Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Tomasz Kokoszka
Nr albumu: 189405

Sprawdzanie poprawności logicznej dokumentów XML

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Wrzesień 2006

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza praca jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Streszczenie

Budowa dokumentu XML jest ściśle określona przez konsorcjum W3C. Rekomendacja XML-Schema pozwala dodatkowo narzucić pewne proste warunki na zawartość dokumentu, np. więzy unikatowe i referencyjne, układ elementów w hierarchii itp.

Okazuje się, że taki ograniczony zakres warunków nie wystarcza w wielu konkretnych zastosowaniach. Dokumenty XML są często wykorzystywane do generowania innych dokumentów (np. PDF) lub kodu źródłowego (np. C++). W takich przypadkach zawartość dokumentu XML musi spełniać ściśle określone reguły, w przeciwnym razie wygenerowany kod nie będzie zgodny z oczekiwanym formatem wyjściowym.

Istniejące rekomendacje nie pozwalają na definiowanie złożonych warunków. Już nawet tak pozornie prosta przykładowa reguła mówiąca: „każdy klucz w więzach referencyjnych A musi być użyty dokładnie raz jako referencja" wykracza poza możliwości oferowane przez XML-Schema.

W pracy prezentuję rozwiązanie tego problemu. Definiuję cel, który chcę osiągnąć. Przedstawiam analizę możliwych rozwiązań. Opisuję implementację wybranego na etapie analizy rozwiązania.

Słowa kluczowe

XML, XML-Schema, CLiXML, CLiX, Schematron, sprawdzanie poprawności, poprawność logiczna

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.2. Software Engineering
D.2.4. Software/Program Verification – Assertion checkers, Validation

Tytuł pracy w języku angielskim:

Logical validation of XML documents

Spis treści

Wprowadzenie **7**

1. Definicja problemu **9**

1.1. Podstawowe pojęcia 9
1.2. Format XML 11
1.3. XML-Schema 12
1.4. Definicja problemu 14
1.5. Definicja wymagań 16
1.5.1. Szeroki zakres definiowania warunków 17
1.5.2. Czytelność i prostota definiowanych warunków 17
1.5.3. Przezroczystość 17
1.5.4. Współpraca z XML-Schema 17
1.5.5. Definiowanie informacji o błędach 17
1.5.6. Oparcie na istniejących standardach 17
1.5.7. Oparcie na istniejących implementacjach 18
1.5.8. Interfejs programisty dla języka Java 18

2. Analiza możliwych rozwiązań, wybór optymalnego rozwiązania **19**

2.1. Reguły jako klasy języka Java 20
2.2. Schematron 22
2.3. CLiXML 24
2.4. Podsumowanie, wybór rozwiązania 27

3. XDV – moduł sprawdzania poprawności dokumentów XML **29**

3.1. Moduł XPath 29
3.2. API do sprawdzania poprawności dokumentów XML 31
3.3. Implementacja języka CLiXML 32
3.3.1. Wykonywanie reguł 32
3.3.2. Rozszerzenie – wznawianie wykonania 33
3.3.3. Rozszerzenie – komunikat błędu 34
3.3.4. Rozszerzenie – wykonywanie makr 35
3.3.5. Rozszerzenie – definiowanie nowych operatorów 35
3.3.6. Zgodność ze specyfikacją, ograniczenia 36
3.4. Podsumowanie 36

4. Studium przypadku – projekt FESA **37**

4.1. Projekt FESA 37
4.2. Pierwsze podejście do problemu sprawdzania poprawności dokumentów XML 39

4.3. Zastosowanie modułu XDV ... 39
4.4. Podsumowanie ... 40

5. Podsumowanie ... 43

Dodatek A: Opis zawartości dołączonej płyty CD ... 45

Bibliografia ... 47

Spis rysunków

Rysunek 1. XPath – przykład 10
Rysunek 2. XSLT – przykładowa transformacja..11
Rysunek 3. XML – przykładowy dokument...12
Rysunek 4. XML-Schema – przykładowa definicja schematu...14
Rysunek 5. Warunek dotyczący struktury danych – przykład 15
Rysunek 6. Warunek dotyczący zawartości (warunek logiczny) – przykład.........................15
Rysunek 7. Przykładowy pojedynczy warunek logiczny dotyczący zawartości dokumentu XML 19
Rysunek 8. Interfejs opisujący regułę w języku Java oraz obiekt z opisem błędu 20
Rysunek 9. Reguła opisana na rysunku 7 wyrażona w języku Java 21
Rysunek 10. Reguła opisana na rysunku 7 wyrażona w języku Schematron 23
Rysunek 11. Reguła opisana na rysunku 7 wyrażona w języku CLiXML 26
Rysunek 12. Zależności pomiędzy częściami składowymi modułu XDV.............................29
Rysunek 13. Przykład użycia modułu XPath – diagram sekwencji 31
Rysunek 14. Reprezentacja w pamięci przykładowej reguły CLiXML 33
Rysunek 15. Wznawianie wykonania formuł – definicja reguły i przykładowy dokument z zaznaczonymi błędami, które zostaną zasygnalizowane użytkownikowi 34
Rysunek 16. Rozszerzony komunikat błędu – przykład 35
Rysunek 17. Projekt FESA – schemat działania, zastosowanie technologii XML................38
Rysunek 18. FESA-Shell – ogólny graficzny edytor XML sterowany przez XML-Schema .38
Rysunek 19. Porównanie wydajności i złożoności dwóch metod sprawdzania poprawności logicznej dokumentów XML – języka Java oraz CLiXML (implementacja XDV) – w oparciu o wyniku otrzymane dla aplikacji FESA-Shell...............................41

Wprowadzenie

Dokumenty XML przechowują coraz bardziej złożone informacje. Informacje te są często przekształcane do innych postaci. Mogą być, zależnie od zastosowań, wykorzystane do wygenerowania dokumentacji (np. PDF), diagramów (np. SVG) lub kodu źródłowego w zadanym języku programowania (np. C++). W takich wypadkach niezwykle ważne jest, aby dokument XML spełniał określone reguły spójności. W przeciwnym razie wygenerowany kod nie będzie zgodny z postawionymi wymaganiami. Dokumentacja nie będzie poprawnym dokumentem PDF, diagram będzie niespójny, kod źródłowy nie będzie się poprawnie kompilował w używanym przez nas kompilatorze lub nie będzie działał zgodnie z intencjami autora.

Budowa dokumentu XML jest ściśle określona przez konsorcjum W3C. Rekomendacje i standardy, takie jak XML-Schema, pozwalają narzucić pewne dodatkowe, proste warunki na zawartość dokumentu. Przykładowo pozwalają określić hierarchię elementów, typy wartości, więzy unikatowe, referencyjne itp.

W wielu przypadkach okazuje się, że to zdecydowanie za mało. Istniejące rekomendacje nie wystarczają do zdefiniowania warunków potrzebnych w konkretnych zastosowaniach. Dobrym przykładem jest projekt FESA (ang. *Front End System Architecture*) realizowany w Europejskiej Organizacji Badań Jądrowych CERN w Genewie. Celem projektu jest dostarczenie architektury do programowania specjalizowanych komputerów, które sterują pracą urządzeń podłączonych do akceleratorów cząstek w CERN. FESA definiuje kompletny proces od zaprojektowania do zaimplementowania i uruchomienia programu sterującego. Dostarcza narzędzia, które wspomagają każdy etap pracy. Użytkownik tworzy tzw. klasę FESA, która jest projektem programu sterującego wyrażonym w XML. Z tego opisu jest automatycznie generowany szkielet programu w języku C++. Dostarcza on funkcjonalność wymaganą do komunikacji z centrum kontrolnym CERN, obsługi alarmów itp. Użytkownik musi uzupełnić szkielet kodu jedynie o fragmenty specyficzne dla urządzenia, którym chce sterować. Aby móc wygenerować prawidłowy kod C++, źródłowy dokument XML musi spełniać ściśle określone reguły poprawności. Narzędzia FESA używają XML-Schema do sprawdzania poprawności XML. Jednak bardzo szybko osiągnięto kres możliwości tego rozwiązania. Już nawet prosta reguła mówiąca: „każdy klucz w więzach referencyjnych A musi być użyty dokładnie raz jako referencja” wykracza poza funkcjonalność XML-Schema.

W pracy prezentuję sposób na wypełnienie luki między możliwościami XML-Schema a rzeczywistymi potrzebami użytkowników. Specyfikuję listę wymagań (rozdział 1), przeprowadzam analizę możliwych rozwiązań (rozdział 2). Wynikiem prac jest program w języku Java – ogólny pakiet do sprawdzania poprawności dokumentów XML, który między innymi implementuje wybrany, optymalny model (rozdział 3). Ostatnia część pracy to studium przypadku (rozdział 4). Opisuję w niej jak mój program sprawdził się w projekcie FESA, jak

wpasowała się w istniejącą architekturę. Porównuję wyniki uzyskane po zastosowaniu mojego programu z poprzednim, doraźnym rozwiązaniem (stworzonym „ad-hoc”).

1. Definicja problemu

W rozdziale przedstawiam szczegółowo problem sprawdzania poprawności logicznej dokumentów XML. Wyjaśniam podstawowe pojęcia używane w dalszej części pracy. Opisuję rekomendację XML-Schema służącą do sprawdzania dokumentów XML, jej główne cechy i ograniczenia. Na końcu określam listę wymagań, które musi spełniać poszukiwany model do definiowania i sprawdzania poprawności logicznej dokumentów XML.

1.1. Podstawowe pojęcia

Definicje pojęć używanych w pracy w kolejności alfabetycznej:

API (ang. *Application Programming Interface*)
Interfejs programowania aplikacji. Specyfikacja procedur, funkcji lub interfejsów umożliwiających komunikację z biblioteką, systemem operacyjnym lub innym systemem zewnętrznym w stosunku do aplikacji korzystającej z API.

CERN (fr. Conseil Européen pour la Recherche Nucléaire)
Europejski Ośrodek Badań Jądrowych, ośrodek naukowo-badawczy położony w Szwajcarii. Głównym zadaniem ośrodka jest dostarczanie narzędzi dla fizyków do prowadzenia badań. W tym celu stworzono szereg detektorów i akceleratorów cząstek, w tym największy na świecie akcelerator kołowy o średnicy 27 km. CERN jest uznawany za kolebkę WWW. W 1989 roku pracując nad usprawnieniem wymiany informacji pomiędzy badaczami opracowującymi wyniki eksperymentów stworzono język HTML.

DTD (ang. *Document Type Definition*)
Definicja typu dokumentu, pozwala zdefiniować formalną strukturę dokumentu zapisanego w HTML, XML, XHTML, SGML itp. W przypadku XML obecnie wypierana przez XML-Schema, która oferuje więcej możliwości.

FESA (ang. *Front End Software Architecture*)
Rozwiązanie dostarczające wspólnej architektury oraz metodologii do programowania specjalizowanych komputerów (ang. *front-end computers*), które sterują pracą urządzeń podłączonych do akceleratorów cząstek w CERN. FESA definiuje kompletny proces od projektowania, przez implementację i uruchomienie, do monitorowania działających programów. Projekt bazuje na technologii XML. Powstał w CERN i jest realizowany siłami organizacji (por. [15]).

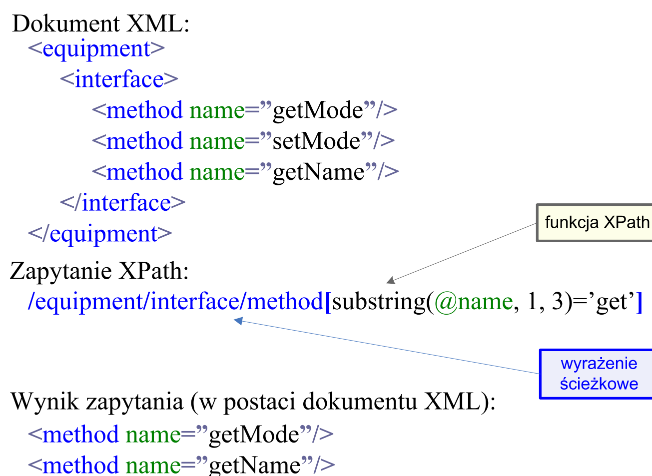
Przestrzeń nazw XML (ang. *XML namespace*)
Mechanizm stworzony przez W3C zapobiegający konfliktom nazw elementów i atrybutów zagnieżdżonych w jednym dokumencie XML. Przestrzeń nazw jest uzupełnieniem nazwy elementu. Jest to URI, ale nie musi wskazywać istniejącego zasobu (por. [2]).

SGML (ang. *Standard Generalized Markup Language*)
Nadrzędny język znaczników służący do ujednolicania struktury i formatu wszelkich informacji dających się zapisać w formie dokumentu tekstowego. W odróżnieniu od języków znaczników przeznaczonych do konkretnych zastosowań (np. HTML), SGML nie jest zbiorem określonych znaczników i reguł ich użytkowania, lecz ogólnym, nadrzędnym językiem służącym do definiowania dowolnych znaczników i ustalania zasad ich poprawnego użytkowania. Standard ISO8879.

URI (ang. *Uniform Resource Identifier*)
Standard internetowy określony w RFC2396 umożliwiający łatwą identyfikację zasobów w sieci. URI jest łańcuchem znaków zapisanym zgodnie z określoną w standardzie składnią. Łańcuch ten określa nazwę lub adres zasobu.
Przykładowy URI: `http://www.w3.org/2001/XMLSchema`

W3C (ang. *WWW Consortium*)
Organizacja zajmująca się rozwojem WWW. Powstała w 1994 roku z inicjatywy Massachusetts Institute of Technology oraz Europejskiej Organizacji Badań Jądrowych CERN. Wydaje rekomendacje jak technologie WWW powinny być używane, lecz nie ma oficjalnej władzy by domagać się ich przestrzegania. Obecnie 80% działań organizacji dotyczy XML.

XPath
Skrót od angielskiego *XML Path Language*. Język pozwalający na wskazanie, odwołanie się i dostęp do fragmentów dokumentu, elementów lub atrybutów dokumentów XML. Rekomendacja konsorcjum W3C (por. [3]). Wersja 1.0 została opracowana w roku 1999, obecnie trwają prace na wersją 2.0. Rysunek 1 pokazuje przykład użycia XPath.



Rysunek 1. XPath – przykład

XSLT (ang. eX*tensible Stylesheet Language Transformations*)

Język transformacji dokumentów XML. Pozwala na przetłumaczenie dokumentów XML do innego schematu XML, jak również na HTML, PDF i inne. Wersja 1.0 rekomendacji została opublikowana przez W3C w roku 1999 (por. [4]). Rysunek 2 pokazuje przykładową transformację XSLT.

Wejściowy dokument XML:

```
<equipment>
  <interface>
    <method name=”getMode”/>
    <method name=”setMode”/>
    <method name=”getName”/>
  </interface>
</equipment>
```

Transformacja XSLT:

```
<xsl:stylesheet xmlns:xsl=”http://www.w3.org/1999/XSL/Transform” version=”1.0”>
  <xsl:output method=”html”/>
  <xsl:template match=”equipment”>
    <html><body>
      Methods:
      <ol>
        <xsl:for-each select=”interface/method”>
          <li>
            <xsl:value-of select=”@name”/>
          </li>
        </xsl:for-each>
      </ol>
    </body></html>
  </xsl:template>
</xsl:stylesheet>
```

instrukcja XSLT (przedrostek xsl)

zapytanie XPath

Wynik transformacji (dokument HTML):

```
<html><body>
  Methods:
  <ol>
    <li>getMode</li>
    <li>setMode</li>
    <li>getName</li>
  </ol>
</body></html>
```

Rysunek 2. XSLT – przykładowa transformacja

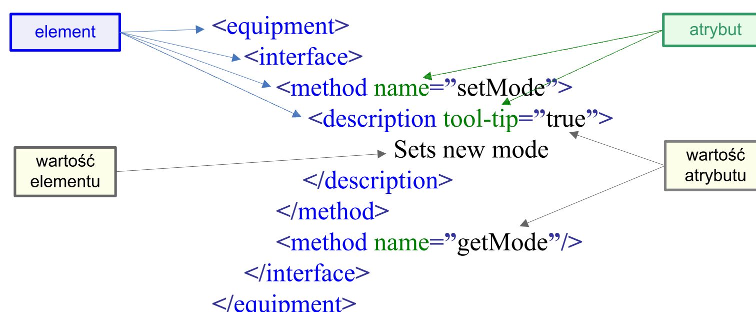
1.2. Format XML

XML (ang. *eXtensible Markup Language*) to prosty, elastyczny format tekstowy wywodzący się ze standardu SGML. Pierwsza idea XML zrodziła się w roku 1994 podczas drugiej konferencji WWW. W 1996 roku prace nad XML przejęło konsorcjum W3C, a dwa lata później

stał się oficjalną rekomendacją. Pierwotnie został opracowany do tworzenia publikacji elektronicznych, jako rozwiązanie pośrednie pomiędzy SGML (skompilowany, ale elastyczny) a HTML (prosty, o ograniczonym zastosowaniu). Obecnie odgrywa coraz większą rolę przy przechowywaniu informacji i przesyłaniu danych pomiędzy systemami (por. [1]).

Informacje w dokumencie XML mają określoną strukturę i postać. Informacja jest dzielona na jasno wyodrębnione sekcje, które zawierają konkretne dane. Jest to podział ze względu na treść informacji.

Sam dokument XML to zbiór elementów tworzących określoną hierarchię. Elementy mogą przechowywać kolejne elementy, dodatkowe atrybuty oraz wartości (por. rys. 3).



Rysunek 3. XML – przykładowy dokument

W ostatnich latach XML staje się coraz bardziej popularny. Wiele nowych technologii używa dokumentów XML do przechowywania informacji.

1.3. XML-Schema

XML-Schema [5] to język do definiowania struktury (schematu) dokumentu XML. Definicja języka została oficjalnie opublikowana przez konsorcjum W3C jako rekomendacja w roku 2001. Wcześniej wielu producentów dostarczało własne rozwiązania tego typu, np (na podstawie [6]):

1. XML-Data, XML-Data Reduced – opracowany przez firmę Microsoft, opublikowany w 1998 roku.

2. SOX (ang. *Schema for Object-Oriented XML*) – opracowany przez firmę ComerceOne w 1998 roku.

3. DDML (ang. *Document Definition Markup Language or XSchema*) – praca zbiorowa osób z grupy dyskusyjnej XML-dev, opracowana w 1999 roku.

4. RELAX NG – specyfikacja organizacji OASIS, pierwsza wersja pochodzi z roku 2000, standard ISO.

Wszystkie wymienione pomysły oferują jednakowe lub zbliżone możliwości, ale w innej formie. Od chwili ogłoszenia XML-Schema jako oficjalnej rekomendacji przez konsorcjum

W3C pozostałe rozwiązania systematycznie traciły na znaczeniu i popularności. Obecnie praktycznie nie są stosowane.

XML-Schema to zestaw reguł, które musi spełniać dokument XML, aby był uznany za zgodny z danym schematem. Przy pomocy XML-Schema można opisać i zdefiniować (por. rys. 4 oraz opracowanie [5]):

1. Przestrzeń nazw – możliwe nazwy elementów i atrybutów.

2. Hierarchie – sposób organizacji elementów i atrybutów wewnątrz dokumentu, kolejność, liczba wystąpień itp.

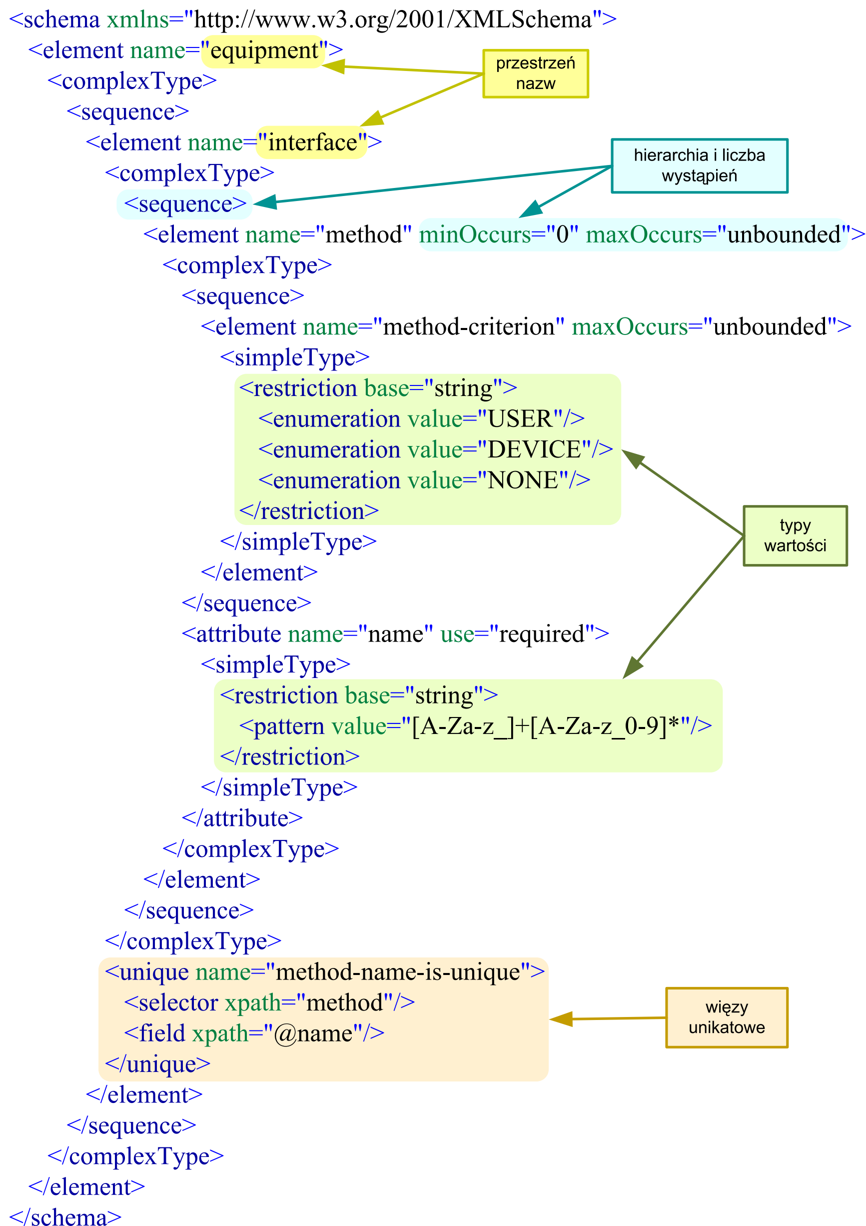
3. Typy wartości elementów i atrybutów – rekomendacja oferuje bogaty zestaw typów podstawowych, między innymi: liczby całkowite i rzeczywiste o różnej precyzji, daty, napisy. Każdy z typów prostych można modyfikować przez podanie zakresu dopuszczalnych wartości (dla liczb), wzorce dopasowania, listę wartości (enumeracje) itp.

4. Więzy unikatowe – analogicznie jak w języku SQL; gwarantują niepowtarzalność wartości. Innymi słowy: nie może pojawić się inny element w tym samym miejscu hierarchii z taką samą wartością.

5. Więzy referencyjne – analogicznie jak w języku SQL; określają powiązania (referencje) pomiędzy wartościami. Innymi słowy: wartość musi być taka sama jak wartość innego elementu/atrybutu.

6. Dodatkowe informacje specyficzne dla narzędzia, które będzie interpretować i używać tego schematu. Może to być dowolna informacja zapisana w postaci tekstowej (również dokument XML).

Sam opis schematu jest dokumentem XML. Dokumenty zawierające definicje XML-Schema zapisuje się zwyczajowo w plikach z rozszerzeniem `xsd` (od ang. *XML Schema Definition*).



```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="equipment">
    <complexType>
      <sequence>
        <element name="interface">
          <complexType>
            <sequence>
              <element name="method" minOccurs="0" maxOccurs="unbounded">
                <complexType>
                  <sequence>
                    <element name="method-criterion" maxOccurs="unbounded">
                      <simpleType>
                        <restriction base="string">
                          <enumeration value="USER"/>
                          <enumeration value="DEVICE"/>
                          <enumeration value="NONE"/>
                        </restriction>
                      </simpleType>
                    </element>
                  </sequence>
                  <attribute name="name" use="required">
                    <simpleType>
                      <restriction base="string">
                        <pattern value="[A-Za-z_]+[A-Za-z_0-9]*"/>
                      </restriction>
                    </simpleType>
                  </attribute>
                </complexType>
              </element>
            </sequence>
          </complexType>
          <unique name="method-name-is-unique">
            <selector xpath="method"/>
            <field xpath="@name"/>
          </unique>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>
```


Rysunek 4. XML-Schema – przykładowa definicja schematu

1.4. Definicja problemu

Dokument XML przechowuje informacje w usystematyzowanej formie. Z założenia może to być dowolna informacja w dowolnej postaci, która nie jest narzucona z góry.

Co należy zrobić, jeśli dodatkowo chcemy wprowadzić pewne warunki dotyczące zawartości? Taka sytuacja pojawia się bardzo często. Jeśli dla przykładu aplikacja odczytuje konfigurację z pliku XML, to oczekuje, że informacje będą dostarczone w zadanej kolejności i będą zachowane określone warunki spójności.

Warunki spójności możemy podzielić na dwie grupy:

1. Warunki dotyczące struktury danych

Określają nazwy elementów i atrybutów, sposób ich organizacji, hierarchię, typy danych itp. (por. rys. 5).

Element equipment jest elementem głównym, musi występować.
Zawiera dokładnie jedno dziecko – element interface, który z kolei może składać się z dowolnej liczby elementów method.
Element method musi zawierać niepusty atrybut name typu tekstowego.
Element method może zawierać jeden element description typu tekstowego.

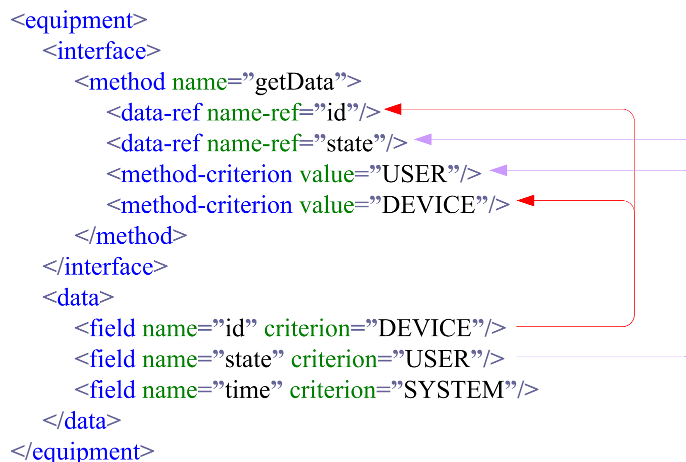
```
<equipment>
    <interface>
        <method name=”setMode”>
            <description>Sets new mode</description>
        </method>
        <method name=”getMode”/>
    </interface>
</equipment>
```

Rysunek 5. Warunek dotyczący struktury danych – przykład

2. Warunki dotyczące zawartości – warunki logiczne

Określają zależności logiczne pomiędzy elementami, atrybutami i wartościami w dokumencie. Warunki tego typu mogą być bardzo rozbudowane (por. rys. 6).

Nazwa metody interfejsu nie może się powtarzać (element method, atrybut name).
Nazwa pola danych nie może się powtarzać (element field, atrybut name).
Jeśli metoda wskazuje na pole danych (element data-ref), to atrybut name-ref musi zawierać nazwę istniejącego pola danych.
Jeśli metoda interfejsu wskazuje na pola danych, to zbiór wartości *kryteriów* tej metody (elementy method-criterion, atrybut value) musi być równy zbiorowi wartości *kryteriów* wskazywanych pól (wskazywane elementy field, atrybut criterion).



```
<equipment>
    <interface>
        <method name=”getData”>
            <data-ref name-ref=”id”/>
            <data-ref name-ref=”state”/>
            <method-criterion value=”USER”/>
            <method-criterion value=”DEVICE”/>
        </method>
    </interface>
    <data>
        <field name=”id” criterion=”DEVICE”/>
        <field name=”state” criterion=”USER”/>
        <field name=”time” criterion=”SYSTEM”/>
    </data>
</equipment>
```

Rysunek 6. Warunek dotyczący zawartości (warunek logiczny) – przykład

Do określania warunków pierwszego typu można użyć standardu XML-Schema. Pozwala on zdefiniować dowolną hierarchię i ma bardzo bogaty zestaw typów. Jeśli chodzi o drugi typ warunków, to sytuacja jest o wiele bardziej skomplikowana. XML-Schema pozwala definiować tylko więzy referencyjne oraz klucze unikatowe. Jest to niewielki podzbiór warunków logicznych.

Czy jednak taka funkcjonalność jest potrzebna?

CERN od ponad 2 lat z powodzeniem realizuje projekt FESA. Wszystkie informacje w projekcie są przechowywane i przesyłane jako dokumenty XML. Z dokumentów XML jest generowany szkielet kodu w języku C++. Oczywiście, jeśli zawartość dokumentu XML jest niewłaściwa lub niespójna, to wygenerowany kod C++ jest nieprawidłowy – nie kompiluje się lub, co gorsza kompiluje, lecz zawiera błędy logiczne, które powodują, że wygenerowany kod nie działa zgodnie z intencją autora.

W projekcie FESA do sprawdzania poprawności jest używany XML-Schema. Niestety bardzo szybko osiągnięto kres możliwości tej technologii – znacznej części warunków po prostu nie udało się wyrazić za pomocą XML-Schema.

Rekomendacja XML została oficjalnie ogłoszona w 1998 roku, jest to stosunkowo nowy pomysł. Jednocześnie format jest na tyle elastyczny, że bardzo szybko zyskał dużą popularność i nowe zastosowania. Nie spełnia on jednak wszystkich oczekiwań – nadal brakuje modułu do definiowania i sprawdzania warunków logicznych dotyczących zawartości dokumentu.

W tym momencie nasuwa się porównanie do baz danych. Dokument XML oraz baza danych służą do przechowywania informacji. W stanie surowym oba są białą płachtą, którą użytkownik zapełnia danymi. Aby jednak dane mogły być ponownie odczytane i użyte musimy narzucić im pewną ustaloną postać. W przypadku XML robimy to definiując elementy, atrybuty, hierarchię oraz typy wartości. W bazach danych tworzymy tabele, kolumny nadajemy określone typy danych. W obu przypadkach dysponujemy dobrymi narzędziami do realizacji tego celu: XML-Schema dla XML oraz SQL DDL dla baz danych.

Jeśli chcemy zdefiniować złożone warunki logiczne, które utrzymają spójność danych, to musimy sięgnąć znacznie głębiej. W przypadku baz danych mamy tak zwane wyzwalacze, których oprogramowanie wykracza poza standard SQL. Producenci dostarczają rozwiązania przeznaczone dla konkretnego systemu baz danych, np. PL/SQL dla baz danych firmy Oracle, PG/SQL dla bazy danych PostgreSQL.

A co w przypadku XML? Tutaj nie ma ustalonej metody na rozwiązanie tego problemu.

1.5. Definicja wymagań

W poprzednich podrozdziałach (1.1–1.4) zdefiniowałem problem i niezbędne pojęcia. Określiłem czego brakuje w ogólnie używanych i przyjętych rekomendacjach (XML-Schema). Ostatecznie stwierdziłem, że potrzebujemy modułu do definiowania i sprawdzania poprawności logicznej dokumentów XML. Nadal jest to ogólne stwierdzenie. W tym podrozdziale precyzuję moje wymagania.

Zamieszczone zestawienie (punkty 1.5.1–1.5.8) to szczegółowa lista wymagań, które powinien spełniać moduł do definiowania i sprawdzania poprawności logicznej dokumentów XML. Część z nich to bardziej zalecenia, które pozwolą przedłużyć czas życia i używania modułu. Lista uwzględnia spostrzeżenia i wnioski z prac nad projektem FESA. Posiadanie takiej listy wymagań pomoże mi w wyborze optymalnego rozwiązania.

1.5.1. Szeroki zakres definiowania warunków

Moduł nie może z góry ograniczać klas warunków, tak jak ma to miejsce w przypadku XML-Schema (tylko więzy unikatowe i referencyjne). Musi to być niezależny mechanizm do opisu dowolnych warunków logicznych.

1.5.2. Czytelność i prostota definiowanych warunków

Aby ułatwić proces tworzenia, utrzymywania i rozbudowy zestawu reguł, sposób wyrażania warunków logicznych powinien być w miarę możliwości prosty, czytelny i przejrzysty.

1.5.3. Przezroczystość

Zastosowanie modułu musi być przezroczyste. Nie może wymagać wprowadzania zmian w dokumencie XML.

1.5.4. Współpraca z XML-Schema

Moduł nie musi, a nawet nie powinien powielać funkcjonalności, którą daje XML-Schema. Ma być uzupełnieniem XML-Schema – tak jak PL/SQL uzupełnia standard SQL w przypadku baz danych. Dodatkową zaletą będzie możliwość osadzania warunków poprawności lub referencji do nich w tym samym dokumencie, co opis schematu.

1.5.5. Definiowanie informacji o błędach

W razie niespełnienia określonego warunku logicznego musi istnieć możliwość wyspecyfikowania:

(a) poziomu błędu – wymagane jest rozróżnianie przynajmniej dwóch poziomów: **błąd** (ang. *error*) i **ostrzeżenie** (ang. *warning*),

(b) komunikatu błędu – komunikat może być generowany dynamicznie i może zawierać informacje pobrane bezpośrednio ze sprawdzanego dokumentu,

(c) referencji do elementów – informacja opcjonalna, zestaw wskaźników do elementów dokumentu XML. Dla przykładu mogą to być referencje do obiektów, na które należy „zwrócić uwagę” przy poprawianiu dokumentu.

1.5.6. Oparcie na istniejących standardach

Moduł powinien się opierać – w miarę możliwości – na istniejących standardach i rekomendacjach. Dzięki temu rozwiązanie będzie otwarte i łatwiejsze do przyswojenia i opanowania przez nowych użytkowników.

1.5.7. Oparcie na istniejących implementacjach

Rozwiązanie powinno opierać się na istniejących implementacjach. Dzięki temu pojawienie się nowej funkcjonalności w jednej z używanych implementacji może automatycznie podnieść funkcjonalność całego modułu.

1.5.8. Interfejs programisty dla języka Java

Musi być dostarczony interfejs programisty (API) dla języka Java w wersji 1.4.x oraz 1.5.x. Dodatkowe wymagania dotyczące interfejsu:

(a) dane wejściowe – dokument XML w postaci obiektu języka Java typu `org.w3c.dom.Document`.

(b) dane wyjściowe – pełna informacja o błędach, łącznie z referencją do elementów (obiekt języka Java typu `org.w3c.dom.Node`).

2. Analiza możliwych rozwiązań, wybór optymalnego rozwiązania

W rozdziale analizuję możliwe sposoby definiowania i sprawdzania poprawności logicznej dokumentów XML. Opis każdego z nich obejmuje:

1. Sposób definiowania reguł wraz z przykładową definicją dla warunku opisanego na rysunku 7.
2. Wady i zalety z uwzględnieniem wymagań opisanych w rozdziale 1.5.

W ostatnim podrozdziale podsumowuję wyniki analizy i wybieram optymalne rozwiązanie w celu późniejszej implementacji.

Dla każdej metody (/equipment/interface/method) zbiór wartości *kryteriów* metody (atrybut value elementów method-criterion) musi być równy zbiorowi wartości *kryteriów* wskazywanych pól danych (wskazywane elementy /equipment/data/field, atrybut criterion).

```
<equipment>
   <interface>
      <method name=”getData”>
         <data-ref name-ref=”id”/>
         <data-ref name-ref=”state”/>
         <method-criterion value=”USER”/>
         <method-criterion value=”DEVICE”/>
      </method>
      <method name=”getState”>
         <data-ref name-ref=”state”/>
         <method-criterion value=”USER”/>
      </method>
   </interface>
   <data>
      <field name=”id” criterion=”DEVICE”/>
      <field name=”state” criterion=”USER”/>
      <field name=”time” criterion=”SYSTEM”/>
   </data>
</equipment>
```

Rysunek 7. Przykładowy pojedynczy warunek logiczny dotyczący zawartości dokumentu XML

2.1. Reguły jako klasy języka Java

W tym rozwiązaniu, opis warunku jest wyrażony w języku programowania Java. Przyjrzyjmy się najprostszej wersji tego modelu, która składa się z dwóch klas (por. rys. 8):

1. `XdvCondition` – interfejs opisujący regułę. Posiada tylko jedną metodę (`validate`) służącą do sprawdzenia czy podany dokumentu XML spełnia regułę. Metoda zwraca pusty obiekt (`null`) jeśli reguła jest spełniona lub opis błędu w przeciwnym razie.

2. `XdvValidationError` – obiekt z opisem błędu. Wyjaśnia, dlaczego dokument XML nie spełnia reguły.

Implementacją reguły jest klasa języka Java zgodna z interfejsem `XdvCondition`. Wywołanie reguły sprowadza się do utworzenia instancji klasy oraz wywołania metody `validate` z odpowiednimi parametrami. Rysunek 9 pokazuje jak można zapisać w tym modelu regułę opisaną na rysunku 7.

```
public interface XdvCondition {
    public XdvValidationError validate(org.w3c.dom.Document document);
}

public class XdvValidationError {
    public String message;
    public int errorLevel; //1=błąd, 2=ostrzeżenie
    public org.w3c.dom.Node node;
}
```

Rysunek 8. Interfejs opisujący regułę w języku Java oraz obiekt z opisem błędu

```java
import java.util.Map;
import java.util.Set;

public class CriterionMustMatch implements XdvCondition {
   public XdvValidationError validate(org.w3c.dom.Document doc) {
      //Utwórz mapę wiążącą nazwę pola danych z nazwą kryterium tego pola.
      Map fieldToCrit = new java.util.HashMap(); //nazwa pola->kryterium pola
      org.w3c.dom.NodeList fields = doc.getElementsByTagName("field");
      for(int f = 0; f < fields.getLength(); f++) {
         org.w3c.dom.Element field = (org.w3c.dom.Element) fields.item(f);
         fieldToCrit.put(field.getAttribute("name"), field.getAttribute("criterion"));
      }

      //Dla każdej metody:
      org.w3c.dom.NodeList methods = doc.getElementsByTagName("method");
      for(int i = 0; i < methods.getLength(); i++) {
         org.w3c.dom.Element method = (org.w3c.dom.Element) methods.item(i);

         //a. Pobierz zbiór kryteriów metody
         Set methodCrits = new java.util.HashSet();
         org.w3c.dom.NodeList critNl = method.getElementsByTagName("method-criterion");
         for(int j = 0; j < critNl.getLength(); j++) {
            org.w3c.dom.Element e = (org.w3c.dom.Element) critNl.item(j);
            methodCrits.add(e.getAttribute("value"));
         }

         //b. Pobierz zbiór kryteriów powiązanych pól danych
         Set fieldCrits = new java.util.HashSet();
         org.w3c.dom.NodeList fieldNl = method.getElementsByTagName("data-ref");
         for(int j = 0; j < fieldNl.getLength(); j++) {
            org.w3c.dom.Element e = (org.w3c.dom.Element) fieldNl.item(j);
            methodCrits.add(fieldToCrit.get(e.getAttribute("name-ref")));
         }

         //c. Porównaj zbiory kryteriów
         if (! methodCrits.equals(fieldCrits)) {
            XdvValidationError error = new XdvValidationError();
            error.errorLevel = 1; error.node = method;
            error.message = "W metodzie " + method.getAttribute("name") +
                " zbiory kryteriów metody i powiązanych pól danych nie są równe";
            return error;
         }
      }
      return null; //reguła jest spełniona
   }
}
```

Rysunek 9. Reguła opisana na rysunku 7 wyrażona w języku Java

Zalety prezentowanego rozwiązania:

1. Szeroki zakres możliwych do zdefiniowania warunków (por. rozdział 1.5.1) – jedynym ograniczeniem są możliwości języka Java.

2. Przezroczystość (por. rozdział 1.5.3).

3. Możliwość definiowania informacji o błędzie (por. rozdział 1.5.5) – obiekt `XdvValidationError` zawiera wszystkie wymagane informacje.

4. Interfejs dla języka Java (por. rozdział 1.5.8).

5. Prosta implementacja – wystarczy dostarczyć dwie klasy (por. rys. 8).

6. Wysoka wydajność – w razie potrzeby można wyliczyć często używane informacje raz, na początku wykonania. Takie podejście zastosowano w przykładzie z rysunku 9, gdzie informacje o kryterium i nazwie pola danych są wiązane ze sobą raz i wykorzystywane wielokrotnie w pętli.

Wady:

1. Skomplikowany sposób wyrażania reguł (por. rozdział 1.5.2) – implementacja reguły jest długa i mało czytelna. Widać to wyraźnie w przykładzie z rysunku 9. Definicja reguły zajmuje ponad 40 wierszy. Bez dodatkowych wyjaśnień i komentarzy w kodzie trudno się zorientować, jaki warunek sprawdza i w jaki sposób. To bardzo komplikuje proces rozbudowy i utrzymania reguł.

2. Wymagana dobra znajomość języka Java i niezbędnego API – osoby zajmujące się definiowaniem reguł najczęściej znają technologie związane z XML, takie jak XPath, XSLT, XML-Schema, ale nie zawsze są programistami znającymi język Java.

3. Brak jednego wzorca definiowana reguł – ta sama reguła może być zapisana na wiele różnych sposobów, co może prowadzić do powstania niejednolitego, trudnego w utrzymaniu i rozbudowie zestawu reguł.

Podobne wady i zalety będzie miał każdy model zbudowany w oparciu o język programowania ogólnego przeznaczenia, jak C++, Perl, Python, XSLT itp.

2.2. Schematron

Schematron jest językiem służącym do wyrażania asercji dotyczących dokumentu XML. Asercja w Schematron jest formułą XPath wyliczaną do wartości logicznej: prawda – asercja jest spełniona, fałsz – asercja nie jest spełniona. Pojedyncze warunki są grupowane w reguły zapisane jako XML, zwyczajowo przechowuje się je w plikach z rozszerzeniem `sch`.

Język został opracowany w 1999 roku w akademickim ośrodku badawczym Academia Sinica (Taiwan). W maju 2006 został oficjalnie zatwierdzony jako standard ISO/IEC 19757-3:2006. Specyfikacja języka jest ogólnie dostępna i może być używana bez ograniczeń. Schematron oferuje następujące możliwości (por. rys. 10 oraz opracowanie [7]):

1. Definiowanie reguł poprawności jako zestawu asercji wyrażonych przy użyciu XPath.

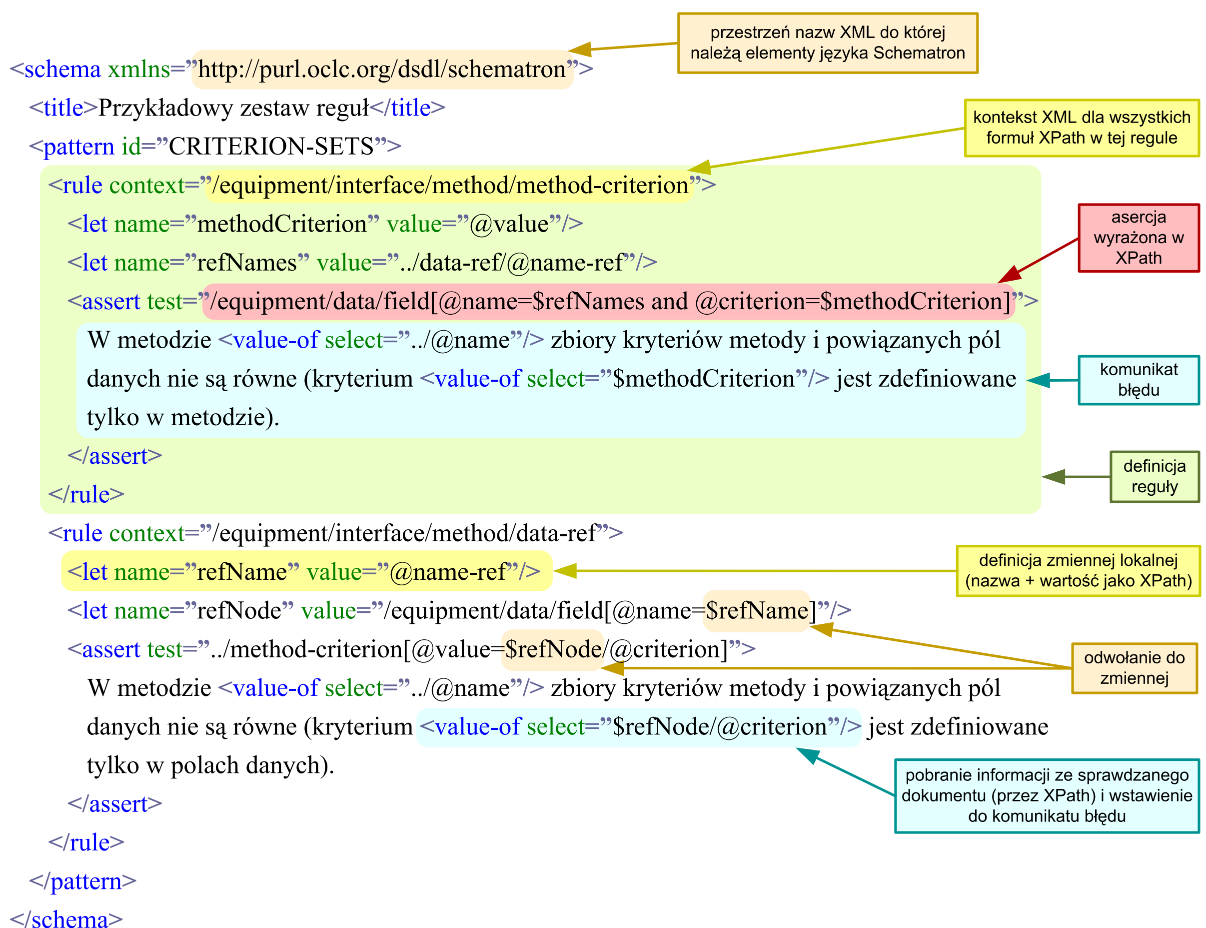
2. Definiowanie ogólnych, typowych reguł w celu późniejszego wielokrotnego użycia – tzw. reguły abstrakcyjne.

3. Raportowanie informacji o niespójnościach. Komunikat nie ma narzuconego formatu, może być dowolnym ciągiem tekstowym (w tym dokumentem XML). Można umieścić w nim informacje pobrane bezpośrednio ze sprawdzanego dokumentu (słowo kluczowe `value-of`).

4. Załączanie fragmentów kodu z plików zewnętrznych (słowo kluczowe `include`).

5. Definiowanie zmiennych globalnych i lokalnych, widzianych tylko w obrębie jednej reguły (słowo kluczowe `let`).
6. Współpraca z przestrzeniami nazw XML – możliwość wskazania, do której przestrzeni odwołuje się asercja (słowo kluczowe `ns`).
7. Możliwość osadzenia wewnątrz dokumentu XML-Schema. Elementy języka Schematron pochodzą z innej przestrzeni nazw niż elementy XML-Schema i mogą być umieszczone w tym samym dokumencie XML co definicja schematu.

Rysunek 10 pokazuje jak zapisać przy użyciu Schematron regułę opisaną na rysunku 7.



Rysunek 10. Reguła opisana na rysunku 7 wyrażona w języku Schematron

Istnieje popularna, darmowa implementacja Schematron wykonana w XSLT (por. [8]). Przekształca ona zestaw reguł do postaci transformacji XSLT, która zaaplikowana do sprawdzanego dokumentu XML daje odpowiedź czy dokument spełnia reguły czy nie. Niestety nie implementuje ona wszystkich cech języka, np. reguł abstrakcyjnych, zmiennych, przestrzeni nazw.

Większość darmowych implementacji dla innych języków (np. Java, PHP) to proste obudowy (ang. *wrapper*) dla implementacji XSLT. Przez to dziedziczą wszystkie ograniczenia rozwiązania bazowego. W chwili pisania tej pracy nie istniała natywna, darmowa implementacja dla języka Java.

Zalety prezentowanego rozwiązania:

1. Czytelność i prostota definiowanych reguł (por. rozdział 1.5.2) – osoby znające XPath potrafią stosunkowo łatwo i szybko zrozumień znaczenie reguły. Jedynym ograniczeniem jest konieczność opisu asercji pojedynczym wyrażeniem XPath – w przypadku bardzo złożonych warunków, z wieloma wyjątkami, asercja może zajmować kilka wierszy tekstu.

2. Przezroczystość (por. rozdział 1.5.3).

3. Współpraca z XML-Schema (por. rozdział 1.5.4) – możliwość osadzenia w tym samym dokumencie XML co definicja schematu (por. punkt 7 z listy możliwości języka).

4. Możliwość definiowania informacji o błędzie (por. rozdział 1.5.5) – informacja o błędzie może mieć dowolny format, może przechowywać dowolne dane w tym wartości pobrane bezpośrednio ze sprawdzanego dokumentu (por. punkt 3 z listy możliwości języka).

5. Oparcie na istniejących standardach (por. rozdział 1.5.6) – XML, XPath, sama definicja języka jest standardem ISO.

6. Popularność – kilka firm komercyjnych jak IBM, Topologi, SyncRO używa Schematron w swoich produktach (edytory XML, specyfikacje bazujące na XML, np. BI-ICS).

Wady:

1. Zakres możliwych do zdefiniowania warunków (por. rozdział 1.5.1). Ograniczeniem jest język XPath, który w obecnej wersji (1.0) ma ubogi zestaw wbudowanych funkcji i typów. W praktyce nie wystarcza to do zdefiniowania niektórych warunków.

2. Brak API dla języka Java (por. rozdział 1.5.8) – nie istnieje implementacja dla języka Java, która spełnia postawione wymagania.

2.3. CLiXML

CLiXML (ang. *Constraint Language in XML*) jest językiem służącym do definiowania reguł poprawności dla dokumentów XML. Jest to połączenie logiki pierwszego rzędu i języka XPath. Uniwersum dla predykatów logiki jest przestrzeń XML (badany dokument), wyrażenia XPath służą do wskazywania (pobierania) obiektów tego uniwersum. Każda reguła to

jedna formuła logiczna. Reguła jest spełniona wtedy i tylko wtedy, gdy powiązana z nią formuła jest prawdziwa.

Reguły są zapisywane jako XML, zwyczajowo przechowuje się je w plikach z rozszerzeniem `clx`. Zdefiniowano schemat XML-Schema, który określa format dokumentu z opisem reguł (por. [9]).

Język został opracowany w 1998 roku na uniwersytecie w Londynie (University College London). Z czasem z grupy badawczej wyodrębniono firmę Systemwire, która używa CLiXML w swoich produktach i zajmuje się dalszym rozwojem języka (na podstawie [10]). Mimo komercyjnego charakteru firmy, specyfikacja języka jest ogólnie dostępna i może być używana bez ograniczeń. CLiXML oferuje następujące możliwości (por. rys. 11 oraz opracowanie [9]):

1. Definiowanie reguł przy użyciu formuł logiki pierwszego rzędu i wyrażeń XPath. Język określa następujące elementy znane z logiki: kwantyfikator ogólny ($\forall$) i egzystencjalny ($\exists$), koniunkcja ($\wedge$), alternatywa ($\vee$), negacja ($\neg$), implikacja ($\Rightarrow$), równoważność ($\Leftrightarrow$). Wyrażenia XPath służą do wskazywania elementów przestrzeni XML, działają np. jako selektory w kwantyfikatorach. Język wprowadza zestaw operatorów do porównywania obiektów uniwersum: równy, mniejszy, większy, ten sam.

2. Definiowanie ogólnych, typowych formuł w celu późniejszego wielokrotnego użycia – tzw. makra.

3. Raportowanie informacji o niespójnościach. Komunikat nie ma narzuconego formatu, może być dowolnym ciągiem tekstowym (w tym dokumentem XML). Nie ma mechanizmu pozwalającego na umieszczenie w komunikacie wartości pobranych bezpośrednio ze sprawdzanego dokumentu.

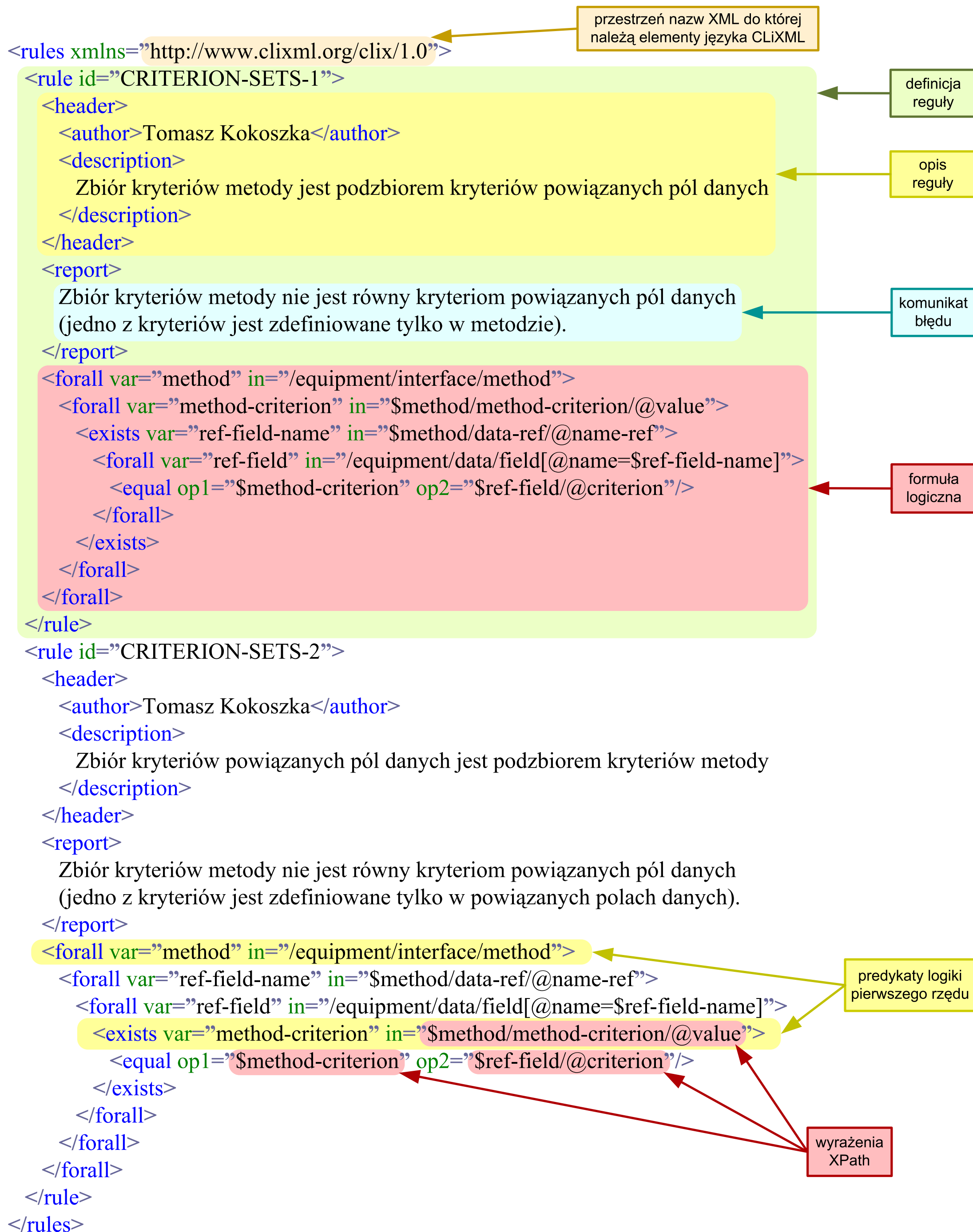
4. Załączanie makr z plików zewnętrznych.

5. Definiowanie zmiennych globalnych.

6. Możliwość osadzenia wewnątrz dokumentu XML-Schema. Elementy języka CLiXML pochodzą z innej przestrzeni nazw niż elementy XML-Schema i mogą być umieszczone w tym samym dokumencie XML razem z definicją schematu.

7. Rozszerzanie języka o nowe operatory. Mechanizm można wykorzystać do komunikacji z systemami zewnętrznymi, do programowania skomplikowanych obliczeń, które trudno wyrazić efektywnie w logice pierwszego rzędu itp. Specyfikacja języka określa sposób używania operatorów w formułach, sposób definiowania samych operatorów zależy od implementacji.

Rysunek 11 pokazuje jak zapisać przy użyciu CLiXML regułę zdefiniowaną na rysunku 7.



Rysunek 11. Reguła opisana na rysunku 7 wyrażona w języku CLiXML

Firma Systemwire udostępnia kompletny komercyjny produkt do pracy z językiem CLiXML – **xlinkit**. W jego skład wchodzi:

(a) graficzny edytor do definiowania, wykonywania i śledzenia wykonania reguł (jako wtyczka Eclipse),

(b) silnik wykonujący reguły,

(c) API dla języka Java,

(d) zestawy typowych, gotowych do użycia reguł.

W chwili pisania pracy była dostępna jedna darmowa implementacja CLiXML – **OpenCLiXML** [11]. Powstała w 2004 roku na uniwersytecie we Fryburgu (Szwajcaria). Oferuje bardzo uproszczone API dla języka Java (wykonanie sprawdzenia, wynik jako dokument XML), operuje na obiektach XML typu JDOM (nie W3C DOM), nie zawsze prawidłowo obsługuje zmienne w wyrażeniach XPath. Od chwili oficjalnej publicznej prezentacji w 2005 roku (wersja 0.3) projekt jest zawieszony – nie pojawiają się poprawki ani nowe wersje.

Zalety prezentowanego rozwiązania:

1. Szeroki zakres możliwych do zdefiniowania warunków (por. rozdział 1.5.1) – logika pierwszego rzędu w połączeniu z XPath oraz możliwość rozszerzania języka o nowe operatory dają bogate możliwości (por. punkty 1 i 7 z listy możliwości języka).

2. Intuicyjne wyrażanie reguł (por. rozdział 1.5.2) – język logiki pierwszego rzędu.

3. Przezroczystość (por. rozdział 1.5.3).

4. Współpraca z XML-Schema (por. rozdział 1.5.4) – możliwość osadzenia w tym samym dokumencie XML co definicja schematu (por. punkt 6 z listy możliwości języka).

5. Możliwość definiowania informacji o błędzie (por. rozdział 1.5.5) – informacja o błędzie może mieć dowolny format, może przechowywać dowolne dane (por. punkt 3 z listy możliwości języka).

6. Oparcie na istniejących standardach (por. rozdział 1.5.6) – rekomendacje XML i XPath oraz sama definicja języka są ogólnie dostępne.

Wady:

1. Brak możliwości umieszczania informacji pobranych bezpośrednio ze sprawdzanego dokumentu w komunikacie błędu (por. rozdział 1.5.5 oraz punkt 3 z listy możliwości języka).

2. Brak API dla języka Java (por. rozdział 1.5.8) – nie istnieje darmowa ani komercyjna implementacja dla języka Java, która spełnia wszystkie postawione wymagania.

2.4. Podsumowanie, wybór rozwiązania

W poprzednich podrozdziałach przedstawiłem trzy znane mi możliwe sposoby na definiowanie warunków logicznych dotyczących zawartości dokumentów XML. W świetle zapre-

zentowanych zalet i wad uważam, że optymalnym rozwiązaniem jest język CLiXML. Tym, co wyróżnia go spośród innych rozwiązań jest:

1. Bardzo szeroki zakres możliwych do zdefiniowania warunków – lepiej niż Schematron (por. [10] i [11]), równie dobrze jak Java.

2. Przejrzystość i czytelność zdefiniowanych reguł (lepiej niż Schematron i Java).

Problemy:

1. Brak możliwości umieszczania w komunikacie błędu informacji pobranych bezpośrednio ze sprawdzanego dokumentu.

2. Brak odpowiedniego API dla języka Java.

Pierwszy problem można rozwiązać tworząc stosowne rozszerzenie języka. Można to zrobić tak, by dokument z regułami zawierający rozszerzone instrukcje był nadal zgodny ze schematem języka (XML-Schema). Problem odpowiedniego API to kwestia implementacji.

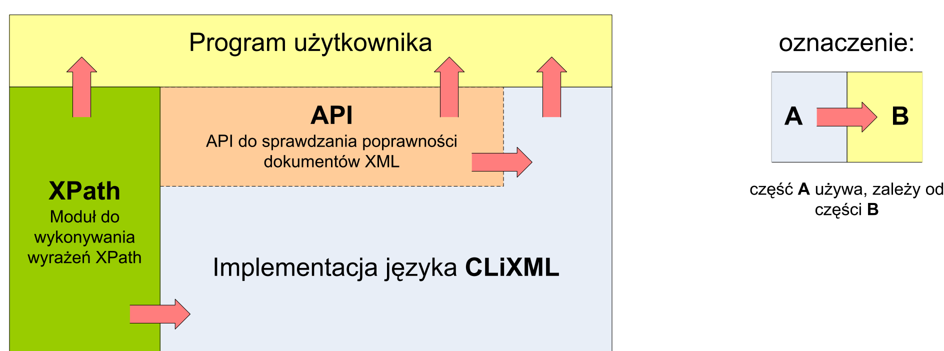
W następnym rozdziale opisuję wykonaną w języku Java implementację CLiXML. Implementacja dostarcza wymagane API oraz rozszerza język o brakujące elementy.

3. XDV – moduł sprawdzania poprawności dokumentów XML

XDV (ang. *XML Document Validator*) to moduł napisany w języku Java służący do sprawdzania poprawności dokumentów XML. Spełnia wymagania postawione w rozdziale 1.5 z uwzględnieniem wyników analizy przeprowadzonej w rozdziale 2. Moduł XDV składa się z trzech części (zależności pomiędzy nimi przedstawia rysunek 12):

1. Moduł do wykonywania wyrażeń XPath.
2. API do sprawdzania poprawności dokumentów XML.
3. Implementacja języka CLiXML (por. rozdział 2.3) – dostarcza niezbędne API, zawiera rozszerzenia opisane w rozdziale 2.4.

W kolejnych podrozdziałach opisuję szczegółowo każdą z części.



Rysunek 12. Zależności pomiędzy częściami składowymi modułu XDV

3.1. Moduł XPath

Wyrażenia XPath są wykorzystywane zarówno w Schematron (por. rozdział 2.2), jak i CLiXML (por. rozdział 2.3). Jeśli chcemy stworzyć implementację jednego z tych języków, potrzebujemy odpowiedniego API do wykonywania wyrażeń XPath. Istnieje kilka darmowych implementacji XPath dla języka Java z ogólnie dostępnym kodem źródłowym (na podstawie [12]):

1. Saxon – procesor XSLT, udostępnia API do obsługi XPath. Operuje na danych w formacie W3C DOM. Wersja 8.7.3 implementuje szkic rekomendacji XSLT 2.0 i XPath 2.0. Występuje w wersji komercyjnej i darmowej.

2. Xalan-J – procesor XSLT 1.0 rozwijany przez fundację Apache. Udostępnia API do obsługi XPath, operuje na danych w formacie W3C DOM.

3. Jaxen [14] – specjalizowany procesor XPath 1.0, może obsługiwać różne formaty danych wejściowych: W3C DOM, JDOM, dom4j itp.

Każde z tych rozwiązań udostępnia zupełnie inne API. Firma SUN wprowadziła ujednolicony interfejs dostępu do dokumentów XML, w tym do wykonywania poleceń XPath – JAXP [13] (ang. *Java API for XML Processing*). Jednak w obecnej wersji (1.3) jego możliwości są mocno ograniczone. Nie pozwala modyfikować wartościowania zmiennych po skompilowaniu wyrażenia. Wymaga podania typu wyniku (zgodnie z rekomendacją XPath 1.0) przed wykonaniem wyrażenia. Przykładowo wykonując zapytanie: `/equipment/interface` jako typ wyjściowy należy podać „zbiór obiektów” (ang. *node set*).

Z powodu tych ograniczeń wprowadziłem nowe API do używania wyrażeń XPath w kodzie Java – moduł XPath. Z założenia ma to być **fasada** [17] dla istniejących implementacji XPath 1.0. Oferuje on następujące możliwości:

1. Kompilowanie wyrażeń XPath w celu późniejszego wielokrotnego wykonania.

2. Wartościowanie zmiennych.

3. Automatyczne rzutowanie wyniku wykonania na odpowiedni typ XPath 1.0.

4. Abstrakcje typów danych zdefiniowanych w rekomendacji XPath 1.0; możliwość konwersji na typy języka Java (`java.lang.String`, `double`, `boolean` itp.).

5. Dowolność formatu wejściowego (W3C DOM, JDOM itp.) – każda implementacja API może operować na innym formacie.

Moduł XPath stanowi niezależną część projektu XDV. Definicja API znajduje się w pakiecie `org.xdv.xpath`. Pakiet `org.xdv.xpath.jaxen` zawiera fasadę zbudowana dla implementacji Jaxen [14], operuje na formacie W3C DOM (`org.w3c.dom.*`).

Architektura modułu bazuje na wzorcu projektowym **fabryka abstrakcyjna** [17]. Przykład użycia pokazano na rysunku 13. Na początku użytkownik tworzy konkretną instancję fabryki – `A::XPathEngine`. Potem kolejno, używając fabryki i operując tylko na abstrakcyjnych interfejsach (por. rys. 13):

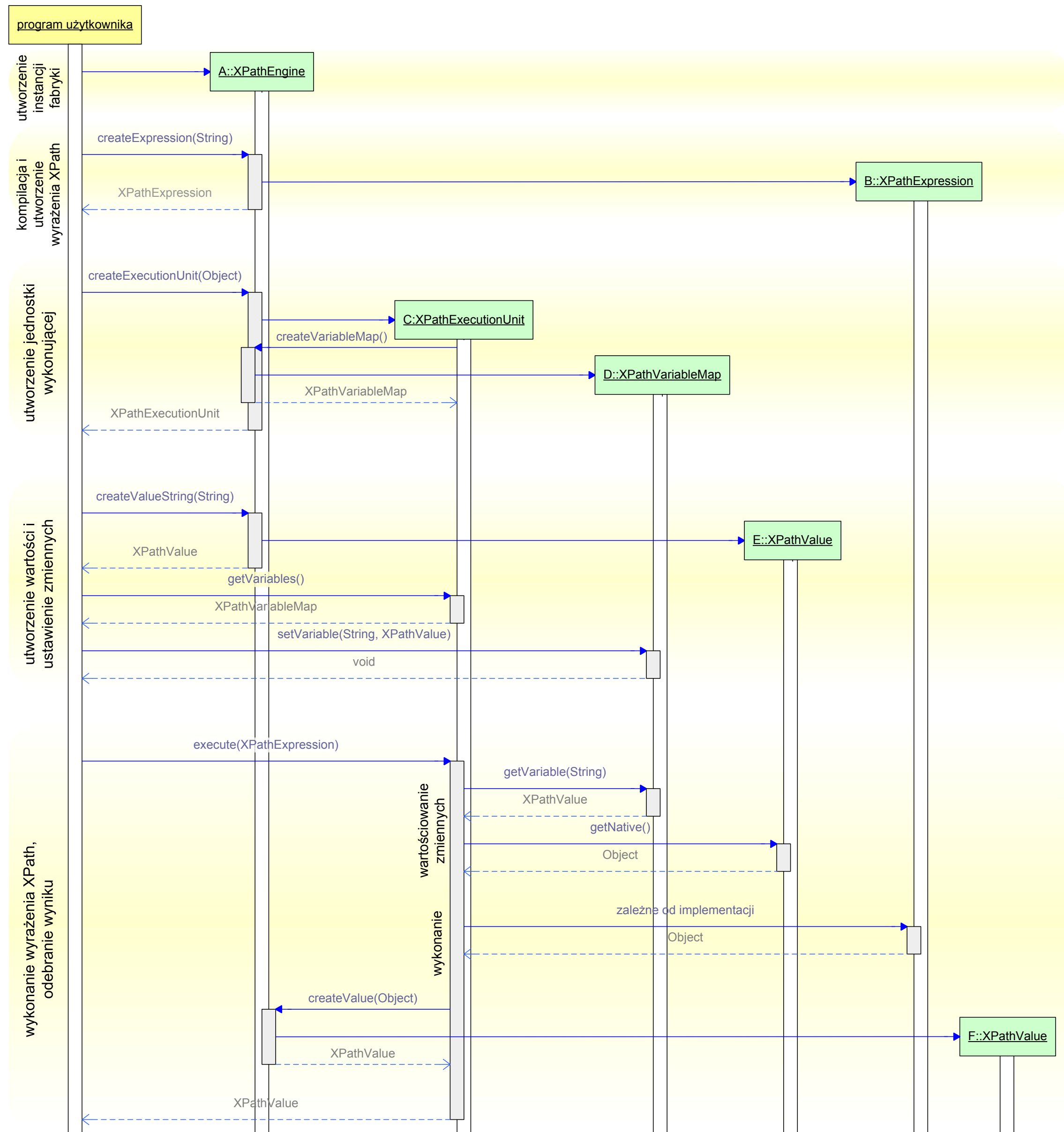
1. Kompiluje i tworzy wyrażenie XPath – `B::XPathExpression`.

2. Tworzy jednostkę wykonującą – `C::XPathExecutionUnit`.

3. Tworzy nowe wartości (`E::XPathValue`) i ustawia zmienne (`D:XPathVariableMap`).

4. Wykonuje wyrażenie (`D`) korzystając z jednostki wykonującej (`C`).

5. Odbiera wynik obudowany w stosowną fasadę – `F:XPathValue`.



Rysunek 13. Przykład użycia modułu XPath – diagram sekwencji

3.2. API do sprawdzania poprawności dokumentów XML

Jako część modułu XDV zdefiniowałem API, które ujednolica sposób wykonywania sprawdzeń dokumentów XML w kodzie Java. Jest to **fasada**, która przesłania szczegóły imple-

mentacji różnych rodzajów reguł – XML-Schema, Schematron, CLiXML itp. Dostarcza funkcjonalność opisaną w punkcie 1.5.8 listy wymagań (rozdział 1.5). Składa się z trzech podstawowych klas (pakiet `org.xdv.common`):

1. `XdvValidator` – interfejs do wykonania sprawdzenia dokumentu XML.
2. `XdvValidationError` – przechowuje informacje o wykrytych błędach; zawiera: komunikat błędu, poziom błędu (błąd, ostrzeżenie), powód błędu (np. element XML, który należy poprawić) – por. rozdział 1.5.5.
3. `XdvValidationException` – wyjątek zgłaszany, gdy sprawdzenie nie może być wykonane (np. nieprawidłowa definicja reguły).

API nie narzuca formatu danych wejściowych – każda implementacja może operować na innym typie danych (np. W3C DOM, JDOM itp.).

Podobną funkcjonalność oferuje pakiet JAXP [13] (ang. *Java API for XML Processing*) firmy SUN, jednak ze względu na ograniczenia w sposobie zbierania wyników błędów zdecydowałem się na wprowadzenie nowego API.

3.3. Implementacja języka CLiXML

Zasadniczą częścią modułu XDV jest implementacja języka CLiXML opisanego w rozdziale 2.3. Jak pokazałem podczas analizy, jest to optymalny język do wyrażania reguł poprawności dla dokumentów XML (por. rozdział 2.4). W kolejnych podrozdziałach opisuję podstawowe aspekty implementacji.

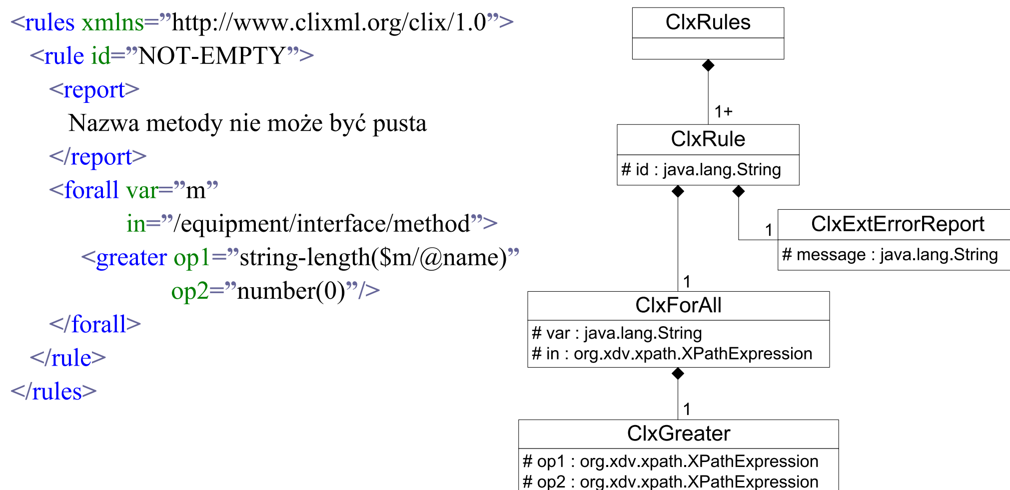
3.3.1. Wykonywanie reguł

Język CLiXML to połączenie logiki pierwszego rządu i wyrażeń XPath – ta definicja znajduje swoje odzwierciedlenie w implementacji. Wykonywanie reguł polega na interpretowaniu predykatów logiki i wykonywaniu wyrażeń XPath w celu pobierania obiektów uniwersum. Do obsługi XPath można użyć dowolnej implementacji zgodnej z interfejsem określonym w module XPath (por. rozdział 3.1). Format sprawdzanego dokumentu XML jest zdeterminowany tylko przez użytą fabrykę XPath – jest to jedyny fragment operujący na źródle danych (uniwersum), pozostała część implementacji CLiXML to interpretacja predykatów logiki pierwszego rzędu.

Aby wykonać sprawdzenie, reguła musi być w całości załadowana do pamięci. Wczytanie reguły polega na zastąpieniu każdego elementu języka CLiXML instancją odpowiadającej klasy języka Java z zachowaniem hierarchii. Obiekty układają się w drzewo. Jeśli element posiada argumenty w postaci wyrażeń XPath, to są one kompilowane. Za wczytywanie reguł odpowiadają klasy z pakietu `org.xdv.clx.builder` (implementacja w oparciu o wzorzec projektowy **budowniczy** [17]). Rysunek 14 pokazuje reprezentację w pamięci przykładowej reguły.

Przed wykonaniem reguły należy wczytać definicje makr, a następnie powiązać definicje makr i operatorów z odpowiednimi wywołaniami (por. rozdziały 3.3.4, 3.3.5).

Wykonanie reguł polega na wyliczeniu wartości powiązanej formuły logicznej. Zależnie od rodzaju, formuła może rekurencyjnie wywoływać formuły zależne lub bezpośrednio wyliczyć wartość. Kwantyfikatory działają jak iteratory – wyliczają wartość selektora (wyrażenie XPath), a następnie wykonują w pętli formułę zależną, wartościując zmienną kwantyfikowaną kolejnymi wynikami z selektora. Aby wykryć wszystkie miejsca, które nie spełniają reguły, zaimplementowałem mechanizm wznawiania obliczania formuły (por. rozdział 3.3.2).



Rysunek 14. Reprezentacja w pamięci przykładowej reguły CLiXML

3.3.2. Rozszerzenie – wznawianie wykonania

Prezentowana implementacja CLiXML nie zatrzymuje sprawdzania po napotkaniu pierwszego błędu, lecz wylicza wszystkie punkty niespójności. Jest to funkcjonalność znana z niektórych kompilatorów (np. GNU C Compiler) – w razie błędu kompilacji próbują pokazać użytkownikowi informacje o wszystkich miejscach w kodzie, które zatrzymują kompilację.

W mojej implementacji reguła może przekazać kolekcję błędów zamiast pojedynczego błędu. Jeśli powiązana z regułą formuła przekazuje fałsz, to reguła generuje komunikat błędu i przekazuje formule informację, z którego miejsca ma się wykonać ponownie. Reguła działa wedle tego schematu dopóty, dopóki formuła da się wznawiać.

Wznawianie ma sens tylko dla formuł zawierających kwantyfikatory (ogólne lub egzystencjalne). Kwantyfikatory podczas wyliczania nie zawsze przechodzą po wszystkich obiektach pobranych przez selektor – jeśli kwantyfikator ogólny napotka na wartość, która powoduje, że formuła kwantyfikowana nie jest spełniona, to natychmiast przekazuje wartość fałsz (analogicznie dla kwantyfikatora ogólnego). Wznawianie polega na wykonaniu ostatniego kwantyfikatora w formule od miejsca następującego po tym, które spowodowało wstrzymanie. Jeśli formuła przekazała prawdę lub wszystkie kwantyfikatory z formuły wykonały się „do końca”, to reguła nie wznawia wykonania formuły.

Każda formuła jest odpowiedzialna za wznowienie „swojego” wykonania. W najprostszym przypadku może to polegać na zleceniu wznowienia formule zależnej – tak wznawia

wykonanie predykat not (¬). Implementacja dostarcza klasę `ClxExecutionPoint` służącą do przechowywania informacji potrzebnych do wznowienia. Rysunek 15 pokazuje prosty przykład reguły i dokumentu, gdzie zastosowanie wznowień pozwala wykryć wszystkie niespójności.

Definicja reguły (CLiXML):

```
<rules xmlns=”http://www.clixml.org/clix/1.0”>
  <rule id=”METHOD-NAME”>
    <report>Nazwa metody musi się zaczynać od małej litery</report>
    <forall var=”m” in=”/equipment/interface/method”>
      <and>
        <greaterOrEqual op1=”substring($m/@name, 1, 1)” op2=”’a’”/>
        <lessOrEqual op1=”substring($m/@name, 1, 1)” op2=”’z’”/>
      </and>
    </forall>
  </rule>
</rules>
```

Sprawdzany dokument:

```
<equipment>
   <interface>
      <method name=”getData”/>
      <method name=”GetState”/>
      <method name=”SetState”/>
   </interface>
</equipment>
```

przerwanie wykonania formuły z wynikiem *„fałsz”*

po znalezieniu pierwszej niespójności, nastąpi wznowienie wykonania od tego miejsca

Rysunek 15. Wznawianie wykonania formuł – definicja reguły i przykładowy dokument z zaznaczonymi błędami, które zostaną zasygnalizowane użytkownikowi

3.3.3. Rozszerzenie – komunikat błędu

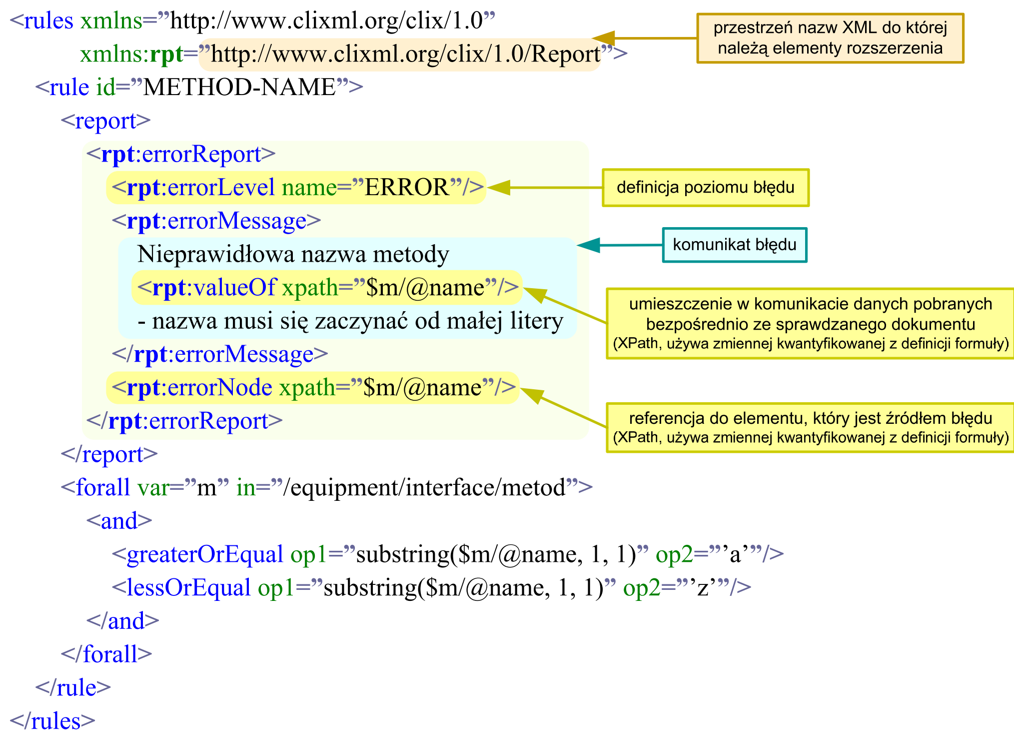
Aby spełnić warunek 1.5.5 z listy wymagań (rozdział 1.5), rozszerzyłem standard języka o możliwość umieszczania dodatkowych informacji w komunikacie błędu (por. rys. 16):

(a) poziom błędu (błąd, ostrzeżenie),

(b) referencja do elementu sprawdzanego dokumentu – poprzez XPath; możliwość używania zmiennych z formuły,

(c) dane pobrane bezpośrednio ze sprawdzanego dokumentu – poprzez XPath; możliwość używania zmiennych z formuły.

Moduł XDV zawiera schemat XML-Schema z definicją formatu rozszerzenia. Nowe elementy należą do innej przestrzeni nazw XML niż standardowe elementy języka, dzięki czemu reguły używające rozszerzenia pozostają zgodne z podstawowym schematem języka CLiXML. Rysunek 16 pokazuje przykład użycia opisywanego rozszerzenia.



Rysunek 16. Rozszerzony komunikat błędu – przykład

3.3.4. Rozszerzenie – wykonywanie makr

Język CLiXML pozwala definiować makra – parametryzowane, ogólne formuły identyfikowane unikatową nazwą. Specyfikacja podaje jak zbudować makro, ale nie określa sposobu wywoływania makr z definicji reguły (por. [9]).

Prezentowana implementacja wprowadza nowy element języka – `macroCall` – należący do przestrzeni nazw `http://www.clixml.org/clix/1.0/Macro`. Załączony do implementacji schemat XML-Schema precyzuje sposób używania nowego elementu. Wywołania makr są łączone z ich definicją po wczytaniu tekstu reguł i makr. Dzięki temu można tworzyć makra używające rekurencji bezpośredniej i pośredniej.

3.3.5. Rozszerzenie – definiowanie nowych operatorów

Specyfikacja języka CLiXML precyzuje sposób użycia operatorów w formułach, definiowanie samych operatorów zależy od implementacji (por. [9]). Rozwiązanie wprowadzone w XDV bazuje na wzorcu projektowym **fabryka abstrakcyjna** [17]. Składa się z dwóch części:

1. Fabryka (`ClxOperatorDefFactory`) – używana podczas ładowania reguły do tworzenia instancji definicji operatorów.

2. Definicja operatora (`ClxOperatorDef`) – używana w czasie wykonywania reguły do wyliczania wartości operatora (prawda/fałsz).

Kod związany z operatorami znajduje się w pakiecie `org.xdv.clx.operator`. Klasa `ClxXPathConditionImpl` to przykładowa implementacja operatora.

3.3.6. Zgodność ze specyfikacją, ograniczenia

Prezentowany moduł jest zgodny ze specyfikacją 1.0 języka CLiXML. Dodatkowo rozszerza język (por. rozdziały 3.3.2, 3.3.3), precyzuje kwestie otwarte (por. rozdziały 3.3.4, 3.3.5), dostarcza API zgodne z ogólnym modelem (por. rozdział 3.2). Implementuje wszystkie elementy języka z wyjątkiem słowa kluczowego `key`. Jest to odpowiednik elementu o tej samej nazwie z XSLT – pozwala definiować zbiory krotek <klucz, wartość>, dostarcza nową funkcję XPath (`key`) do pobierania wartości wskazanego klucza.

3.4. Podsumowanie

Przed przystąpieniem do pracy za główny cel postawiłem sobie stworzenie ogólnego, kompletnego pakietu do sprawdzania dokumentów XML. Dodatkowo miałem precyzyjną listę wymagań (por. rozdział 1.5). Moduł XDV dobrze łączy te dwa pomysły: dostarcza ogólne mechanizmy oraz zapewnia działające, gotowe do wykorzystania rozwiązanie. Jego podstawowa funkcjonalność to możliwość interpretowania (wykonywania) reguł poprawności zapisanych w języku CLiXML. Dodatkowo wprowadza pewne rozszerzenia (por. rozdział 3.3), przy jednoczesnym zachowaniu zgodności ze standardem języka. To pozwoliło stworzyć pakiet, który spełnia wszystkie wymagania opisane w rozdziale 1.5.

Podstawowe cechy modułu XDV:

1. Implementacja języka CLiXML – jest to jedyna (w chwili pisania pracy) w pełni funkcjonalna, darmowa implementacja języka CLiXML, która spełnia wymagania postawione w rozdziale 1.5. Oferuje lepszą metodę wykrywania niespójności niż wersja komercyjna (por. rozdział 3.3.2). Pozwala na łatwą podmianę silnika używanego do obsługi wyrażeń XPath (domyślnie Jaxen).

2. Bogaty mechanizm do wykonywania wyrażeń XPath – moduł XPath jest (w chwili pisania pracy) jedynym dostępnym API dla języka Java, które oferuje tak szeroki zakres możliwości (por. rozdział 3.1).

3. Modularność, otwarta budowa – dostarczone API do wykonywania sprawdzeń poprawności oraz moduł XPath to niezależne części. Stanowią bardzo dobrą podstawę do budowy implementacji innych języków służących do definiowania reguł – np. Schematron.

Pakiet XDV został z powodzeniem zastosowany w projekcie FESA realizowanym w CERN (por. rozdział 4). Wierzę, że rozwiązanie jest na tyle ogólne, że na pewno znajdzie zastosowanie również w innych projektach w przyszłości (już teraz, kolejny zespół projektowy w CERN wyraził wstępne zainteresowanie produktem).

4. Studium przypadku – projekt FESA

Moduł XDV został wykorzystany w projekcie FESA (ang. *Front End Software Architecture*) realizowanym w CERN. W kolejnych podrozdziałach opisuję czynniki, które skłoniły zespół projektowy do zastosowania właśnie tego rozwiązania. Przedstawiam sposób wprowadzenia modułu do istniejącej aplikacji.

4.1. Projekt FESA

Głównym celem projektu FESA jest dostarczenie architektury do programowania urządzeń sterujących pracą akceleratorów cząstek. FESA definiuje kompletny proces od projektowania, przez implementację i uruchomienie, po monitorowanie działającego oprogramowania. Odbiorcą jest specjalista, który buduje urządzenie sterujące i chce je podłączyć do systemów zewnętrznych. Obecnie FESA jest jedyną platformą tego typu w CERN.

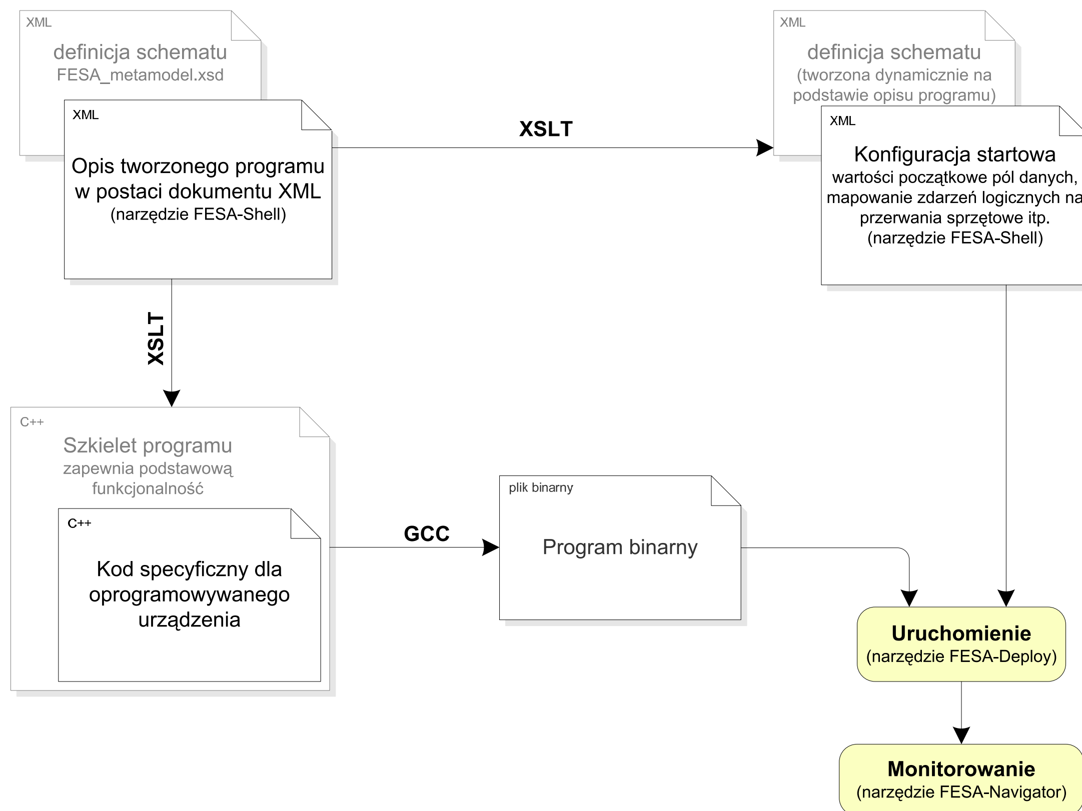
Model projektu opiera się na danych w formacie XML (por. rys. 17). Informacje o wzorcu programu, dostępnych interfejsach zewnętrznych, polach wewnętrznych, zdarzeniach itp. są przechowywane w postaci XML. Z nich jest generowany szkielet kodu C++, który zapewnia podstawową funkcjonalność, współpracę z systemami zewnętrznymi i synchronizację. Użytkownik musi go uzupełnić jedynie o kod specyficzny dla danego urządzenia – np. dla magnesu będzie on odpowiedzialny za zmianę natężenia prądu sterującego zależnie od informacji z systemu czasu. Kompletny program C++ jest kompilowany do platformy docelowej (PowerPC lub x86) i uruchamiany na specjalnym komputerze przemysłowym (ang. *front end*) podłączonym do niezbędnych interfejsów i urządzeń.

Projekt FESA dostarcza narzędzia, które wspomagają użytkownika na każdym etapie pracy i ukrywają szczegóły implementacji. Jednym z nich jest **FESA-Shell** – ogólny graficzny edytor XML sterowany przez XML-Schema. Prezentuje on dokument w postaci drzewa (por. rys. 18) i oferuje:

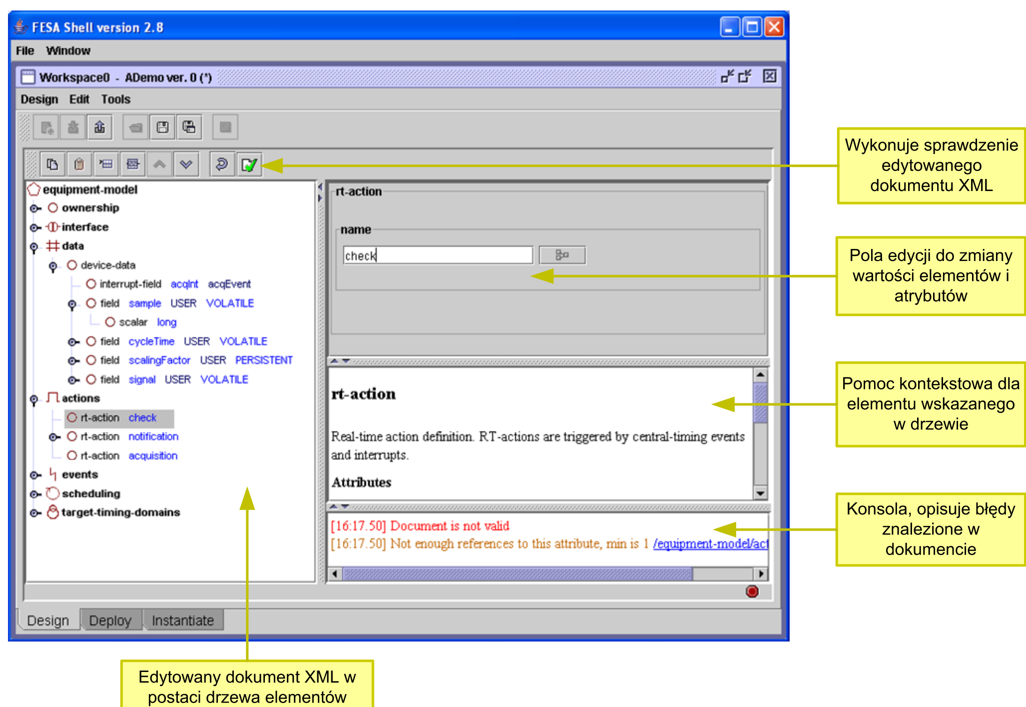
(a) pełne możliwości modyfikacji dokumentu w ramach zadanego schematu XML-Schema (można dodawać tylko elementy zdefiniowane w schemacie),

(b) pomoc kontekstową (treść pomocy jest pobierana ze schematu),

(c) sprawdzanie poprawności dokumentu, prezentację listy błędów z referencjami do elementów, które powodują błąd.



Rysunek 17. Projekt FESA – schemat działania, zastosowanie technologii XML



Rysunek 18. FESA-Shell – ogólny graficzny edytor XML sterowany przez XML-Schema

Aby kompilacja wygenerowanego kodu C++ i późniejsze uruchomienie programu wynikowego przebiegło pomyślnie, zawartość źródłowego dokument XML musi spełniać określone warunki. Przykładowo: nazwy pól danych muszą być poprawnymi identyfikatorami języka C++, każda akcja musi być skorelowana z przynajmniej jednym zdarzeniem (por. [16]). Jeśli dokument źródłowy jest poprawny, to w razie błędu można założyć, że wygenerowany szkielet programu również jest poprawny i użytkownik może się skupić na poszukiwaniu błędu tylko w kodzie, który sam dostarczył.

Od początku w projekcie FESA do testowania poprawności dokumentów używano XML-Schema. Wraz z rozwojem projektu pojawiało się coraz więcej warunków, których nie można było sprawdzać w ten sposób. Użytkownicy zwracali uwagę na częste przypadki, gdy ich program nie działał prawidłowo lub nie kompilował się, mimo iż narzędzia twierdziły, że wzorzec programu (dokument XML) jest poprawny, a kod który sami wpisali został sprawdzony i nie zawiera błędów.

Aby zmniejszyć liczbę napływających od użytkowników wniosków o pomoc, postanowiono rozszerzyć aplikację FESA-Shell o możliwość sprawdzania warunków logicznych w edytowanych dokumentach XML. Zadanie to zostało mi powierzone w trakcie praktyk studenckich w CERN.

4.2. Pierwsze podejście do problemu sprawdzania poprawności dokumentów XML

Ponieważ chciałem szybko dostarczyć działającą aplikację, postanowiłem zapisać dodatkowe warunki w języku Java i włączyć je do kodu edytora (również napisanego w Java). Zastosowałem rozszerzoną wersję rozwiązania opisanego w rozdziale 2.1. Dodałem możliwość generowania wielu błędów przez jedną regułę, zastosowałem wzorzec **fabryki** [17] do tworzenia instancji reguł, fabryki połączyłem w **łańcuch**. Dodanie/usunięcie fabryki z łańcucha pozwalało łatwo włączać i wyłączać wybrane reguły. Specyfika edytora FESA-Shell narzuciła schemat działania: zestaw reguł jest tworzony raz przez łańcuch fabryk (zaraz po załadowaniu aplikacji) i aplikowany wielokrotnie do zmieniającego się dokumentu XML.

Implementacja spełniła wymagania użytkowników, ale okazała się bardzo trudna w utrzymaniu. Wprowadzenie każdej modyfikacji zmuszało do kompilacji całego kodu aplikacji FESA-Shell i dostarczenia użytkownikom nowej wersji. Sposób wyrażania reguł był skomplikowany i pracochłonny – jedna reguła to średnio 94 wiersze kodu (bez komentarzy i wierszy pustych, najprostsza reguła – 40, najbardziej skomplikowana – 212). Po roku używania okazało się, że średni czas potrzebny do pełnego zaimplementowania, udokumentowania, przetestowania i dostarczenia obsługi nowej reguł to około 4 godziny pracy jednej osoby.

4.3. Zastosowanie modułu XDV

Po dyskusji wewnątrz zespołu FESA i po przeprowadzeniu stosownej analizy (por. rozdział 2), zdecydowałem się zastąpić istniejący model sprawdzania poprawności językiem

CLiXML (por. rozdział 2.3). Ponieważ istniejące implementacje języka nie spełniały naszych wymagań stworzyłem moduł XDV.

Jak moduł XDV zintegrował się z istniejącą aplikacją?

Pierwszym etapem prac było zapisanie wszystkich wymaganych warunków [16] w języku CLiXML z użyciem rozszerzonego komunikatu błędu (por. rozdział 3.3.3). Było to o tyle ułatwione, że dysponowałem listą wszystkich wymaganych warunków opisanych szczegółowo w języku naturalnym (por. [16]). W chwili włączania modułu XDV, lista zawierała 13 warunków. Po przepisaniu ich na język CLiXML otrzymałem 18 reguł – niektóre warunki było łatwiej wyrazić po rozdzieleniu ich na kilka części. Definicja reguły składała się średnio z 24 wierszy kodu (bez komentarzy i wierszy pustych, najprostsza reguła – 18, najbardziej złożona – 37). Cały proces zabrał dwa dni pracy. W wyniku powstał dokument XML z opisem wszystkich reguł – plik `/xdv-performance/fesa-all.clx` na dołączonym do pracy dysku CD.

Kolejnym etapem było wprowadzenie modułu XDV do edytora FESA-Shell. W tym celu wykorzystałem istniejące rozwiązanie – obudowałem wywołanie reguł CLiXML w interfejs wymagany przez reguły języka Java. Innymi słowy potraktowałem zestaw reguł CLiXML jako jedną regułę wyrażoną w języku Java. Dodałem dwie klasy:

1. Fabrykę – którą włączyłem w istniejący łańcuch fabryk produkujących instancje reguł,
2. Implementację reguły – obudowę (ang. *wrapper*) do wywoływania sprawdzeń CLiXML.

Na końcu wyłączyłem pozostałe fabryki reguł. Zaprogramowanie całości zajęło mi jeden dzień pracy. Po tym czasie otrzymałem w pełni funkcjonalną, nową wersję aplikacji FESA-Shell działającą w oparciu o język CLiXML i moduł XDV.

Według wstępnych obserwacji szacuję, że dostarczenie jednej reguł w nowym modelu zajmuje około 50 minut pracy.

4.4. Podsumowanie

Moduł XDV udało się łatwo włączyć w istniejącą aplikacją. Integracja nie wymagała dużych zmian w kodzie. Przejście na nowe rozwiązanie zajęło w sumie trzy dni pracy. Było to możliwe dzięki:

(a) otwartej, rozszerzalnej architekturze samej aplikacji FESA-Shell,

(b) szczegółowej dokumentacji – dysponowałem dokładnym opisem wymaganych warunków,

(c) przejrzystemu API dostarczonemu z modułem XDV,

(d) prostocie języka CLiXML.

Mając dwie działające wersje aplikacji FESA-Shell – jedną używającą XDV, drugą bazującą na regułach w języku Java – postanowiłem porównać wydajność obu rozwiązania. Wyniki porównania, wraz z zestawieniem złożoności reguł w obu wersjach zebrałem na rysunku 19.

Zgodnie z moimi oczekiwaniami moduł XDV działa nieznacznie wolniej od rozwiązania bazującego na języku Java – różnice rzędu 20%, por. rys. 19. Jednak przy wielkościach typu 300 ms i aplikacji takich, jak FESA-Shell nie ma to znaczenia – użytkownik nie zauważy różnicy w wydajności.

Zastąpienie reguł zapisanych w języku Java przez język CLiXML i moduł XDV pozwoliło osiągnąć zakładany cel: udało się uprościć proces utrzymania i rozwijania reguł przy zachowaniu dotychczasowej funkcjonalności. To wszystko osiągnięto przy niewielkim nakładzie pracy – wprowadzenie modułu XDV do aplikacji zajęło trzy dni pracy, z czego dwa zostały poświęcone na przepisanie istniejących reguł na język CLiXML.

		Java			CLiXML (XDV)		
Czas w milisekundach – średnia, odchylenie standardowe średniej i mediana na podstawie 30 prób		wartość średnia	odchylenie standardowe	mediana	wartość średnia	odchylenie standardowe	mediana
Wykonanie zestawu reguł zależnie od rozmiaru dokumentu XML i liczby niespójności	21KB, zgodny z regułami	**72**	31.8	**63**	**27**	26.2	**16**
	21KB, 2 niespójności	**64**	8.9	**63**	**19**	6.8	**16**
	150KB, zgodny z regułami	**325**	14.2	**328**	**374**	9.7	**375**
	150KB, 8 niespójności	**328**	24.8	**328**	**395**	15.9	**391**
	Załadowanie zestawu reguł	**39**[J1]	8.9	**46**[J1]	**13**[C1]	8.3	**15**[C1]
Liczba reguł		**16**[J2]			**18**[C2]		
Stopień kompilacji opisu jednej reguły – liczba wierszy kodu (bez komentarzy i wierszy pustych)		**94**			**24**		
Szacowany średni czas potrzebny na dodanie jednej nowej reguły do aplikacji (godziny:minuty)		**4h00**			**0h50**		

J1 – utworzenie instancji reguł z fabryk
C1 – załadowanie pliku z regułami, stworzenie odpowiedniej reprezentacji w pamięci
J2 – liczba różnych klas z definicjami reguł
C2 – liczba reguł, niezależnych formuł logiki pierwszego rzędu
Komputer testowy: Pentium4 2.8GHz, 768MB RAM, system operacyjny Windows XP Pro SP2
Maszyna wirtualna: SUN Java HotSpot™ 1.4.2_04-b05, limit pamięci 64MB

Rysunek 19. Porównanie wydajności i złożoności dwóch metod sprawdzania poprawności logicznej dokumentów XML – języka Java oraz CLiXML (implementacja XDV) – w oparciu o wyniku otrzymane dla aplikacji FESA-Shell

5. Podsumowanie

Podstawowy cel pracy, jakim było stworzenie w pełni funkcjonalnego pakietu do sprawdzania poprawności logicznej dokumentów XML został osiągnięty. Opracowany moduł XDV dostarcza ogólne API do testowania poprawności, zapewnia fasadę do obsługi wyrażeń XPath, implementuje język CLiXML. Z punktu widzenia użytkownika, ostatnia cecha jest szczególnie interesująca. CLiXML to połączenie logiki pierwszego rzędu z wyrażeniami XPath: formuły logiczne opisują regułę, XPath służy do pobierania obiektów uniwersum. W chwili pisania pracy XDV jest jedyną, w pełni funkcjonalną, darmową implementacją tego języka.

Ciekawą i ważną częścią była analiza i poszukiwanie optymalnego rozwiązania. Początkowo przeglądałem rozwiązania konkurencyjne do XML-Schema. Uznałem, że skoro jest ich tak dużo (por. [6]) to pewnie oferują dodatkowe możliwości. Bardzo szybko okazało się, że to ślepa uliczka – wszystkie są do siebie bardzo podobne i poza gramatyką niewiele się różnią. Jednocześnie zdziwił mnie fakt, że schemat RELANX NG jest mało znany i praktycznie nieużywany w porównaniu z XML-Schema, mimo że pierwszy jest standardem ISO, a drugi „tylko" rekomendacją. Kolejnym punktem na mojej liście był język Schematron (rozdział 2.2). Na pierwszy rzut oka wydawał się idealnym rozwiązaniem. Proste warunki udało mi się szybko i łatwo zdefiniować. Jednak zapis pierwszej skomplikowanej reguł zabrał mi niezmiernie dużo czasu a wynik, był niezbyt zrozumiały i czytelny. Kolejnym etapem analizy był język CLiXML (rozdział 2.3), który bardzo szybko okazał się optymalnym rozwiązaniem. Zaskoczeniem był dla mnie fakt, że oferuje większe możliwości niż Schematron, mimo że to „tylko" specyfikacja niszowej firmy z Londynu (w dodatku mało popularna), a Schematron to standard ISO dobrze znany wśród osób zajmujących się XML.

Na etapie implementacji ważną decyzją projektową było wprowadzenie niezależnej fasady do wykonywania wyrażeń XPath (rozdział 3.1). Dzięki temu implementacja języka może działać z dowolnym silnikiem XPath (domyślnie Jaxen). Dość kontrowersyjne okazało się nie narzucanie jednego formatu danych wejściowego. W założeniu pozwala to, przez prostą podmianę fasady XPath, sprawdzać dokumenty XML reprezentowane w różnych postaciach (W3C DOM, JDOM, dom4j itp.). Oczywiście dzięki temu program jest bardziej ogólny, jednak z drugiej strony format W3C DOM jest najbardziej popularny i najczęściej używany, a uogólnienie skomplikowało kod i obniżyło nieznacznie wydajność (format danych jest sprawdzany podczas wykonania a nie podczas kompilacji). Trudno jednoznacznie ocenić czy był to dobry pomysł czy nie.

Moduł XDV został z powodzeniem wprowadzony do projektu FESA realizowanego w CERN. Uprościł proces utrzymania i rozwoju aplikacji, przy zachowaniu istniejącej funkcjonalności. Jednocześnie samo wprowadzenie pakietu nie wymagało dużego nakładu pracy. Inny zespół projektowy w CERN również wyraził zainteresowanie modułem XDV.

Dzięki modularnej budowie można łatwo rozszerzyć funkcjonalność pakietu lub wykorzystać tylko jego część (np. obsługę wyrażeń XPath). Niniejsza praca magisterska nie oznacza końca rozwoju projektu XDV. W kolejnych wersjach planuję dodać fasady dla innych, wydajniejszych silników XPath (np. Saxon) oraz zaimplementować język Schematron – byłaby to pierwsza darmowa implementacja w pełni zgodna z nowym standardem ISO.

Dodatek A: Opis zawartości dołączonej płyty CD

Do pracy dołączyłem płytę CD o następującej zawartości:

(a) **doc/** – niniejsza praca oraz opracowanie [16] w wersji elektronicznej (format PDF),

(b) **xdv/** – kod źródłowy modułu XDV, może być łatwo zaimportowany do środowiska Eclipse 3.x (plik `.project`); zawiera skrypt budujący ANT (`build.xml`),

(c) **xdv/examples/** – przykładowe definicje reguł języka CLiXML,

(d) **xdv/lib/** – wymagane biblioteki zewnętrzne (Jaxen 1.0),

(e) **xdv/src/** – kod źródłowy (stan na dzień 2006-08-11),

(f) **xdv/schema/** – schematy XML-Schema definiujące składnię dodanych rozszerzeń języka CLiXML (por. rozdziały 3.3.3, 3.3.4),

(g) **xdv-api/** – dokumentacja techniczna, specyfikacja API wygenerowana automatycznie z kodu źródłowego,

(h) **xdv-jar/** – skompilowany moduł XDV (plik `*.jar`) wraz z wymaganymi bibliotekami zewnętrznymi (Jaxen 1.0),

(i) **xdv-performance/** – dokumenty XML oraz definicje reguł użytych podczas wykonywania testów wydajności (por. rozdział 4.4, rys. 19).

Bibliografia

[1] W3C, "*Extensible Markup Language (XML)*"
http://www.w3.org/XML/

[2] W3C, "*Namespaces in XML*", 14.01.1999
http://www.w3.org/TR/REC-xml-names/

[3] W3C, "*XML Path Language (XPath) version 1.0*", 16.11.1999
http://www.w3.org/TR/xpath

[4] W3C, "*XSL Transformations (XSLT) version 1.0*", 16.11.1999
http://www.w3.org/TR/xslt

[5] W3C, "*XML-Schema: primer, structures, datatypes*", 28.10.2004
http://www.w3.org/TR/xmlschema-0/, (xmlschema-1/, xmlschema-2/)

[6] Eric van der Vlist, "*Comparing XML Schema Languages*", rozdział "*A Short History of XML Schema Languages*", 12.12.2001
http://www.xml.com/pub/a/2001/12/12/schemacompare.html#history

[7] ISO/IEC FDIS 19737-3 – specyfikacja ISO Schematron, 2004
http://www.schematron.com/iso/dsdl-3-fdis.pdf

[8] Rick Jelliffe, Academia Sinica Computing Center, "*Beta Skeleton Module for the Schematron 1.5 XML Schema Language*", 2001
http://xml.ascc.net/schematron/1.5/skeleton1-5.xsl

[9] Systemwire, "*CLiX language specification*", 2003
http://www.clixml.org/spec.html

[10] Christian Nentwich (Systemwire), "*CLiX - A Validation Rule Language for XML*", konferencja "*W3C Workshop on Rule Languages for Interoperability*", 2005
http://www.w3.org/2004/12/rules-ws/paper/24/

[11] Dominik Jungo, "*Open CliX – an open source CLiXML Schema Validator*", 2005
http://clixml.sourceforge.net/

[12] Elliotte Rusty Harold, "*Processing XML with Java*", rozdział "*XPath Engines*", 2001
http://www.cafeconleche.org/books/xmljava/chapters/ch16s05.html

[13] Sun Microsystems, "*Java API for XML Processing (JAXP)*"
http://java.sun.com/xml/downloads/jaxp.html

[14] Codehaus, "*Jaxen – universal Java XPath engine*"
http://jaxen.org/

[15] Ana Guerrero, Jean-Jacques Gras, Jean-Luc Nougaret, Michael Ludwig, Michel Arruat, Stephen Jackson, „*CERN Front-End Software Architecture for accelerator controls*", konferencja ICALEPCS, 2003
http://icalepcs2003.postech.ac.kr/Proceedings/PAPERS/WE612.PDF

[16] Tomasz Kokoszka, "*FESA Tools – non-standard constraints*", część dokumentacji technicznej projektu FESA, 11.01.2006

[17] James W. Cooper "*Java design patterns*", second printing 2000