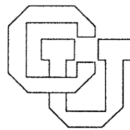
Cecil/Cesar User's Guide

Kurt M. Olender

CU-CS-402-88 June 1988



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Cesar Architecture . . . 2

2 Languages used with Cesar **7**
- 2.1 Cecil: a sequencing constraint language . . . 7
  - 2.1.1 Syntax and semantics . . . 7
  - 2.1.2 A file example . . . 9
  - 2.1.3 DAVE revisited . . . 10
- 2.2 Tepee: a tree pattern language . . . 11
  - 2.2.1 Syntax and semantics . . . 11
  - 2.2.2 Procedure or function call events . . . 12
  - 2.2.3 Entry, exit, and predicate events . . . 13
  - 2.2.4 Tepee patterns for output file example . . . 13
  - 2.2.5 Limitations . . . 14
- 2.3 Excess: an external routine specification language . . . 14
  - 2.3.1 Syntax and semantics . . . 14
  - 2.3.2 Examples . . . 16

3 An Example **17**

4 Cesar tools **24**
- 4.1 Introduction . . . 24
- 4.2 Atomic source file types . . . 25
- 4.3 Evaluation tools . . . 25
  - 4.3.1 Viewers . . . 25
  - 4.3.2 Generators . . . 26
- 4.4 Analysis tools . . . 26
  - 4.4.1 Viewers . . . 26
  - 4.4.2 Generators . . . 27
- 4.5 Resolution tools . . . 27
  - 4.5.1 Viewers . . . 27
  - 4.5.2 Generators . . . 28

4.5.3 Errors 29
4.6 Tepee tools 29
4.6.1 Generators 29
4.6.2 Errors 29
4.7 Excess tools 29
4.7.1 Generators 29
4.7.2 Errors 29
4.8 Cecil tools 30
4.8.1 Viewers 30
4.8.2 Generators 30
4.8.3 Errors 30
4.9 FORTRAN front-end and graphing tools 30
4.9.1 Viewers 30
4.9.2 Generators 31
4.9.3 Errors 32
4.9.4 Predefined Tepee routine names for FORTRAN 33

A Annotated Odin output 36

B Cesar Derivation Graph 45

List of Figures

1.1 Top level Cesar architecture 3
1.2 Graphing subsystem 4
1.3 Analysis and evaluation subsystem 5

2.1 Cecil grammar 8
2.2 A common compute/write loop structure 10
2.3 Cecil constraint for file events 10
2.4 Tepee grammar 11
2.5 Tepee patterns for output file operations 13
2.6 Excess grammar 15

3.1 Finite state machine for AQRE #3 21

4.1 Odin derivation naming conventions 25
4.2 Predefined Tepee patterns for FORTRAN 34

A.1 Timing data for file example 44

Chapter 1

Introduction

1.1 Background

Cecil is a language for the specification of sequencing constraints in software that is oriented toward the static evaluation of those constraints. Cesar is a static evaluation system that takes a program, a Cecil constraint, some other information and determines if (and more importantly, where) a violation of that Cecil constraint occurs.

The current implementation of Cecil/Cesar has been done in Ada on the Sun-3 workstation using the Verdix VADS Ada compilation system, version 5.41. The user accesses Cesar through Odin[1], which is used as a user interface and an object manager. This has the advantage for the user that when changes are made to a source file, whether it be in the program, Cecil specification, or other information needed, Odin will automatically re-compute only that information affected by the change, on the granularity of a file. The Cesar system has been designed to generate data files in such a way as to minimize the necessary recomputation of information without overloading Odin with the responsibility of maintaining an excessively large number of files.

At the present time, Cesar supports sequencing evaluation of FORTRAN programs. Additional tools to support analysis and evaluation of C and Ada programs are under construction. However, nearly all the commands and possible derivations within Cesar are language independent, so that few, if any, new commands need be learned to utilize the system with any newly supported programming language.

This manual will describe only how to invoke the Cesar tools from within Odin and how to interpret the results obtained. It will not attempt to be a comprehensive guide to the Odin system itself. The next section of the introduction will briefly outline the architecture of the Cecil/Cesar system so the reader will have some passing familiarity with the terms and overall structure of the system when reading the subsequent sections. Chapter 2 gives the details of the three specification languages used within the Cesar system. Chapter 3 gives an example of the use of Cesar for sequencing analysis and introduces the most useful tools for the typical use of Cesar. Chapter 4 describes all tools and their invocation for all subsystems.

1.2 Cesar Architecture

The current implementation of Cesar is divided into subsystems as shown in Figure 1.1. The graphing subsystem and analysis and evaluation subsystem are shown in Figures 1.2 and 1.3.

Each programming language supported will have a set of tools that produce the information required by Cesar. We partition these tools into parsing and other front end tools that may be used for other purposes than Cesar, and those specifically built for Cesar.

This latter set, the *graphing* subsystem, consists of tools to produce a labeled flowgraph and its associated data objects from a parse tree or some other intermediate form of the language for each routine, plus a special routine, a *super-main* program. This super-main program is assumed to perform all the activities of the run-time system when a program in that language begins and ends execution. A FORTRAN super-main program will, among other things, allocate space for variables in `COMMON`, define those variables declared to be initialized by `DATA` statements in `BLOCK DATA` subprograms, open and close the standard i/o units, and call the user's main program. This super-main program is always called `.main`, since this name is unlikely to conflict with a valid routine name in most programming languages.

The front end tools available for FORTRAN include a lexical scanner, parser, semantic analyzer, and viewers for the data objects these tools produce.

The Cecil subsystem consists of front end translator tools for the Cecil sequencing constraint language. These tools produce a parse tree and the semantic information necessary to drive the sequencing analysis and evaluation performed by Cesar. Detailed syntax and semantics of Cecil are described in Section 2.1. There is also a viewing tool for the semantic information generated by the Cecil semantic analyzer, but not the Cecil parse tree.

The Tepee subsystem consists of a front end translator for the Tepee tree pattern specification language. Since a Cecil specification is language independent, the user must specify which programming language constructs constitute the events in the Cecil constraint. This is done with Tepee. The detailed syntax and semantics of Tepee are described in Section 2.2.

The Excess subsystem consists of a front end translator for the Excess external specification language and the necessary tools to produce labeled flowgraph information from the Excess parse tree. Excess specifications are used to assert effects of routines for which source code is not available. Examples might be language-defined functions, library routines available only in object code form, and routines not yet written, or that exist only as stubs. An Excess source file contains a list of routine specifications that define the sequence of events that occur for each parameter and global object affected by that particular routine with a regular-expression-like syntax. The detailed syntax and semantics of Excess are described in Section 2.3.

The analysis and evaluation subsystem consists of the resolution, analysis, and evaluation phase of Cesar. All these phases are language independent. The resolution phase takes the local information produced by the language-dependent graphing tools and produces the information actually needed by the Cesar system. One task performed by the resolution phase is to build a call graph from call data produced by the graphing system. The call graph is used to control the computation of the resolved information. Generally, information for a routine cannot be computed until the information for the routines that it calls is available. Odin handles the correct ordering of computations based on the call graph.

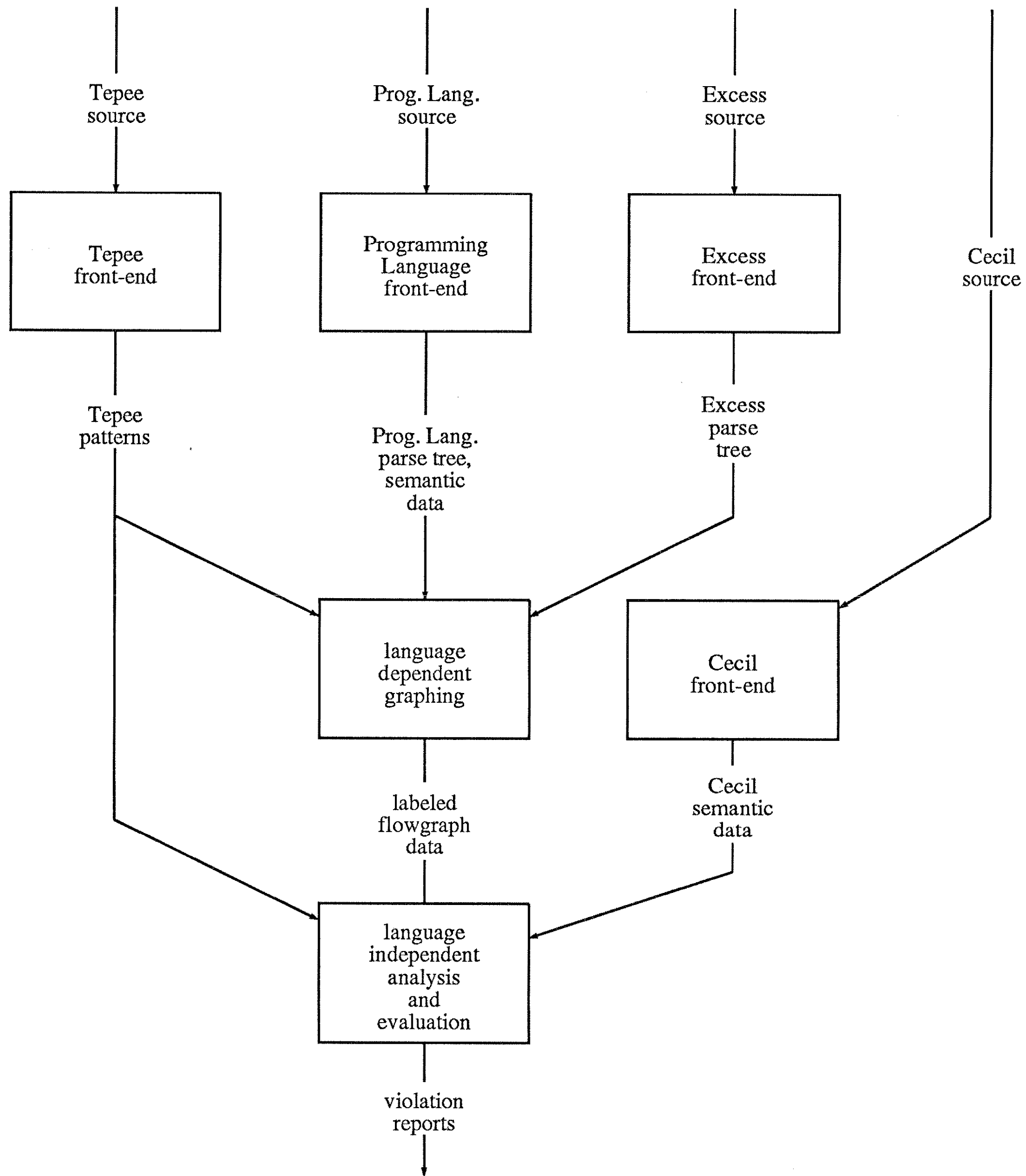


Figure 1.1: Top level Cesar architecture

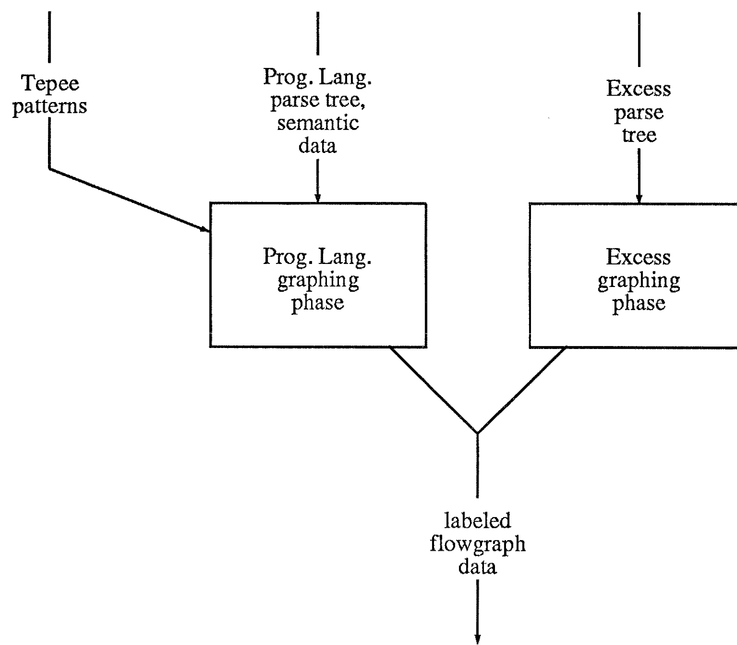


Figure 1.2: Graphing subsystem

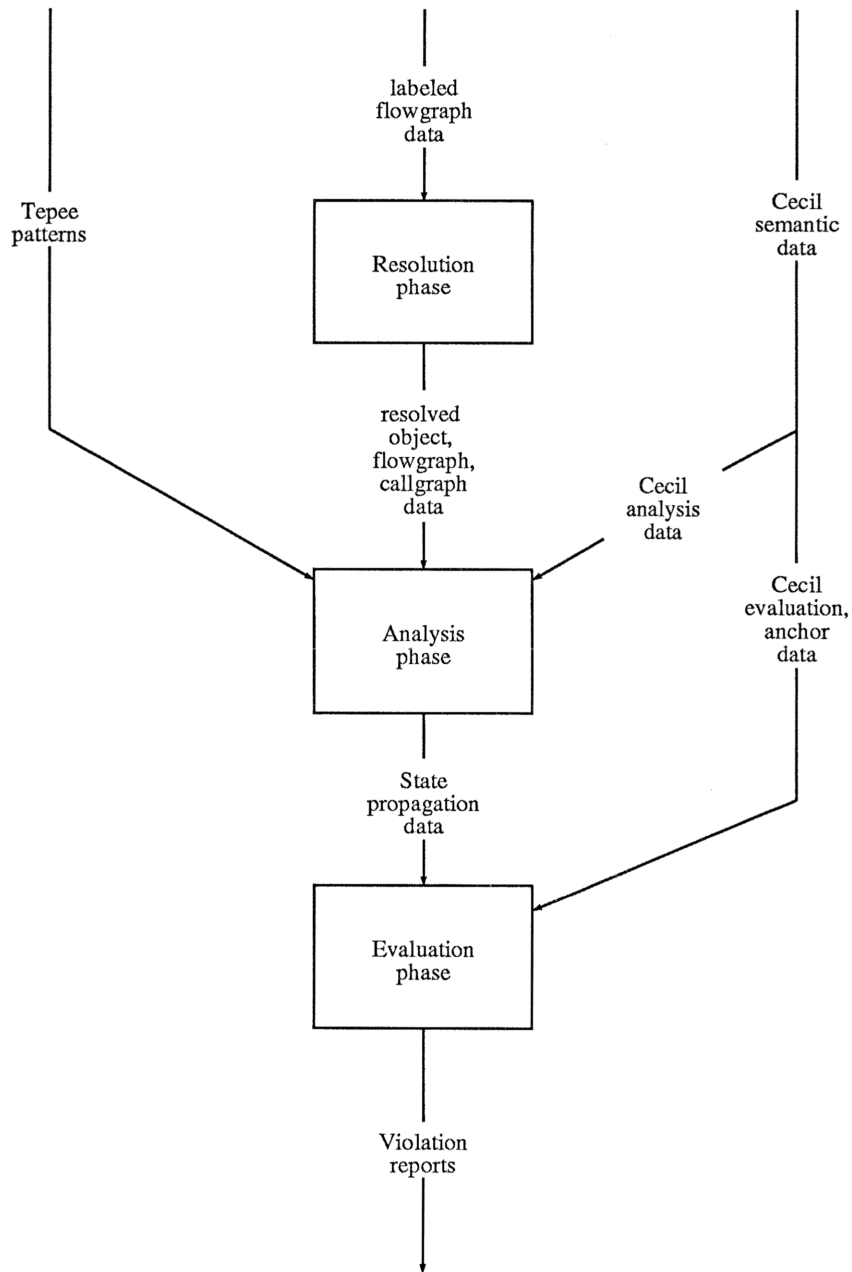


Figure 1.3: Analysis and evaluation subsystem

Another part of this phase is the resolution of local data into program-wide global data. For example, it is possible in many languages for a routine to affect data that is not directly visible to it. In C, this can happen when that routine calls a routine in another file that affects a static variable in that file. Ada has a similar feature with data objects declared inside a package body. During the local processing that occurs in the language-dependent part, it is not known what non-parameter variables may be affected by a call to another routine, since information about that other routine may not be directly available. Information about which variables a routine may transitively affect must be computed.

Once resolution has taken place, Cesar computes the state information for a finite state machine at each node of the resolved flowgraph during the analysis phase followed by an evaluation of whether or not the program satisfies the Cecil constraint and a determination of the locations of any violations. The method by which this is done is explained in detail in [2].

Chapter 4 will give a more detailed description of the information computed during all phases.

Chapter 2

Languages used with Cesar

2.1 Cecil: a sequencing constraint language

2.1.1 Syntax and semantics

The language used to specify the actual sequencing constraints is called Cecil. A BNF grammar of Cecil is given in Figure 2.1.

Cecil is a relatively terse expressional notation for sequencing constraints. The basic term is the AQRE, or Anchored Quantified Regular Expression. *Events* are operations that are applied to *objects*. The *alphabet* is the set of events over which the *regular expression*, or *regexp*, is defined. This regular expression defines the sequence of events that must occur. In the grammar, the *alphabet* is associated with a *spec* to simplify expressions where several AQRE terms have the same alphabet by not requiring the alphabet to be repeated each time.

The basic semantics of an AQRE are that, for each object, the sequence of events on that object along the number of paths specified by the *quantifier* from each event in the first *anchor* (the start anchor) to the second anchor (the end anchor), non-inclusive, is a sequence in the language denoted by the regular expression. We should also note that the anchors are optional. The semantics of an AQRE with one or more anchors omitted is discussed in the example below.

The logical operators, `and`, `or`, `not`, have the expected meaning. The conjunction of two expressions is satisfied if both expressions are satisfied, the disjunction when at least one is satisfied, and the negation of an expression when the expression is not satisfied.

The events are identifiers that are any sequence of the upper- or lower-case letters, digits, underscore, period, or hyphen characters. The question mark is used to represent any single character in the alphabet, and the hash mark represents the null event. The regular expression operators are semi-colon for concatenation, vertical bar for union, asterisk for iteration zero or more times, a plus sign for iteration one or more times, and tilde for complement.

Note that the events in an anchor are not required to be in the alphabet. In general, any anchor events not in the alphabet are considered to be null events by the regular expression and are ignored during the analysis phase. All anchors, whether or not they are in the alphabet, are used during the evaluation phase to locate those nodes in the flowgraph where evaluation is to take place, in effect delineating the paths that must be examined.

```
spec         ::=  alphabet expr
               |  '(' spec ')'
               |  spec 'and' spec
               |  spec 'or' spec
               |  'not' spec
alphabet     ::=  '{' evtlist '}'
expr         ::=  aqre
               |  '(' expr ')'
               |  expr 'and' expr
               |  expr 'or' expr
               |  'not' expr
aqre         ::=  anchor quantifier regexp anchor
anchor       ::=  '[' evtlist ']'
               |
evtlist      ::=  evtlist ',' event
               |  event
event        ::=  [a-zA-Z._-]+
quantifier   ::= 'forall' | 'exists'
regexp       ::=  '(' regexp ')'
               |  regexp '|' rexexp
               |  regexp ';' rexexp
               |  regexp '*'
               |  regexp '+'
               |  ~ regexp
               |  event
               |  '?'
               |  '#'
```

Figure 2.1: Cecil grammar

2.1.2 A file example

Some examples are in order. Let `s` be the event that occurs when a program begins execution and `t` be the termination of the program. Every program will have exactly one `s` event and exactly one `t` event for each object. The events `open`, `close`, and `write` will correspond to the execution of FORTRAN `OPEN`, `CLOSE`, and `WRITE` statements. The object is a FORTRAN i/o unit. Defining the correspondence between Cecil event names and actual program constructs is discussed in Section 2.2. We might write a Cecil specification for operations on a file to be used solely for output as the AQRE:

```
{open, write, close} [s] forall (open; write*; close)* [t]
```

This AQRE states that on every path from the entry to the exit of the program, the sequence of events `open`, `close`, and `write` must occur in the order given by the regular expression. This Cecil expression is satisfied by a program that never tries to close or write to an unopened file, or open an already opened file.

The Cecil language gives the user a great deal of flexibility in the analysis performed by allowing the user to specify exactly what is of interest at the moment. Suppose now that, along with ensuring that we never try to write to an unopened file, we are also interested in finding sequences where we open a file, and then close it with no possibility of ever writing to it. In effect, nothing happens. This is not an error; it will not cause the program to crash. In fact, it may be used to create an empty file. Even so, it may be something we are interested in looking at more closely. There may be some code intended to write to the file that was inadvertantly omitted from the program. Cesar cannot make a distinction between an intentional and unintentional use of the sequence `open; close` in a program, but it can allow us to tailor our sequencing specification to locate instances of that sequence or not as we wish and make distinctions between instances that occur on every execution path and those on only some execution paths.

With this "ineffectual" construct in mind, we might change the expression above to be:

```
{ open, write, close } [s] forall (open; write+; close)* [t]
```

In this expression, we force the existence of at least one write. However, a very frequent programming construct is to embed a write statement into a loop, where some processing is done and the results written out, as in the FORTRAN `DO` loop of Figure 2.2. This program may not execute the loop at all, depending on the values of `J` and `K`, but there is at least the possibility that the `WRITE` statement will be executed for some input data. If we prefer to reduce the number of reports that Cesar generates, (and that we must subsequently examine) we might instead add a second AQRE to the first Cecil expression with the `and` operator to state that both AQRE's must be satisfied. This second AQRE could be:

```
{open, write, close} [open] exists write
```

The end anchor in this AQRE is omitted intentionally. It is often the case that we want to make some specification about the first or last segment of a sequence of events, and don't really care about the rest of the sequence. When we omit the end anchor, we implicitly mean that the end anchor must be the termination event and append `?*` to the regular expression. This AQRE is shorter to write but otherwise equivalent to

```
          DO 10, I=J,K
             CALL DOWORK(DATA(I), RESULT)
             WRITE (1,101) RESULT
   10     CONTINUE
```

Figure 2.2: A common compute/write loop structure

```
{open, write, close} (
   [s] forall (open; write*; close)* [t] and
   [open] exists write [t] and
   exists write [close] )
```

Figure 2.3: Cecil constraint for file events

```
{open, write, close} [open] exists (write; ?*) [t]
```

This second AQRE handles a situation where an `open` event might possibly never lead to a `write` by requiring at least one path from every `open` event to have a `write` as the first (non-null) event on it. A similar situation can occur with a `close` event that never has a `write` event executed before it. So our final expression for output file operations could be that given in Figure 2.3

This Cecil expression will ensure that file operations are performed in the correct order, and that every opening and closing of a file has at least the potential of valid work being performed.

2.1.3 DAVE revisited

Cesar grew out of work by Fosdick and Osterweil[3] with the DAVE static analysis system that found data flow anomalies such as undefined references. The data flow anomalies discovered by DAVE and those added by subsequent static data flow analysis systems[4,5, among others] can be modeled as sequences of the events reference (`r`), definition (`d`), and undefinition (`u`) of variables. A reference occurs when the value of a variable is used, a definition when the variable is assigned to, and an undefinition corresponds to the variable obtaining the value *undefined.* This happens at initial allocation and final deallocation of a variable, when it has no set value, and in some other places depending on the programming language. Pascal "undefines" a `for` loop index variable at the termination of the loop. DAVE and these other systems fixed the alphabet and sequences that formed the sequencing constraint and consisted of analyzers that could look only for those fixed sequences.

We can write Cecil specifications for these data flow anomalies as long as Tepee will permit us to define a correspondence between the Cecil event name and the program constructs that form these three operations.

```
patlist      ::=  patlist ';' pattern
               |  pattern
pattern      ::=  routinename paramlist ':' cecilevent
             ::=  routinename paramlist predvalue ':' cecilevent
               |  '@entry'
               |  '@exit'
routinename  ::=  identifier
predvalue    ::=  '=true' | '=false'
paramlist    ::=  paramlist ',' param
               |  param
param        ::=  identifier
               |  '!'
               |  '?'
cecilevent   ::=  identifier
```

Figure 2.4: Tepee grammar

As an example, a Cecil expression for an undefined reference is

```
{r, d, u} forall (r | d) [r]
```

This expression states that the last event on every path into a reference event must be either a reference or a definition. An equivalant specification is

```
{r, d, u} (not (exists u [r]))
```

which states that it must never be the case that the last event on even one path is an undefinition.

This Cecil expression will find any reference that might *possibly* be undefined. If instead, we were concerned with finding only those references that are *guaranteed* to be undefined, we can change the `exists` quantifier to `forall`, obtaining:

```
{r, d, u} (not (forall u [r]))
```

2.2 Tepee: a tree pattern language

2.2.1 Syntax and semantics

An important task in performing sequencing evaluation with Cesar is defining how to statically recognize in the program text when an event occurs. The Tepee tree pattern language fulfills this requirement. Currently, Tepee permits the definition of only very simple patterns, but even these simple patterns permit us to evaluate very powerful sequencing constraints.

The grammar for Tepee is given in Figure 2.4. A Tepee specification is a list of patterns. The canonical Tepee pattern defines a procedure or function call. The routine name is the name of

the procedure as it would appear in the program. For languages like Ada, this might be required to be a fully qualified name. In fact, Ada's overloading presents particular problems that Tepee does not fully address in its current form, which will be discussed in Section 2.2.5. The routine name is followed by patterns for the parameters of that procedure. The question mark (an ANY parameter) means that any parameter will match, in other words, we don't care what is in that parameter position. The exclamation point (a BIND parameter) specifies the parameter position that contains the parse subtree for the object actually being operated on by that event. An identifier (a CONSTANT parameter) specifies that the parameter position must be filled by exactly that identifier. This pattern is mapped to the Cecil event name following the colon.

The order in which patterns are listed is significant. If more than one pattern in the list matches a given construct, the event associated with the first one in the list to match is generated. The other possible matching events are ignored.

2.2.2 Procedure or function call events

A Tepee pattern matches a procedure or function call if the routine names are identical, have the same number of parameters, and each parameter in the call matches the corresponding parameter (by position) in the Tepee pattern.

We might use the Tepee pattern

```
pop(!, ?)  :  pop
```

to map the call of the C procedure with the following definition to a Cecil event in a Cecil sequencing constraint on stack operations if we are concerned with which stack is being popped and not what value is removed from the stack.

```
void pop ( s, e ) STACK s; int e;
```

Alternatively, we could use

```
pop (?, !)  :  pop
```

if we were interested in the variable that was receiving the value from the stack. No Tepee parameter list may have more than one BIND parameter.

In the event the parameter list has no BIND parameter at all, the object that the event operates on is considered to be the state of the program as a whole, which includes all possible objects. The pattern

```
pop(?, X) : pop
```

has no BIND parameter, so the object that is considered to be popped is the state of the program as a whole. The "state" is popped whenever procedure `pop` is called with variable `X` in its second parameter position.

```
@entry      :  s;
@exit       :  t;
open(!)     :  open;
write(!)    :  write;
close(!)    :  close
```

Figure 2.5: Tepee patterns for output file operations

2.2.3 Entry, exit, and predicate events

Tepee also offers some additional types of event definitions. Two of these are `@entry` and `@exit`. These patterns correspond to the entry and termination events of the program. They never have any parameters; they apply to every possible object if they are set in the Tepee specification.

Additionally, Tepee permits us to define events that correspond to the evaluation of predicates within control-flow statements. While in general we cannot determine the value of an expression statically, predicates are a different matter because their value is directly reflected in the control flow graph. A conditional statement has two possible control flow exits and we can determine which of those corresponds to the value of the predicate being false and which true. Thus we can encode, in a very limited fashion, some dynamically determined information into the events that we recognize. We do this by adding the qualifiers `=true` or `=false` to the pattern. A predicate matches if it is an event in the Boolean or logical expression of a control flow operation along the corresponding edge of the control flow graph, and satisfies the other conditions of a match.

Lastly, although a Tepee pattern canonically represents a procedure call, the particular graphing tool may define certain routine names to reflect other program constructs. In a FORTRAN system, the routine name `open` may be interpreted as the execution of an `OPEN` statement, and the routine name `ref` as the reference of a variable. These predefined values may change with every programming language supported. The exact values used should be identifiers that could never occur in a real program in that language. For that reason, Tepee, Cecil, and Excess permit identifiers that include hyphens, periods, and underscores in any position, and are case-sensitive. That should make it relatively easy to find some form for identifiers that is valid in Tepee, Cecil, and Excess but not in the programming language. FORTRAN identifiers are single case (usually uppercase), so lowercase might be used for predefined pattern names. C does not permit initial periods or hyphens, Ada does not permit initial underscores, and so on.

2.2.4 Tepee patterns for output file example

Tepee patterns to map the events in the Cecil output file expression of Section 2.1 to FORTRAN program constructs are given in Figure 2.5, assuming that `open` , `write` , and `close` are predefined patterns with exactly one parameter (the i/o unit being operated on).

2.2.5 Limitations

Tepee is currently a very limited type of tree pattern with one major problem; we cannot define parameters that are trees themselves.

One situation where we need that capability is when events correspond to functions and the object operated on is the value returned by that function. In C, the stdio library routine `fopen` is a function that returns a file descriptor value. That value is exactly what we want to bind as the object being operated on. We might want to define a Tepee pattern for the C `fopen` as the assignment of the value returned by `fopen` to a variable. This might look like:

```
.asgn ( !, fopen(?,"w") )
```

The first parameter of the `.asgn` function (which would be predefined by the C graphing tool to correpond to the C assignment statement) is the variable assigned to, and the second is the value assigned. This Tepee pattern would match the assignment of an `fopen` value to a variable. The initial period of the name `.asgn` assures that we will never confuse the predefined name with a functon named `asgn`.

Another situation where this capability is needed occurs when we must specify additional constraints on the parameter value than its physical form, such as its type. We could do that by defining Tepee tree patterns that include type information. This is especially important in Ada as without the type information, we can not tell the difference between overloaded routine names defined in the same scope.

As this research progresses, the functionality of Tepee will be increased to permit more powerful tree patterns.

We can also see that while Tepee as a language is independent of programming language, a given Tepee specification may be interpreted in different ways by graphing tools for different programming languages. For this reason, we cannot currently mix routines from different programming languages in the same program. One current direction of research is to overcome this limitation so that multiple language programs can be analyzed and evaluated. In one sense, Cesar already does this with Excess, as we shall see in the next section, so the matter is simply one of finding an effective implementation technique.

2.3 Excess: an external routine specification language

2.3.1 Syntax and semantics

A common occurrence in the analysis of real programs is the call of a routine for which no source code exists. Functions predefined in the language, or existing only in object libraries are two examples. The usual technique to handle these so-called *external* routines is to make some conservative assumption about their behavior. The Cesar system permits such a flexible variation of analyses, however, that there is no single assumption we can make about the behavior of any external routine. One assumption that might make sense is to let all external routines have no effect, but this is not very conservative. These external routines may have effects that cause sequencing violations. We could also assume that external routines always cause violations of a Cecil constraint. This

```
speclist     ::= speclist ’;’ spec
               |   spec
spec         ::= routinename ‘(’ paramlist ‘)’ ‘[’ globlist ‘]’
               |   routinename ‘(’ paramlist ‘)’
               |   routinename ‘[’ globlist ‘]’
routinename  ::= identifier
paramlist    ::= paramlist ‘,’ regexp
               |   regexp
globlist     ::= globlist ‘,’ global
               |   global
global       ::= identifier ‘:’ regexp
```

Figure 2.6: Excess grammar

would cause the detection of an error on every path where an external routine is called; it is too conservative to be practical.

The tack taken by Cesar is to allow the specification by the user of the effects of external routines via the Excess language. Routines stubbed as an Excess specification are treated exactly as routines for which the usual source code exists. After the labeled flowgraph is produced by the Excess front end and graphing tools, there is no distinction made between routines that originated as Excess specifications and those that originated in the source code. The grammar for Excess is given in Figure 2.6.

The regular expressions are identical to Cecil regular expressions, with the exceptions that the tilde complement operator and the question mark "any event" designator are not allowed. Parameter effects are enclosed in parentheses and consists of a a list of regular expressions. Effects are related by parameter position; the third parameter of the call of a routine will be associated with the effect given by the third regular expression in the parameter list.

Since global objects have no position that uniquely defines them, we must explicitly list the name of the global object with its effect. The name must match exactly the object name that would be given to that object by the programming language graphing tool.

If a routine name exists in an Excess specification and the number of parameters does not match the actual call, the extra parameters in the actual call are considered to have no effect. The quickest way to stub a routine that is to have no effect on any value is to give it one parameter with the "null event" regular expression as its effect.

Currently, every routine mentioned or called in the source code for which no source code exists must have an Excess specification. Warnings are generated by Cesar that list any missing routines, so the user can produce the appropriate Excess stubs.

2.3.2 Examples

Assume that in a large FORTRAN program, a library routine called `ZYOUTT(X, Y, Z)` writes its three parameters to i/o unit 2 in a single write operation. We want to determine whether this program satisfies the Cecil constraint of Figure 2.3. The three parameters of `ZYOUTT` do not affect the writing to i/o unit 2, so we need only specify an effect for the i/o unit, which is considered to be a global variable by the FORTRAN graphing system, and is called `io_unit.2` by that system. We would write the Excess specification for that routine as:

```
ZYOUTT [io_unit.2 :  write]
```

Since the parameters all have no effect, and there is at least one global object, we specify only the global effect. The Excess routine specification must have at least one of either a parameter or a global. As mentioned previously, any parameters not explicitly listed will be considered to have no effect.

Now assume that we want to check for any undefined references in that same program. The FORTRAN `WRITE` statement only references variables mentioned in it's expression list, so the effect would instead be:

```
ZYOUTT ( r, r, r)
```

Chapter 3

An Example

In this chapter, we will go through a small example of the use of the Cesar system. The format of the example will be simulated Odin input and output interspersed with explanatory text. We will give the Odin output with `LogLevel=1` to suppress printing of log messages by Odin. Appendix A gives an example of the log messages for Odin, with commentary, at `LogLevel=4` for this example. The Odin prompt is "`->`".

The example to be used is an evaluation of a FORTRAN program for its compliance with the output file constraint of Figure 2.3. The FORTRAN program to be analyzed is:

```
-> file.f
        PROGRAM FILE
        READ *, ICODE
        OPEN ( UNIT=1, FILE='junk.out' )
        IF ( ICODE .NE. 1 ) THEN
          DO 10 I = 1,10
            CALL PRINT ( I )
10        CONTINUE
        ELSE
          CALL MYCLOS
        ENDIF
        CALL MYCLOS
        END

        SUBROUTINE MYCLOS
        CLOSE ( 1 )
        RETURN
        END
```

The `PRINT` routine is not included in the FORTRAN source code, so we must come up with an Excess specification for it.

```
-> file.xs
PRINT [ io_unit.1 : write, io_unit.stdout : write  ]
```

PRINT does not act on its parameter with respect to output files, but rather on two global objects, which happen to be i/o units. We can surmise that PRINT writes the value of I both to i/o unit 1 and to the standard output, but the value written is irrelevant here. The Tepee patterns that define the program constructs corresponding to the Cecil events are from Figure 2.5.

```
-> file.tp
@entry       : s;
@exit        : t;
open  ( ! ) : open;
close ( ! ) : close;
write ( ! ) : write
```

To perform the analysis, we must have a `.ref` file that lists the FORTRAN and Excess source files that will be used and a `.cesar` file that lists the `.ref` file with the Cecil and Tepee source files.

```
-> file.ref
file.f
file.xs

-> file.cesar
file.ref
file.cec
file.tp
```

Having defined these files, we can now ask Odin for the `:report` derivation which will give us a violation report for the entire program.

```
-> file.cesar :report

 Routine .main at call DAG node 1
 FORWARD : {write,close,open} [s] forall (open; write*; close)* [t]
    io_unit.1              { 12}

 Routine FILE at call DAG node 2
 No violations.

 Routine MYCLOS at call DAG node 3
 FORWARD : {write,close,open} exists write [close]
    io_unit.1              { 2}

 Routine PRINT at call DAG node 4
 No violations.
```

The `:report` derivation causes the generation of a violation report for every routine in the program, whether in FORTRAN or Excess, for the Cecil expression. Routines PRINT and FILE have no

violations, but MYCLOS and .main do. If a violation exists, the direction of analysis, AQRE violated, object name and set of nodes in the flowgraph where violations occur are listed. This report does not tell us the actual source code that contains the event causing the violation, but that information is available through other Cesar tools. Now that we know which routines have violations (and which nodes in the call DAG those routines correspond to), we can track down the violations by asking Cesar for the events labeling the edges in the flowgraph.

```
-> file.cesar +node=3 +routine=MYCLOS :prevent
 -------------------------
 Resolved flowgraph events
 -------------------------
 Stmt  Fwd  Bck  Class         Event
    2    2    3  primitive     close(io_unit.1)
```

We know the analysis is FORWARD, so we look under the Fwd column of the report to find 2, the location of the violation for MYCLOS. As it happens, there is only one event in MYCLOS and this is the one we want. This close event came from statement number 2. We find out which statement that is by asking for the listing of the file in which MYCLOS is located.

```
-> file.f :list
    TOOLPACK FORTRAN 77 SCANNER - RELEASE 2

    1 -       1          PROGRAM FILE
    2 -       4          READ *, ICODE
    3 -       9          OPEN ( UNIT=1, FILE='junk.out' )
    4 -      20          IF ( ICODE .NE. 1 ) THEN
    5 -      28            DO 10 I = 1,10
    6 -      36              CALL PRINT ( I )
    7 -      42 10         CONTINUE
    8 -      45          ELSE
    9 -      47            CALL MYCLOS
   10 -      50          ENDIF
   11 -      52          CALL MYCLOS
   12 -      55          END

    1 -      58          SUBROUTINE MYCLOS
    2 -      61          CLOSE ( 1 )
    3 -      66          RETURN
    4 -      68          END
```

Statement 2 in MYCLOS is a CLOSE(1) statement as we expect. We know that some call of MYCLOS was in a position that had no possibility of a WRITE operation before it. In this example, a simple examination of the context of the two calls of MYCLOS in FILE shows that the call at statement 9 is the offending one. Unfortunately, at the present time, Cesar cannot directly tell us which calls caused the violation. The information for this is present, but no tool yet exists to automatically

trace down the path on which the violation occurred. Until this tool is added to Cesar, we can trace it down ourselves with the information that Cesar does provide, but to do that we must know a little more about how Cesar operates.

As briefly mentioned earlier and explained in more detail in [2], Cesar does its analysis by propagating the states of a finite state machine through the flowgraph. Let's look at the analysis and evaluation of semantic data for the Cecil AQRE that `MYCLOS` violates. This data is entirely concerned with a finite state machine. The viewer for the Cecil expression semantic data is called `:praqre` . As it happens, the index of the AQRE term we want is 3. The terms are numbered sequentially in the order of their appearance in the expression.

```
-> file.cesar +aqre=3 :praqre
 ----------------------------------
 Cecil analysis and evaluation data
 ----------------------------------
 For AQRE: {write,close,open} exists write [close]

 Analysis data
 -------------
 FORWARD
     Transitions   :
       write               : { ( 0, 1) ( 1, 1)}
       close               : { ( 0, 0) ( 1, 0)}
       open                : { ( 0, 0) ( 1, 0)}

 Evaluation data
 ---------------
 FORWARD
     Quantifier       : EXISTS
     Accepting States : { 1}
     Initial State    :     0
     Number of States :     2
```

The analysis data consists of the state transitions for a finite state machine that accepts the regular expression in the AQRE term (for the FORWARD direction). In the BACKWARD direction, the finite state machine accepts the reverse of the regular expression. We have taken the liberty of editing out the BACKWARD portions of the `:praqre` report to avoid confusion, since we know that the analysis direction for AQRE #3 is FORWARD. This is a simple two state FSM as depicted in Figure 3.1. This machine records whether the last event found was `write` (state 1) or something else (state 0). Obviously, the accepting state is 1 since we want a `write` to be the last event found. The quantifier is `exists`, so at every `close` event, the set of states at the node immediately preceding it (remember events label edges, not nodes) should have a non-empty intersection with $\{1\}$. Look at the state propagation data for `MYCLOS`.

```
-> file.cesar +node=3 +routine=MYCLOS :prstate
```

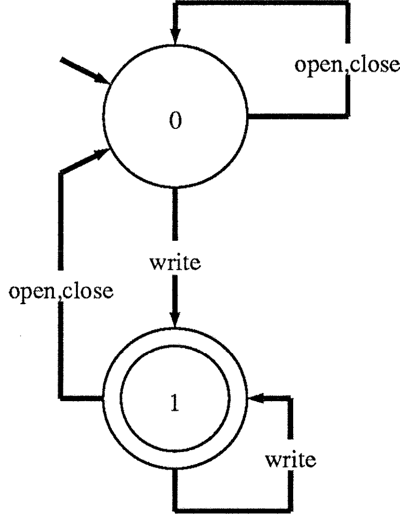


Figure 3.1: Finite state machine for AQRE #3

```
-----------------------------
State Set Propagation Summary
-----------------------------
FORWARD
    io_unit.1
      Node: 1
        { 0}
        { 0, 1}
      Node: 2
        { 0}
        { 0, 1}
      Node: 3
        { 0}
      Node: 4
        { 0}
```

Node 2 is the location of the violation. There are two possible state sets, one for each call. The first is the one causing the violation, {0} is disjoint from {1}. The problem with Cesar currently is that we don't know which call caused which state set. This is because these state sets are collected as a set themselves to avoid the combinatorial explosion of carrying around data from every call of every routine. It is possible that {0} was caused by more than one call of `MYCLOS` in a program. We know where all the calls come from, however, so we can look at the state propagation and event data for `FILE`.

```
-> file.cesar +node=2 +routine=FILE :prevent
```

```
 -------------------------
 Resolved flowgraph events
 -------------------------
 Stmt  Fwd  Bck  Class         Event
    3    3    4  primitive     open(io_unit.1)
   11    7    8  in out call   MYCLOS'1(io_unit.1)
    6    9   12  in out call   PRINT'2(io_unit.1)
    9   10    7  in out call   MYCLOS'1(io_unit.1)
    6   12    6  in out call   PRINT'1(io_unit.stdout)

-> file.cesar +node=2 +routine=FILE +aqre=3 :prstate

 -----------------------------
 State Set Propagation Summary
 -----------------------------
 FORWARD
     io_unit.1
       Node: 1
         { 0}
       Node: 2
         { 0}
       Node: 3
         { 0}
       Node: 4
         { 0}
       Node: 5
         { 0}
       Node: 6
         { 0, 1}
       Node: 7
         { 0, 1}
       Node: 8
         { 0}
       Node: 9
         { 0, 1}
       Node: 10
         { 0}
       Node: 11
         { 0}
       Node: 12
         { 1}
     io_unit.stdout
       Node: 1
```

```
  { 0}
Node: 2
  { 0}
Node: 3
  { 0}
Node: 4
  { 0}
Node: 5
  { 0}
Node: 6
  { 0, 1}
Node: 7
  { 0, 1}
Node: 8
  { 0, 1}
Node: 9
  { 0, 1}
Node: 10
  { 0}
Node: 11
  { 0, 1}
Node: 12
  { 0, 1}
```

The `:prevent` report tells us that the FORWARD nodes corresponding to the `MYCLOS` calls are 7 and 10. The call at node 10 passes $\{0\}$ and is the source of the problem. This node corresponds to the call at statement 9, or the one inside the `ELSE` clause as we would expect.

Chapter 4

Cesar tools

4.1 Introduction

Each subsystem contains tools that are used to generate information, tools to view that information, and tools to cause Odin to derive or collect the proper information for input to the generator and viewer tools. We will term this last category *Cesar internal* tools although they are not necessarily implemented using only the internal tools provided by the Odin system.

Generally, the user need only be concerned with the viewing tools, as a request for an object generated by that tool will cause Odin to automatically invoke the other tools necessary to produce it. Odin will also avoid re-computing any data object when that object is still in its cache and no object on which it depends has been changed. Only viewer tools are displayed by the Odin help system when `HelpLevel` is set to 1.

At times, however, errors may occur during the generation of data and the user may have to request the error, warning, or other derivations of these data objects to track down the error. In each of the following sections, we will first list the viewing tools, then the generation tools, and give an indication of the errors that might occur. Unless otherwise noted, additional information about an error can be obtained by applying either `:err` or `:warn` to the offending derivation. Any occurrence of the error message "`Internal error`" should be reported to the author. Additional help from Odin for all tools can obtained by setting `HelpLevel` to 2. We will ignore the Cesar internal tools in this manual altogether. Their description can be found in the Odin derivation graph source for Cesar in Appendix B.

The Cesar system derivations follow a naming convention on the suffixes of the derivation names as listed in Figure 4.1. As the derivations are described, occasional slight deviations from this convention, as well as other naming conventions, will become apparent. These sections will take the format of giving a sample Odin derivation command and describing the output produced.

nam	a Cesar internal derivation
spec	a Cesar internal derivation
M	a set of objects for a single module (file)
S	a set of objects for a single node in the call DAG
D	a set of objects upon which a single call DAG node depends
U	a set of objects for all routines in the program
N	a set of objects that are themselves sets

Figure 4.1: Odin derivation naming conventions

4.2 Atomic source file types

The Odin system assumes that there is a set of base types from which all other information can be derived in some way. The types listed here are the atomic source types used by Cesar.

`file.f`
: A FORTRAN source file.

`file.xs`
: An Excess source file.

`file.tp`
: A Tepee source file.

`file.cec`
: A Cecil source file.

`file.ref`
: A reference file containing the names of all FORTRAN and Excess source files for all routines in the program.

`file.cesar`
: A file listing the reference, Cecil, and Tepee source files to be used for a given evaluation, in that order.

4.3 Evaluation tools

4.3.1 Viewers

`file.cesar :report`
: View the node locations of violations of an entire Cecil constraint for every routine in the program. This is the derivation that one would use most often. Note that any logical

negations in the Cecil expression are pushed down to the individual AQRE terms using DeMorgan's Laws, and that this is reflected in the report.

`file.cesar +node=2 +routine=NAME :preval`
: View the node locations of violations of an entire Cecil constraint for a specific routine.

`file.cesar +node=2 +routine=NAME +aqre=2 :prvio`
: View the node locations of violations for a single routine and AQRE in a Cecil constraint. This does not reflect the possible negation of the AQRE in the original specification, and is mostly for debugging purposes.

4.3.2 Generators

`file.cesar +node=2 +routine=NAME :eval`
: Compute the node locations of violations of an entire Cecil constraint for a single routine.

`file.cesar +node=2 +routine=NAME +aqre=2 :vio`
: Compute the node locations of violations for a single AQRE in a Cecil constraint for a single routine.

4.4 Analysis tools

4.4.1 Viewers

`file.cesar +aqre=2 :analdir`
: The direction of analysis for a single AQRE. FORWARD indicates that the analysis was performed by propagating states from the entry of the flowgraph toward the exit, and BACKWARD from the exit toward the entry. The direction chosen depends on the anchor sets specified and whether `@entry` or `@exit` Tepee patterns correspond to events in the anchor sets. Current restrictions are that one of the anchor sets must be either the entry event or the termination event. This may occur implicitly when one of the anchors is omitted from the AQRE term or explicitly when the event in the anchor set is bound to either `@entry` or `@exit` in the Tepee patterns. The direction chosen will be from the anchor that corresponds to entry or exit (either explicitly or by omission) toward the other set. If both anchors correspond to entry or exit, then FORWARD is chosen.

`file.cesar +node=2 +routine=NAME +aqre=2 :prrelsum`
: Print relations for each object that describe the effect of a routine from its entry to exit. These effects are possible finite state machine transitions. "Entry" and "exit" depend on the direction of analysis.

`file.cesar +node=2 +routine=NAME +aqre=2 :prreltbl`
: Print relations for each object that describe the effect of a routine from its entry to every other node on that object. This is a superset of `:prrelsum`. Again, "entry" depends on the direction of analysis.

`file.cesar +node=2 +aqre=2 :prinit`
View the sets of finite state machine states that are passed to routines called by any routine in this call DAG node.

`file.cesar +node=2 +routine=NAME +aqre=2 :prstate`
Print the actual states of the finite state machine that might be in effect at each node of the flowgraph from the relation table data for every call of that routine.

4.4.2 Generators

`file.cesar +node=2 :relsum`
Contains the raw transition effect summary (`:rsumS`) and table data (`:rtblS`) described above for every routine in a call DAG node.

`file.cesar +node=2 :stateprop`
Contains the raw state propagation data described above for every routine in a call DAG node (`:ssumS`), as well as the initial state data for every call of a routine by a routine in the call DAG node (`:init`).

`file.cesar +node=2 +routine=NAME +aqre=2 :ancloc`
The nodes corresponding to anchor events in the flowgraph for a routine where evaluation must take place.

4.5 Resolution tools

4.5.1 Viewers

`file.cesar :srcnam`
The names of all routines in the program, including Excess routines.

`file.cesar :prcdag`
View the call DAG. This is actually a DAG of the strongly connected components of the call graph. Since in general, a programming language may permit recursion, the call graph may contain cycles. To process routines in the proper order during resolution, analysis, and evaluation, Odin requires an acyclic graph. Routines that are mutually recursive are all processed simultaneously by the remaining tools using an iterative algorithm, when that is required.

`file.cesar :prcdl`
The node label portion of `:prcdag`.

`file.cesar :prcds`
The graph structure portion of `:prcdag`.

`file.cesar :prcgrf, file.cesar :prcgl, file.cesar :prcgs`
The same information as above for the call graph instead of the DAG.

`file.cesar +node=2 :prresrou`
: The names of routines located in or called by routines in a single call DAG node, in the particular order used by the resolution data. The node number can be obtained from `:prcgs` or `:prcdag`.

`file.cesar +node=2 +routine=NAME :prresobj`
: View the object resolution data. This consists of a list of the object names, their index in the resolution data structure, a table listing the object index for parameter objects by parameter position, and a table listing the index of non-parameter objects in the resolution data structures of other routines. The node number for a given routine can be obtained from `:prcgs` or `:prcdag`.

`file.cesar +node=2 +routine=NAME :prresevt`
: View the event resolution data for a single routine. This consists of a mapping from flowgraph edges to resolved events, an updated flowgraph structure and edge-to-statement-number mapping. The resolved events use information from the object resolution data as indices into tables produced by other tools.

`file.cesar +node=2 +routine=NAME :prfgrf`
: View the flowgraph structure portion of `:prresevt`.

`file.cesar +node=2 +routine=NAME :prevent`
: View the event labeling portion of `:prresevt`.

`file.cesar +routine=NAME :prnode`
: View the node in the call DAG where a routine is located. This information is also provided for all routines in `:prcdl` but this separate tool is also provided.

`file.cesar +node=2 +routine=NAME :prcloc`
: Print data about the location of events corresponding to calls of other routines. This is a subset of the `:prfesl` data.

4.5.2 Generators

`file.cesar +node=2 :resobj`
: The raw object resolution data for every routine in a single call DAG node.

`file.cesar +node=2 +routine=NAME :resevt`
: The raw event resolution data for a single routine. This is one case where we are not required to process all mutually recursive routines together.

`file.cesar :cgrf`
: The raw call graph data. It contains the call graph structure (`:cgs`) and node labels (`:cgl`).

`file.cesar :cdag`
: The raw call DAG data. It contains the call DAG structure (`:cds`) and node labels (`:cdl`).

`file.cesar +node=2 :dagplan`
: Contains the names of routines in the specified node of the call DAG (`:sccplan`), the names of routines and DAG node numbers of routines called by any routine in this DAG node (`:depplan`), and call DAG nodes that contain routines calling any routine in this DAG node (`:supplan`).

`file.cesar +node=2 +routine=NAME :cloc`
: Compute the raw call location data. This is a subset of `:resevt` that gives only the locations of the call events, so we may pass initial states to called routines during state propagation.

4.5.3 Errors

Errors will be generated by `:pcgrf` if a routine is mentioned or called by another routine, but does not have a routine definition either in source code or in Excess. The derivation `:pcgrf :err` will list the names of these routines.

4.6 Tepee tools

4.6.1 Generators

`file.tp :tpfe`
: The Tepee front end. Produces a structure that maps from routine names to a list of possible parameter patterns for that name.

4.6.2 Errors

Syntax errors may occur. Locations of the detected errors will be found in the `:tpfe :err` derivation.

4.7 Excess tools

4.7.1 Generators

`file.xs :xsfe`
: The Excess front end. Produces a parse tree structure.

`file.xs :xsgrf`
: The Excess grapher. Produces a flowgraph structure, edge-to-event labeling, call data, object data, edge-to-statement number labeling, variable name resolution data, and list of routine names. See Section 4.9 for more detail on this information.

4.7.2 Errors

Syntax errors can be generated by `:xsfe`. Error locations are listed in the `:xsfe :err` derivation.

4.8 Cecil tools

4.8.1 Viewers

`file.cec :aqrekey`

Lists the key numbers of all AQRE terms in the Cecil expression. These will be integers from 1 to the number of AQRE terms in the expression and are used as parameters in most resolution, analysis, and evaluation tools.

`file.cec +aqre=2 :praqre`

View the semantic information for a single AQRE in the Cecil expression. The value of the aqre parameter is an integer from the range given by `:aqrekey`. This information consists of the finite state machine transition functions, initial, crash, and accepting states, and lists of the anchor events.

4.8.2 Generators

`file.cec :cecfe`

The Cecil front end. Generates a parse tree.

`file.cec :cecsem`

Generates the semantic data that is used to locate anchors (`:ancU`), for analysis (`:cadU`), and for evaluation (`:cedU`) for each separate AQRE in the Cecil expression.

4.8.3 Errors

Syntax errors can be generated by `:cecfe`. Error locations are given by the `:err` derivation. If the finite state machine for the regular expression is too large, `:cecsem` will generate an error. The current limit is 8 states, including a crash state.

4.9 FORTRAN front-end and graphing tools

All the FORTRAN front end tools are based on the IST/TOOLPACK tool set currently being distributed by Numerical Algorithms Group, Ltd. as described in [6].

4.9.1 Viewers

`file.f :list`

View a listing of the FORTRAN source as produced by the lexical scanner. This listing gives both token numbers and statement numbers in each FORTRAN program unit which are used later on by other FORTRAN tools and by Cesar tools.

`file.f :istpst`

View the FORTRAN symbol table for all program units in a single file.

`file.f :istppt`
: View the FORTRAN parse tree for all program units in a single file.

`file.f :istpfg`
: View the flowgraph as generated by IST/TOOLPACK routines for all program units in a single file. This flowgraph is used as a basis for the Cesar flowgraphs.

`file.f :fpls`
: View the parser listing. This listing is generally uninteresting, but is available. Parsing errors are listed in the `:fprs :err` derivation.

`file.f :fxls`
: View the semantic analysis listing. This listing is generally uninteresting, but is available. Semantic errors are listed in the `:fsem :err` derivation.

`file.cesar +routine=NAME :prvarnam`
: View the set of variables that correspond to a single object in a single routine. This may be a set due to explicit `EQUIVALENCE` statements or the implicit equivalences of `COMMON` block definitions that are not identical.

In general, object names for FORTRAN are chosen as follows:

Variables in COMMON or equivalenced to variables in COMMON:
  Common block name and offset in bytes. Blank common is called `.COMMON`
  Examples: `COMBLK.44, .COMMON.0`
Non-equivalenced local variables, formal parameters:
  Name of routine and variable.
  Examples: `SUBR.VAR, FUNC.PAR`
I/O units:
  "io_unit" and unit number, "stdin", or "stdout"
  Examples: `io_unit.1, io_unit.stdin`
Variables in local equivalences:
  Name of the routine, "eqv-", local equivalence class number.
  Examples: `SUBR.eqv-1, FUNC.eqv-2`

4.9.2 Generators

`file.f :fscn`
: The FORTRAN lexical scanning information. Contains the token stream (`:ftkn`), comment file (`:fcmt`), and listing (`:fsls`). This last is identical to `:flist` above.

`file.f :fprs`
: The FORTRAN parse information. Contains the basic parse tree (`:fbpt`), parser listing (`:fpls`), symbol table (`:fbst`), and comment index (`:fbci`).

`file.f :fsem`

The FORTRAN semantic analysis information. Contains an extended parse tree (`:fxpt`), extended symbol table (`:fxst`), an extended attribute table (`:fxat`) and a semantic listing (`:fxls`).

`file.f :fcan`

The FORTRAN canonical parse tree. Contains a canonical parse tree (`:fcpt`) and symbol table (`:fcst`). The canonical parse tree is the extended parse tree with some modifications made for easier flowgraph generation in the `:ffgrf` tool. These modifications include altering all FORTRAN arithmetic if statements with only two different target labels to logical if statements and adding a unique `CONTINUE` statement to terminate every `DO` loop.

`file.f :ffgrf`

The FORTRAN flowgraph information. Contains a set of flowgraph structures (`:ffgsM`), mappings from flowgraph edges to the parse tree node for the statement corresponding to that edge (`:ffplM`), mappings from flowgraph edges to the actual statement number for that statement (`:ffslM`) and a list of the names of all routines found in that file (`:frouM`). All but the last derivation are Odin compound objects, since a single file may contain more than one routine definition. For the purposes of Cesar, each FORTRAN `ENTRY` statement defines a new routine. The `:varnamM` object has been described above. An object group consists of data about the objects found in the routine.

`file.f +tepee=(file.tp :tpfe) :flocal`

This alters some of the data produced by `:ffgrf` to conform to what Cesar expects and does the actual pattern matching of the Tepee pattern to the FORTRAN parse tree. Again, since a single FORTRAN file may contain more than one routine, each component of `:flocal` except `:flocalM` is a set. The components are mappings from edges to events (`:ffelM`), object groups (`:fogrp`), call data (`:fcallM`), mappings from object names to variable names (`:fvarnamM`), and global information used to build the super-main routine (`:fglobalM`).

`file.ref +tepee=(file.tp :tpfe) :fmain`

The super-main routine data for FORTRAN. Composed of the data objects produced by the union of `:ffgrf` and `:flocal`, `:fmainfel`, `:fmainfgs`, `:fmainvarnam`, `:fmaincall`, `:fmainogrp`, and `:fmainfsl`.

4.9.3 Errors

Syntax and semantic errors may be detected. The locations of the errors are available in the appropriate `:err` derivations. The `:flocal` tool may detect a file with more than one main program unit and `:fmain` may detect more than one main subprogram unit across all source files. The `:fmain` tool may also detect that there is no main subprogram unit in the source files. Exactly one main subprogram unit is required in a single analysis group.

4.9.4 Predefined Tepee routine names for FORTRAN

Figure 4.2 lists the predefined routine names for FORTRAN statements in Tepee. All routines are considered to accept the indicated number of parameters. The identifiers x and y may be any of the three types of Tepee parameters, ANY, BIND, or CONSTANT, depending on what is desired to be an event. The figure shows the Tepee pattern alongside a typical FORTRAN construct that matches that pattern.

Note that the commutative expressional operators will be matched in two different configurations. For example `eq(!, 0)=true` is a pattern that will test for a variable being equal to 0 in a predicate. The FORTRAN `:flocal` tool will attempt to match this pattern with the parameters in the positions listed, and in the reverse order. This is not strictly necessary, as we can give two different Tepee patterns to reflect commutativity, but it is convenient.

asgn (x,y)	`X = Y`
ref (x)	reference to variable `X`
def (x)	definition of variable `X`
undef (x)	undefinition of variable `X`
rewind (x)	`REWIND` i/o unit x
backspace (x)	`BACKSPACE` i/o unit x
open (x)	`OPEN` i/o unit x
close (x)	`CLOSE` i/o unit x
write (x)	`WRITE, PRINT` to i/o unit x
read (x)	`READ` from i/o unit x
end_of_file (x)	`END=` i/o keyword (predicate)
error_on_io (x)	`ERR=` i/o keyword (predicate)
eqv (x,y)	`X .EQV. Y` (predicate)
neqv (x,y)	`X .NEQV. Y` (predicate)
or (x,y)	`X .OR. Y` (predicate)
and (x,y)	`X .AND. Y` (predicate)
not (x)	`.NOT. X` (predicate)
lt (x,y)	`X .LT. Y` (predicate)
le (x,y)	`X .LE. Y` (predicate)
eq (x,y)	`X .EQ. Y` (predicate)
ne (x,y)	`X .NE. Y` (predicate)
gt (x,y)	`X .GT. Y` (predicate)
ge (x,y)	`X .GE. Y` (predicate)
plus (x,y)	X + Y
minus (x,y)	X − Y
mul (x,y)	X ⋆ Y
div (x,y)	X / Y
exp (x,y)	X ⋆⋆ Y

Figure 4.2: Predefined Tepee patterns for FORTRAN

Bibliography

[1] Geoffrey M. Clemm, *The Odin System: An Extensible Object Manager for Software Environments*, Ph. D. Thesis, University of Colorado, 1986.

[2] Kurt M. Olender and Leon J. Osterweil, "Specification and static evaluation of sequencing constraints in software", *Proc. of the Workshop on Software Testing*, Banff, Alberta, July 1986, pp 2-9.

[3] Lloyd G. Fosdick and Leon J. Osterweil, "DAVE—a validation, error detection and documentation system for FORTRAN programs", Software—Practice and Experience, Vol. 6, 1976, pp. 473-486.

[4] Stefan M. Freudenberger, *On the Use of Global Optimization Algorithms for the Detection of Semantic Programming Errors*, Ph. D. Thesis, Courant Institute, New York University, 1984.

[5] Rieder and Conradi, "FORTVER—a FORTRAN verifier", *Proc. of the 8th Int'l. Conf. on Software Engineering*, Sept. 1986.

[6] Leon J. Osterweil, "TOOLPACK—An Experimental Software Development Environment Research Project", *IEEE Trans. on Software Engineering*, Vol. SE-9, Nov. 1983, pp. 673-685.

Appendix A

Annotated Odin output

This output was generated with `LogLevel=4`. All tool invocations are shown. Normally, Odin prints the full path name of all source files and derived objects. We have taken the liberty of removing all but the base file name to make the output more readable.

The first task is to build the call graph. That requires the call data from the routines and the super-main program, which in turn requires that the source files be compiled.

```
-> file.cesar :report
** Generating file.cesar :ces
** Generating file.cesar :cgrf
** Generating file.ref :prog
** Generating file.tp :tpfe
** Generating file.ref +tepee=(file.tp :tpfe) :callN
** Generating file.f :fscn
** Generating file.f :fprs
** Generating file.f :fsem
** Generating file.f :fcan
** Generating file.f :ffgrf
** Generating file.f +tepee=(file.tp :tpfe) :flocal
** Generating file.xs :xsfe
** Generating file.xs :xsgrf
** Generating file.ref +tepee=(file.tp :tpfe) :callU
** Generating file.ref +tepee=(file.tp :tpfe) :fglobalU
** Generating file.xs :xsfglobalM
** Generating file.ref +tepee=(file.tp :tpfe) :varnamN
** Generating file.ref +tepee=(file.tp :tpfe) :varnamU
** Generating file.ref +tepee=(file.tp :tpfe) :fmain
** Generating file.ref +tepee=(file.tp :tpfe) :pcgrf
** Generating file.cesar :cdag
```

Now that the call DAG has been generated, we must build the actual report by creating an Odin specification for the `:preval` derivations of each individual routine. This requires the list of the

AQRE term indices obtained by compiling the Cecil source file.

```
** Generating file.cesar :cesref
** Generating file.cesar :cesname
** Generating file.cesar :report
** Generating file.cesar :aqrekeyptr
** Generating file.cec :cecfe
** Generating file.cec :cecsem
** Generating file.cec :aqrekeyU
** Generating file.cec :ancU @1 :key
** Generating file.cec :ancU @2 :key
** Generating file.cec :ancU @3 :key
** Generating file.cec :aqrekey
```

Now we must do the resolution phase tasks for every routine. We start at the root of the call graph and traverse downward, building Odin specifications of the data required. Remember that derivations ending in `plan`, `spec`, `nam` are merely there to build the Odin specifications we need to cause the right data to be generated.

```
** Generating file.cesar +node='1' :dagplan
** Generating file.cesar +node='1' +routine='.main' :viospec
** Generating file.cesar +node='1' +routine='.main' +aqre='1' :ssumnamN
** Generating file.cesar +node='1' :grfspec
** Generating file.cesar +node='1' +routine='.main' :ornamN
** Generating file.cesar +node='1' :ogrpSN
** Generating file.cesar +node='1' :ogrpS
** Generating file.cesar +node='1' :ordep
** Generating file.cesar +node='2' :dagplan
** Generating file.cesar +node='2' :ogrpSN
** Generating file.ref +tepee=(file.tp :tpfe) :ogrpU
** Generating file.ref +tepee=(file.tp :tpfe) :ogrpU @FILE
** Generating file.cesar +node='2' :ogrpS
** Generating file.cesar +node='2' :ordep
** Generating file.cesar +node='3' :dagplan
** Generating file.cesar +node='3' :ogrpSN
** Generating file.ref +tepee=(file.tp :tpfe) :ogrpU @MYCLOS
** Generating file.cesar +node='3' :ogrpS
** Generating file.cesar +node='3' :ordep
```

We can now build the object resolution data for nodes 3 and 4, which are leaves of the call DAG. The `:rovD`, `:rptD`, `:rgtD`, `ordD` are the object resolution data on which this node is dependent, and happen to be empty for leaves of the call DAG. Given the data for the leaves we can traverse back up the call DAG and build the object resolution data for the internal nodes as well.

```
** Generating file.cesar +node='3' :rovD
```

```
** Generating file.cesar +node='3' :rptD
** Generating file.cesar +node='3' :rgtD
** Generating file.cesar +node='3' :orsD
** Generating file.cesar +node='3' :resobj
** Generating file.cesar +node='4' :dagplan
** Generating file.cesar +node='4' :ogrpSN
** Generating file.ref +tepee=(file.tp :tpfe) :ogrpU @PRINT
** Generating file.cesar +node='4' :ogrpS
** Generating file.cesar +node='4' :ordep
** Generating file.cesar +node='4' :rovD
** Generating file.cesar +node='4' :rptD
** Generating file.cesar +node='4' :rgtD
** Generating file.cesar +node='4' :orsD
** Generating file.cesar +node='4' :resobj
** Generating file.cesar +node='2' :rovD
** Generating file.cesar +node='2' :rptD
** Generating file.cesar +node='2' :rgtD
** Generating file.cesar +node='2' :orsD
** Generating file.cesar +node='2' :resobj
** Generating file.cesar +node='1' :rovD
** Generating file.cesar +node='1' :rptD
** Generating file.cesar +node='1' :rgtD
** Generating file.cesar +node='1' :orsD
** Generating file.cesar +node='1' :resobj
```

Given the object resolution data, we can build the event resolution data for each routine. The event resolution data depends only on the object resolution data and the original flowgraph for each routine, so we can do this in any order we wish. Odin actually triggers the event resolution data computation by requesting the analysis data.

```
** Generating file.cesar +node='1' +routine='.main' :ornamU
** Generating file.cesar +node='1' +routine='.main' :orspec
** Generating file.cesar +routine='.main' :fgnamN
** Generating file.cesar +routine='.main' :fgnamU
** Generating file.cesar +routine='.main' :fgspec
** Generating file.cesar +node='1' +routine='.main' :resevt
** Generating file.cesar +node='1' +routine='.main' :cloc
** Generating file.cesar +node='1' :clocS
** Generating file.cesar +node='1' :rfgsS
** Generating file.cesar +node='1' :rfelS
** Generating file.cesar +node='1' +aqre='1' :analspec
** Generating file.cesar +node='2' :grfspec
** Generating file.cesar +routine='FILE' :fgnamN
** Generating file.ref +tepee=(file.tp :tpfe) :felN
```

```
** Generating file.ref +tepee=(file.tp :tpfe) :felU
** Generating file.ref +tepee=(file.tp :tpfe) :felU @FILE
** Generating file.ref :fgsU
** Generating file.ref :fgsU @FILE
** Generating file.ref :fslU
** Generating file.ref :fslU @FILE
** Generating file.cesar +routine='FILE' :fgnamU
** Generating file.cesar +routine='FILE' :fgspec
** Generating file.cesar +node='2' +routine='FILE' :ornamN
** Generating file.cesar +node='2' +routine='FILE' :ornamU
** Generating file.cesar +node='2' +routine='FILE' :orspec
** Generating file.cesar +node='2' +routine='FILE' :resevt
** Generating file.cesar +node='2' :rfgsS
** Generating file.cesar +node='2' :rfelS
** Generating file.cesar +node='2' +routine='FILE' :cloc
** Generating file.cesar +node='2' :clocS
** Generating file.cesar +node='2' +aqre='1' :analspec
** Generating file.cesar +node='3' :grfspec
** Generating file.cesar +routine='MYCLOS' :fgnamN
** Generating file.ref +tepee=(file.tp :tpfe) :felU @MYCLOS
** Generating file.ref :fgsU @MYCLOS
** Generating file.ref :fslU @MYCLOS
** Generating file.cesar +routine='MYCLOS' :fgnamU
** Generating file.cesar +routine='MYCLOS' :fgspec
** Generating file.cesar +node='3' +routine='MYCLOS' :ornamN
** Generating file.cesar +node='3' +routine='MYCLOS' :ornamU
** Generating file.cesar +node='3' +routine='MYCLOS' :orspec
** Generating file.cesar +node='3' +routine='MYCLOS' :resevt
** Generating file.cesar +node='3' :rfgsS
** Generating file.cesar +node='3' :rfelS
** Generating file.cesar +node='3' +routine='MYCLOS' :cloc
** Generating file.cesar +node='3' :clocS
```

We have computed the event resolution data for node 3, a leaf, so we can compute the analysis data for it, and then go back up the call DAG. The analysis data for a node depends on the analysis data of its descendents in the call DAG. We have to do this for all AQRE terms in the Cecil expression. The analysis data computations are somewhat intermixed with the evaluation data because of the way Odin makes requests and traverses the call DAG. Data is computed when all its required inputs are available.

```
** Generating file.cesar +node='3' +aqre='1' :analspec
** Generating file.cesar +node='3' +aqre='1' :rsumD
** Generating file.cesar +aqre='1' :aqrenamN
** Generating file.cesar +aqre='1' :aqrenamU
```

```
** Generating file.cesar +aqre='1' :aqrespec
** Generating file.cesar :tpfeptr
** Generating file.cesar +aqre='1' :analdir
** Generating file.cesar +node='3' +aqre='1' :relsum
** Generating file.cesar +node='4' :grfspec
** Generating file.cesar +routine='PRINT' :fgnamN
** Generating file.ref +tepee=(file.tp :tpfe) :felU @PRINT
** Generating file.ref :fgsU @PRINT
** Generating file.ref :fslU @PRINT
** Generating file.cesar +routine='PRINT' :fgnamU
** Generating file.cesar +routine='PRINT' :fgspec
** Generating file.cesar +node='4' +routine='PRINT' :ornamN
** Generating file.cesar +node='4' +routine='PRINT' :ornamU
** Generating file.cesar +node='4' +routine='PRINT' :orspec
** Generating file.cesar +node='4' +routine='PRINT' :resevt
** Generating file.cesar +node='4' :rfgsS
** Generating file.cesar +node='4' :rfelS
** Generating file.cesar +node='4' +routine='PRINT' :cloc
** Generating file.cesar +node='4' :clocS
** Generating file.cesar +node='4' +aqre='1' :analspec
** Generating file.cesar +node='4' +aqre='1' :rsumD
** Generating file.cesar +node='4' +aqre='1' :relsum
** Generating file.cesar +node='2' +aqre='1' :rsumD
** Generating file.cesar +node='2' +aqre='1' :relsum
** Generating file.cesar +node='1' +aqre='1' :rsumD
** Generating file.cesar +node='1' +aqre='1' :relsum
** Generating file.cesar +node='1' +aqre='1' :initD
** Generating file.cesar +node='1' +aqre='1' :stateprop
** Generating file.cesar +node='1' +routine='.main' +aqre='1' :ssumnamU
** Generating file.cesar +node='1' +routine='.main' +aqre='1' :ssumptr
** Generating file.cesar +node='1' +routine='.main' +aqre='1' :ancloc
** Generating file.cesar +node='1' +routine='.main' +aqre='1' :vio
** Generating file.cesar +node='1' +routine='.main' +aqre='2' :ssumnamN
** Generating file.cesar +node='1' +aqre='2' :analspec
** Generating file.cesar +node='2' +aqre='2' :analspec
** Generating file.cesar +node='3' +aqre='2' :analspec
** Generating file.cesar +node='3' +aqre='2' :rsumD
** Generating file.cesar +aqre='2' :aqrenamN
** Generating file.cesar +aqre='2' :aqrenamU
** Generating file.cesar +aqre='2' :aqrespec
** Generating file.cesar +aqre='2' :analdir
** Generating file.cesar +node='3' +aqre='2' :relsum
** Generating file.cesar +node='4' +aqre='2' :analspec
```

```
** Generating file.cesar +node='4' +aqre='2' :rsumD
** Generating file.cesar +node='4' +aqre='2' :relsum
** Generating file.cesar +node='2' +aqre='2' :rsumD
** Generating file.cesar +node='2' +aqre='2' :relsum
** Generating file.cesar +node='1' +aqre='2' :rsumD
** Generating file.cesar +node='1' +aqre='2' :relsum
** Generating file.cesar +node='1' +aqre='2' :initD
** Generating file.cesar +node='1' +aqre='2' :stateprop
** Generating file.cesar +node='1' +routine='.main' +aqre='2' :ssumnamU
** Generating file.cesar +node='1' +routine='.main' +aqre='2' :ssumptr
** Generating file.cesar +node='1' +routine='.main' +aqre='2' :ancloc
** Generating file.cesar +node='1' +routine='.main' +aqre='2' :vio
** Generating file.cesar +node='1' +routine='.main' +aqre='3' :ssumnamN
** Generating file.cesar +node='1' +aqre='3' :analspec
** Generating file.cesar +node='2' +aqre='3' :analspec
** Generating file.cesar +node='3' +aqre='3' :analspec
** Generating file.cesar +node='3' +aqre='3' :rsumD
** Generating file.cesar +aqre='3' :aqrenamN
** Generating file.cesar +aqre='3' :aqrenamU
** Generating file.cesar +aqre='3' :aqrespec
** Generating file.cesar +aqre='3' :analdir
** Generating file.cesar +node='3' +aqre='3' :relsum
** Generating file.cesar +node='4' +aqre='3' :analspec
** Generating file.cesar +node='4' +aqre='3' :rsumD
** Generating file.cesar +node='4' +aqre='3' :relsum
** Generating file.cesar +node='2' +aqre='3' :rsumD
** Generating file.cesar +node='2' +aqre='3' :relsum
** Generating file.cesar +node='1' +aqre='3' :rsumD
** Generating file.cesar +node='1' +aqre='3' :relsum
** Generating file.cesar +node='1' +aqre='3' :initD
** Generating file.cesar +node='1' +aqre='3' :stateprop
** Generating file.cesar +node='1' +routine='.main' +aqre='3' :ssumnamU
** Generating file.cesar +node='1' +routine='.main' +aqre='3' :ssumptr
** Generating file.cesar +node='1' +routine='.main' +aqre='3' :ancloc
** Generating file.cesar +node='1' +routine='.main' +aqre='3' :vio
** Generating file.cesar +node='1' +routine='.main' :vioU
** Generating file.cesar :cecfe
** Generating file.cesar :cecsem
** Generating file.cesar +node='1' +routine='.main' :anclocU
** Generating file.cesar +node='1' +routine='.main' :eval
** Generating file.cesar :txtlist
** Generating file.cec :txtset
** Generating file.cesar :dirset
```

```
** Generating file.cesar :dirlist
```

The analysis data is computed for all routines, and the evaluation data computed for the root of the call DAG. We traverse back down to compute the rest of the routines for all AQRE terms.

```
** Generating file.cesar +node='1' +routine='.main' :preval
** Generating file.cesar +node='2' +routine='FILE' :viospec
** Generating file.cesar +node='2' +routine='FILE' +aqre='1' :ssumnamN
** Generating file.cesar +node='2' +aqre='1' :initD
** Generating file.cesar +node='2' +aqre='1' :stateprop
** Generating file.cesar +node='2' +routine='FILE' +aqre='1' :ssumnamU
** Generating file.cesar +node='2' +routine='FILE' +aqre='1' :ssumptr
** Generating file.cesar +node='2' +routine='FILE' +aqre='1' :ancloc
** Generating file.cesar +node='2' +routine='FILE' +aqre='1' :vio
** Generating file.cesar +node='2' +routine='FILE' +aqre='2' :ssumnamN
** Generating file.cesar +node='2' +aqre='2' :initD
** Generating file.cesar +node='2' +aqre='2' :stateprop
** Generating file.cesar +node='2' +routine='FILE' +aqre='2' :ssumnamU
** Generating file.cesar +node='2' +routine='FILE' +aqre='2' :ssumptr
** Generating file.cesar +node='2' +routine='FILE' +aqre='2' :ancloc
** Generating file.cesar +node='2' +routine='FILE' +aqre='2' :vio
** Generating file.cesar +node='2' +routine='FILE' +aqre='3' :ssumnamN
** Generating file.cesar +node='2' +aqre='3' :initD
** Generating file.cesar +node='2' +aqre='3' :stateprop
** Generating file.cesar +node='2' +routine='FILE' +aqre='3' :ssumnamU
** Generating file.cesar +node='2' +routine='FILE' +aqre='3' :ssumptr
** Generating file.cesar +node='2' +routine='FILE' +aqre='3' :ancloc
** Generating file.cesar +node='2' +routine='FILE' +aqre='3' :vio
** Generating file.cesar +node='2' +routine='FILE' :vioU
** Generating file.cesar +node='2' +routine='FILE' :anclocU
** Generating file.cesar +node='2' +routine='FILE' :eval
** Generating file.cesar +node='2' +routine='FILE' :preval
** Generating file.cesar +node='3' +routine='MYCLOS' :viospec
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='1' :ssumnamN
** Generating file.cesar +node='3' +aqre='1' :initD
** Generating file.cesar +node='3' +aqre='1' :stateprop
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='1' :ssumnamU
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='1' :ssumptr
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='1' :ancloc
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='1' :vio
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='2' :ssumnamN
** Generating file.cesar +node='3' +aqre='2' :initD
** Generating file.cesar +node='3' +aqre='2' :stateprop
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='2' :ssumnamU
```

```
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='2' :ssumptr
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='2' :ancloc
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='2' :vio
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='3' :ssumnamN
** Generating file.cesar +node='3' +aqre='3' :initD
** Generating file.cesar +node='3' +aqre='3' :stateprop
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='3' :ssumnamU
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='3' :ssumptr
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='3' :ancloc
** Generating file.cesar +node='3' +routine='MYCLOS' +aqre='3' :vio
** Generating file.cesar +node='3' +routine='MYCLOS' :vioU
** Generating file.cesar +node='3' +routine='MYCLOS' :anclocU
** Generating file.cesar +node='3' +routine='MYCLOS' :eval
** Generating file.cesar +node='3' +routine='MYCLOS' :preval
** Generating file.cesar +node='4' +routine='PRINT' :viospec
** Generating file.cesar +node='4' +routine='PRINT' +aqre='1' :ssumnamN
** Generating file.cesar +node='4' +aqre='1' :initD
** Generating file.cesar +node='4' +aqre='1' :stateprop
** Generating file.cesar +node='4' +routine='PRINT' +aqre='1' :ssumnamU
** Generating file.cesar +node='4' +routine='PRINT' +aqre='1' :ssumptr
** Generating file.cesar +node='4' +routine='PRINT' +aqre='1' :ancloc
** Generating file.cesar +node='4' +routine='PRINT' +aqre='1' :vio
** Generating file.cesar +node='4' +routine='PRINT' +aqre='2' :ssumnamN
** Generating file.cesar +node='4' +aqre='2' :initD
** Generating file.cesar +node='4' +aqre='2' :stateprop
** Generating file.cesar +node='4' +routine='PRINT' +aqre='2' :ssumnamU
** Generating file.cesar +node='4' +routine='PRINT' +aqre='2' :ssumptr
** Generating file.cesar +node='4' +routine='PRINT' +aqre='2' :ancloc
** Generating file.cesar +node='4' +routine='PRINT' +aqre='2' :vio
** Generating file.cesar +node='4' +routine='PRINT' +aqre='3' :ssumnamN
** Generating file.cesar +node='4' +aqre='3' :initD
** Generating file.cesar +node='4' +aqre='3' :stateprop
** Generating file.cesar +node='4' +routine='PRINT' +aqre='3' :ssumnamU
** Generating file.cesar +node='4' +routine='PRINT' +aqre='3' :ssumptr
** Generating file.cesar +node='4' +routine='PRINT' +aqre='3' :ancloc
** Generating file.cesar +node='4' +routine='PRINT' +aqre='3' :vio
** Generating file.cesar +node='4' +routine='PRINT' :vioU
** Generating file.cesar +node='4' +routine='PRINT' :anclocU
** Generating file.cesar +node='4' +routine='PRINT' :eval
** Generating file.cesar +node='4' +routine='PRINT' :preval
** Generating file.cesar :report :.cat
```

All derivations are complete. Odin can print the final report, which is as shown previously. The total derivation took about 2 minutes, 50 seconds elapsed time, of which about 60 seconds was

Phase	Time(sec)
resolution	5.9
analysis	25.0
evaluation	9.6
FORTRAN	3.6
Cecil	5.6
Tepee	0.3
Excess	0.8
Cesar internal	8.8
TOTAL	59.6

Figure A.1: Timing data for file example

actually spent in the various tools. Times for the various phases are listed in Figure A.

Appendix B

Cesar Derivation Graph

```
cesar
        ATOMIC
        "required program and specification files for a CESAR run"

ces <
    refptr^ref   "names of program source files"*
    cecptr^cec   "Cecil sequencing specification"*
    tpptr^tp     "Tepee tree pattern"*
    >
        "composite object for Cesar analysis"*:
            STRUCT
                : cesar

cesname
        "Cesar grp ODIN name"*:
            NAME
                : cesref

cesref(.null)
        "ref to Cesar grp"*:
            USER copy_first.cmd
                : ces

# *** The sequencing evaluation subsystem
# User interface tools
report(preval@)
        "report of violations of all routines for all aqre's":
            USER report.cmd
                : cdl
                : cesname

preval
        "violations of all aqre's for a routine":
            USER preval.cmd
```

```
                : eval
                : rovptr
                : txtlist
                : dirlist
                : rounam
                : rounod

prancloc
        "anchor locations in a routine":
            USER prancloc.cmd
                : ancloc
                : rovptr
                : analdir

prvio
        "violations of an aqre in a routine":
            USER prvio.cmd
                : vio
                : rovptr

prinit
        "initial state summary for a cdag node":
            USER prinit.cmd
                : init
                : rovD
                : rrv

prstate
        "sequencing analysis state propagation results":
            USER prprop.cmd
                : ssumptr
                : rovptr
                : analdir

ssumptr^ssum
        "state summary for a routine/aqre"*:
            USER ssumspec.cmd
                :(ssumnamU)

ssumnamU(.null)
        "union of ssum for a routine"*:
            FLATTEN
                :(ssumnamN)

ssumnamN(.null@)
        "nested union of ssums for a routine"*:
            USER ssumnam.cmd
                : dagnode
                : cesname
```

```
                : refptr
                : PARAMETERS < routine aqre >

# working tools
eval
        "violations of all aqre's for a routine"*:
            USER eval.cmd
                : (vioU)
                : cecexp
                : (anclocU)

anclocU(ancloc)
        "anchor locations of all aqre's for a routine"*:
            UNION
                : (anclocN)

vioU(vio)
        "set of violations of all aqre's for a routine"*:
            UNION
                : (vioN)

viospec <
    vioN(vio@)        "nested vio set of all aqre's for routine"*
    anclocN(ancloc@) "nested ancloc set of all aqre's for routine"*
    rounam            "name of routine"*
    rounod            "node of routine"*
    >
        "violation and anchor locations for all aqre's for a routine"*:
            USER viospec.cmd
                : aqrekeyptr
                : dagnode
                : cesname
                : PARAMETERS < routine >

vio
        "violations of an aqre in a routine"*:
            USER mkvio.cmd
                : ssumptr
                : cedptr
                : ancloc
                : analdir

stateprop<
    ssumS[ssum] "sequence state summary set"*
    init         "initial state mapping"*
    >
        "sequencing state summary data"*:
            USER stateprop.cmd
                : (clocS)
```

```
                : (rtblS)
                : (initD)
                : (orsD)
                : (orsS)
                :  rrv
                :  recurs
                :  cedptr
                :  analdir


# *** The sequencing analysis subsystem

# User interface routines
prreltbl
        "sequencing analysis relation table":
            USER prreltbl.cmd
                : rtblptr
                : rovptr
                : analdir

prrelsum
        "sequencing analysis relation summary":
            USER prrelsum.cmd
                : rsumptr
                : rovptr
                : analdir


# Working routines

rsspec <
    rsumptr^rsum "rsum for a routine"*
    rtblptr^rtbl "rtbl for a routine"*
    >
        "relation data for a routine"*:
            USER rsspec.cmd
                : (rsnamU)

rsnamU(.null)
        "union of rsum for a routine"*:
            FLATTEN
                :(rsnamN)

rsnamN(.null@)
        "nested union of rsums for a routine"*:
            USER rsnam.cmd
                : dagnode
                : cesname
                : refptr
```

```
                : PARAMETERS < routine aqre >

relsum <
    rsumS [rsum]  "rsums's of cdag node"*
    rtblS [rtbl]  "rtbls's of cdag node"*
    >
        "relation data for a cdag node"*:
            USER relsum.cmd
                : (rfgsS)
                : (rfelS)
                : (orsS)
                : (clocS)
                : (rsumD)
                : rrv
                : recurs
                : cadptr
                : analdir

rsumD(rsum)
        "rsum's reqd by a cdag node"*:
            UNION
                : (rsumDN)

initD(init)
        "init's reqd by a cdag node"*:
            UNION
                : (initDN)


analspec <
    rsumDN(rsum@)   "nested rsum's reqd by a cdag node"*
    initDN(init@)   "nested init's reqd by a cdag node"*
    >
        "analysis data for a cdag node"*:
            USER analspec.cmd
                : depplan
                : supplan
                : cesname
                : refptr
                : PARAMETERS < aqre >

# locations of anchors and analysis direction

ancloc
        "node location of anchor events"*:
            USER ancloc.cmd
                : orsptr
                : rfel
                : ancptr
```

```
                : analdir

dirlist
        "list of analysis directions for all aqre's":
            CAT
                :(dirset)

dirset(analdir@)
        "set of analysis directions for all aqre's":
            USER dirlist.cmd
                : cesname
                : aqrekeyptr

analdir
        "direction in which to run analysis":
            USER pickdir.cmd
                : ancptr
                : tpfeptr

# *** Event resolution subsystem
# User interface routines
prresevt(.null)
        "flowgraph/event resolution data":
            COLLECT
                :prevt
                :prrfgs

prfgrf
        "resolved flowgraph structure":
            USER prfg.cmd
                : rfgs

prevent
        "resolved flowgraph event and statement labels":
            USER prevent.cmd
                : rfel
                : rfsl
                : rrv
                : rovptr

prcloc
        "call location table for a routine":
            USER prclt.cmd
                : cloc
                : rovptr
                : rrv

# working routines
# collections of resolved flowgraph data for a cdag node
```

```
clocS(cloc)
        "cloc's of a cdag node"*:
            UNION
                : (clocSN)

rfelS(rfel)
        "rfel's of a cdag node"*:
            UNION
                : (rfelSN)

rfgsS(rfgs)
        "rfgs's of a cdag node"*:
            UNION
                : (rfgsSN)

grfspec <
    rfgsSN(rfgs@) "nested rfgs's of a cdag node"*
    rfelSN(rfel@) "nested ffel's of a cdag node"*
    clocSN(cloc@) "nested cloc's of a cdag node"*
    >
        "graph data for a cdag node"*:
            USER grfspec.cmd
                :dagnode
                :sccplan
                :cesname
                :refptr

cloc
        "location of calls"*:
            USER cloc.cmd
                :orsptr
                :rfel
                :rrv

resevt <
    rfgs  "Resolved fgs"*
    rfel  "Resolved fel"*
    rfsl  "Resolved fsl"*
    >
        "resolved fgrf/event data"*:
            USER resevt.cmd
                : felptr
                : fgsptr
                : fslptr
                : rovptr
                : rptptr
                : rgtptr
                : rrv
```

```
# Object resolution subsystem
# User interface routines
prresobj
        "object resolution data":
            USER prresobj.cmd
                : rovptr
                : rptptr
                : rgtptr
                : orsptr
                : rrv

# Working routines
orspec <
    rovptr^rov "rov for a routine"*
    rptptr^rpt "rpt for a routine"*
    rgtptr^rgt "rgt for a routine"*
    orsptr^ors "ors for a routine"*
    >
        "object resolution data for a routine"*:
            USER orspec.cmd
                :(ornamU)

ornamU(.null)
        "union of object resolution data for a routine"*:
            FLATTEN
                :(ornamN)

ornamN(.null@)
        "nested union of object resolution data for a routine"*:
            USER ornam.cmd
                : cesname
                : refptr
                : PARAMETERS < node routine >

resobj <
    rovS [rov]  "rov's for a cdag node"*
    rptS [rpt]  "rpt's for a cdag node"*
    rgtS [rgt]  "rgt's for a cdag node"*
    orsS [ors]  "ors's for a cdag node"*
    >
        "objres data for cdag node"*:
            USER resobj.cmd
                :(ogrpS)
                :(rovD)
                :(rptD)
                :(rgtD)
                :(orsD)
```

```
                : rrv

rovD(rov)
        "rov's reqd by a cdag node"*:
            UNION
                :(rovDN)

rptD(rpt)
        "rpt's reqd by a cdag node"*:
            UNION
                :(rptDN)

rgtD(rgt)
        "rgt's reqd by a cdag node"*:
            UNION
                :(rgtDN)

orsD(ors)
        "ors's reqd by a cdag node"*:
            UNION
                :(orsDN)


ordep <
    rovDN(rov@)   "nested rov's  reqd by a cdag node"*
    rptDN(rpt@)   "nested rpt's  reqd by a cdag node"*
    rgtDN(rgt@)   "nested rgt's  reqd by a cdag node"*
    orsDN(ors@)   "nested ors's  reqd by a cdag node"*
    >
        "object resolution data required by a cdag node"*:
            USER ordep.cmd
                :depplan
                :refptr
                :cesname

# unresolved flowgraph data for a given routine
fgspec <
    felptr^fel  "fel for the routine"*
    fgsptr^fgs  "fgs for the routine"*
    fslptr^fsl  "fsl for the routine"*
    >
        "unresolved flowgraph data for a single routine"*:
            USER fgspec.cmd
                : (fgnamU)

fgnamU(.null)
        "names of unresolved flowgraph data for a single routine"*:
            FLATTEN
                : (fgnamN)
```

```
fgnamN(.null@)
        "ODIN names of unresolved flowgraph data for single routine "*:
            USER fgnam.cmd
                : refptr
                : tpptr
                : PARAMETERS < routine >

# collect object groups for al routines in a cdag node

ogrpS(ogrp)
        "ogrp's for a cdag node"*:
            UNION
                :(ogrpSN)

ogrpSN(ogrp@)
        "nested ogrps for a cdag node"*:
            USER ogrpspec.cmd
                :dagnode
                :sccplan
                :refptr
                :tpptr

# *** Call graph subsystem

# User interface routines
prnode
        "node in call dag where routine is located":
            USER cgrflook.cmd
                :cdn
                :PARAMETERS < routine >

prcdag(.null)
        "dag of strongly connected components in call graph":
            COLLECT
                :prcdl
                :prcds

prcdl
        "call dag labels":
            USER prcdl.cmd
                :cdl

prcds
        "call dag structure":
            USER prfg.cmd
                :cds

prcgrf(.null)
```

```
        "call graph data":
            COLLECT
                :prcgl
                :prcgs

prcgl
        "call graph labels":
            USER prcgl.cmd
                :cgl

prcgs
        "call graph structure":
            USER prfg.cmd
                :cgs

prresrou(.null)
        "names of routines in/called by call dag node in resolution order":
            COLLECT
                :rrv

# Working routines
dagplan <
    dagnode  "cdag node"*
    sccplan  "planning data for routines in cdag node"*
    depplan  "planning data for called source routines"*
    supplan  "planning data for calling source routines"*
    rrv      "resolved routine name vector"*
    recurs   "recursiveness boolean"*
    >
        "ODIN planning data for cdag node"*:
            USER dagplan.cmd
                :cds
                :cdl
                :PARAMETERS < node >

cdag <
    cds "cdag structure"*
    cdl "cdag routine name set node label"*
    cdn "cdag node-> routine name mapping"*
    >
        "scc dag of cgrf"*:
            USER cdag.cmd
                :cgs
                :cgl

cgrf <
    cgs^pcgs@   "ref to cgrf structure"*
    cgl^pcgl@   "ref to cgrf label"*
    cgn^pcgn@   "ref to cgrf routine name mapping"*
```

```
    >
        "reqd cgrf for Cesar run"*:
            USER cgrfspec.cmd
                :refptr
                :tpptr

pcgrf <
    pcgs "cgrf structure"*
    pcgl "cgrf label"*
    pcgn "cgrf routine name mapping"*
    >
        "cgrf data"*:
            USER cgrf.cmd
                : (callU)
                : maincall

# Call data collection
callU(call)
        "program call data set"*:
            UNION
                : (callN)

callN(call)
        "set of module call data sets"*:
            P-HOMOMORPHISM ( :callM )
                : (prog)

# Object group data collections
ogrpU(ogrp)
        "set of module object group sets"*:
            P-HOMOMORPHISM ( :ogrpM )
                :(prog)

# Variable name data collections
prvarnam
        "names of variables represented by (possible) objects in a routine":
            USER prvarnam.cmd
                :varnamspecU

varnamspecU(.null)
        "var name resolve spec for routine"*:
            FLATTEN
                : (varnamspecN)

varnamspecN(.null@)
        "ODIN name of var name resolve for routine"*:
            USER varnamspec.cmd
                : refptr
                : tpptr
```

```
                : PARAMETERS < routine >

varnamU(varnam)
        "set of object->variable name maps"*:
            UNION
                :(varnamN)

varnamN(varnam)
        "set of module object->variable name map sets"*:
            P-HOMOMORPHISM ( :varnamM )
                :(prog)

# Event data collections
felU(fel)
        "program fel set"*:
            UNION
                :(felN)

felN(fel)
        "module fel sets"*:
            P-HOMOMORPHISM ( :felM )
                :(prog)

# Flowgraph collections
srcnam
        "names of routines in program source":
            CAT
                :(rouN)

rouN(rouM)
        "module routine name lists"*:
            HOMOMORPHISM ( :rouM )
                :(prog)


fgsU(fgs)
        "program fgrf structure set"*:
            HOMOMORPHISM ( :fgsM )
                :(prog)


fslU(fsl)
        "program fsl set"*:
            HOMOMORPHISM ( :fslM )
                :(prog)

# *** f77 subsystem
# User interface routines
list <= flist
```

```
flist(.null)
        "Fortran-77 listing":
            COLLECT
                : fsls

istpfg
        "Fortran-77 IST/TOOLPACK flowgraph":
            USER istpfg.cmd
                : fcpt
                : fcst
istpat
        "f77 semantic attribute table":
            USER istpat.cmd
                : fxst
                : fxat

istpst
        "Fortran-77 canonical symbol table":
            USER istpst.cmd
                : fcst

istppt
        "Fortran-77 canonical parse tree":
            USER istppt.cmd
                : fcpt
                : fcst

# f77 local call, event, object group data

felM     <= ffelM
ogrpM    <= fogrpM
callM    <= fcallM
varnamM  <= fvarnamM

mainfel      <= fmainfel
mainogrp     <= fmainogrp
maincall     <= fmaincall
mainvarnam   <= fmainvarnam
mainfgs      <= fmainfgs
mainfsl      <= fmainfsl

fmain <
    fmainfel    "f77 main prog fel"*
    fmainfgs    "f77 main prog fgs"*
    fmainvarnam "f77 main prog varnam"*
    fmaincall   "f77 main prog call"*
    fmainogrp   "f77 main prog ogrp"*
    fmainfsl    "f77 main prog fsl"*
    >
```

```
        "f77 main program local data"*:
            USER fmain.cmd
                : (fglobalU)
                : varnamU
                : PARAMETERS < tepee >

fglobalU(fglobalM)
        "set of all f77 module global data"*:
            P-HOMOMORPHISM (:fglobalM)
                : (prog)

flocal <
    ffelM[ffel]        "f77 module flowgraph event label"*
    fogrpM[fogrp]      "f77 module object group"*
    fcallM[fcall]      "f77 module call data"*
    fvarnamM[fvarnam] "f77 module object to variable name map"*
    fglobalM           "f77 module global data"*
    >
        "f77 module local data"*:
            USER flocal.cmd
                : fcpt
                : fcst
                : fxat
                : frouM
                : ffplM
                : PARAMETERS < tepee >

# f77 fgrf data

fgs   <= ffgs
fsl   <= ffsl

fgsM  <= ffgsM
fslM  <= ffslM
rouM  <= frouM

ffgrf <
    ffgsM[ffgs]     "f77 module fgrf structures"*
    ffslM[ffsl]     "f77 module stmt-number fgrf edge-labels"*
    ffplM[ffpl]    "f77 module parse-tree fgrf edge-labels"*
    frouM          "f77 list of routine names in a file"*
    >
        "f77 module local fgrf data"*:
            USER ffgrf.cmd
                : fcpt
                : fcst

# Canonicalized f77 parse tree
fcan <
```

```
    fcpt    "f77 canonical parse tree"*
    fcst    "f77 canonical symbol table"*
    >
        "f77 canonical parse structure"*:
            USER fcan.cmd
                : fxpt
                : fxst
                : fcmt

# f77 semantic analysis information

fsem <
    fxpt    "f77 extended parse tree"*
    fxst    "f77 extended symbol table"*
    fxat    "f77 extended attribute table"*
    fxls    "f77 semantic analyzer listing"*
    >
        "f77 semantic analyzer"* :
            USER fsem.cmd
                : fbpt
                : fbst

# f77 syntactic analysis information

fprs <
    fbpt    "f77 basic parse tree"*
    fbst    "f77 basic symbol table"*
    fbci    "f77 basic comment index table"*
    fpls    "f77 parser error list"*
    >
        "f77 parser"*:
            USER fprs.cmd
                : ftkn
                : fcmt

# f77 lexical analysis information

fscn <
    ftkn    "f77 token stream"*
    fcmt    "f77 comment table"*
    fsls    "f77 scanner listing"*
    >
        "f77 scanner"* :
            USER fscn.cmd
                : f

src <= f
f
        ATOMIC
```

```
        "Fortran-77 source file"

# Handling a f77 multiple-file program. Assume each file is a module.
prog(src)
        "references to program source files"*:
            COMPOUND
                : ref

ref
        ATOMIC
        "list of source/Excess file names"


# *** The Cecil sequencing specification language subsystem
# User interface routines

praqre
        "Cecil AQRE semantic data":
            USER praqre.cmd
                :cedptr
                :cadptr
                :ancptr
                :txtptr

aqrespec <
    cedptr^ced "name of ced"*
    cadptr^cad "name of cad"*
    ancptr^anc "name of anc"*
    txtptr^txt "name of txt"*
    >
        "ODIN name of AQRE data for single aqre"*:
            USER aqrespec.cmd
                : (aqrenamU)

aqrenamU(.null)
        "set of ODIN names of AQRE data for a single aqre"*:
            UNION
                : (aqrenamN)

aqrenamN(.null@)
        "set of ODIN names of AQRE data for single aqre"*:
            USER aqrenam.cmd
                : cesname
                : cecptr
                : PARAMETERS < aqre >

# Working routines
aqrekeyptr^aqrekey@
        "pointer to aqrekey object"*:
```

```
            USER aqrekeyptr.cmd
                :cecptr

aqrekey
        "keys of aqre's for a cecil expression":
            CAT
                :(aqrekeyU)

aqrekeyU(.simple)
        "union of keys of aqre's for a Cecil expression"*:
            HOMOMORPHISM (:key)
                :(ancU)

txtlist^txtset@
        "pointer to list of aqre text objects"*:
            USER txtlist.cmd
                :cecptr

txtset
        "set of aqre text objects"*:
            CAT
                :txtU

cecsem <
    ancU[anc] "Cecil anchor data set"*
    cadU[cad] "Cecil analysis data set"*
    cedU[ced] "Cecil evaluation data set"*
    txtU[txt] "text of the AQRE"*
    cecexp    "Cecil expression structure"*
    >
        "Cecil semantic analysis"*:
            USER cecsem.cmd
                :cecfe

cecfe
        "Cecil syntax tree"*:
            USER cecfe.cmd
                :cec

cec
        ATOMIC
        "Cecil source code"

# *** The Excess external sequencing assertion language subsystem

# graph data for an Excess module
fgs     <= xsfgs
fel     <= xsfel
rou     <= xsrou
```

```
ogrp    <= xsogrp
varnam  <= xsvarnam
fsl     <= xsfsl
call    <= xscal

fgsM    <= xsfgsM
felM    <= xsfelM
rouM    <= xsrouM
ogrpM   <= xsogrpM
varnamM <= xsvarnamM
fslM    <= xsfslM
callM   <= xscalM

#  We must add a tool that generates ".main" global information for
#  every supported programming language.  This should be easy since
#  we know that that empty data structure's file looks like.
#  Every supported programming language will have to collect information
#  from all modules and we want to have "Excess" files treated exactly
#  like other program files.  E. g.  Fortran is a list of 2 empty sets and
#  the nil identifier in Lisp format  ( () () () ), one paren per line.

fglobalM <= xsfglobalM
xsfglobalM
        "empty FORTRAN global information"*:
            USER xsfglobal.cmd
                : xsfe

xsgrf <
    xsfgsM[xsfgs]       "fgs for Excess module"*
    xsfelM[xsfel]       "fel for Excess module"*
    xscalM[xscal]       "call data for Excess module"*
    xsogrpM[xsogrp]     "ogrp for Excess module"*
    xsfslM[xsfsl]       "fsl's for Excess module"*
    xsvarnamM[xsvarnam] "varnams for Excess module"*
    xsrouM              "names of routines for Excess module"*
   >
        "graph data for Excess module"*:
            USER xsgrf.cmd
                : xsfe

# Excess front end tool
xsfe
        "Excess intermediate form"*:
            USER xsfe.cmd
                : xs

src <= xs
xs
        ATOMIC
```

```
        "Excess source code"

# *** The Tepee event pattern language subsystem
tpfeptr^tpfe@
        "pointer to Tepee front end intermediate form"*:
            USER tpspec.cmd
                : tpptr


tpfe
        "Tepee front end"*:
            USER tpfe.cmd
                : tp

tp
        ATOMIC
        "Tepee source code"

# Standard Specification Header


err "errors generated while producing derivation for display" :
    ERROR
        : (.view)

warn "warnings generated while producing derivation for display" :
    WARNING
        : (.view)

name "names of the elements of a compound file" :
    NAME
        : .compound

key "key values (for selection)" :
    KEY
        : .simple

union (.null) "union of the elements of a compound file" :
    UNION
        : (.compound)


# .stat <= .derived
# .stat <= .c.name
# .stat <= .cat

.view <= .c.name
.view <= .cat
.view <= .s.name
```

```
.view <= .simple


.c.name "names of elts of composite file"* :
    NAME
        : .composite


.cat "contents of compound file"* :
    CAT
        : (.compound)


.error "errors gen'd while producing drvn"* :
    ERROR
        : (.derived)


.cmpd (.null) "files named in ref file"* :
    COMPOUND
        : .simple

.s.name "names of sentinels watching a file"* :
    NAME
        : .sentinel

.sentinel (.null) "sentinels watching a file"* :
    SENTINEL
        : .simple
        : "| : .cmpd"

.check (.null) "check effect of copy command file"* :
    COPYTST
        : (.copy_sntU)
        : (.copy_dsc)
        : PARAMETERS < (check) >

.copy_sntU (.null) "sentinels of dests of copy command file"* :
    UNION
        : (.copy_snt)

.copy_snt (.null) "sentinels of dests of copy command file"* :
    HOMOMORPHISM ( :.sentinel )
        : (.copy_dst)

.copy_dsc (.null) "descr of copy command"* :
    COLLECT
        : (.copy_org)
        : (.copy_dst)
```

```
.copy_chk <
   .copy_org (.null) "origin files in copy command file"*
   .copy_dst (.null) "dest files in copy command file"*
   > "files in a copy command file"* :
    COPYCHK
        : .simple
```