```
class Statement : public ExecutionObject {
  public:
        static const LString &get_class_name();
        virtual Iter<Statement* > get_child_statement_iterator() const;
        ...
}
class IfStatement : Statemet {
  public
        static const LString &get_class_name();
        virtual Expression* get_condition () const;
        virtual Expression* set_condition (Expresion* the_value);
        virtual Statement* get_then_part () const;
        virtual Statement* set_then_part (Statement* the value);
        virtual Statement* get_else_part () const;
        virtual Statement* set_else_part (Statement* the value);
        ...
}
```

4 Conclusions

This document only contains an introduction to the key concepts in the SUIF2 infrastructure. More documentation is available via http://suif.stanford.edu/. Included on the document are: a more detailed description of the SUIF program representation, a current list of all the available programs, libraries, and utilities, some examples, and documetation of the important interfaces of the system.

5 Acknowledgment

Chris Wilson was a key member of the SUIF2 team and was responsible for many of the ideas found in the initial design of the SUIF hierarchy, much of which are still present in the current system. He also wrote the first prototype of the system. This work was supported by the National Compiler Infrastructure grant, which was provided by both Darpa and NSF.

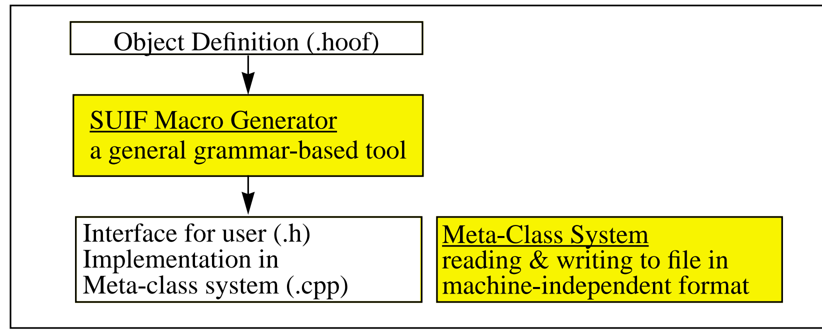

Figure 3.4 Generating New IR Nodes

```
}
```

SMGN will generate the following C++ class declaration for the Example class.

```
class Example : public SuifObject
{
  public:
    static const LString &get_class_name();
    int get_x();
    void set_x(int the_value);
    ~Example();
        ...
  protected:
    friend class TestObjectFactory;
    Example();
        ...
  private:
    int _x;
}
```

The default base class for all classes defined in hoof is SuifObject. The class name can be accessed with the get_class_name method. Public accessor methods get_<fieldname> and set_<fieldname> are automatically generated for reading and writing the private data to represent the declared field. It has a public destructor, but no public constructor, as the user should create all objects using an object factory with the name <ModuleName>ObjectFactory (Section 3.2.1).

3.3.2 Examples of IR Nodes

Here is the definition of two IR node definitions to illustrate how multiple levels of abstractions are specified in the hoof specification language.

```
abstract Statement : ExecutionObject
{
        virtual list<Statement* owner> child_statements;
        ...
}
concrete IfStatement : Statement
{
        Expression * condition in source_ops;
        Statement * owner then_part in child_statements;
        Statement * owner else_part in child_statements;
}
```

The Statement class includes an abstract iterator child_statements. The IfStatement class is derived from Statement. It has a condition (which will be iterated over in the source_ops iterator in the ExecutionObject and a then_part and else_part which make up the child_statements in the Statement class. The C++ interface generated by SMGN is:

FileSetBlock
    external_symbol_table
        symbol_table_objects
    file_set_symbol_table
    file_blocks
        symbol_table
        definition_block
            variable_definitions
                initialization
            procedure_definitions
                body
                    source_ops
                symbol_table
                definition_block*
    information_blocks
BasicSymbolTable, GroupSymbolTable
    symbol_table_objects
ExecutionObject
    source_ops
Statement
    child_statements
StatementList
    statement (child_statements)
ScopeStatement
    body (child_statements)
    symbol_table
    definition_block
WhileStatement, DoWhileStatement
    condition (source_ops)
    body (child_statements)
ForStatement
    lower_bound (source_ops)
    upper_bound (source_ops)
    step (source_ops)
    body (child_statements)
    pre_pad (child_statements)
IfStatement
    condition (source_ops)
    then_part (child_statements)
    else_part (child_statements)
BranchStatement, MultiBranchStatement
    decision_operand (source_ops)
JumpIndirectStatement
    target (source_ops)
ReturnStatement
    return_value (source_ops)
StoreStatement
    value (source_ops)
    destination_address (source_ops)
StoreVariableStatement
    value (source_ops)
CallStatement
    callee_address (source_ops)
EvalStatement
    expressions (source_ops)
VaStartStatement, VaStartOldStatement, VaEndStatement
    ap_address (source_ops)

Expression
BinaryExpression
    source1 (source_ops)
    source2 (source_ops)
UnaryExpression
    source (source_ops)
SelectExpression
    selector (source_ops)
    selection1 (source_ops)
    selection2 (source_ops)
LoadExpression
    source_address (source_ops)
LoadValueBlockExpression
    value_block (source_ops)
ArrayReferenceExpression
    base_array_address (source_ops)
    index (source_ops)
MultiDimArrayExpression
    base_array_address (source_ops)
    offset (source_ops)
    index (source_ops)
    elements (source_ops)
FieldAccessExpression
    base_group_address (source_ops)
VaArgExpression
    ap_address (source_ops)

SuifBrick
OwnedSuifObjectBrick
    object
AnnotableObject
    annotes
BrickAnnote
    bricks
Type
ArrayType
    lower_bound
    upper_bound
MultiDimArrayType
    lower_bounds
    upper_bounds
GroupType
    group_symbol_table
Symbol
FieldSymbol
    bit_offset
NestingVariableSymbol
    bit_offset
ValueBlock
ExpressionValueBlock
    expression (source_ops)
MultiValueBlock
    sub_block
RepeatValueBlock
    sub_block

Figure 3.3 Ownership Relationships of ExecutionObjects

3.2.1 Object Factories

We centralize the creation of objects using the concept of object factories. This design makes it possible to experiment with different management schemes in the future by changing the implementation of these factories.

In the SUIF system, there is no public constructor, so new cannot be used to create an instance of the object. Instead, associated with each module of IR nodes is an ObjectFactory responsible for creating all objects defined in the module. The object factory is added to the SuifEnv at the time the module is initialized, and it can be accessed by the method SuifEnv::get_object_factory. For each class of name "ClassName", there is a function called "create_class_name" method in the ObjectFactory for the creation of instances of the objects. More specifically, the name of this routine is derived by appending the word create to the name of the class after appending an "_" before each capital letter and converting all letters to the lower case. 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 the create 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 addition to these create methods, there are global functions which serve the same purpose. They have exactly the same names as those in the object factory. The parameters are also the same except that there is an additional parameter, given at the beginning of the parameter list, for a pointer to a SUIF environment. These global functions remove the need to know which factory creates a given type. They are a little slower than the calls to the object factory methods since they introduce an additional indirection.

3.2.2 Ownership Tree

A SUIF program is a collection of nodes organized mainly as a tree with occasional cross references between nodes. Roughly, a program owns a symbol table information, which contains the definitions of types and variables, and computations, which own procedure, which in turn own statements and expressions. All this information can naturally be represented as a tree. Accesses to variables in expressions, on the other hand, are cross references between the different components of the tree. This concept of an ownership tree is helpful to SUIF compiler writers for managing memory. For example, when we wish to free an object, we should free all the objects owned, but not references to objects owned by other entities.

Pointers in the SUIF representation are thus either classified as a "ownership" pointer or a reference pointer, which should not be confused with C++ references. We use the convention of using a keyword "owner" in the definition to identify the ownersihp links. A node can be pointed to by an arbitrary number of reference pointers. With the exception of the root node (an instance of class FileSetBlock) each node is pointed to by exactly one ownership pointer. We show the owner relationships in the SUIF representation in Figure 3.1 .

3.3 Hoof Specification Language and the SUIF Object Schema

The user can define new IR nodes without knowing the details of how the metaclass system works by using a high-level specification language called hoof. The SUIF Macro Generator (smgn) automatically translates the hoof representation to an interface file (.h) and a C++ implementation file (.cpp). Smgn is a ge neral tool as it can be parameterized by a grammar file and an action file. Using Smgn, we have been able to experiment and evolve the IR implementation easily by changing the grammar and the action files without impacting the users of the system. More over, the consistency and uniformity across the IR nodes obtained as a result of using a tool to generate interfaces according to a common schema makes it easier for new users to learn the system.

3.3.1 A Simple Example

Here is a simple example of a Hoof specification in a file named test.h that defines a new TestModule. The module one class known as Example, which has only one component x.

```
module test {
        concrete Example
        {
                int x;
        }:
```

- ExecutionObject. All IR nodes representing computation are derived from the ExecutionObject. It is an "abstract object" meaning that all instances of this class are instances belonging to its derived subclasses. Derived from the ExecutionObject class are the Statement and Expression subclasses. Statements capture the changes to the state of the execution, which includes control flow and the contents of the memory store and Expressions represent expression trees. There is a large collection of nodes derived from statements and expressions to capture different program semantics that are useful for program optimizations.

3.1.1 Multi-Level Representations

There is no single level of program representation that suffices even in a compiler for a specific language and machine. Typically we have high-level program transformations such as parallelization and loop-level transformations that have to be applied to constructs like loops and array index expressions, and simple data flow passes that typically operate at the lower level. Our design is to provide a large collection of different IR nodes to allow the program to be represented at multiple levels. For example, there are statements that range from high-level constructs such as For and While loops and simple store statements and branch statements and expressions that range from multidimensional array accesss to simple arithmetic operations. It is not necessary for any pass to work on all the nodes in the representation.

The expected mode of operation is for the front end or control flow restructurer to represent the program at the highest level. Higher-level transformers are first applied to the program. Passes that dismantle higher level constructs to lower level constructs are then used to generate a representation that can be handled by the lower level program analyses. It is sometimes desirable to have the same functionality at both a high and low level, examples include dead code elimination. This can be achieved by buiding an analysis that can operate on a richer set of nodes, rather than two different passes which would be necessary if we have two totally different levels of program representation.

3.1.2 Levels of Abstractions

Our design is designed to make it easy to write reusable passes at different levels of abstraction. It is easy for users to write passes that exploit the semantics of the individual constructs as statements and expressions have component names that are mnemonic of their semantics. For example, an IfStatement has a then_part and an else_part; a BinaryExpression has a source1 and source2; a LoadExpression has a source_address.

It is also easy for users to write passes that are indifferent to the specific semantics. The object hierarchy enables us to define useful abstractions that hide details that can be safely ignored by many program passes. Three different levels of abstractions are available in the SUIF system today:

- source_ops, source_vars and defined_labels. The ExecutionObject offers three iterators that visit all the *roots* of the expression tree**s** used in the Executionobject (source_ops), all the source variables accessed directly by name (source_vars) and all the labels defined (defined_labels).
- child_statements and destination_vars. Statements offer iterators that list all the nested statements (child_statements) and all the variables that have been written to directly by name (destination_vars).

To maximize its reuse potential, a pass should be written to operate at the highest level of abstraction. For example, a data flow analysis should use the generic iterator source_ops to refer to all the different operands in all the IR nodes if it does not care about the kinds of expressions being processed. Our system allows the same pass to operate on SUIF programs that have been defined with instances of lower-level IR nodes unknown at the time the pass was compiled. This is achieved by explicitly associating components in an IR node with the abstract iterators of its base objects.

3.2 Memory Management Concepts

Memory management is an important consideration in developing an efficient and easy-to-maintain compiler system. The SUIF system design includes two concepts, object factories and ownership trees, to aid users in managing memory.

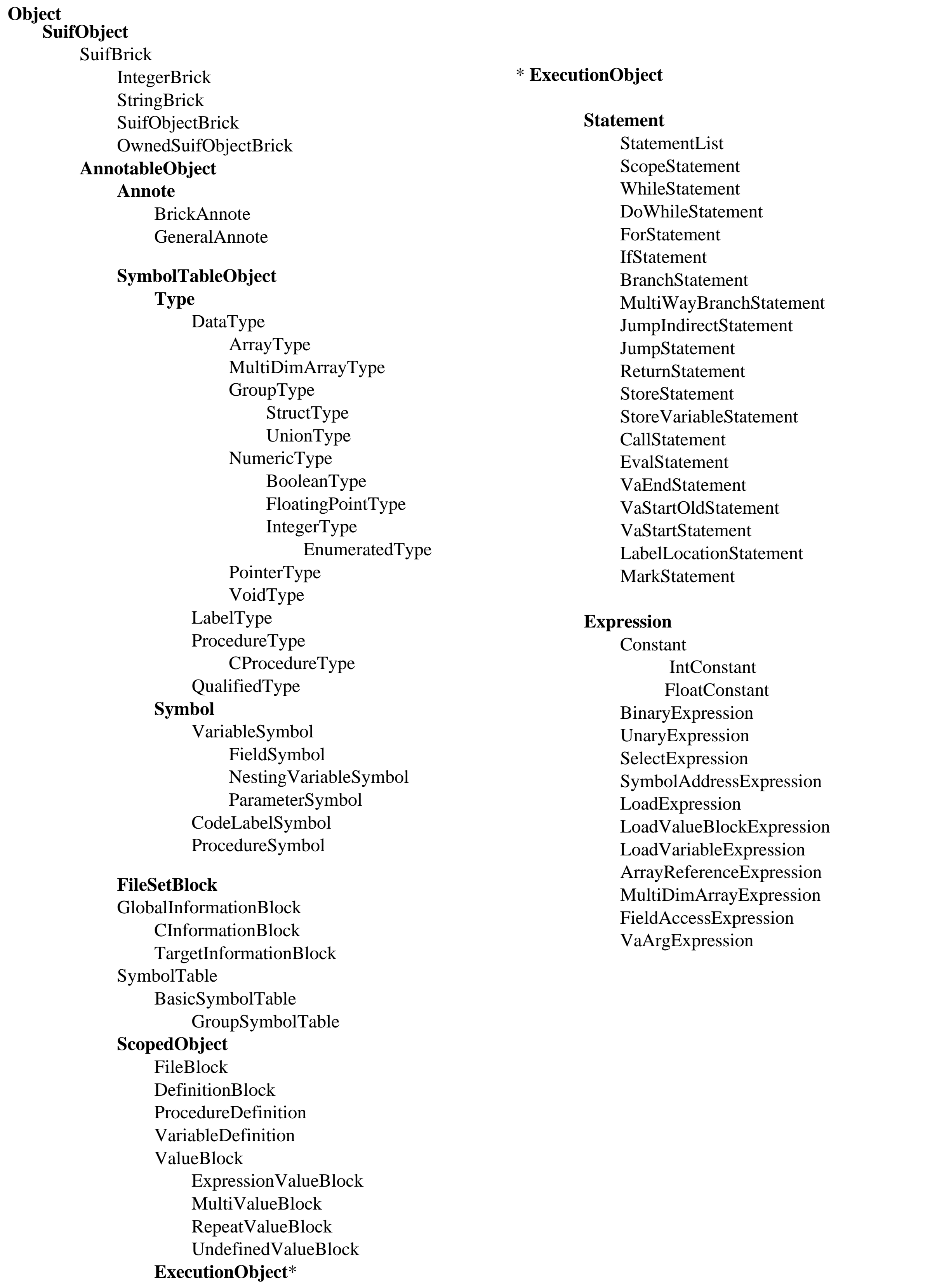

Figure 3.1 SUIF IR Hierarchy

that operate on the highest semantic level to maximize reuse. The system is designed so that a pass written and compiled without the knowledge of a subclass can still operate on SUIF programs created with instances of the unknown subclass. This promotes sharing among different research groups

3.1 The SUIF Object Hierarchy

The highest and most important levels of the IR object hierarchy are shown in Figure 2.1and the complete SUIF object hierarchy is given in Figure 3.1 All the higher levels in the object hierarchy are defined in the basic module (basesuif/basicnodes/basic.hoof) and all the different kinds of statements, expressions and types are defined in the suif module (basesuif/suifnodes/suif.hoof). Details on the semantics of the IR are in the Suif Program Representation Guide.

At the top of the hierarchy, are the Object, SuifObject and AnnotableObject classes; they each represent an important abstraction and do not have any special program semantics. Derived from these three classes are all the SUIF IR nodes that represent different components of a program.

- Object: Encapsulated by the Object is the metaclass information describing the contents of an IR node. This support for reflection is used by the IOkernel to implement a persistent object model. Users need not know anything about the interface or details of this level of abstraction.

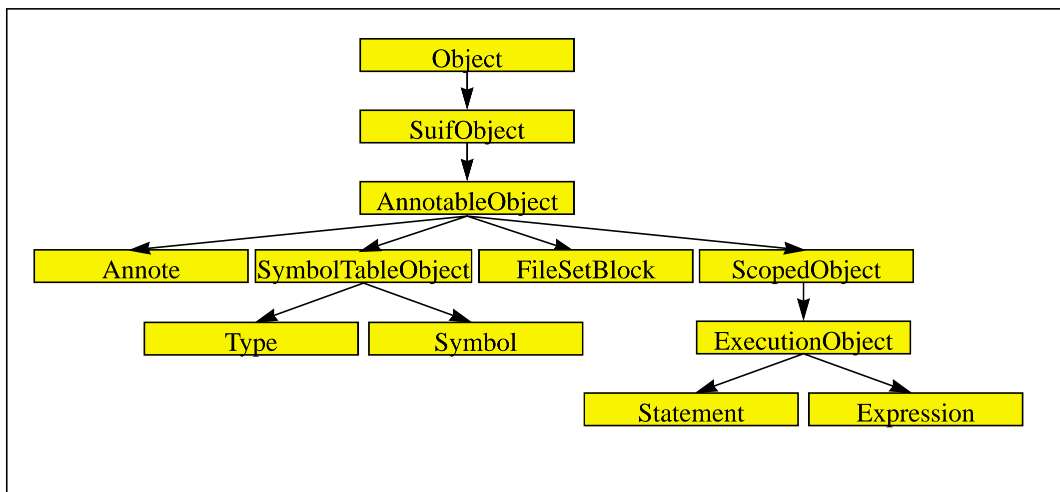

Figure 3.2 Top Levels of the IR Object Hierarchy

- SuifObject: SuifObject provides all the basic user-level functions such as printing, cloning, and data structure traversals. This wraps the Object interface and insulates it from the user.
- AnnotableObject: Annotations is the mechanism we provide to allow compiler passes to hang the derived information about a program to the various parts of a program representation. In this system, we can hang annotations on all objects, except for the primitive building blocks (SuifBricks) that we use to construct annotations.
- Annote: Annotations are themselves annotable objects. Programmers can use annotations to capture whatever information they like to keep with the intermediate representation.
- SymbolTableObject: This has two major subclasses, types and symbols.
- FileSetBlock: The FileSetBlock is the root of the representation of a program. It contains some global information about the program, symbol tables, and the procedure definitions, etc.
- ScopedObject. With the exception of a small amount of global information, everything in a program is usually defined within some scope, and is subclassed from the ScopedObject. This includes the definitions of procedures and variables, the initialization operations. Currently, no methods are associated with this object, and this class is defined as a placeholder for potential future development.

```
    return inherited::parse_command_line(command_line_stream);
  }
```

Further details about programming the command line option can be found in the Suif Programming Guide.

- *Execute the module.* If the command line was successfully parsed, the ModuleSubSystem will invoke the Module via the Module::execute() method. In either case, the Module::delete_me method is invoked to determine if module needs to be deleted, and the ModuleSubSystem acts accordingly. The delete_me method returns true if the clone method returns a new instance of the module.

2.2.2 Pass and PipelineablePass Classes

As discussed above, a Module can be either a set of IR nodes or it can be a compiler pass. Section 3 will describe how the user can define new IR nodes, and here we present two important abstractions that are helpful for defining new compiler passes. The Pass class is derived from the Module class, and the PipelineablePass is derived from the Pass class. The user can define new passes by simply subclassing from these classes and specify only the functions to be applied to the various components in a program representation (e.g. the global symbol table, the procedure definitions, the variable definitions etc).

The standard method is to apply a pass to all the procedures in a SUIF program before applying another procedure. PipelineablePasses, however, allow the freedom to apply the passes in a pipelined fashion. That is, the driver can operate on a procedure at a time; it can apply a series of different passes on the same procedure before applying them to another procedure. Pipelining the passes improves the locality of the compiler, which can be important for large programs.

2.3 Drivers

To create a compiler or a standalone pass, the user needs to supply a "main" program that creates the SuifEnv, imports the relevant modules, loads a SUIF program and applies a series of transformations on the program and eventually writes out the information. The sequence of operations is shown in Figure 2.2 The suifdriver described in Section 1 is one such driver that allows the user to dynamically specify the components and passes applied.

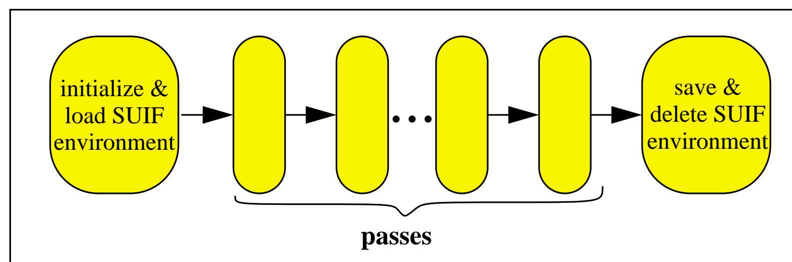

Figure 2.4 A Typical SUIF Compiler

3 The SUIF Extensible Program Representation

Experience in developing compilers all the years taught us that it is impossible to define a program representation that captures all the concepts useful in compilation. We may wish to capture high-level program semantics of new program constructs such as object-oriented class definitions and methods or the event-driven primitives in hardware description languages. Even with traditional programming languages, it is sometimes necessary to augment the representation to capture information necessary for new program analysis. For example, traditional compilers typically represent array element accesses as a series of arithmetic operations. To support data dependence analysis, which is important for loop transformations and instruction level parallelism, it is important that the original array index expressions are retained.

Extensibility in SUIF is provided by an extensible class hierarchy. We have developed an object hierarchy that captures the different levels of abstraction useful for representing programs in compilers. Users can derive new objects from the lowest level of the hierarchy that shares the same semantics. Users are encouraged to write codes

More specifically, when a new library *lib* is imported, the import module invokes the init_*lib*(SuifEnv*) method included with every library. An example of such a function is shown in Figure 2.3. The function invokes the register_module() method in the ModuleSubSystem to register each module in the library.

```
#include "suifkernel/module_subsystem.h"
#include "suifkernel/suif_env.h"

extern "C" void init_mylibrary( SuifEnv* );

void init_mylibrary( SuifEnv* suif_env ) {
  ModuleSubSystem* module_subsystem =
                      suif_env->get_module_subsystem();

  module_subsystem -> register_module( new MyPass( suif_env ) );
}
```

Figure 2.3 Registration of a Module

The register_module method creates an instance of the module in the library, uses the get_module_name method to find the module's name, and registers the module by name. If the module has already been registered, it deletes the prototypical module and exits quietly, as importing the same module multiple times is not considered to be an error.

There are three steps that the ModuleSubSystem when a module is invoked:

- *Initialize the variables in the module*. While we wish to register a module only once, the same module may be invoked multiple times. A module may or may not contain states to keep the intermediate and final results of the analysis. Thus, when a module is invoked, the ModuleSubSystem retrieves the prototype module by name (Module::get_module_name) and clones it (Module::clone). The clone method returns a new instance of the module if it contains state, and returns the prototype module otherwise. It then checks if it has been initialized (Module::is_initialized) and initializes the clone Method::initialize) if not.
- *Pass parameters to the module*. A compiler pass often allows the user to customize its execution by specifying a number of options. We have adopted the familiar *command line* model as a flexible mechanism for communicating parameters to a module. Typically found in the initialization routine is the code that defines the grammar of the command lines accepted by the module. A typical example of an initialization routine is shown below:

```
void initialize()
  // call the inherited initialize method
  inherited::initialize();
  // initialize the variables in the module ...
  // create the parse tree for parsing the command line
  _flags->add(new OptionFlag( "-flag1", &_flag1, false));
  _flags->add(new OptionFlag( "-flag2", &_flag2, true));
}
```

  The code first invokes the initialization routine of the inherited object, initializes its own variables, and specifies the command line options accepted by the module. The ModuleSubSystem accepts a stream of token from the driver invokes the Module::parse_command_string method on the token stream. The method returns a boolean specifying whether the command string was parsed properly. The parse_command_string method in the Module base class will automatically parse the command string according to the information set up by the initialization routines; the user's parse_command_line needs only to initialize the flags defined by the initialization routines, and invokes the inherited::parse_command_line.

```
bool parse_command_line(TokenStream* command_line_stream) {
  // initialize all flags to their default value
  _flag1 = true;
  _flag2 = false;
  _string1 = "string1";
  // then the original method is called
```

```
class Module {
public:
  Module (SuifEnv* suif_env, const LString& moduleName = emptyLString );
  virtual ~Module();

  virtual SuifEnv *get_suif_env() const;
  virtual LString get_module_name() const;
  virtual Module* clone() const = 0;
  virtual bool delete_me() const;
  virtual bool is_initialized() const;
  virtual void initialize();
  virtual bool parse_command_line( TokenStream* command_line_stream );
  virtual void execute();
  ...
};
```

get_suif_env	Retrieves the SUIF environment this module belongs to.
get_module_name	Retrieves the name of this pass. This name must be unique. The recommended way to initialize the return value of the get_module_name method is to set the inherited instance variable _module_name in the constructor of your pass. Overriding the get_module_name method is not necessary.
clone	A module may include instance-specific data structures. If it does not, clone can simply return the prototype, i.e., `clone() {return (MyModule*)this;)` Otherwise, the clone operation must create a new instance of this class, i.e., `clone() {return new MyModule();}`
delete_me	Delete_me returns whether instances of modules need to be deleted.
is_initialized	Returns whether the method initialize has already been invoked.
initialize	This method contains the code that initializes a given module. It is called before execution happens and if is_initialized() returns false. The user typically creates the parse tree used for parsing command line options in this function and also initializes instance variables to the value expected by the execute function.
parse_command_line	This method is called while parsing the command line. Parse_command_line returns true if it succeeds in parsing.
execute	Execute is invoked by the SUIF environment whenever an execution of a module with that name is requested

Figure 2.2 Interface of a Module

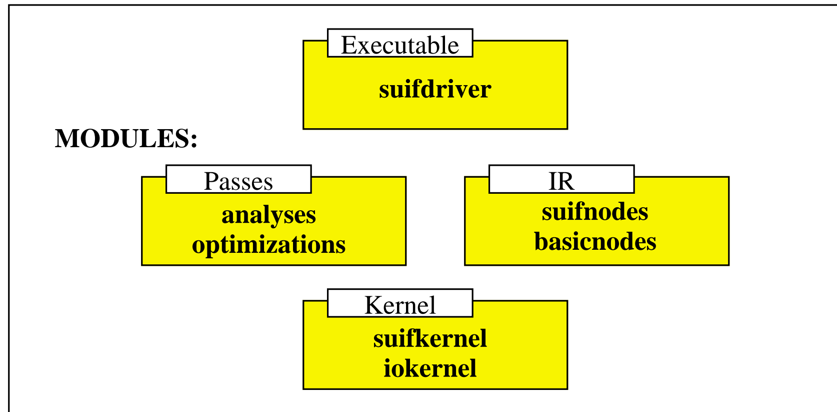

Figure 2.1 The SUIF System Architecture

- the program representation. This is the SUIF intermediate representation of the program currently stored in the SUIF environment.
- the object factory. The system uses this factory internally. It creates all persistent objects in the environment.
- the subsystems. An object of type SuifEnv has multiple subsystems that have distinct duties. There are subsystems for printing node information, printing error information, cloning of trees, dynamically loading SUIF programs, and the initialization and registration of modules. Dynamically registered modules will become part of the subsystems accessible from the environment.

2.2 Modules

Every library includes a registration function (init_<dllname>), which is invoked when the library is loaded to register all the modules in the SuifEnv dynamically.

The bulk of the SUIF compiler system is structured as modules, each of which is a C++ class identified by a unique module_name. A dynamically linked library (dll) contains one or more modules. The dll incudes an init_<dllname> function, invoked at the time the library is loaded, that registers all the modules within the library in the SuifEnv. The system comes with a number of basic modules, as well as some tools to help user construct their own modules. Modules can be one of two kinds:

- a set of nodes in the intermediate representation. The infrastructure includes a set of basicnodes, which represents a number of basic programming constructs, and a set of suifnodes, which captures standard programming constructs in standard languages such as C and Fortran. Users can easily define new program representations using hoof a higher level specification language.
- a program analysis pass. The infrastructure comes with a number of basic modules such as loading and printing a SUIF program. Users can easily define new passes by deriving them from basic pass modules and supplying the specific processing functions.

2.2.1 The Module Subsystem

The ModuleSubSystem, which is a part of the SuifEnv, is the central repository that keeps a list of all known modules in the system. The ModuleSubSystem has two important functions: it registers modules when a library is loaded, and it invokes modules, passing to them all the relevant parameters. A short description of the interface of a module is shown in Figure 2.2 .

applies the mypass code on the loaded SUIF file, prints the result textually in a separate file known as test.out, and saves the result of the optimization in binary format in a file test.2.suif. Note that import, load, print, and save are just simple pre-defined and pre-registered modules in the suifdriver.

1.3 Available Compiler Construction Tools

The infrastructure is designed to provide programmers with tools with which he or she can develop new compilers. The system provides the following set of tools and capabilities:

- Basic infrastructure. The basic infrastructure allows users to extend the system with richer program represenatations while using the functionalities provided by the system. The system automatically provides persistent data structures, allowing the program representation to be saved to disk if so desired.
- Available programs and passes. We have developed a set of basic compiler functions that the user can assemble into simple compilers. Work in progress includes an interprocedural analysis framework and program transformations for parallelism and locality.
  - front ends
  - converters from SUIF1 to SUIF2 and vice versa
  - converters from SUIF2 back to C
  - dismantlers to break down higher level constructs to lower levels
  - useful passes to check the integrity of the IR of a program.
- Tools for developing new components
  - a high-level specification of objects
  - classes from which passes can be derived
  - program traversal tools
- Tools for examining the SUIF representations.
  - sbrowser. It allows the user to examine the SUIF files and the sources files from which they are derived.

1.4 Organization of This Document

Section 2 will give an overview of the SUIF compiler architecture, and Section 3 will describe the SUIF extensible IR design. This document is intended only as introduction to the SUIF compiler system and does not provide all the detail necessary for programming a SUIF pass. More information is available on http://suif.stanford.edu.

2 The SUIF Architecture

The SUIF System has a simple and modular architecture, as shown in Figure 1.1. It comprises a small *kernel* which implements a set of basic functions found to be useful across all compilation passes, a number of *modules* loaded dynamically under user control, and a *driver* that controls the system operation.

2.1 The Kernel

The kernel itself consists of two layers: the IO kernel and the SUIF kernel. The IO kernel implements a persistent object system that is independent of the applications of writing compilers. Reflection is supported in the objects in this system by explicitly keeping their compositions in data structures known as *meta* objects. This information along with the contents of the objects are written out and read from a file. The kernel can thus restore data from a file without having to be compiled with the definitions of the objects. The details of how this meta information is kept are insulated from the user and are described in a separate document.

The SUIF kernel defines and implements the Suif compiler environment (SuifEnv) that is all the user needs to know when writing a SUIF program. The SuifEnv holds the entire state and components of the compiler system, and there are no other global variables or states in the system. A SUIF program would start by creating an instance of the SuifEnv object, which is made up of the following components:

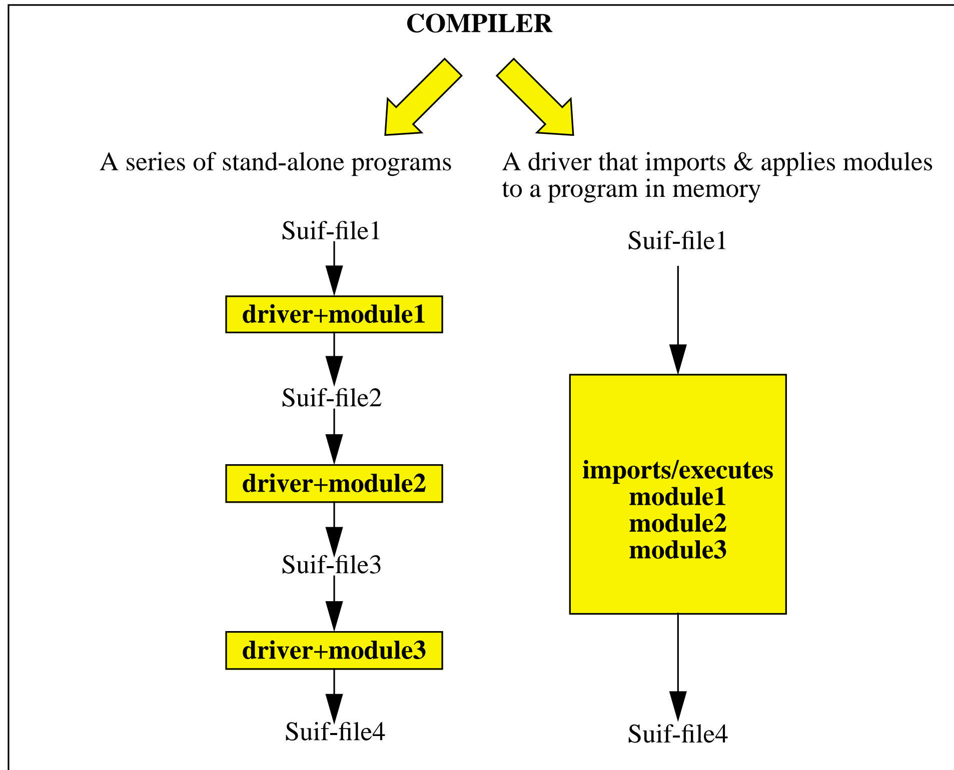

Figure 1.1 A Modular Compiler Architecture

1.2 Infrastructure Supports Modular Compiler Construction

The SUIF system characteristics make it quick and easy for a SUIF compiler writer to develop a flexible and modular compiler system and to build modules that can inter-operate with modules developed independently by other research groups. The interface of the suifdriver program that we have developed is a demonstration of the flexibility and modularity provided by the SUIF compiler framework. The suifdriver accepts a simple scripting language that allows the user to specify dynamically the program representation to be used and the sequence of built-in or user-defined compiler passes that should be applied to some SUIF program in a file. To make this more concrete, here is a sequence of commands that a user may issue to experiment with a new compiler pass:

```
> suifdriver -i
suif> import basicnodes suifnodes
suif> import mylibrary
suif> load test.1.suif
suif> mypass
suif> print -o test.out
suif> save test.2.suif
>
```

In the above sequence, the user *imports* the intermediate representation (IR) formats defined in the basicnodes and suifnodes modules, which are used by his compiler pass (mypass) in a dynamically linked library called mylibrary. The library contains a registration function that the suifdriver invokes as it imports the library. Once registered, the suifdriver will accept the module names as additional commands in the rest of the compilation session. In the above session, the user then loads in a SUIF program generated from some other compiler passes. The user's pass will work on any SUIF program includes information encapsulated by the basicnodes and suifnodes IR. That is, the SUIF program may have been generated by using a superset of IR nodes, the user's pass will still work by simply importing the pass and loading the extended SUIF program into the same environment without any recompilation. The user then

An Overview of the SUIF2 Compiler Infrastructure

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

Computer Systems Laboratory
Stanford University

*Portland Group, Inc.

1 Introduction

The SUIF system is a compiler infrastructure designed to support collaborative research and development of compilation techniques, based upon a program representation, also called SUIF ("Stanford University Intermediate Format"). The emphasis is to maximize code reuse by providing useful abstractions and frameworks for developing new compiler passes and by providing an environment that allows compiler passes to easily inter-operate. The SUIF 2 system is a new design and implementation, and is completely different from the SUIF1 system. Conversion tools exist between the SUIF1 and SUIF2 representations so that SUIF1 compiler passes can still be used.

The SUIF2 redesign is motivated by the expanded scope of the new infrastructure. The SUIF1 system was originally designed to support high-level program analysis of C and Fortran programs. Not only do we wish to support the planned components such as object-oriented programming languages and better machine-level optimizations, we also wish to support new research topics that have yet to be defined as much as possible. Furthermore, the SUIF1 system is architected as a series of standalone program passes. While the user can easily control the compilation process at the shell level, it is also inefficient to have the program read and write to disk in every pass. We wish to develop a modular system that enables components to interoperate in a flexible manner.

1.1 Key Features of the Infrastructure

To meet the goals of a research compiler infrastructure, we have created the SUIF2 design with the following key features:

- a modular subsystem that allows different components (program representations and program analyses) to be combined easily. One or more modules developed by a programmer under this model can be combined with a driver to produce a standalone program. A compiler can be a series of standalone programs that read and write a SUIF file, or it is a program that dynamically imports and applies a series of different modules to the program in memory, as shown in Figure 1.1.
- an extensible program representation that allows users to create new instructions to capture new program construct semantics or new program analysis concepts. We have predefined an object hierarchy to capture the program semantics, and users are able to further refine these abstractions for their needs. Thus a SUIF program will always contain the same basic information but may contain different subsets of nodes representing with refined program semantics. The system is designed to maximize code reuse. For example, all the basic functionalities apply to the user-defined program representations without even the need to recompile the infrastructure. The user is insulated from the details of the implementation as new representations can be defined succinctly with a high-level specification language, called "hoof". Hoof specifications are translated by a grammar-based tool called Smgn (Suif Macro Generator) into C++ interfaces and implementations.