РАЗРАБОТКА ДОПОЛНЕНИЯ (ПЛАГИНА) К MICROSOFT VISUAL STUDIO 2005 ДЛЯ СИНТАКСИЧЕСКИ-ОРИЕНТИРОВАННОЙ ФИЛЬТРАЦИИ СООБЩЕНИЙ ОБ ОШИБКАХ КОМПИЛЯЦИИ C++

Татьяна Борисова[*]
10 класс, лицей «Вторая школа», Москва

Научный руководитель: И. Р. Дединский, лицей «Вторая школа», Москва

При написании программ на C++ с использованием шаблонов или STL-классов у разработчиков часто возникает проблема, связанная с тем, что компилятор выдает слишком большое сообщение об ошибке, называемое ошибкой-романом. Для облегчения анализа таких сообщений был написан синтаксически-ориентированный фильтр для сообщений об ошибках шаблонов C++, представляющий ошибки-романы в более удобном и читаемом виде – в виде синтаксического дерева. Он выполняет синтаксический анализ вывода компилятора и сравнивает синтаксическое дерево сообщения об ошибке с деревом возможно правильного варианта (если таковой предоставляется компилятором).

Для взаимодействия с пользователем был написан плагина к популярной среде разработки Visual Studio 2005, который представляет вывод компилятора в виде дерева с возможностью разворачивать/сворачивать дерево, переходить к интересующему месту, прослеживать, где именно произошло несоответствие с возможным правильным вариантом.

Актуальность и постановка задачи

При написании программ на языке C++, особенно с использованием шаблонов или STL-классов, у разработчиков часто возникает проблема, связанная с тем, что компилятор выдает слишком большое сообщение об ошибке, обычно называемое ошибкой-романом. Его размер может достигать нескольких десятков тысяч символов. Зачастую оно бывает очень запутанным и плохо поддается анализу, в результате чего поиск ошибок компиляции в коде на C++, использующем шаблоны, становится крайне затрудненным. Такое происходит не только при написании сложных систем шаблонных классов, но и в относительно небольших программах. Обычный пропуск инициализатора шаблонного класса (листинг 1) может вызвать появление сообщения об ошибке в Visual Studio 2005, размер которого 21 строка (рис. 1). И это один из самых безобидных примеров.

```
template <typename Source>
IncludeTypes (Source& s)
    : CurType::AddMembers < IncludeTypes<BaseType/*, Tail*/> > (s)
    {}
```

Листинг 1. Код на *C*++, вызывающий появление слишком большого сообщения об ошибке.

[*] E-mail для связи: tanyatik@yandex.ru.

Другая особенность этих ошибок (кроме размера) заключается в том, что сообщение об ошибке часто указывает не на то место, где пользователь ее допустил. Для того чтобы выяснить, где, собственно, произошла ошибка, нужно долго и тщательно анализировать сообщение транслятора. Такое, в частности, характерно для ошибок компиляции с *несоответствиями,* т.е. таких ошибок, где не возникает соответствия типа (или другой сущности языка) ни одному из возможных вариантов. Например, к таким ошибкам относятся несоответствие типа передаваемого аргумента прототипу функции или отсутствие функции в семействе перегруженных функций. Иногда в этом случае компилятор предлагает возможный правильный вариант кода.

Цель работы

Представление «ошибок-романов» в более удобном и читаемом виде — в виде синтаксических деревьев.

Требования к программному продукту

1. Выделение сообщений об ошибках, содержащих шаблонный код, из лога компиляции;
2. Синтаксический анализ участков сообщений об ошибке компиляции, в которых содержится код на C++;
3. Представление этих участков кода в виде дерева;
4. Сравнение синтаксических деревьев для ошибок с несоответствиями;
5. Выделение цветом и пиктограммами тех участков синтаксических деревьев, в которых конкретно произошло несоответствие.

Обзор и анализ существующих решений проблемы

Существует утилита под названием *STLFilt* **(***http://www.bdsoft.com/tools/stlfilt.html***),** которая представляет собой скрипт, написанный на Perl, форматирующий стандартный вывод компилятора и скрывающий некоторые детали инициализации шаблонных классов. Результат работы утилиты — стандартный вывод компилятора, форматированный отступами (так, как это делают в коде программисты на C++, см. листинг 1). Детали инстанцирования шаблонов STL игнорируется. Функциональности этой утилиты недостаточно, так как STLFilt не выполняет сравнение синтаксических деревьев и никак не выделяет участки синтаксических деревьев, в которых произошло несоответствие (см. табл. 1).

Таблица 1. Сравнение возможностей *STLFilt* и *TemErFil*

Возможность	***STLFilt***	***TemErFil*** **(данная работа)**
Синтаксический анализ сообщений об ошибках	Да	Да
Структурирование участков кода сообщения об ошибке	Да	Да
Динамическое представление структурированной информации об ошибке	Нет	Да
Сравнение синтаксических деревьев для ошибок с несоответствиями	Нет	Да
Выделение цветом тех участков, где произошло несоответствие	Нет	Да

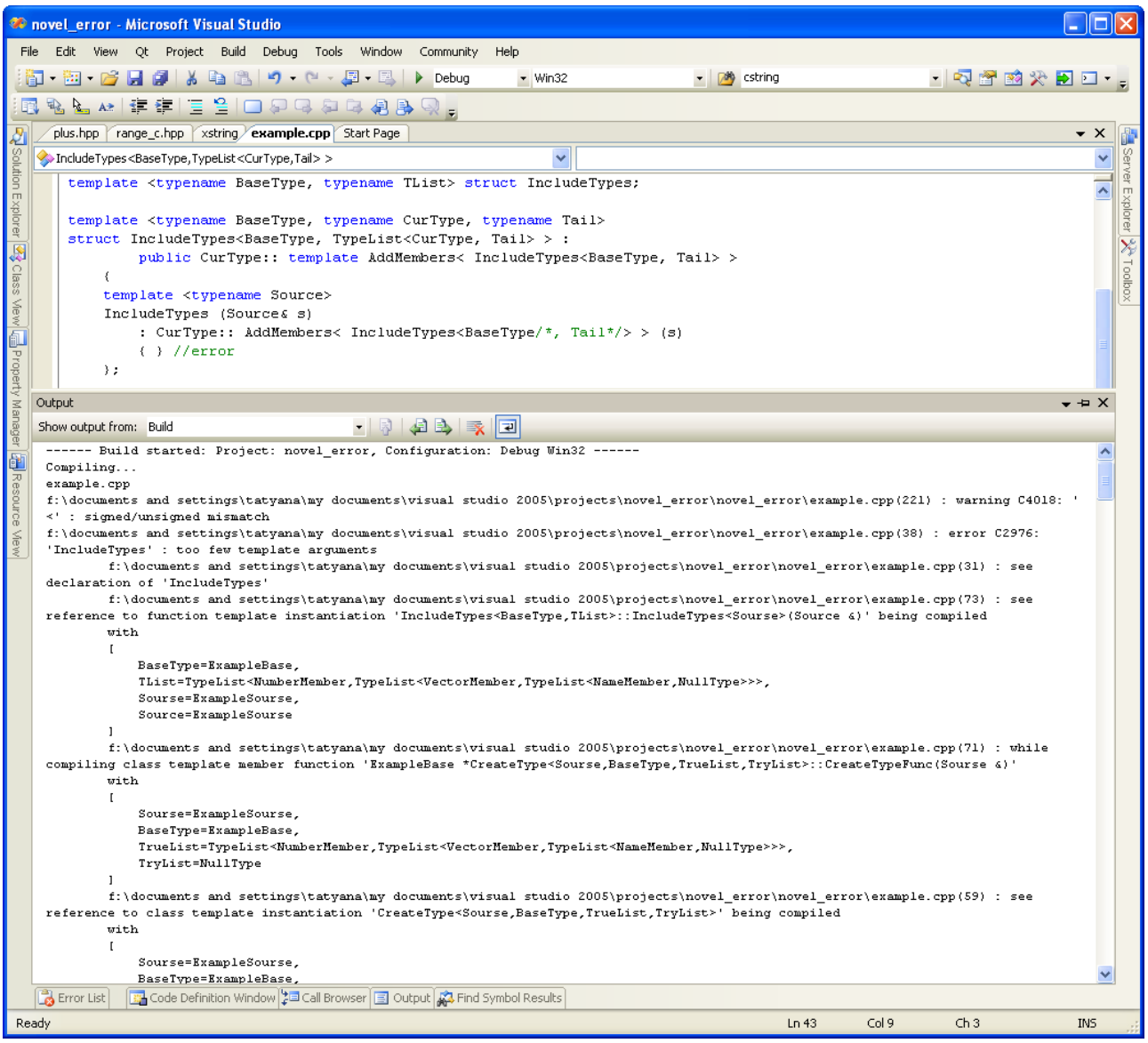

Рис. 1. Скриншот вывода компилятора.

```
meta2.cpp:
e:\boost\boost\mpl\plus.hpp(53): error C2825: 'T1::value': cannot form a
    qualified name
e:\boost\boost\mpl\plus.hpp(66) : see reference to class template
instantiation
   'boost::mpl::plus<
        boost::mpl::aux::fold_backward_impl<
            2, boost::mpl::aux::fold_backward_impl<
                6, boost::mpl::aux::fold_backward_impl<
                    10
                  , boost::mpl::begin<boost::mpl::range_c<long, 0, 10>
>::type
```

Листинг 1. Пример вывода компилятора, форматированного с помощью STLFilt (фрагмент).

Постановка задачи

1. Разработка синтаксического анализатора, соответствующего изложенным выше требованиям:

    1.1. Разработка алгоритма поиска интервалов сообщения об ошибке и участков кода, для которых нужно выполнить сравнение;

1.2. Разработка алгоритма синтаксического разбора интервалов сообщения об ошибке;

1.3. Разработка алгоритма сравнения синтаксических деревьев для ошибок с несоответствиями.

2. Разработка *Microsoft Visual Studio Integration Package* (дополнение для Microsoft Visual Studio 2005) для взаимодействия с пользователем:

2.1. Представление сообщений об ошибках в виде синтаксического дерева с возможностью динамического представления структурированной информации о сообщении об ошибке (возможность сворачивать/разворачивать дерево, переходить к интересующему участку дерева);

2.2. Выделение цветом тех участков сравниваемых синтаксических деревьев, где произошло несоответствие.

Описание методов решения задачи

Синтаксический анализатор

Синтаксический анализатор производит синтаксический анализ участков C++ кода в сообщениях об ошибке и сравнение синтаксических деревьев. Эта часть написана на языке C++, так как это – высокоуровневый, быстрый и вместе с тем подходящий для системного программирования язык.

Хранение данных

Для хранения данных в модуле синтаксического анализа были использованы контейнеры из библиотеки *STL (Standard Template Library),* которая была надстроена небольшой, более удобной и компактной оболочкой классов, входящих в библиотеку *beaver*. Эта библиотека уже была представлена на Балтийском научно-инженерном конкурсе в 2008 году в проекте «Динамический анализатор *C++* кода». Библиотека расширяет функциональность контейнеров STL, позволяет сокращать запись и делает работу с *STL* более удобной. Также в библиотеке имеются «умный указатель», который легко взаимодействует с другими умными указателями (сравнивается, присваивается), и класс-инициализатор стандартных типов (начальное значение по умолчанию указывается в качестве параметра шаблона).

Поиск интервалов сообщения об ошибке

Вначале анализатор получает текст всех сообщений об ошибках (содержимое *Output*) от плагина.

Затем он определяет *интервал* (интервал – пара начало-конец) каждого из сообщений об ошибках. Это делается путем поиска *ключевых слов* (ключевое слово – слово, которое, будучи обнаруженным синтаксическим анализатором вне интервала кода на C++, однозначно определяет наличие ошибки компиляции или ее тип.).

В каждом сообщении об ошибке есть фразы на естественном языке (английский) и фрагменты кода на C++, которые находит и сохраняет анализатор. Фрагменты кода на C++ заключены в кавычки (“ ’ ”). После фрагментов кода иногда следует информация об инстанцировании шаблонов (заключенная в квадратные скобки: «*with* […]»). Эту информацию анализатор также сохраняет (путем поиска по ключевому слову *with*). Далее анализатор определяет, есть ли в ошибке несоответствия. Это делается по следующему алгоритму:

1. Если найдены ключевые слова *from*, *to* – то это ошибка, возникающая при невозможности конвертации одного типа (или другой сущности языка) в другой (пример: листинг 3).

2. Если ошибка найдена по ключевым словам *could be, and* – то это ошибка, для которых для ошибочного кода существует несколько возможно правильных вариантов кода. (Пример: листинг 4).

```
1>w:\projects\errorsparser_debug\errorintervals.cpp(50) : error C2664:
'std::basic_istream<_Elem,_Traits>
&std::basic_istream<_Elem,_Traits>::seekg(std::fpos<_Statetype>)' : cannot
convert parameter 1 from 'beaver::StreamPosition' to 'std::fpos<_Statetype>'
1>        with
1>        [
1>            _Elem=char,
1>            _Traits=std::char_traits<char>,
1>            _Statetype=_Mbstatet
1>        ]
1>        and
1>        [
1>            _Statetype=_Mbstatet
1>        ]
1>        No user-defined-conversion operator available that can perform
this conversion, or the operator cannot be called
```

Листинг 3. Пример ошибки, возникающей при невозможности конвертации одного типа в другой.

```
1>w:\projects\novel_error\!main.cpp(24) : error C2665: 'f' : none of the 2
overloads could convert all the argument types
1>        w:\projects\novel_error\!main.cpp(16): could be 'void f<C<T>>(P)'
1>        with
1>        [
1>            T=B<A<X>>
1>        ]
1>        w:\projects\novel_error\!main.cpp(19): or 'void f<C<T>>(Q)'
1>        with
1>        [
1>            T=B<A<X>>
1>        ]
1>        while trying to match the argument list '(X)'
```

Листинг 4. Пример ошибки, для которой для ошибочного кода существует несколько возможно правильных вариантов кода

Необходимо учитывать, что в одной ошибке компиляции может быть несколько несоответствий.

Результат работы этот компонент компилятора сохраняет в структурах данных, представленных на рис. 2.

Основные структуры данных этого компонента

class CodeInterval

Вспомогательный класс для класса *ErrorIntervalPart*. Хранит интервал кода на C++ и прилегающий к нему интервал инстанцирования шаблонов («*with* […]»).

class ErrorIntervalPart

Класс, представляющий интервалы части сообщения об ошибке – интервал кода (*CodeInterval*), называемый *эталонным*, и один или несколько интервалов кода, называемых *сравниваемыми*. Дерево, соответствующее эталонному интервалу кода, будет затем сравниваться с каждым из деревьев, соответствующим сравниваемым интервалам.

Сообщение об ошибке представляется набором частей сообщения об ошибке.

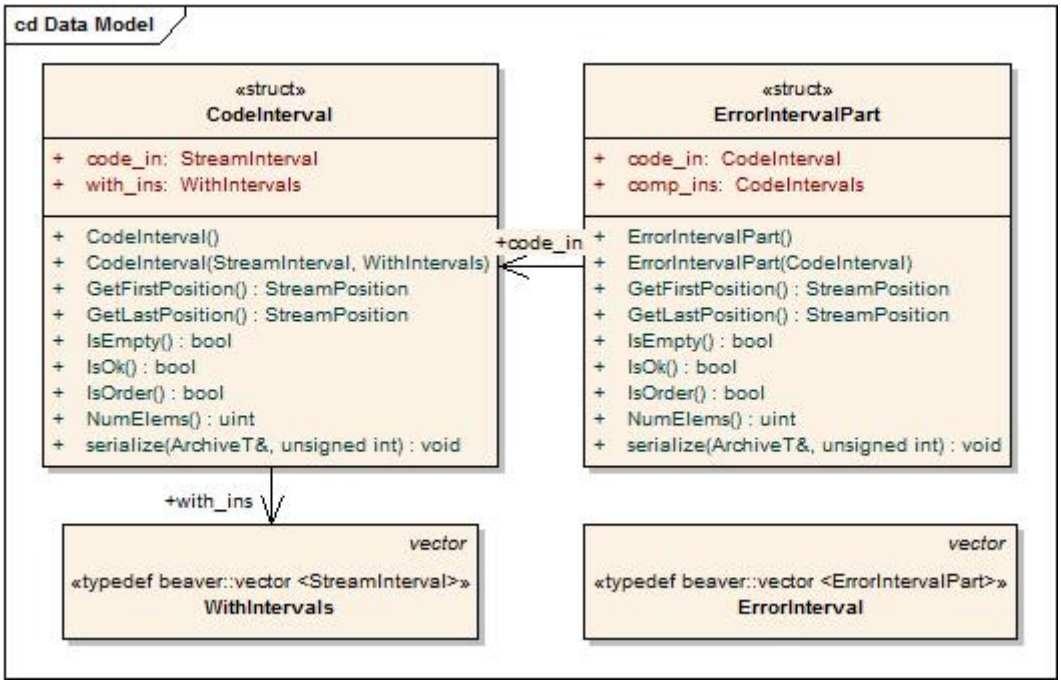

Рис. 2. Диаграммы основных структур данных модуля поиска интервалов.

Синтаксический анализ интервалов сообщения об ошибке

Синтаксический анализатор получает на входе интервалы C++ кода (без интервалов инстанцирования шаблонов) и осуществляет синтаксический анализ этого кода. Синтаксический анализ реализуется с помощью алгоритма рекурсивного спуска, наиболее подходящего для данной задачи.

Краткое описание алгоритма синтаксического анализа

1. Первое встретившееся слово сохраняется в текущем узле дерева.
2. Если встретился символ перехода на следующий уровень (например, «::», «<». «(»), весь текст этого уровня сохраняется в текущем узле дерева.
3. Если встречается символ перехода на следующий уровень, создается новый дочерний узел в дереве и алгоритм переходит на шаг 1.
4. Если встречается символ перехода на предыдущий уровень (например, «>», «)»), алгоритм возвращается на родительский уровень текущего узла дерева и переходит на шаг 1.

В результате анализатор получает синтаксическое дерево кода сообщения (класс *ErrorMessageTree*).

Основные структуры синтаксического анализатора представлены на рис. 3.

Основные структуры данных синтаксического анализатора:

class ErrorMessageTree

Представляет узел дерева. Хранит лексему (текст), соответствующую этому узлу, и набор дочерних деревьев (экземпляров класса *TreeBranch*).

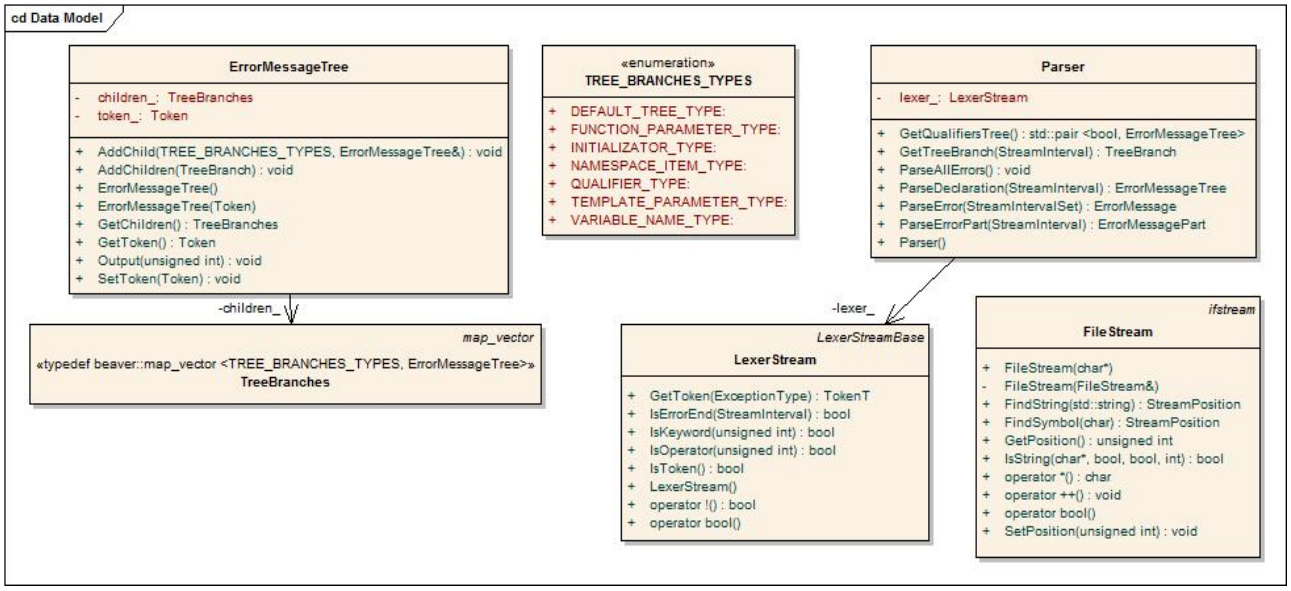

Рис. 3. Диаграммы основных структур данных синтаксического анализатора

typedef TreeBranches

Представляет множество *веток* дерева – набор дочерних деревьев с указанием типа параметра всех этих дочерних деревьев. Тип параметра (*enum TREE_BRANCHES_TYPES*) – это, например, шаблонный параметр, имя переменной, член пространства имен.

class Parser

Занимается синтаксическим анализом. Содержит все основные функции, занимающиеся синтаксичеким анализом.

class LexerStream

Оболочка для библиотечного класса *beaver*::*lex*::*lexer*.

Для лексического анализа использовалась библиотеки *beaver*::*lex*. Она уже была представлена на Балтийском научно-инженерном конкурсе в 2008 году в проекте «Динамический анализатор C++ кода». Библиотека написана с применением модульности и технологий обобщённого программирования. Это даёт возможность задавать извне библиотеки тип лексем, поток данных (файл или любой объект, предоставляющий нужное API: получение текущего символа, переход к следующему символу). Также лексический анализатор можно надстраивать извне дополнительными модулями (можно написать отдельный модуль для разбора какого-то типа лексем - слов или операторов, или написать модуль, пропускающий комментарии).

class FileStream

Оболочка для класса *std*::*ifstream*, предоставляющая необходимое API для лексического анализатора (класса *LexerStream*). Помимо необходимой функциональности, класс предоставляет возможность поиска в файле конкретного символа или строки, а также некоторые другие возможности.

Сравнение синтаксических деревьев

Этот компонент получает на входе синтаксическое дерево, называемое *эталонным*, и множество синтаксических деревьев, называемых *сравниваемыми*. Эталонное дерево необходимо сравнить с каждым сравниваемым.

Краткое описание алгоритма сравнения

1. Сравнение выполняется для коренных узлов двух деревьев (эталонного и одного из сравниваемых) – сравнивается такст, содержащийся в этих узлах. Если сравнение перешло неудачно, алгоритм переходит к шагу 3. Если удачно – к шагу 2.
2. Шаг 1 выполняется для всех дочерних узлов обоих деревьев.
3. Узел сравниваемого дерева, для которого сравнение с соответствующим узлом эталонного дерева прошло неудачно, помечается. В дальнейшем эта информация будет использована плагином.

Системы контроля программных ошибок

Проверка корректности внутренних данных

Для проверки корректности данных у каждого класса имеется функция, проверяющая корректность внутренних данных класса. Эта функция вызывается у всех параметров и у возвращаемого значения в каждой функции. Это позволяет быстро выявить, в каком модуле и в каком алгоритме произошла ошибка.

Журнал отладочной информации

Для облегчения отладки по завершению работы каждого модуля производится сброс информации об его данных в журнал отладочной информации. В случае возникновения ошибки в программе эта информация используется для обнаружения ошибки. Журнал отладочной информации вместе со входными данными программы (текстом вывода компилятора) пересылается автору программы.

Сброс данных в журнал отладочной информации был реализован с помощью библиотеки *log*, разработанной с применением технологий обобщённого программирования (с использованием стратегий, классов-свойств, частичных шаблонных специализаций). Это было сделано для повышения настраиваемости и гибкости библиотеки. Она уже была представлена на Балтийском научно-инженерном конкурсе в 2008 году в проекте «Динамический анализатор *C++* кода».

Unit-тесты модулей.

Для проверки корректности работы отдельных модулей синтаксического анализатора использовался метод тестирования, известный как unit-тестирование (англ. *unit testing*). Unit-тесты для каждого модуля подают модулю заранее известные входные данные и сравнивают вывод этого модуля с заранее известными выходными данными. Если вывод модуля является верным для данного набора входных данных, unit-тест прошел успешно.

Unit-тесты позволяют модифицировать код с возможностью быстро проверить, не привело ли очередное изменение кода к появлению ошибок в уже написанных и оттестированных частях программы, а также облегчает устранение таких ошибок. Также они позволяют отслеживать время выполнения программы для проверки того, что изменение кода не привело к обратной оптимизации.

Алгоритм unit-тестирования модуля программы:

1. Считывание из файла настроек unit-тестирования данного модуля имен файлов, в которых хранятся входные данные для данного модуля и соответствующие им корректные выходные данные.
2. Считывание корректных выходных данных программы и инициализация ими выходных структур данных модуля.
3. Запуск модуля с входными данными.
4. Проверка соответствия выходных данных модуля с корректными выходными данными, загруженными на шаге 2.
5. Повторение шагов 2-4 для каждой пары входных данных и корректных выходных данных.

При обнаружении набора данных, на котором модуль программы работает корректно, можно автоматически сохранить выходные данные модуля в файл. Сохранение выходных данных в файл и их загрузка из файла реализованы с помощью библиотеки *boost*::*serialization*.

Дополнение к Visual Studio

Дополнение к Visual Studio (*Microsoft Visual Studio Integration Package, VSIP*) – это часть проекта, отвечающая за взаимодействие с пользователем. Оно демонстрирует синтаксическое дерево каждой ошибки, визуально выделяет причину ошибки (участки сравниваемых деревьев, где они не совпадают). Дополнение написано на языке *C# .NET*, так как этот язык программирования хорошо подходит для написания пользовательского интерфейса, к тому же он лучше поддерживается в Visual Studio 2005 SDK. Основные структуры данных дополнения к Visual Studio представлены на рис. 4.

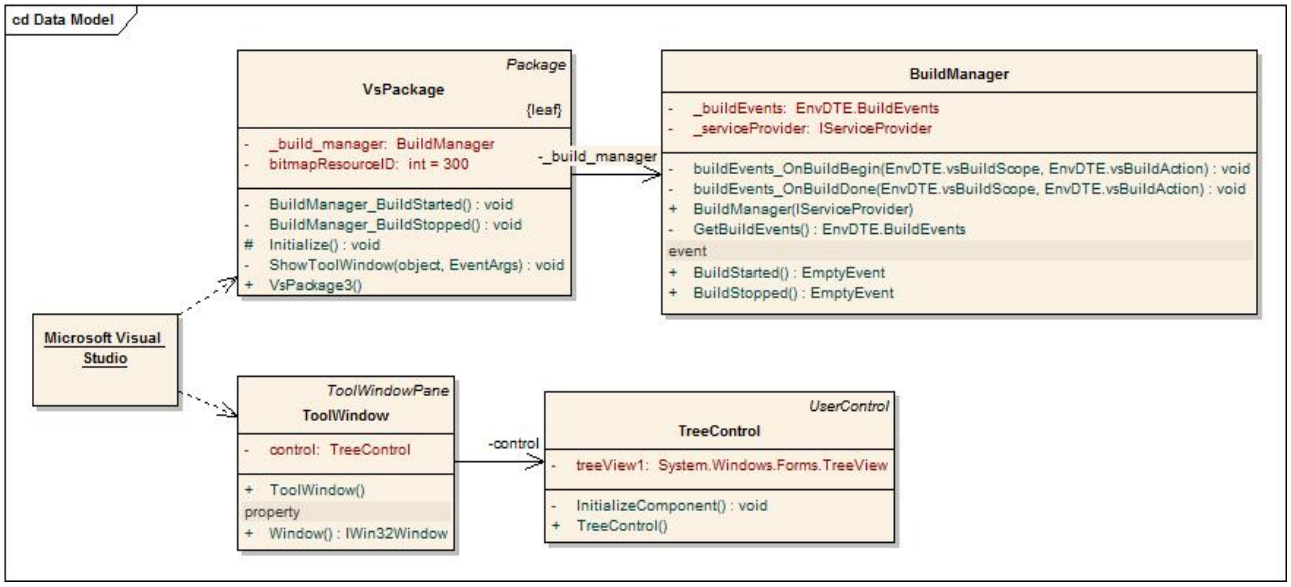

Рис. 4. Диаграмма основных структур данных дополнения к Visual Studio.

Основные структуры данных дополнения к *Visual Studio*

class VsPackage

Этот класс регистрирует плагин в Visual Studio. Он реализует интерфейс *IVsPackage*, реализуя интерфейс *Microsoft.VisualSrudio.Shell.Package*. Каждый плагин к Visual Studio

должен реализовывать интерфейс *IVsPackage* и зарегистрироваться в оболочке Visual Studio. Для упрощения процесса регистрации плагина используется интерфейс *Package*.

class ToolWindow

Этот класс представляет *ToolWindow* плагина (*ToolWindow* – особый тип окна в Visual Studio IDE, который можно убирать или закреплять сбоку экрана), на котором будут отображаться элементы управления.

class TreeControl

Класс *TreeControl* представляет элемент управления (control), отображающий стандартную форму *TreeView* и заполняющий ее после завершения синтаксического анализа. Стандартная форма *TreeView* (класс *System.Windows.Forms.TreeView*) используется для динамического представления структурированной (в виде дерева) информации о сообщении об ошибке (см. рис. 5, 6). Эта форма позволяет разворачивать и сворачивать узлы дерева, так что пользователь может сам выбрать, какую информацию об ошибке он хочет просматривать. Также она позволяет установить произвольные иконки для узлов дерева. Сделав специальные иконки для тех узлов сравниваемых деревьев, для которых сравнение прошло неудачно, можно продемонстрировать пользователю, где конкретно произошла ошибка несоответствия (пользователь может проследить это место по специальным иконкам).

class BuildManager

Для того чтобы предоставлять пользователю актуальную информацию об ошибках компиляции, необходимо запускать синтаксический анализ каждый раз после того, как собирается пользовательский проект. Этот класс получает доступ к классу *EnvDTE*, с помощью которого плагин подписывается на событие о завершении сборки пользовательского проекта – событие (event) *OnBuildEndEventHandler*.

Взаимодействие с модулем синтаксического анализа

Модуль синтаксического анализа реализован в виде динамически подключаемой библиотеки (DLL). Это позволяет, во-первых, совместить в одной программе код на двух разных языках программирования (C# .NET и C++), а, во-вторых, производить «горячую замену» модуля синтаксического анализа. Этот модуль больше подвержен ошибкам, чем дополнение к Visual Studio, следовательно, и обновляться он будет чаще.

Результаты работы. План дальнейшего развития.

В результате работы был разработан синтаксически-ориентированный фильтр, осуществляющий синтаксический анализ участков сообщений об ошибке компиляции, в которых содержится код на C++, представляющий эти участки кода в виде деревьев и сравнивающий синтаксические деревья для ошибок с несоответствиями. Для отображения деревьев был разработан плагин к Microsoft Visual Studio 2005, визуально выделяющий несовпадающие участки сравниваемых деревьев.

В дальнейшем планируется:

1. Пробное внедрение программы в кабинете программирования среди школьников;
2. Выкладывание программы в сеть *Internet* для свободного использования;
3. Написание плагинов для других сред разработки (*Eclipse*, *Code*::*Blocks*).

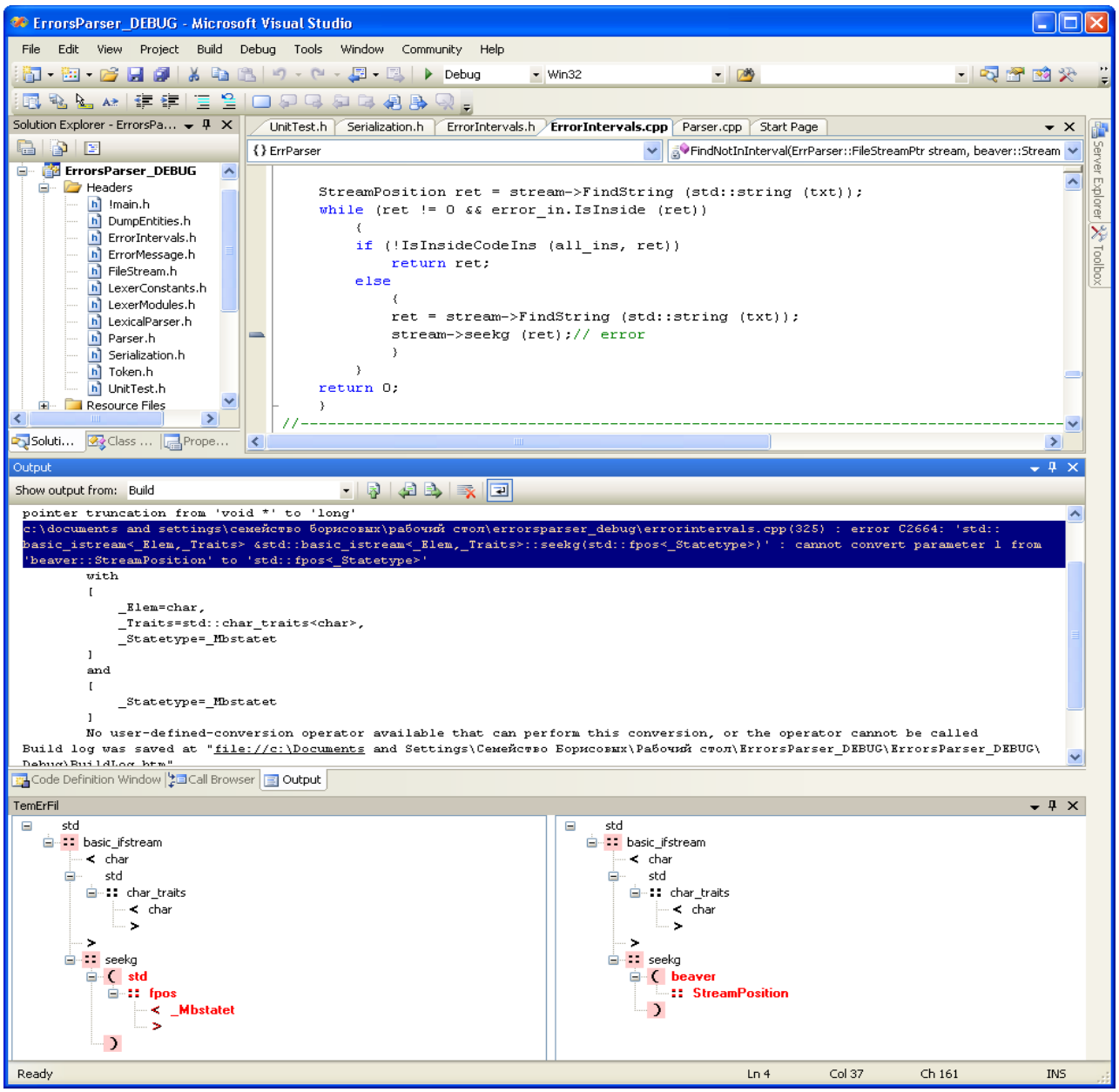

Рис. 5. Панель фильтра TemErFil в среде Visual Studio (скриншот).

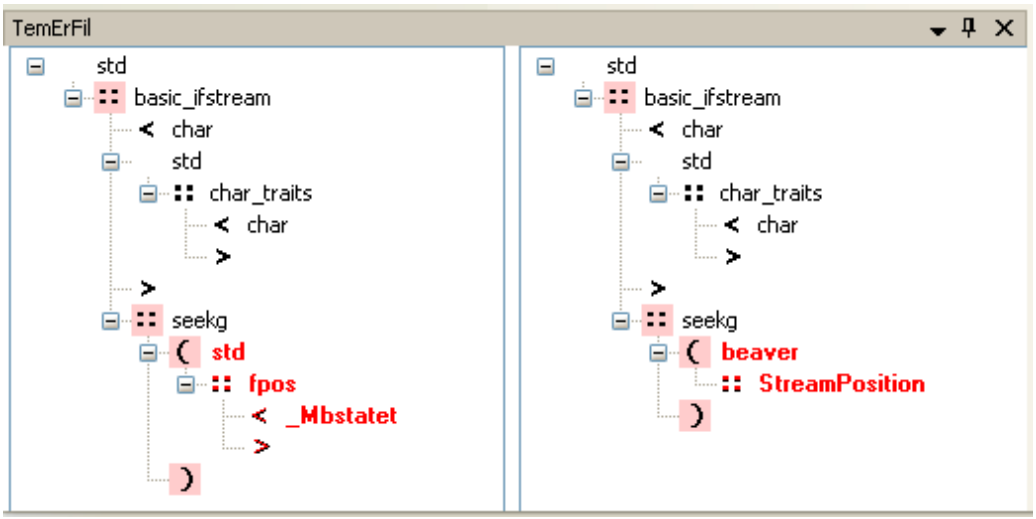

Рис. 6. Скриншот работы плагина TemErFil.

Используемая литература

1. Прата, Стивен. Язык программирования C++. Лекции и упражнения:– 5-е изд. – М.: “И.Д.Вильямс”, 2007
2. Шилдт, Герберт. Самоучитель C++: – СПб.: БХВ-Петербург, 2005. – 688 с.
3. Хантер, Робин. Основные концепции компиляторов.: – М.: “И.Д.Вильямс”, 2007. – 256 с.