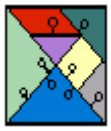

A Reusable Syntax Directed Processing System

Author: Chuck Strempler

Microsoft Corporation

Microsoft Research – ComApps

chuckstr@microsoft.com

Last Updated: March 18, 1999

Abstract: This paper discusses a system under development that takes a representation of any LL(k) language [1] and some source text written in that language and outputs a component hierarchy representing that source. This hierarchy may an be intermediate representation of the language or a stand-alone component composite.

To understand this paper it will be useful to have knowledge of Microsoft COM technology [2]. It will also be useful to have knowledge of our hierarchical component composition technology called assemblies [3].

1. Syntax Directed Processing and Component Hierarchies

In our research, we've come across many instances requiring interpreting a language or translating a language into another language. The following is a list of some of the problems we've encountered that may be addressed using syntax directed processing:

- C+Com Compiler- We use this tool to automate component generation and production of our component type libraries for use in our assembly tools. It translates annotated C files into C code and component type source files.
- C+Com source wizard - This tool allows declarative specification of component attributes and generates C+Com source files. It builds up an intermediate representation of a C+Com source file and dynamically alters that representation as component attributes are added or changed.
- IDL+Com Compiler- This tool allows type repositories to be built containing extended information unavailable through normal interface definition processing. It translates a type specification language into an IDL file and component type source files.
- A component wiring specification language - This could be an interpreted language that dynamically wires components together.

- Precise interface specification - This could be used to generate interface test and precondition checking code from a rigorous specification.
- Form layout specification - This could be used to generate user interface assemblies to implement complex interfaces.

Analysis of these problems yielded the following list of requirements for a system that facilitates syntax directed processing:

- It must be reusable. That is, is must be usable for language-processing across varied domains. This includes flexible processing of ambiguous language constructs.
- It must be extensible. That is, allow for easy algorithm substitution in and augmentation of the various phases of processing.
- It must be able to generate an intermediate representation for compiling and interpreting and a stand-alone composition for assembly creation.
- It must support nested invocation for mixed languages.
- It must support seamless error reporting.

Although existing syntax directed systems such as lex and yacc addressed some of these requirements, we found the most flexible, complete solution was to develop a set of components each addressing a particular phase of syntax directed processing, and use our assembly technology to combine them into an extensible, reusable composite currently known as SDPAssy. SDPAssy generates either a component hierarchy as the intermediate language representation or the language directed component hierarchy assembly.

2. Using SDPAssy For Syntax Directed Processing

The following sections describe how the SDPAssy assembly addresses the above requirements and how to use it to build assemblies, compilers and interpreters.

2.1 The SDPAssy Assembly

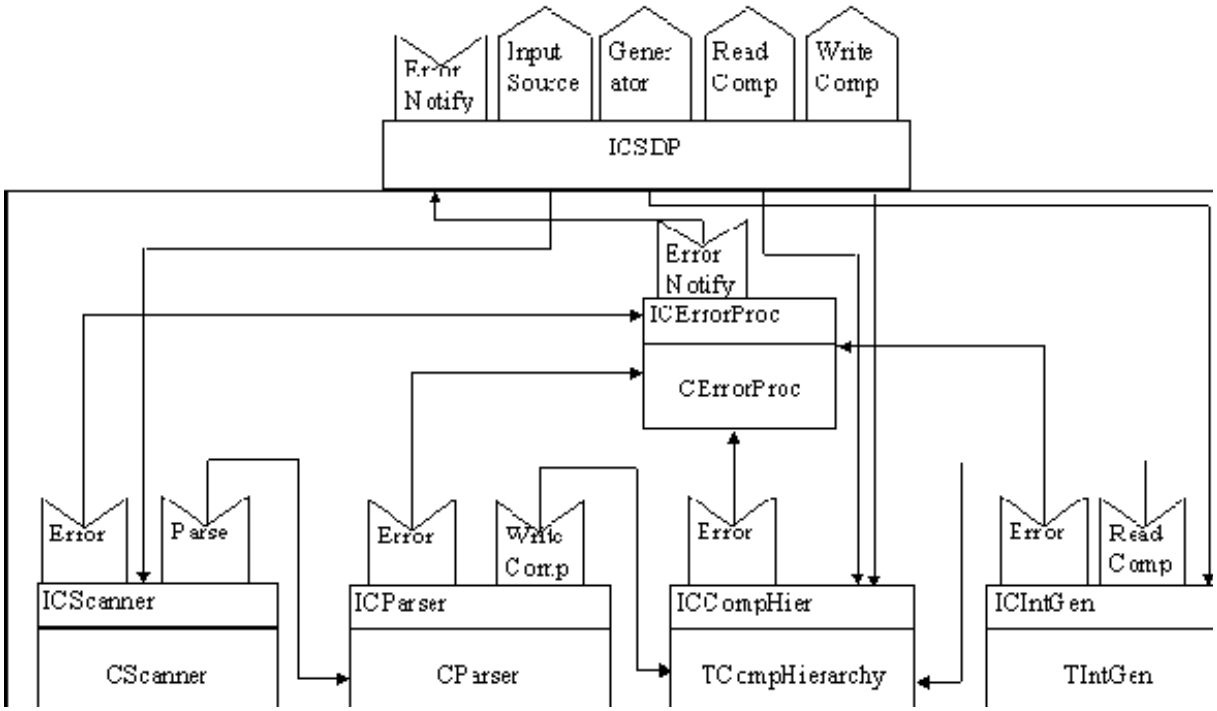

Figure 1 , The SDPAssy assembly. Please see [3] for a description of assemblies.

SDPAssy itself imports an error notification callback interface and exports an interface on the scanner to specify the input source text, an interface on the generator or interpreter and interfaces on the component hierarchy element to navigate and alter the hierarchy. Altering the hierarchy may be required for applications such as the C+Com source generator described in Section 2.1. During language processing, another instance of SDPAssy can be invoked to parse a contained language. This arises, for example, in the C+Com compiler described in Section 2.1. C+Com files can import IDL+Com files to extract type information from them.

The error handling element, CErrorProc, receives a pointer to an interface outside the assembly through the ErrorNotify role, that is called back with a formatted error message and an error code when an error occurs. Other top level elements (those listed in the diagram above) receive a pointer to CErrorProc so that they can report errors.

The lexical analysis component, CScanner, is called from outside the assembly via the Input Source pin, with the language source text to be tokenized. It calls CParser to process each token. CScanner is specialized with a binary representation of the regular-expression grammar describing the tokens in the source language and a keyword table. This binary representation is created once per token set outside SDPAssy. This allows CScanner to reuse this binary representation to efficiently generate tokens for any instances of source text with the given token set.

CParser generates a component hierarchy from the token stream. It is a predictive parser that will recognize any LL(k) [1] language annotated with commands that tell the parser to have the component hierarchy element add a new component to the hierarchy and/or call a method on an existing component. CParser is specialized with a binary representation of the predictive parse table for the source language. This binary representation is created once per language outside the SDPAssy assembly. This allows parsing arbitrary source text in the source language without reanalyzing the language each time. The language can also be annotated with commands that allow the parser to call some custom code to handle ambiguous grammar conditions. One place where this is needed is in the C+Com compiler described in Section 2.1. The C+Com grammar makes use of nested braces and matching a given '}' with the correct '{' is required. Since the grammar representing this match is ambiguous, it cannot be handled by the parser automatically. The C+Com language annotates the grammar with a reference to code that tracks the correct brace to match and can resolve the ambiguity.

The TCompHierarchy element is a placeholder instantiated at assembly initialization time with an actual component hierarchy implementation. This allows applications with differing intermediate representation or assembly generation requirements to reuse the parser and scanner implementations while providing a custom or extended component hierarchy element. The root component in the hierarchy, must implement interfaces to traverse and alter the hierarchy as well as an interface (IParseHelper) that allows communication between the hierarchy components and the parser. Other components in the hierarchy (i.e. those that are not used to specialize the TCompHierarchy place holder), may only implement IParseHelper. The example presented in Section 2.2 shows how the parser calls IParseHelper methods to direct the creation of the component hierarchy. It is worth noting that in assembly generation, the component hierarchy is the final product and the generator/interpreter is not needed. Currently, we only use domain specific hierarchy elements. Because the hierarchy models language entities, new components must be written for every language in which these entities are different. It seems plausible to have a reusable hierarchy component that keeps track of children by type and properties by name. This would allow a user of SDPAssy to write only a custom generator or interpreter and not worry about the intermediate representation at all.

The TIntGen element is also a placeholder that is instantiated with the actual interpreter or generator implementation at assembly initialization time. This component receives a pointer to the root of the component hierarchy, and can then navigate the hierarchy through private interfaces.

2.2 A Simple Example Using SDPAssy

This example illustrates the use of SDPAssy to perform syntax directed processing with a simple

language. The language is a simple form description language and is presented in annotated BNF form with terminals in caps, non terminals lower case and annotations in bold. The annotations are not part of the grammar itself but meta language tags that direct the parser to perform some action. They are attached to their following terminals or non terminals in the parse table so that the parser operates only on grammar symbols. # indicates the beginning of a compound annotation, **$** indicates to the parser that the most recently created hierarchy component should be made current and may annotate non terminals, **CREATE <*clsid*>** causes a new hierarchy component of type <clsid> to be created and passes this as an argument to the current hierarchy component's IParseHelper::Create call. All other annotations are tag names which are passed as arguments to the current hierarchy component's IParseHelper::Call along with the token being processed.

(1)forml -> **#CREATE <*formclsid*>** FORM **#$NAME** STRING **#$X** INT **#$Y** INT BEGIN **$**fieldl END forml

(2)forml -> null

(3)fieldl -> **#CREATE <*buttonclsid*>** BUTTON **#$NAME** STRING **#$X** INT **#$Y** INT **#$FORM** STRING fieldl

(4)fieldl -> **#CREATE <*labelclsid*>** LABEL **#$NAME** STRING **#$X** INT **#$Y** INT fieldl

(5)fieldl -> **#CREATE <*editclsid*>** EDIT **#$NAME** STRING **#$X** INT **#$Y** INT fieldl

(6)fieldl -> null

Here is a very small example of the language:

```
form Form1 10 10
begin
button Button1 0 0 Form2
end
Form Form2 10 10
begin
label "Look at this:" 0 0
edit Edit1 100 0
end
```

In this example, the TCompHierarchy is initialized with CFormRoot. The result of processing the sample form description language is the assembly in Figure 2a.

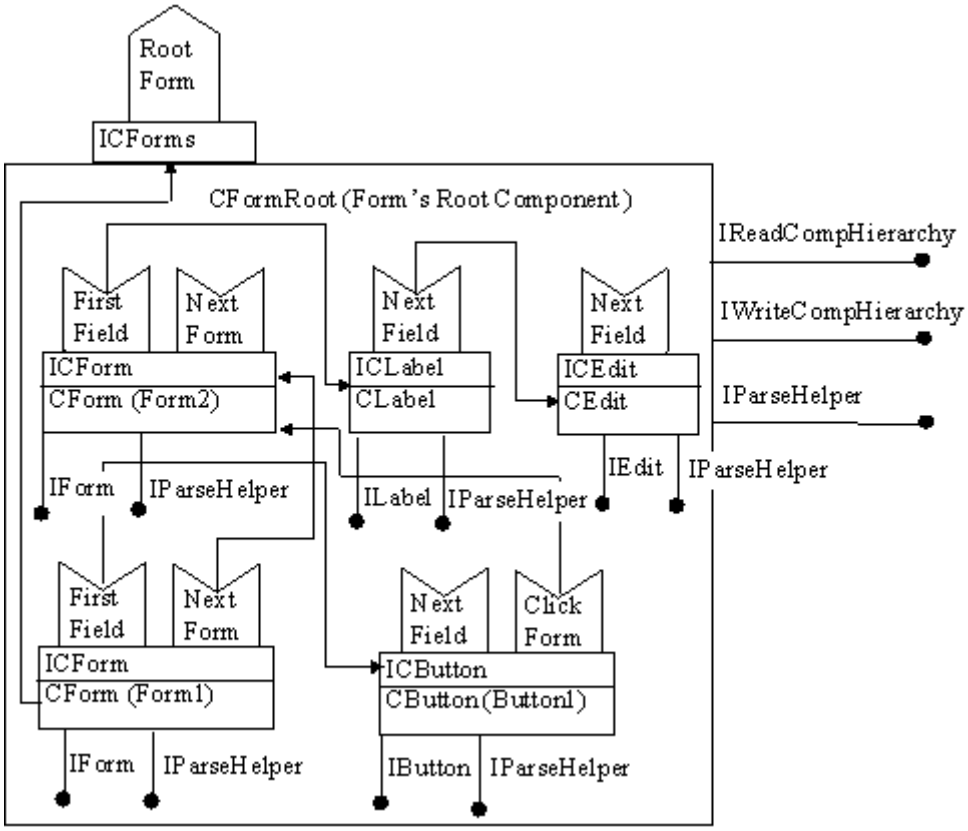

Figure 2, The CFormRoot assembly generated by SDPAssy.

Here is a derivation sequence and the corresponding semantic actions taken by the component hierarchy. The number to the left of the hyphen in each paragraph below corresponds to the rule the parser applies at that point in the parsing. The number is the rule number from the form description grammar listed above:

1 – The FORM token is matched by the parser. The annotation #CREATE <formclsid> resulted in a parse table entry for that token match notifying the parser to create a new component of type <formclsid> and pass it to the current (root) hierarchy component's IParseHelper::Create method. The parser continues and the STRING token is matched by "Form1" in the source. The **#$NAME** annotation resulted in a parse table entry for this token match notifying the parser to set the current component to the newly created form (the **$** triggered this) and call the current component's IParseHelper::Call with the annotation name (**NAME** in this case) and the token. The current component is then reset to the old (root) component. Similarly, the INT, INT and BEGIN tokens are matched with the 10, 10 and begin source text respectively. In each case, the current component is set to the new form component and its IParseHelper::Call is called with the annotation name and the current token. Since the non terminal fieldl is marked with the **$** annotation, the current component becomes the Form1 component for all of the field list processing. This sequence allows the root component to wire in Form1 and export its IForm interface, and to set the form's internal name, X, and Y coordinate attributes.

3 – The button token is matched which causes a new button hierarchy component to be created and passed to the current component (Form1). Form1's IParseHelper::Create is called to wire the field into the form. The NAME, X, Y and FORM attributes are set for the new button due to the matching of the STRING, INT, INT and STRING tokens to "Button1", 0, 0 and "Form2" respectively and the resulting calls to the button's IParseHelper::Call.

6 – There are no more fields left in Form1 and no annotations here. The END token is matched from rule 1 and the current component is reset to the root.

1 – The FORM token is matched resulting in Form2 being created and passed to the root's IParseHelper::Create. The NAME, X and Y properties are set for Form2. The BEGIN token is matched. The current component is set to the newly created form when the non terminal fieldl is processed. The wiring of Form1's button to form2 could also be done in the root's IParseHelper::Create since Form2 is a known entity now. This is the form that will appear when the user clicks the button in form1.

4 – The LABEL token is matched resulting in the creation of a new label component. Form2's IParseHelper::Create is called so that the new field can be wired into the form. The NAME, X and Y attributes are set using the label's IParseHelper::Call method as a result of matching the label name, X and Y coordinate tokens.

5 – The EDIT token is matched resulting in the creation of the edit component and its wiring into the assembly via the call to form2's IParseHelper::Create. The NAME, X and Y attributes are set for the edit field as a result of matching the edit name, x and y coordinate tokens.

6 – There are no more fields left for form2. The END token for form2 is matched in rule1.

2 – The ENDOFINPUT token is matched and processing is complete.

An external component could extract the resulting assembly's root IForm interface (the one for form1), through the root form pin and call IForm::Display to display form1. Clicking on the button in form1 would result in the calling of form2's IForm::Display.

3. References

[1] Aho, Alfred; Sethi, Ravi; Ullman Jeffrey; Section 4.4 Compilers Principals, Techniques and Tools, Addison-Wesley, 1988.

[2] Microsoft, Papers on COM, http://www.microsoft.com/com/dcom.asp.

[3] Peltz, Chris, "A Hierarchical Technique For Composing COM Based Components", 1999.

Back to top

Home