The Basic SUIF Programming Guide

Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam,
David L. Moore[†], Brian R. Murphy, Constantine Sapuntzakis

Computer Systems Laboratory
Stanford University

[†]The Portland Group, Inc.

January 17, 2001

Contents

1 Introduction **3**

2 An introduction to the SUIF IR **4**
2.1 The overall structure . . . 4
2.2 Scoping and Global Symbols . . . 4
2.3 Types, QualifiedTypes, and DataTypes . . . 5
2.4 Statements . . . 8
2.5 Expressions . . . 8

3 Programming with the SUIF program representation **9**
3.1 The IR Tree . . . 9
3.2 Casting and Type Identification . . . 9
3.3 Cloning . . . 9
3.4 Visitors . . . 10
3.4.1 CascadingMap . . . 10
3.4.2 VisitorMap . . . 10
3.5 Iterators . . . 12
3.6 Walkers . . . 13
3.6.1 Using A Selective Walker . . . 14
3.6.2 Using A Group Walker . . . 14

4 Writing a Pass **15**
4.1 Writing a Module . . . 15
4.2 Command line parsing . . . 15
4.3 Passes and Pipelinable Passes . . . 17

5 The Basic Data Structures **20**

6 Generating SUIF Intermediate Representation Nodes **21**
6.1 From Hoof to C++ . . . 21
6.2 Creating Objects . . . 23
6.3 Implementation of IR Modules . . . 25

7 Hoof Files in Detail **26**
7.1 Module Definitions . . . 26
7.2 Class Definitions . . . 26
7.3 Type Specifications in Hoof . . . 28
7.4 Collection Types . . . 29
7.4.1 Simple List . . . 29
7.4.2 Searchable List . . . 29
7.4.3 Indexed List . . . 30
7.4.4 Vector . . . 30

8 The Makefile System and System Installation **31**
8.1 Variables in Makefiles . . . 31
8.2 The Install System . . . 32
8.3 Building in Remote Directories . . . 33

1 Introduction

This document provides the basic information to help you get started programming in SUIF. We assume that you have already read the document *An Overview of the SUIF2 Compiler Infrastructure*, which can be found through `http://suif.stanford.edu`. As explained in that document, the system is based on two fundamental concepts: an extensible program immediate representation (IR) and a flexible module system. The system comes with a set of predefined IR nodes, the details of which can be found in the document *The SUIF Program Representation*. This guide discusses how to write code using the basic SUIF infrastructure. More specifically, it describes how to program in the module subsystem and how to define new IR nodes. Details of the implementation of the SUIF infrastructure are outside the scope of this document.

Many of you will use the IR that comes with the basic infrastructure and need not define any new SUIF objects. If you work with new programming constructs, you might want to create new nodes as subclasses of some predefined SUIF IR nodes. Even if you are not working with new programming semantics, you might also find it useful to define data structures in your analysis as new SUIF objects. SUIF objects have a standard interface and inherit many useful functions, such as the ability to be written out and read from files.

Here is the organization of the rest of this document:

Chapter 2.	An introduction to the base SUIF IR
Chapter 3.	Programming with the SUIF program representation
Chapter 4.	Writing a pass
Chapter 5.	The basic data structures
Chapter 6.	Generating SUIF intermediate representation nodes
Chapter 7.	Hoof files in detail
Chapter 8.	The Makefile system and system installation

Besides the basic infrastructure, the SUIF system comes with a suite of SUIF standalone programs, passes, and useful data structures and utilities. We are continuing to create more software based on the system which will be released when they are ready. Please consult the SUIF home page `http://suif.stanford.edu` for an up-to-date list of the available software.

2 An introduction to the SUIF IR

This section introduces the structure of a program represented in the SUIF IR. The purpose of this section is to introduce you to some idioms of SUIF that might be hard to discern otherwise and answer a number of questions which first time users of SUIF usually ask.

2.1 The overall structure

The root node of a SUIF program representation is an instance of the `FileSetBlock`. It has one or more `FileBlocks`, each of which corresponds to a single compilation unit. Each `FileBlock` itself contains symbol tables and procedure definitions and so forth. While the program representation is a collection of objects which may point to each other, embedded within the graph is an *ownership* tree. That is, every object in a SUIF IR has precisely one *owner*. This IR tree is very useful, as it guarantees that all nodes are visited exactly once if in visiting a node we visit each of the objects it owns.

You can examine the structure of a program by using the `print` method provided by the SUIF system to get a textual version of the representation. Suppose you have a file called `ex1.suif` with the programming code:

```
extern int i;
int l;
int j = 0;
static int a[3];
int proc(int *k,const int i)
    a[i] = *k + j;
    return j;
```

By running

```
suifdriver -e ‘‘import basicnodes suifnodes; load ex1.suif; print -o ex1.out’’
```

we get a print out in the file `ex1.out`, whose first 20 lines are shown in Figure 1.

The print function will traverse the ownership tree qin the representation and dump out all the information there is on the nodes. It prints out a node's class name (which includes the names of all the classes the node is derived from), its field names and their contents. If a field contains a `SuifObjects` the node owns, the printer will recursively print out the detailed information of the owned node. The output can be quite long as a result; for example, the textual representation of the short example above has over 500 lines.

The class names are shown in braces. The name of each class is output before the contents of that type's fields. When a class is derived, the name of the most derived class is printed, followed by the fields defined in that derivation, then the name of its parent, and its fields, and so on. So, in the above example, BooleanType is derived from NumericType which is derived from DataType which is derived from Type.

The print out is verbose but complete, making it a useful aid for understanding the structure of the representation. Once we are familiar with the structure, we would like to concentrate mainly on the contents. For that reason the SUIF system comes with an alternate printing style for all the basic nodes that display in the program in a more readable format. Simply import the `suifprinter` before printing the representation, and you will get the output shown in Figures 2 and 3.

2.2 Scoping and Global Symbols

The SUIF system captures the scoping information in the source program by using a nesting of symbol tables. The two `SymbolTable` fields in the `FileSetBlock` hold the outermost scope:

- `external_symbol_table` holds the symbols and types visible to the outside, and

```
{FileSetBlock}[
  _external_symbol_table[0x0806AEA8]->[
    {BasicSymbolTable}[
      _symbol_table_objects[0][0x080703D0]->[
        {BooleanType}[
          {NumericType}[
            {DataType}[
               bit_size=[32]
               bit_alignment=[32]
               {Type}
                 [{SymbolTableObject}
                   [name=[]{AnnotableObject}[]]]]]]]
      _symbol_table_objects[1][0x08071480]->[
        {VoidType}[
           {DataType}[
             bit_size=[0]
             bit_alignment=[0]
             {Type}
               [{SymbolTableObject}
                 [name=[]{AnnotableObject}[]]]]]]
```

Figure 1: Textual Representation of Example 1 (First 20 Lines)

- `file_set_symbol_table` holds the symbols and types visible to the code in this file set.

The next level in the hierarchy are the `symbol_table`s belonging to `FileBlocks`. They contain all the symbols local to a single file. For example, C `static` symbols will be placed in this symbol table.

Our sample program declares four variables in different ways. Each of these has a different meaning in C. The first variable, `i`, is declared extern. The second variable, `l`, is also externally visible but is not declared extern. These are actually different declarations in C with different semantics. The third variable, `j`, is given an initial value. All three of these variables are placed in the external symbol table, as they are all visible outside of this SUIF file. The definition of `j` is found in the definition block in a file block. Finally, the variable `a` is declared static, and can thus be found in the symbol table for the file block. There are no symbols in the file set symbol table, the table containing objects that are visible across compilation units that have been incorporated into this SUIF file but which are not externally visible. This is because there are no such symbols in C. If one was compiling a module in Ada which contained child modules, then there might be entries in this symbol table.

One more thing needs to be observed about variable declarations. The type of symbol, must be no more deeply nested than the declaration of the variable. For the purposes of this rule, the file set symbol table is considered to be nested inside the external symbol table.

2.3 Types, QualifiedTypes, and DataTypes

The class from which all types derive is `Type`. From this are derived four types: `QualifiedType` for symbols, `DataType` for expressions, `ProcedureType` for procedures, and `LabelType` for labels.

`QualifiedType` is used for declaring symbols. A qualified type can simply be a pointer to a type, or it can be a pointer to a type with qualifications, such as ”const”. Attributes such as C's ”volatile” are also handled in this way, even though this is not really type information. ”static” is handled by setting the `is_static` field in a symbol definition to true, not by entering ”static” in the qualifications. A qualified type will never point to another qualified type. There is a module TypeBuilder which will enforce this if you use its methods to build qualified types.

```
************* EXTERNALS *************
External Symbol Table
  type:t0:  bool 32 align:32
  type:t1:  void 0
  type:t2:  int signed:0 32 align:32
  type:t3:  int signed:1 32 align:32
  type:t4:  int signed:1 8 align:8
  type:t5:  int signed:0 8 align:8
  type:t6:  int signed:1 16 align:16
  type:t7:  int signed:0 16 align:16
  type:t8:  float 32 align:32
  type:t9:  float 64 align:32
  type:t10:  float 64 align:64
  type:t11:  qual of @t3:i.32 qualifications:<None>
  var_sym:t12:  "i" with t:@t11:q.(@t3:i.32) addrTaken:0
    <None>
  var_sym:t13:  "l" with t:@t11:q.(@t3:i.32) addrTaken:0
    <None>
  var_sym:t14:  "j" with t:@t11:q.(@t3:i.32) addrTaken:0
    <None>
  type:t15:  array [(@t3:i.32) 0..(@t3:i.32) 2] of @t11:q.(@t3:i.32)
  type:t16:  qual of @t15:a[] qualifications:<None>
  type:t17:  ptr 32 align:32 to @t11:q.(@t3:i.32)
  type:t18:  qual of @t17:p.32 qualifications:<None>
  type:t19:  qual of @t3:i.32 qualifications:const
  type:t20:  Cfunc(@t18:q.(@t17:p.32), @t19:q.(@t3:i.32))
               vararg:0 argknwn:1 algn:0 ret:@t3:i.32
  proc_sym:t21:  "proc" with t:@t20:f((@t3:i.32)()) addrTaken:0
    <None>
  type:t22:  ptr 32 align:32 to @t16:q.(@t15:a[])
  type:t23:  array [(@t3:i.32) 0..(@t3:i.32) 0] of @t11:q.(@t3:i.32)
  type:t24:  qual of @t23:a[] qualifications:<None>
  type:t25:  ptr 32 align:32 to @t24:q.(@t23:a[])
************* GLOBALS *************
Global Symbol Table
  <None>
```

Figure 2: Example 1 Output using the Suifprinter Style (a)

```
************* FILE   ex.cpp.cil *************
Annotes:
  <None>
Symbol Table:
  var_sym:t26:  "a" with t:@t16:q.(@t15:a[]) addrTaken:1
    <None>
Definition Block:
  Variable Definitions:
  @t13:(@t11:q.(@t3:i.32)) "l" =
  @t14:(@t11:q.(@t3:i.32)) "j" = ExprVal: (@t3:i.32) 0
  @t26:(@t16:q.(@t15:a[])) "a" =
Procedure Definitions:

  PROC  @t21:"proc"
    <None>
  Proc Symbol Table:
    param_sym:t27:  "k" with t:@t18:q.(@t17:p.32) addrTaken:0
      <None>
    param_sym:t28:  "i" with t:@t19:q.(@t3:i.32) addrTaken:0
      <None>
  Definitions:
    Variable Definitions:
      <None>
    Procedure Definitions:
      <None>
    Annotes:
      <None>
  Args:
    @t27:(@t18:q.(@t17:p.32)) "k"
    @t28:(@t19:q.(@t3:i.32)) "i"
  Body:
    STR <dst> = <src>
      <dst>:   (@t17:p.32) array <base> [ <idx> ]
          <base>:  (@t25:p.32) convert <exp>
          <None>
            <exp>:  (@t22:p.32) SYMADDR @t26:(@t16:q.(@t15:a[])) "a"
            <None>
          <idx>:  (@t3:i.32) @t28:(@t19:q.(@t3:i.32)) "i"
      <src>:  (@t3:i.32) <e1> add <e2>
        <None>
          <e1>:  (@t3:i.32) LOADEXP <addr>
            <addr>:  (@t17:p.32) @t27:(@t18:q.(@t17:p.32)) "k"
          <e2>:  (@t3:i.32) @t14:(@t11:q.(@t3:i.32)) "j"
    RET <retval>
      <retval>:  (@t3:i.32) @t14:(@t11:q.(@t3:i.32)) "j"
  PROC END   @t21:"proc"
Annotes:
   <None>
```

Figure 3: Example 1 Output using the Suifprinter Style (b)

DataType is used for values of expressions. For example, when a variable is loaded, the type of the load expression is the type pointed at by the qualified type.

Current the ProcedureType has only one derived type: CProcedureType. Procedures represented by this class follow the procedure linkage convention of C. Procedures in SUIF are represented by Symbols. A procedure symbol has a type and may have a definition.

2.4 Statements

The body in a procedure is defined as a set of Statements. The collection of Statements includes some high-level constructs whose semantics may be useful for high-level transformations. Various dismantlers can be applied to reduce these statements down to a small core set so that passes that do not need the high-level semantics can be simply written.

2.5 Expressions

The following are some useful hints on how to handle expressions.

First, you should be careful about typing in expressions. If you take a symbol address, the result type should be a pointer to the type of the symbol you are addressing. If you do a load, the result type should be one pointer less than the expression you are loading. Stores should have one less reference on the right hand side than the left. For example, in i=j, the type of the left hand side is a pointer to int while the type of the right hand side is an int.

Array indexing operations must be performed on pointers to arrays, not on pointers to element types. If the source code includes indexing of element types, the front end casts it to an array type.

Because subexpressions are owned by the expressions that contain them, you can never share an expression node. To do common subexpression elimination, you must store the value in a temporary and load from the temporary. For constants, you do not want to do this since it loses the const-ness of the expression. If you look in expression_utils.h in the directory nci/suif/suif2b/utils you will find some useful utilities for manipulating constant expressions.

Finally, when you do store an expression into a temporary, don't give the temporary a name! Leave it anonymous. There is a pass that will add names, should you need them, after compilation has completed, and it will make sure the names do not clash with any other name. This maintains the maximum flexibility. If you invent an identifier, you either have to invent something which is lexically invalid in your source language, which will cause problems if you later try to output the code as source, or you have to produce an identifier which could conflict with a yet-to-be-seen identifier in the source.

3 Programming with the SUIF program representation

This section discusses some concepts and tools useful for navigating and operating on the SUIF representation. We discuss how we determine the types of the representation, how to dispatch code based on the type using the concept of a visitor, and how to traverse the code using iterators and walkers.

3.1 The IR Tree

The state of the entire compilation process is held by the `suif_env` object, and there are no other global variables or states. The `suif_env` holds onto the root of the program being compiled `FileSetBlock` and references to all the modules imported into the environment.

As discussed in Section 2, the program is embedded in an ownership tree. Every object in the SUIF IR has precisely one owner, which can be retrieved by the `get_parent()` method in a `SuifObject`. You do not need to set the owner explicitly; it is set implicitly when a pointer to the object is assigned to a pointer field with the "owner" property defined in the *Hoof* description. You will get an error message if you try to create multiple owners for a single node.

The set of nodes owned by a node can be obtained using the `collect_instance_objects` iterator, defined in `basesuif/suifkernel/utilities.h`, as is done in the `WalkingMap` implementation in `walking_maps.cpp` of the same directory.

3.2 Casting and Type Identification

SUIF is designed so that a compiler pass can operate on a SUIF program with IR nodes whose declarations were not included at the time the pass was compiled. For example, a pass that is defined for the base SUIF representation can be applied to program representations that include new IR subclasses that a programmer defines. Information specific to these new classes are stored away and will be written out when the representation is saved to disk. The system keeps track of the type of each SUIF object internally, and you should not use standard C++ techniques such as dynamic casts to determine the types of these objects. Instead, SUIF objects, which are all descended from the class `SuifObject`, provide their own type identification and object casting methods. Table 1 shows the correct methods of casting and identifying types. To use these, you must include the header file `iokernel/cast.h`.

Method	**Meaning**
`bool is_a<`*suifclass*`>(Object *)`	returns `true` if the object is exactly of type *suifclass*, which must be descended from `Object`
`bool is_kind_of<`*suifclass*`>(Object *)`	returns `true` if the object's class is type *suifclass* or derived from it.
suifclass `* to<`*suifclass*`>(Object *)`	A dynamic cast to type *suifclass*. Asserts if the cast is illegal. Returns `NULL` if the object is `NULL`.

Table 1: Casting and Type Identification

3.3 Cloning

Cloning is a basic operation that is useful for many compiler transformations. The SUIF has an elaborate cloning subsystem (`CloneStream` class in `iokernel/clone_stream.h`) that allows the programmer to perform cloning in many different ways. The details of this system are beyond the scope of this document.

Most SUIF programmers only need to know about the two cloning methods that are defined in `SuifObject`. The `deep_clone` method clones this object and all the objects in the ownership subtree rooted by this object, and returns the clone. The shallow_clone method clones this object, copies all references this object have to the new clone, and returns the new clone.

3.4 Visitors

A Visitor takes a pointer to an object and does something depending on what "kind" of object it is (of which derived class of SuifObject it is an instance). The builder of a Visitor need not specify what to do for every conceivable kind of object; if a behavior is not assigned to the class of the object, then the most specific superclass's behavior is used instead.

In SUIF several kinds of Visitor are used internally, and are also available to users of the system. The CascadingMap<T> is a template class which associates data of type T with a class. The VisitorMap class applies a specified function to the provided object. In both cases, the dynamic dispatch is implemented using the internal MetaClass system, querying the object's MetaClass and determining the closest ancestor MetaClass with specified behavior. Most SUIF programmers need not be concerned with information at the MetaClass level.

3.4.1 CascadingMap

CascadingMap is declared in suifkernel/cascading_map.h. A CascadingMap<T> is a template class which maps objects to T values. The T values must have copy operations and an inequality test. A default T value is provided to the constructor; this is returned for objects with no binding. The operations are

- Assign some T value data to a objects of a named class and its subclasses:

```
void assign( const LString &className, T data);
```

- Lookup the binding for an object, or return the default value, if none is applicable:

```
virtual T lookup( Object* object );
```

Here is an example of use:

```
// register the data
CascadingMap<char> *map = new CascadingMap<char>(suif_env);
char *my_info1 ="procedure";
char *my_info2 ="statement";
map->assign( ProcedureDefinition::get_class_name(), my_info1 );
map->assign( Statement::get_class_name(), my_info2 );

// Pick an object:
SuifObject *so;
...
map->lookup(so);
// returns procedure for any procedure definition,
// returns my_info2 for any statement (or subclass)
// returns 0 for anything else
```

3.4.2 VisitorMap

VisitorMap is declared in suifkernel/visitor_map.h. The VisitorMap is a special-purpose Visitor which dynamically dispatches a SuifObject and the address of state to one of several functions, depending on its type. Each function is of type VisitMethod:

```
typedef void *Address; // declared in iokernel/iokernel_forwarders.h
typedef void (*VisitMethod)( Address state, Object *object );
```

The constructor takes a pointer to the SuifEnv, which is used to interact with the MetaClass system to determine the appropriate action. It has the following operations:

- Register a method and a state for a class. (The state parameter is passed to the method whenever it is called.)

```
void register_visit_method(
        Address state,
        VisitMethod method,
        const LString &className );
```

- Optionally, a default function and state can be registered; otherwise, no action occurs when the object belongs to no registered class name.

```
virtual void register_unknown_method( Address state,
                                      VisitMethod visitMethod );
```

- Dispatch applies the registered function which is most specific to the given object. (The registered method is applied to its registered state and the given object.)

```
virtual void apply( Object* object );
```

Here is an example use:

```
class MyVisitorInfo {
     static void do_static_procedure_definition(
                     void *info,
                     ProcedureDefinition *obj) {
         (MyVisitorInfo *)info->do_procedure_definition(obj);
     }
     void do_procedure_definition(ProcedureDefinition *proc_def) {
         ...do something...
     }
}
// register the visitor functions
VisitorMap *map = new VisitorMap(suif_env);
MyVisitorClass my_info;
map->registerVisitMethod( &myinfo,
                          &MyVisitorClass::do_static_procedure_definition,
                          ProcedureDefinition::get_class_name() );
// Pick an object:
SuifObject *so;
...
map->apply(so);
```

3.5 Iterators

There are two common ways of traversing the SUIF IR representation: iterators and walkers. We discuss iterators here and walkers in the next section. The interface of an iterator can be found in `suifkernel/iter.h`. An iterator is an ordered collection of objects and an interface that allows one to traverse the objects. It has the notion of a current index, which is initialized to the first position of the collection, and provides methods to move the index up and down. The following example shows the use of an iterator:

```
for (Iter<Expression*> iter = st->get_expression_iterator();
     iter.is_valid(); iter.next()) {
    Expression *e = iter.current();
}
```

The important methods in its interface are:

Method	**Meaning**
`length`	Returns the number of elements in the whole collection.
`current`	Returns the current element.
`next`	Advances the current position.
`previous`	Sets back the current position.
`is_valid`	Returns T if more elements exist beyond the current index.
`set_to`	Sets the current index to a particular value.

The SUIF system provides a set of useful templates for constructing iterators and sets of `SuifObjects` in the `suifkernel/utilities.h` file.

- `object_iterator`. Build an Iterator that will walk over all Owned objects in the ownership tree rooted at the `start_object`. Example:

  ```
  for (Iter<CallStatement> iter =
       object_iterator<CallStatement>(file_set_block);
       iter.is_valid(); iter.next())
    {
      CallStatement *call = &iter.current();
      // do something with the call
    }
  ```

  Even though this iterator will give you access to a reference to the desired object, it is a VERY BAD idea to try to modify the object during iteration. Instead consider using the `collect_objects` defined below to build a list of all the objects to operate on, then make transformations.

- `collect_objects`. Build a list of all Owned objects in the ownership tree rooted at the `start_object` that subclass from the given type. Example:

  ```
  list<ProcedureSymbol*>* sym_list = collect_objects<ProcedureSymbol>(root);
  for (list<ProcedureSymbol*>::iterator iter = sym_list->begin();
       iter != sym_list->end(); iter++)
    {
      ProcedureSymbol *ps = *iter;
      // do something with the procedure symbol
    }
  // clean up
  delete sym_list;
  ```

Use this iterator to collect all of the objects that subclass from the given type. You can then use the and modify the returned objects after collecting the objects.

- `collect_instance_objects` Build an Iterator that will walk over all of the immediate child SuifObjects that are subtypes of the template type that are *owned* by this SuifObject. Example:

```
for (Iter<Expression> iter =
     collect_instance_objects<Expression>(statement);
     iter.is_valid(); iter.next())
  {
     Expression *exp = &iter.current();
     // do something with the OWNED expression
  }
```

  Like the `Statement::get_child_statement_iterator()` method for Statements, this iterator does not dive into the entire ownership tree rooted at the start_object.

- `suif_object_ref_iterator`. Build an Iterator that will walk over all of the immediate child SuifObjects that are subtypes of the template type that are *referenced* (not *owned*) by this SuifObject. Example:

```
for (Iter<Type> iter =
     suif_object_ref_iterator<Type>(expression);
     iter.is_valid(); iter.next())
  {
     Type *t = &iter.current();
     // do something with the referenced type.
  }
```

  This can be used with an iterator over the ownership tree like the `object_iterator<>()` to find all the SuifObject links in a system.

3.6 Walkers

A Walker is a class that provides support for walking trees and visiting certain nodes. The classes from which you will normally derive are `SelectiveWalker` and `GroupWalker`, both found in the file `suifkernel/group_walker.h`. `SelectiveWalker` allows you to walk one class of object (including its derived classes), while `GroupWalker` allows you to register a list of walkers for a variety of classes of object. This is a much better way of performing different actions on differing classes of objects than using a giant `if` statement because it is easy to add new cases.

The `ProcedureWalker` in the file `basepasses/transforms/procedure_walker_utilities.h` is useful if you want to apply a SelectiveWalker to each procedure. If you want to visit all SUIF objects, without selecting by class, you can derive instead from the class `SuifWalker`, found in `suifkernel/suif_walker.h`.

The work for all walkers is performed by the function operator:

```
Walker::ApplyStatus operator () (SuifObject *x)
```

which is called for every object matching your selection criteria. The returned status, drawn from the `enum ApplyStatus`, controls the further progress of the walk:

ApplyStatus	**Meaning**
Continue	Normal completion
Stop	Terminate the walk successfully
Abort	Terminate the walk with error
Truncate	Do not walk sub-trees of this node (only useable in pre-order walks)
Replaced	This subtree has been replaced. The `set_address` walker method must have been called before returning this status with the pointer to the new sub-tree. In pre-order, this new subtree will be walked instead of the old.

3.6.1 Using A Selective Walker

The following is a simple example of a walker. This walker visits all variable symbols and ensures that they have names. The full code can be found in `nci/suif/suif2b/transforms/symbol_walkers.cpp`. Only the traversal code is shown.

```
class SymbolNamer : public SelectiveWalker {
    public:
        SymbolNamer(SuifEnv *env)
             : SelectiveWalker(env,
                    VariableSymbol::get_class_name()) {}
        ApplyStatus operator () (SuifObject *x);
    };
Walker::ApplyStatus SymbolNamer::operator () (SuifObject *x) {
    VariableSymbol *symbol = to<VariableSymbol>(x);
    name_variable(symbol); // not included in sample
    return Walker::Continue;
    }
```

3.6.2 Using A Group Walker

As discussed above, a group walker allows you to call a different function operator for each class that you are interested in. Below is an example, simplified from the actual code. In this code, the constructor appends a set of selective walkers to the group walker. These are the individual walkers for the various classes of object of interest. They will be invoked in a single pass through the program. It is a good idea to append the selective walkers in the constructor, as it allows others to extend the walker easily to work with new constructs. For example, a programmer can define some new loop constructs as an extension to SUIF. He or she can simply derive from this class and add more cases to the derived class's constructor. Any changes we make to this code automatically takes effect on the new walker.

```
class RemoveIfAndLoopWalker: public GroupWalker {
public:
    RemoveIfAndLoopWalker(SuifEnv* the_env, ProcedureDefinition* proc_def);
};
RemoveIfAndLoopWalker::RemoveIfAndLoopWalker
    (SuifEnv* the_env, ProcedureDefinition* proc_def)
    : GroupWalker(the_env) {
    append_walker(new while_statement_walker(the_env));
    append_walker(new do_while_statement_walker(the_env));
    append_walker (new if_statement_walker(the_env));
    append_walker (new c_for_statement_walker(the_env);
}
```

4 Writing a Pass

In the SUIF compiler infrastructure, a programmer can construct his or her own compiler by putting together different modules. Shared libraries (also known as DLLs or dynamically linked libraries) consisting of one or more modules can be imported in the compiler environment dynamically There are two common kinds of modules: those that contain definitions of new IR nodes and those that contain new passes. The following discusses how to define a module, how to handle the command line interface, and finally discuss two important subclasses of modules for writing compiler passes, the Pass module and the PipelinablePass module.

4.1 Writing a Module

Figure 4 shows the code that you need to create a module. The high-level interface of a Module is documented in *An Overview of the SUIF2 Compiler Infrastructure.* The constructor simply passes down the pointer to the suif environment to the parent class together with the name of this module. The clone routine does nothing except return the current pointer since the module contains no state. The initialize function is executed at the time the module is registered. It usually defines the parameter profile, which will be described in more detail next, and performs the necessary initialization for the specific algorithm implemented. The last, and certainly the most important method is the execute routine, which does all the real work. It is executed whenever the module is invoked.

Each shared library has an initialization routine which is called when the shared library is loaded. This has the name init_<*library_name*>. In this example, we plan to put our example module into the sample library, hence we have the init_sample function. This library initialization routine is responsible for registering the modules defined in the library. It creates a new instance of each of the modules and registers it using the register_module method in the suif_env's ModuleSubSystem. It should also call all the initialization routines for all the shared libraries it depends upon. For example, if the module uses the suifnodes library, then it should include init_suifnodes(suif_env) in its init routine. Note that these libraries may already be initialized and, if so, the initialization will not be performed a second time.

4.2 Command line parsing

Once a module is registered, it can then be invoked by its name. A module usually accepts flags or inputs as parameters to the module. The SUIF system includes a command-line-option package to help with this tedious task of parsing the input parameters. The module's initialization routine should store the definition of its parameter profile in the _command_line field. The _command_line is an OptionList, which is a subclass of Option. It defines the options to be parsed, the locations where the input values should be saved, and some "help" information to be displayed when the user fails to supply the expected inputs.

As an example, consider writing a pass that that writes the SUIF representation to a given file. The pass therefore requires a filename as an argument. This is how the command line options are specified:

```
#include <suifkernel/command_line_parsing.h>
String _file_name;

_command_line->set_description("Write SUIF out to a given file.");

OptionString *_file_name_argument = new OptionString("Filename", &_file_name);
_file_name_argument->set_description("Input file name");
_command_line->add(_file_name_argument);

// the input file name can be accessed as _file_name.c_str();
```

```
#include <suifkernel/suif_env.h>
#include <suifkernel/module_subsystem.h>
#include <suifkernel/module.h>

class SampleModule : public Module {
public:
  SampleModule (SuifEnv* suif_env, const LString &name);
  virtual ~SampleModule();
  Module *clone() const;
  virtual void initialize();
  virtual void execute();
};

SampleModule::SampleModule (SuifEnv* suif_env, const LString &name)
    : Module (suif_env, name) {
}

SampleModule::~SampleModule(){
}

Module *SampleModule::clone() const {
  return(Module*)this;
}

void SampleModule::initialize() {
  Module::initialize();
}

void SampleModule::execute() {
}

extern "C" void init_sample(SuifEnv *suif_env) {
  ModuleSubSystem *ms = suif_env->get_module_subsystem();
  ms->register_module(
     new SampleModule (suif_env, "sample"));
}
```

Figure 4: Definition of a Module

The code above simply adds to the _command_line the requirement to accept a string as an argument. The argument is given the mnemonic name of “Filename”, and the user supplied string is expected to be stored in the _file_name String. The actual C string can be retrieved from the String data structure as _file_name.c_str(). It is also possible to retrieve the file name with the method invocation

```
_file_name_argument->get_string()->get_string()
```

The code also gives a short description of the pass as well as the argument. The description will be printed out if the user invokes the pass incorrectly. The following will be printed for the example above if parsing fails:

```
Description:
save_file
Write SUIF out to a given file

<Filename>
Input file name
ERROR IN
FILE: module.cpp   LINE: 42 MODULE: suifkernel
Parsing of command line has failed.
```

The system provides the support for many common command line options, such as a collection of optional or repeated flags, which may be followed by a parameter. This is implemented with a class hierarchy; deriving from the Option base class are different subclasses to represent different expected inputs. For example, the OptionString class accepts an input string. All options have similar interfaces. The first argument of an option's constructor is its mnemonic name as a string. The second argument is optional, and it corresponds to the address to which the input is to be saved if parsing succeeds. Some options accept additional parameters to change its default behavior. It is also possible to retrieve the inputs using methods supplied by the option class. Table 2 shows all the Option subclasses.

The command_line_parsing.h includes additional functions to support two common option styles. The first is the build_prefixed_string function which creates an Option tree that matches a literal followed by a string (e.g. -I /usr/include). The second is the build_multi_string function that creates an option that matches a set of strings.

4.3 Passes and Pipelinable Passes

Most of the modules are compiler passes, which performs some analysis on the program representation, rooted at the file_set_block in the textttsuif_env. This common schema is captured in the definition of a Pass, which is a subclass of a Module.

```
class Pass : public Module {
protected:
  Pass( SuifEnv* suif_env, const LString &name );
  virtual ~Pass();

public:
  // the default implementation invokes the method do_file_set_block
  virtual void execute();
  virtual Module* clone() const = 0;

  // your stuff goes here
  virtual void do_file_set_block( FileSetBlock* file_set_block ) = 0;
};
```

`Option`	The base class from which all options are derived. All Options have a `set_description` method which allows the user to supply a short explanation of the option.
`OptionLiteral`	Matches a given literal in the input. The `is_set` method specifies whether there is a match. The result is also stored in the `bool` to the constructor.
`OptionString`	Accepts a string token in the command line. The `get_string()->get_string()` method returns the input String. Alternatively, the address of a String can be passed to the constructor as a second parameter and this string will receive the value.
`OptionInt`	Accepts an integer from the command line. Similarly to `OptionString`, a second parameter can be given to the constructor giving a pointer to an integer where the accepted value is to be stored. The result can also be retrieved with `get_int()->get_int()`.
`OptionList`	Accepts a sequence of options. Options are added to the `OptionList` with the `add` method. Note that these added options are not alternatives but rather required parameters.
`OptionSelection`	Contains a list of optional parameters. Choices are specified by using the `add` method. The default is that it has to match exactly one of the options. By passing `true` to the constructor, it can match zero or one option.
`OptionLoop`	A kleene star option. The constructor takes a pointer to the `Option`, and matches the given option zero or more times. By passing `false` to the constructor, the option has to be matched at least once. There is no way to tell how many times the option matched.
`OptionStream`	Matches part of a command line enclosed in braces ({ and }). No parsing is done inside the braces.

Table 2: Parameter Profile Classes

A `Pass` simply adds a new method, `do_file_set_block`, which is invoked in the `execute` routine on the `file_set_block` of the `suif_env`. To write a pass, you must derive from this class and define the `clone` and `do_file_set_block` methods.

Another useful schema of a pass is captured by the `PipelinablePass`, which is a subclass of `Pass` itself. Its declaration is shown below. A pass is pipelinable if its computation can be applied to each procedure independently. The system execute a sequence of `PipelinablePass`es by applying all the functions from the different passes to the same procedure in the program, before moving onto another procedure. This execution order should have a better data locality. The syntax to call a group of pipelined passes in a `suifdriver` is "`pipeline {pass1;pass2; }`".

```
class PipelinablePass : public Pass {
public:
   PipelinablePass( SuifEnv* suif_env, const LString &name );
   virtual ~PipelinablePass() = 0;

   // will execute the Pass as if it is not pipelined
   virtual void execute();

   // May return this if the pass has no state.
   virtual Module* clone() const = 0;

   // override one or more of the following methods
   // the default implementation is empty
   virtual void do_file_set_block( FileSetBlock* file_set_block );
   virtual void do_file_block( FileBlock* file_block );
   virtual void do_variable_definition( VariableDefinition* var_def );
   virtual void do_procedure_definition( ProcedureDefinition* proc_def );
   virtual void finalize(){};
};
```

The pipelined execution starts by calling the `do_file_set_block` methods in all the pipelined modules. Next, for each file block in the program, it calls the `do_file_block` methods in all the modules. Next, for each variable definition in the file block, it calls the `do_variable_definition` methods in all the modules. Next, for each procedure in the program, it calls the `do_procedure_definition` methods in all the modules. Finally, it calls the `finalize` methods in all the modules.

```
int main() {
    SuifEnv* suif = new SuifEnv();
    suif->init();

    init_basicnodes(suif);              // import basicnodes
    init_suifnodes(suif);               // import suifnodes
    suif->read("hello_word.suif");      // load suif file
    init_s2c(suif);
    // TBD: how do we call this?
    //   There seems to be no good interface
    //   if parameters must be passed
}
```

Figure 5: Calling a Pass Directly (incomplete)

5 The Basic Data Structures

The basic SUIF infrastructure uses a set of basic data structures that you might also find useful for your own code. These data structures are defined in the directory `nci/suif/suif2b/common`.

Data Structure	**Filename**	**Implements**
`IInteger`	`i_integer.h`	Infinite precision integer
`String`	`MString.h`	String
`LString`	`lstring.h`	String with pointer equality
`list`	`suif_list.h`	Simple list template
`indexed_list`	`suif_indexed_list.h`	An indexed list containing key and value pairs
`suif_hash_map`	`suif_hash_map.h`	A map function based on a hash table
`suif_vector`	`suif_vector.h`	Vector template
`SparseVector`	`sparse_vector.h`	Sparse vector template

Table 3: Basic Types

Some of these are similar to STL types and it is intended that they can be replaced by the STL types if so desired. The STL templates have not been used from the beginning because some compilers still have trouble compiling the STL templates and compile time will probably be slower.

6 Generating SUIF Intermediate Representation Nodes

This section describes how to create Modules of intermediate representation nodes. Throughout this section, we will use a simple example to illustrate the process. We first show how to translate a simple *Hoof* file into C++ .h and .cpp files. We then explain the contents of the files generated.

6.1 From Hoof to C++

The SUIF internal representation is used by all front ends and many optimization passes. It consists of a set of classes defined using *Hoof* files. These are translated by a macro processor called `smgn` to produce the actual header files, and some code that defines the classes to the input-output system. You can extend SUIF by defining your own classes in *Hoof* files. This section describes how this works and how to add your own extensions. The actual predefined SUIF classes are documented in the file `suifguide.ps`.

Each class has a definition in the *Hoof* file. The most important part of each declaration is a list of fields. Methods are not usually declared. Instead, they are derived from the declarations of the fields.

Consider the following simple *Hoof* file:

```
module my {
   include "basicnodes/basic.h";
   include "suifnodes/suif.h";
   import basic;
   import suif;

   concrete Example : SuifObject {
       int x;
   };
}
```

The first line gives the name of this module, "`My`". It is followed by four lines that perform some book-keeping. The "`include`" statements cause corresponding "`#include`" statements to be added to the generated source. The `import` statements causes calls to the initialization routines for the modules `basic` and `suif` to be generated as part of the initialization routine for this module. The next few lines give the actual definition of a new class. Notice that it derives from `SuifObject`. All class definitions that do not derive from any other SUIF class must derive from `SuifObject`.

To process this *Hoof* definition, we need the following `Makefile`

```
SUIF_INTERNAL = defined
OBJ_FILES = my.o
HOOF_MAC = $(SMGNDIR)/suif.mac
HOOF_DEP = ../basicnodes/basic.hoof ../suifnodes/suif.hoof
SMGN_INCLUDES = $(SMGNDIR)/searchable_list_builders.mac \
                $(SMGNDIR)/indexed_list_builders.mac \
                $(SMGNDIR)/list_builders.mac \
                $(SMGNDIR)/vector_builders.mac
HOOF = my
HOOF_GEN = my.h my_factory.h my.cpp my_forwarders.h
SRCS = my.cpp
```

Note that this assumes that the files are in the directory `nci/suif/suif2b/basesuif/my`. Otherwise, you will have to adjust the relative paths in lines for `HOOF_DEP` and the `include` of `Makefile.std`.

Typing `make` will generate the following files:

```
#ifndef __MY_H__
#define __MY_H__
#include "suifkernel/suifkernel_forwarders.h"
#include "suifkernel/suif_object.h"
#include "common/suif_list.h"
#include "common/suif_vector.h"
#include "common/suif_indexed_list.h"
#include <new>
#include "my_forwarders.h"
#include "common/formatted.h"
#include "suifkernel/iter.h"
#include "common/i_integer.h"
#include "basicnodes/basic.h"
#include "suifnodes/suif.h"
#include "iokernel/meta_class.h"
extern "C" void EXPORT init_mynodes(SuifEnv* suif);
class Example : public  SuifObject {
  public:
    virtual int get_x () const;
    virtual void set_x(int the_value);
    virtual ~Example();
    virtual void print(FormattedText &x) const;
    static const LString &get_class_name();
  protected:
    //use the factory to create objects
    friend class MyObjectFactory;
    Example();
    static void constructor_function( Address address );
  private:
    int _x;
    };
#endif
```

Figure 6: The File `my.h`

File	**Contents**
`my.h`	The declaration of the class.
`my_forwarders.h`	Forward declarations for the class. This is all you need to declare pointers to the class.
`my_factory.h`	The declarations for the object factory that is used to create objects of the class.
`my.cpp`	Bodies for all of the above.

Figure 6 shows the file `my.h`. Notice that the methods `set_x` and `get_x` have been generated automatically. The rest of the declarations are associated with the SUIF infrastructure, for the support of input/output of SUIF data and iteration over SUIF data. Every SUIF class contains a `get_class_name` static function. It returns the name of the class (in this case `"Example"` as an `LString`. An `LString` is just a string with the property that all equal `LStrings` share a pointer. `LString` is one of the basic, predefined types of SUIF. Notice that because this is static, you cannot use this function to find the type of an actual instance. Type casting and identification are addressed in Section 3.2.

After the "`protected`" you will see the declaration of a friend class `MyObjectFactory`. This is the object factory that creates `Example` objects. It is declared in the `my_factory.h` file. Next is a default constructor for the class. As this is protected, you cannot do "`new Example`" in your code. Instead one calls

```
#ifndef MY_FACTORY_H
#define MY_FACTORY_H
#include "common/suif_list.h"
#include "common/MString.h"
#include "common/i_integer.h"
#include "my_forwarders.h"
#include "iokernel/meta_class.h"
#include "suifkernel/real_object_factory.h"
#include "iokernel/stl_meta_class.h"
Example* create_example(SuifEnv *env,int  x);

class MyObjectFactory : public RealObjectFactory {
  public:
    static const LString &get_class_name();
    virtual const LString &getName();
    virtual Example* create_example(int  x);

    virtual void init_io( ObjectFactory* of );
  protected:
    };

#endif
```

Figure 7: `my_factory.h`

a "`create_example`" routine found in `MyObjectFactory`. This is described below. `constructorFunction` is a function which is called to create an instance of the class at a given address. This is used by internally by the SUIF infrastruture implementation. Finally, you will see the private declarations of the fields of the class, in this case just "`x`".

6.2 Creating Objects

Next we look at the declarations in `my_factory.h`, shown in Figure 7. You cannot use `new` to create instances of objects, because the constructor is protected. Instead you must call a create method from the object factory. The name of this routine is the name of the class, converted to lower case and with underlines inserted before capitals, preceded by *create*. For the example class above, the create routine would be called `create_example`. For a class `SymbolTableObject` it would be called `create_symbol_table_object`.

The parameters for the create routine consist of a parameter for each field in the class, excluding list types, followed by the list of parameters for creating of the parent class, if any. The actual list of parameters can be modified by some options on the field declarations. These options are described in the section on the syntax of *Hoof* files. In particular, because of the limitations of C++, all parameters with default values are gathered together at the end of the parameter list. To call these create methods, you need a pointer to an instance of the associated object factory. This pointer is usually initialized during initialization of the system.

As well as the create methods in the object factory, there are equivalent global functions. By using the global functions, you do not need to know which module contains a given class, nor do you need to keep pointers to all the object factories in hand. The global functions take the active Suif Environment (a pointer to `SuifEnv`) as the first parameter. The remaining parameters are identical to the parameters of the corresponding create methods. These methods are a little slower than the calls to the object factory methods since they have to look up the correct object factory on each call. In practice, this overhead will be negligible.

```
#include "common/system_specific.h"
#include <string.h>
#include "my.h"
#include "my_factory.h"
#include "iokernel/aggregate_meta_class.h"
#include "iokernel/union_meta_class.h"
#include "iokernel/clone_stream.h"
#include "iokernel/pointer_meta_class.h"
#include "iokernel/stl_meta_class.h"
#include "iokernel/object_factory.h"
#include "iokernel/virtual_iterator.h"
#include "iokernel/field_description.h"
#include "suifkernel/real_object_factory.h"
#include "suifkernel/suif_env.h"
#include "suifkernel/module_subsystem.h"
#include "suifkernel/module.h"
#include "basicnodes/basic_factory.h"
class MyModule : public Module {
  public:
    MyModule( SuifEnv* suif ) : Module( suif ,"my") {}
    virtual void initialize() {
      MyObjectFactory * current = new MyObjectFactory();
      _suif_env->add_object_factory( current );
      }
    virtual Module *clone() const {
      return((Module*)this);
      }
    };

extern "C" void EXPORT init_basic( SuifEnv* suif );
extern "C" void EXPORT init_suif( SuifEnv* suif );
extern "C" void EXPORT init_mynodes( SuifEnv* suif ) {
  ModuleSubSystem* mSubSystem = suif->get_module_subsystem();
  if (!mSubSystem->retrieve_module("my")) {
    init_basic( suif );
    init_suif( suif );
    mSubSystem -> register_module( new MyModule( suif ) );
  }
};
```

Figure 8: My Module Initialization

```
static const LString example_class_name("Example");

const LString &Example::get_class_name() {
  return example_class_name;
  }
int Example::get_x() const{
  return _x;

  }
void Example::set_x(int the_value) {
  _x = the_value;
  }

void Example::constructor_function( Address address ) {
  new (address) Example;
  }

Example::Example() {}
Example::~Example() {
  }
void Example::print(FormattedText &x) const
  {
  x.start_block("{Example}");
  x.start_block("x=");
  x.set_value(_x);
  x.end_block();

  x.end_block();
  }
```

Figure 9: The Example Methods from `my.cpp`

6.3 Implementation of IR Modules

The `my.cpp` file, shown in Figure 9, contains the implementation of the module. The module is implemented as a subclass of the `Module` class. The initialize method, which is called when the module is registered, creates an instance of the object factory for this module and adds it to the list in the Suif environment.

Next are two `extern` declarations for the `basicnodes` and `suifnodes` modules. These are generated from the *Hoof* files `basic.hoof` and `suif.hoof` and contain the base classes for the SUIF Intermediate Representation. The `init_mynodes` routine initializes these modules before creating and instance of `MyModule` and registering it. This occurs because of the `import` lines in the *Hoof* file.

Next in the `my.cpp` file you will see various bodies for methods from the `Example` class, shown in Figure 9. These are straightforward. The print routine produces a simple print of the class and is primarily for use in debugging. `FormattedText` is a class that pretty prints the output with suitable indentation of fields.

The final portion of the file is the factory methods for `MyObjectFactory`. It is not necessary to understand the details of how the factory is defined. Of interest are the two create routines for the `Example` class, which is shown in Figure 10. The first one is the method in the object factory while the second is the global function. Notice that the global function simply finds the object factory and then calls the method. This lookup is quite fast because it is being performed using `LString`s so only a pointer comparison is needed.

```
Example* MyObjectFactory::create_example(int  x)   {
  Example* obj= (Example*)
        _object_factory->create_empty_object_by_name(
                Example::get_class_name() );

  obj->set_x(x);

  return obj;
}

Example* create_example(SuifEnv *suif_env,int  x)   {
  MyObjectFactory * current = (MyObjectFactory *)
        (suif_env->get_object_factory(MyObjectFactory::get_class_name()));
  suif_assert_message(
        current != NULL,
        ("My Object Factory not registered\n"));
  return current->create_example(x);
}
```

Figure 10: My Object Factory

7 Hoof Files in Detail

In a *Hoof* file, lines starting with "#" are treated as comments. The full syntax of non-comment lines can be found in nci/suif/suif2b/tools/smgn/suif.grm.

7.1 Module Definitions

A *Hoof* file consists of a list of modules. There will only be one module in any given file. Before code is generated, the modules that a given module depends upon are concatenated to form a temporary file so that the file actually compiled may contain multiple modules.

```
<modules> ::= <module_spec> | <module_spec> <modules>
```

Each module contains imports and declarations. The includes are just list of files that are to be included into the generated source code.

```
<module_spec> ::= module <identifier> "{" <includes> <declarations> "}"
```

For example:

```
module basic {
  include "cast.h";
  ...
}
```

The declarations declare the classes that are to appear in the modules.

7.2 Class Definitions

```
<declaration> ::= <construct_declaration> | <value_declaration> | ;
```

The (obscurely named) <construct_declaration> declares a class. The <value_declaration> declares a union.

```
<value_declaration> ::= value <identifier> = <clauses>;
<clauses> ::= <type_specification> <forwarders>
            | <type_specification> <forwarders> "|" <clauses>
```

The <identifier> is the name given to the union. Each clause in <clauses> defines one field of the union.

The <forwarders> describe fields of the type which are to be forwarded as if they were present in the union.

Construct declarations take the form:

```
<construct_declaration> ::=
          <construct_type> <identifier> <rename> <parent>
          "{"
               <construct_parts>
               <member_declarations>
               <member_definitions>
          "}" ;
<construct_type> ::= abstract | concrete
<rename> ::= [ <identifier> ] | <empty>
<parent> ::= : <identifier> | <empty>
```

The <construct_type> is ignored, as is the <rename> clause. The <parent> field gives the name of the class from which this is derived. If a class does not derive from another *Hoof* defined class, the parent must be given as SuifObject.

The body of the declarations contains a set of fields (<construct_parts>) and occasionally a list of member declarations and definitions. The declarations are just a way of introducing extra members into the classes.

The <construct_parts> contains a list of fields.

```
<construct_parts> ::= <construct_part>
                    | <construct_part> <construct_parts>

<construct_part> ::= <regular_field_declaration>
                    | <friend_declaration>
                    | <empty>

<regular_field_declaration> ::= <virtual> <type_specification>
                     <identifier> <in_clause> <implements_clause>
                     <may_be_present> <key_dec> <notify>;

<virtual> ::= virtual | virtual default "{" <verbatim> "}" | <empty>
<in_clause> ::= in <identifier> | <empty>
<implements_clause> ::= implements <identifier> | <empty>
<key_dec> ::= key | <empty>
<notify> ::= notify | <empty>
```

A <friend_declaration> just allows you to declare a friend class as you would in C++. The field declarations are all <regular_field_declarations>.

A field declaration may be virtual. This means that the field does not actually exist in the class but will exist in classes derived from this one. The methods that would be associated with such a field are declared in this class in the resulting code but they are virtual and will be overridden in the derived classes. This is useful when various derived classes will require different implementations of the virtual field.

You can give a default as part of the virtual field. Normally, if you call the get method associated with a virtual field, you will receive an assertion failure. The default associated with the virtual keyword allows you to give some other code to be performed. A common case is to write:

```
virtual default return 0;
```

The <type_specification> gives the type of the field. These are discussed in the section of type specifications below.

The <identifier> in the field declaration is the name to be given to the field. This name is used in all the methods. The name of the field in the generated code will have an underline added at the start.

The <in_clause> is used to specify that this field is part of a group of fields that can be iterated over. For example, the declaration of conditional statements states the field condition is an operand:

```
Expression owner * condition in operands;
```

The <implements> clause declares that this field implements a field that has been declared virtual in a parent class. The identifier in the implements clause is the name of the implemented field.

The <key_dec> clause declares that this field is a key for this class. An equality operator will be defined for this class which takes the field type as parameter and, if the class is used as the element type of a searchable_list find routine will be generated for the searchable list using this field type as key.

The <may_be_present> clause defines a number of possible options for the initialization of the field when an object of the class is constructed using the create_ routine defined in the object factory.

```
<may_be_present> ::= optional | <default_value> | <build_value> |
                     <omitted> |
<omitted> ::= omitted
<default_value> ::= default "{"<verbatim>"}"
<build_value> ::= build "{"<verbatim>"}"
```

A field which is declared optional will be given a default value of 0 in the create_ routine. The type must be such that this is acceptable. If some other default value is wanted, the default_value qualifier should be used instead.

The build_value qualifier is used for fields that require the default value to be built in the context of the object factory rather than the context of the call. This is usually only used for pointers. The default value will be 0 (NULL) but a real value will be build inside the create routine. This is used, for example, when building a definition block to avoid having to build the symbol tables that it will contain explicitly.

Finally, if a field is declared as omitted it will not appear in the list of parameters in the create routine at all. Instead, it will be assigned the value 0 in the create routine.

Default values are a problem with C++ because of the lack of named parameters. A derived class's default parameters will appear before the non-defaulted parameters of the parent. This causes compilation errors on some compilers.

7.3 Type Specifications in Hoof

Our description of *Hoof* files is now complete except for the description of the type specifications given to fields. While the specifications themselves are fairly simple, there are a number of predefined suif types for which the semantics must be understood in order to fully understand *Hoof* file type declarations.

The syntax is:

```
<type_specification> ::= <type_spec> | <type_spec> * <pointer_qual>
<type_spec> ::= <identifier> <template_part>
<template_part> ::= "<" <type_specifications> ">" |
<pointer_qual> ::=  owner | reference
```

A type specification is either a simple identifier, a template type, or a pointer to either of these. In the case of a pointer type, the is followed by either `owner` or `reference` to denote whether the object pointed to by this field is owned by this object.

7.4 Collection Types

There are a number of predefined template types in SUIF. These are handled differently from ordinary types in a number of ways:

1. The fields are omitted from the create routine parameter lists.
2. There are additional methods associated with the symbols.
3. The type names in the *Hoof* file can differ from the type names actually used in the implementation. Different *Hoof* type can be used to get different methods even though the implementations of the fields is the same.

In the following descriptions of the collection types, the term *field* means the name of the field declared in the *Hoof* file. If it is plural, it will be converted to the singular. *element_type* is the type of element for a list. *key_type* is the key type of an indexed list.

7.4.1 Simple List

The declaration

```
list<element_type> field;
```

declares a simple list with the following methods:

```
void append_field(element_type x);
int get_field_count() const;
Iter<element_type> get_field_iterator();
void insert_field(int pos, element_type x);
element_type remove_field(int pos);
element_type get_field(int pos);
```

7.4.2 Searchable List

The line

```
searchable_list<element_type>field;
```

declares a searchable list. Each entry has an equality operation defined on it. It may also contain secondary keys, defined by the "key" field attribute. These keys may occur multiple times. The following methods are defined:

```
void add_field(element_type key);
Iter<element_type> get_field_iterator();
void remove_field(element_type key);
bool has_field_member(element_type key);
```

(for any fields of *element_type* with key attribute)

```
bool remove_field_by_key_key_field (const key_type);
element_type lookup_field_by_key_key_field(const key_type);
```

(To support multiple occurences of keys)

```
int num_field_of_key_field(const key_type);
element_type lookup_field_by_key_field(const key_type, int no);
bool remove_field_by_key_field(const key_type,int no);
```

7.4.3 Indexed List

The declaration

```
indexed_list<key_type, value_type> field;
```

defines a list with an external primary key which is stored in the list. Compare this to the searchable_list in which the key (or keys) is part of the element type. It should really be possible to look these up using a secondary key defined with the key attribute but that is not currently needed or implemented. These are the operations defined:

```
void add_field(key_type, value_type);
typedef indexed_list<key_type, value_type>::pair field_pair;
Iter<field_pair > get_field_iterator();
<value_type> remove_field(key_type);
bool has_field_member(key_type);
int num_field_with_key(key_type);
value_type lookup_field(key_type, int no);
value_type remove_field(key_type, int no);
value_type lookup_field(key_type);
```

7.4.4 Vector

A vector type is declared with the line

```
vector<element_type> field;
```

and has the following methods:

```
void append_field(element_type);
int get_field_count() const;
Iter<element_type> get_field_iterator();
void insert_field(int pos,element_type);
void remove_field(int pos);
element_type get_field(int pos);
void replace_field(int pos, element_type);
```

8 The Makefile System and System Installation

A simple but flexible make system is used by the system. It is driven by the file `Makefile.std` in the `nci` directory.

At present the system works on Linux, OSF (Alpha) and Sun Solaris. If you are porting to other systems, you may encounter problems. In that case, please ask for help if you need it, and we would like to merge any changes you have to make back into the main system. We recommend that you use egcs version 2.91 or later, or the newest merged gcc, 2.95 or later. (The system will also compile with Microsoft C but the system build process is totally different).

Makes can be initiated in any directory, just by typing "`make`", or possibly "`gmake`" if your `make` command does not invoke `gmake`. The directory and all its sub-directories are made.

A number of options files are available for configuration of the make system. These files are all included with the "`install`" command so that no error is given if the file does not exist. The options files are included in this order:

```
-include $(NCIHOME)/Makefile.options.$(COMPILER_NAME)
-include $(NCIHOME)/Makefile.options.$(ARCH)
-include $(NCIHOME)/Makefile.options.$(HOSTNAME)
-include $(NCIHOME)/Makefile.options.$(LOGNAME)
```

Normally, the last two will not exist. These allow you to configure the make system for a particular platform or for a particular user.

You should avoid changing the `Makefile.std` except to fix bugs therein since if your changes are not accepted back into the main distribution you will have to continue changing this in the future.

The following variables may need to be changed in options files for your environment. Some of this is handled automatically by the install script described later:

variable	default	description
`SHARED_LIB_FLAGS`	`-shared`	Flag required on build of shared library
`DYNAMIC_LINKING_LIBRARY`	`-ldl`	Library containing dynamic link routines (dlopen etc)
`STATIC_LIBEXT`	`.a`	Extension used for static libraries
`DYNAMIC_LIBEXT`	`.so`	Extension used for dynamic libraries
`CCFLAGS`	`-g`	Compiler flags
`SYSINCLDIRS`	`-I. -I$(BASETYPEDIR)`	Standard include directories
`DEPSFLAG`	`-M`	Flag used with C++ compiler to generate dependencyfiles
`CC`	`gcc`	Compiler name for C compiler
`CXX`	`g++`	Compiler name for C++ compiler
`AWK`	`gawk`	The name of the awk processor, Usually gawk or nawk, not plain awk

8.1 Variables in Makefiles

The make files in each directory contain a number of variables with standard names. Note that not all of these variables are defined in every directory. If a directory contains no *Hoof* generated files, none of the *Hoof* related items will appear. Most directories either produce a library or a program, so only one of `TARGET_LIB` or `PROGRAM` will usually appear. The list of variables are:

Variable	Description
PACKAGES	A list of the subdirectories that are to be recursively made.
SUIF_INTERNAL	Define this to prevent the suif libraries being included automatically.
SRCS	List of source files in the directory. Used to build the dependencies file.
HEADERS	List of header files in the directory.
OBJ_FILES	List of object files created by directory
HOOF_GEN	List of files generated from *Hoof* files.
SMGN_INCLUDES	Files included in `suif.mac`. This is in the wrong place. It really should be in the `smgn` directory but in fact you have to include it in the target directory.
HOOF	Name of the module generated from *Hoof* in this directory.
TARGET_LIB	Name of target library built in this directory.
INCLDIRS	Extra include directories required to compile.
LIBS	Extra libraries required to compile.
PREPROCESSORFLAGS	Extra preprocessor flags for compilation
PROGRAM	Name of program built by this directory.

8.2 The Install System

SUIF comes with a simple installation script that will attempt to identify various components of your system and set some variables in `Makefile.std` appropriately. If you are installing on a system for which we have not catered, there is a good chance you will have to modify these scripts. Please ensure that when you do modify the scripts the existing funcionality is retained, and please send us your changes for inclusin in the release. This is especially important if you are installing the system on a new machine or with a new compiler.

Installation of the system requires the following steps:

- Untar the release.
- `cd` to the nci directory.
- type "`./install`"
- type "`make setup`" (or "`gmake setup`")
- type "`make`" (or "`gmake`")

The following files are part of the install system. These are all in the "`nci`" directory:

File	Description
`install`	The installation script
`Makefile.proto`	The prototype for `Makefile.std`
`packing.lst`	List of files present in release
`Makefile.std`	Generated by the install script from the prototype Makefile and from variables deduced by the install script (see below)
`nci_setup.csh`	Script to "source"; to set environment variables for use with nci system
`nci_setup.sh`	Script to "." for users of Bourne Shell and derivatives.

When you run the `install` script, a number of variable values are deduced and these are placed into `Makefile.std` together with the contents of `Makefile.proto`. The deduced variables are as follows:

Variable	Description
ARCH	First thing from "uname -a" output
NCIHOME	Always the directory the install script is run from.
MAKE	gmake if possible, else make. Note that the Makefiles use a number of gmake features and are not compatible with many other make programs.
AWK	Looks for gawk, then nawk, then awk. On SunOS, awk does not work; you need at least nawk. This should be available.
CXX	C++ compiler name. Looks for eg++, g++, CC and cc, in that order.
CC	C compiler name. Looks for egcc, gcc and then cc, in that order.
COMPILER_NAME	This is important because it is used to access option file names. It is the variable you will most likely want to change. The install script guesses a value for this based on the COMPILER_NAME and the ARCH values.
PERL	Currently always perl.

8.3 Building in Remote Directories

If you are working in a group, you may find it convenent to share source code and build in a separate area to the source. Unfortunately, this is not yet supported. Adding this capability should not be hard. gmake has capabilities to work with object files and source files in separate directories. Changing the location of the library files and executable should be easy as these are already parameterized using make variables. If you do this, please send us the results.