get
>
> 29/02/1983 [which doesn't exist, and java fixed it to 01/03/1983... lenient calendar i fink]
>
> when i set day we get
>
> 29/03/1983
>
> nice bug eh ?
>
> owned it =D

More appropriately, the bug may be described as follows. The order in which fields in a calendar are set were:

1. set the year.
2. set the month.
3. set the day.

By default, a calendar is created with the current date and time, thus 2005-01-29 in this case. Next the test case wanted to set the date of 1983-02-29. The implementation would first set the year, resulting in a date of 1983-01-29. The bug kicks in when the implementation tries to set the month. By setting the month to 02, the new date becomes first 1983-02-29, but as the date does not exist, it is converted to 1983-03-01. This is because we are referencing the 29^{th} day of a 28 day month which is "equivalent" to referencing the first day of the next month in a lenient calendar. Finally, the implementation would set the day to 29, resulting in the date 1983-03-29. The test however expected a failure and correctly reported the bug. It did not report failures in the past as days ≤ 28 existed in February 1983—our code was a time bomb. The fix currently is to reset the day and month to 1 before setting them.

A curious test which deals with the complexity of XPath expressions in a more philosophical manner is one which solves the Traveling Salesman Problem. XPath is expressive enough to solve an N node traveling salesman problem, where N is fixed to that expression.[4] Maybe a more general solution exists, but we have not dwelt on it as it was not part of the primary goals for the project. A proof of concept test case is included and seems to work.

[3]The author of the patch did not spend 2 hours locating the bug, but rather was informed about it around 1 something.

[4]More generally for any graph of x nodes where $x \leq n$

6.1.4 Test Coverage

The second goal of our testing scheme was to cover as much code possible with our cases. The oracle turned out to be the jpcoverage tool included in the JProbe suite. The project is divided into the following packages:

- xpath. The main package and "glue" between components.
- ast. The Abstract Syntax Tree nodes which represent XPath expressions.
- types. The types supported in XPath.
- function. The functions supported in XPath.

In order to test all functionality, all methods in these packages needed to be called. We did not concentrate on whether every single line of code was being evaluated such as examining if all possible branches in an `if, else` fragments have been followed. The bulk of the functionality is in the types, function and ast packages. The xpath package contains some debugging utility functions, error types and other less relevant classes.

The jpcoverage tool was used to run the test suite and obtained results regarding how many methods were not called in each package and more in depth detail about them if required. Using this tool it was possible to write enough test cases to bring the method miss percentage close to 0. In some circumstances it was useful to detect unreachable or old methods which have not been eliminated. In other cases, it was useful to rethink the design to eliminate some redundant methods.

An example of this was redesigning the type hierarchy. Previously, all atomic types such as strings, integers, etc. could be constructed. A problem arose with a special atomic type called "untyped atomic" which could not be constructed. It must have a constructor method as required by the atomic type base class, but it would always return an error. Clearly this was a miss in the tests as there was no possible way of constructing such a type.

To eliminate the miss, a new base class called "constructor types" which derives from atomic type was created. All constructible atomic types would derive from this new class, whereas the untyped atomic would derive directly from the atomic type class. Figure 6.1 depicts the change in the type hierarchy.

Up to milestone 4, the average method miss percentage across the packages was 53.875%. Currently it is 4.375% where all packages, except for xpath, are at 0%. The untested code in the xpath package mainly has to do with testing errors such as parse errors, dynamic errors, etc. Table 6.1 summarizes the test case coverage up to milestone 4 and the current coverage.

6.2 Performance

The main requirement for the project was certainly not performance. Most effort was put into having a clean design, being conformant to the specification, implement as much of it as possible and obtaining correct results.

In contrast however, performance is definitely the most important practical requirement for a project of this nature. What makes an XPath processor better

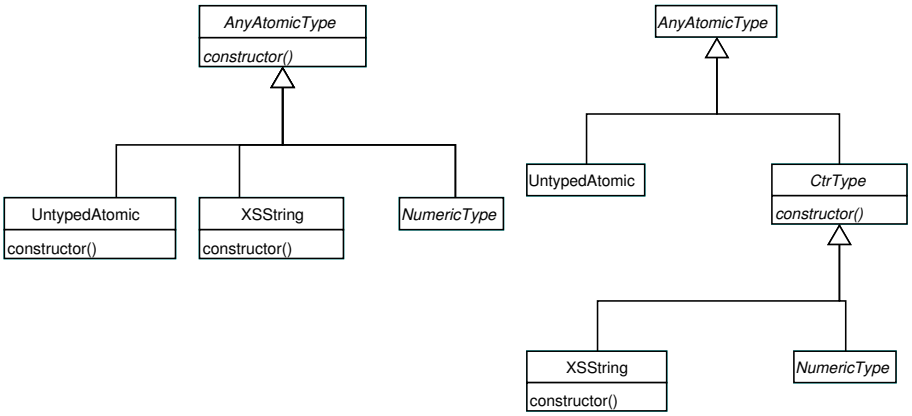


Figure 6.1: The diagram on the left shows the original type hierarchy. The constructor method of `UntypedAtomic` would raise an exception if invoked since such a type, by definition, may not be created by the user. The current revised hierarchy is displayed on the diagram on the right.

Package	Milestone 4 method miss %	Current method miss %
xpath	53.3	17.5
ast	35.2	0
types	72.9	0
function	54.1	0

Table 6.1: Comparison between the coverage of test cases from milestone 4 to current date. Ideally the method miss percentage should drop to 0.

than another is its speed and not only its correctness. A client will most likely prefer a "black-box" solution where its design and implementation are somewhat obscure but on the other hand its performance is highly efficient. Having pretty code which does the same thing but slower is probably not much of a win for the end user.

Optimization per se is a whole milestone on its own which in the case of this project has not been accomplished mainly due to time limitations. Premature optimization is a well known practice which must be avoided and from which we carefully stayed away. However, a curiosity of how our solution performed arose, regardless of the matter that no optimizations has been attempted.

Fortunately, another open source implementation of XPath 2.0 exists and we used it in order to compare our solution against it. The other processor is saxon [5] written by Michael Kay. The version used in the comparison is `saxonb8-1-1`. It is important to notice that Michael Kay is one of the editors of the XPath 2.0 specification, thus being highly competent in the field. It may imply that his product is a very good implementation of the working standard, and as it will be shown later, he does care about performance. This simply means that we are starting a fight against a guru, and we do not expect to win!

6.2.1 Range Expression Case Study

As there was no planned milestone dedicated to performance optimizations, we decided to analyze a single XPath expression in detail and understand where the bottlenecks occur and why. An important question to be answered was whether the actual design was the bottleneck, or whether parts of the implementation may be tuned incrementally increasing the speed of the most critical components and functionality.

The whole curiosity about performance arose after our client implicitly suggested that indeed he was interested in the answer "Who is faster?", referring to PsychoPath vs. Saxon. The easiest XPath expression which reveals metrics about evaluation time is:

```
(10 to 20000)[19909]
```

This expression was used throughout the whole case study of performance.[5] Unfortunately it is not the most used and wanted XPath expression so there is a question whether we are assessing important aspects of performance. However, the results were interesting and some of the discovered optimizations may be applied to other types of expressions too. Also, the study was mainly proof of concept, and we had to start somewhere. To conduct the study two equivalent benchmark tools have been written, one for PsychoPath and one for Saxon, which evaluate an XPath expression and return elapsed time metrics in milliseconds.

The first crude results yielded by PsychoPath (using the the SimpleXPath sample application) at the end of milestone 4 were:

```
Task                    Time (ms)       Cumulative time (ms)
====                    ========        ====================
XML Loading             953             953
```

[5]The expression still has its original typo—The intended predicate actually was 19990

```
Dynamic Context init    113             1213
XPath parse             260             1326
XPath static check      34              1360
XPath evaluation        342             1702
```

The results from Saxon were:

```
Load XML                477             477
StaticContext           0               477
Compile XPath           108             585
XPath Evaluation        50              635
```

No surprise—Saxon wins. The surprise came when a 0 was added to 20000...

6.2.2 Understanding the metrics

Overall, the evaluation of an XPath expression consists of two main phases:

1. Loading of the XML document on which the expression will be applied.
2. Evaluation of the expression.

Furthermore, the evaluation phase may be split into the following steps:

1. Parsing of the XPath expression.
2. Static checks on expression.
3. Normalization of the expression.
4. Dynamic evaluation of the expression.

Both Saxon and PsychoPath have a common way of loading an XML document by relying on an external DOM package. There is nothing that may be done to speed up the XML loading phase other that write a DOM parser. The reason why PsychoPath loads and XML file much slower is because it does XML Schema checking which Saxon does not.

An intermediary step between loading the XML document and evaluating the XPath is initializing the static and dynamic context. This involves setting default values for certain properties and most importantly making sure XPath functions may be called. Saxon takes no time to finish this task, as it literally does nothing. Saxon initializes its functions statically and this time is not visible from the metrics.[6] On the other hand, PsychoPath registers all of its functions dynamically which explains the poor efficiency in creating a dynamic context. Arguably however the design of PsychoPath in this respect is more clean. This fragment of code illustrates how its functions are registered:

```
add_function(new FnStringToCodepoints());
add_function(new FnCompare());
add_function(new FnConcat());
```

And here is the Saxon version:

[6]The time between invoking the Java interpreter and when main is actually reached should be measured. However, the measure is not very reliable

```
static {
    Entry e;
    ...
    e = register("concat", Concat.class, 0, 2,
                 Integer.MAX_VALUE, Type.STRING_TYPE,
                 StaticProperty.EXACTLY_ONE);

        arg(e, 0, Type.ANY_ATOMIC_TYPE,
            StaticProperty.ALLOWS_ZERO_OR_ONE);

    e = register("codepoints-to-string", Unicode.class,
                 Unicode.FROM_CODEPOINTS, 1, 1,
                 Type.STRING_TYPE,
                 StaticProperty.EXACTLY_ONE);

        arg(e, 0, Type.INTEGER_TYPE,
            StaticProperty.ALLOWS_ZERO_OR_MORE);
```

The next steps are the more interesting ones. Saxon does the parsing and static checking in one pass. It then evaluates the expression to obtain the final result.

Our implementation does the parsing using JFlex and CUP therefore relying on their performance. Saxon parses expressions in its own way which is much more efficient. The downside is that in order to modify the parsing engine, you must learn about its internal mechanisms. PsychoPath has a separate grammar thus isolating the BNF syntax of the language from the actual code enabling modifications to be done with more ease.

Static checks are done sequentially after the parsing via an appropriate visitor. Ideally, a Normalizer visitor should follow a static check in order to comply fully with the specification. Currently our implementation does not do so. Again there is a trade-off between design choices and performance. Visitors are great when you want to have multiple version of them. For example, it is very easy to plug a Normalizer visitor, possibly an Optimizer visitor in the XPath evaluation pipeline. Also, it is easy to have different versions of the same visitor, such as an OptimizedEvaluator instead of the DefaultEvaluator and enable to user to choose between implementations. However, you pay the overhead of traversing the AST several times and the cost of double-dispatch while visiting. On the other hand, changing the methodology of evaluation in Saxon would require modifying the classes of each expression instead of only locally modifying the visitor itself as it is the case with PsychoPath. In order to compare the performances, saxon's `Compile XPath` time may be compared to the sum of PsychoPath's `XPath parse` and `XPath static check` time.

Finally the most important metric is the actual evaluation time. This is the time which has to be brought down. The other metrics are either constant or we have no control over as we rely on external packages. The only exception is the static checking time which may be reduced and virtually eliminated by implementing a visitor which will do static checking and evaluation in a single pass. After installing JProbe [9], and more specifically jpprofiler, we have all the necessary tools to begin our quest. The battle is our 342 milliseconds versus Saxon's 50 milliseconds. However as suggested earlier, the real problem is how

these metrics increase as the input increases.

6.2.3 Understanding the expression

The expression `(10 to 20000)[19909]` means, in human terms, "return the 19909^{th} integer from the integers ranging from 10 to 20000". In XPath 2.0 terms it perhaps means "Return the sequence obtained from the range expression 10 to 20000 filtered by the predicate 19909. The filter 19909 implicitly means that the predicate is true if and only if the context position is equal to 19909."

Fully following the specification, the following pseudo-code may solve the problem (and this is what our implementation does):

```
// initialize range sequence
range[]
for(j = 10; j <= 20000; j++) {
    item = make_integer(j)
    range.add(item)
}

// evaluate predicate for each item in the sequence
result[];
for(pos = 0; pos < range.length; pos++) {
    // "evaluate" filter
    filter = make_integer(19909);
    item = range[pos];

    // check if predicate is true
    if(equals(filter, pos))
        result.add(item);

}
return result;
```

The first part of the problem consists in creating $20000 - 10 + 1 = 19991$ integers. The next part is running each one of these integers through the filter, and returning only the ones for which the predicate is true. This means we need to evaluate the filter 19991 times. Although the filter in this case is a constant value of 19991, in the general case we do not know this, so we need to re-evaluate it each time and in our implementation it is a recursive call with double dispatch! When evaluating the predicate, we need to perform 19991 comparisons in order to check whether the current integer being tested is equal to the result of the filter (19909).

What makes things worse, is that everything in XPath is a sequence. Thus, although the filter will always return a constant integer, each time a new sequence containing this single element needs to be created. A further complication is that equals is a function call and it expects sequences as arguments and it returns a sequence. Thus the context position needs to be wrapped in a sequence, and the result of the comparison must be unwrapped from the sequence returned.

Operation	Calls	Percentage of total method time
Equality	19 991	7.5
Sequence constructor	179 930	7.5
ArrayList constructor	219 922	4.9
Convert function arguments	19 991	4.9
Atomization	39 984	3.0

Table 6.2: Profiler results based on the implementation as of milestone 4

6.2.4 Profiler results

After understanding the problem and the way the solution is currently implemented the results to expect are bad. Table 6.2 summarizes the relevant profiler results. As expected, we are doing 19991 comparisons and that is the bulk of the computation. We also create many sequences which are "necessary" to pass and return arguments from function calls and between different sections of evaluation within the visitor. Sequences are implemented using Java's ArrayList and as a consequence much time is spent in constructing them.

The function call overhead is very large. Although the operation is a simple innocuous equality between two integers, the general equality algorithm of XPath is quite complex. For example, the arguments of the equality function, and most function calls in general, need to be atomized which adds the implicit overhead of calling the data function. The data function in turn will usually return a new sequence with the atomized arguments. Memory allocation will be an issue.

In Java memory allocation is a bottleneck. The programmer has no idea when the garbage collection will be invoked and in such cases performance rapidly degrade. The summary of memory usage is presented in figure 6.2. There were 38 garbage collections during the whole run, and the red dots indicate their frequencies.

The spikes are definitely a sign of poor memory usage which is mainly due to the fact that many sequence objects are being created only for temporary use. The first main increase in memory usage is due to the allocation of the 19991 integers from the range expression. The sawtooth which follows describes the behavior of the sequences being allocated and later freed by the garbage collector as they become unreferenced.

6.2.5 First optimization pass

There are two main issues which needed to be dealt with:

1. Memory management of sequences
2. Function call overhead

The solution proposed for the first problem was using a factory in order to allocate new sequences. Sequences no longer needed may be given back to the factory for future re-use. This will drastically cut down allocation time for temporary sequences as these will be retrieved from a pool of unused sequences. The cons is that the code will look uglier and some care must be taken. Instead of allocating a new sequence using Java's `new` operator, a new sequence must

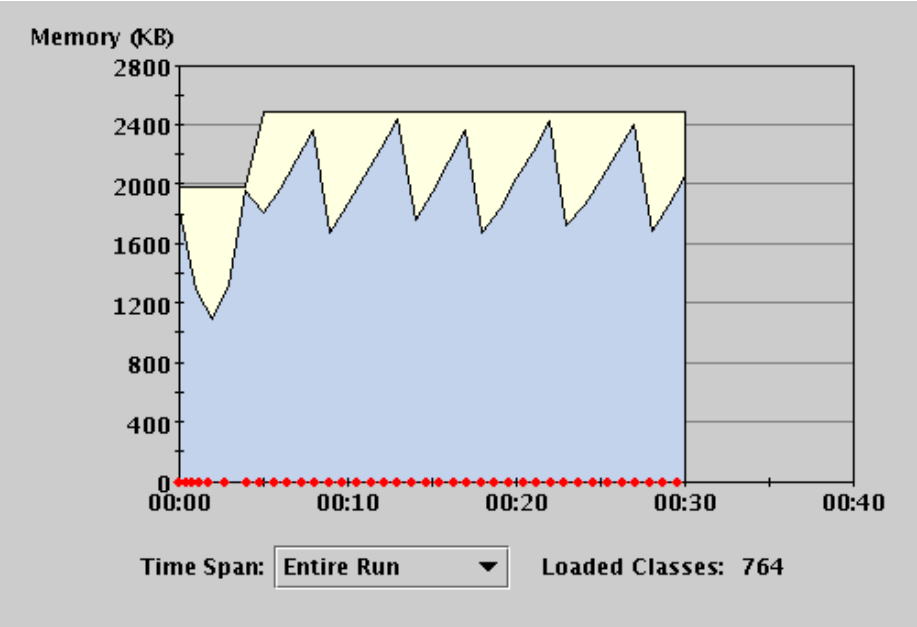


Figure 6.2: Memory usage of implementation as of milestone 4. Excessive temporary sequence objects are created and unreferenced resulting in memory spikes and numerous garbage collections (shown by the red dots).

Operation	Calls	Percentage of total method time
Equality	19 991	1.9
Sequence constructor	51	0.0
ArrayList constructor	62	0.0
Convert function arguments	-	-
Atomization	2	0.0

Table 6.3: Profiler results after the introduction of a sequence factory and elimination of some of the function overhead

be requested from the factory. What needs care however is to release sequences only when they really are unreferenced.

The second problem was tackled by eliminating some flexibility. The major change was making operators such as equality take two single arguments instead of a sequence, which will always have a single argument. Also, the return value will be a primitive boolean instead of a sequence. Doing so there is no flexibility for the future in case XPath developers decide that a single value may be compared against a sequence, or that the result may be a sequence. A new version of the atomization function was added in which the argument being passed is atomized itself, instead of returning a copy of the atomized version of the original argument.

The results obtained after these optimizations are summarized in table 6.3. Only 51 sequences are actually constructed because of object reuse. The factory itself creates a pool of 50 sequences. As the equality operator now takes single

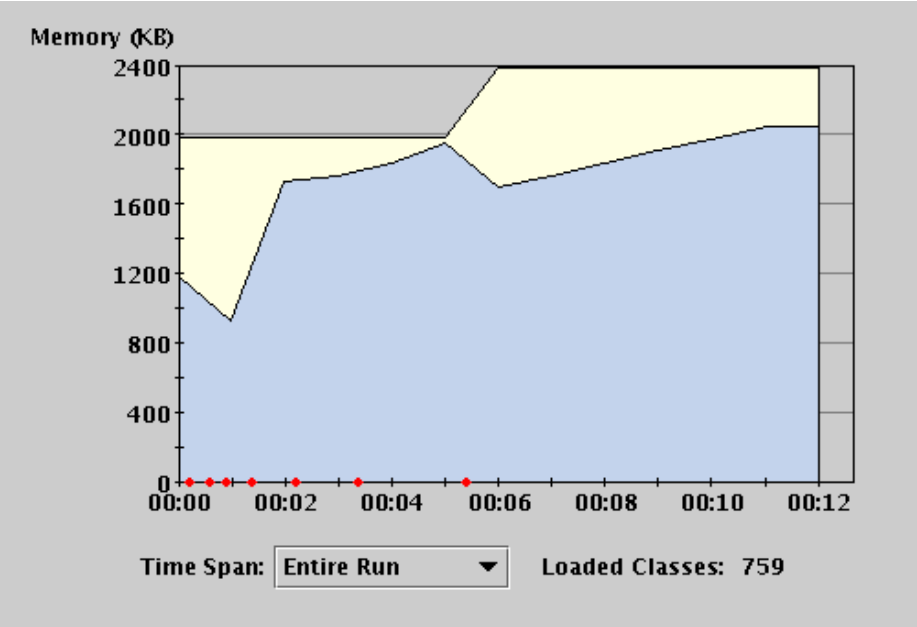


Figure 6.3: Memory usage after implementation if sequence factory. Now there is only one initial spike depicting the creation of the range integers. After that, sequences are re-used, thus memory allocation stays rather constant. Garbage collections were reduced drastically.

arguments instead of sequences, there is no need for conversions and the implied atomization. The memory use is much more stable as shown in figure 6.3. Only 7 garbage collections have been invoked. The initial spike is still present as that is when the range integers are allocated.

The final result after this optimization was an evaluation time of 128 milliseconds compared to the initial 342. This is still far from being as good as Saxon, but we decided it was good enough. For curiosity we decided adding a 0 to the magnitude of the range and a 9 to the filter and our evaluation completed in 694 milliseconds. We tried it on Saxon just to see how bad it was, and the result was 79 milliseconds. Our ideas changed on being "good enough"—Saxon must have been doing something smart.

6.2.6 Second optimization pass

The first hint that Saxon knew about this expression was using a predicate filter of 2. No matter how big the input was, it would return very fast. Indeed Saxon knows about this expression and performs a special optimization. If the filter expression is a constant numerical value, there is no need to re-evaluate it each time as it is constant. It would be equivalent to removing the `make_integer` call in the pseudo code presented earlier. This explains why evaluation time grows relatively slowly compared to input size.

Saxon also notes that this predicate will be true, if ever, only once. The condition being that the context position must equal that constant numeric

value. Thus Saxon will stop evaluating the predicate once a true condition is found. This is similar to adding a `return result` right after the addition of the context item. However, this does not fully explain why Saxon returns right away when fed with a filter of 2.

The overhead of generating n integers is relatively high. Thus, even if the filter is 2, n integers would still have to be generated. Saxon however works differently, it generates an iterator instead, and items are generated "on demand". As a consequence, a filter of 2 will generate only the first 2 integers of the range expression.

We adopted a similar strategy to inspect performance benefits. A special sequence was created for range expressions which exploits "lazy evaluation". Like Saxon, integers are created only on request. Also, a check was put in order to determine if the filter was a constant numeric literal in order to skip the re-evaluation of the filter each time.

To put the icing on the cake, we went a step forward. Suppose the filter is the numeric constant x and the range magnitude is n. Instead of generating the first x integers and throwing them away like Saxon, we simply obtain the x^{th} item directly from the sequence. After all, that is what the expression means intuitively. This resulted in a constant time algorithm which solves the expression in 47 milliseconds, slightly better than Saxon. The real win is, that it is constant time no matter the range of the input. By using lazy evaluation the initial memory spike, which would create the initial integers in the sequence, disappears as shown in figure 6.4. Method times are not reported as they are all under 1%.

6.2.7 Future optimizations

The expression analyzed in the case study is probably the most useless XPath expression. However each expression may be optimized by analyzing the problem closely. Saxon has many of these specific optimizations. Not always will clean design reflect fast execution as it was shown previously, tradeoffs between design and performance exist.

An future attempt to improve performance would be using factories for primitive types as well. Some expression create objects heavily and by enabling object re-use the garbage collector will run less frequently.

Another improvement, at the cost of design, may be including an evaluate method in the AST. There will be no need for a visitor to perform evaluation and the overhead of double dispatch will be gone. It is possible to leave the visitor interface too and provide both solutions. We tried evaluating without a visitor the case study expression and it returned in 33 milliseconds instead of 47.

The most important XPath expressions should be analyzed carefully and optimized according to their most frequent use. Performing lazy evaluation on other expressions will be another major speed gain.

6.3 Evaluation

The first sections of PsychoPath's evaluation will concern how complete and conformant it is according to the XPath specification. Next, it will be compared

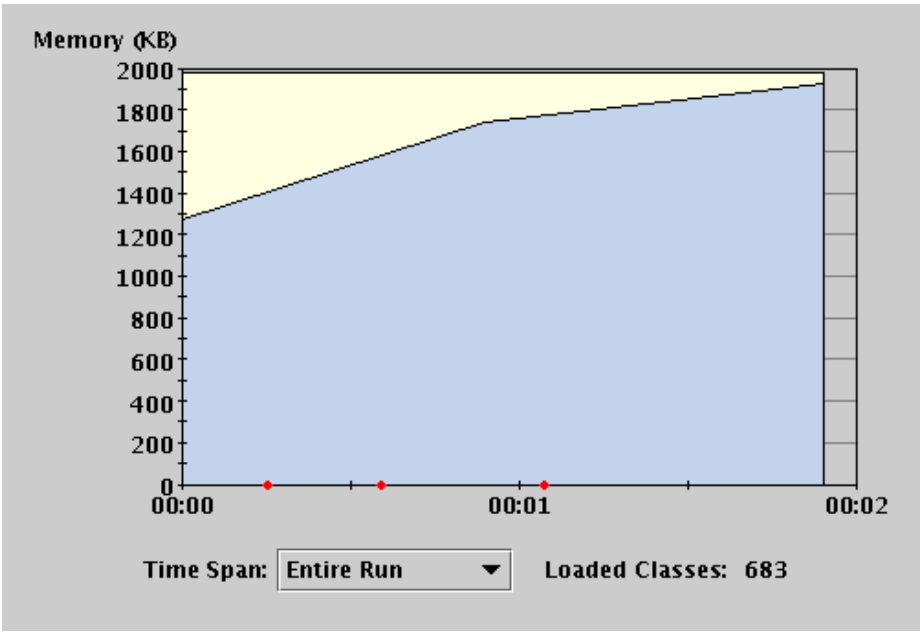


Figure 6.4: Memory usage of current implementation. The use of lazy evaluation in the range sequence eliminates the initial overhead of creating all the integers in the range. Memory use is almost constant as sequences are recycled. Only three garbage collections occurred.

to existing solutions to see where it stands in respect to its competitors.

6.3.1 Implementation Completeness

A substantial amount of the specification was implemented with extensive design for the various components. The PsychoPath product competently captures full functionality with its DOM loader, XPath expression parser, static checker and evaluator. The architectural relationships between these components formulate the skeleton of our XPath 2.0 processor; which is why their implementation was prioritized in the requirements specification.

On evaluation we find that these components meet the mandatory expectations of stability, else various dynamic/static error reports would have been evident during the main testing phase. Aside from these required intrinsic components, the more useful areas of interest were the behavior of functions, operators and expressions when used to manipulate data embedded in XML documents.

The major deficiency in the core XPath language is support for numeric type promotion. The extension needed in the implementation to accommodate this should not be too demanding and problematic. Other areas not covered are support for collations in string comparisons and backward compatibility with XPath 1.0.

The type system is not fully implemented although the most important types are present. There are some issues with date/time types. For example, all times are converted to UTC, thus making any subsequent time extraction function return the UTC time and not, perhaps, the original time specified in some other time-zone. All the other types seem to work properly.

Functions which have been implemented seem to work as expected. The only limitation in the implementation is that no function overloading is supported. Adding this feature should not be too problematic. In the case of overloaded functions, the function with most arguments has usually been implemented in order not to restrict expressiveness, as many times the version with less arguments will supply a default argument to the most specific version.

6.3.2 Implementation Conformance

From what was implemented the conformance to specification is fairly extensive considering the amount of time delegated to tackle this project. Referring to appendix B we can see the extent of completeness of the grammar productions/expressions, functions, operators and data types of XML schema for the XPath 2.0 processor.

After the code for the product had been finished extensive testing was carried out for each of the grammar productions, functions, operators and data types. The amount of time spent on testing was quite lengthy in relation to the amount of code formulated. On testing these components many errors and irregularities were unearthed and documented. This gave a clearer picture of how much coverage we obtained of the specification. Some functional aspects were simply not administered in the code for reasons rooting to the time available. Other aspects were fully functioning and in cases better deployed than the comparison processor Saxon.

All XPath grammar production rules were implemented and nearly all conformed to the specification stipulated by the W3C. Cases where our implemen-

tation was flawless was for [04][7] ForExpr—our implementation fully captured specification requirements and in fact was implemented more optimally than Saxon for the same expression.

The XPath Functions are generally implemented fairly competently and conform well to the specification but not to the same completeness metric as for grammar production rules. Instances where ideal conformance to specification is exhibited is exemplified through our implementation of Functions and Operators on Boolean Values (see Appendix B.2). More importantly on evaluation through testing some functions were found to conform irregularly to the specification. Functions and Operators on Durations, Dates and Times, particularly Component Extraction Functions[8] do not work perfectly as illustrated in the following example:

```
fn:year-from-dateTime(xs:dateTime("1999-12-31T21:30:00-05:00"))
```

outputs:

```
1) xs:integer: 2000
```

Our output for this instance is incorrect. It should be xs:integer: 1999.

Our implementation deviates from conformance to the schema specification here. The argument within the quotes is a localized value in time zone -05:00 or -PT5H. Therefore when we extract a time component (year in this case) from the lexical representation input we store it as a localized value and only extract the component from the localized value. The localized valued stored in the data model for the above would be represented by the tuple (1999-12-31T21:30:00, -PT5H). It is from this kind of localized representation that we extract a component—so the year would be 1999.

What our implementation does is to add 05:00 to the time (proceeding T) outputting the extraction from the normalized time std i.e. UTC or Z (refer to specification) This is inconsistent with the guidelines of the specification as this sort of extraction is not done in the proper manner. This is a major flaw in our implementation as it misinterprets the intentions of the specification.

To legally output a value as above (2000), we would have to use this syntax:

```
fn:(year-from-dateTime(
     adjust-dateTime-to-timezone(
      xs:dateTime("1999-12-31T19:20:00-05:00"),
      xdt:dayTimeDuration("PT0H")))
```

Which takes the localized dateTime for zone -05:00 and wants to extract the year component adjusted to the normalized time zone (UTC) stipulated by the dayTimeDuration 'PT0H' which basically means time zone 0 or GMT. In the data model this lexical representation would be stored as a normalized value like the tuple (1999-12-31T24:20:00Z , -PT5H), which would basically increment to the following year as 05:00 hours have been added.

The XPath Operators have almost 100% conformance with specification except for sections 12.1.1-2 Comparisons of base64Binary and hexBinary Values

[7] In this section, grammar production rule numbers will be expressed using this notation. Not to be confused with a citation to a specific reference.

[8] refer to [15] for specification details.

and section 13.1.1 Operators on NOTATION which are not implemented. Section 6.2.2-6 Operators on Numeric Values exhibit partial conformance to the specification except for numeric type promotion (see Appendix B.3).

Conformance of XPath Types to specification is rather poorly implemented in comparison with grammar productions, functions and operators, with a lot of types not supported by PsychoPath. For those types which are implemented the majority show full conformance to specification with the remainder having a few features absent from implementation such as no support for numeric type promotion and conversion (refer to Appendix B.4).

On reflection, had more time been given for this project the specification would have been fully implemented and improved further for conformance, after many iterations of extensive testing. Especially for Functions and Operators on Durations, Dates and Times which, the client deemed somewhat important. But since we were made aware of this toward the end of the project little time was left to address this issue as it was not prioritized with importance in the contract. To rectify non-conformance issues would not have taken too long to put right, achieving full conformance after a second iteration of testing if not at most the third. The final stage of optimizing code for performance would have sealed a fully working comprehensive XPath 2.0 processor.

6.3.3 Comparison With Other Products

As mentioned previously, the only other XPath 2.0 processor that we were able to compare PsychoPath with was Saxon [5]. This piece of software was written by Michael Kay, who is an editor of the XPath 2.0 specification, so one would suppose (and indeed, one would hope) that he knows what he is doing.

Well he certainly does know what he is doing. There can be no doubt that Saxon is extremely good at what it does. It has already been proven that Saxon performs a lot better than PsychoPath in terms of speed. A lot of effort has clearly gone into optimizing Saxon so that it can be as fast as possible. Also, the fact that the development of Saxon has been an ongoing process for over five years now, means that Michael Kay has had a lot more time to implement functionality in Saxon, and fix whatever bugs may crop up from time to time. Saxon was originally an XPath 1.0 processor, so the current version is backwards compatible with XPath 1.0, except for the few cases where the XPath 2.0 specification and the XPath 1.0 specification are incompatible. PsychoPath has merely been developed over the course of a few months, and because of this, we have not been able to implement as much functionality as we would have liked. For example, backwards compatibility with XPath 1.0 was an optional requirement that we would have liked to have fulfilled, but it just was not possible in the time that we had.

Having said all that, there are a few things that we have found that Saxon could do better, and that PsychoPath can do without a problem. During the PsychoPath testing process, in which we tested many XPath expressions, we also tested the same XPath expressions on Saxon. Sometimes this was because we were not entirely sure what the result to the evaluation of an XPath expression should be and were seeking a second opinion. Mainly though, we ran these tests on Saxon in order to reveal necessary information for us to be able to compare the two XPath processors in terms of implementation of the XPath 2.0 specification and in terms of inaccuracies and other bugs.

However, it should be stated that on Saxon's SourceForge web site, it is possible to submit details of bugs that one has found with Saxon. Although there were some hundreds of bugs listed on the web site, amongst them we were unable to find a lot of the bugs that we had ourselves found with Saxon. This is surprising as we would consider some of the bugs that we found with Saxon to be rather major. This raises the possibility that our usage of Saxon may have been incorrect in some way, thus limiting our ability to harness some of the capabilities that Saxon has to offer. If this is the case, it is unfortunate because it would invalidate this comparison between Saxon and Psychopath, although it is unlikely that every problem we have found with Saxon will turn out to not really be a fault.

How Saxon was used

The way that we used Saxon was to write a Java file in order to process an XPath expression. It involved the four distinct steps of the Saxon process:

1. The XPathEvaluator object is initialized, and the source document is set.

   ```
   InputSource is = new InputSource(
                     new File(filename).toURL().toString());
   SAXSource ss = new SAXSource(is);
   XPathEvaluator xpe = new XPathEvaluator(ss);
   ```

2. The static context is initialized for the source document.

   ```
   StandaloneContext sc = (StandaloneContext)
                           xpe.getStaticContext();
   ```

3. The XPath expression is compiled so that it can be evaluated.

   ```
   XPathExpression expression = xpe.createExpression(exp);
   ```

4. The expression is evaluated in order to find the results.

   ```
   List results = expression.evaluate();
   ```

   The List of results is then cycled through, and each item in the List is examined in turn. This is the start of the `for` loop:

   ```
   for (Iterator iter = results.iterator(); iter.hasNext();) {

       Object item = iter.next();
   ```

   The types of the List items are checked, allowing the program to output its results so that they are labeled as being of the correct types. For example, this part of the code checks if the item is a string, and if so, it outputs a string:

   ```
   if(item instanceof String)
       System.out.println("String: " + (String)item);
   ```

There did not appear to be anything wrong with this way of doing things at the time, but now it seems as though this may not have been the best way of doing it. However it is how we did it, and the information that we gathered this way has got to be used now so that we can compare Saxon with PsychoPath, even though there is a slight possibility that some of the faults that we discovered with Saxon might not actually be real.

Problems with Saxon

The most serious errors that we got with Saxon were to do with various types of expressions. For example, we could not get a for expression `ForExpr` to work. For this expression:

```
for $x in (1,2,3), $y in (4,5,6) return 'something'
```

We would get the correct result from PsychoPath, that result is:

```
1) xs:string: something
2) xs:string: something
3) xs:string: something
4) xs:string: something
5) xs:string: something
6) xs:string: something
7) xs:string: something
8) xs:string: something
9) xs:string: something
```

However Saxon just raises an error as it would appear it does not know how to handle binding tuples or how to evaluate expressions on them. `ForExpr` is not the only type of expression to fail for us where PsychoPath did not. One of many other possible examples would be a quantified expression `QuantExpr` such as this:

```
every $x in (0)  satisfies $x=$x
```

Once again, this input causes Saxon to fail, but PsychoPath returns the correct result, which is:

```
1) xs:boolean: true
```

Another example of a situation where Saxon fails but Psychopath succeeds is where a test is implemented inside a path expression. This test case:

```
document-node(element())
```

should return:

```
Empty results
```

Which it does return this when processed using PsychoPath, but not when it is processed using Saxon, which fails with this expression.

Finally, there was an interesting error that we found with Saxon which involved putting a clock forward a minute at 23:59:59 on New Year's Eve, or putting it back a minute at 00:00:59 on New Year's Day, and then asking what year it was. Faced with this expression:

```
year-from-dateTime(xs:dateTime('1999-12-31T23:59:59-00:01'))
```

Saxon incorrectly says that the new time is still in 1999, whereas PsychoPath correctly returns:

```
1) xs:integer: 2000
```

And when faced with this expression:

```
year-from-dateTime(xs:dateTime('1999-01-01T00:00:59+00:01'))
```

Saxon says that the new time is in 1999, which is wrong. PsychoPath gives the right answer again:

```
1) xs:integer: 1998
```

This is not supposed to be a gloating list of things that Saxon cannot do, but that PsychoPath can. As previously mentioned, it is entirely possible that we were not using Saxon correctly and therefore it might not have been working properly for us. The truth is that Saxon has a lot more of the XPath 2.0 specification implemented than PsychoPath, and has a lot more functionality overall, and is generally more stable. But this is only to be expected given that Saxon has been being developed for so long. It would be nice to think that if the PsychoPath development process had been going on for as long as that, then we would have implemented so much more, possibly even more than Saxon has implemented. This is just a conjecture, but nevertheless, the fact that there are problems with Saxon that do not arise with PsychoPath is very encouraging, and hopefully this will lead to people using PsychoPath instead of—or at least as well as—Saxon, although it is probably unlikely that this will ever happen in any particularly large amounts of numbers.

PsychoPath or Saxon? Is that a Question?

Obviously, although it is very important, the amount of the specification that has been implemented is not the only issue that one would take into account when trying to decide which XPath 2.0 processor to use. Other issues would include things like ease of use, although this is probably not an issue in this situation as PsychoPath and Saxon both have very similar front-ends. An issue that might make a difference is extensibility. The way that PsychoPath has been designed and built means that it is very easy to add support for new functions and other items of grammar. Since the XPath 2.0 specification itself is still only at a draft stage, and therefore subject to any amount of change, this is very useful, as any changes that are made to the specification can easily be made to PsychoPath. It is much harder to implement these changes in Saxon, because Saxon has clearly been designed and built with performance in mind. In order to add new functionality in Saxon, one has to "unwrap" code in order to find the right place to add new code, and then "rewrap;; the new code after it has been added. This is a very complicated and arduous process.

All in all, it is clear that Saxon is "better" than PsychoPath, but this does not mean that there is no reason for people to use PsychoPath. A very large amount of the XPath 2.0 specification has been implemented, and most of it works properly. We have seen that the competition (Saxon) has plenty of bugs itself. PsychoPath may not perform as well as Saxon, but it allows for much easier addition of new code in order to support whatever new functionality may arise in the future.

Chapter 7

Conclusions

Our project was a success on many levels but beaten in other respects. Firstly PsychoPath supports most of the XPath 2.0 specification. Even with 95% of the specification implemented and functioning, there are still some expressions and functions which are yet to be supported being a limitation of our system at present. We did under estimate the large number of operators and types present in XPath 2.0 and we did not leave enough time for the implementation of each one of them. Even Saxon the program which we have based many comparisons with, has not fully implemented the entire specification. We firmly believed that implementing the DOM loading and visitor pattern of the AST would be the hardest milestone to complete, although implementing the operators were not hard when we finally understood what they were meant to do, it was the sheer volume of work needed to implement them which required a large amount of effort.

A success of our system which is a significant improvement on Saxon is that PsychoPath is schema aware and free. This added a separate unique dimension upon our project. To our best knowledge we would be the first XPath 2.0 processor to be schema aware and open source. Although this additional function slows our overall speed, additional work could be implemented to modify the existing code and optimize it to its full potential and improve the speed drastically.

7.1 Satisfying the Contract

Toward the end of the project's life cycle, we managed to produce the deliverables before the deadlines were passed. According to the contract an additional milestone report was produced to list a set of milestones which needed to be completed in order to fulfill the contract agreements. Although we missed one milestone date, it did not effect the final deliverable milestone and we were able to finish the project with the requirements met.

We also listed a possible set of extended optional requirements which were not implemented. These optional milestones were re-evaluated during the course of the project and were decided upon as low priority, as building the base platform of PsychoPath was the highest priority where the optional requirements were merely extensions to this system. However, some of the optional require-

ments have been implemented such as the support for alternative DOM implementations.

7.2 Evaluation of the Impact

Optimization would be a key future requirement which was not considered during our initial implementation. Although listed as an optional extension we were never able to optimize the code as we simply ran out of time and as the contract stated we needed to deliver the project. We did not however understand that part of producing a successful system like PsychoPath was not only to comply to the specification, but to also run as quickly as possible. The use of such systems as PsychoPath and Saxon in the industry would require the full support of the specification but also use the most efficient program. We failed to produce a system that could truly rival Saxon in accordance to speed, but we believed that a schema aware system was an improvement on the free projects currently available.

7.3 Improvements and Changes

There are the obvious improvements that should be made on PsychoPath as it stands at this very moment. The entire specification is not fully supported, such as defining namespaces automatically.

Another improvement would be to optimize our system to an extent that is runs faster, or just as fast as Saxon. In an industry where our system PsychoPath is not the only solution, users rely on productivity to distinguish products. If we were given more time on the project we could have re-factored and optimized the code as best as possible.

7.4 Group and Organization Issues

The group functioned well as individuals and as a unit, but as the project progressed into further complicated areas team members became more complacent due to a number of factors. Internal factors such as lack of communication and not working as much as possible, external factors like the commitments to outside work and course work all played a factor in missing the deadline for milestone 2. We were able to learn quickly from our mistake and quickly resolved our differences between team members who felt that others were not inputting as much effort as some others. Implementing a further way to keep track of team member's situation and keeping tabs on the project as a whole helped push the project back on schedule.

The organization of the group was reliable in theory, but due to these hindering factors the organization at certain points of the project life cycle needed to be disregarded and team member's roles began to merge. As with lots of successful projects flexibility within the group was key to keeping the project forwardly progressing and to help each other.

7.5 The Future

Any future work to improve PsychoPath would be refactoring the code. It would become easier to read, add additional code and become simpler to fix bugs due to the factoring out of common code. Thus further future work would be integrated in a simpler way.

Optimizations of the most used expressions would also be a possible work that would improve PsychoPath and enable it to compete with Saxon. An optimization would produce a more industrial competitive program and would have PsychoPath create more of a impact, possibly attracting developers and contributors.

Another simple extension to the project would be in fact to complete it and implement unsupported features. This would correct PsychoPath and make it support the XPath 2.0 specification fully. This would be another obvious way to increase the impact of PsychoPath with users that require the full range expressiveness. This could be implemented after refactoring, as it would aid the ease of integrating and modifying the code to support all features.

We believe that a solid base has been created for anyone whom desires to continue in our quest of creating the first open source XML Schema Aware XPath 2.0 processor.

Appendix A

User Manual

A.1 How to feed Psychopath XPath expressions

Since PsychoPath has been implemented as an external library and not as a complete program, in order to use it, it needs to be accessed from inside another program. To process XPath 2.0 expressions using PsychoPath from another programs one needs to go through the following process:

1. Load the XML document
2. Optionally validate the XML document
3. Initialize static and dynamic context in respect to the document root
4. Parse the XPath 2.0 expression
5. Statically verify the XPath 2.0 expression
6. Evaluate the XPath 2.0 expression in respect to the XML document

To give a better idea of how this process actually works, we'll go through an example of processing and evaluating the string expression "Hello World!". In this example the XML document that we load is called "XPexample.xml".

All 5 main steps have been explained in detail in Chapter 5, User Interface, so below is just a brief code summary:

```
/**
 * First load and optionally validate the XML document
 */
// Create an InputStream from the XML document
InputStream is = new FileInputStream("XPexample.xml");
// Initializing the Xerces DOM loader.
DOMLoader loader = new XercesLoader();
// Optionally set flag to validate XML document
loader.set_validating(validate);
//Loads the XML document and stores the DOM root
Document doc = loader.load(is);
```

```
/**
 * Dynamic contexts must be initialised to defaults
 * dependent on the XML Schema.
 */
// Extracting the schema from DOM root of XPexpression.xml.
ElementPSVI rootPSVI = (ElementPSVI)doc.getDocumentElement();
XSModel schema = rootPSVI.getSchemaInformation();
// Initialising the DynamicContext.
DynamicContext dc = new DefaultDynamicContext(schema, doc);

// Register the namespaces of the XPath 2.0 predefined datatypes
dc.add_namespace("xs","http://www.w3.org/2001/XMLSchema");
dc.add_namespace("xdt",
                 "http://www.w3.org/2004/10/xpath-datatypes");

// Register the XPath 2.0 standard functions
dc.add_function_library(new FnFunctionLibrary());
dc.add_function_library(new XSCtrLibrary());
dc.add_function_library(new XDTCtrLibrary());

/**
 * Parsing the XPath 2.0 expression into an AST representation
 */
// Initialises PsychoPath's supplied parser.
XPathParser xpp = new JFlexCupParser();
// Parses the XPath expression.
XPath xp = xpp.parse(xpath);

/**
 * Static check the AST to verift structural validity of
 * XPath 2.0 expression
 */
// Initilising StaticChecker.
StaticChecker name_check = new StaticNameResolver(sc);
// Static Checking the XPath expression "'Hello World!'"
name_check.check(xp);

/**
 * Evaluate the XPath 2.0 expression
 */
// Initialising the evaluator with DynamicContext and the name
// of the XML document (XPexample.xml) as parameters.
Evaluator eval = new DefaultEvaluator(dc, doc);
// Evaluates the XPath 2.0 expression, storing the result
// in the ResultSequence
ResultSequence rs = eval.evaluate(xp);
```

XPath 2.0 defines everything to be a sequence of items, including the arguments to expressions and the result of operations. Thus, the overall result of an XPath expression evaluation is also a sequence of items. PsychoPath uses the

class `ResultSequence` as a Collections wrapper to store these sequences and therefore, the result of an evaluation is of this type also. The ResultSequence consists of zero or more items; an item may be a node or a simple-value. “Hello World!” is an example of a single value with length 1. A general sequence could be written as (“a”, “s”, “d”, “f”).

Extraction of certain items from the ResultSequence class is described below, with details of the different operations that one might apply on the ResultSequence. Consider that ’rs’ is the ResultSequence, then:

```
//Will return the number of elements in the sequence, in this
//case of "'Hello World!'" expression size = 1.
rs.size();

//Will return the n'th element in the sequence, in this case of
//"'Hello World!'", if n = 1, then it will return
//XSString of "Hello World!", but if n = 2, it will return
//Empty Result.
rs.get(n);

//Will return true if the sequence is empty.
rs.empty();

//Will return the first element of the sequence,
// in this examlpe it will return XSString of "Hello World!"
rs.first()
```

However, all the items extracted will be of the type’s base class `AnyType` and need to be casted into its actual subtype.
Certain operations always return a particular type and using this knowledge, the extracted item can be immediately casted. In our example “Hello World!” returns a string (easily known as it is inside the quotes ’ ’), so this can safely be casted as such:

```
XSString xsstring = (XSString)(rs.first());
```

The actual result can now be extracted from this XSString in the following manner:

```
String str = xsstring.value();
```

The details of how to cast extracted items from `AnyType` into their actual subtypes with examples is in the next section on How to use each production in the grammar.
However, if the expected return type is unknown or multiple types are possible, the types hierarchy can be traversed in a breadth first manner making use of the Java `instanceof` operator to ascertain the actual type.
This is addressed in full detail in Chapter 5, Section 2.6.

A.2 How to use the XPath 2.0 grammar with PsychoPath

In this section we will try to give you an overview of the XPath 2.0 grammar in general and how each production in the grammar should be used with PsychoPath. For the formal specifications, see the W3C web-site for XPath 2.0 specification[1].

A.2.1 Constants

String literals are written as "Hello" or 'Hello'. In each case the opposite kind of quotation mark can be used within the string: 'He said "Hello" ' or "London is a big city". To feed PsychoPath, " 'Hello World!' "or " "Hello World!" " can be used to feed it with strings. Remember that the ResultSequence returns AnyType so since a string is being expected as the result, first it has to be casted in the code like this:

```
XSString xsstring = (XSString)(rs.first());
```

Numeric constants follow the Java rules for decimal literals: for example, 4 or 4.67; a negative number can be written as -3.05. The numeric literal is taken as a double precision floating point number if it uses scientific notation (e.g. 1.0e7), as a fixed point decimal if it includes a decimal point, or as an integer otherwise. When extracting number literals from the ResultSequence, possible types to be returned include `XSDecimal` (e.g. : xs:decimal: 4.67), `XSInteger` (e.g. : xs:integer: 4) or `XSDouble` (e.g. : xs:double 1e0). All of which need to be casted in the same manner as stated before: from `AnyType` to their corresponding types.

There are no boolean constants as such: instead the function calls `true()` and `false()` are used.

Constants of other data types can be written using constructors. These look like function calls but require a string literal as their argument. For example, xs:float("10.7") produces a single-precision floating point number. To see the full list of the other data types that PsychoPath implements, See Appendix B.

A.2.2 Path expressions

A path expression is a sequence of steps separated by the / or // operator. For example, ../@desc selects the desc attribute of the parent of the context node.

In XPath 2.0, path expressions have been generalized so that any expression can be used as an operand of /, (both on the left and the right), as long as its value is a sequence of nodes. For example, it is possible to use a union expression (in parentheses) or a call to the id() function.

In practice, it only makes sense to use expressions on the right of "/" if they depend on the context item. It is legal to write $x/$y provided both $x and $y are sequences of nodes, but the result is exactly the same as writing ./$y.

Note that the expressions ./$X or $X/. can be used to remove duplicates from $X and sort the results into document order. The same effect can be achieved by writing $X|().

[1] http://www.w3.org/TR/xpath20

The operator "//" is an abbreviation for "/descendant-or-self::node()/". An expression of the form "/E" is shorthand for "root(.)/E", and the expression "/" on its own is shorthand for "root(.)".

A.2.3 Axis steps

The basic primitive for accessing a source document is the axis step. Axis steps may be combined into path expressions using the path operators "/" and "//", and they may be filtered using filter expressions in the same way as the result of any other expression.

An axis step has the basic form `axis::node-test`, and selects nodes on a given axis that satisfy the node-test. The axes available are:

ancestor selects ancestor nodes starting with the current node.

ancestor-or-self Selects the current node plus all ancestor nodes.

attribute Selects all attributes of the current node (if it is an element).

child Selects the children of the current node, in document order.

descendant Selects the children of the current node and their children, recursively (in document order).

descendant-or-self Selects the current node plus all descendant nodes.

following Selects the nodes that follow the current node in document order, other than its descendants.

following-sibling Selects all subsequent child nodes of the same parent node.

parent Selects the parent of the current node.

preceding Selects the nodes that precede the current node in document order, other than its ancestors.

preceding-sibling Selects all preceding child nodes of the same parent node.

self Selects the current node.

When the child axis is used, `child::` may be omitted, and when the attribute axis is used, `attribute::` may be abbreviated to `"@"`. The expression `parent::node()` may be shortened to "." . consider the following node in an XML document:

```
<character>
   <name>Oscar</name>
   <since>2001-10-02</since>
   <age>20</age>
   <qualification>Sheep</qualification>
</character>
<character>
   <name>Sorbo</name>
   <since>2004-10-05</since>
   <age>21</age>
   <qualification>Dog</qualification>
</character>
```

An example of axis steps, along with the result, would be:

```
"//character/child::age"

1) element: age
2) element: age
```

The rest of the axes act in the same manner.

A.2.4 Set difference, intersection and Union

The expression E1 except E2 selects all nodes that are in E1 unless they are also in E2. Both expressions must return sequences of nodes. The results are returned in document order. For example, @* except @note returns all attributes except the note attribute. The expression E1 intersect E2 selects all nodes that are in both E1 and E2. Both expressions must return sequences of nodes. The results are returned in document order. The expression E1 union E2 selects all nodes that are in either E1 or E2 or both. Both expressions must return sequences of nodes. The results are returned in document order. A complete example of the above expression would be as follows. Consider an XML document which looks like this:

```
<nodes>
  <a>
    <connected_a>A</connected_a>
    <connected_a>B</connected_a>
    <connected_a>C</connected_a>
  </a>
  <b>
    <connected_b>B</connected_b>
    <connected_b>C</connected_b>
    <connected_b>D</connected_b>
  </b>
</nodes>
```

then an example of each of the expressions would be:

```
data(/a/*) union data(/b/*)
result:
1) xs:string: A
2) xs:string: B
3) xs:string: C
4) xs:string: D

data(/a/*) intersect data(/b/*)
result:
1) xs:string: B
2) xs:string: C

data(/a/*) except data(/b/*)
result:
1) xs:string: D
```

A.2.5 Arithmetic Expressions

Unary minus and plus: The unary minus operator changes the sign of a number. For example -1 is minus one, and -1e0 is the double value negative -1.

Multiplication and division: The operator * multiplies two numbers. If the operands are of different types, XPath 2.0 specifications say that one of them is promoted to the type of the other but this is currently unsupported in PsychoPath. The result is the same type as the operands after promotion.

The operator div divides two numbers. Dividing two integers produces a double; in other cases the result is the same type as the operands.

The operator idiv performs integer division. For example, the result of 10 idiv 3 is 3.

The mod operator returns the modulus (or remainder) after division.

The operators * and div may also be used to multiply or divide a range by a number. For example, (1 idiv 1 to 3) returns the result: 1) xs:integer: 1 2) xs:integer: 2 3) xs:integer: 3

Addition and subtraction: The operators + and - perform addition and subtraction of numbers, in the usual way. Once again, if the operands are of different types, XPath 2.0 specifications say one of them is promoted but numeric type promotion is currently unsupported by PsychoPath. The result is of the same type as the operands.

Examples of above would be:

```
"-(5 + 7)"
result:
1) xs:integer: -12

"- xs:float('1.23')"
result:
1) xs:float: -1.23

"- xs:double('1.23')"
result:
1) xs:double: -1.23

"(+5 - +7)"
result:
1) xs:integer: -2

"(1 to 5 div 0  )"
result:
FAIL  (division by zero!)

"5*6*10*5*96 div 20 div 3 div 1"
result:
```

```
1) xs:decimal: 2400.0

"31 mod 15"
result:
1) xs:integer: 1
```

A.2.6 Range expressions

The expression E1 to E2 returns a sequence of integers. For example, 1 to 5 returns the sequence 1, 2, 3, 4, 5. This is useful in for expressions, for example the first five nodes of a node sequence can be processed by writing for $i in 1 to 5 return (//x)[$i]. Another example:

```
"(1+1  to 10)"
result:
1) xs:integer: 2
2) xs:integer: 3
3) xs:integer: 4
4) xs:integer: 5
5) xs:integer: 6
6) xs:integer: 7
7) xs:integer: 8
8) xs:integer: 9
9) xs:integer: 10
```

A.2.7 Comparisons

The simplest comparison operators are eq, ne, lt le, gt, ge. These compare two atomic values of the same type, for example two integers, two dates, or two strings. (Collation hasn't been implemented in current version of PsychoPath). If the operands are not atomic values, an error is raised.

The operators =, !=, <, <=, >, and >= can compare arbitrary sequences. The result is true if any pair of items from the two sequences has the specified relationship, for example $A = $B is true if there is an item in $A that is equal to some item in $B.

The operators "is" and "isnot" test whether the operands represent the same (identical) node. For example, "title[1] is *[@note][1]" is true if the first title child is the first child element that has a '@note" attribute. If either operand is an empty sequence the result is an empty sequence (which will usually be treated as false).

The operators << and >> test whether one node precedes or follows another in document order. Consider this XML document:

```
<book>
  <title>Being a Dog Is a Full-Time Job</title>
  <author>Charles M. Schulz</author>
  <character>
      <name>Snoopy</name>
      <friend-of>Peppermint Patty</friend-of>
      <since>1950-10-04</since>
      <age>2</age>
```

```
      <qualification>extroverted beagle</qualification>
  </character>
  <character>
      <name>Peppermint Patty</name>
      <since>1966-08-22</since>
      <age>4</age>
      <qualification>bold, brash and tomboyish</qualification>
  </character>
</book>
```

Examples:

```
"book/character[name="Snoopy"] <<
 book/character[name="Peppermint Patty"]"
result:
1) xs:boolean: true

"book/character[name="Peppermint Patty"] <<
 book/character[name="Snoopy"]"
result:
1) xs:boolean: false
```

A.2.8 Conditional Expressions

XPath 2.0 allows a conditional expression of the form if (E1) then E2 else E3. For example, if (@discount) then @discount else 0 returns the value of the discount attribute if it is present, or zero otherwise.

A.2.9 Quantified Expressions

The expression "some $x in E1 satisfies E2" returns true if there is an item in the sequence E1 for which the effective boolean value of E2 is true. Note that E2 must use the range variable $x to refer to the item being tested; it does not become the context item. For example, some $x in @* satisfies $x eq "" is true if the context item is an element that has at least one zero-length attribute value.

Similarly, the expression every $x in E1 satisfies E2 returns true if every item in the sequence given by E1 satisfies the condition. Example:

```
"every $x in (every $y in (1, 2, 3, 4) satisfies $y = $y*2),
       $z in (3 to 7) satisfies 5"
result:
1) xs:boolean: true
```

A.2.10 For Expressions

The expression "for $x in E1 return E2" returns the sequence that result from evaluating E2 once for every item in the sequence E1. Note that E2 must use the range variable $x to refer to the item being tested; it does not become the context item. Example:

```
"for $x in (1,2,3), $y in (4,5,6) return $x + $y"
result:
```

```
1) xs:integer: 5
2) xs:integer: 6
3) xs:integer: 7
4) xs:integer: 6
5) xs:integer: 7
6) xs:integer: 8
7) xs:integer: 7
8) xs:integer: 8
9) xs:integer: 9
```

A.2.11 And, Or expressions

The expression E1 and E2 returns true if the effective boolean values of E1 and E2 are both true. The expression E1 or E2 returns true if the effective boolean values of either or both of E1 and E2 are true. Example: (for a truth table)

```
"1 and 1"
result:
1) xs:boolean: true

"1 and 0"
result:
1) xs:boolean: false

"1 or 0"
result:
1) xs:boolean: true

"0 or 1"
result:
1) xs:boolean: true
```

A.2.12 SequenceType Matching Expressions

The rules for SequenceType matching compare the actual type of a value with an expected type. These rules are a subset of the formal rules that match a value with an expected type defined in XQuery 1.0 and XPath 2.0 Formal Semantics[2], because the Formal Semantics must be able to match a value with any XML Schema type, whereas the rules below only match values against those types expressible by the SequenceType syntax.

Some of the rules for SequenceType matching require determining whether a given type name is the same as or derived from an expected type name. The given type name may be "known" (defined in the in-scope schema definitions), or "unknown" (not defined in the in-scope schema definitions). An unknown type name might be encountered, for example, if a source document has been validated using a schema that was not imported into the static context. In this case, an implementation is allowed (but is not required) to provide an implementation-dependent mechanism for determining whether the unknown

[2]http://www.w3.org/TR/xpath20/#XQueryFormalSemantics

type name is derived from the expected type name. For example, an implementation might maintain a data dictionary containing information about type hierarchies. consider the following XML document:

```
<sorbo>
  <is>elite</is>
  <!-- life sux -->
</sorbo>
```

then, the following are some example of SequenceType matchings:

```
"element(*)"
result:
1) element: sorbo

"element(elite)"
result:
Empty results

"sorbo/comment()"
result:
1) comment:  life sux

"data(/sorbo/comment())"
result:
1) xs:string:  life sux

"sorbo/node()"
result:
1) text:

2) element: is
3) comment:  life sux
4) text:
```

A.3 How to use XPath 2.0 functions with PsychoPath

The aim of this section is to give the user an overview of the available XPath 2.0 functions that are implemented in PsychoPath. For the formal specifications, see the W3C web-site for XPath 2.0 functions and operators[3].

A.3.1 Accessors

In order for PsychoPath to operate on instances of the XPath 2.0 data model, the model must expose the properties of the items it contains. It does this by defining a family of accessor functions. These functions are not available to users or applications to call directly. Instead, they are descriptions of the information that an implementation of the model must expose to applications.

[3] http://www.w3.org/TR/xpath-functions/

fn:node-name returns zero or one `QName` items.

fn:nilled returns zero or one `XSBoolean` items.

fn:string returns an `XSString` item.

fn:data returns a sequence of zero or more `AnyAtomicType` items.

fn:base-uri returns zero or one `XSAnyURI` items.

fn:document-uri returns zero or one `XSAnyURI` items.

For the above functions, the return types are listed, and it is safe to cast the return values for these functions to those types.

Example

If we wanted to evaluate the XPath expression:

```
data('string')
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
String n = ((XSString)rs.first()).string_value();
println(n);
```

in order to get the result of `'string'`

A.3.2 The Error and Trace Functions

fn:error does not return anything.

fn:trace returns a sequence of zero or more items.

The error function does not return anything that could be cast to anything, and the trace function returns exactly the same items that are used as its input arguments. So it would be possibly to cast accordingly, should one desire to do so.

A.3.3 Constructor Functions

Constructor functions exist for all supported types (See Appendix B.4). The result of the function will always be the same type as the Constructor, so one can safely cast the return value to that type (provided that the expression only consists of a single Constructor function).

Example

If we wanted to evaluate the XPath expression:

```
xs:dateTime("2002-02-01T10:00:00+06:00")
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
String n = ((XSDateTime)rs.first()).string_value();
println(n);
```

in order to get the result of `'2002-02-01T04:00:00Z'`

A.3.4 Functions on Numeric Values

fn:abs

fn:ceiling

fn:floor

fn:round

fn:round-half-to-even

For any of the above functions, the return value will be of the same numeric type as the input argument, therefore it it is safe to cast the return value to that numeric type.

Example

If we wanted to evaluate the XPath expression:

```
ceiling(xs:float('10.4'))
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
float n = ((XSFloat)rs.first()).float_value();
println(n);
```

in order to get the result of `'11.0'`

A.3.5 Functions to Assemble and Disassemble Strings

`fn:codepoints-to-string` returns an `XSString` item.

`fn:string-to-codepoints` returns a sequence of `XSInteger` items.

For the above functions, the return types are listed, and it is safe to cast the return values for these functions to those types.

Example

If we wanted to evaluate the XPath expression:

```
codepoints-to-string(0111)
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
String n = ((XSString)rs.first()).string_value();
println(n);
```

in order to get the result of `'o'`

A.3.6 Compare and Other Functions on String Values

fn:compare returns zero or one `XSInteger` items.

fn:concat

fn:string-join

fn:substring

fn:string-length returns an `XSInteger` item.

fn:normalize-space

fn:upper-case

fn:lower-case

fn:translate

fn:escape-uri

All of the above functions that do not return items of type `XSInteger`, return items of type `XSString`. Therefore, for all of those functions, it is safe to cast the return value to a string. For the other two functions already mentioned, it is safe to cast the return value to an integer.

Example

If we wanted to evaluate the XPath expression:

```
concat('un', 'grateful')
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
String n = ((XSString)rs.first()).string_value();
println(n);
```

in order to get the result of `'ungrateful'`

A.3.7 Functions Based on Substring Matching

fn:contains returns an `XSBoolean` item.

fn:starts-with returns an `XSBoolean` item.

fn:ends-with returns an `XSBoolean` item.

fn:substring-before returns an `XSString` item.

fn:substring-after returns an `XSString` item.

The return values of the above functions that return items of type `XSBoolean` can safely be cast as booleans. Similarly, the return values of the above functions that return items of type `XSString` can safely be cast as strings.

Example

If we wanted to evaluate the XPath expression:

```
contains("abc", "edf")
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
boolean n = ((XSBoolean)rs.first()).value();
println(n);
```

in order to get the result of `'false'`

A.3.8 String Functions that Use Pattern Matching

fn:matches returns an `XSBoolean` item.

fn:replace returns an `XSString` item.

fn:tokenize returns a sequence of at least one `XSString` item.

The return value of the above function that returns an item of type `XSBoolean` can safely be cast as a boolean, and the return value of the above function that returns an item of type `XSString` can safely be cast as a string. Lastly, the above function that returns a sequence of items of type `XSString` can safely be cast as a series of strings

Example

If we wanted to evaluate the XPath expression:

```
matches('abcd', 'abcd')
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
boolean n = ((XSBoolean)rs.first()).value();
println(n);
```

in order to get the result of `'true'`

A.3.9 Functions on Boolean Values

fn:true

fn:false

fn:not

For any of the above functions, the return value will be a boolean value, therefore it it is safe to cast the return value as a boolean.

Example

If we wanted to evaluate the XPath expression:

```
not(true())
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
boolean n = ((XSBoolean)rs.first()).value();
println(n);
```

in order to get the result of `'false'`

A.3.10 Component Extraction Functions on Durations, Dates and Times

fn:years-from-duration

fn:months-from-duration

fn:days-from-duration

fn:hours-from-duration

fn:minutes-from-duration

fn:seconds-from-duration returns zero or one `XSDecimal` items.

fn:year-from-dateTime

fn:month-from-dateTime

fn:day-from-dateTime

fn:hours-from-dateTime

fn:minutes-from-dateTime

fn:seconds-from-dateTime returns zero or one `XSDecimal` items.

fn:timezone-from-dateTime will return zero or one `XDTDayTimeDuration` items.

fn:year-from-date

fn:month-from-date

fn:day-from-date

fn:timezone-from-date returns zero or one `XDTDayTimeDuration` items.

fn:hours-from-time

fn:minutes-from-time

fn:seconds-from-time returns zero or one `XSDecimal` items.

fn:timezone-from-time returns zero or one `XDTDayTimeDuration` items.

All of the above functions that do not return items of type `XSDecimal` or `XDTDayTimeDuration`, return zero or one items of type `XSInteger`. Therefore, for all of those functions, it is safe to cast the return values as integers. For the functions that return items of type `XSDecimal`, it is safe to cast the return values as decimals, and for the functions that return items of type `XDTDayTimeDuration`, it is safe to cast the return values as day time durations.

Example

If we wanted to evaluate the XPath expression:

```
timezone-from-time(xs:time("13:20:00+05:00"))
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
String n = ((XDTDayTimeDuration)rs.first()).string_value();
println(n);
```

in order to get the result of `'PT5H'`

A.3.11 Functions Related to QNames

fn:QName returns a `QName` item.

fn:local-name-from-QName returns zero or one `XSNCName` items.

fn:namespace-uri-from-QName returns zero or one `XSAnyURI` items.

For the above functions, the return types are listed, and it is safe to cast the return values for these functions to those types.

Example

If we wanted to evaluate the XPath expression:

```
local-name-from-QName(QName('http://www.example.com/example',
                            'person'))
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
String n = ((XSNCName)rs.first()).string_value();
println(n);
```

in order to get the result of `'person'`

A.3.12 Functions on Nodes

fn:name

fn:local-name

fn:number returns an `XSDouble` item.

fn:lang returns an `XSBoolean` item.

It is safe to cast the return value of the above function that returns an `XSDouble` to a double, and it is safe to cast the return value of the above function that returns an `XSBoolean` to a boolean. The other two functions both return items of type `XSString`, and therefore it is safe to cast the return values for these functions to strings.

A.3.13 General Functions on Sequences

fn:boolean returns an `XSBoolean` item.

fn:index-of returns a sequence of zero or more `XSInteger` items.

fn:empty returns an `XSBoolean` item.

fn:exists returns an `XSBoolean` item.

fn:distinct-values returns a sequence of zero or more `AnyAtomicType` items.

fn:insert-before

fn:remove

fn:reverse

fn:subsequence

fn:unordered

The above functions that do not have their return types listed, all return sequences of zero or more items. The types of items returned will be the same as the types of items that were used as the input arguments, and therefore it is safe to cast the return values to whichever type of input arguments they correspond to. The above function that returns a sequence of zero or more `AnyAtomicType` items works in exactly the same way - the types of the returned items will be the same as the input arguments, and it possible to cast accordingly. It is safe to cast the return values of the above functions that return `XSBoolean` values to booleans, and it is also safe to cast the return values of the above function that returns a sequence of `XSInteger` values to a series of integers.

Example

If we wanted to evaluate the XPath expression:

```
remove((‘s’,‘o’,‘m’,‘e’,‘t’,‘h’,‘i’,‘n’,‘g’), 6)
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
for (Iterator iter = rs.iterator(); iter.hasNext();) {

    Object item = iter.next();
    String n = ((XSString)item).string_value();
    print(n + " ");
}
println("");
```

in order to get the result of ‘s o m e t i n g’

A.3.14 Functions That Test the Cardinality of Sequences

fn:zero-or-one returns a sequence of zero or one items.

fn:one-or-more returns a sequence of one or more items.

fn:exactly-one returns one item.

Once again, the functions listed above return items that are of the same types as their input arguments, and it is safe to cast accordingly.

Example

If we wanted to evaluate the XPath expression:

```
one-or-more((1,2,3,4,5))
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
for (Iterator iter = rs.iterator(); iter.hasNext();) {

    Object item = iter.next();
    int n = ((XSInteger)item).int_value();
    print(n + " ");
}
println("");
```

in order to get the result of `'1 2 3 4 5'`

A.3.15 Deep-Equal, Aggregate Functions, and Functions that Generate Sequences

fn:deep-equal returns an `XSBoolean` item.

fn:count returns an `XSInteger` item.

fn:avg

fn:max

fn:min

fn:sum

fn:doc returns a sequence of zero or one document-nodes.

It is safe to cast the return values of the functions that return `XSBoolean` items and `XSInteger` items as booleans and as integers respectively. The other functions listed above that the return types are not listed for, all return sequences of zero or one `AnyAtomicType` items. When using PsychoPath, one must cast the return values for these functions to doubles.

Example

If we wanted to evaluate the XPath expression:

```
avg((3,4,5))
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
double avg = ((XSDouble)rs.first()).double_value();
println(avg);
```

in order to get the result of `'4.0'`

A.3.16 Context Functions

fn:position returns an `XSInteger` item.

fn:last returns an `XSInteger` item.

fn:current-dateTime returns an `XSDateTime` item.

fn:current-date returns an `XSDate` item.

fn:current-time returns an `XSTime` item.

fn:default-collation returns an `XSString` item.

fn:implicit-timezone returns an `XDTDayTimeDuration` item.

For the above functions, the return types are listed, and it is safe to cast the return values for these functions to those types.

Example

If we wanted to evaluate the XPath expression:

```
(10 to 20)[position() = 2]
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
int pos = ((XSInteger)rs.first()).int_value();
println(pos);
```

in order to get the result of `'11'`

A.4 How to use XPath 2.0 operators with PsychoPath

The aim of this section is to give the user an overview of the available XPath 2.0 operators that are implemented in PsychoPath. For the formal specifications, see the W3C web-site for XPath 2.0 functions and operators[4].

[4] http://www.w3.org/TR/xpath-functions/

A.4.1 Operators on Numeric Values

op:numeric-add

op:numeric-subtract

op:numeric-multiply

op:numeric-divide

op:numeric-integer-divide returns an `XSInteger` item.

op:numeric-mod

op:numeric-unary-plus

op:numeric-unary-minus

The operators above that the return types haven't been listed for, all return the same numeric type that was input into the operator. This is because PsychoPath does not support numeric type promotion, so the two inputs to every operator have to be of the same type, and the output will be of the same type as well. With this in mind, it is safe to cast the return values for these operators to these types. It is also safe to cast the return values of the operator that returns `XSInteger` items as integers.

Example

If we wanted to evaluate the XPath expression:

```
xs:integer(4) + xs:integer(3)
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
integer n = ((XSInteger)rs.first()).integer_value();
println(n);
```

in order to get the result of `'7'`

A.4.2 Comparison of Numeric Values

op:numeric-equal

op:numeric-less-than

op:numeric-greater-than

All of the above operators return `XSBoolean` items.Therefore it is safe to cast the return values for these operators to booleans.

Example

If we wanted to evaluate the XPath expression:

```
xs:decimal(3.3) = xs:decimal(6.6)
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
boolean n = ((XSBoolean)rs.first()).value();
println(n);
```

in order to get the result of `'false'`

A.4.3 Operators on Boolean Values

op:boolean-equal

op:boolean-less-than

op:boolean-greater-than

All of the above operators return `XSBoolean` items.Therefore it is safe to cast the return values for these operators to booleans.

Example

If we wanted to evaluate the XPath expression:

```
xs:boolean('true') gt xs:boolean('false')
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
boolean n = ((XSBoolean)rs.first()).value();
println(n);
```

in order to get the result of `'true'`

A.4.4 Comparisons of Duration, Date and Time Values

op:yearMonthDuration-equal

op:yearMonthDuration-less-than

op:yearMonthDuration-greater-than

op:dayTimeDuration-equal

op:dayTimeDuration-less-than

op:dayTimeDuration-greater-than

op:dateTime-equal

op:dateTime-less-than

op:dateTime-greater-than

op:date-equal

op:date-less-than

op:date-greater-than

op:time-equal

op:time-less-than

op:time-greater-than

op:gYearMonth-equal

op:gYear-equal

op:gMonthDay-equal

op:gMonth-equal

op:gDay-equal

All of the above operators return `XSBoolean` items.Therefore it is safe to cast the return values for these operators to booleans.

Example

If we wanted to evaluate the XPath expression:

```
xs:time("23:00:00+06:00") < xs:time("12:00:00-06:00")
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
boolean n = ((XSBoolean)rs.first()).value();
println(n);
```

in order to get the result of `'true'`

A.4.5 Arithmetic Functions on Durations

op:add-yearMonthDurations returns an `XDTYearMonthDuration` item.

op:subtract-yearMonthDurations returns type `XDTYearMonthDuration`.

op:multiply-yearMonthDuration returns an `XDTYearMonthDuration` item.

op:divide-yearMonthDuration returns an `XDTYearMonthDuration` item.

op:divide-yearMonthDuration-by-yearMonthDuration returns an item of type `XSDecimal`.

op:add-dayTimeDurations returns an `XDTDayTimeDuration` item.

op:subtract-dayTimeDurations returns an `XDTDayTimeDuration` item.

op:multiply-dayTimeDuration returns an `XDTDayTimeDuration` item.

op:divide-dayTimeDuration returns an `XDTDayTimeDuration` item.

op:divide-dayTimeDuration-by-dayTimeDuration returns an item of the type of `XSDecimal`.

For the above functions, the return types are listed, and it is safe to cast the return values for these functions to those types.

Example

If we wanted to evaluate the XPath expression:

```
multiply-dayTimeDuration(xdt:dayTimeDuration("PT2H10M"), 2.1)
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
String n = ((XDTDayTimeDuration)rs.first()).string_value();
println(n);
```

which returns a xdt:dayTimeDuration value corresponding to 4 hours and 33 minutes `'PT4H33M'`

A.4.6 Arithmetic Functions Dates and Times

op:subtract-dateTimes-yielding-dayTimeDuration returns 0 or 1 objects of type `XDTDayTimeDuration`.

op:subtract-dates-yielding-dayTimeDuration returns zero or one objects of type `XDTDayTimeDuration`.

op:subtract-times returns an `XDTDayTimeDuration` item.

op:add-yearMonthDuration-to-dateTime returns an `XSDateTime` item.

op:add-dayTimeDuration-to-dateTime returns an `XSDateTime` item.

op:subtract-yearMonthDuration-from-dateTime returns an `XSDateTime` item.

op:subtract-dayTimeDuration-from-dateTime returns an object of type `XSDateTime`.

op:add-yearMonthDuration-to-date returns an `XSDate` item.

op:add-dayTimeDuration-to-date returns an `XSDate` item.

op:subtract-yearMonthDuration-from-date returns an `XSDate` item.

op:subtract-dayTimeDuration-from-date returns an `XSDate` item.

op:add-dayTimeDuration-to-time returns an `XSTime` item.

op:subtract-dayTimeDuration-from-time returns an `XSTime` item.

For the above functions, the return types are listed, and it is safe to cast the return values for these functions to those types.

Example

If we wanted to evaluate the XPath expression:

```
add-yearMonthDuration-to-dateTime(
  xs:dateTime("2000-10-30T11:12:00"),
  xdt:yearMonthDuration("P1Y2M"))
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
String n = ((XSDateTime)rs.first()).string_value();
println(n);
```

which returns an xs:dateTime value corresponding to the lexical representation `'2001-12-30T11:12:00'`

A.4.7 Operators Related to QNames And Nodes

op:QName-equal

op:is-same-node

op:node-before

op:node-after

All of the above operators return `XSBoolean` items. Therefore it is safe to cast the return values for these operators to booleans.

Example

If we wanted to evaluate the XPath expression:

```
xs:QName('ao') eq xs:QName('ao')
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
boolean n = ((XSBoolean)rs.first()).value();
println(n);
```

which returns the result of `'true'`

A.4.8 Union, Intersection and Except

op:union

op:intersect

op:except

All of the above operators return sequences of zero or more node items. Therefore it is safe to cast the return values for these operators to a series of nodes.

Example

Assume $seq1 = ($item1, $item2), $seq2 = ($item1, $item2) and $seq3 = ($item2, $item3). If we wanted to evaluate the XPath expression:

```
union(\$seq2, \$seq3)
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
for (Iterator iter = rs.iterator(); iter.hasNext();) {

    Object item = iter.next();
    String n = ((XSString)item).string_value();
    print(n + ", ");
}
println("");
```

which returns the sequence consisting of `$item1, $item2, $item3`.

A.4.9 Operators that Generate Sequences

op:to

The above operator returns sequences of zero or more `XSInteger` items. Therefore it is safe to cast the return values for this operator to a series of `XSInteger` items.

Example

If we wanted to evaluate the XPath expression:

```
(1 to 3)
```

from within a Java application, in order to extract the result from the result sequence, one would have to use this code:

```
int n = (XSInteger)rs.first()).string_value();
println(n);
```

which returns the sequence consisting of `1, 2, 3`.

Appendix B

Implementation Status

The following tables summarize the XPath 2.0 specification implementation status of PsychoPath. Any issues on particular aspects are commented.

B.1 XPath Grammar Production Rules

This table lists all the production rules from the XPath 2.0 specification [11] and the relevant rule numbers from the EBNF section of the specification. The implementation status for each of the rules are noted.

Production Name	Comments on Implementation
01 XPath	Only God knows.
02 Expr	Works fully.
03 ExprSingle	Works fully.
04 ForExpr	Works fully.
05 SimpleForClause	Works fully.
06 QuantifiedExpr	Works fully.
07 IfExpr	Works fully.
08 OrExpr	Works Fully.
09 AndExpr	Works Fully.
10 ComparisonExpr	Works fully for ValueComp and GeneralComp. NodeComp does not work properly when nodes of different documents are being compared.
11 RangeExpr	Works fully.
12 AdditiveExpr	Works fully.
13 MultiplicativeExpr	Works fully.
14 Union	Works fully.
15 IntersectExceptExpr	Fails when comparing two nodes with the same data values: • Except expression does not return Empty results. • Intersect expression does not return the data values.

Production Name	Comments on Implementation
16 InstanceofExpr	Not working.
17 TreatExpr	Works fully.
18 CastableExpr	Works fully.
19 CastExpr	Works fully.
20 UnaryExpr	Works fully.
21 ValueExpr	Works fully.
22 GeneralComp	Works fully when comparing same type, but fails when different types are compared such as string to integer, this should return either true or false.
23 ValueComp	Works fully when comparing same type, but fails when different types are compared such as string to integer, this should return either true or false.
24 NodeComp	Works fully.
25 PathExpr	Works fully.
26 RelativePathExpr	Works fully.
27 StepExpr	Works fully.
28 AxisStep	Works fully.
29 ForwardStep	Works fully.
30 ForwardAxis	Works fully.
31 AbbrevForwardStep	Works fully.
32 ReverseStep	Works fully.
33 ReverseAxis	Works fully.
34 AbbrevReverseStep	Works fully.
35 NameTest	Works fully.
36 NodeTest	Works fully.
37 Wildcard	Works fully.
38 FilterExpr	Works fully.
39 PredicateList	Works fully. Multiple expressions in a single predicate also supported.
40 Predicate	Works fully.
41 PrimaryExpr	Works fully.
42 Literal	Works fully.
43 NumericLiteral	Works fully.
44 VarRef	Works fully.
45 ParenthesizedExpr	Works fully.
46 ContextItemExpr	Works fully.
47 FunctionCall	Works fully.
48 SingleType	Works fully.
49 SequenceType	Works fully.
50 OccurenceIndicator	Works fully.
51 ItemType	Works fully.
52 AtomicType	Works fully.
53 KindTest	Works fully.
54 AnyKindTest	Works fully.
55 DocumentTest	Works fully.
56 TextTest	Works fully.
57 CommentTest	Works fully.
58 PITest	Works fully.
59 AttributeTest	Works fully.
60 AttribNameOrWildcard	Works fully.
61 SchemaAttributeTest	Works fully.
62 AttributeDeclaration	Works fully.
63 ElementTest	Works fully.
64 ElementNameOrWildcard	Works fully.
65 SchemaElementTest	Works fully.

Production Name	Comments on Implementation
66 ElementDeclaration	Works fully.
67 AttributeName	Works fully.
68 ElementName	Works fully.
69 TypeName	Works fully.
70 IntegerLiteral	Works fully.
71 DecimalLiteral	Works fully.
72 DoubleLiteral	Works fully.
73 StringLiteral	Works fully.
75 Digits	Works fully.
76 Comment	Works fully.
77 CommentContents	Works fully.
78 QName	Works fully.
79 NCName	Works fully.
80 Char	Works fully.

B.2 XPath Functions

This table lists all the functions from the XPath Functions and Operators specification [15]. The relevant section number is also provided as well as the comments on the current implementation status of PsychoPath.

Section	XPath Function	Comments on Implementation
2.1	fn:node-name	Works fully.
2.2	fn:nilled	Works partially.
2.3	fn:string	The arity of 0 is not implemented for this function.
2.4	fn:data	Works fully.
2.5	fn:base-uri	Works partially.
2.6	fn:static-base-uri	Not implemented.
2.7	fn:document-uri	Works partially.
3	fn:error	Only arity of 0 is implemented for this function.
4	fn:trace	Works fully.
6.4.1	fn:abs	Works fully.
6.4.2	fn:ceiling	Fails when multiplying an int with a decimal—specifically in that order *probably a global problem as this function binds the result to the first arguments type.*
6.4.3	fn:floor	Works fully.
6.4.4	fn:round	Works fully.
6.4.5	fn:round-half-to-even	Arity of 2 not supported i.e. round-half-to-even(3.114, 2) result should be 3.11
7.2.1	fn:codepoints-to-string	Works fully.
7.2.2	fn:string-to-codepoints	Works fully.
7.3.2	fn:compare	Only arity of 2 is implemented.
7.4.1	fn:concat	Works fully.
7.4.2	fn:string-join	Possibly doesn't works fully when applied to elements of an XML document.
7.4.3	fn:substring	Only Double type supported. Only arity of 2 is implemented.
7.4.4	fn:string-length	The arity of 0 is not implemented for this function.
7.4.5	fn:normalize-space	The arity of 0 s not implemented for this function.
7.4.6	fn:normalize-unicode	Not tested. Arity of 2 is not implemented for this function.

Section	XPath Function	Comments on Implementation
7.4.7	fn:upper-case	Works fully.
7.4.8	fn:lower-case	Works fully.
7.4.9	fn:translate	Works fully.
7.4.10	fn:escape-uri	Works fully.
7.5.1	fn:contains	Collations does not works with this function.
7.5.2	fn:starts-with	Works fully.
7.5.3	fn:ends-with	Works fully.
7.5.4	fn:substring-before	Works fully.
7.5.5	fn:substring-after	Works fully.
7.6.2	fn:matches	Works fully.
7.6.3	fn:replace	Works fully.
7.6.4	fn:tokenize	Works fully.
8.1	fn:resolve-uri	Not implemented.
9.1.1	fn:true	Works fully.
9.1.2	fn:false	Works fully.
9.3.1	fn:not	Works fully.
10.4.1	fn:years-from-duration	Works fully.
10.4.2	fn:months-from-duration	Negative values not supported.
10.4.3	fn:days-from-duration	Negative values not supported.
10.4.4	fn:hours-from-duration	Negative values not supported.
10.4.5	fn:minutes-from-duration	Negative values not supported.
10.4.6	fn:seconds-from-duration	Negative values not supported.
10.4.7	fn:year-from-dateTime	Problem when the time zone is at dateTime boundary, within range to affect extraction component, e.g.: year-from-dateTime(xs:dateTime(“1999-12-31T19:20:00-05:00”)) yields 2000; should be 1999. This is because time-zone -05:00 performs calculation of plus 05:00 hrs to time when it should not. Time zone should be inactive.
10.4.8	fn:month-from-dateTime	Problem when the time zone is at dateTime boundary, within range to affect extraction component. Time zone should be inactive i.e. not perform a calculation.
10.4.9	fn:day-from-dateTime	Problem when the time zone is at dateTime boundary, within range to affect extraction component. Time zone should be inactive i.e. not perform a calculation.
10.4.10	fn:hours-from-dateTime	Problem when the time zone is at dateTime boundary, within range to affect extraction component. Time zone should be inactive i.e. not perform a calculation.
10.4.11	fn:minutes-from-dateTime	Problem when the time zone is at dateTime boundary, within range to affect extraction component. Time zone should be inactive i.e. not perform a calculation.
10.4.12	fn:seconds-from-dateTime	Problem when the time zone is at dateTime boundary, within range to affect extraction component. Time zone should be inactive i.e. not perform a calculation.
10.4.13	fn:timezone-from-dateTime	Fails if no time zone specified.
10.4.14	fn:year-from-date	Problem when the time zone is at date boundary, within range to affect extraction component. Time zone should be inactive i.e. not perform a calculation.

Section	XPath Function	Comments on Implementation
10.4.15	fn:month-from-date	Problem when the time zone is at date boundary, within range to affect extraction component. Time zone should be inactive i.e. not perform a calculation.
10.4.16	fn:day-from-date	Problem when the time zone is at date boundary, within range to affect extraction component. Time zone should be inactive i.e. not perform a calculation.
10.4.17	fn:timezone-from-date	Fails if no time zone specified.
10.4.18	fn:hours-from-time	If the time zone is specified, the output is disrupted.
10.4.19	fn:minutes-from-time	If the time zone is specified with minutes then output is corrupted.
10.4.20	fn:seconds-from-time	Works fully.
10.4.21	fn:timezone-from-time	Fails if no time zone is specified.
10.6.1	fn:adjust-dateTime-to-timezone	Not implemented.
10.6.2	fn:adjust-date-to-timezone	Not implemented.
10.6.3	fn:adjust-time-to-timezone	Not implemented.
11.1.2	fn:QName	Not fully implemented. Takes the second argument as the QName.
11.2.2	fn:local-name-from-QName	Works fully, but depends on fn:QName.
11.2.3	fn:namespace-uri-from-QName	This depends on fn:QName.
14.1	fn:name	Works fully.
14.2	fn:local-name	Works fully.
14.3	fn:namespace-uri	Not implemented.
14.4	fn:number	Works fully, but only arity of 1 is implemented.
15.1.1	fn:boolean	Works fully.
15.1.3	fn:index-of	Only arity of 2 is implemented for this function.
15.1.4	fn:empty	Works fully.
15.1.5	fn:exists	Works fully.
15.1.6	fn:distinct-values	Only arity of 1 is implemented for this function.
15.1.7	fn:insert-before	Works fully.
15.1.8	fn:remove	Works fully.
15.1.9	fn:reverse	Works fully.
15.1.10	fn:subsequence	Only arity of 3 is implemented for this function.
15.1.11	fn:unordered	Works fully.
15.2.1	fn:zero-or-one	Works fully.
15.2.2	fn:one-or-more	Works fully.
15.2.3	fn:exactly-one	Works fully.
15.3.1	fn:deep-equal	Works fully.
15.4.1	fn:count	Works fully.
15.4.2	fn:avg	Works fully.
15.4.3	fn:max	Works fully.
15.4.4	fn:min	Works fully.
15.4.5	fn:sum	Works fully.
15.5.4	fn:doc	Works partially.
16.1	fn:position	Works fully.
16.2	fn:last	Works fully.
16.3	fn:current-dateTime	Works fully.
16.4	fn:current-date	Works fully.

Section	XPath Function	Comments on Implementation
16.5	fn:current-time	Works fully.
16.6	fn:default-collation	Works partially. Arity of 1 not yet implemented.
16.7	fn:implicit-timezone	Works fully.
17	Casting	Works fully.

B.3 XPath Operators

This table lists all the operators from the XPath Functions and Operators specification [15]. The relevant section number is also provided as well as the comments on the current implementation status of PsychoPath.

Section	XPath Operator	Comments
6.2.1	op:numeric-add	Works fully.
6.2.2	op:numeric-subtract	Works fully except for numeric type promotion.
6.2.3	op:numeric-multiply	Works fully except for numeric type promotion.
6.2.4	op:numeric-divideb	Works fully except for numeric type promotion.
6.2.5	op:numeric-integer-divide	Works fully except for numeric type promotion.
6.2.6	op:numeric-mod	Works fully except for numeric type promotion.
6.2.7	op:numeric-unary-plus	Works fully.
6.2.8	op:numeric-unary-minus	Works fully.
6.3.1	op:numeric-equal	Works fully.
6.3.2	op:numeric-less-than	Works fully.
6.3.3	op:numeric-greater-than	Works fully.
9.2.1	op:boolean-equal	Works fully.
9.2.2	op:boolean-less-than	Works fully.
9.2.3	op:boolean-greater-than	Works fully.
10.3.1	op:yearMonthDuration-equal	Works fully.
10.3.2	op:yearMonthDuration-less-than	Works fully.
10.3.3	op:yearMonthDuration-greater-than	Works fully.
10.3.4	op:dayTimeDuration-equal	Works fully.
10.3.5	op:dayTimeDuration-less-than	Works fully.
10.3.6	op:dayTimeDuration-greater-than	Works fully.
10.3.7	op:dateTime-equal	Works fully.
10.3.8	op:dateTime-less-than	Works fully.
10.3.9	op:dateTime-greater-than	Works fully.
10.3.10	op:date-equal	Works fully.
10.3.11	op:date-less-than	Works fully.
10.3.12	op:date-greater-than	Works fully.
10.3.13	op:time-equal	Works fully.
10.3.14	op:time-less-than	Works fully.
10.3.15	op:time-greater-than	Works fully.
10.3.16	op:gYearMonth-equal	Works fully.
10.3.17	op:gYear-equal	Works fully.
10.3.18	op:gMonthDay-equal	Works fully.
10.3.19	op:gMonth-equal	Works fully.
10.3.20	op:gDay-equal	Works fully.
10.5.1	op:add-yearMonthDurations	Works fully.
10.5.2	op:subtract-yearMonthDurations	Works fully.
10.5.3	op:multiply-yearMonthDuration	Works fully.

Section	XPath Operator	Comments
10.5.4	op:divide-yearMonthDuration	Works fully.
10.5.5	op:divide-yearMonthDuration-by-yearMonthDuration	Works fully.
10.5.6	op:add-dayTimeDurations	Works fully.
10.5.7	op:subtract-dayTimeDurations	Works fully.
10.5.8	op:multiply-dayTimeDuration	Works fully.
10.5.9	op:divide-dayTimeDuration	Works fully.
10.5.10	op:divide-dayTimeDuration-by-dayTimeDuration	Works fully.
10.7.1	op:subtract-dateTimes-yielding-dayTimeDuration	Works fully.
10.7.2	op:subtract-dates-yielding-dayTimeDuration	Works fully.
10.7.3	op:subtract-times	Works fully.
10.7.4	op:add-yearMonthDuration-to-dateTime	Works fully.
10.7.5	op:add-dayTimeDuration-to-dateTime	Works fully.
10.7.6	op:subtract-yearMonthDuration-from-dateTime	Works fully.
10.7.7	op:subtract-dayTimeDuration-from-dateTime	Works fully.
10.7.8	op:add-yearMonthDuration-to-date	Works fully.
10.7.9	op:add-dayTimeDuration-to-date	Works fully.
10.7.10	op:subtract-yearMonthDuration-from-date	Works fully.
10.7.11	op:subtract-dayTimeDuration-from-date	Works fully.
10.7.12	op:add-dayTimeDuration-to-time	Works fully.
10.7.13	op:subtract-dayTimeDuration-from-time	Works fully.
11.2.1	op:QName-equal	Works fully.
12.1.1	op:hexBinary-equal	Not implemented.
12.1.2	op:base64Binary-equal	Not implemented.
13.1.1	op:NOTATION-equal	Not implemented.
14.6	op:is-same-node	Works fully.
14.7	op:node-before	Works fully.
14.8	op:node-after	Works fully.
15.3.2	op:union	Works fully.
15.3.3	op:intersect	Works fully.
15.3.4	op:except	Works fully.
15.5.1	op:to	Works fully.

B.4 XML Schema Data types

This table lists all the supported data types defined in XML Schema [12] and their implementation status in PsychoPath. The relevant section numbers from the specification are also noted.

Section	XPath Schema Data type	Comments on Implementation
3.2.1	string	Works fully.
3.2.2	boolean	Works fully.
3.2.3	decimal	Works fully except for numeric type conversion.
3.2.4	float	Works fully except for numeric type conversion.
3.2.5	double	Works fully except for numeric type conversion.
3.2.6	duration	Works fully.
3.2.7	dateTime	Works fully.

Section	XPath Schema Data type	Comments on Implementation
3.2.8	time	Works fully.
3.2.9	date	Works fully.
3.2.10	gYearMonth	Works fully.
3.2.11	gYear	Works fully.
3.2.12	gMonthDay	Works fully.
3.2.13	gDay	Works fully.
3.2.14	gMonth	Works fully.
3.2.15	hexBinary	Not implemented.
3.2.16	base64Binary	Not implemented.
3.2.17	anyURI	Not implemented.
3.2.18	QName	Works fully.
3.2.19	NOTATION	Not implemented.
3.3.1	normalizedString	Not implemented.
3.3.2	token	Works fully.
3.3.3	language	Works fully.
3.3.4	NMTOKEN	Not implemented.
3.3.5	NMTOKENS	Not implemented.
3.3.6	Name	Works fully.
3.3.7	NCName	Works fully.
3.3.8	ID	Works fully.
3.3.9	IDREF	Works fully.
3.3.10	IDREFS	Works fully.
3.3.11	ENTITY	Works fully.
3.3.12	ENTITIES	Works fully.
3.3.13	integer	Works fully except for numeric type promotion.
3.3.14	nonPositiveInteger	Not implemented.
3.3.15	negativeInteger	Not implemented.
3.3.16	long	Not implemented.
3.3.17	int	Works fully except for numeric type promotion.
3.3.18	short	Not implemented.
3.3.19	byte	Not implemented.
3.3.20	nonNegativeInteger	Not implemented.
3.3.21	unsignedLong	Not implemented.
3.3.22	unsignedInt	Not implemented.
3.3.23	unsignedShort	Not implemented.
3.3.24	unsignedByte	Not implemented.
3.3.25	positiveInteger	Not implemented.
3.3.26	yearMonthDuration	Works fully. This is an XPath data type (not Schema).
3.3.27	dayTimeDuration	Works fully. This is an XPath data type (not Schema).

Bibliography

[1] Apache Software Foundation. Xerces2 Java Parser. http://xml.apache.org/xerces2-j/.

[2] E. Gamma. JUnit. http://www.junit.org/.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] S. E. Hudson. CUP Parser Generator for Java. http://www.cs.princeton.edu/~appel/modern/java/CUP/.

[5] M. H. Kay. SAXON — The XSLT and XQuery Processor. http://saxon.sourceforge.net/.

[6] G. Klein. JFlex — The Fast Scanner Generator for Java. http://jflex.de/.

[7] B. McWhirter. jaxen — universal java xpath engine. http://jaxen.org/.

[8] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[9] Quest Software. JProbe Suite. http://www.quest.com/jprobe/.

[10] The World Wide Web Consortium (W3C). Document Object Model (DOM). http://www.w3.org/DOM/.

[11] The World Wide Web Consortium (W3C). XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/.

[12] The World Wide Web Consortium (W3C). XML Schema, Parts 0, 1, and 2. http://www.w3.org/XML/Schema.

[13] The World Wide Web Consortium (W3C). XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/TR/xpath-datamodel/.

[14] The World Wide Web Consortium (W3C). XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics/.

[15] The World Wide Web Consortium (W3C). XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/TR/xquery-operators/.