Автоматическая генерация исходных текстов для повышения качества программных продуктов

Минко В.С., ОАО «ИнфоТеКС», ПиР ПАК

Введение

С точки зрения информационной безопасности природу всех проблем с качеством программного обеспечения можно разделить на три категории:

1. Намеренные закладки, внесённые внутренним злоумышленником на этапе разработки ПО;
2. Ненамеренные (случайные) ошибки в ПО, отражающие несоответствие программ их спецификациям;
3. Ошибки в самих спецификациях, не выявленные до выпуска продукта.

Ошибки второй категории составляют подавляющее большинство от их общего числа. Традиционным способом борьбы с такими ошибками является тестирование ПО перед выпуском. Однако, в силу разных причин тестирование не может гарантировать полное исключение ошибок. Поэтому не спадает актуальность технологий, способствующих исключению некоторых видов ненамеренных ошибок в ПО. Таковых технологий становится всё больше, отметим лишь некоторые из них:

1. Рост «интеллектуальности» компиляторов;
2. Сборка мусора;
3. Языково-ориентированная методология;
4. Автоматическая генерация исходных текстов.

Рассмотрим каждую из перечисленных технология немного подробнее.

Рост «интеллектуальности» компиляторов

В "ранней молодости" языка С следующий код был бы успешно воспринят компилятором без предупреждений:

```
if (x = 5)
{
  /* … */
}

somefunc(void)
{
  return;
}
```

Однако, этот код содержит ряд ошибок: использование присвоения как логического выражения, выход из функции без возврата результата и неявное объявление возвращаемого

значения функции, что уже запрещено по стандарту C99. Современный компилятор с должным набором заданных опций все перечисленные ошибки успешно обнаружит:

```
test.c: ошибка: по умолчанию возвращаемый тип функции - «int»
 somefunc(void)
     ^
test.c: ошибка: оператор «return» без значения в функции, возвращающей не void-значение
  return;
  ^
test.c: ошибка: присваивание, используемое как логическое выражение
   if (x = 0) {
 ^
```

К сожалению, никакой современный компилятор не способен обнаружить все ошибки в исходном тексте. Зачастую компилятору для этого просто недостаточно информации. Указанный недостаток частично исправляется повышением строгости языков. Например, информативности объявления функций, как в примере выше, где стандарт C99 требует явное объявление возвращаемого значения функции. Тем не менее, если внимательно относиться к сообщениям компилятора и не игнорировать их, это позволяет исключить в коде программ некоторые виды распространённых ошибок.

Сборка мусора

Традиционным способом управления памятью является ручной. Его сущность заключается в следующем:

- Для создания объекта в динамической памяти программист явно вызывает команду выделения памяти. Эта команда возвращает указатель на выделенную область памяти, который сохраняется и используется для доступа к ней.
- Когда надобность в объекте проходит, программист явно вызывает команду освобождения памяти, передавая ей указатель на удаляемый объект.

При таком подходе потенциально возможны две проблемы:

- Висячая ссылка – это оставшаяся в использовании ссылка на объект, который уже удалён;
- Утечка памяти – сохранение объекта в динамической памяти после завершения его использования.

Сборка мусора – это технология, позволяющая, устранить эти потенциальные ошибки. В системе со сборкой мусора обязанность освобождения памяти от объектов, которые больше не используются, возлагается на среду исполнения программы. Программист лишь создаёт динамические объекты и пользуется ими, он может не заботиться об удалении объектов, поскольку это делает за него среда. Для осуществления сборки мусора в состав среды исполнения включается специальный программный модуль, называемый «сборщиком мусора». Этот модуль периодически запускается, определяет, какие из созданных в динамической памяти объектов более не используются, и освобождает занимаемую ими память. В настоящее время она используется в Java, Python, Ruby, Perl и других языках.

Языково-ориентированная методология

Языково-ориентированное программирование (ЯОП) – это парадигма программирования, заключающаяся в разбиении процесса разработки программного обеспечения на стадии создания предметно-ориентированных языков (англ. Domain-specific language, DSL) и описания собственно решения задачи с их использованием. Предметно-ориентированный язык – компьютерный язык, специализированный для конкретной области применения (в противоположность языку общего назначения, применимому к широкому спектру областей и не учитывающему особенности конкретных сфер знаний). К таковым языкам относятся:

- Perl для манипулирования текстами;
- TeX для компьютерной вёрстки;
- SQL для СУБД;
- HTML для разметки документов;
- Языки командных оболочек операционных систем (например, rvpn_shell в ПАК ViPNet Coordinator HW).

Построение такого языка и/или его структура данных отражают специфику решаемых с его помощью задач. Из этого вытекает, что, следуя методологии ЯОП, можно создать такой язык DSL, в котором многие ошибочные с точки зрения предметной области языковые конструкции, которые были бы допустимы в случае реализации решения на языке общего назначения, будут просто запрещены компилятором или интерпретатором DSL.

Автоматическая генерация исходных текстов

Дальнейший текст посвящён рассмотрению технологии автоматической генерации исходных текстов как способа борьбы с ошибками отклонения программ от их спецификаций. Для пояснения этой технологии, рассмотрим в качестве контекста задачи, при решении которых было найдено применение технологии автоматической генерации исходных текстов.

Задача №1 - организация лицензионных правил

Программно-аппаратные комплексы ViPNet Coordinator HW имеют сложную систему лицензирования, которая решает целый комплекс задач:

1. Идентификация аппаратной платформы;
2. Определение лицензионных объектов на основании справочно-ключевой информации;
3. Определение свойств различных лицензионных объектов с точки зрения функциональности и производительности;
4. Определение соотношений между аппаратными платформами и лицензионными объектами.

Лицензионный объект является артефактом, разрешающим определенную функциональность на компьютере с ПО ViPNet. Таковым может быть **coordinator-hw-100-A**, разрешающий использование функциональности ViPNet Coordinator на платформе HW100. Свойства лицензионного объекта (или *лицензионные ограничения*) детализируют особенности

лицензионных объектов и задают некоторые настройки функционирования ПО ПАК. В таблице ниже приведены несколько свойств лицензионных объектов и их описаний.

Таблица 1: Перечень различных свойств главных лицензионных объектов узла

Имя	**Значение (по умолчанию)**	**Описание**
iplir-threads	целое число >=0 (0)	Максимальное число потоков шифрования. Если равно нулю, то равно числу ядер процессора.
traffic-shaper	целое число >=0 (0)	Ограничение трафика в Mb/s.* Если равно нулю, то неограниченно.
default-max-connections	целое число >=0 (150000)	Максимальное число соединений межсетевого экрана по умолчанию.
manual-max-connections	целое число >=0 (150000)	Максимальное число соединений межсетевого экрана, устанавливаемое вручную.

В ранних версиях Coordinator HW все лицензионные правила и соотношения между приведенными артефактами кодировались в скриптах или в компилируемом коде программ. Это было крайне неудобно с точки зрения локализации изменений, обзорности и четкости понимания правил по коду. Данный способ реализации правил лицензирования приводил к ошибкам. В итоге это могло привести к тому, что, например, при неправильном максимальном числе соединений МСЭ можно исчерпать доступную оперативную память ПАК и вызвать этим отказ в функционировании других важных подсистем. Иными словами – реализовать атаку типа DoS.

Для исправления этих недостатков было решено применить технологию XSLT-преобразований.

XSLT

XSLT (e**X**tensible **S**tylesheet **L**anguage **T**ransformations) - это язык преобразования XML-документов в XML-документы другой схемы, другие форматы документов (например, HTML), либо простые текстовые файлы. Функцию преобразования выполняет специальная программа – XSLT-процессор. В простейшем случае схема применения XSLT выглядит следующим образом: процессор получает на входе два документа — входной XML-документ и таблицу стилей XSLT — и создает на их основе выходной документ.

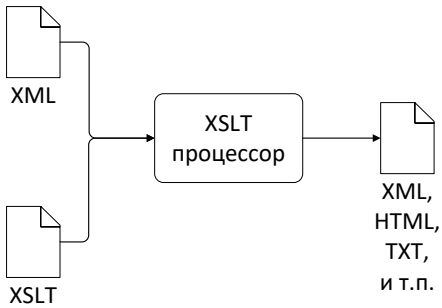

Рисунок 1: Общая схема применения XSLT

XSLT имеет множество различных применений, в основном в области веб-программирования и генерации отчётов. Однако, его можно использовать и для автоматической генерации исходных текстов программ. Рассмотрим, как это может быть выполнено применительно к описанной выше задаче организации лицензионных правил.

Декларация лицензионных правил в формате XML

Все лицензионные декларации могут быть оформлены в виде XML-файлов. Например, следующий файл позволяет сопоставить лицензионным объектам их лицензионные ограничения. Для краткости в нём оставлены декларации только для семейства аппаратных платформ HW1000.

```
<liclimits>
	<group comment="Russian hw1000">
		<licobjects>
			<licobject id="coordinator-hw-1000"/>
		</licobjects>
		<properties>
			<property name="iplir-threads" value="2"/>
			<property name="failover" value="yes"/>
			<property name="default-max-connections" value="800000"/>
			<property name="manual-max-connections" value="1000000"/>
		</properties>
	</group>
</liclimits>
```

В Coordinator HW многие лицензионные объекты имеют одинаковый набор свойств, поэтому представляется целесообразным сопоставлять несколько лицензионных объектов со списком общих для них свойств. Это реализуется внутри структуры *group*, содержащей вложенные структуры *licobjects* и *properties*. Свойства лицензионных объектов задаются тегами *property* и группируются тегом *properties*.

Трансляция лицензионных деклараций в исходный код

Каждый файл с лицензионными декларациями транслируется в отдельный заголовочный C++-файл с использованием XSLT-преобразований. Для каждого XML-файла сформирован свой XSLT-документ с инструкциями. В данном случае схема применения XSLT выглядит следующим образом:

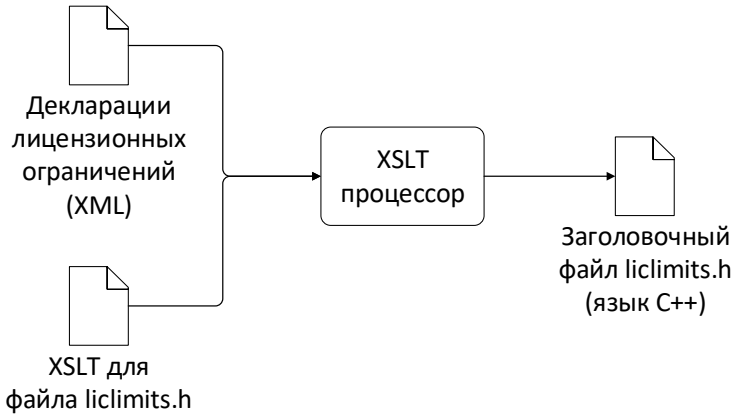

Рисунок 2: Схема применения XSLT для трансляции лицензионных деклараций

Рассмотрим фрагмент таблицы стилей XSLT, которая используется для преобразования приведённого выше файла XML.

```
  <xsl:for-each select="group">
    <xsl:for-each select="licobjects">
      <xsl:for-each select="licobject">
        <xsl:variable name="licobjid" select="@id"/>
        <xsl:for-each select="../../properties/property">
          <xsl:text>     result["</xsl:text>
          <xsl:value-of select="$licobjid"/>
          <xsl:text>"]["</xsl:text>
          <xsl:value-of select="@name"/>
          <xsl:text>"] = "</xsl:text>
          <xsl:value-of select="@value"/>
          <xsl:text>";
</xsl:text>
        </xsl:for-each>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:for-each>
```

В этом фрагменте заложена следующая логика обработки:

1. для каждого тега *group* выбираются вложенные теги *licobject*; которые должны быть вложены в общий тег *licobjects*;
2. для каждого найденного тега *licobject* осуществляется выборка всех тегов *property*, находящихся в том же теге *group*;
3. для каждых полученных пар *licobject* и *property* создаётся строка, добавляющая запись в хеш-таблицу значений свойств лицензионных объектов.

В итоге для каждого лицензионного объекта заполняется таблица, в которой в качестве ключа используется название свойства (значение атрибута *name* тега *property*) ассоциированное со значением (значение атрибута *value* тега *property*). При этом каждая таблица некоторого лицензионного объекта в свою очередь ассоциирована с названием лицензионного объекта (значение атрибута *id* тега *licobject*) в общей хеш-таблице.

Результатом трансляции приведённого выше XML-файла будет файл заголовочный C++-файл **`liclimits.h`**, содержащий функцию инициализации хеш-таблиц со свойствами лицензионных объектов:

```
static Limits
liclimits_create_limits( void )
{
     Limits result;
     result["coordinator-hw-1000"]["iplir-threads"] = "2";
     result["coordinator-hw-1000"]["failover"] = "yes";
     result["coordinator-hw-1000"]["default_max_connections"] = "800000";
     result["coordinator-hw-1000"]["manual_max_connections"] = "1000000";
     return result;
}
```

Как видно, результат трансляции вполне нагляден и доступен для восприятия не только компилятором, но и человеком. Однако, при желании можно написать такой XSLT-файл, который

будет генерировать намеренно запутанный код, сложный для восприятия человеком. При этом в отличие от специальных программ, производящих запутывание кода (обфускаторы), можно будет гарантировать работоспособность результата этого преобразования.

Интерфейс для получения лицензионных деклараций

Сгенерированные заголовочные файлы (например, **liclimits.h**) не являются самодостаточными и неудобны для непосредственного использования в исходных кодах лицензируемых продуктов. Каждый заголовочный файл должен быть снабжён специальным интерфейсом, предоставляющим удобный доступ к данным, проинициализированным в заголовочном файле. Интерфейсы для сгенерированных заголовочных файлов написаны на языке C++, они статичны (т.е. написаны человеком вручную).

Рассмотрим интерфейс для получения свойств лицензионных объектов, который основан на **`liclimits.h`**. Сами заголовочные файлы включаются в файле интерфейса обычной директивой `include`.

```
namespace hwlic {
// Defines liclimits_create_limits
#include <hwlic/module/liclimits.h>
}
```

После чего становится возможной инициализация констант, содержащих свойства лицензионных объектов. Для этого используются функции из сгенерированных заголовочных файлов (в нашем примере это функция `liclimits_create_limits` из файла **liclimits.h**).

```
namespace {
     const Limits liclimits = hwlic::liclimits_create_limits();
}
```

Впоследствии инициализированная переменная `liclimits` используются в интерфейсных функциях. Например, функция `licobject_props()` возвращает ассоциативный массив из лицензионных свойств и их значений для заданного лицензионного объекта.

```
std::map<std::string, std::string>
licobject_props( const std::string& licobjid )
{
     if ( liclimits.count( licobjid ) == 0 )
     {
          throw std::runtime_error( "The specified licobject in unknown." );
     }

     return liclimits.find(licobjid)->second;
}
```

Совокупность функций `licobject_props()` и подобных ей образуют C++-интерфейс, позволяющий получать доступ к лицензионным декларациям из исполнимого кода.

Задача №2 - автоматизация настройки ПАК

Рассмотрим ещё одну задачу из области программно-аппаратных комплексов ViPNet Coordinator HW, при решении которой была использована технология автоматической генерации исходных текстов.

Классическим способом настройки ПАК является использование интерфейса командной строки (англ. Command Line Interface, **CLI**). Например, настройка статического адреса `10.0.88.1` на сетевом интерфейсе `eth2` через CLI выглядит следующим образом:

```
inet ifconfig eth2 up

inet ifconfig eth2 address 10.0.88.1 netmask 255.255.255.0
```

Интерфейс командной строки стабилен (т.е. формат команд изменяется редко) и присутствует во всех модификациях ПАК Coordinator HW.

Тем не менее, CLI удобен не во всех случаях. Например, при первичной настройке большого числа ПАК-ов одинаковым образом, удобнее было бы один раз задать желаемые настройки и иметь возможность переиспользовать их на всех ПАК, нежели вручную задавать одни и те же настройки раз за разом. Аналогичная ситуация возникает при внесении одинаковых изменений в конфигурации уже функционирующих ПАК-ов.

Для решения этих задач в ПАК Coordinator HW предусмотрен функционал автоматической конфигурации при первичной настройке, а также поддержка удалённого централизованного управления. В обоих случаях на ПАК поступает файл в формате XML (*config.xml*) с заданными настройками, которые необходимо обработать и применить. Ниже приведён фрагмент такого файла для конфигурации статического адреса на интерфейсе, идентично примеру выше с командами CLI:

```
<networking>
    <interfaces>
        <interface name="eth2">
            <param name="enable" value="yes"/>
            <param name="mode" value="static"/>
            <param name="ip_mask" value="10.0.88.1/24"/>
         </interface>
    </interfaces>
    <routes/>
</networking>
```

Трансляция настроек ПАК в команды CLI

Поскольку в ПАК уже присутствует готовый и отлаженный механизм задания всех возможных настроек через интерфейс командной строки, для применения автоматических настроек из *config.xml* имеет смысл использовать его повторно, а не создавать новую подсистему. Для конвертации настроек в команды интерпретатора используется уже известная нам технология XLST, схема применения которой для данного случая приведена ниже на рис. 3.

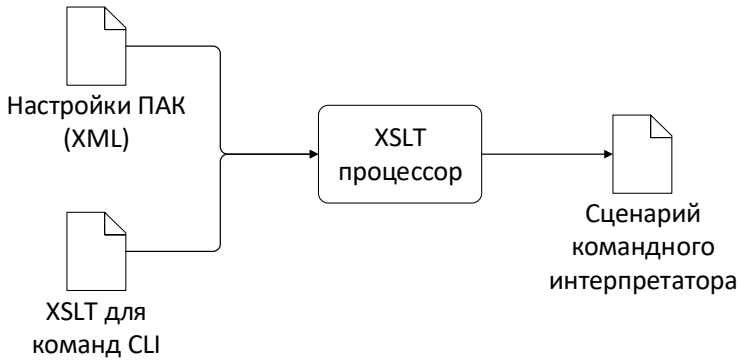

Рисунок 3: Схема применения XSLT для автоматической конфигурации ПАК

Таким образом, вся реализация функции автоматической конфигурации ПАК сводится к трансляции конфигурационных файлов в сценарии командного интерпретатора и передаче их последнему на исполнение. При этом сами таблицы стилей XSLT получаются достаточно простыми и наглядными. Например, следующий фрагмент используется для преобразования конфигурации статического адреса на сетевом интерфейсе в соответствующие команды CLI.

```
<xsl:if test="@name = 'ip_mask'">
    <xsl:text>inet ifconfig </xsl:text>
    <xsl:value-of select="../@name"/>
    <xsl:text> address </xsl:text>
    <xsl:value-of select="substring-before(@value,'/')"/>
    <xsl:text> netmask </xsl:text>
    <xsl:call-template name="mask2mask">
        <xsl:with-param name="mask" select="substring-after(@value,'/')"/>
    </xsl:call-template>
    <xsl:text>
    </xsl:text>
</xsl:if>
```

Упрощение совместимости разных версий ПО

Использование XSLT позволяет не только переиспользовать готовый функционал настройки, сократив тем самым трудозатраты и число потенциальных ошибок. Не менее важным достоинством такого архитектурного решения является абстрагирование автоматической конфигурации от деталей реализации механизма применения настроек. Одни и те же настройки могут быть переданы на программно-аппаратные комплексы, имеющие совершенно разную реализацию функционала, подлежащего конфигурации.

Помимо очевидного упрощения совместимости разных версий ПАК с управляющим ПО, этот факт также упрощает проведение обновления прошивок ПАК. Например, в ПАК ViPNet Security Gateway версии 3.2 для поддержки IPSec использовался демон Racoon. Позднее в версии 4.1 подсистема IPSec была переведена на альтернативную реализацию – StrongSwan. При обновлении прошивки ПАК с версии 3.2 на версию 4.1 необходимо было обеспечить конвертацию настроек

Racoon в настройки StrongSwan. При этом хотелось избежать жёсткой привязки механизма конвертации настроек к текущей реализации IPSec, поскольку, очевидно, через этот функционал проходит ось изменений, и есть существенная вероятность, что в будущих версиях ПАК он также будет изменён.

Поэтому была реализована утилита для преобразования настроек Racoon в уже используемый формат настроек ПАК для автоматической конфигурации.

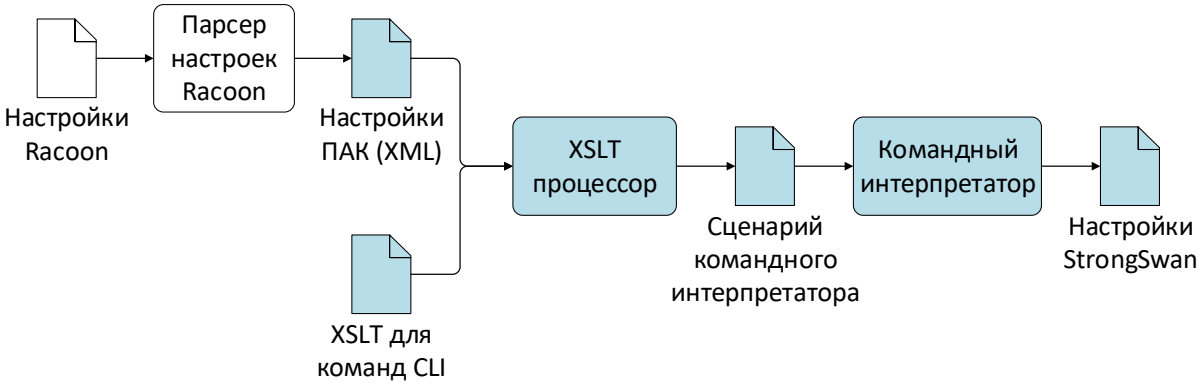

Рисунок 5: Схема использования средств автоматической настройки ПАК для конвертации настроек IPSec

Часть подсистемы, обозначенная на рис. 5 голубым фоном, уже присутствовала в ПАК версии 4.1 и использовалась для автоматической конфигурации. Именно она позволила малыми затратами не только обеспечить совместимость со старшей реализацией IPSec из версии 3.2, но и заложить потенциал для обеспечения совместимости с будущими версиями.

Преимущества и недостатки

В заключении рассмотрим основные преимущества и недостатки метода автоматической генерации исходных текстов программ:

+ Возможность для управления спецификациями ПО непосредственно аналитиком минуя программиста. Это приводит к снижению числа ошибок, характеризуемых отклонением ПО от спецификаций.
+ Упрощение обеспечения совместимости различных версий ПО за счёт возможности генерации эквивалентного кода либо на разных языках программирования, либо с использованием разных технологий.

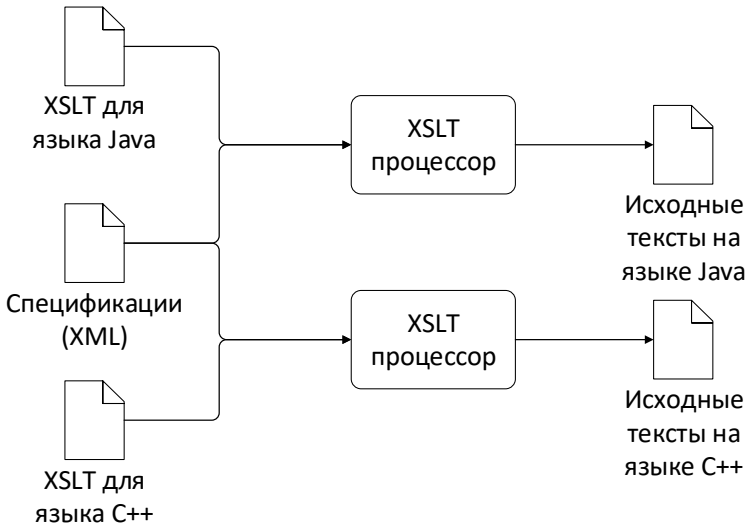

Рисунок 5: Схема применения XSLT для генерации исходных текстов на разных языках

+ На основе спецификаций можно генерировать не только исходный код, но и автоматические тесты ПО. Это удобно для поддержки тестов в актуальном состоянии при внесении изменений в спецификации.

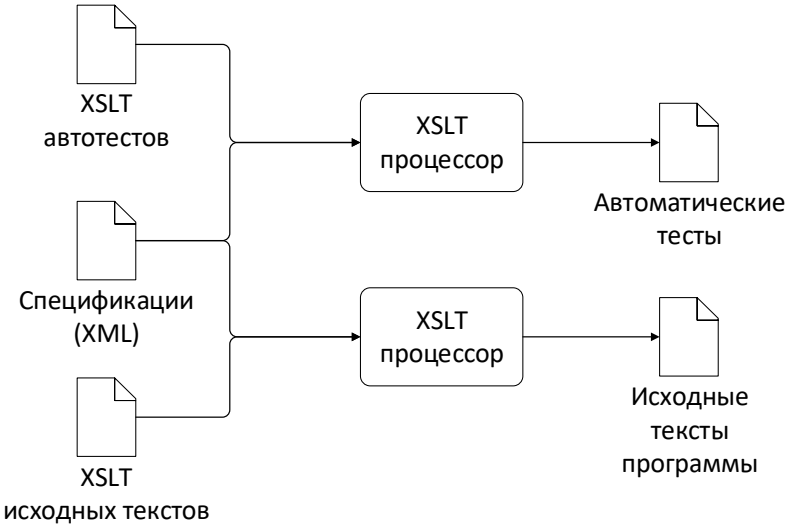

Рисунок 6: Схема применения XSLT для генерации автоматических тестов

+ В некоторых случаях (например, в описанной задаче №2), применение этой методики позволяет сразу сократить сроки реализации требований. Однако, и в общем случае применения метода сокращает сроки разработки за счёт упрощения корректировки спецификаций в перспективе.

– Не каждая задача может быть решена описанным методом. Для него подходят лишь определённые хорошо формализуемые задачи.
– Усложнение процедуры сборки ПО как следствие добавления новой стадии – трансляции XML-деклараций в исходный код.
– Появление новых типов потенциальных ошибок, которые нужно учитывать при реализации. Например, файл спецификации может содержать лексические ошибки.

Другой пример – реализация может быть рассчитана на определённый порядок деклараций в файле спецификации, и при его изменении будет генерироваться ошибочный код.