XML manipulation by non-expert users

Gilbert Tekli

Manipulation des Données XML par des Utilisateurs Non-Experts

XML Manipulation by Non-Expert Users

Gilbert M. Tekli

Thèse de Doctorat en Informatique
Ph.D. Dissertation in Computer Science and Software Engineering

Membres du Jury (*Examination Committee*)

Rapporteurs (Reviewers)**:**
Ahmed LBATH – U. J. Fourier, LIG, France
Nhan LE THANH – U. Nice Sophia-Antipolis, Laboratoire I3S, UMR 6070 CNRS, France

Examinateurs (Examiners)**:**
Frederique LAFOREST – UJM, Télécom Saint-Etienne, France
Florence SEDES, U. Paul Sabatier, IRIT - UMR 5505, France

Co-directeur de Thèse (Co-Supervisor): Jacques FAYOLLE – UJM / Télécom Saint-Etienne, France
Co-directeur de Thèse (Co-Supervisor): Richard CHBEIR – U. Bourgogne, LE2I, France

Laboratoire LT2C-SATIN, Université Jean Monnet, Telecom St-Etienne, France

A man's reach should exceed his grasp, or what's a heaven for? - Robert Browning

I dedicate this work first of all to our lord Jesus Christ in whom I found eternal salvation and without whom this work would have never come to be. I would also like to dedicate this dissertation to mom and dad, to my brothers Antoun, Joe and Jimmy, and to my family (grandparents, aunts, uncles, cousins and future parents in-law) in Lebanon and Australia, whose unconditional love and unwavering support were crucial in achieving this work. I dedicate this thesis as well to my fiancée Hanadi, my bestest of friends Lizzy, and my mentor Antoun Daou whom without their perseverance I would not have lasted. I bestow this work in particular to my aunt Yvonne, cousins Layal, Toni and Carla, our guardians in heaven.

Where there is no guidance, a people falls, but in an abundance of counselors there is safety. - Proverbs 11:14

I also dedicate this work to my larger family, both in France and Lebanon, Dr. Chbeir and his family, to my dear friends Taline Boyajian, Toufiq abilameh, Linda Eid, Maroun Khoury, Charbel Mousallem, Christian Nseir, Said Sfeir and Wajdi Dandach, as well as to all the members of the Jesus Sacred Heart Organization (JSHO), Shouf, Lebanon, for their encouragement and firm support.

Acknowledgements

This dissertation would not have come to fruition without the support of many individuals and institutions, and it is with pleasure that I acknowledge their efforts and contributions.

First, I would like to express my gratitude to my professors and academic supervisors Dr. Richard Chbeir of the LE2I Laboratory UMR-CNRS, University of Dijon, and Dr. Jacques Fayolle of the LT2C laboratory, Telecom St Etienne whose successful teachings have made this dissertation possible.

I would like to express my greatest gratitude to Dr. Richard Chbeir, for his close supervision and constant presence during the past three years. I thank him for his patience, support, and generous guidance during the completion of my Doctorate.

Equally, I would like to thank my senior supervisor Dr. Jacques Fayolle, Co-director of Telecom St Etienne, for his guidance and valuable advice during the thesis.

I would also like to thank Dr. Nhan LE THANH, Professor in the I3S Laboratory, University Nice Sophia-Antipolis, France, as member of my examination committee, and Dr. Ahmed LBATH, Professor in the LIG, U. J. Fourier, as member of the examination committee.

My thesis was financially supported by the Satin team from Telecom St Etienne, through a three year doctoral fellowship, to whom I am grateful.

I would like to acknowledge the support of my colleagues in the Satin team and LE2I laboratory, mainly Christophe Gravier, Michael Ates, Jeremy Lardon, Abakar Mahamat Ahmat, Bechara Al Bouna, Elie Raad and Fekade Getahun, as well as the support of my friends, mainly Christiane, Ali and Wajih.

Lastly and most importantly, my deepest gratitude and love goes to our Lord Jesus Christ who filled me with strength, patience and wisdom to finish this work.

Table of Contents

List of Figures

Chapter 5: Prototype and Experiments

List of Tables

Chapter 2: Related Works

Chapter 4: XA2C Approach

Chapter 5: Prototype and Experiments

CHAPTER 1

INTRODUCTION

Table of Contents

1.1 Introduction

Communication is the key element for human evolution in all its domains: social, medical, chemical, commercial, financial, etc. In the 21st century, computers are everywhere. They have invaded our lives and have become the main source of communication, whether they are used in:

- Instant messaging (e.g., people chatting using instant messaging tools such as Gtalk, Msn, Yahoo messenger, Jabber, etc.)
- Social networks (e.g., friends sharing information over Facebook, Flicker, Linkedin, etc.)
- Scientific data management (e.g., colleagues sharing sensitive data such as medical, financial and scientific records, etc.)
- Data protection (e.g., companies exchanging sensitive encrypted data).

XML (Extensible Markup Language), representing textual structured data, is nowadays the dominant data type of communications in the world of computer science whether these communications are text-based, audio-based, image-based or video-based.

What is XML?

XML, defined by the W3C (World Wide Web Consortium) stands for eXtensible Markup Language. Informally speaking, it is a human readable way of describing structured data. XML is a markup language for documents containing structured information. Structured information contains both data (textual) and meta-data (pictures, audio, etc.) A markup language is a mechanism to identify structures in a document. The XML specification defines a standard way to add markup to documents. XML is made up of tags enclosing text where each tag can have zero or multiple attributes and zero or multiple sub elements such as:

```
<XML version="1.0">
<lib>
        <book id="1">
                <author>Charles Dickens</author>
                <title>A Christmas Carol</title>
                <pub_date> 17-12-1843</pub_date>
        </book>
        <book id="2">
                <author>James Joyce</author>
                <title>Ulysses</title>
                <pub_date> 2-2-1922</pub_date>
        </book>
</lib>
```

XML has a similar structure to HTML, nonetheless it does not withhold any presentation information and its tags are not predefined. They are either defined by the user or from a user based-grammar. XML was standardized mainly for data configuration and transfer over the web as well as any other platform.

XML can be either, the carrier of the communicated data, which is the case of textual-based data, or its descriptor, which is the case of audio [85], image [42] and video-based [16] data. Thus, XML has become one of the essential elements in the communication process in and between all areas/fields. Its use goes beyond computer science. Therefore and as a consequence, there is a daily increasing need for manipulating (controlling, altering, filtering, modifying, adapting, obfuscating, etc.) XML-based data (e.g., XML documents or fragments) transferred between different types of users, applications and systems, from different areas (e.g., business, education, computer science, etc.) in different environments (e.g., desktops, laptops, portable devices, etc.). Nowadays, all users, experts and non-experts, need to manipulate their XML data. As the writer Marylin vos Savant had said:

"Email, instant messaging, and cell phones give us fabulous communication ability, but because we live and work in our own little worlds, that communication is totally disorganized."

To better motivate our research, consider the following 5 additional scenarios illustrating different XML data manipulation in different application domains.

1.2 Motivating Scenarios

Consider a media company running different departments locally and internationally (e.g., Reporting department, Publishing department, Communication department, etc.). Different manipulation/control scenarios are required either in a single department or between departments.

1.2.1 Scenario 1: Information Gathering (Data Filtering)

A reporter working in the IT (information and technology) department is writing an article on the guide books which have been published in the year 2001. The reporter wishes to acquire all the information available in the company's library on guide books published in 2001.

To achieve this, one technique would be required:

(a) **XML Filtering***: Filter XML data based on XML value predicates ('guide' and '2001' in this case).*

1.2.2 Scenario 2: News Gathering and Report Generation

A journalist working in the Reporting department is writing an article covering an event. The journalist wishes to acquire all information being transmitted by different media sources (television channels, radio channels, journals, etc.) in the form of RSS

feeds, filter out their content based on the topic (s)he is interested in, and then compare the resulted feeds. Based on the comparison results, a report covering relevant facts of the event will be generated.

To achieve this, several techniques would be required:

(a) **XML Filtering***: Filter XML data provided by several sources having the same structure (RSS Schema) based on a specific topic*

(b) **XML Content Similarity***: Compare the filtered XML data for content similarities and retrieve significant data*

(c) **Automated XML generation**: *Generate an XML file reporting the filtered out XML data.*

1.2.3 Scenario 3: Sensitive Data Obfuscation

The Communication department posts information and news concerning its activities in form of RSS feeds over the internet. The company wants to keep sensitive parts of the information exclusive to its employees and partners. However, the information needs to be partially available worldwide over the internet. In other words, sensitive data in the RSS feeds are to be encrypted by the information provider (the communication department), decrypted by the corresponding readers (employees and partners), and obfuscated for the rest. The feeds should remain RSS standardized.

To achieve this, several techniques would be required:

(a) ***XML Granular Content Encryption and Signature****:*
- *Encrypt and sign part of the data content transmitted in an XML file without altering the structure. (e.g., <description>38SUJujdgxxvES decided to sign the contract on the Wx34zs5sdZD.</description>)*
- *Decrypt the encrypted data by the corresponding users.*

1.2.4 Scenario 4: Messenger Malicious Content Removal

The company runs a messenger service (e.g., Jabber messenger) on its intranet as a communication system between its employees. The messenger communicates via XML structured data. One of the employers wishes to control the communication between his employees by removing all swear words automatically, replacing them with a notification message and removing any sexual content.

To achieve this, several techniques would be required:

(a) **XML Content Search:** *Detect the existence of malicious data in an XML data I/O*

(b) **XML Content Adaptation:** *Based on the data type found, the malicious data must be either replaced with a customized text message or the entire data content must be deleted.*

1.2.5 Scenario 5: Collaborative Presentation Modification

The departments communicate using collaborative presentations based on XML [42]. The employer wishes to put together and analyze the structure and content of the SMIL documents so that he can ensure the presence of each department's logo and inserts the company's logo on all slides without overlapping. The SMIL documents provided from each department may have different structures.

To achieve this, several techniques would be required:

(a) **XML Structural and Content Search:** *Search for sensitive data in the structure and data content of XML data collected from different sources*

(b) **XML Content Modification:** *Ensures that each department's logo exists and inserts the company's logo.*

These scenarios present the following issues:

What are the main issues to be solved?

1. **Data types**
   Data to be manipulated is XML-based
2. **Manipulation operations**
   Different and separate techniques are required to fulfill the manipulation operations which can vary between (but is not limited to):
   - XML data selection/projection, insertion/removal and modification (e.g., XML element retrieval, insertion, etc.)
   - XML value selection/projection, insertion/removal and modification (e.g., XML textual value extraction, deletion, etc.)
   - XML syntax and semantic filtering
   - XML data restructuring
   - XML data protection (i.e., XML data obfuscation/omission).
3. **User profiles**
   Users are not necessarily expert programmers (e.g., a journalist) and thus require intuitive interfaces
4. **Platforms**
   Data is being communicated over different environments and platforms (e.g., internet, user machines and intranet).

Consequently, providing non-expert users with means to create and execute manipulation operations over XML data is becoming more and more crucial. In the literature, there has not been a unified solution addressing these matters

simultaneously. Nevertheless, several approaches exist addressing the "XML manipulation by non-expert users" subject from separate point of views.

1.3 Related Work

Four main categories are provided: (i) XML-oriented Visual languages, (ii) Mahsups, (iii) XML manipulation via security and adaptation techniques, and (vi) Dataflow visual languages.

1. **XML-oriented Visual Languages:**

These languages have been developed mainly for non-expert users, allowing them to visually query XML data. Several languages exist, such as XML-GL [22], Xing [40], XQBE[15] and VXT[88], providing users with means to visually create their selection/projection queries over XML-based data. While these languages target non-expert users, they require knowledge in data querying and are limited in their manipulation to XML data extraction and structural transformation and do not provide XML data modification (i.e., insertion/update etc.) and/or value manipulations.

2. **Mashups**

Mashup tools have been developed recently to manipulate web data by non-expert users. In this category, 2 tools in particular have been developed allowing the manipulation of XML data, YahooPipes [73] and IBM Damia [93]. These tools are based on the functional composition paradigm, considered to be the closest to the natural human thinking process, where the user creates his manipulation operation by simply linking different functions (modules) together. This is simpler than the query paradigm and does not require any level of expertise. Nevertheless, these tools are limited in their manipulations mainly to XML data/value extraction and transformation. In addition, they are dedicated for web applications. Thus, they have limited expressiveness and do not allow the manipulation of offline data (available on user machines).

3. **XML manipulation techniques**

These techniques are defined originally to provide different methods for adapting XML data to different platforms and systems, and to secure sensitive XML data by means of encryption, access control, etc. In this category, different techniques have been defined under the adaptation approach such as XML filtering [75], adaptation [71, 84] and information extraction [23, 27], and under the security approach such as XML access control [29], usage control [83], encryption and signature [58], firewalls [109], etc. These techniques defined the main manipulation operations (i.e., data

selection/projection, filtering, modification, etc.). Nevertheless, these techniques are defined each separately and require each a high level of expertise to implement it.

4. **DataFlow Visual Programming Languages (DFVPL)**

DataFlow Visual Programming Languages or DFVPLs are essentially developed for non-expert users, mainly scientists, allowing them to manipulate scientific data by means of visual compositions. They follow the Dataflow paradigm, mainly based on functional composition. Different DFVPLs have been defined such as DFL [53], V [10] and Taverna [80]. They provide a well-defined visual syntax allowing non-expert users to create their manipulation operations. Nonetheless and to the best of our knowledge, DFVPLs have not yet been adopted in XML data manipulations.

Since none of the existing approaches/techniques solves the issues (cf. page 16) addressed here, our research mainly aims at defining a derivable XML-oriented framework allowing non-expert users to write/draw and enforce XML manipulation operations based on functional composition. The functions can:

- express any type of manipulations satisfying personal user requirements or security requirements
- be provided in forms of local libraries (e.g., DLL files) or online services (e.g., web-services).

1.4 Proposal and Main Contributions

The framework, called XA2C (XML-oriented mAnipulAtion compositions), is defined as a modular architecture with 3 main modules: (i) the language platform, (ii) the compiler, and (iii) the runtime environment. A prototype, called X-Man, is developed and used to assess our approach. The XA2C approach was published in [97].

1.4.1 Language Platform

The language platform defines formally a DFVPL for manipulating XML-based data. The language, called XCDL (XML-oriented Composition Definition Language), is defined mainly as a visual functional composition language based on the Dataflow paradigm since it is the closest to the natural human thinking process [11, 60]. Its syntax and semantics are based, on one hand, on Colored Petri Nets (CP-Nets) [61, 79] which allow expressing complex compositions with true concurrency (combined serial and parallel executions), and, on the other hand, on OLT (Ordered Labeled Trees) allowing the formal representation of I/O XML-based data. As for the manipulation operation composition, it is denoted by mapping the output of a function to the input of another. The functions are identified in the language library as *SD-functions*

(System-Defined functions) from either offline libraries (i.e., DLL files) or online libraries (i.e., web services). The language platform was published in [95].

1.4.2 Compiler

The compiler is formally defined as a middleware between the language platform and the runtime environment. The compiler validates the compositions' syntax, optimizes (e.g., remove passive transitions) and translates them from high-level petri nets (similar to high level programming languages) as defined by the XCDL syntax into XML-based petri nets (similar to machine code) executable in the Runtime Environment.

1.4.3 Runtime Environment

The Runtime Environment defines formally the execution environment for the resulting compositions translated from high-level petri nets into XML-based petri nets. The Runtime Environment contains 2 execution modes: (i) serial execution, executing one function at a time and (ii) concurrent execution, executing functions in parallel when all dependencies are resolved. The execution modes are generated from a execution step discovery algorithm defined based on the petri nets firing rule [79] and incidence matrix [79]. The algorithm generates from the XML-based petri net a serial and concurrent execution sequences which can be respectively executed on a single processor machine and a multi-processor machine depending on the machine type. The Runtime Environment was published in [96].

1.4.4 Prototype and Evaluation

To validate and evaluate our approach, we developed a prototype, called X-Man (XML mAnipulAtions) developed in visual studio. X-Man was tested in different case studies with a number of participants in order to evaluate XCDL and assess its usability/performance with regard to existing approaches such as YahooPipes [73] and IBM Damia [93]. In addition, we tested the execution step discovery algorithm of X-Man in different scenarios (i.e., serial, parallel and concurrent compositions).

1.5 Thesis Organization

The rest of the thesis is organized as shown in Figure 1. Chapter 2 defines the initial criterions related to XML manipulation by both experts and non-experts, and discusses existing approaches and techniques (i.e., XML visual languages, Mashups, XML manipulation techniques and DFVPLs). Since our approach is DFVPL-based, Chapter 3 provides some background regarding Dataflows. In Chapter 4, we detail the XA2C (XML mAnipulAtion composition framework) approach. Here, the XCDL

specifications and syntax are also defined along with the compiler and the runtime environment. In Chapter 5, we present our prototype (X-Man) as well as a VPL (visual programming language) evaluation framework that we designed. We also present here the set of case studies conducted to evaluate the XA2C approach. Finally, Chapter 6 concludes this study and provides some future research tracks.

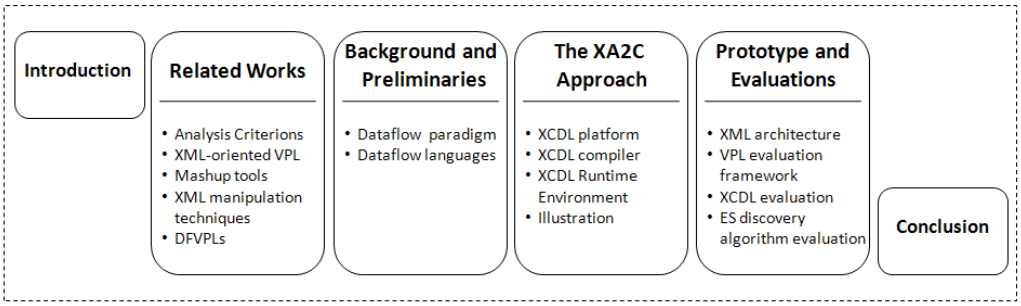


Figure 1: Thesis Structure

CHAPTER 2
RELATED WORKS

XML manipulations in multiple domains have been the focus of many researchers over the years. Whether adapting XML data to different platforms, filtering it for data management, encrypting it for protection purposes, signing and watermarking it for privacy reasons or modifying/transforming it for user requirement satisfaction, XML manipulation has become a wide phenomena due to the increasing use of XML. Nowadays, since XML has become one of the most essential data types used in computer communications, its widespread has crossed over the boundaries of computer science domains and reached other areas such as medical (e.g., medical record storage), mechanical (e.g., graphical map design), social (e.g., instant messaging), commercial (e.g., publicity communication), financial (e.g., online payment) and others. This has brought a new criterion into the XML manipulation research field, XML manipulation by non-experts. In this chapter, we study and analyze existent techniques for manipulating XML from a non-expert point of view while relating it to traditional manipulation techniques defined in the literature such as filtering, adaptation, data extraction, transformation, access control, encryption, etc. XML manipulation techniques by non-experts were categorized under 3 major titles: (i) XML-oriented visual languages dealing with XML data extraction and transformations, (ii) Mashups tackling mainly XML restructuring with value manipulations, and (iii) Dataflow visual programming languages targeting non-experts and providing them with means to visually manipulate scientific data. A full analysis was conducted which allowed existent approaches/techniques to be compared and resulted in an overview of the current requirements of this subject.

Table of Contents

2.1 Introduction

The widespread of XML today has invaded the world of computers and is present now in most of its fields (i.e., internet, networks, information systems, software and operating systems). Furthermore XML has reached beyond the computer domain and is being used to communicate crucial data in different areas such as e-commerce, data communication, identification, information storage, instant messaging and others. Therefore, due to the extensive use of textual information transmitted in form of XML structured data, it is becoming essential to allow all kind of users to manipulate corresponding XML data based on specific user requirements. As an example, consider a journalist who works in a news company covering global events. The journalist wishes to acquire all information being transmitted by different media sources (television channels, radio channels, journals …) in the form of RSS feeds, filter out their content, based on the topic (s)he is interested in, and then compare the resulted feeds. Based on the comparison results, a report covering relevant facts of the event needs to be generated.

In this first simple scenario, several separate techniques are required to generate the manipulation operation required by the user such as XML filtering, string similarity comparison and automated XML generation. In a second scenario, consider a cardiologist who shares medical records of his patients with some of his colleagues and wishes to omit personal information concerning his patients (i.e., name, social security number, address, etc.). In this case, data omission is the manipulation required which can be done via data encryption, removal, substitution or others depending on the operations provided by the system and the requirements of the user (cardiologist in this case).

Based on these scenarios: (i) we need a framework for creating XML-oriented manipulation operations. It should contain all of the XML-oriented manipulation techniques. To the best of our knowledge, such a framework does not exist so far, and (ii) we need the framework to target non-expert users (e.g., scientists, businessmen, novice programmers, etc.).

As discussed in these scenarios, (i) manipulated data is XML-based (ii) separate and several manipulation techniques are required varying between simple data selection/projection, filtering, data restructuring and securing data (i.e., XML filtering, string similarity comparison, XML transformation and data obfuscation/omission, watermarking, etc.), (iii) targeted users are non-expert programmers (e.g., journalist, cardiologist), and (iv) data is circulating between different environments and platforms (e.g., internet, local machines and local networks).

Even though, these issues are progressing more and more, in the literature, there has not been a unified solution addressing these matters simultaneously. Nevertheless, several approaches exist addressing the "XML manipulation by non-expert users" subject from different perspectives such as:

(a) **XML Querying Visual Languages** developed mainly for non-expert users allowing them to visually query XML data

(b) **Mashup tools** developed recently for non-expert users to manipulate web data

(c) **XML security and adaptation techniques** defined originally to provide different techniques for adapting XML data to different platforms and systems, and to secure sensitive XML data by means of encryption, access control, etc.

(d) **DFVPLs (Dataflow Visual Programming Languages)** essentially developed for non-expert users, mainly scientists, allowing them to manipulate scientific data by means of visual compositions.

In this study, we discuss these techniques and approaches regarding "XML manipulation by non-expert users". These approaches are analyzed based on different criterions regarding the subject at hand such as expressiveness, visualization, formalization, expertise and others. This paper summarizes their advantages and drawbacks with regard to the issues mentioned here.

The rest of this chapter is organized as follows. In Section 1, we give some preliminaries and analysis criteria. The second section presents different XML Querying Visual Languages. Section 3 discusses the Mashup approach with different XML-oriented Mashup tools. We present different XML security and adaptation techniques in Section 4. Section 5 discusses the Dataflow paradigm and describes different formalisms of DFVPL. And finally, we conclude and discuss the effect of these approaches on the "XML manipulation by non-expert and expert users" paradigm.

2.1.1 Preliminaries and Analysis Criteria

Being that the "XML manipulation by non-expert and expert users" subject has not been discussed in the literature previously, no analysis criterions have been identified so far concerning this matter. Therefore, we propose some analysis criteria allowing the evaluation of existing approaches, defined in Table 1, regarding the issues identified in the previous scenario. The evaluation criteria are grouped in 4 main analysis categories identified with regard to the 4 main perspectives underlined by "XML manipulation by non-expert users": (i) manipulated data, (ii) manipulation operations, (iii) interaction/visualization and (iv) derivability. These criteria are discussed in the following section are detailed in Table 1.

2.1.2 Manipulated Data

The techniques/approaches need to be XML oriented, target online and offline data (since the data can be user-defined or web-based). Thus, the data manipulated should be XML-based whether it is located online or offline and the manipulated data is identified as an analysis category with the following criteria (cf. Table 1):

- XML-based
- Web-based
- User-based
- Target offline data (stand alone architecture)
- Target online data (client server architecture)

2.1.3 Manipulation Operations

The expressiveness of a technique/approach is essential in order to define the manipulation operation that can be provided, whether it is used for security or adaptation purposes, such as XML data selection, projection, insertion and modification. Provided that XML is structured and text-based, it is imperative to check whether a technique/approach can manipulate textual values (XML textual values) as well as structural values (XML elements and attributes). Therefore, the manipulation operations are considered an analysis category which contains the following criterions but is not limited to: (cf. Table 1)

- selection/filtering
- projection/transformation
- insertion/removal
- modification/protection/obfuscation

2.1.4 Interaction/Visualization

On one hand, seeing that the manipulations need to be created by non-expert programmers, it is essential to denote if a technique/approach is defined with the aid of visual representations and thus can be used by non-expert users (e.g., a journalist and a Cardiologist),. On the other hand, it is also important to note if a technique/approach requires a user to be an expert and/or have some knowledge in programming. Allowing users to integrate their created manipulation operations and being able to reuse them is an essential criterion as well as determining if a technique/approach is based on the functional composition paradigm, as in manipulation operations are built by simply mapping different modules/functions together which is the closest paradigm to the human natural thinking process.

Thus human/machine interaction and system/data visualizations are defined as an analysis category and regroup the following criteria: (cf. Table 1)

- Composition-based operations
- Programming knowledge
- Expertise
- Reusability
- Formalized/intuitive visual syntax
- Expressiveness

2.1.5 Derivability

Since the data can be used on different platforms and environments, it is important to specify whether a technique/approach has been formally defined and can be re-implemented, and if it is defined as a language allowing the users to write their manipulation operations. Also, it is important to note the extensibility of a technique/approach, if it can be extended with new features/operations. As a result, the solutions need to be derivable and their architectures should be flexible and adaptable. Consequently, we identify derivability as an analysis category containing the following criterions: (cf. Table 1)

- Formalized approach/technique
- Formalized Language
- Extensibility

The defined analysis criterions are detailed in Table 1.

Table 1: Analysis criteria

Category	Sub- category	Criteria	Description
Manipulated Data	**Type**	XML-specific	Specifies whether a technique or an approach is oriented towards XML and deals with the particularities of XML structured data.
		Web-based	Determines if the data is web-based (e.g., HTML, RSS, etc.)
		User-based	Determines if the data is defined by the user (e.g., scientific data, graphs, etc.)
	Location	Target offline data (stand alone)	Denotes that a technique/approach can manipulate offline data, from user computers.
		Target online data (client/server)	Determines if a technique/approach can manipulate data from the internet, not stored on the user machine.
Manipulation	**Structural**	Selection/filter	Indicates that the technique/approach can

Operations		ing	provide XML data selection/extraction
		Projection/transformation	Indicates that the technique/approach can provide XML data restructuring/transformation.
		Insertion/removal	Denotes that a technique/approach allows for data removal or new data insertion.
		Modification (obfuscation)	Denotes that a technique/approach allows for existing XML data to be updated/modified. The modification can be viewed as protection such as in the case of data obfuscation.
	Content (textual)	Selection	Indicates if a technique/approach can implement selection queries over textual data
		Insertion/removal	determines whether a technique/approach can implement insertion/deletion queries over textual data
		Textual manipulations	Specifies whether a technique/approach can provide manipulations over XML textual values (e.g., update, obfuscation, signature, etc.).
Interaction/ Visualization	**User**	Programming background	Designates that a technique/approach requires the user to have some knowledge in programming in order to be able to define the manipulation operations.
		Expertise required	Designates that a technique/approach requires the user to be an expert in it in order to be able to define the manipulation operations.
	System	Composition-based	Denotes that a technique/approach is based on simple composition which is the closest paradigm to the human thinking.
		Query-based	Designates that a technique/approach follows the query paradigm
		Reusable	Denotes that created manipulation operations can be reused by others.
		Formal Visual syntax	Signifies that a technique/approach is defined as a formal visual language and its visual representation is well defined (visual representations are essential for non-expert users).
		Expressiveness	Defines the expressiveness power of a technique/approach to manipulate XML data.

Derivability		Formalism	Specifies whether a technique/approach has been formally defined and can be implemented on different platforms.
		Formal language	Indicates that a technique/approach is defined as formal language (formal languages can be implemented to provide the user with means to write their manipulation operations).
		Extensibility	Designates that the technique/approach can be extended with new features/operations.

The following sections will discuss different approaches and techniques related to "XML manipulation by non-expert programmers". To the best of our knowledge, so far there has not been any unified approach resolving the issues discussed in this paper, therefore each technique/approach is presented from its own angle and point of view on the subject such as XML visual languages from the data extraction and restructuring by non-experts point of view, Mashups from the web data manipulation by non-experts point of view, XML security and adaptation techniques from the XML manipulation operations point of view and DFVPL from the data manipulation by non-experts point of view.

2.2 XML Query and Transformation Visual Languages

Since the standardization of XML and its widespread beyond the computer domain, researchers have been trying to provide XML-oriented visual languages allowing the querying of XML data since the existing textual languages (such as XQuery [104], XPath [103] and XSLT [66]) are complicated and require a high level of expertise. These visual languages are mainly extensions of existing approaches such as XML query languages and transformation languages. Their main contribution is to allow non-expert programmers to extract sensitive data from XML document and restructure the output document.

As detailed in the subsections, several languages have been developed over the years such as XML-GL [22], Xing [40], XQBE [15] and VXT [88]. On one hand, Xing and XML-GL were developed before XQuery was standardized and took the SQL querying approach by following the 3 main components of a regular query: selecting, filtering and restructuring the data. XQBE was developed after XQuery has been standardized and, therefore, is based on it. Its expressiveness is greater than previous approaches whereas it allows the creation of complex queries containing aggregation functions, ordering results and negation expressions. Nonetheless, its expressiveness is still limited to data extraction and query reconstruction in XQuery and does not

include textual data manipulation operations. VXT, on the other hand, was based on XSLT [102] which is mainly used for XML data restructuring without any textual data manipulation, nor data insertion nor modification.

From the visual aspect, all of these approaches followed the same pattern. They divide the workspace to 2 main sections, left and right. The left section constitutes the source file with the extraction rules and the right section constitutes the result file. The query is defined by mapping the element to be extracted from the left section to the element to be constructed in the right section as shown in Figure 1.

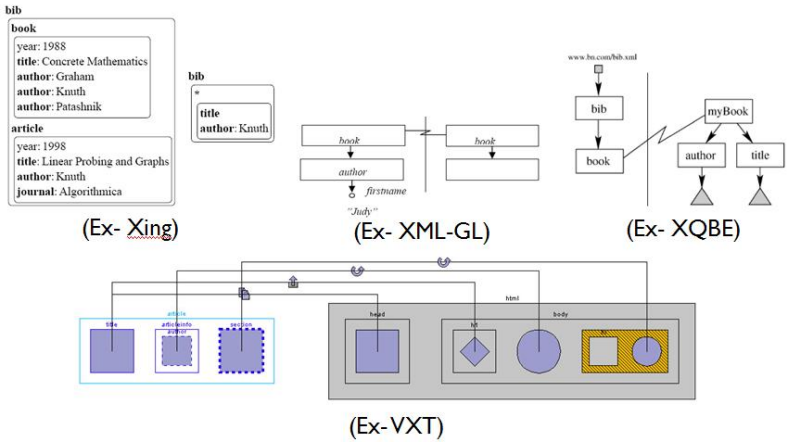


Figure 1: XML query visual languages

The existing visual languages successfully bridged the gap between the complexities of XML data querying and non-expert programmers. However, they were limited only to data extraction, filtering and restructuring. Mainly they provided non-expert programmers with the ability to create XML structural transformations along with data extraction and filtering. They did not address the textual data manipulation issue and XML data insertion and modification (cf. Table 1). The main languages are discussed here below. The following query and XML document is used in the illustrations of the XML query visual languages.

Query 1:

Select all the books from books.xml that have been published in the year 1983

XML document books.xml:

```
<XML version=1.0>
<lib>
        <book>
                <author>Charles Dickens</author>
                <title>A Christmas Carol</title>
                <pub_year>1983</pub_date>
        </book>
        <book>
                <author>James Joyce</author>
                <title>Ulysses</title>
```

```
            <pub_year>1922</pub_date>
            <description>An epic Greek myth.</description>
        </book>
</lib>
```

2.2.1 XML-GL

XML-GL [22] was defined by the World Wide Web consortium (W3C) as a graphical language for querying and restructuring an XML document. XML-GL represents XML documents as labeled graphs [21] and thus aims at being user friendly.

An XML-GL graph is defined formally as a connected, paired and directed graph defined by 2 sets *N* and *A*.

- *N* is a set of nodes representing XML components (e.g., Elements and Attributes). These nodes are divided into 2 disjoint sets *E* and *P*. *E* is the set of XML Elements represented by labeled rectangles, with their tag names as labels. *P* is a set of properties defined by the sets A_t and *C*. A_t defines the set of attribute nodes represented by solid circles and *C* is the set of content nodes represented by hollow circles.
- *A* is a set of labeled arcs represented by directed arrows from *n* to *n'*, where *n* is the source node and *n'* the destination node.

As shown in Figure 2, a query, in XML-GL, is represented by 2 XML-GL labeled graphs separated by a vertical line. The graph on the left side is the source graph and the one on the right is the destination graph. The 2 graphs are linked together by an explicit binding (the line linking a source node with a destination node). The source graph represents the selections to be made from the source XML file. As for the destination graph, it represents the output structure of the executed query. The binding maps the source elements being queried to the output structure being projected.

In Figure 2, we can see an example of query 1 over books.xml where the user wishes to extract all books published in 1983.

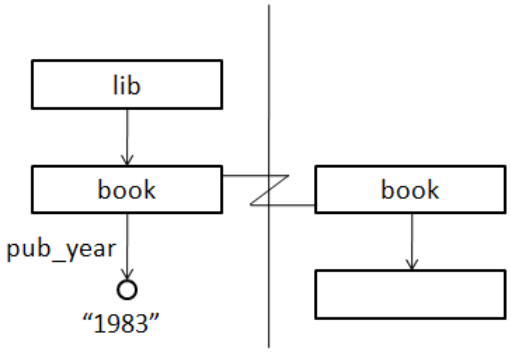


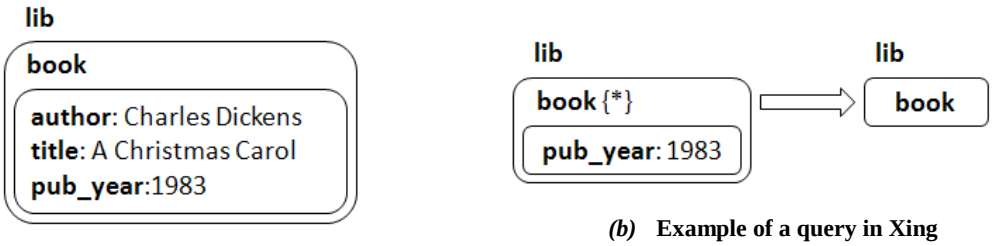
Figure 2: XML-GL query example

The XML-GL query paradigm is based on the "SELECT <attributes> FROM <tables> WHERE <conditions>" query from the SQL language, and thus is limited to data selections and projections.

XML-GL was one of the first graphical querying languages designed for XML documents. The main purpose was to provide users, mainly non-expert programmers, with the ability to restructure and extract sensitive data from XML files. Nonetheless, due to the limitations provided by the existing querying languages at the time, in this case SQL and in particular SQL selection queries, XML-GL's queries were very limited. Like most existing XML-oriented visual languages, XML-GL lacks the ability to manipulate string data, data insertion and update, and is limited to the expressiveness of the query language it is based on. And since it uses the querying paradigm, therefore, the task is rendered more difficult for non-expert programmers seeing that they are required to have some knowledge in querying data.

2.2.2 Xing (XML in graphics)

Xing was conceived as a visual querying and restructuring language for XML documents. Similar to XML-GL, Xing aims at extracting and restructuring XML data by using selections and projections. The main difference between Xing and XML-GL is the representation of the XML data by boxed patterns instead of graphs. As for the rest, it follows the same querying paradigm of XML-GL where a query is represented by 2 patterns, as depicted in Figure 3, one on the left for data selection, called the argument pattern, and one on the right for data reconstruction, called the result pattern. The argument and result patterns are linked together via a binding represented by an arrow directed from left to right.



(a) Example of a Xing expression

(b) **Example of a query in Xing**

Figure 3: Xing, XML in graphics

In Xing, an element is represented as a hybrid textual/graphic expression in a box with the element tag written above it (cf. Figure 3.a). Sub-elements and attributes are written within the borders of the parent element box. Sub-elements are differentiated from attributes by having their tag names written in bold as shown in Figure 3.a. Simple elements without any sub-elements are represented textually with their tag names in bold followed by their values in regular font separated by semi-colons. As for the order of elements, it is represented by their vertical positions.

A simple selection query is represented by a document pattern written/drawn with a Xing expression. As an example, we can see in Figure 3.b the query 1 written in Xing for extracting all books published in 1983.

Xing was defined formally as a visual representation for querying XML data by following the selection projection paradigm. It is defined conceptually based on the SQL selection querying paradigm. Even though it is called XML in graphics, nonetheless it is not based only on visual representations but on textual as well in tabular forms. Similar to XML-GL, its expressiveness is limited seeing that it is based on an existing language. In terms of XML data manipulation, it is only concerned with data extraction and restructuring, no textual manipulations, insertions nor modifications (e.g., updates) exist. Since Xing is based on the SQL paradigm, some knowledge in querying is required.

2.2.3 XQBE (XML Query by Example)

XQBE [15] is an XML-oriented graphical query language defining formally a visual syntax for querying XML data. The main objective of XQBE is being easy to use by non-expert programmers and directly mappable to XQuery. Due to the complexities of XQuery and the need for XQBE to be easy to use, the language was limited to simple querying.

As shown in Figure 4, XQBE is divided to 2 directed graphs, the source graph (on the left) and the construction graph (on the right) separated by a vertical line in the

middle. The source graph represents a pattern matching for the source XML file to be queried and transformed into the structure represented by the construction graph.

Similar to XML-GL, XML elements are represented by rectangles labeled with the elements' tag name. The attributes are represented as black circles with their names drawn on the arc linking the element with its attributes. If an element contains PCDATA, the data is represented by an empty circle with the data value drawn underneath it. To represent the hierarchy between elements, directed arcs are used.

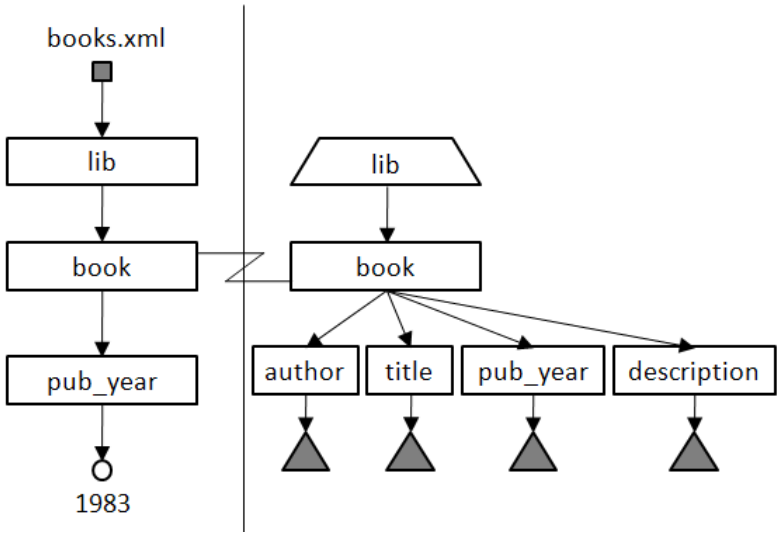


Figure 4: Querying in XQBE

The visual paradigm of XQBE was adopted so that the transformation may have a natural reading order from left to right. Correspondence between elements is represented by an explicit binding, as shown in Figure 4, between the source element and its corresponding node in the construction graph.

The core primitive transformations provided by the language are selection, iteration and projection which are denoted by the source graph, the binding edges and the construction graph respectively. The selection process is executed by evaluating the structure constraints shown in the source graph. The iteration takes place on the nodes of the source graph mapped to the construction graph. Projections are executed with respect to the constraints provided by the construction graph, which may remove or insert new nodes to the queried source node. Figure 4 illustrates query 1 in XQBE. The source graph defines the structure for matching all the book elements with an attribute *year* equal to 1983. The result is an XML fragment satisfying the structure of the construction graph with a root element *lib*, a child element *book* with its sub-elements *author*, *title*, *pub_year* and *description*.

XQBE is a formal visual querying language with a formally defined graphical user interface. Since XQBE is based on the syntax of XQuery, it therefore inherits its

limitations and complexities. On one hand, the manipulation is limited to selections and projections which is useful for data transformation and extraction and does not allow any data adaptation in terms of insertion and update or textual manipulations. On the other hand, it inherits the expressiveness of XQuery but limits its use due to visual constraints and can only be used for simple querying. And since it follows the query paradigm, therefore programming knowledge in querying is required for all users, experts and non-experts.

2.2.4 VXT: Visual XML Transformation Language

VXT is a visual language designed mainly to simplify XML transformations. XML transformations are normally done using the XSLT language which is the most expressive language for transforming XML files. Nonetheless, XSLT is a very complicated language and requires a certain level of expertise in order to use it. Therefore, Pietriga in [88], defined VXT as a visual language based on the XSLT language by providing some graphical elements formally defined and constituting a visual syntax which is translated into an XSLT syntax for rendering.

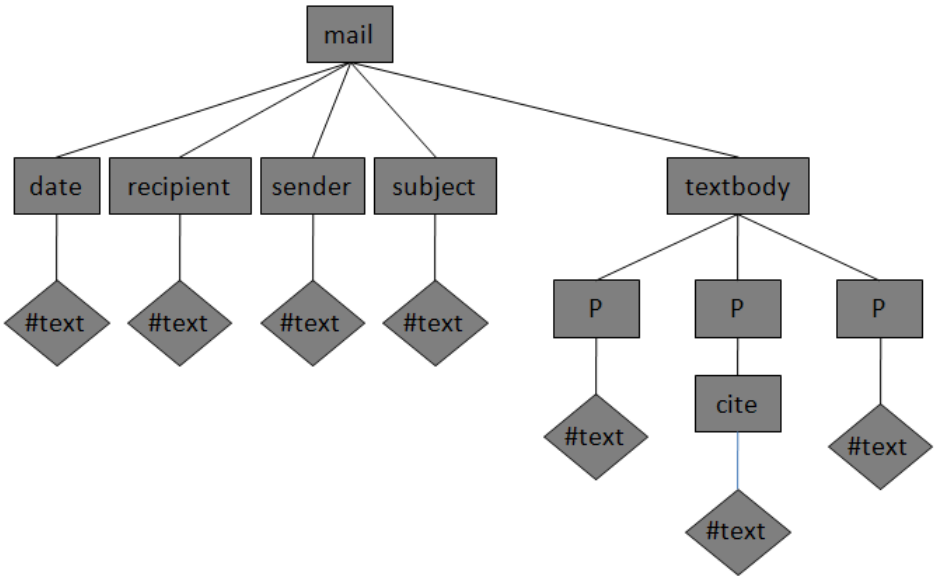


Figure 5: XML view as a treelist

VXT's main contribution was the adoption of treemaps [88] (cf. Figure 6) for XML data representation instead of tree lists (cf. Figure 5).

Pietriga [88] argued that tree lists require a large amount of space and become difficult to read as the XML document structure grows and becomes more complex. Therefore, VXT adopts treemap views which represent XML documents in a more compact space than tree list as shown in Figure 6. Nonetheless, treemaps require additional

computing when it comes to complex structures. A zoom function is required to view complex sub-elements.

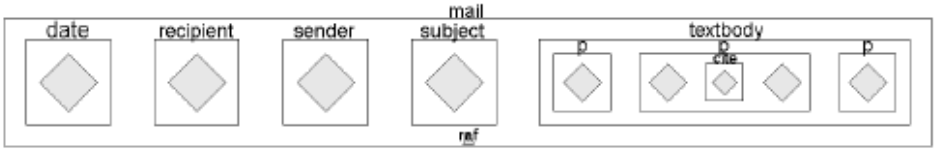


Figure 6: XML view as a treemap

Similarly to Xing, VXT uses pattern matching based on treemaps to draw selections and projections. Based on a similar approach, VXT draws 2 patterns, a source pattern and a construction pattern (destination pattern similar to the source and destination graphs). But since VXT is based on XSLT, therefore, instead of binding nodes by a simple mapping, VXT introduces 3 transformation rules as show in
Table 2.

Table 2: VXT transformation rules

Operation	Image	Input	Output
Copy of node (xsl:copy)		Element Attribute Text	Element Attribute Text
Text extraction (xsl:value-of)		Element Attribute Text	Text Text Text
Apply rules (xsl:apply-template)		Element Attribute Text	Unknown Unknown Unknown

The transformation rules are representations of transformation rules defined in XSLT and thus, are related to structure transformations such as copying a node, extracting textual values or applying a template to a fragment of XML. Figure 7 illustrates an example of creating a reply email template using the 3 rules provided by VXT. This transformation template copies the sender's and recipient values of the received email respectively to the recipient and sender elements of the reply email. The subject and textbody are transformed in the reply email based on predefined templates.

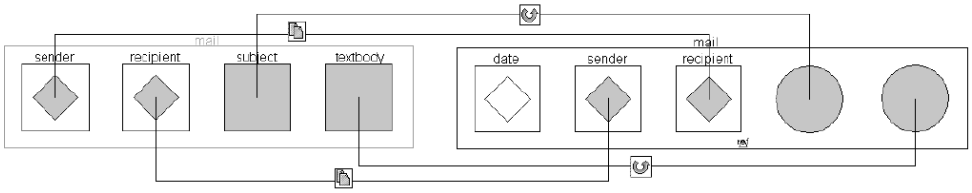


Figure 7: Creating a reply email template

To summarize, VXT tries to express selection projection queries similar to other XML visual languages but from a different perspective. VXT dropped the idea of building its visual syntax on a querying language and went towards a transformation language instead, so that it can be more expressive by introducing some transformation rules. VXT tackled the XML graphical representation issue and adopted the treemap view approach which optimizes the space required for viewing XML structures. Nevertheless and in terms of XML data manipulation, VXT remains limited to data extraction and transformation. No textual manipulations nor data insertion nor update are possible.

2.2.5 Discussion

After presenting the main XML visual languages (XML-GL, Xing, XQBE and VXT), an analysis based on the criteria defined in Table 1 is shown in Table 3.

Based on existing querying and transformation languages, several XML-oriented visual languages emerged and were formally defined. Their main goals were XML data extraction and structural transformations. However, these languages were very limited in their expressiveness mainly due to the graphical constraints and to the languages they are based on.

Table 3: Analysis regarding XML query visual languages

Category	Sub-category	Criteria	XML-GL	Xing	XQBE	VXT
Manipulated Data	**Type**	XML-specific	Yes	Yes	Yes	Yes
		Web-based	-	-	-	-
		User-based	-	-	-	-
	Location	Target offline data	Yes	Yes	Yes	Yes
		Target online data	Yes	Yes	Yes	Yes
Manipulation Operations	**Structural**	Selection/ filtering	Yes	Yes	Yes	Yes
		Projection/ transformation	Yes	Yes	Yes	Yes
		Insertion/ removal	-	-	-	-
		Modification (obfuscation)	-	-	-	-
	Content (textual)	Selection	-	-	-	-
		Insertion/ removal	-	-	-	-
		Textual manipulations	-	-	-	-
Interaction/ Visualization	**User**	Programming knowledge required	Yes	Yes	Yes	Yes
		Expertise required	Low	Low	Low	Low
	System	Composition-based	-	-	-	-
		Query-based	Yes	Yes	Yes	Yes
		Reusable	-	-	-	-
		Formal Visual syntax	Yes	Yes	Yes	Yes
		Expressiveness	Low	Low	Low	Low
Derivability		Formalism	Yes	Yes	Yes	Yes
		Formal language	Yes	Yes	Yes	Yes
		Extensibility	-	-	Limited	-

As shown in Table 3, the languages did not provide means for textual manipulation, data insertion nor data modification. Last but not least, even though the languages

targeted non-expert users, they required some knowledge in data querying and XML data querying in particular.

2.3 XML-oriented Mashups

Mashup is an emerging web application development approach providing users with means to gather and aggregate multiple services, executing each a specific task, and thus creating a new service having its own specific task to perform. Mashup tools are built on the idea of reusing and combining existing services by novice programmers, therefore a graphical interface is generally offered to the user to express most operations. Mashup applications [39, 74, 93] can include but are not limited to:

- Mashups with maps where the objective is to plot various data on a map like Google Map
- Mashups using multimedia content imported from YouTube, Flicker, etc.
- Mashups using e-commerce services such as Amazon.com or Ebay are also flourishing
- The most popular example of Mashups is the feeds Mashups, which subscribe to regular data feeds, typically in RSS or ATOM format, to access data such as news, blogs content, catalog updates, etc.

So far and to the best of our knowledge, the Mashup approach hasn't been formally defined, nevertheless, based on the existing Mashup tools, a preliminary common architecture is elaborated [73]. The Mashup architecture was defined from 3 main criterions:

- Integration between the different types of data (data flow)
- Communication with the components and interaction among them
- Displaying of the content to the end-user.

Therefore, 3 main components were defined:

(a) Data Mediation Level: consists of all possible data manipulations (conversion, filtering, format transformation, etc.) needed to integrate different data sources where each manipulation could be done by analyzing both syntax and semantics requirements.

(b) Process Mediation Level: defines the choreography between the involved applications. The integration is done at the application layer and the composed process is developed by combining functions, generally exposed by the services through APIs.

(c) Presentation Level: is used to extract user information as well as to display intermittent and final process information to the user. Results to the user can be drawn as a simple HTML page, or a more complex web page developed with

Ajax, Java Script, etc. The languages used to implement user interface components and the front-ends visualization support both server side and client-side approaches. But due to the cross-domain problem, using server-side approach such as ASP or JSP is inevitable.

Several Mashup tools have emerged such as YahooPipes [73], Damia [93], Popfly [74], Apatar [73] and MashMaker [39].

- Damia and YahooPipes are mainly designed to manipulate Data Feeds such as RSS feeds
- Popfly is used to visualize data associated to social networks such as Flicker and Facebook. Popfly is a framework for creating web pages containing dynamic and rich visualizations of structured data retrieved from the web through REST web services
- Apatar helps users join and aggregate desktop data such as MySQL, Oracle, PS SQL and others with the web through REST web services
- MashMaker is used for editing, querying and manipulating data from web pages. Its goal is to suggest to the user some enhancements, if available, for the visited web pages.

In this study, our interest mainly falls on YahooPipes and Damia seeing that they allow manipulations of XML-based data and they are based on functional compositions instead of the querying paradigm used by the other tools. As for the other tools, on one hand, they are not XML-oriented and from the other hand, they are based on the query paradigm which has been argued in the XML query visual languages' section, that the tools following the query paradigm have limited operations and are considered more complex for non programmers due to the fact that some knowledge is required for querying data. Thus they are excluded from this study.

2.3.1 YahooPipes

YahooPipes [73] is initially a Mashup tool built on upon an RSS-based data model. Its main purpose is to manipulate and aggregate data feeds from different web sources (e.g., web feeds, Web pages, RSS feeds, etc.). YahooPipes allows users to create manipulation operations by providing modules which can be mapped to one another and thus creating a composed manipulation operation (cf. Figure 8).

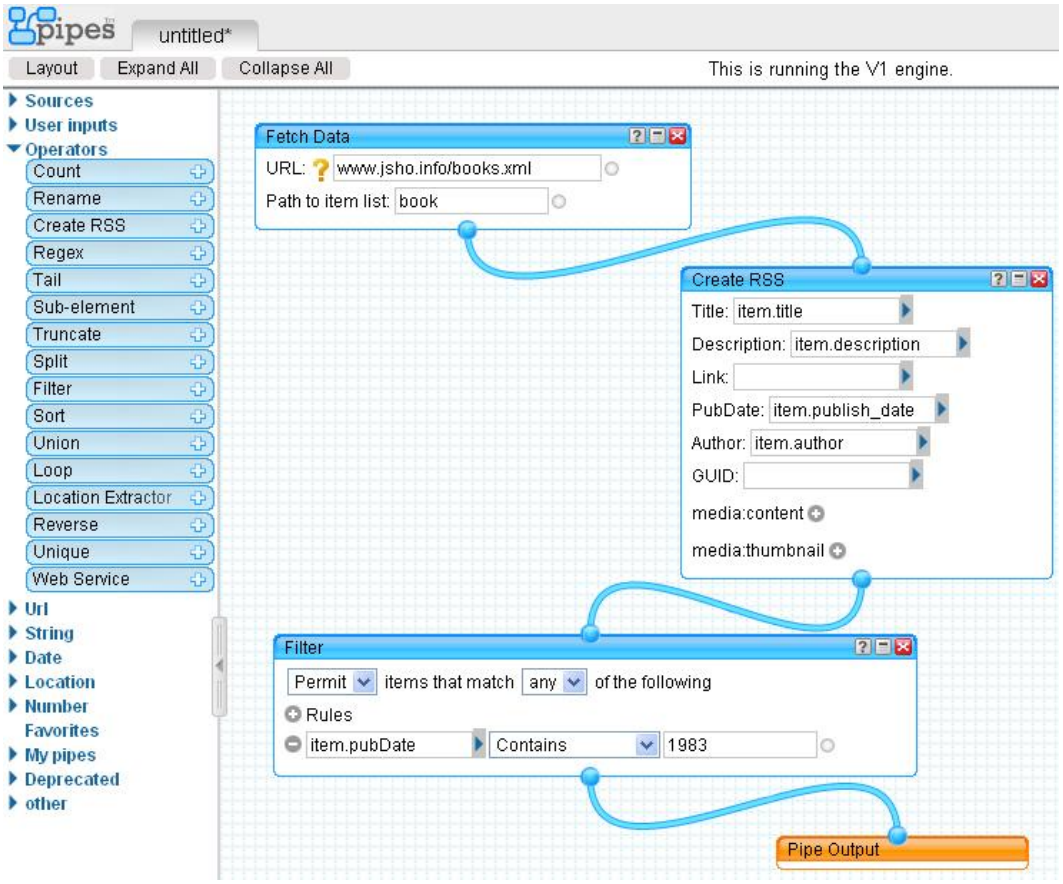


Figure 8: YahooPipes snapshot

Each module performs one task. It can have several inputs, one output and is therefore considered a function. YahooPipes offers a one-typed final output (named Pipe Output in Figure 8), which is RSS-based. That is due to its RSS-based data model which can only interpret RSS structured data. Nonetheless, the output can be visualized in different forms or integrated into web pages.

A snapshot of YahooPipes is shown in Figure 8 with an illustration of Query 1. The user filters all the books published in 1983. It is important to note that in order to create this filter, the input XML document books.xml had to be converted manually as shown in Figure 8 into an RSS structure before it is filtered and the output of this filter (the module named Filter in Figure 8) is structured as RSS feeds.

YahooPipes mainly contains 2 sets of manipulation modules or functions named operator and string modules which respectively target RSS structures and textual values. These modules are discussed in Table 4 and

Table 5.

Table 4: Operator modules which transform and filter data flowing through the pipes

Operator Modules	Description
Count	Counts the numbers of items in the inputted feed and sends the number as an output
Create RSS	Transforms feeds that are not structured as RSS data into RSS feeds by allowing the user to map their element and rename them to RSS elements
Filter	Filters all items in the inputted feed based on specific criterions applied to any of their sub-elements
Location Extractor	Searches the input feeds for geographic data such as "Lat", "Long", "Latitude" and "Longitude" and then adds a y:element with sub-elements including these data
Loop	Allows the use of sub-modules operating on all of the loop module input items
Regex	Searches and replaces sub-element data based on specific patterns specified by the user in a regular expression
Rename	Renames elements in the input feed
Reverse	Reverses the order of the feeds by flipping the order of the items in it in case the inputted feeds were initially ordered
Sort	Sorts all the items in the input feed in an ascending or descending order based on a specific sub-element (e.g. title)
Split	Duplicates an input feed into 2 output feeds
Sub-Element	Extracts sub-elements from a feed
Tail	Limits the output to the last N items of the input feed, where N is specified by the user
Truncate	Limits the output to the First N items of the input feed, where N is specified by the user
Union	Merges up to 5 different items from separate feeds into a single list of items.
Unique	Deletes items containing similar strings
Web Service	Transmits YahooPipes data to a user defined web service for external treatment. The web service needs to have a specific input type, JSON format and must have an RSS typed output

Table 5: String modules for manipulating and combining textual values

String Modules	Description
String Builder	Concatenates sub-strings to one string
Sub String	Retrieves a sub-string defined by a starting index and the number of characters to be retrieved
Term Extractor	Extracts the most significant words in a String
Translate	Translates a text from one language to another
String Regex	Similar to the Regex module, except it runs on a specific string
String Replace	Replaces a specific sub-string with another
String Tokenizer	Splits a string into sub-strings delimited by a specific character
Yahoo! Shortcuts	Categories if possible different words in a string
Private String	Hides a string from other YahooPipes users

Although, YahooPipes is a Mashup tool allowing users to manipulate RSS feeds from different web sources by visual compositions, it, nevertheless, has some limitations when dealing with XML data:

- It is does not target all XML-based data
- Its input must be structured as a feed similar to RSS, Atom or RDF
- It supports only one structure, the RSS-based structure, since the internal data model is based itself on the RSS structure.
- The output of a Yahoo Pipe is limited to an RSS-based structure
- The manipulation modules offered are RSS oriented and can only operate on RSS structured XML data and are mainly based on restructuring.

To the best of our knowledge, no published work on the YahooPipes development process have been recorded and thus we were unable to find neither any formalism nor a language definition used in its conception. YahooPipes has been introduced only as a web application with a visual editor.

2.3.2 IBM Damia

Similar to YahooPipes, Damia is a Mashup tool for manipulating web data and mainly XML data. Its main objective is restructuring and transforming XML data. Its internal data model is XML based and is not specific to any particular structure. The input and output of Damia are XML structured data. Damia is a query composition tool with several integrated operators. The operators can be categorized into “presentation operators” and “building operators” as shown in Table 6 and Table 7. Presentation operators are used for data restructuring. As for the building operators, they create new data from data sources. New operators can be added to Damia by calling web services.

Figure 9: IBM Damia snapshot

Query 1 is illustrated in Damia as shown in Figure 10. It is interesting to note that the source file had to be reconstructed before the filter could be applied. Table 6 and Table 7 discuss the main operators embedded in Damia.

Table 6: Damia presentation operators

Presentation Operators	Description
Transform	Restructures the schema of the input XML data by removing and adding elements and attributes. The output result is a transformation of the initial input structure.
Sort	Sorts feeds in an ascending or descending manner based on a specific element or several.
Group	Evaluates text values and removes redundancies if the evaluation result is true.

Table 7: Damia building operators

Building operators	Description
Merge	Evaluates an expression between 2 elements of different input feeds. If the expression evaluates to true, then the 2 items are merged into one feed.
Union	Combines the entries of 2 feeds. The entries of the first feed are all added then those of the second feed.
Filter	Selects items from a feed satisfying a specific condition.
Augment	Combines 2 feeds into a single feed by evaluating an expression linking the first feed into a variable defined from the second feed.

Although Damia is a visual XML restructuring tool and allows users to restructure XML data, it has some limitations such as:

- the XML visualization is difficult to read since there is no separate visualization of the Mashup's main input and output
- The XML data is visualized as dom trees and no structural schemas are given, even though Damia is used mainly to restructure XML data
- The operators provided by Damia are mainly based on XQuery functions.

Damia has been published as a web application with a graphical user interface. To the best of our knowledge, no formal definitions have been given nor have any languages been defined. As for the manipulation operations, they are limited to the XML structure and do not operate on any textual values.

2.3.3 Discussion

As presented in Table 8, Mashup tools share some main advantages and disadvantages with regard to XML manipulations. The advantages are:

- The majority of tools have internal data models based on XML which makes them more flexible to use even if more programming is required to implement operations on them, especially for programmers [73]
- Mashups offer operators for data elaboration such as filtering and sorting
- Mashup tools are all extensible even though special requirements (e.g. specific programming knowledge such as PHP) are necessary

The disadvantages are:

- They are mainly designed to handle Web data which can be a disadvantage since by doing this, user's data, generally available on desktops cannot be accessed and used.
- The offered operators are not easy to use, at least from a naive user point of view

- The tools don't offer powerful expressiveness since they allow expressing only simple operations.
- All the tools are supposed to target non-expert users, but a programming knowledge is usually required. And so far, there is no tool that requires low or no programming effort which is necessary to claim that the tools target end-users.

An analysis on both YahooPipes and IBM Damia are given in Table 8.

Table 8: Mashup tools analysis

Category	Sub-category	Criteria	YahooPipes	IBM Damia
Manipulated Data	**Type**	XML-specific	-	Yes
		Web-based	Yes	Yes
		User-based	-	-
	Location	Target offline data	-	-
		Target online data	Yes	Yes
Manipulation Operations	**Structural**	Selection/filtering	Yes	Yes
		Projection/transformation	Yes	Yes
		Insertion/removal	Yes	Yes
		Modification(obfuscation)	-	-
	Content (textual)	Selection	-	-
		Insertion/removal	Yes	Yes
		Textual manipulations	-	-
Interaction/ Visualization	**User**	Programming knowledge required	Yes	Yes
		Expertise required	-	-
	System	Composition-based	Yes	Yes
		Query-based	-	-
		Reusable	-	-
		Formal Visual syntax	-	-
		Expressiveness	Limited	Limited
Derivability		Formalism	-	-
		Formal language	-	-
		Extensibility	Limited	Limited

2.4 XML Manipulation Techniques

So far, different visual tools and languages for manipulating XML data by non experts have been discussed. Whether they are visual languages or Mashup tools, they share a common key feature crucial for manipulating XML, their expressiveness. The level of expressiveness defines their capabilities to allow non experts to create complex

manipulation operations. Therefore, it is essential to study existing XML manipulation techniques.

In the literature, XML manipulations have emerged in different application domains (i.e., access control, filtering, encryption etc.) mainly for security and alteration/adaptation purposes satisfying user requirements. Over the years, different approaches and techniques have emerged either for protecting or altering/adapting sensitive data:

- Security-based
  - Access Control: for controlling the access to sensitive data
  - Usage Control: for controlling the on-going access to sensitive data
  - DRM/E-DRM (Digital Rights Management/Enterprise-DRM): for managing and enforcing digital rights over data
  - Proxies and firewalls: for protecting information systems from external threats
  - Encryption and signatures: for encrypting and decrypting sensitive data.
- Adaptation-based
  - Filtering: for filtering and selecting data satisfying some criteria
  - Adaptation: for modifying and adapting data to different environment/platforms
  - Information Extraction: for extracting information from different web sources.

These techniques have been defined targeting different types of data, not necessarily XML-based (i.e., textual, audio, visual, etc.). After the standardization of XML, they were adapted to deal with it. While they require separately high level of expertise and cannot be implemented by non-expert users, it is important to study them individually in order to assess their expressiveness even though they can be defined nowadays in online libraries as web services or offline libraries as DLLs or Jar Files where they can be called upon by non-experts.

In the following section, we will discuss different XML security and adaptation techniques.

2.4.1 XML Security

Since 1984, several approaches have been discussed and developed for controlling and securing resources such as information systems, applications and files. In particular, these approaches were adapted to secure XML files and data. They are mainly divided into 5 main categories

- Access Control

- Usage Control
- DRM/E-DRM
- Proxies and Firewalls
- Encryption and digital signatures.

These techniques can be used to protect XML-based data and some of them such as Access Control and Encryption have been adapted specifically to XML. In the following sections we'll discuss each of these techniques separately and discuss their relatedness towards XML.

2.4.1.1 Access Control

Access control is mainly used to grant or deny access to data. They filter the data upon access. Several access control models have been proposed in the literature such as I-BAC [49], R-BAC [41], T-BAC [98] and Or-BAC [64]. These are conceptual models referring at creating access rules. The appliance of all of these models results in an access control matrix granting or denying rights to subjects over objects.

To identify these entities (subjects and objects) and the access control matrix, a security model is required. This model should tolerate a wide structured SP (security policy) allowing its decomposition and facilitating its definition. It should also be able to express not only authorizations but interdictions (denials) and obligations while accessing the data in an information system. Finally, it should express rules submitted to the conditions of the information system environment.

In the XML field, access control has been adapted to XML from a fine grained perspective [29, 43, 72, 75], granting access to XML values, elements or attributes. XML-oriented access control models are applicable to XML fragments and XML files. The end goal of using access control with XML remained the same, to grant or deny permission over read or update (e.g. insert, delete, replace and rename) operations. The control is on the permission level over the XML data and does not interfere with the data itself. Nevertheless, access control can be viewed sometimes as a filtering or selection approach (cf. section XML filtering) providing XML data selection without any modifications. To increase the expressiveness of access control, the usage control concept has emerged as a dynamic access control [83].

2.4.1.2 Usage Control

Usage Control (UC) is a new emerging concept in the field of access control, trust management and digital rights management such as TUCON and $UCON_{ABC}$. The TUCON [112], even though called Time Usage Control, is based on access control models. It extends access control by defining usage periods of time and maximum

times a privilege can be exercised. Therefore it is still an access control model granting or denying rights to a resource in full. It can be applied to XML data but in the same scope as traditional access control models.

$UCON_{ABC}$ [83] is a generalized dynamic access control model where the SP is being evaluated before (pre), during (ongoing) and after (post) accessing the information. It is a generalization of access control covering authorizations, obligations, conditions, ongoing control and attributes mutability proposed by Sandhu and Park in [83]. Sandhu and Park have addressed usage control from an access control point of view. Figure 10 represents the Control process of $UCON_{ABC}$

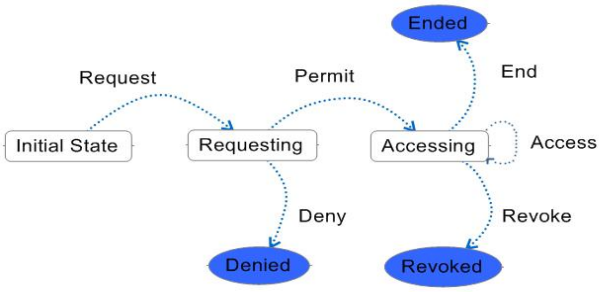


Figure 10: $UCON_{ABC}$ Control Process

As depicted in Figure 10, when a subject tries to access information, it sends a request query (Request) to the reference monitor which will grant or deny access due to the query's legitimacy. This action in a traditional access control normally takes place before (pre) any access has been permitted. With the $UCON_{ABC}$ this action is evaluated before (pre) any access by granting permission (Permit) or a rejection (Deny). It is also evaluated during (ongoing) access by either continuing the access (Access) or revoking it (Revoke) till an end query (End) is returned.

$UCON_{ABC}$ is an attribute based model, the properties of its entities (subjects and objects) are represented by attributes. The dynamic change in the SP is translated by a modification in the value of the attributes instead of the entities themselves.

The attributes are called mutable attributes. They can be modified in all three stages, pre, ongoing and post.

The SP of the $UCON_{ABC}$ is represented by ternary relations between its entities. These relations are divided into 3 categories:

- Authorizations (permissions): they describe the conditions under which the subject can access the information represented by an object
- Obligations: they verify that all conditions have been met when a subject is requesting access and during it

- Conditions: they rely on the system's environment. They are different from obligations because they do not rely on the attribute of the subjects and objects.

To summarize, Usage Control is a generalization of access control to render it more dynamic, therefore it resolves to similar but more developed objectives as access controls, which means the control remains on the permission level but is rendered more dynamic. With regard to XML, any existing XML-oriented access control model can be rendered dynamic and thus, viewed as XML-oriented usage control providing pre-access and post-access filtering of XML data. While it is clear that implementing a usage control model requires some level of expertise, it would be interesting to provide non-experts with usage control pre-defined functions via web services or offline functions which can be used in any visual composition platform.

2.4.1.3 DRM and E-DRM

DRM concept originated from operating system's file protection mechanism. In the DRM field, encryption and watermarking are manipulation operations widely used to protect sensitive content (including XML-based data).

DRM [110] is essentially defined as a modular architecture for modeling access and usage control in the application level. The DRM architecture, as shown in Figure 11, is composed of 3 main components:

- Content Server: contains the content repository and a DRM Packager which combines sensitive contents and their corresponding rights.
- License Server: contains rights, encryption keys, IDs and generates licenses for the corresponding IDs.
- DRM Client: contains the DRM Controller which associates the content to the license.

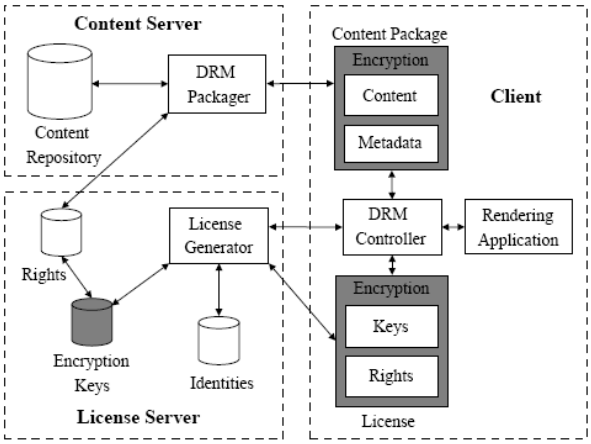


Figure 11: DRM Architecture

The DRM field is divided into two sub categories:

(a) Systems for distributing content to consumers in a controlled way against piracy known as DRM

(b) Systems for managing access to sensitive document content within an enterprise known as E-DRM (Enterprise Digital Rights Management) aiming at reducing information theft, especially by insiders.

Both DRM and E-DRM systems should generally consider the following principles:

- Secure content by distributing encrypted files or files' metadata that links to related files on a protected repository
- Control and audit access to protected content (edit, copy, paste…)
- Introduce minimum changes to enterprise business process and existing user applications
- Enable external users like business partners to access rights-protected content
- Secure the license server or policy server against attack or system failure.

E-DRM remains an ambiguous concept, not following any specific formalism or definition. E-DRM systems are used as distributed architectures for implementing access and usage control. They do not provide a means to describe controls; they define an architecture, not a control model. The scope of DRM is generally multimedia files not XML files and as for E-DRM systems, although, they remain ambiguous in their definition, they can be used for XML document protection against theft and can apply some manipulations in terms of data obfuscation and signature. Even though, XML is not the main target of DRM and E-DRM systems, but since XML has reached the multimedia area and the communication (textual, audio or visual) can be expressed in XML, thus DRM/E-DRM systems can be used for managing rights to different XML files, mainly multimedia-typed files.

2.4.1.4 XML Proxy Servers and Firewalls

A proxy server is a computer system or an application that treats client's requests by forwarding them to the proper servers.

(a) The client sends a request to use some service (e.g. to view a web page, to use a web service etc...).

(b) The proxy server sends a request to the required server on behalf of the client.

The proxy server can manipulate the data by altering the client's request or the server's response if needed. Several types of proxy servers exist such as Caching proxy [87], Web proxy [87], Content Filtering Web Proxy [111], Anonymizing proxy [108] and Reverse Proxy [52].

They are used to intercept outgoing and incoming data and can be designed for XML data. XML proxies provide protection needed against malformed messages and

malicious content in XML documents. Depending on their degree of AI (artificial intelligence), they can alter this data for different circumstances such as extract or filter sensitive data. A proxy is a system or application; it does not specify a conceptual model for describing data flow content control and specifically a model for XML data manipulation. And writing Proxy rules can be complicated and normally requires a high level of expertise.

XML firewalls are divided into two approaches, hardware-based and software-based. Both approaches have the same goal, to protect and prevent attacks to a system from malformed and malicious XML content. Basically an XML firewall comes as a part of an overall XML proxy server. Several solutions have been developed each with specific scopes and objectives such as Xwall [51] and DataPowerXS40 [59]. So far, XML firewalls' aim has been Web services such as in [109]. They are based on SOAP filtering, XML encryption, digital signatures, schema validation and access control. There have been no standard descriptions on how XML firewalls work so far and they are used to manipulate XML data for protection purposes mainly. Similar to all XML security approaches, they require a high level of expertise.

To summarize, both XML firewalls and proxy servers are means to protect a system from malicious content. They can use different techniques, such as filtering, data extraction, removal and others, depending on their AI degree. Nevertheless, there development process is complex and requires high level of expertise and cannot be accomplished by non-experts.

2.4.1.5 XML Encryption and Signature

As the number of applications increased, the usage of XML increased to ensure communications between different applications and platforms. To secure these communications and make sure that the data integrity remains intact between end users, XML encryption and digital signatures were introduced:

- Encryption is used to make sure that data can only be viewed by the corresponding users (applications or humans) and prevent its theft.
- Digital signatures are used to authenticate the identity of the XML data provider and ensure the integrity of the original content of the document.

XML encryption and signature were standardized by the W3C (World Wide Web Consortium). Other formalizations were established allowing both encryption and signature in the same language such as in[58].

Encryption and signature are applicable on 2 levels:

- Document: allows the encryption of the whole document as an entity
- Element-wise: allows 3 different levels of granulation:

- whole element
- attribute of an element
- whole content of an element

XML encryption and signature constitute a small part of XML control (manipulation) as viewed in our research. It can be categorized in either the security field of control or the modification/adaption field of control depending on its use. This approach still lacks the ability to allow a granular encryption or signature of the element content data (e.g., <description> President ***#a0sH2XsA*** had an urgent meeting with Mr ***#sZ4edErZ***.</description>).
While implementing encryption techniques over XML data is complex for non-experts, providing some online or offline encryption/decryption functions can be very usefull for non-experts since, as mentioned earlier, existing composition platforms (i.e., mashup tools) can now call web services or offline functions.

2.4.2 XML Adaptation

The alteration/adaptation field of control resides in modifying and adapting the XML data to satisfy the needs of a user(s). Researchers have been developing different solutions with separate scopes such as filtering, adaptation and information extraction. While these techniques may be complex to implement by non-experts, they define some of the main manipulation operations required which can be used by non-experts if implemented as online or offline functions.

2.4.2.1 XML Filtering

XML filtering has been one of the main fields that researchers have been developing in order to apply some control and adaptation of XML data to user specifications. In the literature, XML filtering was seen from 2 sides: (i) security [75] and (ii) data querying [19]. From one side, it was considered as an approach to enforce access control over XML data, from another side, it was considered an approach for XML data selection or extraction. Technically speaking, XML filtering can be described as: "Given a set of twig patterns, retrieve the data corresponding to these patterns in an input XML document or data". XML filtering results in a granular selection of XML data. Its granularity degree depends on the filter applied. Several filtering techniques have been developed based on either XPath expressions or a subset of XQuery. Some of the main techniques developed are XFilter [3], YFilter [38], QFilter [75], PFilter [18] and AFilter [19]. These techniques have been evolving using mainly DFA (deterministic finite automata) and NFA (non-deterministic finite automata) for either

structural matching or value based-predicates. The supported range of value based predicates has evolved from equality operators to non equality operators, Boolean operators (AND/OR) and finally the special matching operator "%" processed similarly as the LIKE operator in SQL. Basically, the XML filters are based on DFA or NFA diagrams generated from XQueries or XPath expressions defining the twig patterns specified by users in order to find specific XML data corresponding to users' criteria as shown in the example depicted in Figure 12.

In Figure 12, an XML data filter is defined based on XPath expressions. In this example, we can see 8 rules defined in XPath and their translation to an NFA diagram defining the possible patterns for the XML data selection.

The rules can be viewed as access control rules or as selection patterns and the NFA diagram as the execution model for enforcing these rules or querying XML data based on the selection patterns.

R_1: /site/categories//*
R_2: /site/regions/*/item/location
R_3: /site/regions/*/item/quantity
R_4: /site/regions/*/item/name
R_5: /site /regions/*/item/description
R_6: /site /people/person/name
R_7: /site /people/person/address//*
R_8: /site /people/person/emailaddress

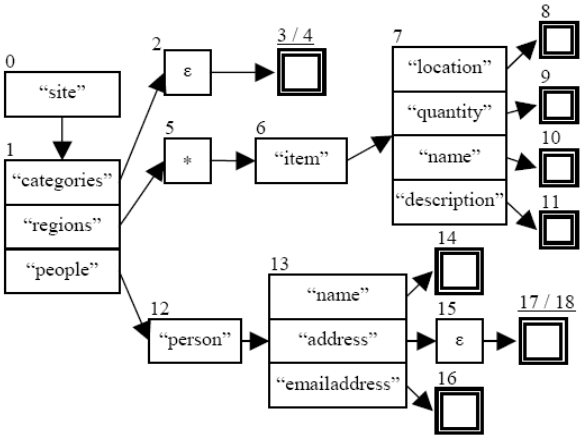


Figure 12: User queries defined with XPath expressions for the required filter and the corresponding NFA

XML filtering is considered a selection tool and does not involve XML data modification. Therefore, it can be considered as part of the XML alteration/adaptation field responding to selection criterions with no XML data modification attached. Even though, their appliance may require high level of expertise, providing some pre-defined filtering functions can be very useful for non-expert users.

2.4.2.2 XML Adaptation

Several researches have been conducted concerning XML content adaptation [68, 99], mostly on XML document such as XHTML [85], SMIL [16] and SVG [42] containing multimedia content. The main goal of XML adaptation has been so far to adapt multimedia content such as images, audio and video sequences to be viewed on appropriate terminals (e.g., portable multimedia devices, mobile phones and HD displays). The adaptations are made mostly in terms of resolutions, aspect ratios and size [71, 84] in correspondence to the terminals displaying the data and their specifications (e.g., viewing an XHTML-based web page on a PDA requires its pictures and text size to be reduced and adapted to the PDA's resolution). The adaptation mechanism in multimedia content adaptation is normally based on the properties of the document containing the data which has a well know structure and is well defined to contain multimedia data such as in SMIL or SVG [71, 84]. There were some researches conducted on adapting XML documents and transforming them to other XML documents to satisfy a certain objective based on the XSLT standard [102]. Due to the complexity found in XSLT, this approach was categorized by users as complicated and limited to the actions allowed by the XSLT language. While these adaptations are complex to implement or develop, providing different adaptation functions which can be called upon from XML-oriented composition tools such as YahooPipes and IBM Damia would be interesting.

2.4.2.3 Information Extraction (IE)

Data extraction and modification are essential aspects in XML data manipulation. Several solutions exist for data extraction [2, 23, 27] or IE (information extraction) based on the usage of wrappers. These solutions are mainly aiming at IE from web pages and are not directly related to XML files. The extracted data is mainly stored in XML files (e.g., extracting the results of a search query on Google and storing the resulting page name, description and link in a structured XML file). Some of them are IEPAD [23], Nodose [2] and ROADRUNNER [27]. These approaches mainly rely on visual info which is either defined by the browser or the user (i.e., data location on the screen). No standardized approach exists yet. IE solutions are viewed as applications or tools which mainly learn from examples given by the user in order to generate IE rules. Most of these approaches view web pages as trees rendering the data extraction process faster. Nevertheless, these approaches are inadequate or insufficient for XML manipulation due to their lack of formalism and being that they are not used on XML files but web pages instead and are limited to the tools used for data transformation which are user based and do not follow any existing models or standards.

2.4.3 Discussion

The following table regroups the different scopes and data types targeted by existing XML security and adaptation techniques. Table 9 shows different XML manipulation techniques used for protection or adaptation purposes. It is noticeable that some of these techniques do not target all types of XML data, nevertheless they constitute the main manipulation operations currently existent.

Table 9: Scope and data types of existing alteration/adaptation control techniques

	Scope	XML data type
AC (Access Control)	Granting or denying access to XML content	All XML data types
UC (Usage Control)	Granting or denying access to sensitive content continuously	No XML appliance yet
DRM/ E-DRM	Applying AC or UC over a document based on user policies	XML documents
Proxies/ Firewall	Manipulating XML data based on pre-defined rules	All XML data types
Encryption	Obfuscating XML data	All XML data types
Filtering	Filtering based granular selection of XML data	All XML data types
Adaptation	Modifying XML data content to render it conform to an alien system	Mainly multimedia XML data
IE (Information extraction)	Extracting Data based on user-defined rules and storage in a DB, XML files or others	Mainly Web Pages

Table 10 shows an analysis of the adaptation and security techniques with regard to the criteria identified in this study.

Table 10: Analysis regarding XML adaptation and security techniques

Category	Sub-category	Criteria	AC	UC	DRM	Firewall	Encryption	Filtering	Adaptation	IE
Manipulated Data	Type	XML-specific	Yes	Yes	Yes	Yes	Yes	Yes	Depends	-
		Web-based	-	-	-	-	-	-	-	Yes
		User-based	-	-	-	-	-	-	-	-
	Location	Target offline data	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-
		Target online data	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Manipulation Operations	Structural	Selection/ filtering	Yes	Yes	-	-	-	Yes	-	Yes
		Projection/ transformation	-	-	-	-	-	Yes	-	Yes
		Insertion/ removal	-	-	Yes	Yes	-	-	-	-
		Modification (obfuscation)	-	-	-	-	Yes	-	Yes	-
	Content (textual)	Selection	-	-	-	-	-	-	-	-
		Insertion/ removal	-	-	-	-	-	-	-	-
		Textual manipulations	-	-	-	-	-	-	-	-
Interaction/ Visualization	User	Programming knowledge required	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
		Expertise required	High	High	High	High	High	High	High	High
	System	Composition-based	-	-	-	-	-	-	-	-
		Query-based	-	-	-	-	-	-	-	-
		Reusable	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
		Formal Visual syntax	-	-	-	-	-	-	-	-
		Expressiveness	High	High	High	High	High	High	High	High
Derivability		Formalism	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-
		Formal language	-	-	-	-	-	-	-	-
		Extensibility	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

While several techniques have been developed and formally defined over the years, nevertheless all these techniques are separate from each other, target each a specific manipulation operation concerning XML and require high level of expertise for their implementations. Now that online and offline libraries are widely spreading over the computer domain, providing online and offline manipulation functions would render the XML manipulation task by non-experts simpler and more agile.

2.5 Dataflows

Since the early developments of computers in the 1940s and up till now, researchers and developers have been trying to simplify the programming paradigm in order to allow non-expert-programmers to develop their own applications each in his own area. Programming languages have evolved over the years from low-level languages (i.e., assembly languages) to high-level languages (i.e., Fortran, Java, C++, etc.), domain-specific textual languages (i.e., VHDL for electronic/logic programming) and domain-specific visual programming languages, also known as VPL. As the technology progressed and VPLs surfaced, the gap between non-expert and expert-programmers began to shrink. VPLs are divided into 2 main categories, visual querying languages and Dataflow visual programming languages also known as visual functional composition languages. Each of these languages followed respectively the query paradigm (cf. section 2.2) and the Dataflow paradigm (i.e., functional compositions). While on one hand, the Query paradigm required users to have some knowledge in query languages, the Dataflow paradigm, on the other hand, is closer to the natural human thinking process. It is mainly based on simple mapping (linking) of different modules together. Although and to the best of our knowledge, there has been no XML-oriented DFVPL developed, DFVPLs have been designed and formally defined specifically for data manipulations by non-expert users for e-science data, such as in DFL (Dataflow Language), V (Visual Dataflow language) and Taverna discussed here below.

2.5.1 DFL: a Dataflow language based on petri nets and nested relational calculus

Hidders et al. [53, 54] argued in their papers that since Dataflow languages have not been formally defined and published yet, it is essential that formal descriptions and definitions should be given and published. This will ease and allow for precise analysis and understanding of Dataflows [101] which is essential in the Dataflow research area for:

- Debugging by the authors

- Effective and objective assessment of their merit by researchers
- Clear understanding by the readers

Most importantly, from the research perspective, giving formal definitions and semantics provide the ability to perform formal analysis and automated optimization and verification of the behavior of the program. Therefore, DFL [53] was mainly designed as a well defined formalism for representing Dataflows.

In DFL, static data is represented by tokens, and operations on the tokens' content are performed by transitions. Conditions can be defined in edges which will allow only tokens with values satisfying theses conditions to pass. DFL also provides additional annotations to Dataflows, the unnest/nest annotations which allow to ungroup and group tokens and thus providing "for loops".

DFL is defined based on petri nets as presented in Figure 13 with the addition of labels to transitions giving the computation done by them and the association of NRC (nested relational calculus) [17] values with the tokens to represent the manipulated data. DFL inherits the set of basic operators and the type system from NRC.

- NRC (Nested Relational Calculus): it is considered a query language mainly used for describing functional programs using collection types (e.g. lists and sets etc.). The main feature of NRC is its ability to work with collections. NRC defines a set of basic types which can be combined to form collections (e.g., sets). Based on its semantics, NRC can be seen as a Dataflow description language [54] describing the computations that need to be performed but does not specify their order. Therefore, NRC is inconvenient for Dataflows where the order of execution is essential such as in Dataflows calling external functions (e.g., online and offline libraries) and in expressing control flow.

The main interest for using NRC in DFL is to allow iteration over sets which are translated by the definition of unnest/nest edges as shown in Figure 13.

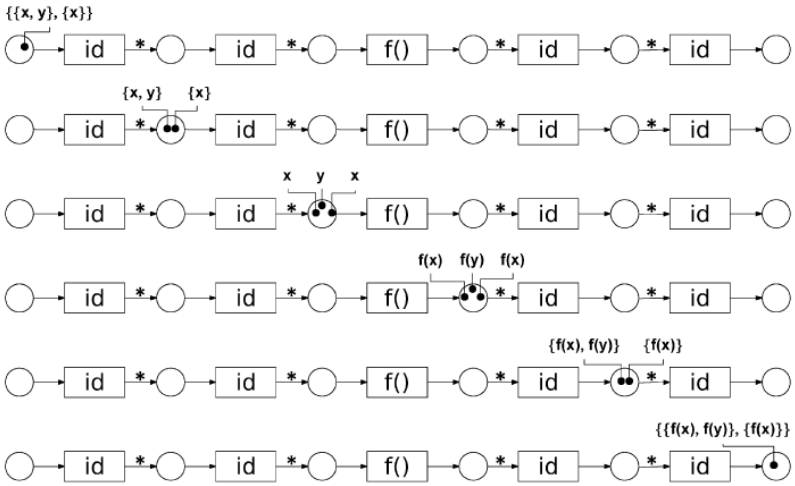


Figure 13: Example of nested iterations

Figure 13 depicts an example of the execution of nested iterations showing the unnesting of a single token "{{x, y}, y} into 3 tokens "x, y and x" and their nesting into one token "{{f(x), f(y)}, f(x)}". In the initial state of the petri net (shown in the 1st line of Figure 13), one token defined of 2 complex elements {x,y} and {x} is provided as the initial marking. This token is than separated into 3 tokens x,y and x which are modified by the function *f()* as shown in line 4 of Figure 13 and then regrouped into one token defined by {{f(x),f(y)},{f(x)}}.

Unnest edges are outgoing edges allowing a transition to consume one token set and to produce a set of tokens. Nest edges are incoming edges allowing a transition to consume a set of tokens and produce a single token.

In the DFL language, a Dataflow is defined formally as a 5-tuple <DFN, EN, TN, EA, PT> where:

- DFN is a Dataflow net defined as a 5-tuple, DFN= <P, T, E, source, sink> where:
  - *<P, T, E>* is an acyclic workflow net, a classical petri net having places *P*, transitions *T* and arcs *E*.
  - *Source* $\in P$ is the source place. It defines the initial state in a petri net
  - *Sink* $\in P$ is the Sink place. It defines the final state in a petri net.

- *EN:* o$T \rightarrow EL$ is an edge naming function that labels edges leading from places to transitions so that a distinction can be made between input edges when a transition has several input edges such as with unnested edges
- *TN:* $T \rightarrow TL$ is a transition naming function that labels transitions allowing the specification of desired operations and functions for each transition

- *EA: (*oT → *{"=true", "=false", "=Ø", "≠Ø", "*", ε})* ∪ (oP → *{"*", ε})* is an edge annotation function annotating each edge with a condition. Tokens only satisfying the condition can be transported over the edge. "*" denotes unnest/nest edges.
- *P: P → CT* is a place type function providing a specific type for each place and thus restricting the values accepted by each place. The types mainly used are the basic types defined by NRC (e.g., Boolean, Integer, etc.).

The semantics of DFL is defined as a transition system shown in Figure 13, similar to classical petri nets, where each place can contain 0 or more tokens representing data values. The distributions of tokens over the Dataflow places define the current state of the Dataflow and are called markings. Transitions are considered to be the active components in a Dataflow, since they are defined based on the petri net firing rule and thus, allowing them to transit the Dataflow from one state to another by consuming input tokens and producing output tokens. In DFL, a transition represents a computation step determined by the function associated with the transition label. Consumed tokens by a fired transition represent the input values of these functions and the produced tokens are their output.

Although DFL stands for "DataFlow Language", nevertheless its main purpose is to formalize Dataflows and particularly visual Dataflows. While DFL provides a formal syntax and semantics of a generic Dataflow language based on the petri net algebraic grammar, it does not define the language as a proper VPL, it does not formally define a visual syntax for Dataflows which are considered to be a particular type of VPLs and require to have their unique visual syntax.

Since DFL was conceived to formalize Dataflows, it was defined as a generic language for Dataflows and does not target specific data types, in other words it is not defined to be specific to XML nor any other data. Instead, its concern was more on the computation aspect of Dataflows.

To the best of our knowledge, DFL is defined as formalism for Dataflows and has not yet been implemented. No case studies were conducted which is natural due to its generic and formal aspects which render the task difficult.

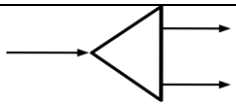
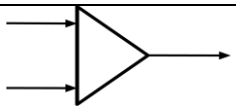
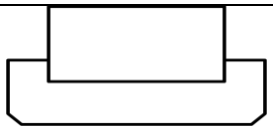
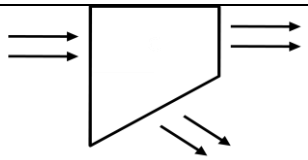
DFL mainly relies on the Dataflow paradigm and does not aim at providing a simpler VPL for novice programmers.

Last but not least, in DFL [53], Hidders et al. combine 2 approaches to define their language, petri nets and NRC, and apply some additions to them which renders the task of understanding the syntax not very easy for researchers.

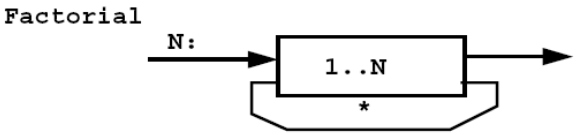
2.5.2 The V language (Visual Dataflow language)

The V language was developed as an experiment by Auguston and Delago [9, 10] for representing Dataflows and more particularly dependencies between data and processes such as in Labview [62] and prograph [62]. The V language was designed mainly as a visual formalism for Dataflows. Table 11 presents the graphical components formalized in the V language.

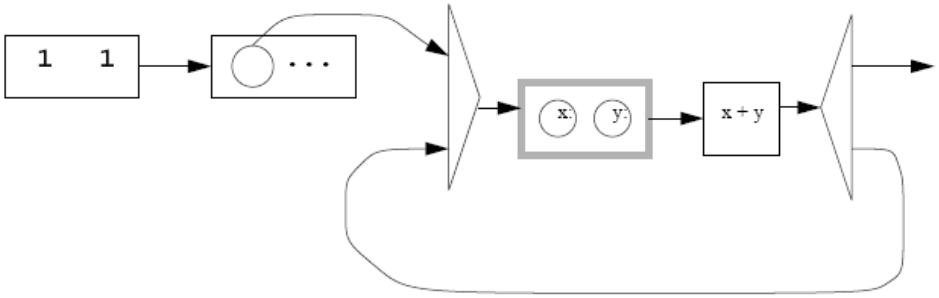
Table 11: Visual formalism of the V language

Graphical Representation	Name	Description
	Value box	Denotes a value, either a scalar or an aggregate
	Operation box	Denotes a function to be executed when all inputs are available
	Single Iteration pattern	Defines a pattern that matches a single value
	Iteration pattern	Defines a pattern that matches a group of values
	Fork	Duplicates the input
	Merge	Lets through whatever input becomes available first
	Regular computations	Applies aggregation operations such as Sum, Max, Min, Count etc. over a group of data
	Conditional switch	Evaluates a Boolean expression and transmits the input to the output based on the result of the expression

In Figure 14, 2 examples are given showing how we can represent different operations using the V language. In Figure 14.a, the factorial of N is defined using the regular computation component. Figure 14.b shows a diagram represented in the V language that generates Fibonacci sequences.



(a) Factorial in V Language



(b) Fibonacci stream in V language

Figure 14: Iterative constructs in the V language

The V language provided a series of visual constructs formally defined for representing Dataflow diagrams. Nonetheless, the V language is merely a formal visual representation and not a VPL, whereas it does not provide any formal syntax based on a grammar or algebra.

Since its purpose was to provide a visual formalism, therefore it is not specific to any data type. Nevertheless, to prove its simplicity, the V language was implemented as a simple graphic editor supporting only integer data types. To the best of our knowledge, no use case scenarios were published.

2.5.3 Taverna Workflows

Taverna [80, 86] is a practical workbench for defining and executing scientific workflows. Turi et al. [101] presented a formal syntax and semantics for Taverna workflows. The main motivation behind their research (defining a formal syntax for Taverna) was to apply process analysis techniques and enable unambiguous mapping between different models [81].

Turi et al. [101] defined formally Taverna as a functional composition language based on the Lambda Calculus algebra. They defined Tavern *workflows* as a composition of several *processors* having several *typed inputs and outputs* as presented here:

- Types are formally defined as:

$$\tau ::= s|L(\tau)|\tau \times \tau|1 \text{ where:}$$

  - S is a base type
  - $L(\tau)$ is a complex type based on a basic type s
  - $\tau \times \tau$ is a multi-input/output type
  - 1 is a 0-ary product type for workflows without any output.

- Typed inputs were formally defined as *Contexts*. A context is a list of typed inputs such as:

$$\Gamma \equiv x_1 : \sigma_1, \dots, x_n : \sigma_n \text{ where:}$$

  - $x_1, \dots, x_n$ are input variables of type $\sigma_1, \dots, \sigma_n$
  - $\sigma_1, \dots, \sigma_n$ are of types τ.

- A processor is defined as an axiom of the form:

$$\Gamma \vdash p : \tau \text{ where:}$$

  - Γ defines the input variables
  - τ defines the output type
  - p defines the processor.

- A workflow is defined as a collection of processors with mapped inputs and outputs as:

$$\Gamma \vdash P : \tau \text{ where:}$$

  - Γ defines the inputs of the workflow
  - τ defines the output type
  - P defines the workflow.

In order to create a workflow, 3 main compositions were defined:

- Simple: it is the mapping of one workflow's output to another workflow's input with the same type
- Iterative : it maps one output 'a' to a list of inputs $[b_1, \dots, b_n]$ resulting in a list of pairs $[<a,b1>,\dots,<a,b_n>]$
- Wrapped: it maps one output 'a' to a one element list [a].

Since these compositions represent different types of mapping but do not provide any control over how the execution should be done, therefore a control link was added. The control link denotes that a processor cannot be executed before another processor has terminated (the controller).

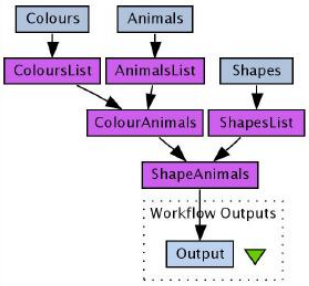


Figure 15: Taverna workflows diagram

Figure 15 shows an example of a Taverna workflow diagram for representing animal shapes. The formal syntax of the language defined the language as a functional composition language where error free workflows are produced. Nonetheless, on one hand, due to the lack of visualization provided by the Lambda Calculus, the compositions remained mathematical and no formal visual representations were given. On the other hand, due to the lack of synchronization in the Lambda Calculus, the authors had to define a controller processor which needs to be implemented between compositions in order to synchronize their execution and thus noticeably increases the execution time and memory.

Last but not least, Taverna was developed for e-science workflows and does not manipulate XML data nor is it defined formally as a DFVPL.

2.5.4 Discussion

Table 12 shows a summary of the DFL, V and Taverna Dataflow languages with regard to a XML-oriented formal DFVPL. Although DFVPLs are the closest to the human thinking process and therefore considered the easiest to learn, nevertheless and from one point of view, we can mainly identify so far and to the best of our knowledge that no DFVPL has been yet defined specifically for manipulating XML data. From another point of view, the formal definition of a DFVPL syntax and its runtime environment remains blur. It is unclear where one ends and the other begins. As for the implementation and practical use of formally defined DFVPL, lots of difficulties are confronted when applying the theoretical approach into a machine language with visual representations such as memory usage, multy-threading and graphical representations.

Table 12: DFVPL analysis

Category	Sub-category	Criteria	DFL	V	Taverna
Manipulated Data	**Type**	XML-specific	-	-	-
		Web-based	-	-	-
		User-based	Yes	Yes	Yes
	Location	Target offline data	Yes	Yes	Yes
		Target online data	-	-	-
Manipulation Operations	**Structural**	Selection/filtering	-	-	-
		Projection/transformation	-	-	-
		Insertion/removal	-	-	-
		Modification(obfuscation)	Yes	Yes	Yes
	Content (textual)	Selection	-	-	-
		Insertion/removal	-	-	-
		Textual manipulations	-	-	-
Interaction/ Visualization	**User**	Programming knowledge required	-	-	-
		Expertise required	-	-	-
	System	Composition-based	Yes	Yes	Yes
		Query-based	-	-	
		Reusable	Yes	Yes	Yes
		Formal Visual syntax	Yes	Yes	Yes
		Expressiveness	High	-	High
Derivability		Formalism	Yes	Yes	Yes
		Formal language	Yes	Yes	Yes
		Extensibility	Yes	Yes	Yes

2.6 Discussion and Conclusion

Since the widespread of XML to all areas and to most communication medias worldwide both online and offline, XML manipulation by non-expert users has become crucial and imperative. Users from different areas have increasing needs for manipulating and controlling their communications (i.e., cardiologists who wish to communicate their records with other colleagues in partial, journalists who wish to gather, filter and construct their personalized report on different events, etc.).

So far, in the literature, we have not found a unified approach resolving this matter. Nevertheless, we identified several approaches/techniques related to the topic from different angles where each of them handles a specific aspect concerning XML manipulations by non-experts. These approaches were organized into 4 main categories: XML Querying Visual Languages, Mashup tools, XML Security and Adaptation, and DFVPLs. While each of these approaches has been separately discussed and analyzed, we elaborated a global analysis and diagnostic of all

approaches put together in correspondence to our topic while based on the criteria defined earlier. The results are shown in Table 13. This analysis allows us to compare existing approaches and shows their limitations regarding the required criterions. In general, manipulating XML data by non-expert users requires the approaches/techniques to be:

- XML Specific
- Web-based and User-based
- Located offline and online

The manipulation operations should allow:

- Structural selection, projection, insertion, removal and modification
- Value selection, insertion, removal and manipulation

From the interaction and visualization perspectives:

- No programming background or expertise should be required
- The approach should be based on functional compositions
- Composed operations should be reusable
- A formal syntax is required for analysis and error handling purposes
- Expressiveness should be high allowing complex operations to be created

Finally, the approaches need to be derivable. They should be formally defined as visual programming languages and extensible so that they can be adapted to any environment and futuristic requirements.

Table 13: Analysis of XML manipulation approaches

Category	Sub-category	Criteria	XML-VL	Mashup	DFVPL	Security/ Adaptation	Required
Manipulated Data	**Type**	XML-specific	Yes	Possible	-	Dependent on the technique	Yes
		Web-based	-	Yes	-	-	Yes
		User-based	-	-	Yes	-	Yes
	Location	Target offline data	Yes	-	Yes	Yes	Yes
		Target online data	Yes	Yes	Yes	Yes	Yes
Manipulation Operations	**Structural**	Selection/ filtering	Yes	Yes	-	Dependent on the technique	Yes
		Projection/ transformation	Yes	Yes	-	Dependent on the technique	Yes
		Insertion/ removal	-	Yes	-	Dependent on the technique	Yes
		Modification (obfuscation)	-	-	Yes	Dependent on the technique	Yes
	Content (textual)	Selection	Yes	-	-	-	Yes
		Insertion/ removal	-	-	-	-	Yes
		Textual manipulations	-	-	-	-	Yes
Interaction/ Visualization	**User**	Programming knowledge required	Yes	-	-	Yes	-
		Expertise required	Low	-	-	High	-
	System	Composition-based	-	Yes	Yes	-	Yes
		Reusable	-	-	Yes	Yes	Yes
		Formal Visual syntax	Yes	-	Yes	-	Yes
		Expressiveness	Low	Limited	High	High	High
Derivability		Formalism	Yes	-	Yes	Yes	Yes
		Formal language	Yes	-	Yes	-	Yes
		Extensibility	Low	Limited	Yes	Yes	Yes

As for the existing approaches, in a nutshell, each of them has its advantages and disadvantages. While XML visual languages are oriented towards XML and formally define their graphical and language syntax, they lack high expressiveness, data modification and still require users to have some knowledge in programming, querying and XML. As for Mashup tools, they are closer to human thinking by providing functional compositions and can be used to manipulate data, but they are not formalized yet, not necessarily oriented towards XML, and their compositions cannot always be reused. XML security and adaptation techniques are highly expressive and may provide a variety of manipulation operations. Nevertheless, they are defined separately and are specific each to an operation. They are not defined as languages and require high level of expertise for their implementation. From the point of view of non-expert users, these manipulation operations can be found very useful if embedded in offline or online libraries, specifically now that we have visual systems/tools rich enough to call upon functions from such libraries (i.e. YahooPipes and IBM Damia). Finally, DFVPLs show to be the most promising by successfully bridging the gap between non-expert programmers and providing high expressiveness. While they have been formalized as visual languages and do not require any programming knowledge, they cannot manipulate XML data due to the lack of DFVPLs oriented towards XML. Therefore, although they can provide a major contribution in the future, nevertheless they remain currently inadequate and ineffective for XML manipulations by non-expert users.

CHAPTER 3

BACKGROUND AND PRELIMINARIES

In this chapter, we present the main approaches/techniques used while defining our approach, called XA2C (XML mAnipulAtion Composition), starting with an overview on the Dataflow paradim, followed by the Dataflow languages and DFVPLs (DataFlow Visual Programming Languages).

Table of Contents

3.1 Introduction

Before we discuss our approach in detail, we define here its background and pillars. Our research entitled, "XA2C, a framework for XML-oriented mAnipulAtion composition by non-expert users" is mainly defined as a visual studio for XCDL (XML-oriented Composition Definition Language) a visual programming language following the Dataflow paradigm. Our aim is to define a solid framework for non-expert users to manipulate their XML data flows. As discussed in the related works chapter, there hasn't been yet any approach in the literature providing a solution for this subject. Nonetheless, the Dataflow paradigm in particular is the most relevant of all since it:

- targets non-expert users,
- is most suited for data manipulation,
- is the closest to the natural human thinking process.

Nonetheless, they are not XML-oriented. Thus, we adopted the Dataflow paradigm in our approach, more precisely DFVPLs, while rendering it XML-oriented.
Since we are opting for a DFVPL, which falls in the category of VPLs, this chapter and the next subsections are dedicated for providing some background on the Dataflow paradigm and DFVPLs.

3.2 Dataflows

The Dataflow approach was motivated by the exploitation of massive parallelism [30, 107, 113]. The Dataflow architecture was based on using only local memory and by executing instructions as soon as their operands become available. A program written based on the Dataflow paradigm is a directed graph, as shown in Figure 1, where data flows between instructions along its arcs [5, 33, 113].

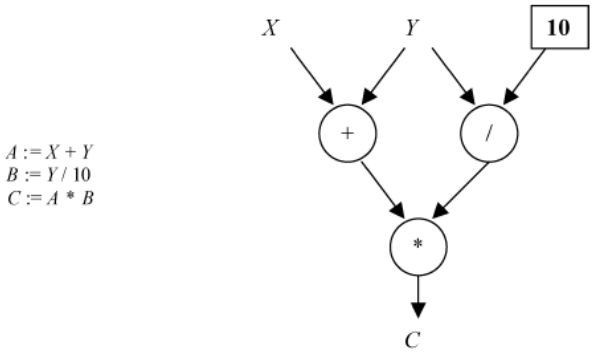


Figure 1: Dataflow graph of a simple mathematic problem

3.2.1 Dataflow Execution Model

In a Dataflow execution model, a program is represented by a directed graph. Conceptually, data flows as tokens along the arcs which behave like unbounded first-in, first-out (FIFO) queues [65].

When a programs starts, special activation nodes place data onto certain key input arcs, and thus triggering the rest of the program. Whenever a specific set of input arcs of a node (called a firing set) has data on it, the node is said to be fireable. A fireable node is executed at some undefined time after it becomes fireable. The result is that it removes a data token from each node in the firing set, performs its operation, and places a new data token on some or all of its output arcs. Instructions are scheduled for execution as soon as their operands become available in contrast to the von Neuman execution model (the serial execution model) [37, 107] in which an instruction is only executed when the program counter reaches it, regardless of whether or not it can be executed earlier than this. It is clear that Dataflow provides the potential to provide a substantial speed improvement by utilizing data dependencies to locate parallelism.

Theoretically, in Dataflow programs, data controls the execution. Two approaches were defined in the literature:

(a) **Data driven approach** (or data availability driven approach) [34, 100]: where the execution is dependent on the availability of data in the input nodes. An overall management device notifies and fires the nodes when their data become available:

i. A node is activated when all its inputs are available

ii. A node absorbs its inputs' tokens, and places tokens on its output arcs.

(b) **Demand driven approach [33, 63]**: where a node is activated only when it receives a request for data from its output arcs as follows:

i. A node's environment requests data

ii. The node is activated and requests data from its environment

iii. The environment responds with data

iv. The node places tokens on its output arcs.

It is arguable that the demand driven approach prevents the creation of certain types of programs such as modern and real-time softwares which are mostly event-driven. In these cases, it is not enough for the output environment to simply request input, a data driven approach is required instead.

3.2.2 Early Dataflow Architectures

When implementing Dataflow programs, the main concerns are token-storage techniques and number of parallel instructions that can execute in reality, since they assume theoretically an unlimited number of parallel executions. Thus, three approaches have emerged:

(a) **Static approach:** was proposed by Dennis and Misunas [37] and discussed by other authors [35, 36, 92]. Under this approach, the FIFO design of arcs is replaced by a simpler design where each arc can hold, at most, one data token. Therefore, the firing rule for a node is that a token must be present on each input arc, and no tokens present on any of the output arcs. The implementation of such architecture requires the implicit addition of acknowledgement arcs to the Dataflow graph in the opposite direction to each existing arc which will carry acknowledgment tokens. Its main strength is its simplicity and quickness to detect whether a node is fireable or not. In addition, it allows for the memory to be allocated for each arc at the compile-time since each arc can hold no more than 1 data token. However, the static model suffers though from a serious problem, data traffic. The data traffic is increased by a factor of 1.5 to 2.0 due to the additional acknowledgement arcs. Also, the execution of loops is severely limited.

(b) **Dynamic or tagged token approach:** was proposed by Watson and Grud [7, 106]. The conceptual view of the tagged token model is that it exposes additional parallelism by allowing multiple invocations of a sub-graph that is often an iterative loop. But in reality, only one copy of the graph is kept in memory. Tags are used to distinguish between tokens that belong to each invocation. A tag holds a unique ID used to invoke a sub-graph, as well as an iteration ID in case the sub-graph is a loop. These IDs put together are commonly known as the *color* of the token. As opposed to the single-token-per-arc rule of the static model, the dynamic model allows each arc to contain any number of tokens, each with a different tag [92]. In this case, a given node is said to be fireable whenever the same tag is found in a data token on each input arc. The main advantage of this architecture is that it can execute in parallel separate loop iterations. However, its main disadvantage is the extra overhead required to match tags on tokens. Therefore, more memory is required and an associative memory is impractical. Thus, memory access is limited and not as fast as it could be [92].

(c) **Synchronous Dataflow approach**: was a later development in the Dataflow paradigm and became widely used [70]. It is a subset of the pure Dataflow model where the produced and consumed number of tokens is known at compile-time. As a consequence, loops can only be defined when the numbers of iterations is known at compile-time. The main advantages of this approach are two: (i) it can be statically scheduled, and (ii) the execution can be converted into a sequential program where no dynamic scheduling is required.

3.2.3 Early Dataflow Programming Languages

Dataflow languages were derived from a specific type of functional languages [57]. In early Dataflow languages, Dataflow graphs were merely an illustration of the Dataflow programs. They were used as simple presentations of the compiled code [37]. The graphs were drawn by hand or through a third-party application. Therefore, these early graphs are not to be mistaken for Dataflow languages. A Dataflow programming language required some basic features.

3.2.3.1 What are the bases of a Dataflow programming language?

Traditional Dataflow languages were not graphical even though they could be expressed graphically. They were mainly text-based. The boundaries of what constitutes Dataflow languages are somewhat blurred due to the existing overlap with other classes of languages (e.g., functional languages). Some core features can be defined though which are essential for any Dataflow language:

1. *Freedom from side effects*: Dataflow programs do not allow the definition of global variables and prohibit its functions from modifying its parameters and thus guarantying freedom from side effects
2. *Locality of effect*: Dataflow programs disallow the definition of global variables, which renders the effects of its execution local
3. *Data dependencies equivalent to scheduling*: Scheduling is determined based on data dependencies, being that a node in a Dataflow program does not execute unless all of its firing sets are available, in other words, all of its operands become obtainable
4. *Single assignment of variables*: Since scheduling is determined based on data dependencies, it is crucial that variables values do not change between their definition and their use. Therefore, reassignment of variables to new values is prohibited
5. *Lack of history sensitivity in procedures*: In general, since scheduling is based on data dependencies and to prevent traffic overflow, Dataflow programs disregard execution history.

3.2.3.2 Dataflow languages

Several text-based Dataflow languages were designed over the years. Some of them are: TDFL [107], LAU [44], Lucid [8], Id [6], LAPSE [45], VAL [1], Cajole [48], DL1 [89], SISAL [77] and Valid [4]. These languages shared some main similarities such as functional semantics, single assignment of variables, and limited constructs to support concurrency.

One of the main advantages of the Dataflow paradigm was that it allowed concurrent and parallel executions that were considered a blockage for the Von Neuman architectures, in the 80s, which were based on sequential executions. In the early 90s, Lee and Hurson [69] raised the issue of granularity which became one of the key points to be addressed in Dataflows, after it was realized that the Von Neuman architecture did not oppose to the Dataflow architecture but instead could be complementary to the latter and could create possibilities for new and more efficient architectures [82, 92]. Thus, *fine-grained* Dataflow could be considered as a multithreaded architecture where each low-level instruction is executed separately on its own thread and the Von Neuman architecture was seen as a particular case of a multithreaded architecture where there was only one thread running in the execution.

Based on these updates, a major change in the Dataflow approach took place. Hybrid Dataflows became the dominant area of research in the Dataflow community by the mid 90s. In 1995, Sterling et al. [94] explored the performance of different levels of granularity in Dataflow systems.

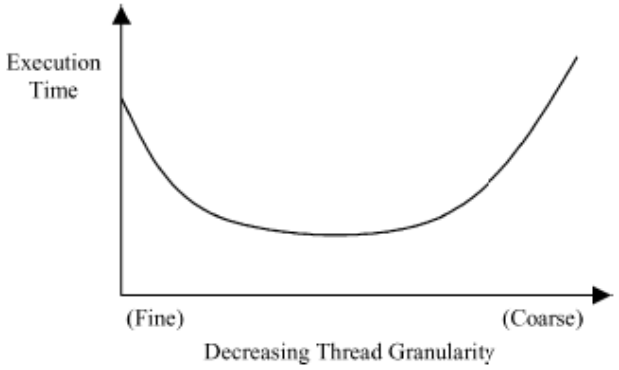


Figure 2: Dataflow granularity curve from Sterling et al.

Figure 2 summarizes the results and conclusions reached which indicates that neither fine-grained (pure multithreaded Dataflow) nor coarse-grained[1] (sequential execution) approaches were optimal. Instead, a common approach should be used, the *medium-grained* approach. Due to these changes in the Dataflow area, a key aspect became and remains an open question for researchers:

[1] The coarse-grained Dataflows are used for serial executions and do not allow any parallel executions.

"What is the best degree of granularity?"

3.2.4 Recent Dataflow Programming Languages

From the late 70's and till late 80's, Dataflow languages were all text-based. Nonetheless, the machine languages designed to be run on Dataflow hardware were based on the Dataflow graph so as the reasoning behind the definition of Dataflow programs. In the early 80's, it was realized that Dataflow graphs could have major advantages on the programmer [33]. On one hand, and as discussed in [11, 78, 91], graphs allow easy and simpler communication to novice programmers and thus increases the productivity between providers and consumers. On the other hand, VPLs' (Visual Programming Language) researchers [47, 56, 90] have indicated that providing visual syntaxes has significant advantages and particularly when based on the Dataflow paradigm, seeing that several Dataflow environments have been the basis of successful commercial products as mentioned by Baroth and Hartsough [11]. These researches [11, 60] have shown that mostly users and developers naturally think similarly to the Dataflow paradigm, in particular its graph conception. Thus, DataFlow Visual Programming languages (DFVPL) have emerged removing the complexities forced on the developer when coding in textual based programming languages. Some of the first DFVPLs are discussed below.

3.2.4.1 Early DFVPLs

(a) **DDNs (Data Driven Nets):** DDNs was created as a graphical programming concept and was argued to be the first DFVPL where graphs were no longer used for representation purposes only [30-32]. A DNNs program is represented as a cyclic Dataflow graph where arcs are defined as FIFO queues which contain typed data. The program is displayed as a graph but stored in a textual file as a parenthesized character string. The program was considered a very low level operating language and Davis commented that it was not intended for developers to program directly in it. In DDNs, Davis illustrated some key concepts in DFVPLs such as providing procedure calls and conditional executions without the use of a textual language.

(b) **GPL (Graphical Programming Language):** GPL was developed in the early 80s by Davis and Lowder [34]. It was defined as a higher-level DFVPL and in particular a higher-level version of DDNs. Davis argued that textual programming lacked intuitive clarity. Therefore, it was contended that graphs needed to be used more than just for design purposes. GPL provided structured programming with top-down development where each

node in the graph can be either an atomic node or can be expanded to reveal a sub-graph.

(c) **FGL (Function graph language):** Keller and Yen [67] developed FGL in the early 80s from the same concept where Dataflow programs need to be defined from Dataflow graphs directly. Similarly to GPL, FGL supported the top-down stepwise refinement. Nonetheless, unlike GPL, FGL is not based on the token based model but the structure model instead where data is grouped into a single structure on each arc, rather than floating around the system.

3.2.4.2 Recent DFVPLs

(a) **Labview:** is one of the most known DFVPL developed in the late 80s [11]. It was conceptualized and developed to allow users to visually construct virtual instruments for electronic data analysis in laboratories. As such, it was intended for novice programmers. The Jet Propulsion Laboratory reported empirical evidence in [11], showing that Labview provided a very favorable experience when used for large projects compared to developing the same system in C. The main advantage shown was the significantly fast development time with regard to the C language due to the facilitated communication provided by the visual syntax.

(b) **ProGraph:** was more of a general purpose DFVPL that combined the principles of Dataflow with object oriented programming [25, 26]. The main advantage of Prograph was the definition of objects and their methods as Dataflow diagrams.

(c) **NL:** was developed in the mid 90s by Harvey and Morris [50] along with a supporting programming environment which was based on the Dataflow execution model. The main advantage of NL was its programming environment which featured a visual debugger allowing execution step by step and the use of breakpoints.

3.3 Dataflow in a Nutshell

As a conclusion, based on the researches mentioned previously (e.g., [11, 62, 78, 91]), 8 major aspects were identified in DFVPLs:

(a) The area of DFVPLS does not provide a clear distinction between the language and the execution environment,

(b) The distinction between the coding and testing of DFVPL-based software is blurred,

(c) The blurring of the testing, environment and language definition makes DFVPLs easier for rapid prototyping,

(d) When developing a software, the design phase benefits the most when using DFVPLs,

(e) The semantics of DFVPLs are generally considered intuitive and easy to understand for none and novice programmers,

(f) Dataflow programs generally have a deterministic nature because the Dataflow concept allows for mathematical analysis and proofs,

(g) Research in the DFVPL field shows that there is a lack of control-flow which remains an open issue up to now,

(h) Iterations remain an open issue in DFVPLs and no unified solutions have yet been defined. So far, each DFVPL, if required, defines its own method for creating iterations based on its own needs.

CHAPTER 4

XA2C APPROACH

(XML mAnipulAtion Compositions)

In this chapter, we present our XA2C framework intended for non-expert users, providing them with means to write/draw their XML data manipulation operations. The framework is defined based on the dataflow paradigm (visual compositions). It takes advantage of both Mashups and XML-oriented visual languages by defining a well-founded modular architecture and an XML-oriented visual functional composition language. The language is based on colored petri nets allowing functional compositions. The framework uses existing XML manipulation techniques by defining them as XML-oriented manipulation functions. It defines a language platform for creating/composing XML manipulation operations, a compiler for translating the composed operations into executable machine code, and a Runtime Environment for executing these operations.

Table of Contents

4.1 Introduction

The purpose of our research is to provide non-expert users with means to create XML oriented manipulation operations, thus altering and adapting XML-based data to their needs. The approach needs to be both generic to all XML data (text-centric and data-centric) and needs to be well-founded, in order to allow it to be portable and reusable in different domains and platforms (i.e., Mashups, XML manipulation platforms, XML transformation and extraction, textual data manipulations, online and offline systems, different operating systems, etc.).

As stated in previous sections, there have been no existing approaches answering such matters. Nonetheless, several approaches have emerged undertaking different aspects of our research such as, (i) Mashups, which are neither formalized nor XML specific, are being oriented towards functional compositions and scope non expert programmers, (ii) XML visual languages, while they are formalized and XML specific, they provide only XML data extraction and structural transformations but no XML data manipulations, mainly text-centric based, and (iii) XML manipulation techniques. They are dispersed from one another resolving each a different objective (e.g., filtering, data extraction, etc.) and require expertise in their appliances. As for DFVPLs, while they haven't been oriented towards XML manipulations, nonetheless they are designed for scientific data manipulations by non-expert programmers, and have proven to be closest to the natural human thinking process.

Consequently, in order to well define our framework, we clearly identify the main objectives and properties of our approach, cross-reference them with related works and elaborate the solutions answering these objectives.

The following objectives have been identified:

(a) **Modularity**: We need a well-defined framework allowing the **creation, evaluation/validation and deployment/execution of manipulation** operations clearly and separately.

In order to define a fully functional framework from the creation phase to the deployment phase of a program, it needs to:

a. be based on a modular architecture so that each phase can be identified and developed separately
b. define a programming language allowing users to create their manipulation operations
c. identify an internal data model used for program evaluation and validation.

d. provide a runtime environment allowing the execution of the validated programs separately from the language platform

(b) **Simplicity and expressiveness**: It should target non-expert users.

i. Since the approach is intended for non-expert users, thus it should follow a natural programming paradigm closest to the human natural thinking process

ii. Users may require complex operations, thus the approach needs to be highly expressive.

Since the approach is intended for non-expert users, thus it should be as user friendly as possible, require a low level of programming knowledge and retain high expressiveness. As discussed in the related work section, DFVPLs fulfill these requirements. DFVPLs are VPLs in nature, thus, they provide a graphical representation for non-expert programmers. They are based on the dataflow paradigm which gives them the advantage of using the natural programming paradigm which is closest to the human thinking process. And finally, they are specifically designed for data manipulation and provide high expressiveness.

(c) **Flexibility and extensibility**: The framework should be portable, reusable and extensible to satisfy varying requirements from different environments/areas. In order to render the framework portable, reusable and extensible on different platforms and in different environments, it should be well designed with a formally defined DFVPL. Providing formal syntax and semantics of the language will allow it to be redeployed and developed on different platforms and can be extended and adapted to new needs. Also, being formally defined will allow the definition of analysis and evaluation techniques for improving the language.

(d) **Adaptability**: The framework needs to be XML-oriented. To render the approach XML-oriented, the DFVPL must be designed for XML data manipulation by combining ordered labeled trees to their syntax which can represent any XML-based data and can be projected to graphical tree views to be integrated in a VPL as defined in XML-oriented querying visual languages.

The rest of this chapter is organized as follows. Section 2 presents an overview of our approach. Section 3 discusses in detail the language's syntax and semantics. The compiler is defined in Section 4 as a middleware between the language platform and its Runtime Environment. Section 5 defines the Runtime Environment. Finally, an illustration and a conclusion are given in Section 6.

4.2 XA2C Overview

Figure 1 shows where the approaches/techniques discussed in the related work stand from our approach called XA2C (XML mAnipulAtion Compositions).

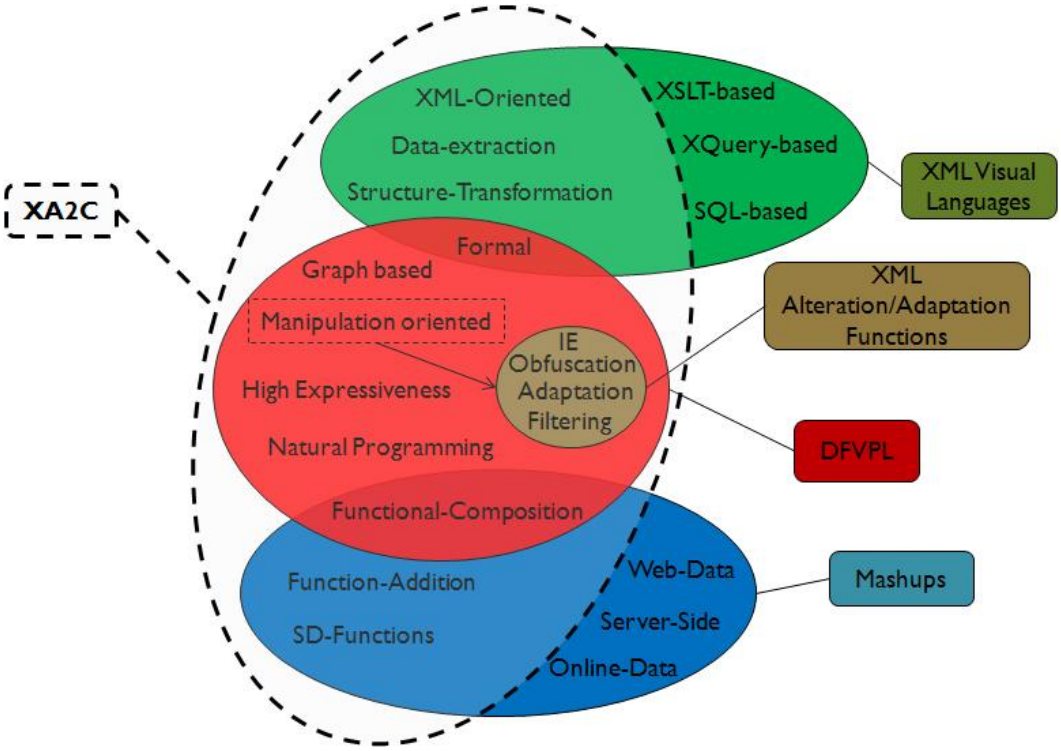


Figure 1: XA2C approach

As we can see here, the XA2C approach can not be entirely based on any existing DFVPLs, it needs to be further extended. Thus, it inherits some of the features of Mashups and XML-oriented visual languages as well. On one hand, it

1. has a similar architecture to Mashups that renders the framework flexible thanks to its modular aspect
2. is based on functional compositions which are considered simpler to use than query by example techniques.

On the other hand, it

1. defines formally a visual composition language (a DFVPL)
2. separates the inputs and outputs to source and destination structures,

thus making the framework XML-oriented and portable. Similar to the XML-oriented visual languages, the approach targets non-expert users. The visual composition language defined in XA2C can be adapted to any composition-based Mashup tool or visual functional composition tool. Nevertheless, our language is defined XML-

oriented and generic to all types of XML data (standardized, grammar-based and user-based).

4.2.1 XA2C Properties

Our framework is mainly based on 6 properties defined in its objectives: *simplicity*, *expressiveness*, *flexibility*, *extensibility*, *adaptability* and *modularity*.
In order to satisfy *simplicity*, we defined the language as a FDVPL, having a visual representation and following the dataflow paradigm. It is based on simple drag and drop actions of graphical components in order to compose manipulation operations. To provide *expressiveness*, *flexibility* and *extensibility*, we based the framework and the syntax/semantics of the XCDL (XML-oriented Composition Definition Language) on CP-Nets instead of other algebras or grammars (e.g., Lambda Calculus).

Why CP-Nets?

- CP-Nets have a very well defined semantics and can describe any type of workflow system, behavioral and syntax wise simultaneously
- They allow us to define our language as visual in a more simplified manner than other algebras and grammars (e.g., lambda calculus).
- CP-Nets allow the expressiveness of both state and behavioral changes simultaneously.
- Both, the execution and compilation of our language are based on CP-Nets.
- Dealing with concurrency is straight forward with CP-Nets, and does not require any adaptations. This allows us to define the DFVPL based on the medium-grained approach (cf. Chapter 3).
- CP-Nets are easily adapted to define Object-Oriented languages due to their ability to deal with different types of data (colors) and the use of global variables[1].
- CP-Nets can be extended to cope with different contexts, such as temporal and QoS constraints(e.g., for online services)
- CP-nets have several behavioral and dynamic properties [79] such as, boundedness, home state, coverability, persistence, synchronic distance, liveness, fairness and analysis methods such as incidence matrix, reachability graph, and coverability tree which facilitate and enrich the execution and compilation of the language.

In terms of *adaptability*, we separated the composition, from the input and output flows, which allowed us to orient the language towards different data types. In our study, we defined an ordered labeled tree structure representing XML-based data to render the language XML-oriented.

[1] They are variables which can be used anywhere in the CP-Net while preserving their values.

To ensure modularity, the XA2C framework is defined as a modular architecture as shown in Figure 2.

4.2.2 XA2C Architecture

Our framework is composed of 3 main modules:

1. The **XCDL Platform** is the most essential module and the major contribution in our work. It defines the XCDL language, the essential component of our research, providing non-experts with the means to create their manipulation operations. The language mainly allows users to define their functions from offline or online libraries and create manipulation operations through compositions achieved by mapping functions together. The XCDL is based on the graphical representations and algebraic grammar of CP-nets, thus, rendering the language extensible and generic (adaptable to different data types), and allowing the expression of true concurrency along with serial compositions (Dataflow medium-grained approach). As a user defines a new function or modifies a composition (adding, removing, replacing a function), the syntax is transmitted to the compiler module to be continuously evaluated and validated.
2. The **Compiler** is a middleware between the language platform and the runtime environment. It can be viewed as a compiler transforming the language syntax into a machine code executable in the runtime environment. It plays the role of a syntax analyzer/optimizer and code generator through the internal data model of the XA2C which are based on the same grammar used to define the syntax of the XCDL language (naturally based on CP-Nets). We define an internal data model for validating the components of the language (functions defined in our system and compositions). The validation process is event-based, any modification to the language components or composition, such as additions, removals or editions, triggers the validation process.
3. The **Runtime Environment** defines the execution environment of the resulting compositions defined in the XCDL platform. This module contains 3 main components: (i) the "Process Sequence Generator" used to validate the behavioral aspect of the composition (e.g., makes sure there are no open loops, no loose ends, etc.) and generates 2 processing sequences, a concurrent and a serial one to be transmitted respectively to the Concurrent and Serial Processing components for execution. (ii) "Serial Processing" (or fine-grained processing with one thread) allowing a sequential execution of the "Serial

Sequence" provided by the process sequence generator. It is more suitable for machines equipped with a single processor as it will not take advantage of a multi-processor unit. (iii) "Concurrent Processing" (medium-grained processing with multi-threading) allowing the execution in a concurrent manner of the "Concurrent Sequence" generated from the process sequence generator. It is imperative to note that this type of processing is most suitable for machines allowing multi-processing tasks (e.g., dual processor machines developed for parallel executions).

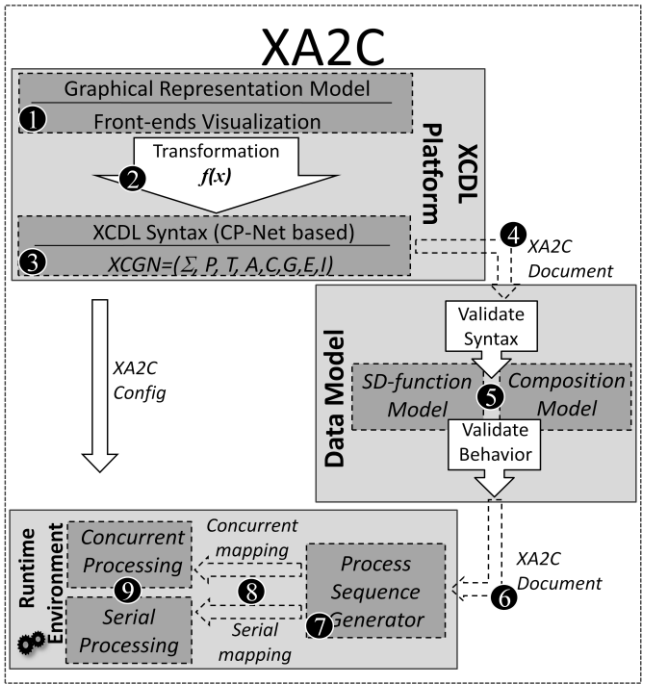


Figure 2: Architecture of the XA2C framework

In the following sections we discuss each of these modules in detail.

4.3 XCDL Platform

The XCDL is a visual functional composition language based on system-defined functions and oriented towards XML. The language is a VPL following the dataflow paradigm and is defined using petri nets, in particular CP-Nets. It is a simple drag and drop function-based composition.

In the following subsection, we give a brief description regarding visual languages and petri nets/CP-Nets.

4.3.1 Overview on Petri Nets and Visual Languages

In [46], the term Visual Language is used to describe several types of languages: languages manipulating visual information, languages for supporting visual interactions, and languages for programming with visual expressions. The latter generally refers to visual programming languages, which is the case of the XCDL provided here. Visual programming languages define programs from pictures as defined in [46]. A visual language is a set of pictures. A picture is a collection of picture elements. A picture element is a primitive graphical object such as a line, generic shapes or a text string. The syntax of a visual language is specified by distinguishing the set of pictures forming the language. A visual language is mainly divided into 3 levels:

(a) The graphical representation model which defines the graphical elements that will be used in the languages (e.g., basic shapes: lines, circles, etc.).

(b) The language syntax which is normally defined based on an existing grammar (in our case Colored Petri Nets).

(c) The transformation syntax which is used to map the language syntax to the graphical model.

As stated in [61] and [79], a Petri Net is foremostly a mathematical description, but it is also a visual or graphical representation of a system. Petri nets are state and action oriented simultaneously, in contrast to most specification languages. They provide an explicit description of both the states and the actions. Petri nets were mainly designed as a graphical and mathematical tool for describing and studying information processing systems, with concurrent, asynchronous, distributed, parallel, non deterministic and stochastic behaviors. They consist of a number of places and transitions with tokens distributed over places. Arcs are used to connect transitions and places. When every input place of a transition contains a token, the transition is enabled and may fire. When a transition fires a token from every input place is consumed and a token is placed into every output place.

CP-nets have been developed, from being a promising theoretical model, to being a full-fledged language for the design, specification, simulation, validation and implementation of large software systems.

In a CP-Net:

- The states are represented by means of places (which are drawn as ellipses).
- The actions are represented by means of transitions (which are drawn as rectangles).

- An incoming arc indicates that the transition may remove tokens from the corresponding place while an outgoing arc indicates that the transition may add tokens.
- The exact number of tokens and their data values are determined by arc expressions (which are positioned next to the arcs).
- Data types a referred to as color sets.
- It is possible to attach an expression guard (with variables) to each transition.

A CP-Net is formally defined as follows:

Definition 4.1-*Colored Petri Nets, A* ***CP-net*** *is a 8-tuple such as:*

CP-Net = (Σ, P, T, A, C, G, E, I) *where:*

- *Σ is a finite set of non-empty types, called color sets*
- *P is a finite set of places*
- *T is a finite set of transitions*
- *A is a finite set of arcs such that:*
  - $P \cap T = P \cap A = T \cap A = \emptyset$
- *C is a color function. It is defined from P into Σ*
- *G is a guard function. It is defined from T into expressions such that:*
  - $\forall t \in T: [Type(G(t)) \subseteq \Sigma]$
- *E is an arc expression function. It is defined from A into expressions such that:*
  - $\forall a \in A: [Type(E(a)) = C(p) \wedge Type(Var(E(a))) \subseteq \Sigma]$

  where p is the place of N(a)
- *I is an initialization function. It is defined from P into expressions such that:*
  - $\forall p \in P: [Type(I(p)) = C(p)]$

The types of a variable *v* and an expression *expr* are denoted *Type(v)* and *Type(expr)* respectively. Also, we denote by $|X|$ the number of elements in a set *X*. An example of a CP-Net is depicted in Figure 3. This CP-Net has 3 places: two of them have a type `Int×String`, and one has a type `Int`. The transition takes one token of the pair type and one of the integer type, and produces one token of the pair type.

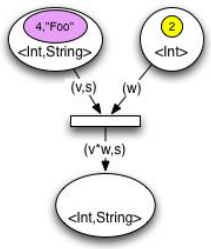


Figure 3: Example of a CP-Net

Both, the language syntax and graphical model of the XCDL are based on CP-Nets with some adjustments and restrictions.
In our approach, we are particularly interested in 2 main properties of CP-Nets, the Incidence Matrix and the Transition Firing Rule.

Definition 4.2-*Incidence matrix A*, *it is defined for a CP-Net N with m transitions and n places as:*

$$\boldsymbol{A = [a_{ij}], an\ n \times m\ matrix\ of\ integers}\ \textit{where:}$$

- $a_{ij} = a_{ij}^{+} - a_{ij}^{-}$ where
  - $a_{ij}^{+} = w(i,j)$ is the weight of the arc from transition *i* to its output place *j*
  - $a_{ij}^{-} = w(i,j)$ is the weight of the arc to transition *i* from its input place *j*

$a_{ij}^{+}, a_{ij}^{-}\ and\ a_{ij}$ represent the number of tokens removed, added, and changed in place *j* when transition *i* fires once.

Table 1 shows the Incidence Matrix of the CP-Net in Figure 3. It indicates that the transition *t* has 2 input places p_1 and p_2 and one output place p_3. As for the arcs, they have a weight of one (allowing one token to pass).

Table 1: Incidence Matrix of CP-Net in Figure 3

A=

	t
p_1	-1
p_2	-1
p_3	1

Definition 4.3-*Transition Firing Rule*, *it is the conditions for a transition to fire and is defined as:*

$$\boldsymbol{t\ is\ enabled\ if\ M(p) \geq W(p,t)\ for\ all\ input\ p\ to\ t}\ \textit{where:}$$

- *A transition "t" is enabled if each input place "p" of "t" is marked with at least "w(p,t)", where "w(p,t)" is the weight of the arc from "p" to "t"*
- *An enabled transition t may or may not fire (depending on whether event takes place or not)*
- *A firing of an enabled transition t removes w(p,t) token from each input place p to t and adds w(t,p) tokens to each output place p of t*

The XCDL language is presented in the following section.

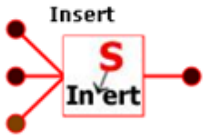
4.3.2 XCDL Overview

XCDL allows users to compose XML-oriented manipulation operations using system-defined functions. We denote by system-defined functions (*SD-functions*), functions which will be defined in the language environment. These *SD-functions* can be provided by local/offline DLL/JAR files or online services (e.g., Web service).
XCDL is divided into 2 main parts:

- The Inputs/Outputs (I/O).
- The *SD-functions* and the composition which constitute the XCDL Core.

The I/O are defined as XML Content Description trees (XCD-trees). They are ordered labeled trees summarizing the structure of XML documents or fragments, or representing a DTD or an XML schema, illustrated as tree views (cf. Figure 8).
SD-functions are defined as CP-Nets. Their inputs and outputs are defined as places and represented graphically as circles filled with a single color each defining their types. It is important to note that in this study, a function can have one or multiple inputs but only one output. The operation of the function itself is represented in a transition which operates on the inputs and sends the result to the output. Graphically, it is represented as a rectangle with an image embedded inside it describing the operation. Input and output places are linked to the transition via arcs represented by direct lines. Four sample functions are shown in Figure 4.

It tests an input string if it starts with a string provided by the user.



It inserts a string into another one starting from a specific index.

It transforms a string into a hash code.

Figure 4: Several sample functions defined in XCDL

The composition is also based on CP-Nets. It is defined by a sequential mapping between the output and an input of *SD-functions*. It is represented by a combination of graphical functions which are dragged and dropped, and then linked together with a

sequence operator which is depicted by a direct dashed line between the output of a function and an input of another one having the same color/type as shown in Figure 5.

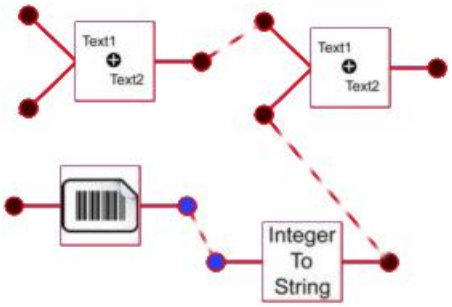
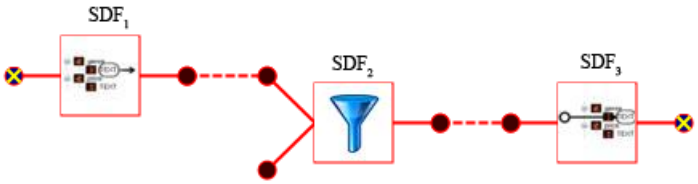


Figure 5: Functional composition in XCDL

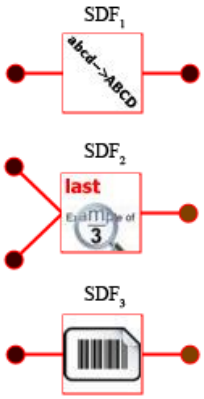
As a result, a composition might be:

- **Serial:** it means that all the functions are linked sequentially. To each function one and only one function can be mapped as illustrated in Figure 6.a. In this case, the sequential operator is enough.
- **Parallel**: it is a composition between several functions with no mapping between them whatsoever as described in Figure 6.b. In this case we introduce an abstract operator, the parallel operator indicating that the functions are parallel to each other and independent from each other.
- **Concurrent:** it contains concurrency, as in several functions can be mapped to a single one as depicted in Figure 6.c. In this case we introduce another abstract operator, the concurrency operator, which is a combination of multiple parallel operators followed by a sequence operator, indicating that the functions are concurrently mapped (parallel with dependencies).

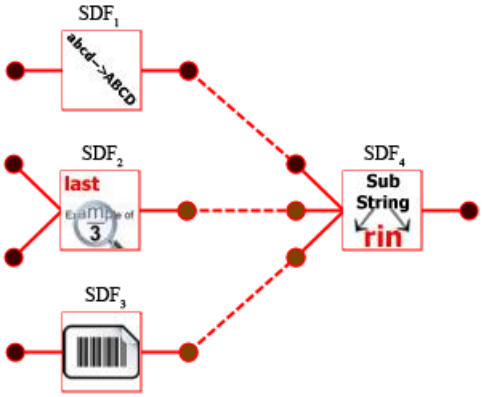
The geometric properties of the functions are shown in Figure 14, such as, input places are drawn in a symmetric manner in correspondence with the X-axis considered to be situated in the middle of the transition.



(a) Serial Composition: SDF_1 ➔ SDF_2 ➔ SDF_3



(b) Parallel Composition: (SDF_1 // SDF_2 // SDF_3)



(c) Concurrent Composition: (SDF_1 // SDF_2 // SDF_3) ➔ SDF_4

Figure 6: XCDL compositions

The distance between the circles is automatically calculated as described in Section 4.3.4.3. In the following subsections, we provide a formal definition of the I/O followed by the language syntax and its properties.

4.3.3 I/O XCD-trees

Since XCDL is XML-oriented, it aims at manipulating XML data, whether they are user-based (XML documents or fragments), or grammar-based (Document Type Definition, DTD or XML Schema Definition, XSD).

In order to describe XML data structure, we introduce a representation called XCD-tree (XML structural content description tree) depicted in Figure 8. It is based on the tree model defined in the standardized W3C DOM [105] model. It views an XML document as a root node with a set of ordered sub-trees. In our research, we design the XCD-tree as an ordered labeled tree allowing us to represent the structure defining the content of XML data. XML data content is defined by XML elements, attributes, and element/attribute values, which we assume to be textual[2]. An ordered labeled tree is defined as follows.

[2] Similarly to most approaches targeting XML data management (e.g., search, indexing, etc.) we disregard the various types of values that could occur in XML documents (e.g., Decimal, Integer, Date, etc.) for the sake of simplicity.

Definition 4.4-*An* ***OL-tree*** *(Ordered labeled tree****)*** *is a root node "R" with a set of ordered Sub-trees****, OL-tree= (N, L, A, f) where:***

- ***N*** *is the set of nodes*
- ***L*** *is a set of labels associated to each node*
- ***f : N → L*** *is the function associating a label to each node*
- ***A ⊆ N x N*** *is the set of arcs associating 2 nodes together*

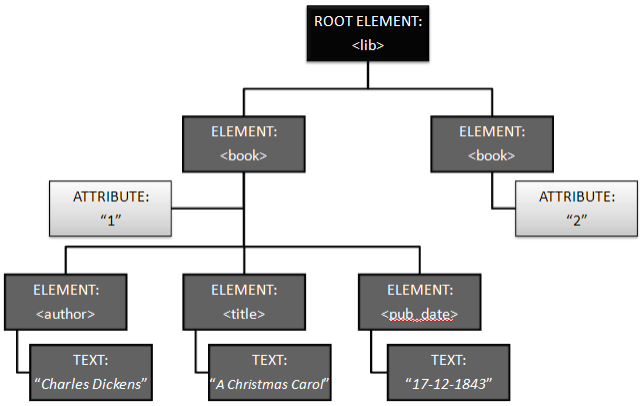


Figure 7: OL-tree representation of an XML document

The XCD-tree allows us to represent any type of XML, data-centric and text-centric. For XML files and fragments, we adapt tree structural summarization techniques with repetition reduction [28] in order to extract the structure of the XCD-tree. In this study, we defined an algorithm generating an XCD-tree. The algorithm reads throughout an XML document and builds the ordered labeled tree recursively as new elements/attributes appear while neglecting any redundancies.

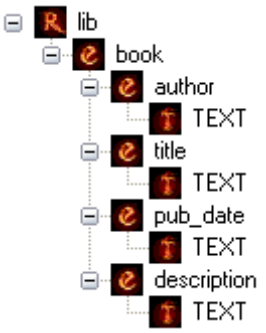


Figure 8: XCD-tree representing the XML document/DTD/XSD books

Exemple-XML document books.xml:

```
<XML version 1.0>
<lib>
      <book>
            <author>Charles Dickens</author>
            <title>A Christmas Carol</title>
            <pub_date> 17-12-1843</pub_date>
      </book>
      <book>
            <author>James Joyce</author>
            <title>Ulysses</title>
            <pub_date> 2-2-1922</pub_date>
            <description>An epic Greek myth.</description>
      </book>
</lib>
```

Exemple-DTD books:

```
<!DOCTYPE lib [
<!ELEMENT lib (book+)>
      <!ELEMENT book (author, title, pub_date, description?)>
            <!ELEMENT author (#PCDATA)>
            <!ELEMENT title (#PCDATA)>
            <!ELEMENT pub_date (#PCDATA)>
            <!ELEMENT description (#PCDATA)>
]>
```

Exemple-XSD books.xsd:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://xa2cspace.com/XMLSchema">
<xs:element name="lib">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="book">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="author" type="xs:string"/>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="pub_date" type="xs:string"/>
            <xs:element name="description" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
```

```
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

The XCD-tree representation of DTDs and XSDs is straightforward since they already give a structural view of XML documents. To simplify, XPointers and grammar constraints, such as max occurrence and min occurrence are out of the scope of our work. An XCD-tree is formally defined as follows:

Definition 4.5-*XCD-tree:* *it is a root node with a set of ordered sub-trees:*

$$XCD\text{-}tree = (N_X, T_X, L_X, f_X, A_X) \text{ where:}$$

- N_X *is the set of nodes in the XCD-tree (i.e., XCD-nodes)*
- $T_X = \{ELEMENT, ATTRIBUTE, TEXT\}$ *is the set of node types associated to each XCD-tree-node*
- L_X *is a set of labels associated to each node*
- $f_X : N_X \rightarrow L_X, T_X$ *is the function associating a label and a type to each node*
- $A_X \subseteq N_X \times N_X$ *is the set of arcs associating 2 nodes together*

Definition 4.6-*XCD-tree-node* $\in N_x$ *is represented by a doublet:*

$$XCD\text{-}tree\text{-}node = \langle type, label \rangle \text{ where:}$$

- $type \in T_X$
- $label \in L_X$

A node can have one and only one parent except for the root node, denoted by $R_{XCD\text{-}tree}$*, which has no parents. If the XML data is a fragment of XML and contains no unique root element, then a virtual root node is inserted, called v_root (cf. Figure 9). Each node has a list of child nodes. Attributes are child nodes of their elements. A node with an empty list of child nodes is a leaf node and TEXT nodes are the only leaf nodes.*

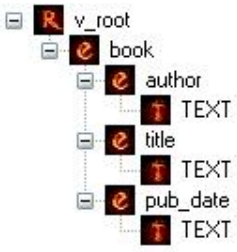


Figure 9: XCD-tree representing an XML fragment

Exemple-XML document fragment from books.xml:

```
 ---
<book>
      <author>Charles Dickens</author>
      <title>A Christmas Carol</title>
      <pub_date> 17-12-1843</pub_date>
</book>
<book>
      <author>James Joyce</author>
      <title>Ulysses</title>
      <pub_date> 2-2-1922</pub_date>
</book>
 ---
```

Table 2: Different types of XCD-tree-nodes

ELEMENT nodes: XCD-node= <ELEMENT,Name>	Ex1: <***db:exe*** xmlns:db="http://www.ex.com/">Hello</db:exe> • XCD-node = <ELEMENT,db:exe> Ex2: <***number***>14</number> • XCD-node = <ELEMENT, number>
ATTRIBUTE nodes: XCD-node= <ATTRIBUTE,Name>	Ex3: <product ***effDate***="10-1-08"> • XCD-node = <ATTRIBUTE, effDate>
TEXT nodes are leaf nodes: XCD-node= <TEXT,>	Ex4: <number>***14223***</number> • XCD-node = <TEXT,> Ex5: <product effDate="***10-1-08***"> • XCD-node = <TEXT,>

An XCD-tree-node can have 3 types as shown in Table 2. The ELEMENT and ATTRIBUTE typed nodes represent structural data. Their labels denote corresponding element/attribute tag names. As for a TEXT typed node, it represents data content, and is thus assigned a *TEXT* label in our tree representation model (since we are only interested in the content structure).

After defining the I/O of XCDL, we present next the syntax of XCDL.

4.3.4 XCDL Syntax and Semantics[3]

As discussed in the previous sections, the XCDL is a visual language defined on 3 levels as shown in Figure 10. The following subsections explain each one of them.

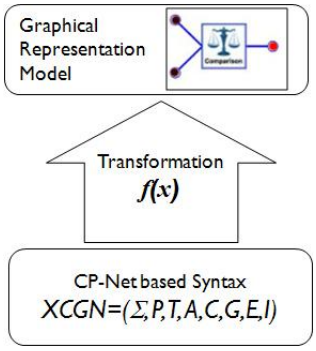


Figure 10: XCDL overview

4.3.4.1 XCDL-Graphical Representation Model (XCDL-GR)

The XCDL-GR model defines the graphical components used to represent visually the language syntax. It contains the following components: ***Point***, ***AD*** *(Abstract Drawing)*, ***Color***, ***Circle***, ***Line*** and ***Rectangle*** as shown in Figure 11.

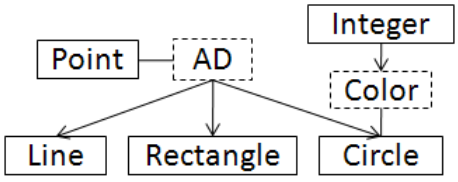


Figure 11: XCDL-GR components

The graphical components are formally defined as follows:

Definition 4.7-*Point* *is a spatial point defined by 2 coordinates as:*

***Point* = <x, y>:**

Where x and y are Integers defining the Cartesian coordinates respectively over the X-Axis and the Y-Axis

---

We denote by ***P.x*** the value of coordinate ***x*** and ***P.y*** the value of coordinate ***y***.

We define ***AD*** as an abstract drawing type which has no representation and is used as a super type for the subsequent drawing types.

---

[3] In this study, the language semantics are defined simultaneously with its syntax as a transitional system (denoting how the language operates/executes) since it is defined on petri nets.

Definition 4.8-*AD* *is an abstract drawing type defined as a doublet:*

***AD* = <*P1, P2*>:**

Where P1 and P2 are 2 Points defining reference points for the sub-types of AD

We denote by *AD.P1* and *AD.P2* respectively the instances of *P1* and *P2*.

Definition 4.9-*Color* *is an abstract drawing type defining an RGB color as:*

***Color* = <*c*>:**

Where **c** *is an* ***Integer*** $\in [0,16777215]$ *defining an RGB color*

Definition 4.10-*Circle* *is a drawing type, sub-type of* ***AD****, represented by an ellipse shape and is defined as:*

***Circle* = <*AD1, radius, color*>** *where:*

- *AD1 is an AD where AD1.P1=AD1.P2 define the center of Circle*
- *radius is an Integer defining the radius of Circle*
- *color is a Color used to fill Circle*

Definition 4.11-Line *is a drawing type, sub-type of* ***AD****, represented by a segmented line shape and is defined as:*

***Line* = <*AD1, style*>** *where:*

- *AD1 is an AD where AD1.P1 and AD1.P2 define respectively the starting and ending points of the segment Line*
- *Style* $\in$ *{dashed, normal} defines the style of the line*

Definition 4.12-*Rectangle* *is a drawing type, subtype of* ***AD****, represented by a rectangular shape enveloping an image as:*

***Rectangle* = <*AD1, w, h, img*>** *where:*

- *AD1 is an AD where AD1.P1=AD.P2 defines the point of the upper left corner of Rectangle*
- *w and h are Integers defining respectively the width and height of Rectangle*
- *img is an Image defining a thumbnail image resized proportionally to w and h and drawn in the middle of Rectangle*

If we consider *D* an instance of a drawing type and *x* one of its tuples, we denote by *D.x* the required tuple (e.g., Consider *r* as a ***Rectangle***, *r.img* retrieves *img* of ***Rectangle*** *r).*

The following section presents the syntax of the XCDL core which is based on CP-Nets.

4.3.4.2 Syntax and Semantics Definition of the XCDL Core

The syntax and semantics of the XCDL core are based on the grammar XCGN (XML oriented Composition Grammar Net) defined using CP-Nets' algebra (and therefore retains their operational semantics and properties such as, petri net firing rule and incidence matrix). Since the language is based on CP-Nets, therefore the semantics (operational semantics) are defined simultaneously with the syntax as a transitional system. The computations or operational semantics of the language (detailed in the Runtime Environment section) are simply inherited from petri nets, particularly from their firing rule (cf. Definition 4.3) while respecting the constraints posed by XCGN.

Definition 4.13-*XCGN* *stands for XML oriented Composition Grammar Net. It represents the grammar of the XCDL which is compliant to CP-Nets. It is defined as:*

XCGN = (Σ, P, T, A, S, C, G, E, I) *where:*

- *Σ is a set of data types available in the XCDL*
  - *The XCDL defines 7 main data types, Σ = {Char, String, Integer, Double, Boolean, Date, XCD-Node} where Char, String, Integer, Double, Boolean and Date are standard types and XCD-Node defines a super-type designating an XML component (cf. Definition 4.14)*
- *$P = P_{In} \cup P_{Out}$ is a finite set of places defining the input and output states of the functions used in XCDL, respectively P_{In} and P_{Out}*
  - *$\forall p \in P, [w(p) = 1]$*[4]
- *T is a finite set of transitions representing the behavior of the XCDL functions and operators*
- *$A \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs associating input places to transitions and vice versa*
  - *$\forall a \in A$: a.p and a.t denote the place and transition linked by a*
- *S is the set of operations/functions available in the platform's libraries (e.g., concat(string,string))*[5]
- *$C: P \rightarrow \Sigma$ is the function associating a type from Σ to each place*
  - *$\forall p \in P, [|C(p)| = 1]$*

---

[4] *w(p)* denotes the number of tokens in place *p*

[5] *S* is a set added to the initial CP-Net definition. It has no effect on the CP-Net's functionality. Therefore it is omitted in the rest of the definitions based on CP-Nets.

- *G:T→S is the function associating an operation to a transition*
- *E:A→Expr is the function associating an expression E(a) ∈ Expr to an arc such that:*
  - *∀a ∈ A: [Type(E(a))=C(a.p) ∧ w(a)=1]*
- *I:P→D is the function associating initial values from a domain D*[6] *to the I/O places such that:*
  - *∀p ∈ P, ∀v ∈ D : [Type(I(p))=C(p) ∧ Type(v) ∈ Σ]*

Definition 4.14-*XCD-Node* *is a super type designating an XML Component. It has 3 main sub-types as defined in the XCD-tree:*

XCD-Node *∈* ***{XCD-Node:Element, XCD-Node:Attribute*** *and* ***XCD-Node:Text}***

where:

- *XCD-Node:Element defines the XML Element type*
- *XCD-Node:Attribute defines the XML Attribute type*
- *XCD-Node:Text defines the XML Element/Attribute Value type*

Before defining the syntax of our language, we define an empty CP-Net "*Φ*" which will be used in the rest of this work.

Definition 4.15-*Φ* *is an empty CP-Net defined as:*

Φ = (Σ, P, T, A, C, G, E, I) *where:*

- *Σ = Ø*
- *P = Ø*
- *T = Ø*
- *A = Ø*
- *Since the CP-net is empty, therefore the functions do not perform any operations.*

We define now the syntax of the XCDL core. As mentioned previously, the core of the language is defined using *SD-functions*, a sequential operator, a parallel operator, a concurrency operator and the composition which is realized between different instances of *SD-functions* and operators. Therefore we introduce next the 4 main components of XCDL: (i) *SD-function*, (ii) *sequence operator "➔"*, (iii) *parallel operator* "//", and (vi) *Concurrency operator* "//➔". The parallel and concurrency operators are abstract operators denoting respectively that related functions are parallel/independent, and concurrent (parallel/dependent) to one *SD-function*.
Subsequently, we introduce the composition which is defined mainly by 3 types:

---

[6] *D* denotes the set of values pre-defined by the user as initial values in a CP-Net

1. ***Serial***: It is a sequential composition between multiple instances of *SD-functions* and sequence operators as shown in Figure 13.a.
2. ***Parallel***: Depicted in Figure 13.b, it is a composition between several instances of *SD-functions* that are independent from each other. The abstract parallel operator is used in this case to indicate that *SD-functions* are parallel to each other.
3. ***Concurrent***: It is a composition between multiple instances of *SD-functions* and sequence operators to a single instance of a *SD-function* as shown in Figure 13.c.

An *SD-function* is formally defined here below. It represents a function defined in the system's library, through a DLL file or web-services, having one or multiple inputs and a single output.

Definition 4.16-*SD-function* *is a system defined function based on CP-Nets, describing an operation based on an identified function in the system's library and is defined as:*

SD-function = (Σ, P, T, A, C, G, E, I) *where:*

- *Σ is the set of colors defining the types of data available in the SD-function*
  - *$\Sigma \subseteq XCGN.\Sigma$*
- *P is a finite set of places defining the input and output states of the SD-function*
  - *$P = P_{In} \cup P_{Out}$ and $P_{In} \cap P_{Out} = \emptyset$ where $P_{In} = \{p_{In0}, p_{In1}, \ldots, p_{Inn}\}$ and $P_{Out} = \{p_{Out}\}$. P_{In} represents the set of input places and P_{Out} represents the set of output places (containing one output place in this case).*
- *T is a finite set of transitions representing the behavior of the SD-function*
  - *$T = \{t\}$ where t contains the operation to be executed.*
- *$A \subseteq (P_{In} \times \{t\}) \cup (\{t\} \times P_{Out})$ is a set of directed arcs associating input places to transitions and vice versa where $P_{In} \times \{t\}$ indicates the set of arcs linking the input places to t and $\{t\} \times P_{Out}$ linking t to the output places (to p_{Out} in this case).*
- *$C: P \rightarrow \Sigma$ is the function associating a type to each place.*
- *$G: \{t\} \rightarrow S$ is the function associating an operation to t where $Type(G(t)) = C(p_{Out})$. The operation can be retrieved with a URI to the DLL file or a web-service.*
- *$E: A \rightarrow Expr$ is the function associating an expression $E(a) \in Expr$ to $a \in A$:*
  - *Expr is a set of expressions where:*

$$\forall E(a) \in Expr:\ E(a) = \begin{cases} M(a.p)\ if\ a.p \neq p_{Out}\ (\text{cf. } \mathbf{Definition\ 4.29}) \\ G(a.t)\ otherwise \end{cases}$$

- *$I: P_{In} \rightarrow D$ is the function associating initial values to input places.*

In Figure 12, we give a graphical representation example of an *SD-function*. This function is defined in the XCDL syntax as follows:

StartsWith = (Σ, P, T, A, C, G, E, I) *where*:

- Σ = *{String, Boolean}*
- $P = P_{In} \cup P_{Out}$ = *{In_Str1, In_Str2}∪{Out_Bool}*
- T = *{t }*
- A = *({In_Str1, In_Str2} x {t }) ∪({t} x {Out_Bool})*
- $C:P \rightarrow \Sigma$ *where C(In_Str1) = C(In_Str2) = C(Out_Bool) = Boolean*
- $G:\{t\} \rightarrow S$ *where G(t)= String_functions.StartsWith and Type(G(t)) = C(Out_Str) = Boolean where String_functions is the DLL containing String manipulation functions and String_functions.StartsWith is a function that checks incoming strings if they start with In_Str2.*
- $E:A \rightarrow Expr$:
    - *Expr={M(In_Str), G(t)} is a set of expressions where:*

$$\forall E(a) \in Expr:\ E(a) = \begin{cases} M(a.p)\ if\ a.p \neq p_{Out} \\ G(a.t)\ otherwise \end{cases}$$

- $I:P_{In} \rightarrow$ *Value where I(In_Str1) = "" and I(In_Str2) = "keyword"*

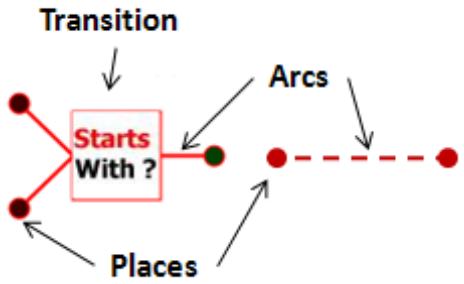


Figure 12: Graphical representations of the XCDL core components (*SD-function* and *Sequence*)

We define now a ***Sequence*** operator ***"➔"*** used to map an output place of an ***SD-function*** to an input place of another.

Definition 4.17-*Sequence* *is an operator denoted by the symbol "➔" which maps 2 places together and is defined as:*

Sequence = ***(Σ, P, T, A, C, G, E, I)*** *where:*

- Σ *is the set of colors where* $|\Sigma| = 1$
- *P is set of 2 places defining the input and output states of the Sequence operator*
    - $P = P_{In} \cup P_{Out}$ *and* $P_{In} \cap P_{Out} = \emptyset$ *where* $P_{In} = \{p_{In}\}$ *and* $P_{Out} = \{p_{Out}\}$ *where* p_{In} *represents the input place and* p_{Out} *represents the output place*
- $T = \{t\}$ *where t contains the sequence operator*

- *$A = (\{p_{In}\} \times \{t\}) \cup (\{t\} \times \{p_{Out}\}) = \{a_{In}, a_{Out}\}$ where a_{In} and a_{Out} are directed arcs associating respectively the input place p_{In} to transition t and t to the output place p_{Out}*
- *$C:P \rightarrow \Sigma$ is the function associating a type to each place where $C(p_{In})=C(p_{Out})$*
- *G: is a function over T Where:*

$$Type(G(t)) = C(p_{In}) \wedge G(t)=M(p_{In})$$

- *$E:A \rightarrow Expr$ is the function associating an expression $E(a) \in Expr$ to $a \in A$:*
  - *Expr is a set of expressions where:*

$$\forall E(a) \in Expr:$$

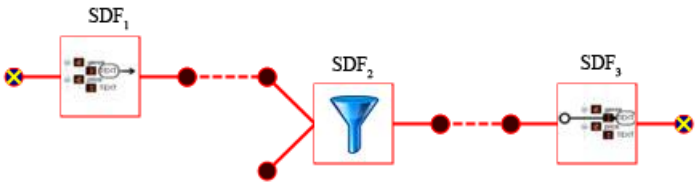
$$E(a) = \begin{cases} M(a.p) \; if \; a.p = p_{In} \\ G(a.t) \; otherwise \end{cases}$$

- *$I:P_{Out} \rightarrow D$ is the function associating initial values to the output place*

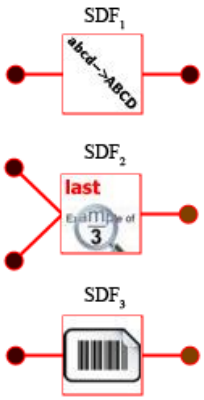
The *parallel* and *concurrency* operators, defined here, are abstract operators. Therefore, they do not have any formal definitions.

Definition 4.18-*Parallel operator* *is an abstract operator denoted by the symbol "//" which indicates that multiple instances of SD-functions are parallel to each other and independent.*

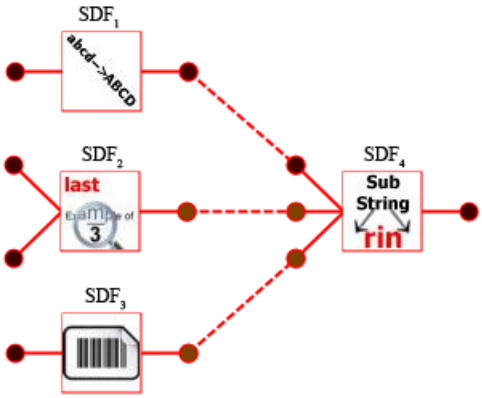
Definition 4.19-*Concurrency operator* *is an abstract operator denoted by the parallel symbol followed by a sequence one "//➔" which indicates that multiple instances of SD-functions are concurrent (parallel with dependencies).*



(a) Serial Composition: SDF_1 ➔ SDF_2 ➔ SDF_3



(b) Parallel Composition: (SDF_1 // SDF_2 // SDF_3)



(c) Concurrent Composition: (SDF_1 // SDF_2 // SDF_3) ➔ SDF_4

Figure 13: Compositions in XCDL

Figure 12 shows a graphical representation of a Sequence operator (on the right). The parallel and concurrency operators are abstract operators and have no graphical representations.

In the XCDL core, we define the composition as a serial composition mapping sequentially several instances of *SD-functions* (i.e., functions can only be executed one after the other in a specific order), a parallel composition describing several instances of *SD-functions* independent from each other and a concurrent composition, mapping several instances of *SD-functions* sequentially to a single instance of *SD-function*. Figure 13.a, b and c illustrate respectively a serial, parallel and concurrent composition.

Definition 4.20-*SC* *is a Serial Composition,* $SC = \prod_{i=0}^{n} SDF_i \rightarrow_i$, *linking sequentially* ***n*** *instances of SD-functions using* ***n-1*** *instances of Sequence operators and is a CP-Net. It is defined as:*

$$SC = \prod_{i=0}^{n} SDF_i \rightarrow_i = (\Sigma, P, T, A, C, G, E, I)$$

where:

- SDF_i *is a SD-function where:*
  - $\forall i,j \in [0,n]$ *and* $i \neq j$, $SDF_i \neq SDF_j$
  - $\rightarrow_i.SDF_{In} = SDF_i$ *and* $\rightarrow_i.SDF_{Out} = SDF_{i+1}$

- $\rightarrow_i$ *is a Sequence operator where:*
  - $\rightarrow_i.\Sigma \subseteq SDF_i.\Sigma$
  - $\rightarrow_i.P_{In} = SDF_i.P_{Out}$ *and* $\rightarrow_i.P_{Out} \in SDF_{i+1}.P_{In}$
  - $\rightarrow_n$ *= (Ø, Ø, Ø, Ø, C, G, E, I) in an empty CP-Net*
- $\Sigma = \bigcup_{i=0}^{n} SDF_i.\Sigma$
- $P = P_{In} \cup P_{Out}$ *where* $P_{In} = \bigcup_{i=0}^{n} SDF_i.P_{In}$ *and* $P_{Out} = \bigcup_{i=0}^{n} SDF_i.P_{Out}$
- $T = \bigcup_{i=0}^{n}(SDF_i.T \cup \rightarrow_i.T)$
- $A = \bigcup_{i=0}^{n}(SDF_i.A \cup \rightarrow_i.A)$
- *C:P→Σ is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where*

$$\forall t \in T,\ G(t) = \begin{cases} SDF_i.G(t),\ t \in \bigcup_{i=0}^{n} SDF_i.T \\ \rightarrow_i.G(t),\ t \in \bigcup_{i=0}^{n} \rightarrow_i.T \end{cases}$$

- *E:A→Expr is the function associating an expression to an arc where E = SD-function.E*
- *I:P_{In}→Value is the function associating initial values to input places, I = SD-function.I*

Definition 4.21-*PC* *is a Parallel Composition,* $PC = \prod_{i=0}^{n} SDF_i //$ *, compliant to a CP-Net denoting* ***n*** *instances of SD-functions totally independent and unmapped together. It is defined as:*

$$\boldsymbol{PC = \prod_{i=0}^{n} SDF_i // = (\Sigma, P, T, A, C, G, E, I)}$$

Where:

- *SDF_i is a SD-function where:*
  - *$\forall i,j \in [0,n]$ and $i \neq j$, $SDF_i \neq SDF_j$*
- $\Sigma = \bigcup_{i=0}^{n} SDF_i.\Sigma$
- $P = P_{In} \cup P_{Out}$ *where* $P_{In} = \bigcup_{i=0}^{n} SDF_i.P_{In}$ *and* $P_{Out} = \bigcup_{i=0}^{n} SDF_i.P_{Out}$
- $T = \bigcup_{i=0}^{n}(SDF_i.T)$
- $A = \bigcup_{i=0}^{n}(SDF_i.A)$
- *C:P→Σ is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where*

$$\forall t \in T, (\ G(t) = SDF_i.G(t),\ t \in \bigcup_{i=0}^{n} SDF_i.T\ )$$

- *E:A→Expr is the function associating an expression to an arc where E = SD-function.E*
- *I:P_{In}→D is the function associating initial values to the Input places, I = SD-function.I*

Definition 4.22-*CC* *is a Concurrent Composition,* $CC = \prod_{i=0}^{n}(SDF_i \rightarrow_i SDF_{n+1})$ // *linking* ***n*** *instances of SD-functions using* ***n*** *instances of Sequence operators concurrently to an instance of SD-function and is compliant to a CP-Net. It is defined as:*

$$\boldsymbol{CC = \prod_{i=0}^{n}(SDF_i \rightarrow_i SDF_{n+1}) // = (\Sigma, P, T, A, C, G, E, I)}$$

where:

- SDF_i *and* SDF_{n+1} *is a SD-function where:*
  - $\forall i,j \in [0,n+1]$ *and* $i \neq j$, $SDF_i \neq SDF_j$
  - $\rightarrow_i.SDF_{In} = SDF_i$ *and* $\rightarrow_i.SDF_{Out} = SDF_{n+1}$
- $\rightarrow_i$ *is a Sequence operator where:*
  - $\rightarrow_i.\Sigma \subseteq SDF_i.\Sigma$
  - $\rightarrow_i.P_{In} = SDF_i.P_{Out}$ *and* $\rightarrow_i.P_{Out} \in SDF_{n+1}.P_{In}$
- $\Sigma = \bigcup_{i=0}^{n+1} SDF_i.\Sigma$
- $P = P_{In} \cup P_{Out}$ *where* $P_{In} = \bigcup_{i=0}^{n+1} SDF_i.P_{In}$ *and* $P_{Out} = \bigcup_{i=0}^{n+1} SDF_i.P_{Out}$
- $T = \bigcup_{i=0}^{n}(SDF_i.T \cup \rightarrow_i.T) \cup SDF_{n+1}.T$
- $A = \bigcup_{i=0}^{n}(SDF_i.A \cup \rightarrow_i.A) \cup SDF_{n+1}.A$
- *C:P→Σ is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where*

$$\forall t \in T, G(t) = \begin{cases} SDF_i.G(t), & t \in \bigcup_{i=0}^{n+1} SDF_i.T \\ \rightarrow_i.G(t), & t \in \bigcup_{i=0}^{n} \rightarrow_i.T \end{cases}$$

- *E:A→Expr is the function associating an expression to an arc where E = SD-function.E*
- $I:P_{In} \rightarrow Value$ *is the function associating initial values to the Input places, I = SD-function.I*

As mentioned previously, XCDL is a visual language. So far, we have defined the language syntax and its graphical representations. Nonetheless, we have not associated the XCDL-GR model to its syntax yet. To do so, we define the XCDL-TS (XCDL Transformation Syntax) allowing us to transform the XCDL syntax into graphical representations based on the components defined in the XCDL-GR model.

4.3.4.3 XCDL-Transformation Syntax (XCDL-TS)

The XCDL-TS is defined in 2 layers:

1. An abstract syntax "$\mathcal{AS}$" which will associate graphical components from the XCDL-GR to CP-Net components.
2. A transformation syntax "$\mathcal{T}$" transforming the XCDL syntax into a visual syntax using "$\mathcal{AS}$".

Since the XCDL is based on CP-Nets, it contains the following main components: Color, Place, Transition and Arc. We formally define here the abstract and transformation syntax which allows us to transform the XCDL syntax into the XCDL-GR model.

Definition 4.23-$\mathcal{AS}$ *is the abstract syntax of XCDL and is defined as:*

$$\mathcal{AS} = < F_{\Sigma}, F_P, F_T, F_A > \text{ where:}$$

- $F_{\Sigma}: \Sigma \rightarrow C$ *is a function associating an abstract drawing type Color to a type* $\varepsilon \in XCFN.\Sigma$
- $F_P: P \rightarrow O$ *is a function associating a drawing type Circle to a place* $p \in XCGN.P$
- $F_T: T \rightarrow R$ *is a function associating a drawing type Rectangle to a transition* $t \in XCGN.T$
- $F_A: A \rightarrow L$ *is a function associating a drawing type Line to an arc* $a \in XCGN.A$

Definition 4.24-$\mathcal{T}$ *is the transformation syntax and is defined as:*

$$\mathcal{T} = < TF_S, TF_F > \text{ where:}$$

- TF_S *is a transformation function used to translate sequence operators into graphical data as:*

  $$TF_S = < x_1, y_1, x_2, y_2, F_S > \text{ where:}$$

  - x_1, y_1, x_2, y_2 *are integers representing the values of 2 spatial points provided by the user's mouse click*
  - $F_S: S \rightarrow D$ *is the function applying the transformation from a drawing type to a sequence* ➔ *where* $a_{In} =$ ➔$A.a_{in}$ *and* $a_{Out} =$ ➔$A.a_{Out}$ *as:*

$$\begin{cases} F_A(a_{In}).AD_1.P_1.x = x_1 \\ F_A(a_{In}).AD_1.P_1.y = y_1 \\ F_A(a_{In}).AD_1.P_2.x = \frac{x_1 + x_2}{2} \\ F_A(a_{In}).AD_1.P_2.y = \frac{y_1 + y_2}{2} \\ F_A(a_{Out}).AD_1.P_1.x = \frac{x_1 + x_2}{2} \\ F_A(a_{Out}).AD_1.P_1.y = \frac{y_1 + y_2}{2} \\ F_A(a_{Out}).AD_1.P_2.x = x_2 \\ F_A(a_{Out}).AD_1.P_2.y = y_2 \\ F_A(a).style = dashed, \quad \forall a \in A \end{cases}$$

- ***TF_F*** *is a transformation function used to translate a SD-function into graphical data as:*

  $$TF_F = <x_1, y_1, x_2, y_2, h, w, h_t, w_t, img, F_F>$$ *where:*

  - x_1, y_1, x_2, y_2 *are integers representing the values of 2 points provided by the user's mouse click*
  - *h is an integer representing the maximum height between the first and last input places*
  - *w is an integer representing the distance between the transition and a place on the x-axis*
  - h_t *and* w_t *are integers representing respectively the height and width of a rectangle representing a transition*
  - *img is an image representing an SD-function*
  - F_F*:* $F \rightarrow D$ *is the function applying the transformation from a drawing type to a SD-function, SDf, as:*
    - ***$F_\Sigma(\varepsilon)$ where $\varepsilon \in SDf.\Sigma$***
    - ***$F_P(p_i)$***

$$for\ n = |P_{In}|, i \in [0, n[, p_i \in SDf.P_{In}\ and\ dy = \frac{h}{n}\ then$$

$$\begin{cases} F(p_i).AD_1.P_1.y = y_1 - \frac{h}{2} + (i \times dy),\ i < \frac{n}{2} \\ F(p_{n\text{-}1\text{-}i}).AD_1.P_1.y = y_1 + \frac{h}{2} - (i \times dy),\ i < \frac{n}{2} \\ F(p_{n/2}).AD_1.P_1.y = y_1,\ n\ mod\ 2 = 1 \\ F(p_i).AD_1.P_1.x = x_1 - \frac{w_t}{2} - w \end{cases}$$

$$for\ p_0 \in SDf.P_{Out}\ then \begin{cases} F(p_0).AD_1.P_1.y = y_1 \\ F(p_0).AD_1.P_1.x = x_1 + \frac{w_t}{2} + w \end{cases}$$

$$for\ n = |P_{In}| + |P_{Out}|,\ i \in [0, n+m[, p_i \in SDf.P_{In} \cup SDf.P_{Out},\ F(p_i).color = F_\Sigma(C(p_i))$$

- ***$F_T(t)$, $t \in$ SDf.T*** *then*

$$\begin{cases} F_T(t).AD1.P1.x = x_1 - \frac{w_t}{2} \\ F_T(t).AD1.P1.y = y_1 - \frac{h_t}{2} \\ F_T(t).img = img \end{cases}$$

- ***$F_A(a_i)$, $a_i \in$ SDf.A*** *then*

$$for\ n = |P_{In}| + |P_{Out}|, i \in [0, n+m[, p_i \in P_{In} \cup P_{Out}$$

$$\begin{cases} F_{\mathrm{A}}(a_{\mathrm{i}}).AD_1.P_1.x = F_{\mathrm{P}}(p_{\mathrm{i}}).AD_1.P_1.x \\ F_{\mathrm{A}}(a_{\mathrm{i}}).AD_1.P_1.y = F_{\mathrm{P}}(p_{\mathrm{i}}).AD_1.P_1.y \\ F_{\mathrm{A}}(a_{\mathrm{i}}).AD_1.P_2.x = \begin{cases} F_{\mathrm{T}}(t).AD_1.P_1.x - \frac{w}{2}, p_{\mathrm{i}} \in P_{\mathrm{In}} \\ F_{\mathrm{T}}(t).AD_1.P_1.x + \frac{w}{2} \end{cases} \\ F_{\mathrm{A}}(a_{\mathrm{i}}).AD_1.P_2.y = F_{\mathrm{T}}(t).AD_1.P_1.y \\ F_{\mathrm{A}}(a_{\mathrm{i}}).style = normal \end{cases}$$

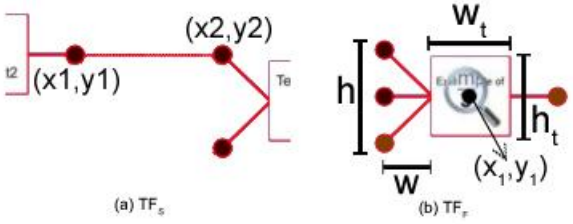


Figure 14: Transformation functions

The transformations of a sequence operator and an *SD-function* based respectively on TF_S and TF_F are depicted in Figure 14.a and b respectively.

Since XCDL is a composition-based visual language allowing different types of compositions ranging from serial, parallel to concurrent and combinations between them, we explore in the following subsection their properties.

4.3.5 XCDL Algebra Properties

Since XCDL is a visual language and the composition is done via drag and drop, the order used by the user to add his functions and map them together is arbitrary. Nonetheless, this does not affect the resulting composition. We prove that by proving that the composition is associative along with other properties stated below.

Consider *a, b, c* and *d* instances of *SD-functions*. We identify the following properties presented in Table 3.

Table 3: XCDL algebra properties

1. Associative property of Sequence	***(a $\rightarrow_a$ b) $\rightarrow_b$ c= a $\rightarrow_a$ (b $\rightarrow_b$ c)***
2. Distributive property of concurrency	***(a//b) $\rightarrow$c=((a $\rightarrow_a$ c) // (b $\rightarrow_b$ c))***
3. Associative property of parallelism	***(a//b)//c = a//(b//c)***
4. Commutative property of parallelism	***(a//b) = (b//a)***
5. Associative property of concurrency	***((a//b)//c) $\rightarrow$ d=(a//(b//c)) $\rightarrow$ d***
6. Commutative property of concurrency	***(a//b) $\rightarrow$ c=(b//a) $\rightarrow$ c***
7. Sequence Identity property (1)	***a $\rightarrow_a$ Φ =a***
8. Sequence Identity property (2)	***Φ $\rightarrow_Φ$ a =Φ***
9. Concurrency Identity property (1)	***a//Φ =a***
10. Concurrency Identity property (2)	*Φ//a =a*

The proofs of the algebra properties are given here below regarding the operators defined previously (sequence "➔", parallel "//" *and concurrency "//➔"*). It is important to note that the concurrency operator is composed of 2 operators as defined earlier, parallel and sequence, but has its own properties.

4.3.5.1 Associative Property of Sequence: $(a \rightarrow_a b) \rightarrow_c c = a \rightarrow_a (b \rightarrow_b c)$

Consider the following compositions SC_1, SC_2, SC and SC' where:

- $SC_1 = (sdf_1 \rightarrow_1 sdf_2)$
- $SC_2 = (sdf_2 \rightarrow_2 sdf_3)$
- $SC = (sdf_1 \rightarrow_1 sdf_2) \rightarrow_2 sdf_3$
- $SC' = sdf_1 \rightarrow_1 (sdf_2 \rightarrow_2 sdf_3)$

In order to prove the associative property of *sequence (SC = SC')*, we need to prove that *XCGN(SC) = XCGN(SC').*

***Proof**:*

SC = (Σ, P, T, A, C, G, E, I)

- sdf_1, sdf_2 *and* sdf_3 *are SD-functions*
- $\{\rightarrow_1, \rightarrow_2\}$ *are Sequence operators where:*
  - $\rightarrow_1.\Sigma \subseteq sdf_1.\Sigma = \mathbf{SC'.} \rightarrow_1\mathbf{.\Sigma}$
  - $\rightarrow_1.p_{In} \in sdf_1.P_{Out}$ *and* $\rightarrow_1.p_{Out} \in sdf_2.P_{In} = \mathbf{SC'.} \rightarrow_1\mathbf{.P}$
  - $\rightarrow_2.\Sigma \subseteq sdf_2.\Sigma = \mathbf{SC'.} \rightarrow_2\mathbf{.\Sigma}$
  - $\rightarrow_2.p_{In} \in SC_1.p_{Out} \in sdf_2.P_{Out}$ *and* $\rightarrow_2.p_{Out} \in sdf_3.P_{In} = \mathbf{SC'.} \rightarrow_2\mathbf{.P}$
- $\Sigma = SC_1.\Sigma \cup sdf_3.\Sigma = sdf_1.\Sigma \cup sdf_2.\Sigma \cup sdf_3.\Sigma = sdf_1.\Sigma \cup SC_2.\Sigma = SC'.\ \Sigma$
- $P = P_{In} \cup P_{Out}$ *where:*
  - $P_{In} = SC_1.P_{In} \cup sdf_3.P_{In} = sdf_1.P_{In} \cup sdf_2.P_{In} \cup sdf_3.P_{In} = sdf_1.P_{In} \cup SC_2.P_{In} = \mathbf{SC'.P_{In}}$
  - $P_{Out} = SC_1.P_{Out} \cup sdf_3.P_{Out} = sdf_1.P_{Out} \cup sdf_2.P_{Out} \cup sdf_3.P_{Out} = sdf_1.P_{Out} \cup SC_2.P_{Out} = \mathbf{SC'.P_{Out}}$
- $T = SC_1.T \cup \rightarrow_2.T \cup sdf_3.T = sdf_1.T \cup \rightarrow_1.T \cup sdf_2.T \cup \rightarrow_2.T \cup sdf_3.T = sdf_1.T \cup \rightarrow_1.T \cup SC_2.T = \mathbf{SC'.T}$
- $A = SC_1.A \cup \rightarrow_2.A \cup sdf_3.A = sdf_1.A \cup \rightarrow_1.A \cup sdf_2.A \cup \rightarrow_2.A \cup sdf_3.A = sdf_1.A \cup \rightarrow_1.A \cup SC_2.A = \mathbf{SC'.A}$
- *C:P→Σ is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where:*

$$\forall t \in T, G(t) = \begin{cases} \text{SD-function}.G(t), & t \in sdf_1.T \cup sdf_2.T \cup sdf_3.T \\ Sequence.G(t), & t \in \rightarrow_1.T \cup \rightarrow_2.T \end{cases}$$

- *E:A→Expr is the function associating an expression E(a) to an arc a where E = SD-function.E*
- $I:P_{In} \rightarrow$ *Value is the function associating initial values to the input places, I = SD-function.I*

And thus,

XCGN(SC) = (SC'.Σ, SC'.P, SC'.T, SC'.A, C, G, E, I) = XCGN(SC')

□

4.3.5.2 Distributive Property of Concurrency: (a // b) ➔ c = ((a ➔$_a$ c) // (b ➔$_b$ c))

Consider the following compositions CC, CC' where:

- $CC = (sdf_1 // sdf_2) \rightarrow sdf_3$
- $CC' = (sdf_1 \rightarrow_1 sdf_3) // (sdf_2 \rightarrow_2 sdf_3)$

In order to prove the distributive property of *concurrency (CC = CC')*, we need to prove that *XCGN(CC) = XCGN(CC').*

Proof:

CC = (Σ, P, T, A, C, G, E, I)

- sdf_1, sdf_2 and sdf_3 are SD-functions
- ➔ is a set of Sequence operators, ➔ = {➔$_1$, ➔$_2$} where:
  - ➔.Σ = {➔$_1$.Σ / ➔$_1$.Σ ⊆ sdf_1.Σ} ∪ {➔$_2$.Σ / ➔$_2$.Σ ⊆ sdf_2.Σ)}
  - ➔.P = ➔.P_{In} ∪ ➔.P_{Out}
    - ➔.P_{In} = {(➔$_1$.p_{In} / ➔$_1$.p_{In} ∈ $sdf_1.P_{Out}$) , (➔$_2$.p_{In} / ➔$_2$.p_{In} ∈ $sdf_2.P_{Out}$)}
    - ➔.P_{Out} = {(➔$_1$.p_{In} / ➔$_1$.p_{Out} ∈ $sdf_3.P_{In}$) , (➔$_2$.p_{In} / ➔$_2$.p_{Out} ∈ $sdf_3.P_{In}$)}
- $\Sigma = sdf_1.\Sigma \cup sdf_2.\Sigma \cup sdf_3.\Sigma =$ **CC'.Σ**
- $P = P_{In} \cup P_{Out}$ where:
  - $P_{In} = sdf_1.P_{In} \cup sdf_2.P_{In} \cup sdf_3. .P_{In} =$ **CC'.P_{In}**
  - $P_{Out} = sdf_1.P_{Out} \cup sdf_2.P_{Out} \cup sdf_3.P_{Out} =$ **CC'.P_{Out}**
- $T = sdf_1.T \cup$ ➔$_1$.$T \cup sdf_2.T \cup$ ➔$_2$.$T \cup sdf_3.T =$ **CC'.T**
- $A = sdf_1.A \cup$ ➔$_1$.$A \cup sdf_2.A \cup$ ➔$_2$.$A \cup sdf_3.A =$ **CC'.A**
- *C:P→Σ is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where:*

$$\forall t \in T, G(t) = \begin{cases} \text{SD-function}.G(t), & t \in sdf_1.T \cup sdf_2.T \cup sdf_3.T \\ Sequence.G(t), & t \in \rightarrow_1.T \cup \rightarrow_2.T \end{cases}$$

- *E:A→Expr is the function associating an expression E(a) to an arc a where E = SD-function.E*
- *I:P_{In}→Value is the function associating initial values to the input places, I = SD-function.I*

And thus,

XCGN(CC) = (CC'.Σ, CC'.P, CC'.T, CC'.A, C, G, E, I) = XCGN(CC')

□

4.3.5.3 Associative Property of Parallelism: (a // b) // c = a // (b // c)

Consider the following compositions PC_1, PC_2, PC, PC' where:

- $PC_1 = (sdf_1 // sdf_2)$
- $PC_2 = (sdf_2 // sdf_3)$
- $PC = ((sdf_1 // sdf_2) // sdf_3)$

- $PC' = (sdf_1 // (sdf_2 // sdf_3))$

In order to prove the associative property of *parallelism (PC = PC')*, we need to prove that *XCGN(PC) = XCGN(PC')*.

Proof:

PC = (Σ, P, T, A, C, G, E, I)

- sdf_1, sdf_2, sdf_3 *and* sdf_4 *are SD-functions*
- $\Sigma = PC_1.\Sigma \cup sdf_3.\Sigma = sdf_1.\Sigma \cup sdf_2.\Sigma \cup sdf_3.\Sigma = sdf_1 \cup PC_2.\Sigma = \mathbf{PC'.\Sigma}$
- $P = P_{In} \cup P_{Out}$ *where:*
  - $P_{In} = PC_1.P_{In} \cup sdf_3.P_{In} = sdf_1.P_{In} \cup sdf_2.P_{In} \cup sdf_3.P_{in} = sdf_1.P_{In} \cup PC_2.P_{In} = \mathbf{PC'.P_{In}}$
  - $P_{Out} = PC_1.P_{Out} \cup sdf_3.P_{Out} = sdf_1.P_{Out} \cup sdf_2.P_{Out} \cup sdf_3.P_{Out} = sdf_1.P_{Out} \cup PC_2.P_{Out} = \mathbf{PC'.P_{Out}}$
- $T = PC_1.T \cup sdf_3.T = sdf_1.T \cup sdf_2.T \cup sdf_3.T = sdf_1.T \cup PC_2.T = \mathbf{PC'.T}$
- $A = PC_1.A \cup sdf_3.A = sdf_1.A \cup sdf_2.A \cup sdf_3.A = sdf_1.A \cup PC_2.A = \mathbf{PC'.A}$
- $C: P \rightarrow \Sigma$ *is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where*
  $\forall t \in T, (G(t) = \textit{SD-function}.G(t), \quad t \in sdf_1.T \cup sdf_2.T \cup sdf_3.T)$
- $E: A \rightarrow Expr$ *is the function associating an expression E(a) to an arc a where E = SD-function.E*
- $I: P_{In} \rightarrow Value$ *is the function associating initial values to the Input places, I = SD-function.I*

And thus,

XCGN(PC) = (PC'.Σ, PC'.P, PC'.T, PC'.A, C, G, E, I) = XCGN(PC')

□

4.3.5.4 Commutative Property of Parallelism: (a // b) = (b // a)

Consider the following compositions PC, PC' where:

- $PC = (sdf_1 // sdf_2)$
- $PC' = (sdf_2 // sdf_1)$

In order to prove the commutative property of *parallelism (PC = PC')*, we need to prove that *XCGN(PC) = XCGN(PC')*.

Proof:

PC = (Σ, P, T, A, C, G, E, I)

- *sdf_1, sdf_2 are SD-functions*
- *$\Sigma = sdf_1.\Sigma \cup sdf_2.\Sigma = sdf_2.\Sigma \cup sdf_1.\Sigma$ = **PC'.Σ***
- *$P = P_{In} \cup P_{Out}$ where:*
  - *$P_{In} = sdf_1.P_{In} \cup sdf_2.P_{In} = sdf_2.P_{In} \cup sdf_1.P_{In}$ = **$PC'.P_{In}$***
  - *$P_{Out} = sdf_1.P_{Out} \cup sdf_2.P_{Out} = sdf_2.P_{Out} \cup sdf_1.P_{Out}$ = **$PC'.P_{Out}$***
- *$T = sdf_1.T \cup sdf_2.T = sdf_2.T \cup sdf_1.T$ = **PC'.T***
- *$A = sdf_1.A \cup sdf_2.A = sdf_2.A \cup sdf_1.A$ = **PC'.A***
- *$C: P \rightarrow \Sigma$ is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where:*

  $$\forall t \in T, \quad G(t) = \text{SD-function}.G(t), \quad t \in sdf_1.T \cup sdf_2.T$$
- *$E: A \rightarrow$ Expr is the function associating an expression E(a) to an arc a where E = SD-function.E*
- *$I: P_{In} \rightarrow$ Value is the function associating initial values to the Input places, I = SD-function.I*

And thus,

XCGN(PC) = (PC'.Σ, PC'.P, PC'.T, PC'.A, C, G, E, I) = XCGN(PC')

□

4.3.5.5 Associative Property of Concurrency: ((a // b) // c) ➔ d = (a // (b // c)) ➔ d

Consider the following compositions CC_1, CC_2, CC, CC' where:

- *$CC_1 = (sdf_1 // sdf_2) \rightarrow sdf_4 = (sdf_1 \rightarrow_1 sdf_4) // (sdf_2 \rightarrow_2 sdf_4)$*
- *$CC_2 = (sdf_2 // sdf_3) \rightarrow sdf_4 = (sdf_2 \rightarrow_2 sdf_4) // (sdf_3 \rightarrow_3 sdf_4)$*
- *$CC = ((sdf_1 // sdf_2) // sdf_3) \rightarrow sdf_4$*
- *$CC' = (sdf_1 // (sdf_2 // sdf_3)) \rightarrow sdf_4$*

In order to prove the associative property of *concurrency (CC = CC')*, we need to prove that *XCGN(CC) = XCGN(CC')*

Proof:

$CC = (\Sigma, P, T, A, C, G, E, I)$

- sdf_1, sdf_2, sdf_3 and sdf_4 are SD-functions
- $\rightarrow$ is a set of Sequence operators, $\rightarrow = \{\rightarrow_{12}, \rightarrow_3\} = \{\rightarrow_1, \rightarrow_2, \rightarrow_3\}$ where:
  - $\rightarrow.\Sigma = \{\rightarrow_1.\Sigma / \rightarrow_1.\Sigma \subseteq sdf_1.\Sigma\} \cup \{\rightarrow_2.\Sigma / \rightarrow_2.\Sigma \subseteq sdf_2.\Sigma)\} \cup \{\rightarrow_3.\Sigma / \rightarrow_3.\Sigma \subseteq sdf_3.\Sigma)\}$
  - $\rightarrow.P = \rightarrow.P_{In} \cup \rightarrow.P_{Out}$
    - $\rightarrow.P_{In} = \{(\rightarrow_1.p_{In} / \rightarrow_1.p_{In} \in sdf_1.P_{Out}), (\rightarrow_2.p_{In} / \rightarrow_2.p_{In} \in sdf_2.P_{Out}), (\rightarrow_3.p_{In} / \rightarrow_3.p_{In} \in sdf_3.P_{Out})\}$
    - $\rightarrow.P_{Out} = \{(\rightarrow_1.p_{In} / \rightarrow_1.p_{Out} \in sdf_4.P_{In}), (\rightarrow_2.p_{In} / \rightarrow_2.p_{Out} \in SDF_3.P_{In}), (\rightarrow_3.p_{In} / \rightarrow_3.p_{Out} \in sdf_4.P_{In})\}$
- $\Sigma = CC_1.\Sigma \cup sdf_3.\Sigma \cup sdf_4.\Sigma = sdf_1.\Sigma \cup sdf_2.\Sigma \cup sdf_3.\Sigma \cup sdf_4.\Sigma = sdf_1 \cup CC_2.\Sigma \cup sdf_4.\Sigma = \mathbf{CC'.\Sigma}$
- $P = P_{In} \cup P_{Out}$ where:
  - $P_{In} = CC_1.P_{In} \cup sdf_3.P_{In} \cup sdf_4.P_{In} = sdf_1.P_{In} \cup sdf_2.P_{In} \cup sdf_3.P_{in} \cup sdf_4.P_{In} = sdf_1.P_{In} \cup CC_2.P_{In} \cup sdf_4.P_{In} = \mathbf{CC'.P_{In}}$
  - $P_{Out} = CC_1.P_{Out} \cup sdf_3.P_{Out} \cup sdf_4.P_{Out} = sdf_1.P_{Out} \cup sdf_2.P_{Out} \cup sdf_3.P_{Out} \cup sdf_4.P_{Out} = sdf_1.P_{Out} \cup CC_2.P_{Out} \cup sdf_4.P_{Out} = \mathbf{CC'.P_{Out}}$
- $T = CC_1.T \cup \rightarrow_{12}.T \cup sdf_3.T \cup \rightarrow_3.T \cup sdf_4.T = sdf_1.T \cup \rightarrow_1.T \cup sdf_2.T \cup \rightarrow_2.T \cup sdf_3.T \cup \rightarrow_3.T \cup sdf_4.T = sdf_1.T \cup \rightarrow_1.T \cup CC_2.T \cup \rightarrow_{23}.T \cup sdf_4.T = \mathbf{CC'.T}$
- $A = CC_1.A \cup \rightarrow_{12}.A \cup sdf_3.A \cup \rightarrow_3.A \cup sdf_4.A = sdf_1.A \cup \rightarrow_1.A \cup sdf_2.A \cup \rightarrow_2.A \cup sdf_3.A \cup \rightarrow_3.A \cup sdf_4.A = sdf_1.A \cup \rightarrow_1.A \cup CC_2.A \cup \rightarrow_{23}.A \cup sdf_4.A = \mathbf{CC'.A}$
- $C: P \rightarrow \Sigma$ is the function associating a color to each place where C = SD-function.C
- G: is a function over T where
  - $\forall t \in T, G(t) = \begin{cases} \text{SD-function}.G(t), & t \in sdf_1.T \cup sdf_2.T \cup sdf_3.T \cup sdf_4.T \\ \text{Sequence}.G(t), & t \in \rightarrow_1.T \cup \rightarrow_2.T \cup \rightarrow_3.T \end{cases}$
- $E: A \rightarrow Expr$ is the function associating an expression E(a) to an arc a where E = SD-function.E
- $I: P_{In} \rightarrow Value$ is the function associating initial values to the input places, I = SD-function.I

And thus,

XCGN(CC) = (CC'.Σ, CC'.P, CC'.T, CC'.A, C, G, E, I) = XCGN(CC')

□

4.3.5.6 Commutative Property of Concurrency: (a // b) → c = (b // a) → c

Consider the following compositions CC, CC' where:

- $CC = (sdf_1 // sdf_2) \rightarrow sdf_3$
- $CC' = (sdf_2 // sdf_1) \rightarrow sdf_3$

In order to prove the commutative property of concurrency ($CC = CC'$), we need to prove that $XCGN(CC) = XCGN(CC')$

Proof:

CC = (Σ, P, T, A, C, G, E, I)

- *sdf_1, sdf_2 and sdf_3 are SD-functions*
- *$\rightarrow$ is a set of Sequence operators, $\rightarrow = \{\rightarrow_1, \rightarrow_2\}$ where:*
  - *$\rightarrow.\Sigma = \{\rightarrow_1.\Sigma / \rightarrow_1.\Sigma \subseteq sdf_1.\Sigma\} \cup \{\rightarrow_2.\Sigma / \rightarrow_2.\Sigma \subseteq sdf_2.\Sigma)\} = \{\rightarrow_2.\Sigma / \rightarrow_2.\Sigma \subseteq sdf_2.\Sigma)\} \cup \{\rightarrow_1.\Sigma / \rightarrow_1.\Sigma \subseteq sdf_1.\Sigma\} =$ **$CC'.\rightarrow.\Sigma$***
  - *$\rightarrow.P = \rightarrow.P_{In} \cup \rightarrow.P_{Out}$*
    - *$\rightarrow.P_{In} = \{(\rightarrow_1.p_{In} / \rightarrow_1.p_{In} \in sdf_1.P_{Out}), (\rightarrow_2.p_{In} / \rightarrow_2.p_{In} \in sdf_2.P_{Out})\} = \{(\rightarrow_2.p_{In} / \rightarrow_2.p_{In} \in sdf_2.P_{Out}), (\rightarrow_1.p_{In} / \rightarrow_1.p_{In} \in sdf_1.P_{Out})\} =$ **$CC'.\rightarrow.P_{In}$***
    - *$\rightarrow.P_{Out} = \{(\rightarrow_1.p_{In} / \rightarrow_1.p_{Out} \in sdf_3.P_{In}), (\rightarrow_2.p_{In} / \rightarrow_2.p_{Out} \in sdf_3.P_{In})\} = \{(\rightarrow_2.p_{In} / \rightarrow_2.p_{Out} \in sdf_3.P_{In}), (\rightarrow_1.p_{In} / \rightarrow_1.p_{Out} \in sdf_3.P_{In})\} =$ **$CC'.\rightarrow.P_{Out}$***
- *$\Sigma = sdf_1.\Sigma \cup sdf_2.\Sigma \cup sdf_3.\Sigma = sdf_2.\Sigma \cup sdf_1.\Sigma \cup sdf_3.\Sigma =$ **$CC'.\Sigma$***
- *$P = P_{In} \cup P_{Out}$ where:*
  - *$P_{In} = sdf_1.P_{In} \cup sdf_2.P_{In} \cup sdf_3. .P_{In} = sdf_2.P_{In} \cup sdf_1.P_{In} \cup sdf_3. .P_{In} =$ **$CC'.P_{In}$***
  - *$P_{Out} = sdf_1.P_{Out} \cup sdf_2.P_{Out} \cup sdf_3.P_{Out} = sdf_2.P_{Out} \cup sdf_1.P_{Out} \cup sdf_3.P_{Out} =$ **$CC'.P_{Out}$***
- *$T = sdf_1.T \cup \rightarrow_1.T \cup sdf_2.T \cup \rightarrow_2.T \cup sdf_3.T = sdf_2.T \cup \rightarrow_2.T \cup sdf_1.T \cup \rightarrow_1.T \cup sdf_3.T =$ **$CC'.T$***
- *$A = sdf_1.A \cup \rightarrow_1.A \cup sdf_2.A \cup \rightarrow_2.A \cup sdf_3.A = sdf_2.A \cup \rightarrow_2.A \cup sdf_1.A \cup \rightarrow_1.A \cup sdf_3.A =$ **$CC'.A$***
- *$C:P \rightarrow \Sigma$ is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where:*

$$\forall t \in T, G(t) = \begin{cases} \text{SD-function}.G(t), & t \in sdf_1.T \cup sdf_2.T \cup sdf_3.T \\ Sequence.G(t), & t \in \rightarrow_1.T \cup \rightarrow_2.T \end{cases}$$

- *$E:A \rightarrow Expr$ is the function associating an expression E(a) to an arc a where E = SD-function.E*
- *$I:P_{In} \rightarrow Value$ is the function associating initial values to the input places, I = SD-function.I*

And thus,

XCGN(CC) = (CC'.Σ, CC'.P, CC'.T, CC'.A, C, G, E, I) = XCGN(CC')

□

4.3.5.7 1st identity Property of Sequence: $a \rightarrow_a \Phi = a$

Consider the following composition:

- *$SC = sdf \rightarrow \Phi$*

In order to prove the identity property of sequence (*SC = sdf)*, we need to prove that *XCGN(SC) = XCGN(sdf)*

Proof:

SC = (Σ, P, T, A, C, G, E, I)

- *Sdf is an SD-function and Φ is an empty net.*
- ➔ *is a Sequence operators where based on the Serial Composition* $SC = \prod_{i=0}^{0} SDF_i \rightarrow_i$ *:*
  - $\rightarrow_0$ *= (Ø, Ø, Ø, Ø, C, G, E, I) in an empty CP-Net*
- $\Sigma = sdf_1.\Sigma \cup \Phi.\Sigma = sdf_1.\Sigma$
- $P = P_{In} \cup P_{Out}$ *where:*
  - $P_{In} = sdf_1.P_{In} \cup \Phi.P_{In} = sdf_1.P_{In}$
  - $P_{Out} = sdf_1.P_{Out} \cup \Phi.P_{Out} = sdf_1.P_{Out}$
- $T = SC_1.T \cup \rightarrow.T \cup \Phi.T = sdf_1.T$
- $A = sdf_1.A \cup \rightarrow.A \cup \Phi.A = sdf_1.A$
- *C:P→Σ is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where:*

$$\forall t \in T, G(t) = \text{SD-function}.G(t), \quad t \in sdf.T$$

- *E:A→Expr is the function associating an expression E(a) to an arc a where E = SD-function.E*
- $I:P_{In} \rightarrow$ *Value is the function associating initial values to the input places, I = SD-function.I*

And thus,

XCGN(SC) = (sdf.Σ, sdf.P, sdf.T, sdf.A, C, G, E, I) = XCGN(sdf)

□

4.3.5.8 2nd Identity Property of Sequence: Φ ➔ a = Φ

Consider the following composition:

- *SC* = Φ ➔ *sdf*

In order to prove the identity property of sequence (*SC = Φ)*, we need to prove that *XCGN(SC) = XCGN(Φ)*

***Proof*:**

SC = (Σ, P, T, A, C, G, E, I)

- *Sdf is a SD-function and Φ is an empty net.*
- ➔ *is a Sequence operators where:*
  - ➔.Σ = Ø / ➔.$\Sigma \subseteq \Phi.\Sigma$ and $\Phi.\Sigma$ = Ø
  - ➔.P = ➔.$P_{In} \cup$ ➔.P_{Out}
    - ➔.P_{In} = Ø / ➔.$\Sigma \subseteq \Phi.P_{In}$ and $\Phi.P_{In}$ = Ø
    - ➔.P_{Out} = Ø / ➔.P_{Out} = ➔.G(➔.t) = ➔.P_{In} = Ø *(cf.* Definition 4.17*)*
  - ➔ *= (Ø, Ø, Ø, Ø, C, G, E, I) in an empty CP-Net and thus based on the Sequence Definition, sdf will always have an empty input place (sdf.*P_{In}*= Ø) and can never fire and the output will result in an empty place (sdf.*P_{Out}*=Ø). Thus:*
- Σ = Ø
- P = Ø
- T = Ø
- A = Ø
- *C:P→Σ is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where:*

$$\forall t \in T, G(t) = \textit{SD-function}.G(t), \quad t \in sdf.T$$

- *E:A→Expr is the function associating an expression E(a) to an arc a where E = SD-function.E*
- *I:*P_{In}*→Value is the function associating initial values to the Input places, I = SD-function.I*

And thus,

XCGN(SC) = (Ø, Ø, Ø, Ø, C, G, E, I) = XCGN(Φ)

□

4.3.5.9 1st and 2nd Identity Property of Parallelism: a //Φ = Φ // a = a

Based on the commutative property of *parallelism*, ***a //Φ = Φ // a***.

In order to prove "a //Φ = a", consider the following composition:

- *PC = sdf //* Φ

In order to prove the identity property of sequence (*PC = sdf)*, we need to prove that *XCGN(PC) = XCGN(sdf)*

Proof:

PC = (Σ, P, T, A, C, G, E, I)

- *Sdf is a SD-function and Φ is an empty net.*
- *Σ = sdf.Σ ∪Φ.Σ =* ***sdf.Σ***
- *P = P_{In} ∪P_{Out} where:*
  - *P_{In} = sdf.P_{In} ∪ Φ.P_{In} =* ***sdf.P_{In}***
  - *P_{Out} = sdf.P_{Out} ∪ Φ.P_{Out} =* ***sdf.P_{Out}***
- *T = SC.T ∪ ➔.T ∪ Φ.T =* ***sdf.T***
- *A = sdf.A ∪ ➔.A ∪ Φ.A =* ***sdf.A***
- *C:P→Σ is the function associating a color to each place where C = SD-function.C*
- *G: is a function over T where:*

$$\forall t \in T, G(t) = \textit{SD-function}.G(t), \quad t \in sdf.T$$

- *E:A→Expr is the function associating an expression E(a) to an arc a where E = SD-function.E*
- *I:P_{In}→Value is the function associating initial values to the Input places, I = SD-function.I*

And thus,

XCGN(PC) = (*sdf*.Σ, *sdf*.P, *sdf*.T, *sdf*.A, C, G, E, I) = XCGN(*sdf*)

□

After defining the language and its syntax, we give now an illustration of scenario 1(cf. Chapter 1 Section 1.2.1) in XCDL.

4.3.6 Illustration

In scenario 1, the user wants to create a manipulation operation that filters his library (books.xml, cf. Figure 8) and retrieve all the books published in the year 2001 and which are guide books related to XML. These goals can be achieved in XCDL as shown in Figure 15. Note that the following composition is one way of solving the issue, there can be others depending on the user's perspectives.

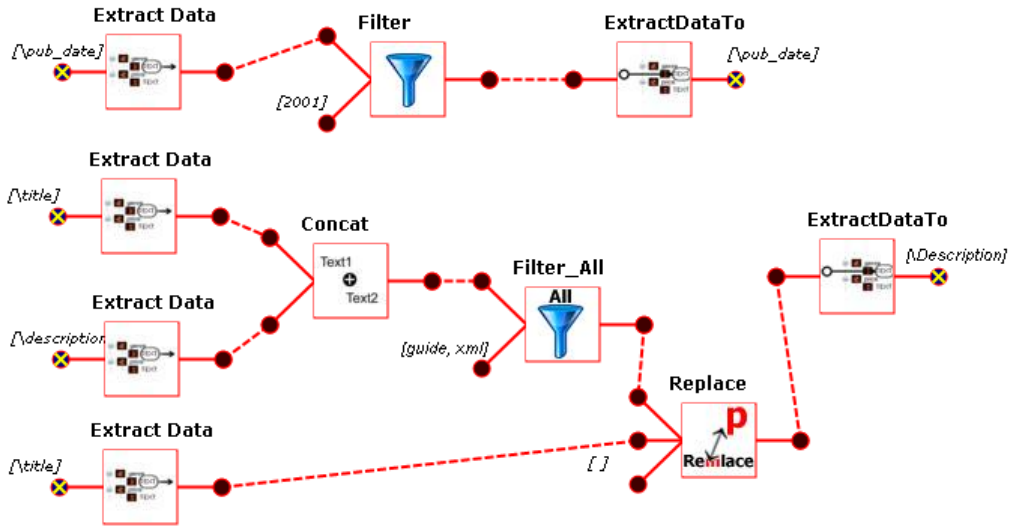


Figure 15: Illustration of scenario 1 in XCDL

In order to create his filter, the user composes 2 parallel filters. The first one selects all books published in 2001. It is defined as a serial composition:

$$Filter_1 = ExtractData \rightarrow Filter \rightarrow ExtractDataTo$$

The second filter groups both the title and description of the books and retrieves only those which are guides and XML related. It is defined as a combination of concurrent and serial compositions:

$$Filter_2 = (((ExtractData//ExtractData) \rightarrow Concat \rightarrow Filter\_All)//ExtractData) \rightarrow Replace \rightarrow ExtractDataTo$$

The main filter is defined as a parallel composition between $Filter_1$ and $Filter_2$:

$$PC\_Filter = Filter_1 // Filter_2 = (ExtractData \rightarrow Filter \rightarrow ExtractDataTo) // ((((ExtractData//ExtractData) \rightarrow Concat \rightarrow Filter\_All)//ExtractData) \rightarrow Replace \rightarrow ExtractDataTo)$$

The functions used in this composition are described briefly in Table 4.

Table 4: Functions used in scenario 1

Function Name	Description
ExtractData	Extracts textual nodes from an XCD-tree using Xpath Expressions
Filter	Filters data containing based on a single key word
Concat	Concatenates 2 strings together
Filter_All	Filters a paragraph based on several keywords
Replace	Replaces the occurrence of a String with another
ExtractDataTo	Transforms strings into textual nodes to be reinserted in an XCD tree using XPath Expressions

In this Section, we defined XCDL, a generic composition language which allows users to visually create functional compositions oriented towards XML data. The language syntax was defined in CP-Nets. The components and composition results are all CP-Nets, thus, allowing the composition to express true concurrency and parallelism. To execute these compositions, they should be translated into a machine code. Therefore, we define next our compiler which is responsible of translating the XCDL syntax into a machine code executable by the Runtime Environment.

4.4 XA2C Compiler

In the conception of most DFVPLs, one of the major issues always being raised was:

"When does the Language end, and the Runtime Environment begins?"

In our case, we answer that question by taking advantage of the Mashup approach and defining a middleware, the compiler module, between the language and the runtime environment as depicted in Figure 2. This module plays the role of a compiler that translates the XCDL syntax into a machine language readable and executable in the XA2C runtime environment.

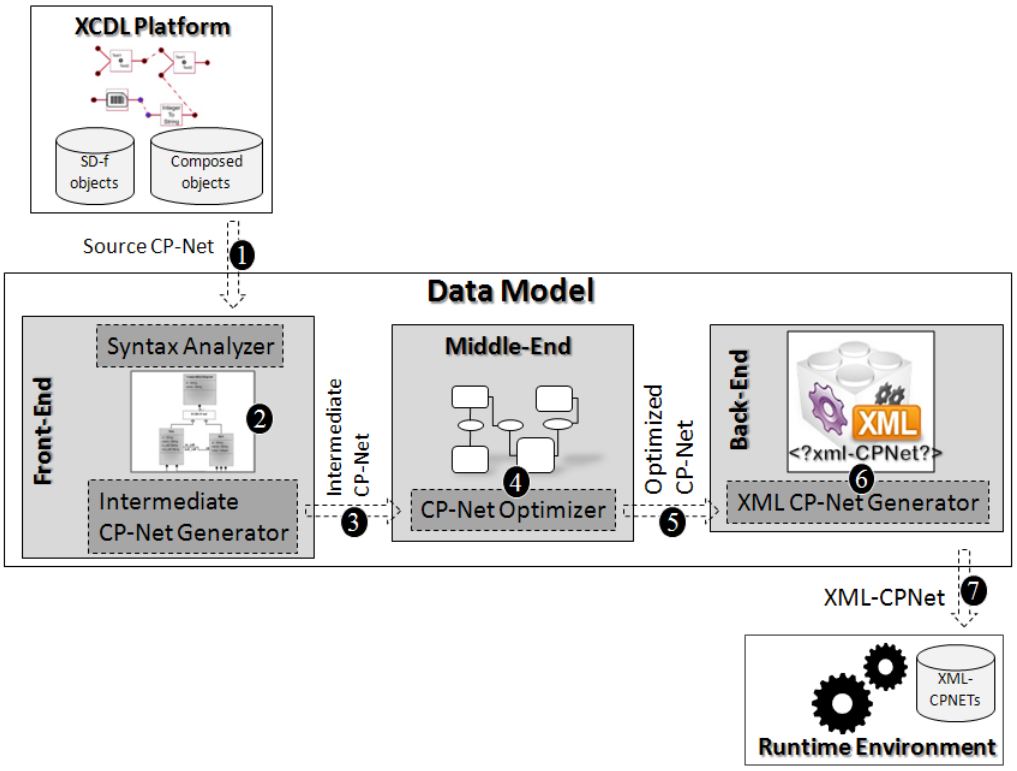


Figure 16: XA2C compiler architecture

As depicted in Figure 16, the compiler's structure contains 3 modules: (i) the Front-End, (ii) the Middle-End and (iii) the Back-End. The Syntax Analyzer in the Front-End receives a high-level petri net, Source CP-Net, from the XCDL platform, and checks it with the internal data model (cf. Figure 22) defined based on XCGN. Once the Source CP-Net is validated, it is sent to the Intermediate CP-Net Generator. The latter transforms it into a CP-Net object and transmits it as an intermediate CP-Net to the Middle-End module. The Intermediate CP-Net is then translated into a dataset based on the internal data model and compliant to a CP-Net defined in XCGN. The CP-Net Optimizer will optimize it by removing any redundancies and passive sub-nets. The optimized CP-Net is transferred to the XML CP-Net Generator which uses an XML-based interchange format for CP-Nets, inspired and adapted from the PNML (Petri Net Markup Language) [55], to transform the optimized CP-Net into an XML-CPNet. The XML-CPNet is sent to the Runtime Environment where it can be executed later in the future.

4.4.1 Front-End

In general terms, the Front-End checks whether a program is correctly written in terms of the programming language syntax and semantics. In our case, since the language is

visual, the Front-End checks whether a program is correctly drawn in terms of CP-Nets based on XCGN.

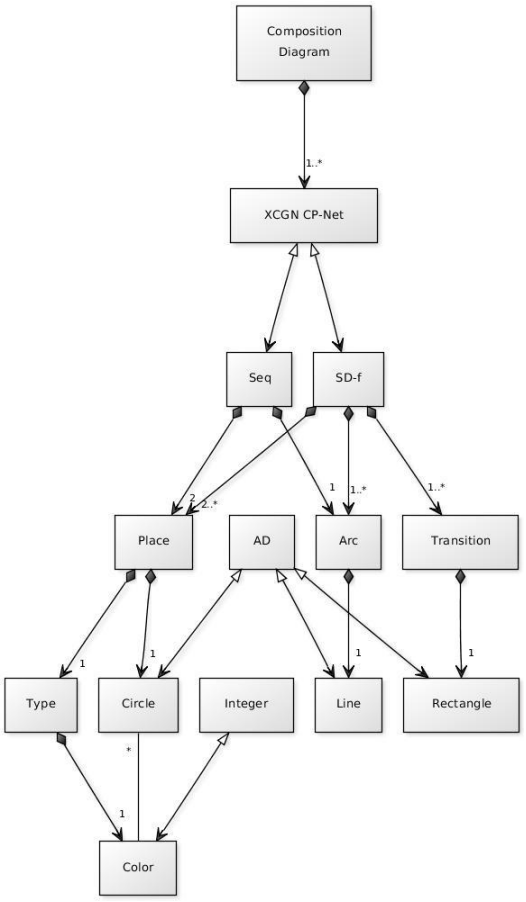


Figure 17: Front-End data types

The Front-End works in 2 modes: (i) Component Validation mode for *SD-functions'* validation and (ii) Composition Validation mode. This is based on the XCDL, which is divided into *SD-functions* and Compositions between different instances of *SD-functions* (cf. Section 4.3.2).

4.4.1.1 Component Validation Mode

The Front-End enters the Component Validation Mode when a new *SD-function* is being added to the system. Before, any *SD-function* can be inserted to the XCDL library, the Syntax Analyzer needs to validate it with the *SD-function data type* (cf. Figure 18) defined in correspondence with the *SD-function's* syntax (cf. Definition 4.16). Each *SD-function* is translated into as a separate object of type *SD-function* as shown in Figure 18. Since an *SD-function* is defined as a CP-Net (cf. Definition 4.16), its data type is composed of Places, Arcs and a Transition that are associated to different XCDL-GRs respectively, Circle, Line and Rectangle. The translation to an

SD-function object is ensured via an *SDf-t* translation syntax transforming the CP-Net elements into objects with the corresponding attributes and dependencies as defined in XCGN.

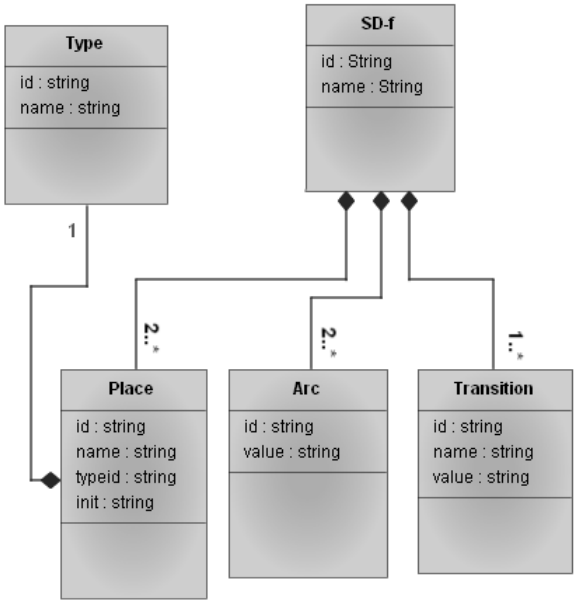


Figure 18: SD-function data type

Definition 4.25-*SDf-t* *is a translation syntax for SD-functions from an XCGN based syntax into an object of SD-function data type*[7] *and is defined as:*

$$\textbf{SDf-t} = < DT_{sdf}, DT_{\Sigma}, DT_{P}, DT_{T}, DT_{A} > \textit{ where:}$$

- DT_{sdf}: ***SD-function → SD-f*** *is a function associating an object sdf of type SD-f (cf.* Figure 18*) to an SD-function SDF:*
  - $DT_{sdf}(SDF) = sdf \begin{cases} sdf.id = cpn\_l(SDF).id \\ sdf.name = cpn\_l(SDF).name \end{cases}$
- DT_{Σ}: $\Sigma \rightarrow$ ***Type*** *is a function associating an object type ∈ Type to an XCGN type ε ∈ Σ:*
  - $DT_{\Sigma}(\varepsilon) = type \begin{cases} type.id = l(\varepsilon).id \\ type.name = l(\varepsilon).name \end{cases}$
- DT_{P}: $P \rightarrow$ ***Place*** *is a function associating an object pl ∈ Place to a place p ∈ P:*

[7] Each component in XCDL is considered to have an identifier "id" and a name.

- $DT_P(p) = pl \begin{cases} pl.id = l(p).id \\ pl.name = l(p).name \\ pl.typeid = l(C(p)).id \\ pl.init = I(p) \end{cases}$

- ***DT_T: T → Transition*** *is a function associating an object tr ∈ Transition to a transition t ∈ T:*
  - $DT_T(t) = tr \begin{cases} tr.id = l(t).id \\ tr.name = l(t).name \\ tr.value = G(t) \end{cases}$
- ***DT_A: A → Arc*** *is a function associating an object ar ∈ Arc to an arc a ∈ A:*
  - $DT_A(a) = Ar \begin{cases} ar.id = l(a.p).id + l(a.t).id \\ ar.value = E(a) \end{cases}$

As an example, consider the *SD-function* "Filter" shown in Figure 19. This function is defined in the XCDL syntax as follows:

Filter = (Σ, P, T, A, C, G, E, I) *where*:

- *Σ = {String}*
- *$P = P_{In} \cup P_{Out}$ = {In_Str1, In_Str2}∪{Out_Str}*
- *T = {t }*
- *A = ({In_Str1, In_Str2} x {t }) ∪({t} x {Out_Str})*
- *C:P→Σ where C(In_Str1) = C(In_Str2) = C(Out_Str) = String*
- *G:{t}→ S where G(t)= String_functions.Filter and Type(G(t)) = C(Out_Str) = String where String_functions is the DLL containing String manipulation functions and String_functions.Filter is a function that filters incoming strings if they contain In_Str2.*
- *E:A→Expr:*
  - *Expr={M(In_Str), G(t)} is a set of expressions where:*

$$\forall E(a) \in Expr:\ E(a) = \begin{cases} M(a.p)\ if\ a.p \neq p_{Out} \\ G(a.t)\ otherwise \end{cases}$$

- *$I:P_{In}$→Value where I(In_Str1) = "" and I(In_Str2) = "keyword"*[8]

[8] Keyword in this case is the initial value given by the user which will be used as a filtering criteria

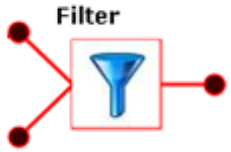


Figure 19: Filter SD-function

The Filter function syntax will be translated into the following objects as presented in Table 5.

Table 5: Filter *SD-function* translation from XCGN to objects

XCGN components	Object data type
Filter	SD-f
String	Type
In_Str1, In_Str1 and Out_Str	Place
T	Transition
In_Str1 x t, In_Str2 x t and t x Out_Str	Arc

If the SD-function is well translated into an *SD-f* object with all its attributes fitting correctly in the *SD-f* data type, the *SD-f* object is then forwarded to Middle-End module as an Intermediate CP-Net where it is translated into a dataset which is validated by the *SD-function* data model presented in Figure 22.

4.4.1.2 Composition Validation Mode

When the user is in the process of composing his manipulation operation, the Front-End is in the Composition Validation Mode. Similar to the Component Validation mode, the Syntax Analyzer checks and validates the composition on every event (Insert, Delete or update of an *SD-function* instance).

The first process when validating the current composition is its translation into a Composition Diagram Object based on the *Composition Diagram data type* shown in Figure 20. The *Composition data type* itself is defined as a projection of a CP-Net-based composition between several instances of XCGN-based CP-Nets generated from *SD-function* instances mapped concretely with instances of the sequence operator as defined in Section 4.3.4.2 (i.e., Serial, Parallel and Concurrent Compositions).

A composition in XCDL, is first of all a CP-Net compliant to XCGN. This CP-Net is not built in the same way as traditional CP-Nets straight from places and transitions. Instead it is based on instances of existing CP-Nets defined either as *SD-functions* or

sequences. Therefore, as shown in Figure 17, a Composition diagram object is an association of multiple XCGN based CP-Nets which can be typed either to an *SD-f data type* (cf. Figure 18) or a *Sequence data type*. The *Sequence data type*, it is defined of 2 Places and an Arc. No Transitions are required since in the sequence operator syntax, the transition simply maps the input to the output place. Therefore, the transition in this case can be omitted. A *Composition-t*, translation syntax, has been defined, facilitating the translation from the XCGN-based syntax to the Composition Diagram data type.

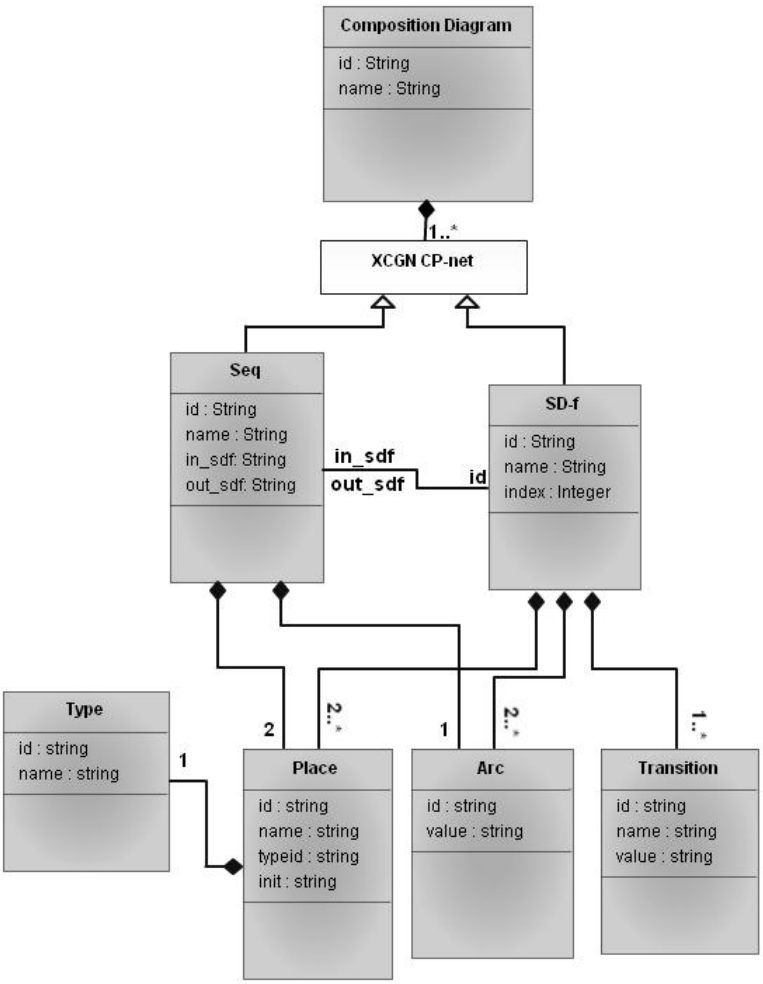


Figure 20: Composition diagram data type

Definition 4.26-*Composition-t* *is a translation syntax for compositions from an XCGN based syntax into an object of Composition data type and is defined as:*

Composition-t $= <$ ***DT_{comp}, DT_{SDf-t}, DT_{seq}*** $>$ *where:*

- ***DT_{comp}: XCGN_Compotision → Composition_Diagrame*** *is a function associating an object cd of type Composition_Diagram (cf.* Figure 20*) to a serial, parallel or concurrent composition c:*
  - $DT(c) = cd \begin{cases} cd.id = cpn\_l(c).id \\ cd.name = cpn\_l(c).name \end{cases}$
- ***DT_{SDf-t}*** $= <$ ***DT_{sdf}, DT_{Σ}, DT_{P}, DT_{T}, DT_{A}*** $>$ *is a translation syntax for an instance of an SD-function SDF_i from an XCGN based syntax into an object of SD-function data type where i is the i_{th} inserted XCGN based CP-Net (SD-function or Sequence) instance. DT_{SDf-t} is similar to the SDf-t syntax with the modification of:*
  - $DT_{sdf}(SDF_i) = sdf \begin{cases} sdf.id = cpn\_l(SDF).id + i \\ sdf.name = cpn\_l(SDF).name \\ sdf.index = i \end{cases}$
  - $DT_P(p) = pl \begin{cases} pl.id = l(p).id + sdf.id \\ pl.name = l(p).name \\ pl.typeid = l\big(C(p)\big).id \\ pl.init = I(p) \end{cases}$
- ***DT_{seq}: Sequence → Seq*** *is a function associating an object s of type Seq (cf.* Figure 20*) to a Sequence ➔$_i$ where ➔$_i$ is the i_{th} inserted XCGN based CP-Net (SD-function or Sequence) instance:*
  - $DT_{seq}(➔_i) = s \begin{cases} s.id = i \\ s.name = cpn\_l(➔).name \\ s.in\_sdf = cpn\_l(➔_i.SDF_{In}).id \\ s.out\_sdf = cpn\_l(➔_i.SDF_{Out}).id \end{cases}$

As an example, consider the Serial Composition $Filter_1$ defined in scenario 1 (cf. Section 4.3.6) and presented in Figure 21.

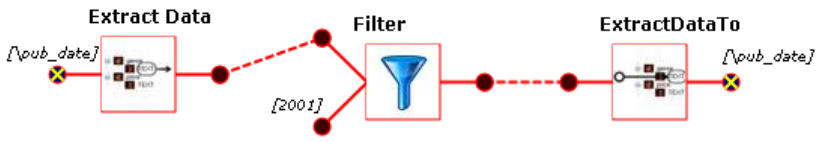


Figure 21: Composition instance

This simple composition captures the books from the XML flow provided in the document "books.xml" that have been published in 2001, and then forwards them

back to the flow. Three simple *SD-functions* (ExtractData, Filter and ExtractDataTo) are used mapped sequentially together with 2 Sequence operators (S1 and S2).
The *SD-functions* ExtractData and ExtractDataTo, and the Sequence Operators S1 and S2 are defined in the XCDL syntax here below.

ExtractData = (Σ, P, T, A, C, G, E, I) *where*:

- *Σ = { XCD-Node:Text, String}*
- *$P = P_{In} \cup P_{Out}$ = {In_XCD}∪{Out_Str}*
- *T = {t }*
- *A = ({In_XCD} x {t }) ∪ ({t} x {Out_Str})*
- *C:P→Σ where C(In_XCD) =XCD-Node:Text and C(Out_Str) = String*
- *G:{t}→ S where G(t)= XCDtree_functions.Extracttext and Type(G(t)) = C(Out_Str) = String where XCDtree_functions is the DLL containing XCDtree related functions and XCDtree_functions.Extracttext is a function that retrieves a string value from an XML Element.*
- *E:A→Expr:*
    - *Expr={M(In_XCD)), G(t)} is a set of expressions where:*

$$\forall expr \in Expr:\ expr = \begin{cases} M(a.p)\ if\ a.p \neq p_{Out} \\ G(a.t)\ otherwise \end{cases}$$

- *$I:P_{In}$→Value where I(In_XCD) = Null*

ExtractDataTo = (Σ, P, T, A, C, G, E, I) *where*:

- *Σ = { XCD-Node:Text, String}*
- *$P = P_{In} \cup P_{Out}$ = {In_Str}∪{Out_XCD}*
- *T = {t }*
- *A = ({In_Str} x {t }) ∪ ({t} x {Out_XCD})*
- *C:P→Σ where C(In_STR) =String and C(Out_XCD) = XCD-Node:Text*
- *G:{t}→ S where G(t)= XCDtree_functions.Extracttextto and Type(G(t)) = C(Out_XCD) = XCD-Node:Text XCDtree_functions.Extracttext is a function that replaces the existing string value an XML Element.*
- *E:A→Expr:*
    - *Expr={M(In_Str)), G(t)} is a set of expressions where:*

$$\forall expr \in Expr:\ expr = \begin{cases} M(a.p)\ if\ a.p \neq p_{Out} \\ G(a.t)\ otherwise \end{cases}$$

- *$I:P_{In}$→Value where I(In_Str) = “”*

Both sequence operator S1 and S2 have similar syntax as shown in the following.

S1= S2 = (Σ, P, T, A, C, G, E, I) *where:*

- *Σ = String*
- *$P = \{p_{In}\} \cup P_{Out} = \{p_{Out}\}$*
- *T = {t} where t contains the sequence operator*
- *$A = (\{p_{In}\} \times \{t\}) \cup (\{t\} \times \{p_{Out}\})$*

- $C:P \rightarrow \Sigma$ *where* $C(p_{In})=C(p_{Out})=String$
- $G(t)=M(p_{In}) \wedge Type(G(t)) = C(p_{In})$
- $E:A \rightarrow Expr$:
    - *Expr is a set of expressions where:*

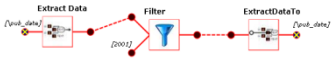
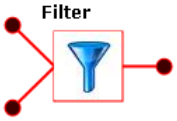
$$\forall expr \in Expr:$$

$$expr = \begin{cases} M(a.p) \; if \; a.p \neq p_{Out} \\ G(a.t) \; otherwise \end{cases}$$

- $I:P_{Out} \rightarrow Value \; I(p_{In}) = \text{“”}$

The Composition syntax will be translated into the following objects:

Table 6: Composition translation from XCGN to objects

XCGN components	Object data type	Graphical Representation
Composition	Composition Diagram	
ExtractData	SD-f	
ExtractData.XCD-Node:Text, ExtractData.String	Type	
ExtractData.In_XCD and ExtractData.Out_Str	Place	
ExtractData.t	Transition	
ExtractData.(In_XCD x t) and ExtractData.(t x Out_Str)	Arc	
Filter	SD-f	
Filter.String	Type	
Filter.In_Str and ToUpper.Out_Str	Place	
Filter.t	Transition	
Filter.(In_Str x t) and ToUpper .(t x Out_Str)	Arc	
ExtractDataTo	SD-f	
ExtractDataTo.XCD-Node:Text, ExtractDataTo.String	Type	
ExtractDataTo.In_Str and ExtractDataTo.Out_XCD	Place	
ExtractDataTo.t	Transition	
ExtractDataTo.(In_Str x t) and ExtractDataTo.(t x Out_XCD)	Arc	
S1	Sequence	
S1.String	Type	
S1.P_{In} and S1.P_{Out}	Place	
S1.(a)	Arc	
S2	Sequence	
S2.String	Type	
S2.P_{In} and S1.P_{Out}	Place	
S2.(a)	Arc	

Similar to the Component Validation mode, if the Composition is well translated into a Composition Diagram object with all its sub data types well defined, the Composition object is then transmitted as an Intermediate CP-Net to the Middle-End module to be transformed into a dataset which is validated by the Composition relational schema presented in Figure 22.

4.4.2 Middle-End

The Middle-End component is module for transforming the Intermediate CP-Net defined as a data object into a dataset and applying any possible optimizations in order to facilitate the transformation to a machine code.

In the Middle-End module, a simple transformation of XCDL-based CP-Nets from data objects to datasets is executed. An SD-f and a composition diagram are respectively transformed into datasets based on an SD-function schema and a composition schema as shown in Figure 22. The *SD-function* schema is a projection of the structure of an *SD-f data type*. And the composition schema is a unified projection of any composition (serial, parallel and concurrent) in terms of CP-Nets.

Both schemas are a transformation/representation of the data objects into conceptual schemas where the data types defined in Figure 18 and Figure 20 such as Places, Transitions and Types are represented as entities with relations between them retaining the association/aggregation/composition relations defined in the data types.

In the composition schema, the sequence data objects map SD-function output places directly to SD-function input places.

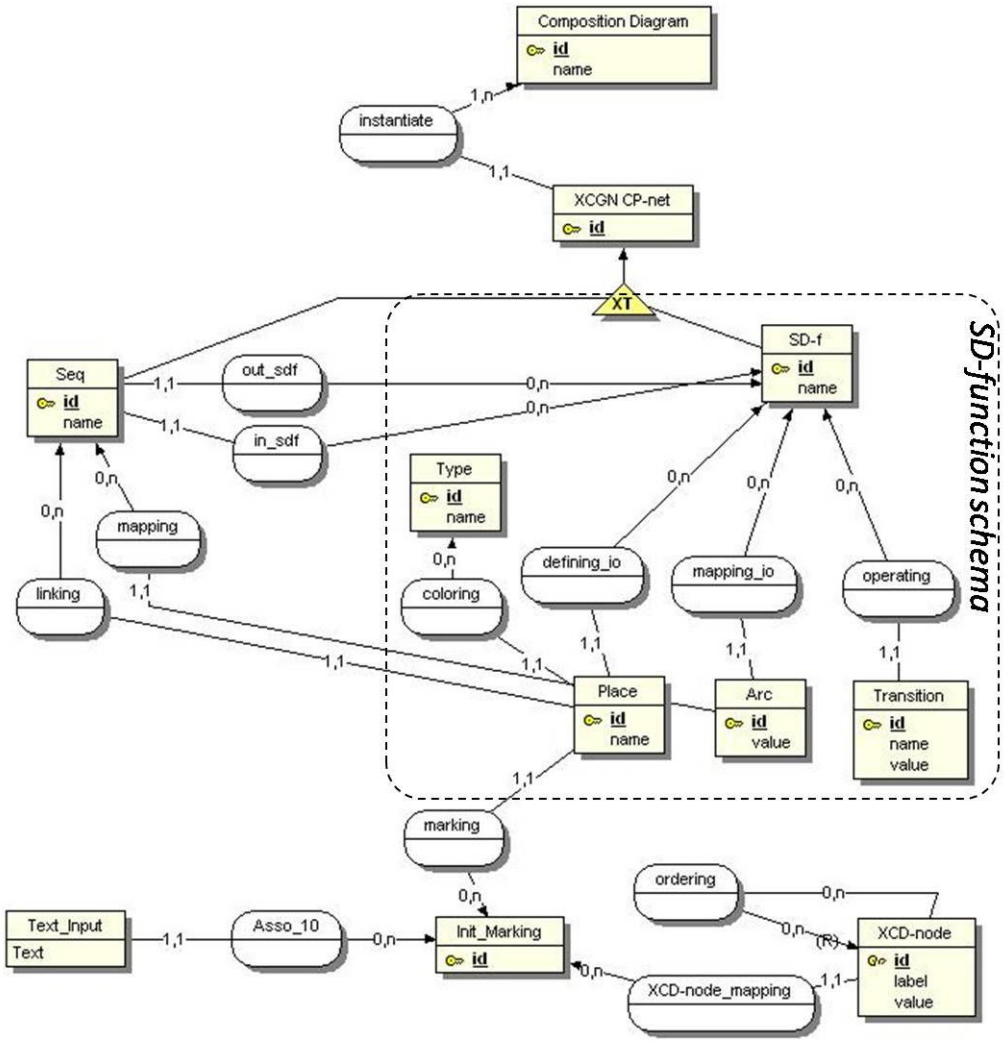


Figure 22: Composition schema compliant with XCGN

So far in our research, we applied one optimization technique, the removal of any passive and redundant CP-Nets which are found in the Intermediate CP-Net. This optimization was elaborated from a natural human interpretation. If we consider the composition in Figure 21, we can obviously notice that the sequence operator mapping the *SD-functions* is passive, and is semantically just a linking chain between an *SD-function's* output and an input. From a mere semantic point of view, this composition can be seen as equivalent to the composition in Figure 23 which shows the *SD-functions* directly linked together without any operators.

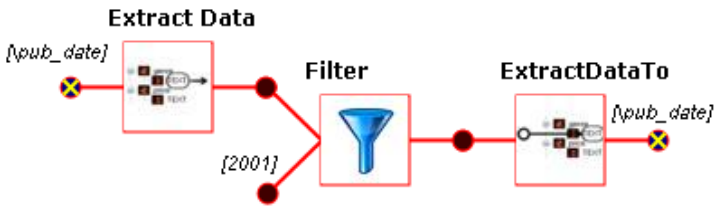


Figure 23: Optimized composition

From a syntactic point of view, a sequence operator technically duplicates the output and input places respectively of 2 separate instances of *SD-functions* and transmits the marking of the output place to the input place as defined in Definition 4.17. Thus, the CP-Net Optimizer, in the Middle-End module, runs through the dataset searching for any redundancies. Once a redundancy occurs, the CP-Net Optimizer maps the input and output directly and deletes their duplicates along with their related arcs and transitions. The resulting CP-Net is then forwarded to the Back-End as an Optimized CP-Net in form of a dataset to be translated to an XML-based CP-Net (XML CP-Net) that can be processed and executed in the XA2C runtime environment.

4.4.3 Back-End

The Back-End is the lowest level of our compiler. Its main purpose is to transform the Optimized CP-Net into an XML CP-Net through an XML-based interchange format for CP-Nets [12]. Our choice for transforming the syntax into XML-based CP-Nets was motivated by the following:

(a) XML is the major standard used nowadays for communicating data
(b) XML-based data allows the framework to be flexible and portable, since XML does not require any prerequisites and can be used on any platform
(c) The XML-based interchange format approach for petri nets was adopted as a standard [55] for petri net portability between different tools and platforms
(d) XML-based machine code allows us to retain conformity in our framework. The framework is intended for use with XML-based data and is itself XML-based.

As of February 2011, PNML [12] (Petri Net Markup Language), an XML-based interchange format for petri nets, was published in the ISO catalogue as Part 2 of the ISO/IEC 15909 standard. Thus, XML-based petri nets are made the standard for petri nets communication and portability between different systems and tools, in particular the petri nets following the model defined in PNML. In our case, we defined our data types (both *SD-f data type* and *Composition data type*) conform to the data model define in PNML and adapted to our XCGN syntax as discussed earlier. In terms of PNML, *SD-f* typed objects are equivalent to PNML modules which are petri nets that

can be instantiated in other petri nets. As for composition diagram typed objects, they are equivalent to PNML petri net files which represent full perti nets with instantiated modules.

Based on XCGN (cf. Definition 4.13) and the relational schemas from the Middle-End component, we elaborated 2 XML grammars, an SD-f and a Composition diagram grammar. Summarized grammars are given here below, for the detailed grammars, refer to Annex A and B.

SD-function XML Grammar:

```
<xs:schema id="sd_function">
  <xs:element name="sd_function">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="SD_f">
        <xs:element name="SD_f_type">
        <xs:element name="SD_f_color">
        <xs:element name="SD_f_transition">
        <xs:element name="SD_f_place">
      </xs:choice>
    </xs:complexType>

    <xs:unique name="SD_f" >
    <xs:unique name="SD_f_type">
    <xs:unique name="Color" >
    <xs:unique name="transition" >
    <xs:unique name="place">
    <xs:keyref name="SD_f_place" refer="SD_f">
    <xs:keyref name="type_place" refer="Color" >
    <xs:keyref name="SD_f_transition" refer="SD_f" >
    <xs:keyref name="XCType_SD_f" refer="SD_f_type" >
  </xs:element>
</xs:schema>
```

Composition XML Grammar:

```
<xs:schema id="Composition" >
  <xs:element name="Composition">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">

        <xs:element name="SD_f">
        <xs:element name="SD_f_type">
```

```
        <xs:element name="SD_f_color">
        <xs:element name="SD_f_transition">
        <xs:element name="SD_f_place">

        <xs:element name="Composition">
        <xs:element name="Composition_type">
        <xs:element name="Composition_SD_f">
        <xs:element name="Composition_SD_f_places">
      </xs:choice>
    </xs:complexType>

    <xs:unique name="SD_f">
    <xs:unique name="SD_f_type">
    <xs:unique name="SD_f_color">
    <xs:unique name="SD_f_transition">
    <xs:unique name="SD_f_place">
    <xs:unique name="Composition_id">
    <xs:unique name="Composition_type_id">
    <xs:unique name="Composition_SD_f">
    <xs:keyref
            name="FK_SD_f_place_Composition_SD_f_places"
            refer="SD_f_place">
    <xs:keyref
            name="FK_Composition_SD_f_Composition_SD_f_pl
            aces" refer="Composition_SD_f">
    <xs:keyref name="FK_SD_f_Composition_SD_f"
refer="SD_f">
    <xs:keyref name="FK_Composition_Composition_SD_f"
            refer="Composition_id">
    <xs:keyref name="FK_Composition_type_Composition"
            refer="Composition_type_id">
    <xs:keyref name="FK_SD_f_SD_f_place" refer="SD_f">
    <xs:keyref name="FK_type_place" refer="SD_f_color">
    <xs:keyref name="FK_SD_f_SD_f_transition"
refer="SD_f">
    <xs:keyref name="CompositionType_SD_fct"
refer="SD_f_type">
  </xs:element>
</xs:schema>
```

Whether a new function is being defined or a manipulation operation is being composed, the resulting dataset representing an optimized CP-Net will be translated by the XML-CPNet generator into an XML document that is validated by its corresponding grammar. Once the XML-CPNet is generated and validated, it can be transmitted to the XA2C Runtime Environment to be executed.

4.5 XA2C Runtime Environment

The XA2C Runtime Environment allows users to execute their compositions separately from the XCDL platform. It receives an XCGN-based CP-Net (represented in an XML interchange format, an XML CPNet). The Runtime Environment executes a composition based on the petri net firing rules.

An execution of a petri net is normaly done by firing a sequence of transitions. Each transition needs to be enabled and ready to fire before it can actually fire. Therefore, an enabling configuration, *Enabled*, is defined over a transition *t* defining when a transition is enabled and ready to fire. When a transition fires, it is identified through a flag *Isfired* and one token is removed from each input place while another one is added to each output place. The value of a token is retrievable through a marking *M* defined over a place. In the case of XCGN-based CP-Nets, some constraints are specified as shown in Definition 4.13. A place in XCGN can withhold data of a single type. Thus its token capacity is limited to one and an arc always has a weight of one. Whenever all enabled transitions fire and the markings change, an execution step *ES* has terminated. Since XCDL is defined as a dataflow language, then executions are done in cycles, where each cycle starts when data is available on the source nodes and terminates when data (XML-based in our case) is retrieved by the sink nodes (destination nodes). We define a full cycle execution as a *Run* specifying a sequence of execution steps allowing the final marking to be reached starting from the initial one. Formaly we define, *Enabled*, *Isfired*, *M*, *ES* and *Run* as follows.

Definition 4.27-*Enabled (t)*, *is the firing rules for a transition t to fire and is defined as:*

$$\forall \boldsymbol{a} \in \{\boldsymbol{P} \times \boldsymbol{T}\}, \qquad \boldsymbol{Enabled}\,(\boldsymbol{t}) = \boldsymbol{true\ if}\ \forall \boldsymbol{p} \in \boldsymbol{a}.\boldsymbol{P}, \qquad \boldsymbol{w}(\boldsymbol{p}) \geq \boldsymbol{w}(\boldsymbol{a}) \geq \mathbf{1}$$

- *A transition "t" is enabled if each input place "p" of "t" is marked with at least "w(a)", where "w(a)=w(p,t)=1" is the weight of the arc from "p" to "t".*
- *An enabled transition t may or may not fire (depending on the level of granularity defined in the medium-grained approach)*
- *A firing of an enabled transition t removes 1 token from each input place p of t and adds a single token to each output place p of t*

Definition 4.28-*Isfired (t)*, *is a flag set over t where t ∈ T and defined as:*

Isfired(t) = true

- *t has fired if a firing configuration is satisfied over t*
  - $\forall a \in P \times \{t\}$, *a single token is removed from each* $p \in P$
  - $\forall a \in \{t\} \times P$, *a single token is added to each* $p \in P$

Definition 4.29-*M(p)* *is a marking over p where* $p \in P$ *and M(p) is the value of the token in p where:*

- M_0 *denotes the set of initial markings of P and I(p) the initial marking of p where* $M_0(p) = I(p)$
- M_{n+1} *denotes the set of final markings of P where:*
  - $\forall t \in T, Isfired(t) = true$
- *We denote by* M_i *the markings of P after i iterations ES have completed*

Definition 4.30-*ES* *is an execution step transforming a marking* M_i *to* M_{i+1}*. It is defined as:*

$$ES: M_i \xrightarrow{ES_i} M_{i+1}$$

- *An execution step ES occurs when all enabled transitions have fired:*
  - $\forall t \in T,\ for\ all\ Enabled(t), Isfired(t) = true$
- M_i *denotes the set markings of P after i execution steps*
- M_{i+1} *denotes the set of markings reached after* ES_i *executions*
- ES_i *is the execution step occurring after i execution steps*

Definition 4.31-*Run* *is a full execution cycle over a composition starting from an initial marking* M_0 *and reaching a final marking* M_{n+1}*. It is defined as:*

$$Run: M_0 \xrightarrow{Run} M_{n+1} = M_i \xrightarrow{ES_0} M_1 \dots M_i \xrightarrow{ES_i} M_{i+1} \dots M_n \xrightarrow{ES_n} M_{n+1}$$

- M_0 *denotes the set of initial markings of P*
- M_i *denotes the set markings of P after i execution steps*
- M_{i+1} *denotes the set of markings reached after* ES_i *executes*
- ES_i *is the execution step occurring after i execution steps*
- M_{n+1} *denotes the set of final markings of P*
- *A Run instance is terminated if:*
  - $\forall i \in [0,n]$, ES_i *is executed and* M_{n+1} *is reached*

The execution of an XCDL program is accomplished via a *Run* instance which will execute sequentially all available steps from 0 to n. As stated in the previous section, an XCGN-based CP-Net can result in either a sequential or concurrent (and parallel) compositions. In the case of a sequence composition, each *ES* will have one and only

one transition to fire. As for the concurrent or parallel compositions, each *ES* can have 1 or more transitions which need to fire simultaneously (cf. Figure 24).The Process Sequence Generator is used to generate 2 execution sequences, serial and concurrent sequences, which specify the order in which the composed functions can be executed by discovering all the *ES* ranging from 0 to n along with their corresponding enabled transitions.

4.5.1 Process Sequence Generator

The Concurrent Sequence specifies different execution steps (*ES*) which must be executed in the correct order from ES_0 to ES_n where *n* is the last *ES*. Each *ES* contains 1 or several functions which can be executed in a concurrent manner (parallel or serial).

The Serial Sequence defines the execution of the functions in a serial manner where each of the functions in the composition has a unique order in which it can be executed ranging from 0 to m-1, m being the number of functions used in the composition.

In order to generate both sequences, we provide an algorithm based on the Incidence Matrix [79] of CP-Nets (cf. Definition 4.2).

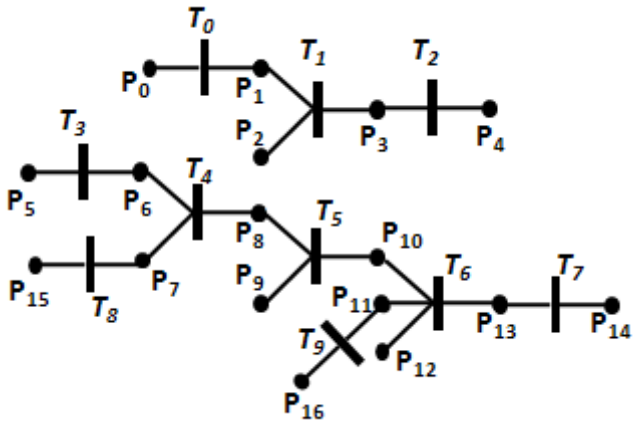


Figure 24: CPN_1, an example of a petri net resulting from scenario 1 in XCDL

Before we give the algorithm, we present the hypothesis defining the background on which the algorithm is based upon.

4.5.1.1 Hypothesis

Based on the XCDL syntax, defined in the XA2C platform, the resulting composition is defined as a CP-Net based on the XGCN and respects the following main properties:

- Each place can contain one and only one token
- A token can be added either through an initial marking provided by the user or an XCD-tree node, or through a fired transition
- All arcs are weighted with the value 1
- A transition is enabled once each of its input places contains at least one token
- A fired transition clears its input places of all tokens and generates one and only one token in each of its output places

Based on these properties, we define our algorithm for simultaneously discovering and generating a serial and concurrent function processing sequence. The processing sequence is stored in a 2 dimensional matrix (called PP for Parallel Processing) where each line represents an *ES* and each column represents a transition (an instance of *SD-function*). Consider the composition CPN_1 in Figure 24, Table 7 represents its PP matrix. The PP matrix shows that we have 5 *ES*s that must be executed sequentially and in order from ES_0 to ES_4 (e.g., T1 and T4 are enabled once T0, T3, T8 and T9 have fired). All transitions in an *ES* can be executed simultaneously in parallel. As shown in Table 7, each transition corresponding to an *ES* is assigned a number. This number represents the sequence order in which a transition should fire in serial processing mode (e.g., in Table 7, T0, T3, T8, T9, T1, T4, T2, T5, T6 and T7 will be executed sequentially in Serial Processing mode).

Table 7: PP matrix of CPN_1

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
ES_0	0			1					2	3
ES_1		4			5					
ES_2			6			7				
ES_3							8			
ES_4								9		

We present next the skeleton of the algorithm followed by the algorithm generating the PP matrix.

4.5.1.2 Algorithm skeleton

The pseudo-code of our algorithm is given in Figure 25. it contains 2 loop steps:

- *Step 1 (line 1-18):*
  - For each place in A, check if the initial value is of type "XCD node" or "user" (in other terms, check if the place is a source place)
  - If so, then for each transition in A, check if the corresponding place is an input to the transition

- If the place is found to be an input, then clear its value from A.
- Check if the transition is enabled
  - If it is enabled and *PP* does not contain a value in the corresponding transition column, then add the value of *m* in *PP(j,n)* where j is the index of the enabled transition and increment *m* by 1
  - If the transition is enabled and *PP* already contains a value in the corresponding transition column, then report an error in the composition and exit the algorithm.

- *Step2 (line 19-42):*
  - While |PP| < T.num, for each transition in PP on ES_{n-1}, clear all its output places and if these places are inputs to other transitions, clear them as well from A
  - Check if their corresponding transitions are enabled
    - If so, check if they were not already added to PP and add them in the corresponding transition line on ES_n
    - Otherwise, report an error in the composition and exit the algorithm.

The formal algorithm is presented here below.

```
   Inputs:
Integer A[,]       // A is the Incidence matrix
String T[],P[]     // T is the Transitions matrix
            // P is the Places matrix
   Outputs:
Integer PP[,]      // PP is the Parallel Processing matrix

   Variables:
Var PP[,] as Integer(T.num,1)
Var m, n as Integer = 0
// m is the sequence number of the next transition
// n is the current level number of the parallel processing

   Begin:
   // step 1
1. for i = 0 to (P.num – 1)
2.   if (P_type(i) = "in xcd") | (P_type(i) = "user") then
3.     for j = 0 to (T.num - 1)
4.       if A(i,j) = -1 then
```

```
5.        A(i,j) = 0
6.        if T_enabled(i,j) then
7.          if not (PP.contains(get_t(out_p))) then
8.            PP(j,n) = m
9.            m = m+1
10.         else
11.           Error("Composition Error")
12.           Exit
13.         end if
14.       end if
15.     end if
16.   end for
17. end if
18.   end for

   // step 2

19.   while (m < T.num)
20. for i = 0 to (T.num - 1)
21.   if PP(i,n) not Null then
22.     t=T(i)
23.     for each out_p in A.outputs(t)()
24.       out_p = 0
25.       for each in_p in A.inputs(get_t(out_p))()
26.         if in_p = out_p then
27.         in_p = 0
28.       end for
29.       if get_t(out_p).enabled then
30.         if not (PP.contains(get_t(out_p))) then
31.           PP(get_t(out_p),n) = m
32.         else
33.           Error("Composition Error")
34.           Exit
35.         end if
36.       end if
37.     end for
38.   end if
39. end for
40. n = n + 1
41.   end while
   End
```

Figure 25: ES discovery algorithm

4.5.1.3 ES Discovery Algorithm proof

In case of a valid composition, the Process Sequence Generator must ensure that

1. All transitions are present in *PP* and each transition is present once and only once
2. After attending the i^{th} level, if all transitions in level *i* fire then all transitions in level *i+1* are enabled
3. All transitions in level *i* can be executed in parallel.

Therefore, to prove the correctness of our algorithm, we must prove the following 3 lemmas.

Lemma 1. If (∃ PP) then ($t_i \neq t_j$, $\forall i,j \in \mathbb{N}$ and $i \neq j$, $i,j < T.num$)

Proof. Before populating the PP matrix, whether in *loop step 1* or *2*, the algorithm checks each time at line 7 and 30 respectively if the added transition already exists. If so, the execution is interrupted and PP is not generated i.e,:

$$\text{If } \Big((\forall i,j \in N, \quad i,j < T.num \text{ and } i \neq j), \quad \exists(t_i = t_j)\Big) \text{ then } (\nexists PP)$$

Therefore, based on the proof by contradiction we prove Lemma 1, PP can exist if a transition exists once and only once in PP. □

Lemma 2. If (∃PP) Then ($\forall t \in T, t \in PP$)

Proof. Based on Lemma 1, if a transition exists in PP, then it can only exist once. And based on the loop step 2 in our algorithm, the algorithm will generate PP and terminate once T.num transitions are added to PP as shown in line 19. Otherwise the execution terminates with an error report without a generation of PP and consequently:

$$\text{If } (\exists PP) \text{ Then } \Big\{\big(t_i \neq t_j, \quad \forall i,j \in N, \; i,j < T.num \text{ and } i \neq j\big) \text{ And } (|PP| = T.num)\Big\}$$

Therefore, by direct proof, we prove Lemma 2, PP can exist if all transitions in T exist in PP. □

Lemma 3. $\forall i \in N \text{ and } i \leq n, \; \{\forall t_i \in T_i, t_i \; isenabled / \forall t_{i-1} \in T_{i-1}, t_{i-1} \; fired\}$

Proof. We prove this Lemma by mathematical induction.

Basis step: for i=0, loop step 1 clears *A* from all input places with initial markings and adds all transitions to *PP* having inputs with only initial markings (from XCD nodes or users). Since all of the transitions in ES_0 have only input places with initial markings, therefore:

$$\forall t_0 \in T_0, t_0 \; isenabled$$

Inductive step: consider $k<n$, we assume that $\forall t_k \in T_k, t_k\ isenabled / \forall t_{k-1} \in T_{k-1}, t_{k-1}\ fired$.

Since $\forall t_k \in T_k, t_k\ enabled$ therefore all t_k in T_k are ready to fire. Based on loop step 2, once all t_k fires, all of their output places are cleared from A (line 24). Based on the hypothesis, a place can either have one token from an initial marking or from a fired transition. In addition, since all transitions with initial markings have already fired in the basis step and their places were cleared from *A*, the places left in *A* can obtain a token only from fired transitions. Once all t_k fire, the input places of t_{k+1}, which are the output places of t_k, are cleared (line 27) and thus all t_{k+1} are enabled having no input places left in *A*. Thus, we conclude by induction that:

$$\forall t_{k+1} \in T_{k+1}, t_{k+1}\ enabled / \forall t_k \in T_k, t_k\ fired \square$$

Now, that we have presented our algorithm for discovering and generating *ESs* corresponding to the resulting composition, we give a detailed illustration of the algorithm over CPN_1, the petri net generated from scenario 1.

4.5.1.4 Illustration

Consider the CP-Net shown in Figure 26. Table 8 represents its Incidence Matrix.

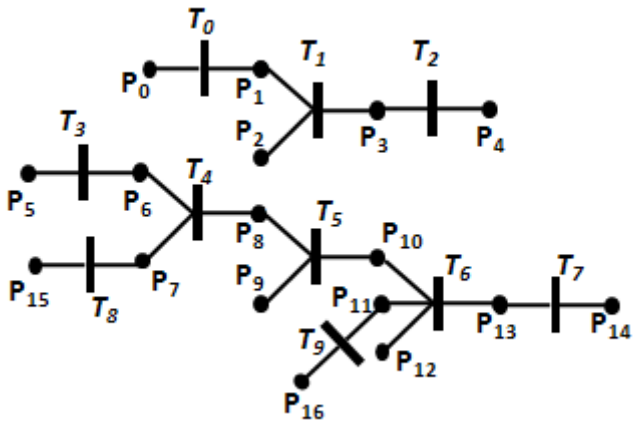


Figure 26: CPN_1, an example of a petri net resulting from scenario 1 in XCDL

Table 8: Incidence Matrix of CPN_1

	P/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
*→	P_0	-1									
	P_1	1	-1								
*→	P_2		-1								
	P_3		1	-1							
→*	P_4			1							
*→	P_5				-1						
	P_6				1	-1					
	P_7					-1				1	
	P_8					1	-1				
*→	P_9						-1				
	P_{10}						1	-1			
	P_{11}							-1			1
*→	P_{12}							-1			
	P_{13}							1	-1		
→*	P_{14}								1		
*→	P_{15}									-1	
*→	P_{16}										-1

The first iteration terminates after executing the first loop step, where the transitions attached to source places "*→" (XCD-nodes or user defined tokens) which must be fired first, are generated and inserted in ES_0 as shown in Table 10.

Table 9: Incidence Matrix after the 1st iteration

	P/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
*→	P_0										
	P_1	1	-1								
*→	P_2										
	P_3		1	-1							
→*	P_4			1							
*→	P_5										
	P_6				1	-1					
	P_7					-1				1	
	P_8					1	-1				
*→	P_9										
	P_{10}						1	-1			
	P_{11}							-1			1
*→	P_{12}										
	P_{13}							1	-1		
→*	P_{14}								1		
*→	P_{15}										
*→	P_{16}										

Table 10: PP matrix after the 1st iteration

ES/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
ES_0	0			1					2	3

The second iteration is executed in the second loop step for an ES_i=ES_1. The execution terminates after i gets incremented by 1.

Table 12 shows the added transitions to be executed in ES_1.

Table 11: Incidence Matrix after the 2nd iteration

	P/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
*→	P_0										
	P_1										
*→	P_2										
	P_3		1	-1							
→*	P_4			1							
*→	P_5										
	P_6										
	P_7										
	P_8					1	-1				
*→	P_9										
	P_{10}						1	-1			
	P_{11}										
*→	P_{12}										
	P_{13}							1	-1		
→*	P_{14}								1		
*→	P_{15}										
*→	P_{16}										

Table 12: PP matrix after the 2nd iteration

ES/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
ES_0	0			1					2	3
ES_1		4			5					

The third iteration is executed to obtain ES_2. The results are shown in Table 13 and **Table** 14.

Table 13: Incidence Matrix after the 3rd iteration

	P/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
*→	P_0										
	P_1										
*→	P_2										
	P_3										
→*	P_4			1							
*→	P_5										
	P_6										
	P_7										
	P_8										
*→	P_9										
	P_{10}						1	-1			
	P_{11}										
*→	P_{12}										
	P_{13}							1	-1		
→*	P_{14}								1		
*→	P_{15}										
*→	P_{16}										

Table 14: PP matrix after the 3rd iteration

ES/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
ES_0	0			1					2	3
ES_1		4			5					
ES_2			6			7				

Table 15 and Table 16 show the results after the 4th iteration.

Table 15: Incidence Matrix after the 4th iteration

	P/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
*→	P_0										
	P_1										
*→	P_2										
	P_3										
→*	P_4										
*→	P_5										
	P_6										
	P_7										
	P_8										
*→	P_9										
	P_{10}										
	P_{11}										
*→	P_{12}										
	P_{13}							1	-1		
→*	P_{14}								1		
*→	P_{15}										
*→	P_{16}										

Table 16: PP matrix after 4th iteration

ES/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
ES_0	0			1					2	3
ES_1		4			5					
ES_2			6			7				
ES_3							8			

The results of the final iteration are shown in Table 17 and Table 18.

Table 17: Incidence Matrix after 5th iteration

	P/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
*→	P_0										
	P_1										
*→	P_2										
	P_3										
→*	P_4										
*→	P_5										
	P_6										
	P_7										
	P_8										
*→	P_9										
	P_{10}										
	P_{11}										
*→	P_{12}										
	P_{13}										
→*	P_{14}								1		
*→	P_{15}										
*→	P_{16}										

Table 18: PP matrix after the 5th iteration

ES/T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
ES_0	0			1					2	3
ES_1		4			5					
ES_2			6			7				
ES_3							8			
ES_4								9		

Finally, the algorithm checks that all the transitions are available once and only once in the PP matrix and ends the execution. Therefore we conclude that in this case, 5 iterations were required for generating the PP matrix. As it is shown in Table 18, the PP matrix contains 5 *ESs* which must be executed from ES_0 to ES_4 sequentially. All

transitions available in the same *ES* can be executed in parallel. As for a serial execution, we can see in the resulting PP matrix that a unique number is associated to each transition which specifies its serial execution order.

4.6 Conclusion

In this chapter, we propose a solution for XML manipulations by non-expert users, called XA2C (XML mAnipulAtion composition), since it was argued that existing approaches/techniques only provide chunks of the solution but not a full-fledged one. XA2C has been defined as a framework for non-experts to create/execute xml-oriented manipulations while taking advantage of related works (i.e., XML-oriented visual languages, mashups, XML manipulation techniques and DFVPLs) as shown in Figure 1.

XA2C is developed as a visual studio for an XML-oriented DFVPL, called XCDL (XML composition definition language). The framework is divided into 3 modules: (i) XCDL platform, (ii) XA2C compiler, and (iii) XA2C Runtime Environment. Each module is formally defined separately based on a CP-Net grammar, called XCGN (XML-oriented composition grammar net). The XCDL platform provides the specifications and syntax of the language which allows mainly users to visually compose XML-oriented manipulation operations. XCDL is designed for non-experts, using functional compositions following the natural human thinking process. The compositions are defined as serial, parallel and concurrent through simple mapping of function outputs to inputs, thus rendering the compositions more intuitive. The result of any composition is an object CP-Net satisfying XCGN. The compiler validated, optimized and translated these CP-Net objects into XML-defined CP-Nets (XML-CPNet), allowing them to be executed in the Runtime Environment. Defining the compiler as a separate module clearly separates the language from its Runtime Environment which lacks in current DFVPL. Before executing/running any XML-CPNet, they have to be analyzed by the Runtime Environment where execution sequences would be generated provided 2 execution modes, concurrent/parallel and serial. Consequently, the approach can take advantage of multi-processor machines and apply parallel executions.

CHAPTER 5

PROTOTYPE AND EXPERIMENTS

The XA2C was defined as a formal framework for creating/composing and executing XML oriented manipulations. The framework defined a visual functional composition language, called XCDL, with a Runtime Environment allowing the execution of composed operation. In this chapter, the X-MAN, a prototype developed for evaluating and validating the XA2X approach, is detailed and evaluated. On one hand, the architecture of the prototype is discussed here with regard to the language platform, compiler and Runtime Environment defined in XA2C. On the other hand, a visual language evaluation framework is defined and used for assessing the XCDL language along related visual tools such as IBM Diama and YahooPipes. User case studies have been conducted and used for evaluating the language which showed the XCDL language to be well more suited for XML manipulations by non-experts compared to other tools.

Table of Contents

5.1 Introduction

To validate the XA2C approach, we implemented and evaluated a prototype called Visual X-Man, based on the XCDL grammar, allowing us to draw and execute XML-oriented manipulation operations identified as functions defined in the system libraries. The functions defined in the prototype were mainly functions stored in DLL files. The prototype was developed in VB.Net.

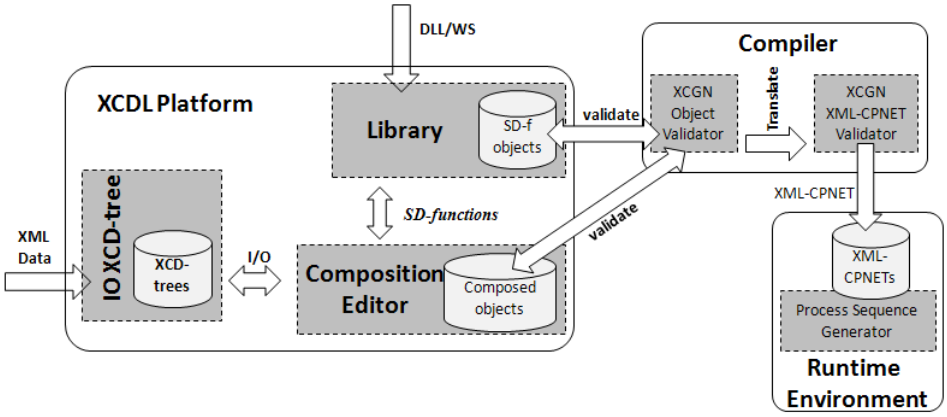


Figure 1: Prototype architecture

The architecture of the prototype, shown in Figure 1, is based on the framework's main architecture. The prototype is composed of 3 main modules: (i) the XCDL platform, (ii) the compiler, and (iii) the Runtime Environment. The X-Man was evaluated through user case studies which were analyzed in a VPL evaluation framework.

In Section 2, we discuss the language platform. Section 3 presents the compiler module. The Runtime Environment is described in Section 4. In Section 5, the language is evaluated along with process sequence generator. An Section 6 concludes.

5.2 XCDL Platform

The XCDL platform is the visual language editor where the user can define/create his compositions, functions and I/O XCD-trees. As defined in our approach, the language is divided into I/O XCD-trees and compositions. Therefore, on one hand we defined the I/O XCD-tree module responsible for transforming XML documents, fragments, DTDs and schemas into XCD-trees and vice versa[1]. On the other hand, and since the composition is based on instances of SD-functions mapped together, we defined the

[1] XML documents generated from output XCD-trees may not be similar to the input XML document in cases of output XCD-tree edit or creation.

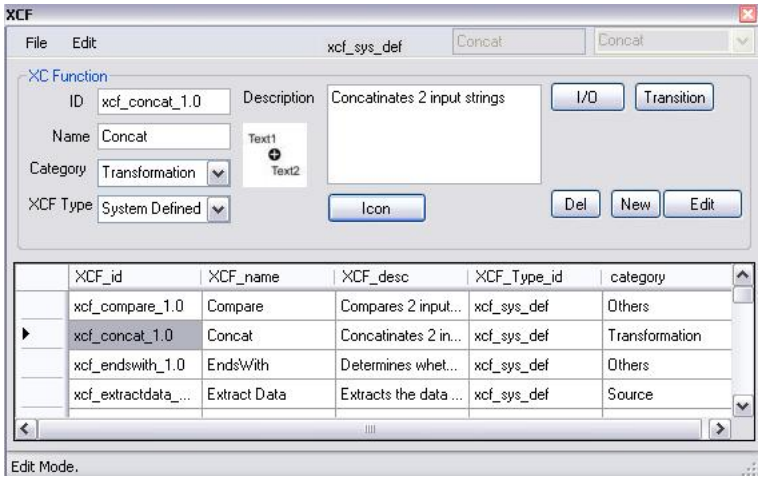
library module which defines and stores all SD-functions that can be instantiated later on in the composition. Finally, we defined a composition editor as a visual editor which instantiates SD-functions visually and maps them together along with I/O XCD-trees. As a result, the XCDL platform is divided into 3 sub-modules: (i) Library, (ii) I/O XCD-tree, and (iii) Composition editor, discussed here below.

5.2.1 Library

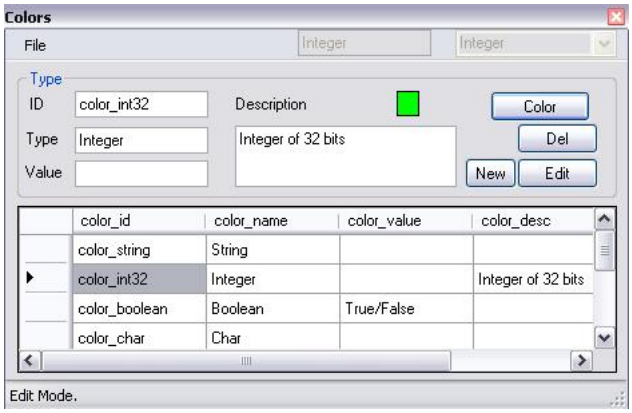
The library sub-module is a set of graphical forms which allow users to customize the language and define the *SD-functions* to embed in the library. The customizations are done by choosing the data types to include in the language, colors to give to each type, dimensions to give to a transition (h_t and w_t), maximum height for the places in a function (h), images used to describe the functions, etc. The functions are identified via a set of forms allowing users to initiate a function definition, define its transition containing the operation to be executed, and define its I/O places. Figure 2 presents some of these forms.

An *SD-function* in our prototype is defined in compliance to the *SD-function* definition. First, an *id* and a *name* are defined, as shown in Figure 2.a, along with other attributes (i.e., *icon, function category and type*). Second, the *SD-function* components are defined such as colors, places and transitions. Arcs are implicitly defined once the transition and places are defined. The places types/colors are defined using the form in Figure 2.b. Colors are mainly identified by an *id, type* and an *RGB color*. The form in Figure 2.c is used to identify the I/O places of a function. A place has an *id*, *name* and *type*. The *type* is chosen from the colors defined in Figure 2.c. Each place is associated with an I/O flag and an initial value. The I/O flag designates whether the place is an output or an input. The initial value is given to input places defining the initial markings of a *SD-function.*

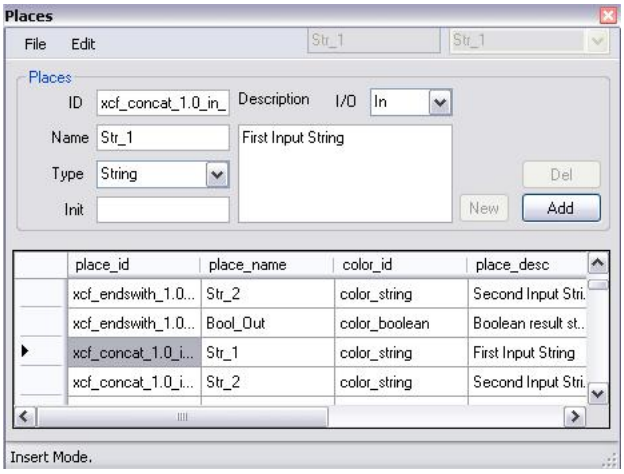
An *SD-function* has one transition defining the operation it accomplishes. In our prototype, the transition is defined through the form in Figure 2.d. A transition has an *id, name, type* and *value*. The type identifies the data type of the operation's output. As for the *value*, it denotes the function operation to be executed, which is retrievable either from a DLL or a web service.



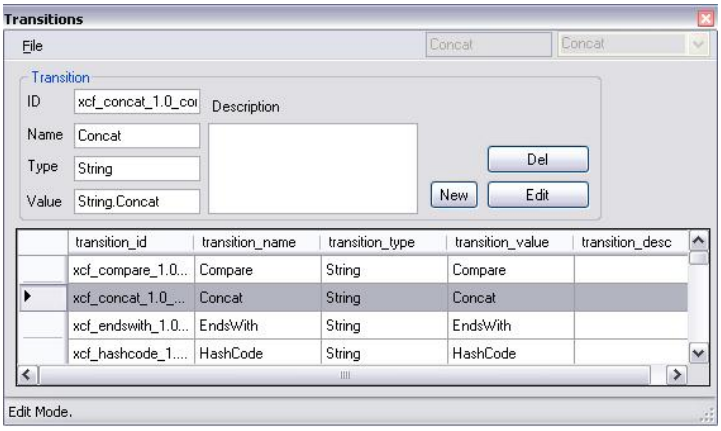
(a) *SD-functions* definition form



(b) XCGN colors configuration form



(c) *SD-function* I/O places configuration form



(d) *SD-function* transitions configuration form

Figure 2: Library configuration forms

Once an *SD-function* is defined with all its components, an automated graphical representation is generated (as shown for the Insert function in Figure 3).

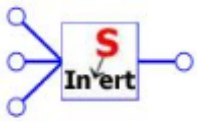


Figure 3: Insert SD-function graphical representation

The graphical representation is drawn with respect to the XCDL transformation syntax. Before a function is finally stored, it goes through the data model module (compiler) where it is validated, translated into an XML-CPNET, and then saved in the XCDL platform library.

5.2.2 I/O XCD-trees

The I/O XCD-tree module is responsible for defining the XML I/O to be manipulated. On one hand, we developed an algorithm which generates a summarized structure of an XML document with repetition reduction. It takes an XML file as an input and generates a tree list as an output representing the summarized structure of the document. The tree list is then drawn as a tree view as shown in Figure 4.

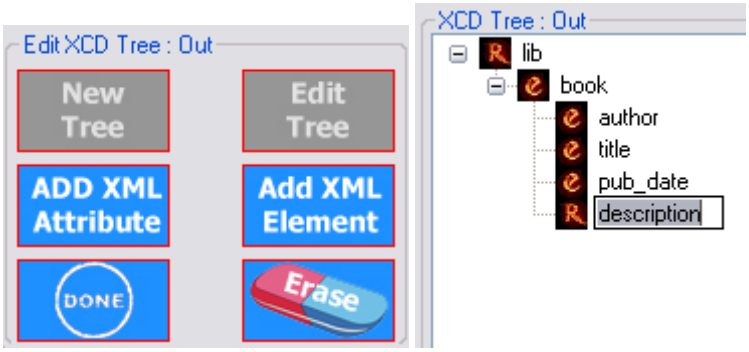


Figure 4: Edit XCD-tree controllers

On the other hand, we developed some controllers, presented in Figure 4, allowing users to visually create their own XCD-trees as well as to edit existing XCD-trees. These XCD-trees are automatically translated into XML structured data. These functionalities are only applicable on the output XCD-trees and can be used for restructuring or creating new output structures.

5.2.3 Composition editor

The composition editor, shown in Figure 5, provides the user with a graphical/visual language editor allowing him to compose his operations from the functions defined by

the Library module. This platform is divided to 4 main sections: the *XCD-tree:In* shows the input XML structure (on the right), the *XCD-tree:Out* shows the output structure (on the left), the *SD-functions* list the functions defined by the Library (on top), and the Composition Workspace (in the middle) where the composition is drawn by dragging and dropping *SD-functions* and sequentially mapping them by consecutively selecting an output and then an input.

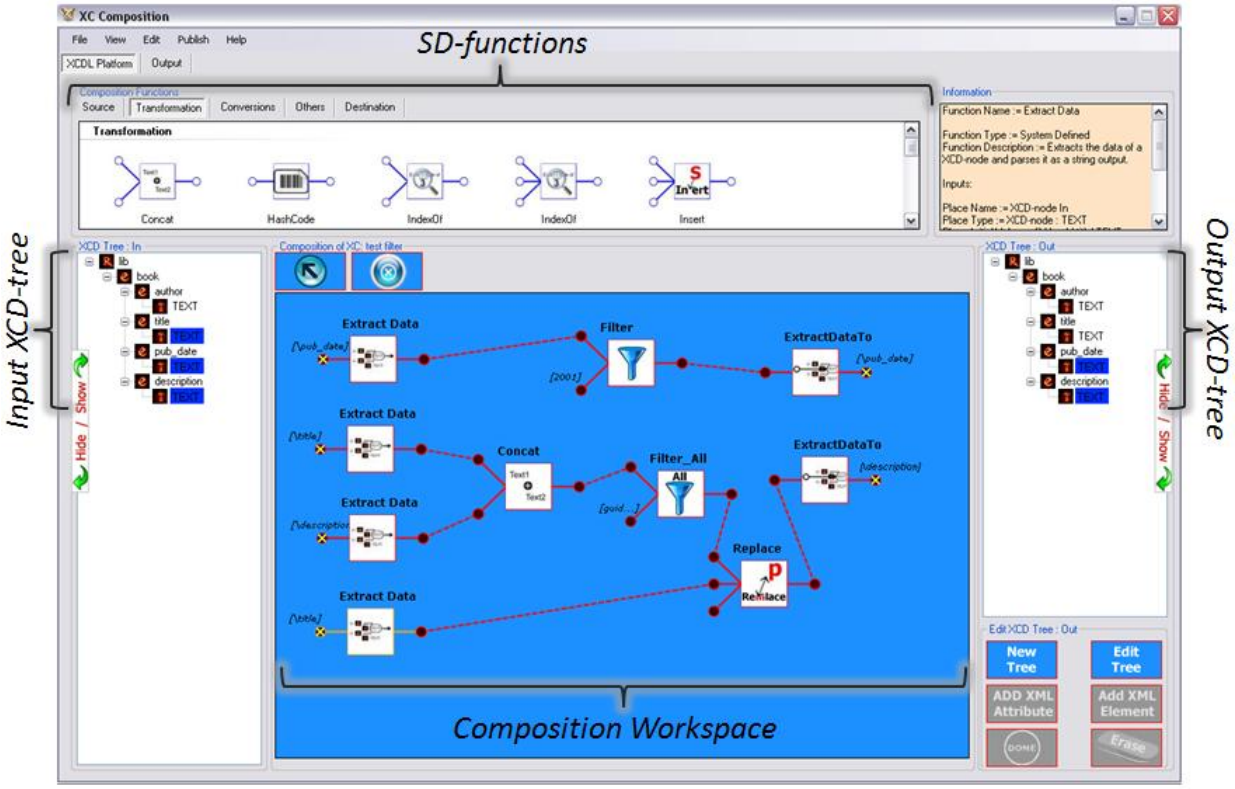


Figure 5 Composition editor

To give an illustration of the use of XCDL, we go back to scenario 1 (cf. chapter 1, Section 1.2.1). A journalist wants to filter out all the books from the company's library based on specific topics (e.g., xml related, guide books) and publication year. Figure 5 shows the composed operation defined in XCDL.

To define the composition operation of this filter, first we define the input and output content description structures as I/O XCD-trees. The input and output XCD-trees generated, represent both the structure of the companies' library (books.xml). Different *SD-functions* are shown in the Composition Function's section. Those functions have already been defined in the XCDL Library sub-module.

In this scenario, to create his composition, the user first checks available *SD-functions*. He starts by selecting the textual content values of the *Elements*, *pub_date*, *title* and *description* as shown above by using the *Extract Data* function which extracts the

values from the selected XCD-nodes. Then, he maps them to the required manipulation functions and redirects them to the output structure.

In this scenario, the user needs to create 2 filters: one for filtering the books based on their publishing date, and the other for filtering them based on the topics required.

As shown in the composition workspace, 2 parallel filters have been created. The first one was defined by extracting the publishing date from the input XCD-tree then applying a simple filter that was given the argument "2001", the year to be selected, and the results are redirected to the output XCD-tree. The second filter was composed parallel to the first one. In this filter, both the book's *title* and *description* are searched for any relatedness to the topics at hand (guide books and XML). To do so, the user extracts the *title* and *description*, merges them together using the *SD-function Concat,* and consecutively sends them to a filter based on multiple keywords, *Filter_All*. The results of this filter are all the books related to the topics at hand with their *title* and *description* merged together. Therefore, the user uses the *SD-function Replace* to remove the title from the results then transmits the description alone to the description field in the output XCD-tree, thus preserving the same structure of the input XML data. Both filters will be executed simultaneously, therefore, only the data corresponding to the criteria of both filters will be transmitted.

5.3 XCDL Compiler

This module plays the role of a compiler, translating a high-level language to a machine readable language. It was developed based on the compiler module defined earlier in our approach (cf. Chapter 2, Section 2.3). Each defined *SD-function* and composition in the XCDL platform is translated from a data object matching respectively the SD-function data object and composition diagram data object (cf. Chapter 3 Section 4.2) to a dataset which is than translated to an XML-based petri net, XML-CPNET (cf. Figure 1).

Figure 6 presents respectively the dataset' schemas for defining *SD-functions* and compositions built in our prototype.

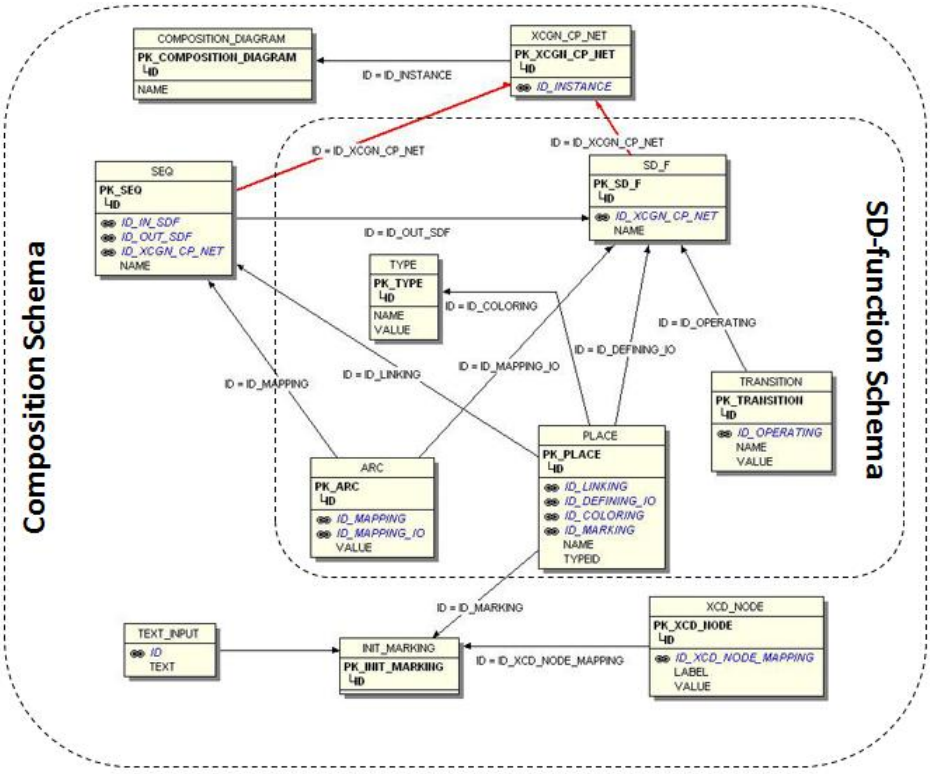


Figure 6: Detailed relational schemas of the internal data models

These schemas yield respectively 2 XML schemas. Once an *SD-function* or a Composition is ready to be stored or transmitted to the Runtime Environment, the corresponding datasets are transformed into XML-CPNETs validated by the corresponding *SD-function* or Composition XML schemas.

5.4 Runtime Environment

The Runtime Environment, implemented as a console application with no graphical interface, executes/runs a composed operation defined as a CP-Net received from the compiler in form of an XML-CPNET. In order to run such petri nets, we implemented the Process Sequence Generator component which analyzes the input petri nets and generates 2 execution sequences, serial and parallel as discussed previously in Chapter 3, Section 5. The execution sequences are generated automatically through the ES (Execution Step) discovery algorithm implemented here (cf. Chapter 3, Section 5.1). Once the execution sequences are generated, they can be executed either by calling the SD-functions sequentially or concurrently through multi-threading respectively for serial and parallel sequences.

5.5 Evaluation and Experiments

To evaluate our approach, 2 studies have been accomplished: (i) evaluating the ES discovery algorithm, and (ii) evaluating the XCDL language. The first study was done in order to evaluate the performance of the ES discovery algorithm. As for the second study, it was done to assess the overall quality of the XCDL language. Both studies are discussed here below.

5.5.1 Evaluating XCDL, an XML-Oriented Visual Language

In the literature, as discussed by Marghescu et al. [76], evaluating a language, editor, or tool consists of 3 steps: (i) defining a model of evaluation, (ii) specifying the proper attributes for measurement, and (iii) identifying the limitations and problems of the language. In our case, we are dealing with a visual language instead of a traditional language. Thus, the evaluation is totally dependent of the visualization techniques adopted by the corresponding VPL (XCDL in this case).

In the visualization literature, very few evaluation frameworks were defined since many researchers [20, 24] emphasize the necessity for systematic empirical evaluation of visualization techniques. Nonetheless, Marghescu et al., in their paper [76], defined a model of evaluation for visual data mining tools based on empirical studies and theories identified in the literature. In their model, they defined 3 levels of analysis: visualization, interaction and information. In our research, in order to evaluate our language, we adapted a similar approach by defining the same analysis levels with attributes specific to XML-oriented visual languages. Our main concern falls on evaluating the quality of use of XML-oriented VPLs in order to review the user satisfaction. Therefore, we take into consideration all the important aspects of XCDL: visualization, interaction with the system and overall language information.

5.5.2 XCDL Evaluation Framework

In XCDL, the success of a visual composition is identified by the user satisfaction that depends on how good the visualizations, easy and simple the interactions, and accurate the language, are:

- A good visualization is properly able to represent the data of interest, the initial settings should be practical and adequate and the visualization system should provide a variety of exploration tasks (i.e., overview and details of data) facilitating the access to the desired information by the user.
- An easy and simple interaction between the user and the system is guaranteed when the language is efficient, accurate and easy to use and learn.
- The language must be accurate and reliable.

Based on these characteristics, we define the 4 criteria *quality of language*, *quality of visualization*, *quality of interaction* and *quality of use*. These criteria were adapted from the evaluation model defined by Marghescu et al. and rendered specific to VPL assessment.

Definition 5.1-*Quality of Language* *is defined as the assessment of the totality of characteristics and features of the language related to user satisfaction. These features are defined in terms of visualizations, interaction and overall use of the language.*

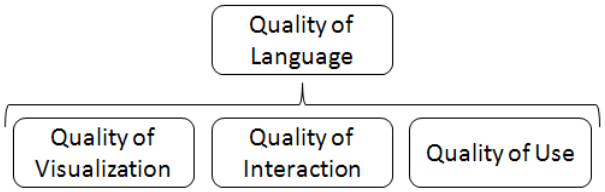


Figure 7: Evaluating the *quality of language*

Definition 5.2-*Quality of Visualization* *is used to evaluate the language in terms of graphical representations and user interface. It is evaluated based on 2 main criterions:*

- ***Initial settings*** *referring to the I/O initial requirements and parameters for visualization.*
- ***Data display*** *assessing the possibility to visualize data structure, description and organization.*

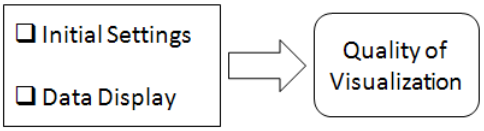


Figure 8: Evaluating the *quality of visualization*

Definition 5.3-*Quality of Interaction* *is used to assess if the language is easy to learn, intuitive and reliable. It is evaluated based on 3 main features:*

- ***Learnability*** *specifies how easy and simple the language is to learn.*
- ***Intuitiveness*** *denotes the degree in which the user feels the language is helpful.*
- ***Reliability*** *reflects the viability of the language to define a task.*

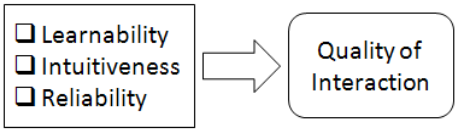


Figure 9: Evaluating the *quality of interaction*

Definition 5.4-*Quality of Use* *is used to elaborate an overall assessment of the language, whether it satisfies its purpose or goals. It is evaluated based on 3 main features:*

- ***Ease of Use*** *denotes the level of simplicity or complexity required by the language to be used.*
- ***Accuracy of Use*** *designates whether the user finds the language specific and oriented towards the data types it presumes to deal with (XML in the case of XCDL).*
- ***Knowledge of Prerequisites*** *assesses the need for users to have knowledge in certain fields.*
- ***Efficiency*** *designates how efficient and quick the language usage is.*

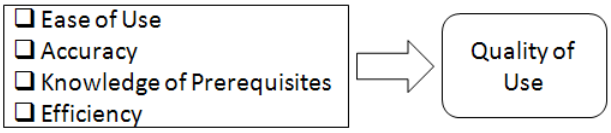


Figure 10: Evaluating the *quality of use*

The overall model of evaluation and the relations between the different attributes and evaluation features are shown in Figure 11. In a case study, when the user is creating his composition, he needs the interaction with the system to be efficient and smooth which is tightly related to the simplicity provided for visualizing and initiating the data. As an overall, the language needs to be easy to use and accurate to the task.

In the evaluation model, the language is evaluated as the user completes a use case scenario and provides input on the attributes corresponding to the overall of the language, interaction and visualization. The inputs are clustered correspondingly to each group and the *quality of visualization*, *interaction* and *use* are elaborated. All of these 3 measurements are then assessed together to provide an overall evaluation, the *quality of language*.

Using this model, allows users to evaluate the language as a whole and to elaborate granular assessments providing information (pros and cons) on different levels, from the language functionality to its graphical representation and to its achievements (e.g., it is XML-oriented, user friendly, easy to learn, etc.).

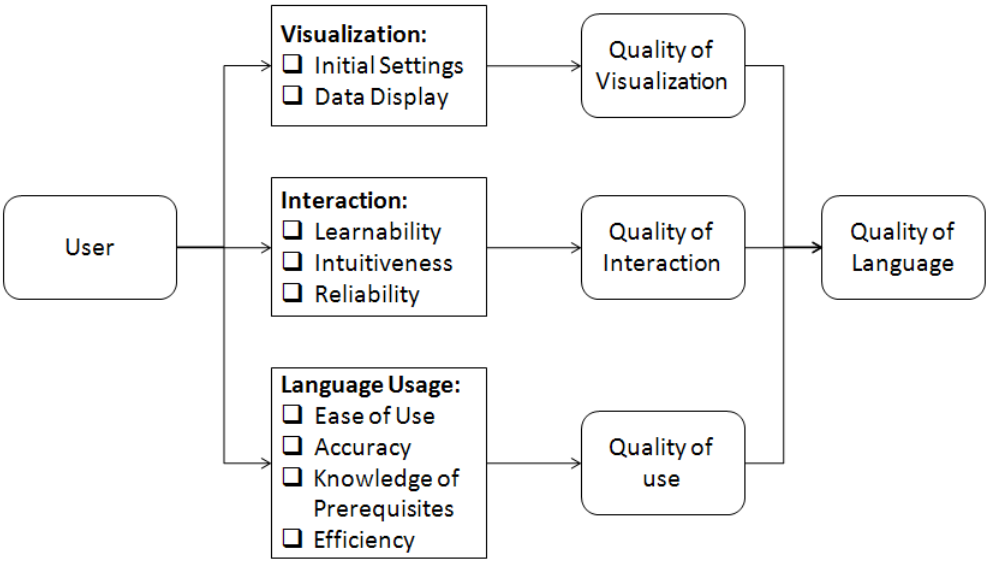


Figure 11: VPL evaluation model

We assessed the XCDL language using the VPL evaluation model.

5.5.3 XCDL Evaluation Case Study

In order to evaluate our language, we provided several use case scenarios that were executed by a number of participants. A questionnaire survey technique was used to collect data. The data collected was then analyzed based on the evaluation model defined in Figure 11.

5.5.3.1 Materials and Participants

Our study was separated to 2 cases. The first one was intended for evaluating the XCDL language from users' perspectives and the second was proposed for comparing XCDL with existing approaches. In our research, YahooPipes and IBM Damia are the closest to our work and therefore were evaluated alongside our prototype X-Man.

The evaluation studies underwent the following phases:

(a) The students took 10 min to get acquainted with each platform (XCDL, IBM Damia and YahooPipes).

(b) The students were given 4 assignments to be solved in X-Man, IBM Damia and YahooPipes.

(c) Once the students have finished the assignments, they were asked to answer a questionnaire containing a set of evaluation attributes.

(a) Participants:

Our study involved 76 participants who were students in an engineering school. The experiments were conducted the participants personal computers. In Table 1, we present the demographic details of the participants.

Table 1: Demographic distribution of the participants

Category	Values	Percentage
Major	Telecommunications	20
	Networking and system	28
	Graphic design	34
	Software engineering	16
University year	4^{th} year	73
	5^{th} year	27
Programming experience	Beginners	54
	Intermediate	46

The participants were given the following 4 cases to accomplish on all of X-Man, YahooPipes and IBM Damia.

(b) Use case scenarios:

1. *Case 1:*

Figure 12: Use case scenario 1

Consider the XML data flow transmitted from "Books.xml". Create a filter allowing only guide books to be forwarded. (The filter needs to be applied on the books' *title*).

2. *Case 2:*

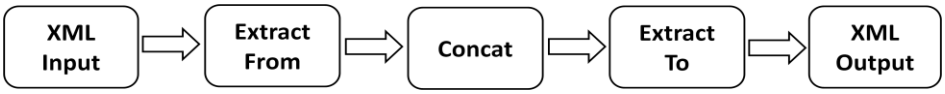


Figure 13: Use case scenario 2

Consider the XML data flow transmitted from "books.xml". Generate for each book a new description with the following template:
"The entitled "Title" was written by Author and published in Pub_Year." where *title, author* and *Pub_year* are sub-elements defined for each book.

3. *Case 3:*

Figure 14: Use case scenario 3

Consider the RSS feed generated from URL1. Bloc all items which are not related to "Rice" neither in their *title* nor *description*. (It is enough that either the title or the description is related to "Rice" for the feed to pass).

4. *Case 4:*

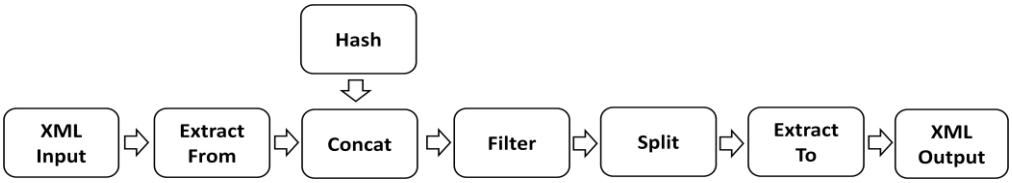


Figure 15: Use case scenario 4

Consider the RSS feed generated from URL1. Bloc all items which are not related to "Rice" neither in their *title* nor *description*. In addition, if the word "Rice" is found in either the feed's title or description, it needs to be obfuscated.
Once the students finished all 4 cases, they were asked to answer the following questionnaire.

(c) Questionnaire of Evaluation:

The questionnaire was divided into 2 parts: (i) common attributes for evaluation in XCDL, IBM Damia and YahooPipes, and (ii) open questions.

i. Common questionnaire:

For each of the following phrases, mark positive if "you totally agree", negative if "you disagree totally" or 0 if "you are neutral".

	(-)---(0)---(+)		
	XCDL	**Yahoo**	**Damia**
1. It is easy to specify/define input data.....	.....	.....	.....
2. It is easy to specify/define output data.....	.....	.....	.....
3. Input data is adequate for XML.....	.....	.....	.....
4. Output data is adequate for XML.....	.....	.....	.....
5. It is easy to understand the required parameters.....	.....	.....	.....
6. It is easy to specify/define the required parameters.....	.....	.....	.....
7. Functions are clearly visualized.....	.....	.....	.....
8. It is easy to understand the functionality of a Function.....	.....	.....	.....
9. The I/O of a function are clearly defined/visualized.....	.....	.....	.....
10. Data types of functions' I/O are clearly defined/visualized.	.....	.....	.....
11. It is easy to differentiate between different data types.....	.....	.....	.....
12. It is easy to visualize/understand XML I/O structures.....	.....	.....	.....
13. It is easy to differentiate between the types of different XML nodes (Element/Attribute/Text).....	.....	.....	.....
14. It is easy to create compositions.....	.....	.....	.....
15. The language is easy and simple to learn.....	.....	.....	.....
16. The language is easy to use.....	.....	.....	.....
17. The Composition does not require lot of steps.....	.....	.....	.....
18. The composition is accurate.....	.....	.....	.....
19. Conversion between different data types is simple.....	.....	.....	.....
20. Many data types can be used (String/Integer/Boolean...)....	.....	.....	.....
21. I/O XML mapping with a composition is simple/easy.....	.....	.....	.....
22. It is easy to modify the output XML structure.....	.....	.....	.....
23. It is easy to create a new output XML structure from scratch.....	.....	.....	.....
24. Programming knowledge is required.....	.....	.....	.....
25. XML knowledge is required.....	.....	.....	.....
Task accomplishment:			
26. Were you able to finish the task?	____	____	____
27. How much time did you take to finish a task?	____	____	____

ii. Open Questions:

1. What are the negative aspects of the XCDL language?

2. What can be done to improve the XCDL language?

The attributes defined in the questionnaire are categorized based on the model defined in section 5.5.2.

5.5.3.2 Evaluation Categories

In order to cope with the VPL evaluation model the attributes were clustered as follows:

(a) Visualization

i. Initial Settings

- Simplicity in defining input data
- Simplicity in defining output data
- Input data adequacy with regard to XML
- Output data adequacy with regard to XML
- Ease of understanding the required parameters
- Simplicity in defining the required parameters

ii. Data Display

- Clarity in visualizing functions
- Ease of understanding a function's operation
- Clarity in visualizing the I/O of a function
- Clarity in visualizing a function's I/O data types
- Ease of differentiating between data types
- Ease of visualizing/understanding XML I/O structures
- Ease of differentiating between the types of different XML nodes (Element/Attribute/Text)

(b) Interaction

i. Learnability

- Composition requiring few steps
- Simplicity between data type conversions
- Simplicity in mapping I/O XML to a composition

i. Intuitiveness

- Ease while modifying the output XML structure
- Ease while creating a new output XML structure

i. Reliability

- Ease while composing a manipulation operation
- Ability to use multiple data types (String/Integer/Boolean...)

(c) Language usage

i. Ease of use

- Simplicity in getting acquainted with (using) the language
- Ease of working with the language

ii. Accuracy

- Accuracy of the composition

iii. Knowledge of Prerequisites

- Low requirement of programming knowledge
- Low requirement of XML knowledge

iv. Efficiency

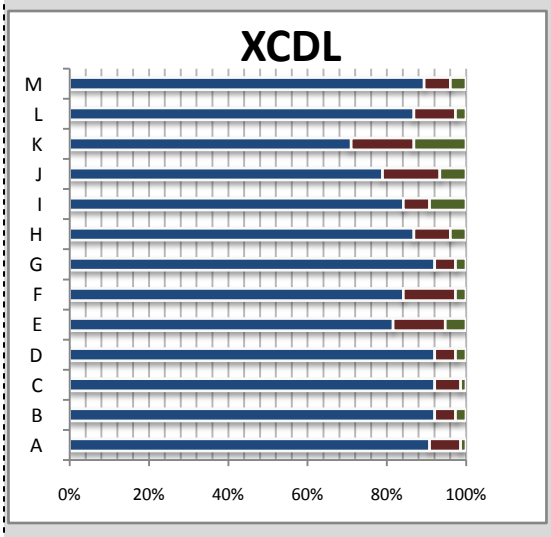
- Ability to accomplish a task
- Time required to accomplish a task

5.5.4 Evaluation Results

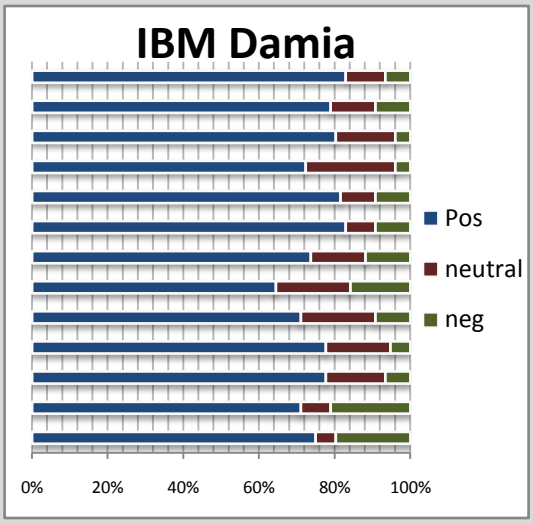
The evaluation process was conducted by collecting information from the participants' answer sheets and clustering them under the evaluation attributes' categories (Visualization, Interaction and Language usage). These categories allow the measurement respectively of the *quality of visualization, interaction and use* which grouped together, result in the assessment of the *quality of language*.

5.5.4.1 Quality of Visualization

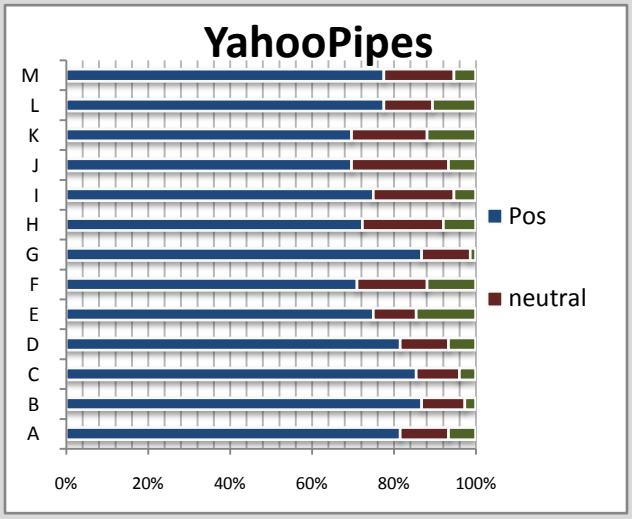
Figure 16.a and b respectively depict users' perspectives concerning the visualization aspects of XCDL, IBM Damia and YahooPipes. As shown in Figure 16.a, no major issues are revealed in terms of visualization. The results clearly show that the overall visualization aspects of XCDL are better than those of IBM Damia and YahooPipes. It is interesting to note that over 90% of the users found that in XCDL, both the inputs and outputs are easily defined and adequate to describe XML data. Also, less than 71% of the users found that differentiating between XML node types is easy and differentiating between *SD-functions'* I/O data types is clear.



(a) XCDL evaluation of visualization



(b) IBM Damia evaluation of visualization

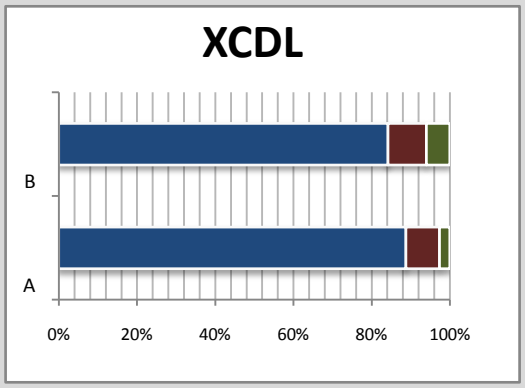


(c) YahooPipes evaluation of visualization

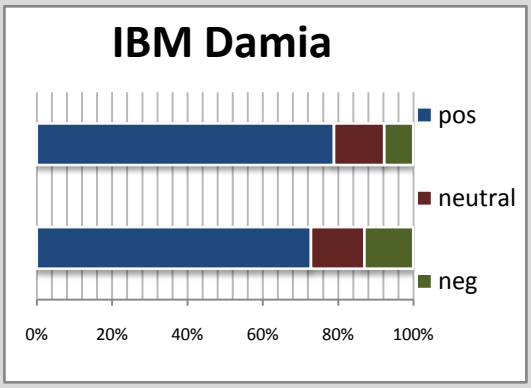
A. *Simplicity of defining input data*
B. *Simplicity of defining output data*
C. *Adequacy of input data with XML*
D. *Adequacy of output data with XML*
E. *Ease of understanding the required parameters*
F. *Simplicity in defining the required parameters*
G. *Clarity in visualizing functions*
H. *Ease of understanding a function's operation*
I. *Clarity in visualizing the I/O of a function*
J. *Clarity in visualizing a function's I/O Data types*
K. *Ease of differentiating between data types*
L. *Ease of visualizing/understanding XML I/O structures*
M. *Ease of differentiating between the types of different XML nodes (Element/Attribute/Text)*

Figure 16: Visualization attributes evaluation

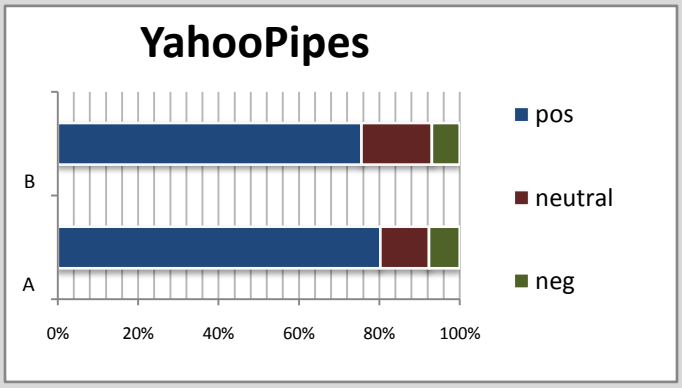
Figure 17 shows the quality of visualization elaborated from evaluating the visualization attributes and confirming that the quality of visualization of XCDL is superior to that of IBM Damia and YahooPipes by a margin over 5%. However, while over 88% of the users gave favorable responses in terms of initial settings for XCDL, less than 85% of the users gave favorable feedback on data display. It is interesting to note that both IBM Damia and YahooPipes received less than 80% of favorable responses to both their initial settings and data display.



(a) Quality of visualization of XCDL



(b) Quality of visualization of IBM Damia



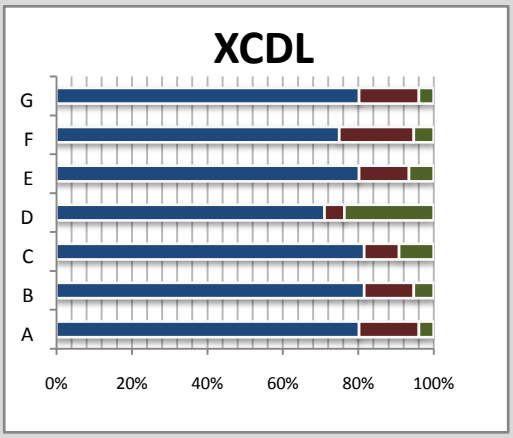
(c) Quality of visualization of YahooPipes

A. Initial settings

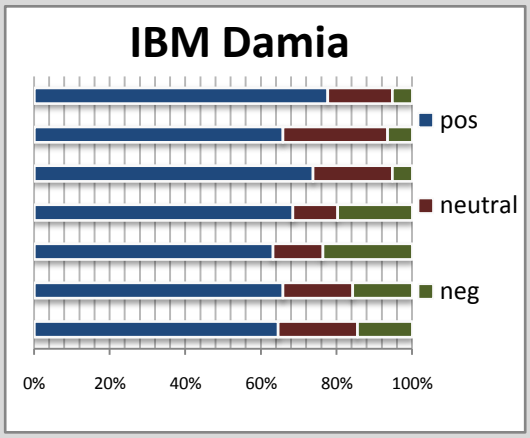
B. Data display

Figure 17: Quality of visualization

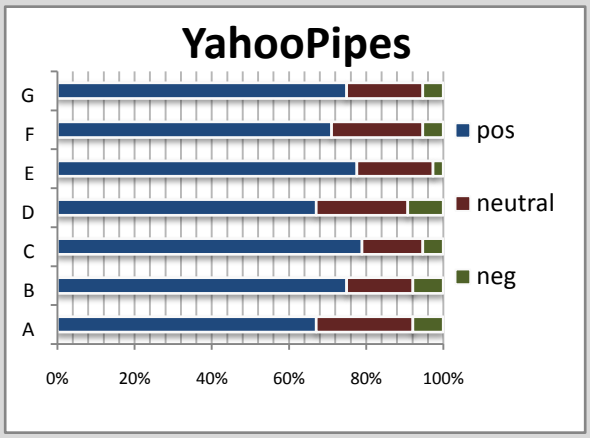
5.5.4.2 *Quality of Interaction*



(a) XCDL evaluation of interaction



(b) IBM Damia evaluation of interaction



(c) YahooPipes evaluation of interaction

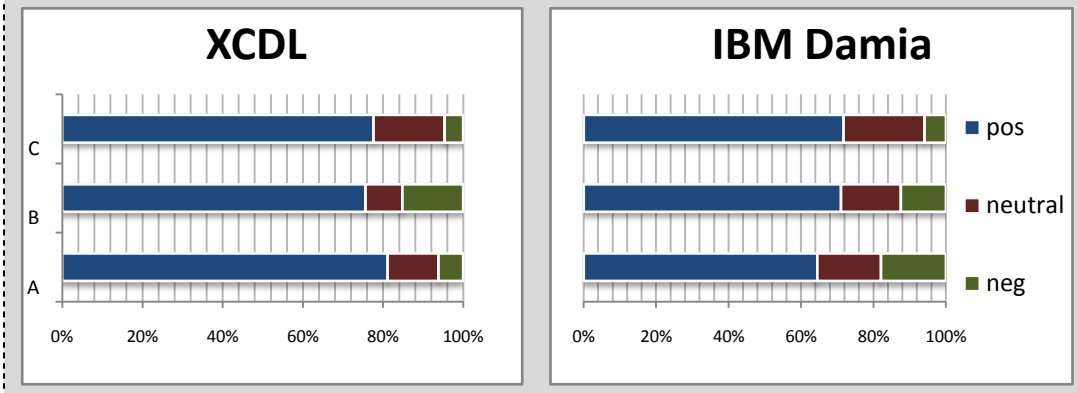
A. Composition requiring few steps
B. Simplicity between data type conversions
C. Simplicity in mapping I/O XML to a composition
D. Ease in modifying the output XML structure
E. Ease in creating a new output XML structure
F. Ease in composing a manipulation operation
G. Ability to use multiple data types (String/Integer/Boolean, etc.)

Figure 18: Interaction attributes evaluation

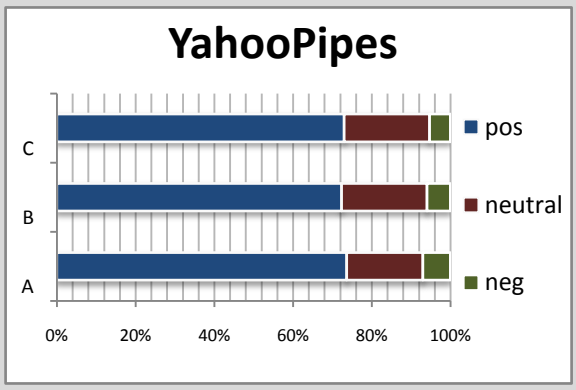
Figure 18.a and b respectively depict users' perspectives concerning the interaction aspects of XCDL, IBM Damia and YahooPipes.

Figure 18.a shows no major issues regarding the aspects related to the user/language interaction. While all the interaction attributes regarding

XCDL received a higher positive percentage than those of YahooPipes, the XCDL "ease in modifying the output XML structure" attribute showed to be less favorable than the rest of the attributes. It is interesting to note, on one hand, that over 80% of the users found it easy to convert data types. On the other hand, less than 72% of the users agreed that modifying an XML output structure is simple which is logical due to the fact that the majority of the participants are in the field of computer science. Therefore, they have little or no knowledge of XML.



(a) Quality of interaction of XCDL (b) Quality of interaction of IBM Damia



(c) Quality of interaction of YahooPipes

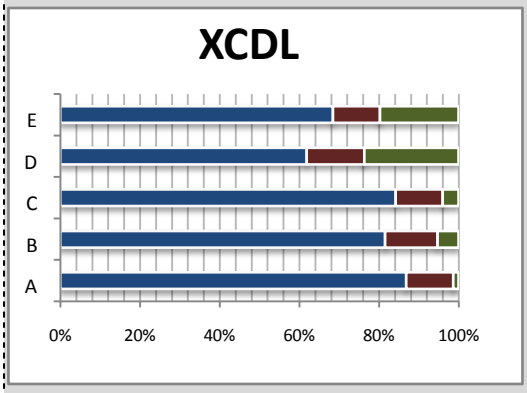
A. Learnabililty
B. Intuitiveness
C. Reliability

Figure 19: Quality of interaction

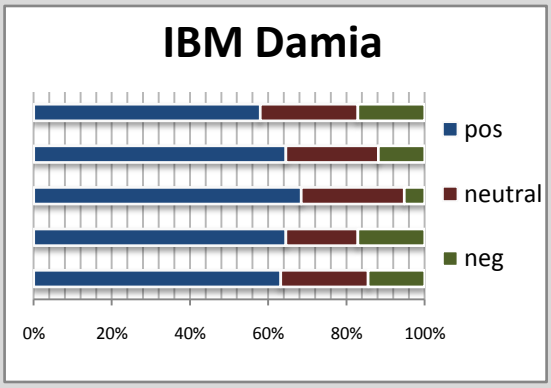
Based on the data collected from evaluating the interaction attributes, as presented in Figure 19, the quality of interaction concerning XCDL is superior then that of IBM Damia and YahooPipes by a margin over 4%. However, while 80% of the users gave favorable responses in terms of learnability regarding XCDL, less than 76% of the

users gave favorable feedback on the user/language interaction being intuitive. That is due to the primitive visual interactive state of the X-Man since it is still in the prototyping phase.

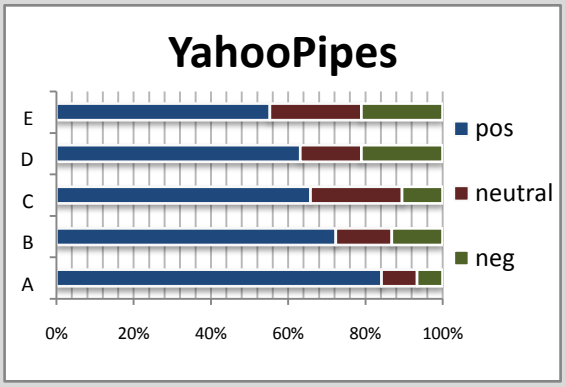
5.5.4.3 Quality of Use



(a) XCDL overall language usage evaluation



(b) IBM Damia overall language usage evaluation



(c) YahooPipes overall language usage evaluation

A. *Simplicity in using the language*
B. *Ease in using the language*
C. *Accuracy of the composition*
D. *Low requirement of programming knowledge*
E. *Low requirement of XML knowledge*

Figure 20: Overall language usage attributes evaluation

Figure 20.a and b respectively depict users' feedback regarding the overall use of XCDL, IBM Damia and YahooPipes.

Figure 20.a shows no major issues regarding the overall aspects of the XCDL language. The positive responses of the users towards the overall attributes of the

XCDL language were higher than those of IBM Damia and YahooPipes. It is interesting to note that 86% of the users found XCDL to be easy to learn. Less than 63% of the users agreed that low programming knowledge are required which is almost similar compared to IBM Damia and YahooPipes.

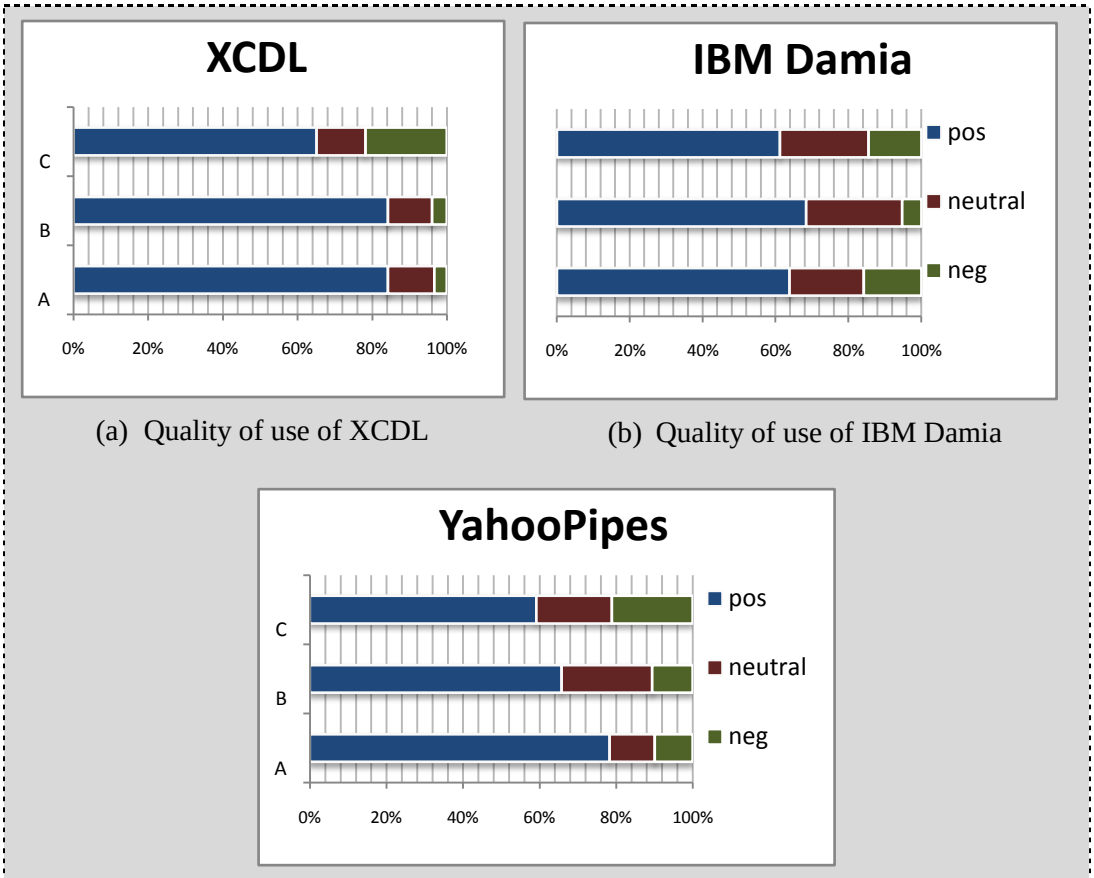
Table 2: Efficiency evaluation of XCDL

Users who accomplished all 4 tasks	66.7%
Time of 1st case	11.9 minutes
Time of 2nd case	8.8 minutes
Time of 3rd case	11.3 minutes
Time of 4th case	10.5 minutes

In terms of efficiency, most participants were unable to finish all 4 tasks on both YahooPipes and IBM Damia due to the fact that:

- YahooPipes requires the input sources to be translated into RSS structures and
- IBM Damia requires the final output to be reconstructed again,

which rendered the tasks difficult to accomplish for non experts. This outcome was anticipated since neither YahooPipes nor IBM Damia are XML-oriented DFVPL but XML-oriented Mashup tools. In the case of XCDL, as shown in Table 2, 66.7% of the users were able to finish all the tasks. It is interesting to note that the time spent to accomplish each case was reduced as the user advanced to the next case, although the cases were becoming more complex.



(a) Quality of use of XCDL

(b) Quality of use of IBM Damia

(c) Quality of use of YahooPipes

A. Ease of use

B. Accuracy

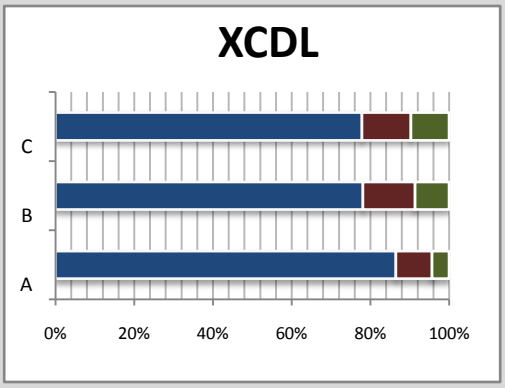
C. Knowledge of Prerequisites

Figure 21: Quality of use

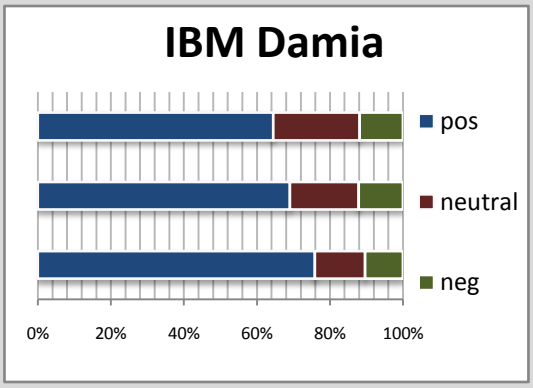
Based on the data collected from evaluating the overall language usage attributes, the quality of use regarding XCDL is proven superior than that of both IBM Damia and YahooPipes as shown in Figure 21. However, while 83% of the users considered XCDL to be easy to use and accurate, less than 66% of the users gave favorable feedback regarding the language requiring little knowledge of prerequisites due to the fact that most users are not computer scientists.

5.5.4.4 Quality of language

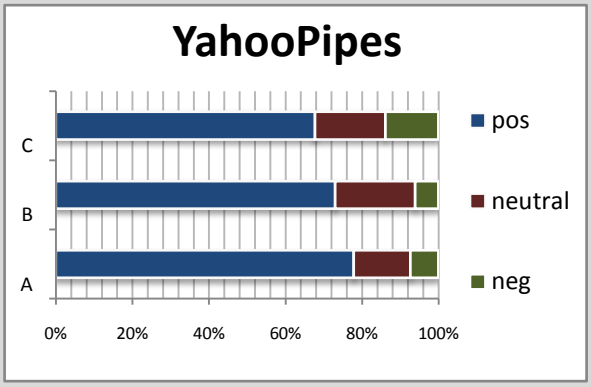
After evaluating the quality of visualization, interaction and use, we elaborate the quality of language concerning XCDL shown in Figure 22.a.



(a) Quality of language of XCDL



(b) Quality of language of IBM Damia



(c) Quality of language of YahooPipes

A. Quality of visualization
B. Quality of interaction
C. Quality of use

Figure 22: Quality of language

As shown in Figure 22, the XCDL quality of use was evaluated to be better than that of IBM Damia and YahooPipes in terms of XML-oriented visual manipulations. While XCDL received over 78% of positive feedbacks regarding all of the quality of visualization, interaction and use, both IBM Damia and YahooPipes received less than 78% of favorable feedbacks regarding all the quality factors. Nevertheless, in XCDL the quality of interaction has the least positive feedback of all which is anticipated due to the lack of error handling and existing bugs in the Visual X-Man prototype developed. It is interesting to note that the Quality of Visualization was assessed positive by over 87% of the participants which is remarkable since both IBM Damia and YahooPipes were less than 79% positive. This has proven for us that visualizing the functions as Petri Nets was more adequate than other means.

After evaluating the quality of language, the feedbacks from the open questions were analyzed as shown in the following section.

5.5.4.5 Open question analysis

Some key features for improving the approach were identified by the participants' answers to the open questions. Some of these features are lacking in the current approach and others need to be improved. They can be summarized in Table 3:

Table 3: Open questions evaluation

Nb of participants	Features to improve	Assessment
28%	The compilation should include error handling allowing users to identify the syntax errors of their compositions	This is due to the fact that X-Man was developed first as an evaluation prototype for XCDL
19%	The language should define control functions such as decisions and loops	The prototype included first some primitive manipulation functions for testing purposes. The second phase is defining control functions
34%	The overall visual interface should be more dynamic	As the language is still is the testing phase, the visual interface is primitive
37%	The Human/Machine interaction should be rendered more dynamic by including drag and drop, and zoom functionalities	So far the drag/drop functionality has not been implemented, since it was not a priority in the theoretical approach
9%	Some functions should be dynamic (e.g., the user should be able to choose the number of inputs to concatenate when using a *Concat SD-function*)	The first step in defining the language was to formally define static functions. The second step is to render these functions dynamic
26%	The language should be enriched with more visual information regarding the *SD-functions*	That is due to the fact that the functions' information were misplaced in the prototype
43%	The language should include some information regarding the main XML elements and structure (i.e., XML Elements, Attributes and Values).	In the prototype no information regarding XML had been provided since the purpose was to evaluate its use by non-experts

Since the Visual X-Man prototype is a beta version released for testing purposes, it did not contain much error handling. It was meant to validate the language's main objective before we could tackle other matters such as error handling. During the testing phase with the participants, some bugs were discovered which affect mostly the quality of interaction. Nevertheless, the evaluation of XCDL in all of its aspect was positive and superior to other tools.

5.5.5 Evaluating the Execution Step Discovery Algorithm

The algorithm was tested with several compositions on an Intel Xeon 2.66GHz with 1Gbyte of Ram memory. We discuss here 4 different cases: serial (cf. Figure 23.a), concurrent (cf. Figure 23.b), parallel (cf. Figure 23.c) and a combined case of serial and concurrent compositions (cf. Figure 23.d).

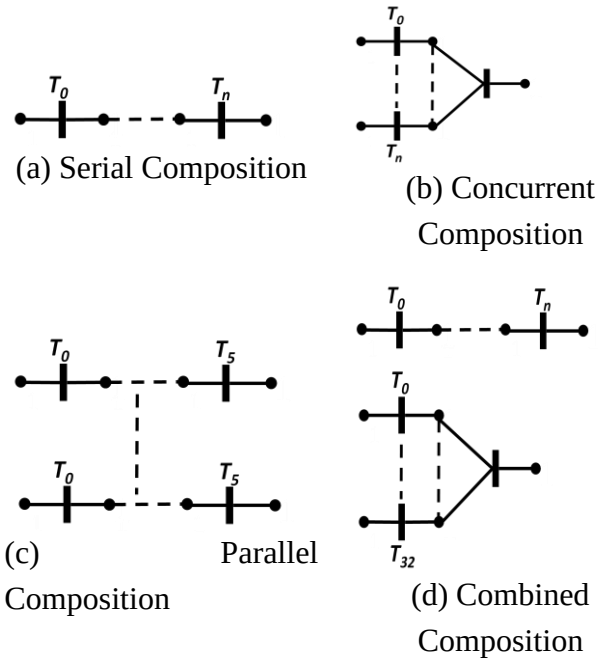


Figure 23: Different composition scenarios

Each of them was defined as shown in the following 4 cases:

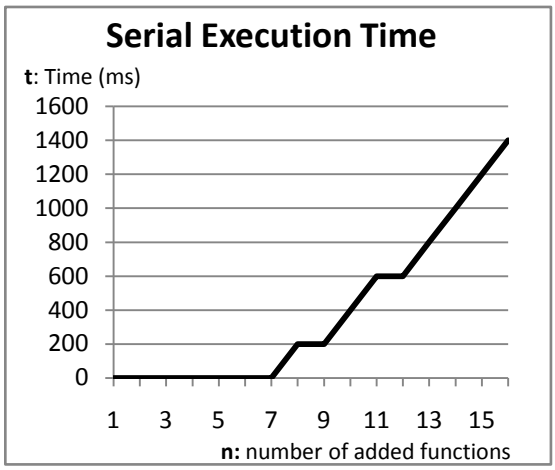
(a) *Case a* represents a serial composition where *n* compositions were tested and each composition had an additional function added to it in a serial composition.

(b) *Case b* denotes a set of concurrent compositions defined *n* times, where each time an additional function was inserted concurrently to the composition.

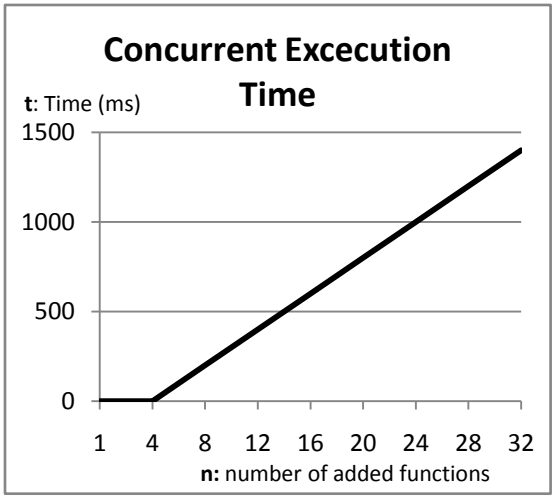
(c) *Case c* specifies a set of parallel compositions between *m* serial compositions of 5 functions each defined *n* times. At each definition, a serial composition is inserted parallel to the previous composition.

(d) *Case d* represents a parallel composition between a serial and a concurrent one where the concurrent composition is composed of 32 functions concurrently mapped and the serial composition is defined *n* times with an additional function sequentially mapped to it each time.

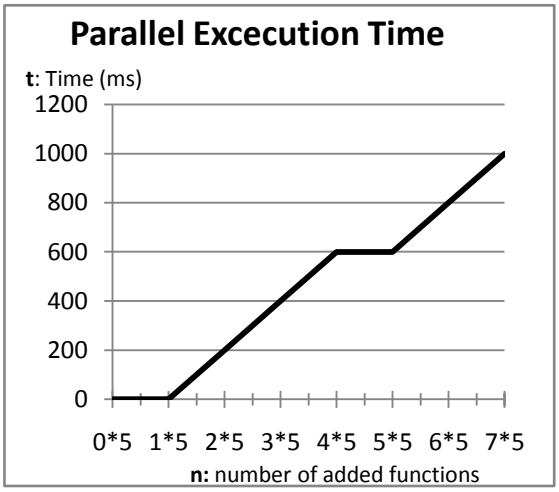
The variable *n* was defined to specify the number of added functions used in each case, ranging from 0 to 15, 32, 7 and 11 respectively in cases a, b, c and d. In all 4 cases the functions were dragged and dropped arbitrarily. Each test was executed 8 times and the results were elaborated from the average of all 8 executions.



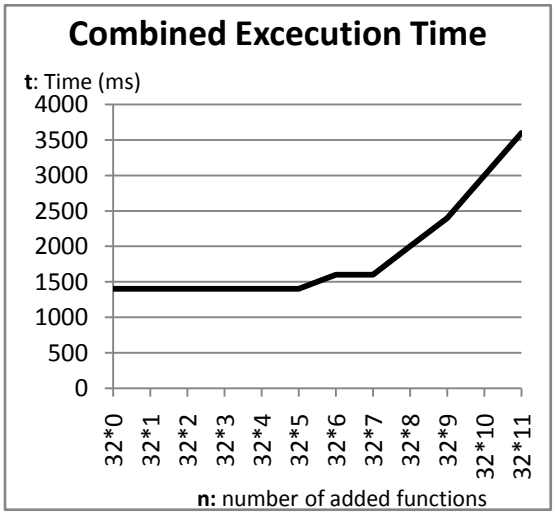
(a) Graph of Case a



(b) Graph of Case b



(c) Graph of Case c



(d) Graph of Case d

Figure 24: Runtime execution of the algorithm

The runtime execution monitored by the tests regarding cases a, b, c and d are shown respectively in the graphs a, b, c and d in Figure 24. As we can see in all 4 graphs, the runtime execution growth remains constant to a certain point then starts growing in

almost a linear form. Therefore, we elaborate the following 4 equations shown in Table 4.

Table 4: Runtime equations of cases a, b, c and d

Cases	Runtime Growth Equation
Case a	t = 200n – 7*(200)
Case b	t = 50n – (200)
Case c	t = 320n – (200)
Case d	t = 300n – 6*(200)

Based on all 4 equations we elaborated that the algorithm has a constant execution period, in the case of the Xeon processor it was equal to 200ms. The execution runtime of concurrent cases is half the execution runtime of serial cases. In combined compositions, we notice that the execution runtime of the algorithm is dependent of the runtime of the maximum independent concurrent composition which sets the minimum runtime of the overall execution. In cases tested here, we could see that 1400ms was the execution runtime for a concurrent composition of 32 functions and almost 0ms was the runtime execution for serial compositions containing less than 7 functions. This was validated in the combined composition, case d, where the minimum runtime execution was 1400ms till the serial composition increased beyond 7 functions. Nonetheless, if we compare the serial execution runtime with the combined execution runtime, we can notice that the algorithm is not yet optimal and optimizations need to be considered in future works. Theoretically, since the runtime of the algorithm in a concurrent composition (*case d)* after 32 functions are inserted equals 1400ms and the runtime in a serial composition (*case a*) containing 11 functions equals 600ms, thus, combining both cases, the concurrent with the serial, should result in a execution runtime equal to 2000ms (1400ms+600ms). Where as in our tests, *case d* showed that for 32 concurrent functions parallel to 11 serial ones, the execution runtime is equal to 3500ms. That difference is due to the fact that the algorithm does not recognize hybrid compositions where several concurrent or serial compositions can co-exist parallel to each other without any dependencies. It always considers them dependent at some level.

5.6 Conclusion

X-man was developed as a prototype for evaluating and validating the XA2C approach, in particular the XCDL language. Its architecture clearly separated the language platform from the Runtime Environment which allowed separate testing of both. 2 use cases studies were achieved:

1. XCDL evaluation
2. ES discovery algorithm validation and evaluation

On one hand, a VPL evaluation framework was defined and used for evaluating XCDL alongside IBM Diama and YahooPipes. The results showed clearly that XCDL was more suitable for XML manipulations by non experts than existing tools and particularly for XML-based dataflows. In terms of quality of language and all of its criteria, visualization, interaction and usage, the assessments were all positive. The tests showed that using:

- simple function visualizations with clear distinction between I/O
- simple mapping in order to compose an operation
- treeviews for XML data representation
- multiple data types
- visual data type conversions,

were essential for an optimized and most efficient use of XML visual languages. XCDL excelled above other tools in it overall quality of language and mainly in its visual and usage aspects.

The evaluation process has shown that an XML-oriented DFVPL is required since the existing tools are not DFVPLs and it had proven that CP-Nets are very useful in terms of visualization as well as execution in such languages.

On the other hand, an evaluation was conducted on the ES discovery algorithm to test its performance. The evaluation showed that the algorithm works properly in all composition cases (serial, parallel, concurrent and combined) and generates correct execution sequences. The algorithm was optimized in the first 3 cases. As for the combined case, optimizations are required since the algorithm does not consider composition independencies.

CHAPTER 6

CONCLUSION

Table of Contents

6.1 Introduction

Since the late 90s and up to now XML has invaded the communication networks in the computer domain, whether it is over the internet (e.g., social networks), intranets (e.g., web services), or offline (e.g., integration between different applications), and whether it is from user to user (e.g., instant messaging), user to machine (e.g., data insertion/removal), or machine to machine (e.g., communication between applications). Thus, the use of XML is not limited anymore to computer scientists, but has become in the grasp of users from other areas such as commercial, medical, social and others. As a consequence, it has become more and more imperative to allow *non-expert users* to *control or manipulate* their *XML communication/data* on *different platforms/environments* (online and offline), even though they may not necessarily have any experience in computer science, programming or XML.

In this research, we analyzed and explored these issues from different perspectives with regard to the literature. Unfortunately, most existing approaches/techniques address "XML manipulations by non-experts" from different angles but provide only partial related solutions. They are grouped in 4 main categories:

(a) XML-oriented querying and transformation visual languages

(b) XML-oriented Mashup tools

(c) XML manipulation techniques

(d) DFVPL (Dataflow Visual Programming Languages)

None of these approaches provided a full-fledged solution for non-expert users allowing them to create and enforce XML-oriented manipulation operations over different platforms and in different environments.

This study was dedicated to defining a new approach/framework for creating and enforcing XML manipulation operations by non-expert users, called XA2C (XML mAnipulAtion Composition). The main contributions of our work are described below.

6.2 Contributions

The XA2C framework is a visual studio for an XML-oriented DFVPL called XCDL (XML-oriented Composition Definition Language) allowing non-expert users to manipulate XML-based data. The major contributions can be elaborated from the XA2C approach, the XCDL language, and the X-Man prototype as discussed in the following subsections.

6.2.1 The XA2C approach

Six contributions are related to the XA2C approach:

- *Originality:* Our approach, to the best of our knowledge, is the first XML-oriented VPL based on the dataflow paradigm
- *Intuitiveness*: Since Dataflows are considered to be the closest to the natural human thinking process, thus XA2C inherits their features and particularly their simplicity and clarity.
- *Expressiveness*: The usage of the Dataflow paradigm makes the approach more expressive than existent XML-oriented visual languages, since it is specifically designed for data manipulations and allowing various kinds of operations
- *Portability:* The XA2C framework defines its data models using XML. Consequently, the framework is rendered platform and environment free, has no constraints, and can be deployed anywhere online or offline
- *Derivability and Extensibility*: The framework is formally defined, along with the language and based mainly on CP-nets (colored petri nets). Thus, the approach is rendered derivable. As well, its formal aspects allows it to be further studied and developed by computer scientists, and gives it the ability to embed new features such as flow control, error handling mechanisms currently lacking in existing approaches, etc. While most researchers agree that such formalities render the reading and understanding of the material difficult, having such definitions provides a strong foundation for future works, and allows a better analysis and debugging of the language.
- *Modularity:* The XA2C framework is defined as a modular architecture and contains 3 modules which respectively define the XCDL syntax as a DFVPL, the XCDL compiler translating the compositions into machine code, and the XCDL runtime environment for executing the translated compositions. Separating the framework into a language platform, a compiler and a Runtime Environment provides a well-defined and structured approach which clearly separates the compositions from the executions. Therefore, this removes any confusions regarding when one ends and the other begins, and allows a better evaluation and assessment of the language separately from the other modules.

6.2.2 The XCDL language

X contributions are related to the XCDL language:

- *Ease of Use:* In the XCDL platform, the main module of our framework, we defined the syntax and semantics of our DFVPL (XCDL) based on our CP-net algebraic grammar, called XCGN (XML Composition Grammar Net), and OLT, called XCD-tree. By defining the language as a DFVPL, XCDL was rendered suitable for non-expert users, since it bases its compositions on the natural human thinking process. This rendered the language most appropriate for non-expert users.
- *Adaptability and Genericity:* To render the language oriented towards XML manipulations, we separated the syntax of the language's inputs and outputs from the main composition syntax. I/Os are defined as XCD-trees (OLT) which represent the structure of any XML-based data (e.g., XML documents, fragments, XML DTDs, and XML schemas) as tree views. Separating the Dataflow's I/O and using OLT for representing XML data, orient XCDL towards XML and define it as an XML-oriented DFVPL which lacks so far in the literature. In addition, XCDL gains the advantage of being generic to any structured data, not only XML.
- *Poly-syntaxity:* XCDL is defined as a visual language and provides a separate syntax for the graphical representations from the language syntax. Thus, the language has been rendered flexible and allows the definition of different visual syntaxes which can be adapted to different environments. From a mere user's point of view, each visual syntax is a unique language (similar to C# and visual basic). Such languages have separate syntaxes, although, they share a common core (the dotnet framework).
- *Operability:* XCDL is a DFVPL allowing users to create their manipulation operations through visual compositions by simply linking the outputs and inputs of *SD-functions*. Since these functions can be identified from offline libraries (e.g., DLL and Jar files) or online libraries (e.g., web services), this renders the language extensible with new operations.

6.2.3 Prototype and Evaluation

In addition to the theoretical contribution, we developed X-Man, a prototype for evaluating and testing the X2AC framework. The prototype was used to test our algorithms and evaluate the language and framework in real case scenarios. Since we were not able to find any frameworks in the literature for evaluating VPLs, we formally defined an evaluation framework assessing the overall quality of a VPL.

It was used in several case studies with a number of participants to evaluate and compare XCDL, YahooPipes and IBM Damia. Two main contributions were discovered. First, the XCDL evaluation returned positive results and validated the usability of our approach. Second, XCDL was assessed to be superior to both YahooPipes and IBM Damia. In particular, the visualization aspects of XCDL were the most favorable, validating our choice for using CP-Nets as the basis of our graphical representations.

6.3 Future Works

After defining our XA2C framework and the XCDL core for manipulating XML data by non-experts, five main future directions became possible: (i) language extensibility, (ii) language derivability, (iii) automated composition, (vi) technical enhancements, and (v) better assessment.

6.3.1 XCDL Extensibility

In our research, since we defined the basis of an XML-oriented visual studio and the XCDL language core, we can take it to the next level and start extending the language to render it more expressive. XCDL should be extended with some new features and operators categorized as follows:

- Language extensions:
    - *N-ary SD-functions*: they are functions that can have *n* inputs and outputs which can be defined by the user (e.g., consider a *concat* function, it can be defined as an n-ary function where the user chooses the number of inputs to be concatenated instead of having only 2)
    - *Composed SD-functions*: they are functions created via compositions. Such functions can allow users to reuse and derive new operations, and simplify the composition task by rendering it less complex
    - *Condition operators*: they are operators similar to "*switch cases*" and "*if else*" statements in textual languages. Such operators are essential since they can provide flow control
    - *Recursive operators*: they are operators similar to "*for, while* and *do-while*" loops which can be particularly useful in XML manipulation, since XML is mostly based on repeated patterns
    - *Error handling operators*: they are operators similar to "*try catch*" statements which can provide the user with error handling. These

operators, which normally lack in VPLs can be defined using the XCGN grammar since it is based on petri nets and thus can take advantage of the petri nets' analysis functions

- *Common operators*: they are generic and commonly used operator such as the *fork* and *union* operators used respectively to duplicate and merge any type of data
- *Multiple I/O XCD-trees*: they allow users to manipulate XML data from several separate sources[1]
- *Function execution time:* this allows each function to have a specific execution interval time.

- Interface/interaction extensions:
  - *Zoom feature*: this feature will allow users to zoom in and out on their compositions which can be very helpful as the compositions get more complex and the visualization gets messy
  - *Drag & drop:* allows users to compose their operations by simply dragging and dropping their functions.

6.3.2 XCDL Derivability

While designing the XCDL language, the initial purpose was to define a language generic to all XML-based data. Nevertheless, different communication standards have emerged from the XML standard such as RSS (Really Simple Syndication), Atom, RDF (Resource Description Framework), SMIL (Synchronized Multimedia Integration Language), etc. Each of these languages has its own semantics and goals and is widely used. It would be interesting to derive specific languages from XCDL such as:

- RSS-XCDL: RSS-oriented XML composition definition language
- RDF-XCDL: RDF-oriented XML composition definition language
- SMIL-XCDL: SMIL-oriented XML composition definition language

Such languages would have their own specific and detailed I/O XCD-tree templates and *SD-functions* which are oriented semantically towards manipulating these specific data types. This can be done by defining new data types which are subtypes of the initial XCD-nodes (XCD-node:Element, Attribute and Text).

[1] In the current approach we use only one input structure and one output structure.

6.3.3 Automated Composition

Since XA2C has been developed mainly for non-expert users, it is essential to render the creation task as simple and easy as possible. So far, the user has to compose all his operations manually. The XCDL language is based on CP-Nets and thus it can be extended to hold information related to semantic data and meta-data, function cost (in terms of bandwidth, size, etc.), function quality of service, etc. Such information can be used in the future to provide automated compositions. These compositions can be created dynamically based on the semantics of the user's description of the required manipulation operation.

As an example consider the following description: *"I want to extract all the books which have been published in 1983 and related to XML"*.

Based on this description and through keywords extraction, and the measurement of semantic relatedness, a composition can be automatically created and suggested to the user. This composition may be complete or partial depending on the *SD-functions* available in the language.

6.3.4 Technical enhancements

In order to improve the usability of XCDL, three features can be added such as:

- *Online function repository*: it creates an online repository allowing the storage of SD-functions that can be downloaded by X-Man users when required
- *Function discovery*: it defines a universal registry allowing functions to be discovered and located by XA2C users
- *Open Source prototype*: it provides an online open source version of the X-Man that can be updated and extended with new features by programmers around the world

6.3.5 Better Assessment

To better evaluate our approach, three features can be added:

- *Enhanced evaluations:* it designates the evaluation of the approach using different user profiles from different disciplines (i.e., students from different universities and faculties)
- *Online prototype:* it adds an online version of the X-Man allowing an online evaluation of the approach from different platforms/environments
- *Application domain evaluation*: it defines and runs different assessment tests oriented towards specific application domains such as security, adaptability, scalability, and others.

Providing such features will allow the quality of the framework to increase. In particular, it will render the practical aspect of the XA2C more mature and accelerate its evolution.

References

[1] W. Ackerman, "Data flow languages," *IEEE Comput. 15, 2*, pp. 15-25, 1982.

[2] B. Adelberg, "NoDoSE-a tool for semi-automatically extracting structured and semistructured data from text documents," *SIGMOD Rec.*, vol. 27, pp. 283-294, 1998.

[3] M. Altinel and M. J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," in *Proceedings of the 26th International Conference on Very Large Data Bases*, 2000, pp. 53-64.

[4] M. Amamiya, *et al.*, "Valid, a high-level functional programming language for data flow machines " in *Rev. Electric. Comm. Lab. 32, 5*, 1984, pp. 793-802.

[5] Arvind and D. E. Culler, "Dataflow architectures," in *Annual review of computer science vol. 1, 1986*, ed: Annual Reviews Inc., 1986, pp. 225-253.

[6] Arvind, *et al.*, "An asynchronous programming language and computing machine," in *Tech. Rep. TR 114a*, ed Irvine, CA: University of California, 1978, p. 8.

[7] K. Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Trans. Comput.*, vol. 39, pp. 300-318, 1990.

[8] E. A. Ashcroft and W. W. Wadge, "Lucid, a nonprocedural language with iteration," *Commun. ACM*, vol. 20, pp. 519-526, 1977.

[9] M. Auguston and A. Delgado, "Iterative Constructs in the Visual Data Flow Language," in *Proceedings of the 1997 IEEE Symposium on Visual Languages*, 1997, pp. 152 - 159.

[10] M. Auguston and A. Delgado, "The V experimental visual programming language," in *Technical report NMSU-CSTR-9611*, ed: New Mexico State University, 1996, p. 6.

[11] E. Baroth and C. Hartsough, "Visual programming in the real world," in *Visual object-oriented programming*, ed: Manning Publications Co., 1995, pp. 21-42.

[12] J. Billington, *et al.*, "The Petri net markup language: concepts, technology, and tools," in *Proceedings of the 24th international conference on Applications and theory of Petri nets*, Eindhoven, The Netherlands, 2003, pp. 483-505.

[13] D. Box, *et al.* (2003), *Simple Object Access Protocol (SOAP) 1.1*. Available: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[14] Boyer. (2000), *Voice eXtensible Markup Language 1.0*. Available: http://www.w3.org/TR/2000/NOTE-voicexml-20000505/

[15] D. Braga, *et al.*, "XQBE (XQuery By Example): a visual interface to the standard XML query language," *ACM Trans. Database Syst.*, vol. 30, pp. 398-443, 2005.

[16] D. Bulterman. (2005), *Synchronized Multimedia Integration Language (SMIL 2.1) Specification*. Available: http://www.w3.org/TR/2005/REC-SMIL2-20051213/

[17] P. Buneman, *et al.*, "Principles of programming with complex objects and collection types," in *Selected papers of the fourth international conference on Database theory*, Berlin, Germany, 1995, pp. 3-48.

[18] C. Byun, *et al.*, "A Keyword-Based Filtering Technique of Document-Centric XML using NFA Representation," *International Journal of Applied Mathematics Computer Science*, pp. 136–143, 2007.

[19] K. S. Candan, *et al.*, "AFilter: adaptable XML filtering with prefix-caching suffix-clustering," in *Proceedings of the 32nd international conference on Very large data bases*, Seoul, Korea, 2006, pp. 559-570.

[20] S. K. Card, *et al.*, "Readings in information visualization: using vision to think," ed San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 686.

[21] S. Ceri, *et al.*, "Complex queries in XML-GL," in *Proceedings of the 2000 ACM symposium on Applied computing - Volume 2*, Como, Italy, 2000, pp. 888-893.

[22] S. Ceri, *et al.*, "XML-GL: A Graphical Language for Querying and Restructuring XML Documents," in *SEBD*, ed, 1999, pp. 151-165.

[23] C.-H. Chang and S.-C. Lui, "IEPAD: information extraction based on pattern discovery," in *Proceedings of the 10th international conference on World Wide Web*, Hong Kong, Hong Kong, 2001, pp. 681-688.

[24] C. Chen and M. P. Czerwinski, "Empirical evaluation of information visualizations: an introduction," *Int. J. Hum.-Comput. Stud.*, vol. 53, pp. 631-635, 2000.

[25] P. Cox, *et al.*, "Prograph: A step towards liberating programming from textual conditioning " in *Procedings of the IEEE Workshop on Visual Languages*, 1989, pp. 150-156.

[26] P. T. Cox, *et al.*, "Prograph," in *Visual Object-Oriented Programming, Concepts and Environments*, A. G. M. Burnett, T. Lewis, Ed., ed Manning, 1995, pp. 45-66.

[27] V. Crescenzi, *et al.*, "Automatic Web Information Extraction in the ROADRUNNER System," in *Revised Papers from the HUMACS, DASWIS, ECOMO, and DAMA on ER 2001 Workshops*, 2002, pp. 264-277.

[28] T. Dalamagas, *et al.*, "A methodology for clustering XML documents by structure," *Inf. Syst.*, vol. 31, pp. 187-228, 2006.

[29] E. Damiani, *et al.*, "A fine-grained access control system for XML documents," *ACM Trans. Inf. Syst. Secur.*, vol. 5, pp. 169-202, 2002.

[30] A. L. Davis, "The architecture and system method of DDM1: A recursively structured Data Driven Machine," in *Proceedings of the 5th annual symposium on Computer architecture*, 1978, pp. 210-215.

[31] A. L. Davis, "Data driven nets---A class of maximally parallel, output-functional program schemata " in *Tech. Rep. IRC Report* ed. San Diego, CA.: Burroughs, 1974.

[32] A. L. Davis, "DDN's---a low level program schema for fully distributed systems " in *Proceedings of the 1st European Conference on Parallel and Distributed Systems*, Toulouse, France, 1979, pp. 1-7.

[33] A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *Computer*, vol. 15, pp. 26-41, 1982.

[34] A. L. Davis and S. A. Lowder, "A Sample management application program in a graphical data-driven programming language," in *Digest of Papers Compcon Spring*, 1981, pp. 162–165.

[35] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium, Proceedings Colloque sur la Programmation*, 1974, pp. 362-376.

[36] J. B. Dennis, "Data Flow Supercomputers," *Computer,* vol. 13, pp. 48-56, 1980.

[37] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *SIGARCH Comput. Archit. News,* vol. 3, pp. 126-132, 1975.

[38] Y. Diao, *et al.*, "Path sharing and predicate evaluation for high-performance XML filtering," *ACM Trans. Database Syst.,* vol. 28, pp. 467-516, 2003.

[39] R. J. Ennals and M. N. Garofalakis, "MashMaker: mashups for the masses," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, Beijing, China, 2007, pp. 1116-1118.

[40] M. Erwig, "A Visual Language for XML," *Visual Languages, IEEE Symposium on*, vol. 0, pp. 47-54, 2000.

[41] D. F. Ferraiolo, *et al.*, "Proposed NIST standard for role-based access control," *ACM Trans. Inf. Syst. Secur.,* vol. 4, pp. 224-274, 2001.

[42] J. Ferraiolo. (2003), *Scalable Vector Graphics (SVG) 1.1 Specification*. Available: http://www.w3.org/TR/2003/REC-SVG11-20030114/

[43] I. Fundulaki and S. Maneth, "Formalizing XML access control for update operations," in *Proceedings of the 12th ACM symposium on Access control models and technologies*, Sophia Antipolis, France, 2007, pp. 169-174.

[44] O. Gelly, "LAU software system: A high-level data-driven language for parallel processing," in *In Proceedings of the International Conference on Parallel Processing*, USA, 1976.

[45] J. R. W. Glauert, "A single assignment language for data flow computing," Master's thesis, University of Manchester, Manchester, U.K., 1978.

[46] E. J. Golin and S. P. Reiss, "The specification of visual language syntax," *J. Vis. Lang. Comput.,* vol. 1, pp. 141-157, 1990.

[47] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A "Cognitive Dimensions" Framework," *J. Vis. Lang. Comput,* vol. 7, pp. 131-174, 1996.

[48] C. L. Hankin and H. W. Glaser, "The data flow programming language CAJOLE - an informal introduction," *SIGPLAN Not.,* vol. 16, pp. 35-44, 1981.

[49] M. A. Harrison, *et al.*, "Protection in operating systems," *Commun. ACM,* vol. 19, pp. 461-471, 1976.

[50] N. Harvey and J. Morris, "NL: A general purpose visual dataflow language " in *Tech. Rep.* , ed. Tasmania, Australia: University of Tasmania, 1993, p. 33.

[51] J. Heasley, "Securing XML data," in *Proceedings of the 1st annual conference on Information security curriculum development*, Kennesaw, Georgia, 2004, pp. 112-114.

[52] K. Heyman, "A New Virtual Private Network for Today's Mobile World," *Computer,* vol. 40, pp. 17-19, 2007.

[53] J. Hidders, *et al.*, "DFL: A dataflow language based on Petri nets and nested relational calculus," *Inf. Syst.,* vol. 33, pp. 261-284, 2008.

[54] J. Hidders, *et al.*, "Petri Net + Nested Relational Calculus = Dataflow.," in *OTM Conferences (1)*, ed, 2005, pp. 220-237.

[55] L. M. Hillah, *et al.*, "A primer on the Petri Net Markup Language and ISO/IEC 15909-2" in *10th International workshop on Practical Use of Colored Petri Nets and the CPN Tools -- CPN'09*, 2009, pp. 9-28.

[56] D. D. Hils, "Visual languages and computing survey: Data flow visual programming languages," *J. Vis. Lang. Comput.,* vol. 3, pp. 69-101, 1992

[57] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.,* vol. 21, pp. 359-411, 1989.

[58] G.-H. Hwang and T.-K. Chang, "An operational model and language support for securing XML documents," *Computers & Security,* vol. 23, pp. 498 - 529, 2004.

[59] C. S. IBM-Group, "IBM WebSphere DataPower XML Security Gateway XS40 " in *IBM Corporation 2008*, ed, 2008 pp. 2-4.

[60] M. Iwata and H. Terada, "Multilateral diagrammatical specification environment based on data-driven paradigm," in *Advanced Topics in Dataflow Computing and Multithreading*, Los Alamitos, CA, 1995, pp. 103-112.

[61] K. Jensen, "An Introduction to the Theoretical Aspects of Coloured Petri Nets," in *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, 1994, pp. 230-272.

[62] W. M. Johnston, *et al.*, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, pp. 1-34, 2004.

[63] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Information Processing '74: Proceedings of the IFIP Congress*, 1974, pp. 471-475.

[64] A. A. E. Kalam, *et al.*, "Organization based access control," in *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, 2003, pp. 120 - 131.

[65] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J. Appl. Math.*, pp. 1390-1411, 1966.

[66] M. Kay. (2007), *XSL Transformations (XSLT) Version 2.0* Available: http://www.w3.org/TR/2007/REC-xslt20-20070123/

[67] R. M. Keller and W. C. J. Yen, "A graphical approach to software development using function graphs," *Digest of Papers Compcon Spring*, pp. 156-161, 1981.

[68] I. Kofler, *et al.*, "Towards MPEG-21-Based Cross-Layer Multimedia Content Adaptation," in *Proceedings of the Second International Workshop on Semantic Media Adaptation and Personalization*, 2007, pp. 3-8.

[69] B. Lee and A. R. Hurson, "Dataflow Architectures and Multithreading," *Computer*, vol. 27, pp. 27-39, 1994.

[70] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, pp. 24-35, 1987.

[71] T. Lemlouma and N. Layaïda, "SMIL Content Adaptation for Embedded Devices," in *IN SMIL EUROPE 2003 CONFERENCE*, 2003, pp. 12-14.

[72] C.-H. Lim, *et al.*, "Access control of XML documents considering update operations," in *Proceedings of the 2003 ACM workshop on XML security*, Fairfax, Virginia, 2003, pp. 49-59.

[73] G. D. Lorenzo, *et al.*, "Data integration in mashups," *SIGMOD Rec.*, vol. 38, pp. 59-66, 2009.

[74] T. Loton, "Introduction to Microsoft Popfly, No Programming Required," in *Bibliometrics*, ed: Lotontech Limited, 2008, p. 128.

[75] B. Luo, *et al.*, "QFilter: fine-grained run-time XML access control via NFA-based query rewriting," in *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, Washington, D.C., USA, 2004, pp. 543-552.

[76] D. Marghescu, *et al.*, "Evaluating the quality of use of visual data-mining tools," in *11th European Conference on IT Evaluation*, Netherland, 2004, pp. 239-250.

[77] J. McGraw and S. Skedzielewski, "Streams and Iteration in a Single Assignment Language Reference Manual (Version 1.0)," ed. Livermore, CA.: Livermore National Laboratory, 1983.

[78] J. P. Morrison, "Flow-Based Programming: A New Approach to Application Development," ed. New York, NY: van Nostrand Reinhold, 1994, p. 240.

[79] T. Murata, "Petri Nets: Properties, Analysis and Applications.," in *Proceedings of the IEEE*, ed, 1989, pp. 541-580.

[80] T. Oinn, *et al.*, "Taverna: lessons in creating a workflow environment for the life sciences: Research Articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, pp. 1067-1100, 2006.

[81] C. Ouyang, *et al.*, "Formal semantics and analysis of control flow in WS-BPEL," *Sci. Comput. Program.*, vol. 67, pp. 162-198, 2007.

[82] G. M. Papadopoulos and K. R. Traub, "Multithreading: A revisionist view of dataflow architectures," in *Proceedings of the 18th annual international symposium on Computer architecture* Toronto, Ontario, Canada 1991, pp. 342--351.

[83] J. Park and R. Sandhu, "The UCONabc usage control model," *ACM Trans. Inf. Syst. Secur.*, vol. 7, pp. 128-174, 2004.

[84] B. Pellan and C. Concolato, "Adaptation of scalable multimedia documents," in *Proceeding of the eighth ACM symposium on Document engineering*, Sao Paulo, Brazil, 2008, pp. 32-41.

[85] S. Pemberton. (2002), *The Extensible HyperText Markup Language: A Reformulation of HTML 4.0 in XML 1.0*. Available: http://www.w3.org/TR/2002/REC-xhtml1-20020801/

[86] S. Pettifer, *et al.*, "myGrid and UTOPIA: an integrated approach to enacting and visualising in silico experiments in the life sciences," in *Proceedings of the 4th international conference on Data integration in the life sciences*, Philadelphia, PA, USA, 2007, pp. 59-70.

[87] M. Piatek, "Distributed web proxy caching in a local network environment," *The Student Research Competition*, p. 8, 2004.

[88] E. Pietriga, *et al.*, "VXT: a visual approach to XML transformations," in *Proceedings of the 2001 ACM Symposium on Document engineering*, Atlanta, Georgia, USA, 2001, pp. 1-10.

[89] C. Richardson, "Manipulator control using a data-flow machine," Doctoral dissertation, University of Manchester, Manchester, U.K., 1981. .

[90] B. Shizuki, *et al.*, "Smart browsing among multiple aspects of data-flow visual program execution, using visual patterns and multi-focus fisheye views," *J. Vis. Lang. Comput.* , vol. 11, pp. 529-548, 2000.

[91] A. Shurr, "BDL - A Nondeterministic Data Flow Programming Language with Backtracking," in *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97)*, 1997, pp. 394 - 401.

[92] J. Silc, *et al.*, "Asynchrony in parallel computing: from dataflow to multithreading," in *Progress in computer research*, ed: Nova Science Publishers, Inc., 2001, pp. 1-33.

[93] D. E. Simmen, *et al.*, "Damia: data mashups for intranet applications," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Vancouver, Canada, 2008, pp. 1171-1182.

[94] Sterling, *et al.*, "Studies on Optimal Task Granularity and Random Mapping," in *Advanced Topics in Dataflow Computing and Multithreading IEEE Computer Society Press*, 1995, pp. 349-365.

[95] G. Tekli, *et al.*, "XCDL: an XML-Oriented Visual Composition Definition Language," in *The 12th International Conference on Information Integration and Web-based Applications & Services*, 2010, pp. 134-143.

[96] G. Tekli, *et al.*, "XA2C Framework for XML Alteration/Adaptation," in *Reliable and Autonomous Computational Science*, S. Y. Shin, *et al.*, Eds., ed: Springer Basel, 2010, pp. 327-346.

[97] G. Tekli, *et al.*, "Towards an XML Adaptation/Alteration Control Framework," in *Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*, 2010, pp. 248-255.

[98] R. K. Thomas and R. S. Sandhu, "Task-Based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-Oriented Autorization Management," in *Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Securty XI: Status and Prospects*, 1998, pp. 166-181.

[99] C. Timmerer and H. Hellwagner, "Interoperable Adaptive Multimedia Communication," *IEEE MultiMedia*, vol. 12, pp. 74-79, 2005.

[100] P. C. Treleaven, *et al.*, "Data-Driven and Demand-Driven Computer Architecture," *ACM Comput. Surv.*, vol. 14, pp. 93-143, 1982.

[101] D. Turi, *et al.*, "Taverna Workflows: Syntax and Semantics," in *Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, 2007, pp. 441-448.

[102] W3C. (1999), *Extensible Stylesheet Language Transformations -XSLT 1.0*. Available: http://www.w3.org/TR/xslt

[103] W3C. (1999), *XML Path Language (XPath) Version 1.0*. Available: http://www.w3.org/TR/xpath/

[104] W3C. (2010), *XQuery 1.0: An XML Query Language (Second Edition)*. Available: http://www.w3.org/TR/xquery/

[105] W3C. (2000), *What is the Document Object Model*. Available: http://www.w3.org/TR/DOM-Level-2-Core/introduction.html

[106] I. Watson and J. Gurd, "A prototype data flow computer with token labelling," in *Proceedings of the National Computer Conference*, Los Alamitos, CA, USA, 1979, p. 623.

[107] K. S. Weng, "Stream oriented computation in recursive data-flow schemas," in *Tech. Rep. 68. Laboratory for computer science*, ed. MA: MIT, Cambridge, 1975, pp. 303-325.

[108] T. Wright, "Security, privacy, and anonymity," *Crossroads Magazine*, vol. 11, pp. 5-5, 2004.

[109] H. Xu, *et al.*, "Formal modelling and analysis of XML firewall for service-oriented systems," *International Journal of Security and Networks*, vol. 3, pp. 147-160, 2008.

[110] Y. Yu and T.-c. Chiueh, "Enterprise Digital Rights Management: Solutions against Information Theft by Insiders," in *Research Proficiency Examination* ed, 2004, pp. 2-24.

[111] B. A. Zenel, "A proxy-based filtering mechanism for the mobile environment," Doctoral Dissertation, Columbia University, 1998.

[112] B. Zhao, *et al.*, "Towards a times-based usage control model," in *Proceedings of the 21st annual IFIP WG 11.3 working conference on Data and applications security*, Redondo Beach, CA, USA, 2007, pp. 227-242.

List of Publications

International Conferences

G. Tekli, *et al.*, "XCDL: an XML-Oriented Visual Composition Definition Language," in *The 12th International Conference on Information Integration and Web-based Applications & Services (iiWAS2010)*, 2010, pp. 134-143.

G. Tekli, *et al.*, "XA2C Framework for XML Alteration/Adaptation," in *Reliable and Autonomous Computational Science*, S. Y. Shin, *et al.*, Eds., ed: Springer Basel, 2010, pp. 327-346.

G. Tekli, *et al.*, "Towards an XML Adaptation/Alteration Control Framework," in *Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*, 2010, pp. 248-255.

International Journals

G. Tekli, *et al.*, "XA2C, a framework for manipulating XML data", *International Journal of Web Information Systems*, 2011

G. Tekli, et al, "A Visual Programming Language for XML manipulation", *Journal of Visual Languages and Computing*, 2011 (submitted)

Appendix A-*XML schema of an SD-function*

```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="sd_function"
targetNamespace="http://tempuri.org/sd_function.xsd"
xmlns="http://tempuri.org/sd_function.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
attributeFormDefault="qualified" elementFormDefault="qualified">
 <xs:element name="sd_function">
  <xs:complexType>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="SD_function">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="SD_function_id" type="xs:string" />
       <xs:element name="SD_function_name" type="xs:string" />
       <xs:element name="SD_function_desc" type="xs:string" minOccurs="0" />
       <xs:element name="SD_function_Type_id" type="xs:string"
minOccurs="0" />
        <xs:element name="category" type="xs:string" default="Others"
minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="SD_function_type">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="SD_function_type_id" type="xs:string" />
       <xs:element name="SD_function_type" type="xs:string" minOccurs="0" />
       <xs:element name="SD_function_type_desc" type="xs:string"
minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="SD_function_color">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="color_id" type="xs:string" />
       <xs:element name="color_name" type="xs:string" minOccurs="0" />
       <xs:element name="color_value" type="xs:string" minOccurs="0" />
       <xs:element name="color_desc" type="xs:string" minOccurs="0" />
       <xs:element name="color_color" type="xs:string" default="-1"
minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
```

```
    <xs:element name="SD_function_transition">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="transition_id" type="xs:string" />
          <xs:element name="transition_name" type="xs:string" minOccurs="0" />
          <xs:element name="transition_type" type="xs:string" minOccurs="0" />
          <xs:element name="transition_value" type="xs:string" minOccurs="0" />
          <xs:element name="transition_desc" type="xs:string" minOccurs="0" />
          <xs:element name="SD_function_id" type="xs:string" minOccurs="0" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="SD_function_place">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="place_id" type="xs:string" />
          <xs:element name="place_name" type="xs:string" minOccurs="0" />
          <xs:element name="color_id" type="xs:string" minOccurs="0" />
          <xs:element name="place_desc" type="xs:string" minOccurs="0" />
          <xs:element name="place_init" type="xs:string" minOccurs="0" />
          <xs:element name="SD_function_id" type="xs:string" minOccurs="0" />
          <xs:element name="place_in_out" type="xs:string" minOccurs="0" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:complexType>
<xs:unique name="SD_function" >
  <xs:selector xpath=".//SD_function" />
  <xs:field xpath="SD_function_id" />
</xs:unique>
<xs:unique name="SD_function_type">
  <xs:selector xpath=".//SD_function_type" />
  <xs:field xpath="SD_function_type_id" />
</xs:unique>
<xs:unique name="Color" >
  <xs:selector xpath=".//SD_function_color" />
  <xs:field xpath="color_id" />
</xs:unique>
<xs:unique name="transition" >
  <xs:selector xpath=".//SD_function_transition" />
  <xs:field xpath="transition_id" />
</xs:unique>
<xs:unique name="place">
  <xs:selector xpath=".//SD_function_place" />
  <xs:field xpath="place_id" />
```

```
    </xs:unique>
    <xs:keyref name="SD_function_place" refer="SD_function">
      <xs:selector xpath=".//SD_function_place" />
      <xs:field xpath="SD_function_id" />
    </xs:keyref>
    <xs:keyref name="type_place" refer="Color" >
      <xs:selector xpath=".//SD_function_place" />
      <xs:field xpath="color_id" />
    </xs:keyref>
    <xs:keyref name="SD_function_transition" refer="SD_function" >
      <xs:selector xpath=".//SD_function_transition" />
      <xs:field xpath="SD_function_id" />
    </xs:keyref>
    <xs:keyref name="XCType_SD_function" refer="SD_function_type" >
      <xs:selector xpath=".//SD_function" />
      <xs:field xpath="SD_function_Type_id" />
    </xs:keyref>
  </xs:element>
</xs:schema>
```

Appendix B-*XML schema of a composition*

```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="ds_Composition"
targetNamespace="http://tempuri.org/ds_Composition.xsd"
xmlns="http://tempuri.org/ds_Composition.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
attributeFormDefault="qualified" elementFormDefault="qualified">
  <xs:element name="ds_Composition">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">

           <xs:element name="SD_function">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="SD_function_id" type="xs:string" />
            <xs:element name="SD_function_name" type="xs:string" />
            <xs:element name="SD_function_desc" type="xs:string" minOccurs="0" />
            <xs:element name="SD_function_type_id" type="xs:string"
minOccurs="0" />
            <xs:element name="category" type="xs:string" default="Others"
minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="SD_function_type">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="SD_function_type_id" type="xs:string" />
            <xs:element name="SD_function_type" type="xs:string" minOccurs="0" />
            <xs:element name="SD_function_type_desc" type="xs:string"
minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="SD_function_color">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="color_id" type="xs:string" />
            <xs:element name="color_name" type="xs:string" minOccurs="0" />
            <xs:element name="color_value" type="xs:string" minOccurs="0" />
            <xs:element name="color_desc" type="xs:string" minOccurs="0" />
            <xs:element name="color_color" type="xs:string" default="-1"
minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
```

```
    </xs:element>
    <xs:element name="SD_function_transition">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="transition_id" type="xs:string" />
       <xs:element name="transition_name" type="xs:string" minOccurs="0" />
       <xs:element name="transition_type" type="xs:string" minOccurs="0" />
       <xs:element name="transition_value" type="xs:string" minOccurs="0" />
       <xs:element name="transition_desc" type="xs:string" minOccurs="0" />
       <xs:element name="SD_function_id" type="xs:string" minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="SD_function_place">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="place_id" type="xs:string" />
       <xs:element name="place_name" type="xs:string" minOccurs="0" />
       <xs:element name="color_id" type="xs:string" minOccurs="0" />
       <xs:element name="place_desc" type="xs:string" minOccurs="0" />
       <xs:element name="place_init" type="xs:string" minOccurs="0" />
       <xs:element name="SD_function_id" type="xs:string" minOccurs="0" />
       <xs:element name="place_in_out" type="xs:string" minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>

            <xs:element name="Composition">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Composition_id" type="xs:string" />
       <xs:element name="Composition_name" type="xs:string"
minOccurs="0" />
       <xs:element name="Composition_desc" type="xs:string" minOccurs="0" />
       <xs:element name="Composition_type_id" type="xs:string"
minOccurs="0" />
       <xs:element name="in_xml_path" type="xs:string" minOccurs="0" />
       <xs:element name="out_xml_path" type="xs:string" minOccurs="0" />
       <xs:element name="cpn_path" type="xs:string" minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="Composition_type">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Composition_type_id" type="xs:string" />
```

```
            <xs:element name="Composition_type" type="xs:string" minOccurs="0" />
            <xs:element name="Composition_type_desc" type="xs:string"
minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Composition_SD_function">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Composition_id" type="xs:string" />
            <xs:element name="SD_function_id" type="xs:string" />
            <xs:element name="Composition_op" type="xs:string" minOccurs="0" />
            <xs:element name="SD_function_index" type="xs:int" default="0"
minOccurs="0" />
            <xs:element name="iteration" type="xs:int" default="1" />
            <xs:element name="Composition_SD_function_id" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Composition_SD_function_places">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Composition_SD_function_id" type="xs:string" />
            <xs:element name="place_id" type="xs:string" />
            <xs:element name="place_mapping" type="xs:string" default="" />
            <xs:element name="place_init" type="xs:string" minOccurs="0" />
            <xs:element name="place_mapping_type" type="xs:string"
minOccurs="0" />
            <xs:element name="place_value" type="xs:string" minOccurs="0" />
            <xs:element name="cycle" type="xs:string" default="0" minOccurs="0" />
            <xs:element name="place_in_out" type="xs:string" minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>

          </xs:choice>
    </xs:complexType>

          <xs:unique name="SD_function">
      <xs:selector xpath=".//SD_function" />
      <xs:field xpath="SD_function_id" />
    </xs:unique>
    <xs:unique name="SD_function_type">
      <xs:selector xpath=".//SD_function_type" />
      <xs:field xpath="SD_function_type_id" />
    </xs:unique>
```

```
    <xs:unique name="SD_function_color">
      <xs:selector xpath=".//SD_function_color" />
      <xs:field xpath="color_id" />
    </xs:unique>
    <xs:unique name="SD_function_transition">
      <xs:selector xpath=".//SD_function_transition" />
      <xs:field xpath="transition_id" />
    </xs:unique>
    <xs:unique name="SD_function_place">
      <xs:selector xpath=".//SD_function_place" />
      <xs:field xpath="place_id" />
    </xs:unique>
    <xs:unique name="Composition_id">
      <xs:selector xpath=".//Composition" />
      <xs:field xpath="Composition_id" />
    </xs:unique>
    <xs:unique name="Composition_type_id">
      <xs:selector xpath=".//Composition_type" />
      <xs:field xpath="Composition_type_id" />
    </xs:unique>
    <xs:unique name="Composition_SD_function">
      <xs:selector xpath=".//Composition_SD_function" />
      <xs:field xpath="Composition_SD_function_id" />
    </xs:unique>
    <xs:keyref name="FK_SD_function_place_Composition_SD_function_places"
refer="SD_function_place">
      <xs:selector xpath=".//Composition_SD_function_places" />
      <xs:field xpath="place_id" />
    </xs:keyref>
    <xs:keyref
name="FK_Composition_SD_function_Composition_SD_function_places"
refer="Composition_SD_function">
      <xs:selector xpath=".//Composition_SD_function_places" />
      <xs:field xpath="Composition_SD_function_id" />
    </xs:keyref>
    <xs:keyref name="FK_SD_function_Composition_SD_function"
refer="SD_function">
      <xs:selector xpath=".//Composition_SD_function" />
      <xs:field xpath="SD_function_id" />
    </xs:keyref>
    <xs:keyref name="FK_Composition_Composition_SD_function"
refer="Composition_id">
      <xs:selector xpath=".//Composition_SD_function" />
      <xs:field xpath="Composition_id" />
    </xs:keyref>
```

```
   <xs:keyref name="FK_Composition_type_Composition"
refer="Composition_type_id">
     <xs:selector xpath=".//Composition" />
     <xs:field xpath="Composition_type_id" />
   </xs:keyref>
   <xs:keyref name="FK_SD_function_SD_function_place"
refer="SD_function">
     <xs:selector xpath=".//SD_function_place" />
     <xs:field xpath="SD_function_id" />
   </xs:keyref>
   <xs:keyref name="FK_type_place" refer="SD_function_color">
     <xs:selector xpath=".//SD_function_place" />
     <xs:field xpath="color_id" />
   </xs:keyref>
   <xs:keyref name="FK_SD_function_SD_function_transition"
refer="SD_function">
     <xs:selector xpath=".//SD_function_transition" />
     <xs:field xpath="SD_function_id" />
   </xs:keyref>
   <xs:keyref name="CompositionType_SD_functionct"
refer="SD_function_type">
     <xs:selector xpath=".//SD_function" />
     <xs:field xpath="SD_function_type_id" />
   </xs:keyref>
  </xs:element>
</xs:schema>
```

Appendix C-*Questionnaire results*

Number of participants: 76

Tools evaluated: X-Man, YahooPipes and IBM Damia

Table 1: Questionnaire feedback results

	X-Man			YahooPipes			IBM Damia		
Questions	**+**	**0**	**-**	**+**	**0**	**-**	**+**	**0**	**-**
1. It is easy to specify/define input data	69	6	1	57	4	15	62	9	5
2. It is easy to specify/define output data	70	4	2	54	6	16	66	8	2
3. Input data is adequate for XML	70	5	1	59	12	5	65	8	3
4. Output data is adequate for XML	70	4	2	59	13	4	62	9	5
5. It is easy to understand the required parameters	62	10	4	54	15	7	57	8	11
6. It is easy to specify/define the required parameters	64	10	2	49	15	12	54	13	9
7. Functions are clearly visualized	70	4	2	56	11	9	66	9	1
8. It is easy to understand the functionality of a Function	66	7	3	63	6	7	55	15	6
9. The I/O of a function are clearly defined/visualized	64	5	7	62	7	7	57	15	4
10. Data types of functions' I/O are clearly defined/visualized	60	11	5	55	18	3	53	18	5
11. It is easy to differentiate between different data types	54	12	10	61	12	3	53	14	9
12. It is easy to visualize/understand XML I/O structures	66	8	2	60	9	7	59	9	8
13. It is easy to differentiate between the types of different XML nodes (Element/Attribute/Text)	68	5	3	63	8	5	59	13	4
14. It is easy to create compositions	61	12	3	49	16	11	51	19	6
15. The language is easy and simple to learn	62	10	4	50	14	12	57	13	6
16. The language is easy to use	62	7	7	48	10	18	60	12	4
17. The Composition does not require lot of steps	54	4	18	52	9	15	51	18	7
18. The composition is accurate	61	10	5	56	16	4	59	15	2
19. Conversion between different data types is simple	57	15	4	50	21	5	54	18	4
20. Many data types can be used (String/Integer/Boolean...)	61	12	3	59	13	4	57	15	4
21. I/O XML mapping with a composition is simple/easy	66	9	1	48	17	11	64	7	5
22. It is easy to modify the output XML structure	62	10	4	49	14	13	55	11	10
23. It is easy to create a new output XML structure from scratch	64	9	3	52	20	4	50	18	8
24. Little Programming knowledge is required.	47	11	18	49	18	9	48	12	16
25. Little XML knowledge is required	52	9	15	44	19	13	42	18	16

Abstract. Computers and the Internet are everywhere nowadays, in every home, domain and field. Communications between users, applications and heterogeneous information systems are mainly done via XML structured data. XML, based on simple textual data and not requiring any specific platform or environment, has invaded and governed the communication Medias. In the 21st century, these communications are now inter-domain and have stepped outside the scope of computer science into other areas (i.e., medical, commerce, social, etc.). As a consequence, and due to the increasing amount of XML data floating between non-expert users (programmers, scientists, etc.), whether on instant messaging, social networks, data storage and others, it is becoming crucial and imperative to allow non-experts to be able to manipulate and control their data (e.g.,parents who want to apply parental control over instant messaging tools in their house, a journalist who wants to gather information from different RSS feeds and filter them out, etc.). The main objective of this work is the study of XML manipulations by non-expert users. Four main related categories have been identified in the literature: XML-oriented visual languages, Mashups, XML manipulation by security and adaptation techniques, and Dataflow visual programming languages. However, none of them provides a full-fledged solution for appropriate XML data manipulation. In our research, we formally defined an XML manipulation framework, entitled XA2C (XML Alteration/Adaptation Composition Framework). XA2C represents a visual studio for an XML-oriented DFVPL (Dataflow Visual Programming Language), called XCDL (XML-oriented Composition Definition Language) which constitutes the major contribution of this study. XCDL is based on Colored Petri Nets allowing non-expert users to compose manipulation operations. The XML manipulations range from simple data selection/projection to data modification (insertion, removal, obfuscation, etc.). The language is oriented to deal with XML data (XML documents and fragments), providing users with means to compose XML oriented operations. Complementary to the language syntax and semantics, XA2C formally defines also the compiler and runtime environment of XCDL. In addition to the theoretical contribution, we developed a prototype, called *X-Man*, and formally defined an evaluation framework for XML-oriented visual languages and tools that was used in a set of case studies and experiments to evaluate the quality of use of our language and compare it to existing approaches. The obtained assessments and results were positive and show that our approach outperforms existing ones. Several future tracks are being studied such as integration of more complex operations (control operators, loops, etc.), automated compositions, and language derivation to define specific languages oriented towards different XML-based standards (e.g., RSS, RDF, SMIL, etc.)

Keywords: XML, XML manipulation, XML control, Visual languages, Dataflow, Mashups, Petri Nets, Visual syntax, Language semantics and syntax, Functional Composition, Concurrency and Parallelism.

Résume. Aujourd'hui, les ordinateurs et l'Internet sont partout dans le monde : dans chaque maison, domaine et plateforme. Dans ce contexte, le standard XML s'est établi comme un moyen insigne pour la représentation et l'échange efficaces des données. Les communications et les échanges d'informations entre utilisateurs, applications et systèmes d'information hétérogènes sont désormais réalisés moyennant XML afin de garantir l'interopérabilité des données. Le codage simple et robuste de XML, à base de données textuelles semi-structurées, a fait que ce standard a rapidement envahi les communications medias. Ces communications sont devenues inter-domaines, partant de l'informatique et s'intégrant dans les domaines médical, commercial, et social, etc. Par conséquent, et au vu du niveau croissant des données XML flottantes entre des utilisateurs non-experts (employés, scientifiques, etc.), que ce soit sur les messageries instantanées, réseaux sociaux, stockage de données ou autres, il devient incontournable de permettre aux utilisateurs non-experts de manipuler et contrôler leurs données (e.g., des parents qui souhaitent appliquer du contrôle parental sur les messageries instantanées de leur maison, un journaliste qui désire regrouper et filtrer des informations provenant de différents flux RSS, etc.). L'objectif principal de cette thèse est l'étude des manipulations des données XML par des utilisateurs non-experts. Quatre principales catégories ont été identifiées dans la littérature : i) les langages visuels orientés XML, ii) les Mashups, iii) les techniques de manipulation des données XML, et iv) les DFVPL (langages de programmation visuel à base de Dataflow), couvrant différentes pistes. Cependant, aucune d'entre elles ne fournit une solution complète. Dans ce travail de recherche, nous avons formellement défini un Framework de manipulation XML, intitulé XA2C (XML-oriented mAnipulAtion Compositions). XA2C représente un environnement de programmation visuel (e.g., Visual-Studio) pour un DFVPL orienté XML, intitulé XCDL (XML-oriented Composition Definition Language) qui constitue la contribution majeure de cette thèse. XCDL, basé sur les réseaux de Pétri colorés, permet aux non-experts de définir, d'arranger et de composer des opérations de manipulation orientées XML. Ces opérations peuvent être des simples sélections/projections de données, ainsi que des opérations plus complexes de modifications de données (insertion, suppression, tatouage, etc.). Le langage proposé traite les données XML à base de documents ou de fragments. En plus de la définition formelle (syntaxique et sémantique) du langage XCDL, XA2C introduit une architecture complète à base d'un compilateur et un environnement d'exécution dédiés. Afin de tester et d'évaluer notre approche théorique, nous avons développé un prototype, intitulé *X-Man*, avec un Framework d'évaluation pour les langages et outils visuels de programmation orientés XML. Une série d'études de cas et d'expérimentations a été réalisée afin d'évaluer la qualité d'usage de notre langage, et de le comparer aux solutions existantes. Les résultats obtenus soulignent la supériorité de note approche, notamment en termes de qualité d'interaction, de visualisation, et d'utilisation. Plusieurs pistes sont en cours d'exploration, telles que l'intégration des opérations plus complexes (opérateurs de contrôle, boucles, etc.), les compositions automatiques, et l'extension du langage pour gérer la spécificité des formats dérivés du standard XML (flux RSS, RDF, SMIL, etc.).

Mots Clés: XML, Manipulation de données XML, Contrôle de données XML, Langages visuelles, Dataflow, Mashups, Réseaux de Pétri, Syntaxe visuelle, Langage sémantique and syntaxique, Composition, Concurrence et Parallélisme.