Organizing Software in a Distributed Environment

Butler W. Lampson
Eric E. Schmidt

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, CA 94304

Abstract

The System Modeller provides automatic support for several different kinds of program development cycle in the Cedar programming system. It handles the daily evolution of a single module or a small group of modules modified by a single person, the assembly of numerous modules into a large system with complex interconnections, and the formal release of a system. The Modeller can also efficiently locate a large number of modules in a big distributed file system, and move them from one machine to another to meet operational requirements or improve performance.

1. Introduction

The *System Modeller* is a complete software development system used in the Cedar project of Xerox PARC's Computer Science Laboratory [2]. The Modeller provides automatic support for the program development cycle followed by programmers using Cedar. It uses information stored in a *system model*, which describes a software system by specifying:

1) The *versions* of various modules that make up a particular software system.
2) The *interconnections* between modules, such as which procedures are used and where they are defined.
3) Additional information needed to compile and load the system.
4) Hints for *locating* the modules in a distributed file system.

Under the direction of the Cedar programmer, the Modeller performs a variety of operations on the systems described by the system models:

1) It implements the representation of the system by source text in a collection of *files.*
2) It tracks *changes* made by the programmer. To do this, it is connected to the Cedar editor and is notified when files are edited and new versions are created.
3) It automatically *builds* an executable version of the system, by recompiling and loading the modules. To provide fast response, the Modeller behaves like an incremental compiler: only those modules that change are analyzed and recompiled.
4) It provides complete support for the integration of packages as part of a *release.*

Thus the Modeller can manage the files of a system as they are changing, providing a user interface through which the programmer edits, compiles, loads and debugs her changes interactively while she is developing her software. The models are automatically updated to refer to the changed components. Manual updates of models by the programmer are not normally necessary.

Related work is described in [1, 3, 4, 5]. This paper is derived from part of the second author's Ph.D. thesis [8].

1.1 Background

The Modeller runs in Xerox PARC's computing environment, in which each programmer has a personal computer, connected to other computers over an Ethernet. It currently supports programming in Cedar, though its techniques do not depend on the languages in which modules are written. Cedar is derived from Mesa [7], and shares with that language a very general mechanism for interconnecting modules; hence it is a good test of the Modeller's facilities for module interconnection.

The programmer writes a model in a language called SML, which is a notation for describing how to compose a set of related programs from their components. The model refers to a component module of the program by its unique name, independently of the location in the file system where its bits are stored. The development of a program can be described by a collection of models, one for each stage in the development; certain models define releases.

SML has general facilities for abstraction. These are of two kinds:

> A model can be organized hierarchically into parts, each of which is a set of named sub-parts called a *binding*. Like the names of files in a directory, the names in a binding can be used to select any desired parts of the binding.
>
> A model can be *parameterized*, and several different versions can be constructed by supplying different arguments for the parameters. This is the way that SML caters for planned variation in a program.

SML is described in detail in [6].

The distributed computing environment means that files containing the source text of a module can be stored in many places. A file is accessed most efficiently if it is on the programmer's own machine. Remote files must first be located and then retrieved. The Modeller imposes minimal requirements on the capabilities of the distributed file system. In fact, it requires only that there be a way to enumerate the versions of a particular file in a remote directory, and to store or retrieve an entire remote file. When possible, it caches information about a module, such as its dependencies on other modules, to avoid retrieving the entire module and parsing its text. It also caches the complete path names of objects, to avoid searches in remote directories.

1.2 Organization of the paper

We begin by describing how a model completely and unambiguously specifies a Cedar program (§ 2). The next section presents the user interface of the Modeller, and shows how it is used during daily program development, and for periodic releases of a complete system (§ 3). The implementation techniques used to obtain good performance with systems containing dozens or hundreds of modules are explained next (§ 4), and then the interactions between the Modeller and the distributed file system (§ 5). A final section describes experience and future plans (§ 6). An appendix gives a complete model for a substantial component of the Cedar system.

2. System models

We take the view that the software of a system is completely described by a single unit of text. An appropriate analogy is the sort of card deck that was used to run a program on a bare computer or under an operating system like FMS that had no file system. Everything is said explicitly in such a system description: there are no parameters (e.g., compiler switches or loader options) supplied after the GO button is pressed, and no dependence on a changing environment. In this kind of system description there is no question about when to recompile something, and version control is handled by distributing copies of the deck with a version number written on the top of each copy, and a diagonal stripe of marker which makes it easy to tell whether the deck has been changed.

The monolithic nature of a card deck makes it unsuitable for a large system. In 1983 a system is specified by text which is stored in files. This provides modularity in the physical representation: a file can name other files instead of literally including their text. In Cedar, these files hold the text of Cedar modules or system models. This representation is convenient for users to manipulate; it allows sharing of identical objects, and facilitates separate compilation. Unless care is taken, however, the integrity of the system will be lost, since the contents of the named files may change.

2.1 Objects

To prevent this, we abstract files into named *objects*, which are simply pieces of text. We require that names be *unique* and objects be *immutable*. By this we mean that:

> Each object has a unique name, never used for any other object. The name is stored as part of the object, so there is no doubt about whether a particular collection of bits is the object with a given name. A name is made unique by appending a *unique identifier* to a human-sensible string.
>
> The contents of an object never change once the object is created. The object may be erased, in which case the contents are no longer accessible. If the file system does not guarantee immutability, it can be ensured by using a suitable checksum as the unique identifier of the object.

These rules ensure that a name can be used instead of the text of an object without any loss of integrity, in the sense that either the entire text of a system will be correctly assembled, or the lack of some object will be detected.

What happens when a new version V_2 of an object is created? In this view, such a version is a new object. Any model M_1 which refers to the old object V_1 continues to do so. However, it is possible to create a new model M_2

which is identical to M_1 except that every reference to V_1 is replaced by a reference to V_2; this operation is called *Notice* and is discussed further in § 3. In this way, the notion that objects are immutable is reconciled with the fact of evolution.

With these conventions, a model can incorporate the text of an object by using the name of the object. This is done in SML by writing an object name preceded by @. The meaning of an SML expression containing an @-expression is defined to be the meaning of an expression in which the @ expression is replaced by its contents. For example, if the object *inner.model* contains

```
"lit"
```

which is an SML expression, the binding

```
[x: STRING ~ @inner.sm,
 y: STRING ~ "lit"]
```

has identical values for *x* and *y*.

With these conventions, a system model is a stable, unambiguous representation for a system. It is easily transferred among programmers and file systems. It has a readable text representation that can be edited by a user at any time. Finally, it is usable by other program utilities such as cross-reference programs, debuggers, and optimizers that analyze inter-module relationships.

2.2 Derived objects

The Modeller's most important function is to rebuild a system from its source components, given the system model as input. The model refers to each component by its unique name. To rebuild the system, the Modeller goes to the file system and tries to find the file which represents each object. This may involve a search in several directories on several machines, as described in § 5. Because each file contains the unique name of the object it represents, the Modeller will never make a mistake and retrieve the wrong version, although it may be unable to retrieve a file. Once it has obtained all the files, the Modeller does any necessary recompilations, loads the resulting binary files, and runs the program.

A model normally refers to source objects rather than the binary objects produced by the compiler. The Modeller takes the view that a binary object is just an accelerator, since it can be recreated by the compiler using the right source object and parameters. Of course, wholesale recompilation is time-consuming, so various caches are used to avoid unnecessary recompilation.

It is not essential that the text of a system component be source text; all that is needed is a way to turn it into a value the Modeller can understand. For a Cedar source module, this is done by parsing the DIRECTORY, IMPORTS and EXPORTS statements at the start of the module (see § 2.4). But it can also be done for a Cedar binary module, which is the output of the compiler and has all its interface parameters bound; binary modules have enough information (originally for the benefit of the loader) to allow an SML function or INTERFACE value to be derived. This is sometimes convenient when dealing with a system in which some elements come from an outside organization in binary form only.

2.3 Unique names

The Modeller uses the creation date of a source object as its unique identifier. Thus an object name might have the form *BTree.cedar*!(July 22, 1982 2:23:56); in this representation the unique identifier follows the ! character. Of course such an identifier is not absolutely guaranteed to be unique, but we have found it satisfactory in practice.

For a derived object such as a binary module, the Modeller uses a 48-bit *version stamp* which is constructed by hashing the name of the source object, the compiler version and switches, and the version stamps of any interfaces which are parameters of the compilation. In this way derived objects constructed at different times will have the same names, as long as they are made in *exactly* the same way. This property can make a considerable difference in the time required to rebuild a system when some binary modules must be rebuilt, especially if there are other modules which depend on the ones being rebuilt.

It is also possible to use an ambiguous name for an object, of the form *BTree.cedar*!H. This means to consider all the objects whose names begin *BTree.cedar*, and take the one with the most recent create date. Since such a name does not denote a unique object, a model containing such a reference does not denote a unique program. Nonetheless, it is often convenient to use this convention.

2.4 Example

A Cedar program consists of a set of modules. There are two kinds of module: *implementation* (PROGRAM) modules, and *interface* (DEFINITIONS) modules. An interface module contains constants (numbers, types, inline procedures, etc.) and declarations for values to be supplied by an implementation (usually procedures, but also types and other values). A module M_1 that calls a procedure in another module M_2 must IMPORT an *instance Inst* of an interface *I* that declares this procedure. *Inst* must be EXPORTed by the PROGRAM module M_2. For example, a procedure *Insert* declared in a module *BTreeImpl* would also be declared in an interface *BTree*, and *BTreeImpl* would EXPORT an instance of *BTree*. A PROGRAM calls *Insert* by IMPORTing this instance of *BTree* and referring to the *Insert* component of the instance. We call the importer

of *BTree* the *client* module, and say that *BTreeImpl* (the exporter) *implements BTree*. Of course *BTreeImpl* may itself IMPORT and use interfaces that are defined elsewhere.

Example 1 shows a very simple system called *BTree*, which defines one interface *BTree* and one instance *BTreeInst* of *BTree*.

```
BTree.model!(Jan 14, 1983, 14:44:11)
LET @[Indigo]<Cedar>CedarInterfaces.model!(July 25, 1982, 14:03:03) IN
LET Instances~@[Indigo]<Cedar>CedarInstances.model!(July 25, 1982, 14:10:12) IN
  BTree: INTERFACE BTree ~ @[Ivy]<Schmidt>BTree.cedar!(Sept 9,1982, 13:52:55)
    [Ascii],
  BTreeInst: BTree ~ @[Ivy]<Schmidt>BTreeImpl.cedar!(Jan 14, 1983, 14:44:09)
    *[] [Instances.Rope, Instances.IO, Instances.Space] ] ]

CedarInterfaces.model!(July 25, 1982, 14:03:03)
[
  Ascii: INTERFACE ~ @[Indigo]<Cedar>Ascii.cedar!(July 10, 1982, 12:25:00)[],
  Rope: INTERFACE ~ @[Indigo]<Cedar>Rope.cedar!(July 10, 1982, 17:00:00)*[],
  IO: INTERFACE ~ @[Indigo]<Cedar>IO.cedar!(July 12, 1982, 11:00:00)*[],
  Space: INTERFACE ~ @[Indigo]<Cedar>Space.cedar!(June 10, 1982, 8:35:00)*[] ]

CedarInstances.model!(July 25, 1982, 14:10:12)
[Ascii, Rope, IO, Space]~
  LET @CedarInterfaces.model!(July 25, 1982, 14:03:03) IN [
    @[Indigo]<Cedar>AsciiImpl.cedar!(July 10, 1982, 12:30:00)[] [],
    @[Indigo]<Cedar>RopeImpl.cedar!(July 10, 1982, 17:10:24)*[]*[],
    @[Indigo]<Cedar>IOImpl.cedar!(July 20, 1982, 13:03:03)*[]*[],
    @[Indigo]<Cedar>SpaceImpl.cedar!(June 11, 1982, 15:00:00)*[]*[] ]
```

Example 1: An example of a model

BTree.model refers to two modules, *BTree.cedar*!(Sept 9, 1982, 13:52:55) and *BTreeImpl.cedar*!(Jan 14, 1983, 14:44:09). Each is named by a user-sensible name (e.g., *BTree.cedar*), part of which identifies the source language as Cedar, and a creation time (e.g. !(Sept 9, 1982, 13:52:55)) to ensure uniqueness. The @ indicates that a unique object name follows. Each object also has a file location hint ([Ivy]<Schmidt>); its use is discussed in § 5.1.

BTree.model refers to two other models, *CedarInterfaces.model*!(July 25, 1982, 14:03:03) and *CedarInstances.model*!(July 25, 1982, 14:10:12). Each of these is a binding which gives names to four interface or instance modules that are part of the Cedar system. A clause such as

```
LET CedarInterfaces.model IN . . .
```

makes the names bound in *CedarInterfaces* (*Ascii*, *Rope*, *IO*, *Space*) denote the associated values (*Ascii.cedar*!(July 10, 1982, 12:25:00)[], etc.) in the expression following the IN.

Models denote dependency by parameterization. There are two kinds of dependency: on interfaces, and on implementations, or instances of the interfaces. Correspondingly, each source module is viewed as a function which takes interface arguments and returns another function which takes instance arguments. Applying the first function to its interface arguments is done by the compiler; applying the resulting second function to its instance arguments is done by the loader as it links up definitions with uses.

In the example above, the *BTree* interface depends on the *Ascii* interface from *CedarInterfaces*. Since it is an interface, it doesn't depend on any implementations. *BTreeImpl* depends on a set of interfaces which the model doesn't specify in detail: the * in front of the first parameter list for *BTreeImpl* means that its arguments are defaulted by name matching from the environment. In particular, it probably has interface parameters *BTree*, *Rope*, *IO*, and *Space*; all these names are defined in the environment, *BTree* explicitly and the others from *CedarInterfaces* through the LET clause. *BTreeImpl* also depends on *Rope*, *IO*, and *Space* instances from *CedarInstances*, as indicated in the second argument list.

The interface parameters are used by the compiler for type-checking, and so that details about the types can be used to improve the quality of the object code. The instance parameters are used by the loader; they specify how procedures exported by one module should be linked to other modules which import them.

3. User interface

The Modeller provides an interactive interface for ordinary incremental program development. When used interactively, the role of the Modeller is similar to that of an incremental compiler: it tries to do as little work as it can as quickly as possible in order to produce a runnable system. To do this, it keeps track incrementally of as much information as possible about the objects in the active models.

3.1 Patterns of software development

For example, consider the following scenario. Assume a model already exists, say *BTree.model*, and the user wants to change one module to fix a bug. Earlier, she has started the Modeller with *BTree.model* as the *current model*. She uses the Cedar editor to make a change to *BTreeImpl.cedar*!(Jan 14, 1983, 14:44:09). When the user finishes editing the module and creates a new version *BTreeImpl.cedar*!(April 1, 1983, 9:22:12), the editor notifies the Modeller by calling its *Notice* procedure, indicating that *BTreeImpl.cedar*!(April 1, 1983, 9:22:12) has been produced from *BTreeImpl.cedar*!(Jan 14, 1983, 14:44:09). If the latter is referenced by the current model, the Modeller notices the new version and updates *BTree.model*!(Jan 14, 1983, 14:44:11) to produce *BTree.model*!(April 1, 1983, 9:22:20), which refers to the new version. The user may edit and change more files. When she wants to make a runnable version of her system, she issues another command to the Modeller, which then compiles everything in correct order and (if there are no errors) produces a binary file.

A more complex scenario involves the parallel development of the same system by two programmers. Suppose both start with a system described by the model M_0, and end up with different models M_1 and M_2. They may wish to make a new version M_3 which *merges* their changes. The Modeller can provide help for this common case as follows: If one programmer has added, deleted or changed some object not changed by the other, the Modeller will add, delete, or change that object in a merged model. If both programmers have changed the same object in different ways, the Modeller cannot know which version to prefer and will either explore the changed objects recursively, or ask the user for help.

More precisely, we have

$M_3 = Merge[Base \sim M_0, New_1 \sim M_1, New_2 \sim M_2]$

and *Merge* traces out the three models depth-first. At each level, for a component named *p*:

If	*Add to result*
$Base.p = M_1.p = M_2.p$	$Base.p$
$Base.p = M_{1/2}.p \neq M_{2/1}.p$	$M_{2/1}.p$
$Base.p = M_{1/2}.p$, no $M_{2/1}.p$	leave *p* out
no $Base.p$ or $M_{1/2}.p$	$M_{2/1}.p$
$Base.p \neq M_1.p \neq M_2.p$, all models	$Merge[Base.p:, M_1.p, M_2.p]$
ELSE	error, or ask what to do.

Of course, there is no guarantee that the resulting thing makes any sense, but it does seem to correspond to current practice.

At all points, the Modeller maintains a model that describes the current program. When the user decides to save her program, she does so with an accurate description of it in her model. Since the models are simply text files, the user always has the option of editing the model as she sees fit, so the Modeller does not have to deal with obscure special cases of editing that may arise.

3.2 Daily evolution

In a session which is part of the daily evolution of a program, the user begins by creating an instance of the Modeller, which provides a window on the Cedar user's screen, as shown in Figure 1. This section gives an overview of its use, suggested by the contents of the figure.

The Modeller window is divided into four regions, which are, from top to bottom:

1) A set of buttons to control it.
2) A region containing fields where names may be typed.
3) A feedback area for compiler progress messages.
4) A feedback area for Modeller messages.

To help explain Modeller operation, let us take a simple example and follow the user's actions.

Step 1. Assume that the Modeller instance has just been created. The user decides to make changes to the modules in *Example.Model*. She enters the name of the model in the field following the *ModelName:* prompt, and pushes the *StartModel* button. From this point on the Modeller is bound to *Example.Model*; *StopModel* must be pushed before using this instance of the Modeller on another model. *StartModel* initializes data structures in this instance of the Modeller, *StopModel* frees the data.

Step 2. The user makes changes to objects on her personal machine. The Cedar editor calls the Modeller's *Notice* procedure to report that a new version of an object exists; the user could do this by hand, but normally new versions correspond one-to-one with editing sessions on modules. If the object being edited is in the model, the Modeller

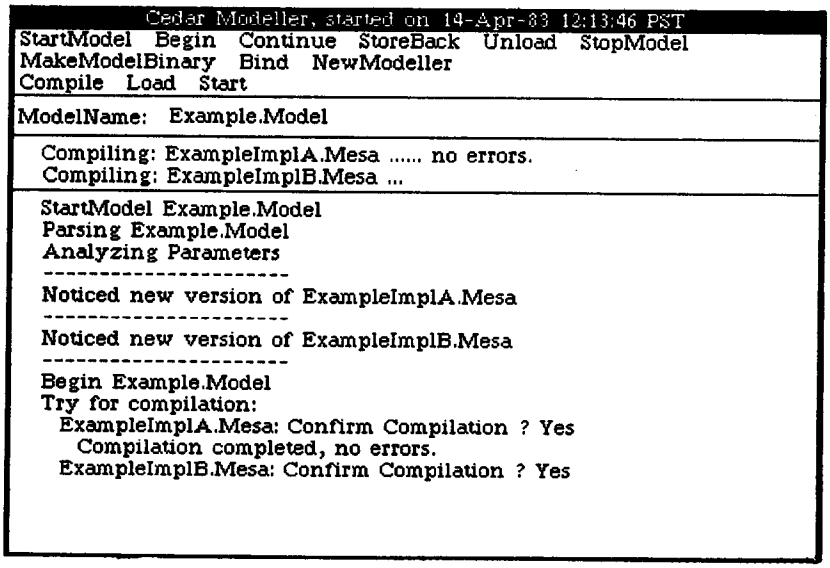

Figure 1 Modeller Window

updates its internal representation of the model to reflect the new version. If the changes involve adding or deleting parameters to modules, the Modeller uses standard defaulting rules to modify the argument list for the object in the model.

Step 3. Once she has made the intended edits, the user pushes *Begin*, which

a) recompiles modules as necessary,

b) loads their object files into memory, and

c) forks a process that starts the user's program running.

Modules need to be recompiled if the corresponding source files have been changed, or if any modules they depend on have been compiled. Should (a) or (b) encounter errors, the Modeller does not proceed to (c).

Step 4. After testing her program, the user may want to make changes simple enough that the old module may be replaced by the new module without re-loading and restarting the system. If so, after editing modules, the user pushes *Continue*, which tries to replace modules in the already-loaded system. If this succeeds, she can go on testing her program and the new code will be used. If the module is not replaceable, she must push *Begin*, which unloads all the old modules in this model and loads the new modules.

Step 5. After completing her changes, the user can push *StoreBack* to store copies of her files on remote file servers, and then push *Unload* to unload the modules previously loaded and *StopModel* to free Modeller data structures.

These steps are illustrated in Figure 2.

The modeller allows a Cedar program to be rebuilt and restarted from scratch. It also is able to replace a module in an already loaded system. This is considerably faster for small program changes, and means that the current state of the program is not lost. Module replacement in Cedar is possible if certain conditions are met: the global data must not change, all previously-defined procedures must still be defined, certain architectural limitations must be observed, and the module being replaced cannot be executing when replacement occurs.

In addition to its interactive interface with the user, the Modeller also provides a procedural interface to its data structures, which contain complete information about the structure of the program: what modules exist, how they are interconnected, and what they are named. The main client of this interface is the Cedar debugger. When the debugger examines a stopped system (e.g. at a breakpoint), it can follow the procedure call stack and find the global variables for the module in which the procedure is declared. The Modeller can provide the debugger with module-level information about the model in which this module appears, and provide file location and version information (i.e. an interface to a sophisticated load map). This is particularly useful when the debugger wants to inspect the symbol table for a module, and the symbol table is stored in another file that is not on the local disk.

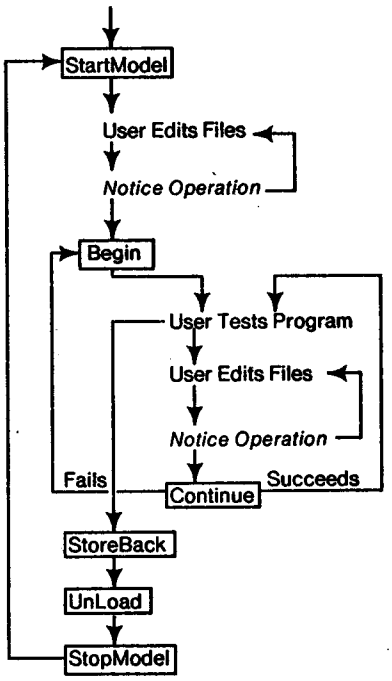

Figure 2 User Sequence

3.3 Releases

A *release* is a software system composed of a collection of modules which have been tested for conformance to some kind of specification, and filed so that any one of them can be retrieved simply and reliably as long as the release remains active. The *Release* procedure in the Modeller takes a model, performs various *checks* on its components, *builds* the system it describes, and *moves* the system and all the components to designated directories. In more detail, *Release*[*M*]:

1) Checks that *M* and each component of *M* is legal: syntactically correct, type-correct, and causes no compiler errors.

2) Ensures that all objects needed by any component of *M* are components of *M*, and that only one version of each object exists (unless multiple versions are explicitly specified).

3) Builds the system described by *M*.

4) Copies all the files representing objects in *M* to a place where they cannot be erroneously destroyed or modified.

A release is *complete* if and only if every source file needed to compile every object file is among the files being released. A release is *consistent* if and only if only one version of each package is being released, and other packages depend only on that version. The release process is controlled by a person acting as a *Release Master*, who runs the Modeller to verify that a proposed release is consistent and complete, and takes corrective action if it is not. Errors in models, such as references to non-existent files or references to the wrong versions of files, are detected by the *Release* procedure of the Modeller. When errors are detected, the Release Master notifies the guilty implementor and has her fix the model.

Releases can be frequent, since performing each release imposes a low cost on the Release Master and on Cedar programmers. The Release Master does not need to know any details about the packages being released, which is important when the software of the system becomes too large to be understood by any one programmer. The implementor of each package can continue to make changes until the release occurs, secure in the knowledge that the package will be verified before the release completes (of course, the release process provides no protection against bugs which do not cause errors at compile time). Many programmers make such changes at the last minute before the release. The release process supports a high degree of parallel activity by programmers engaged in software development.

We have extensive experience with Cedar releases [8]. The Cedar software under release control consists of approximately 5000 files and 465,000 lines of Cedar code. Existing packages are described by *DF files* that contain a subset of the information in system models. In what follows, we describe the release process when, in the future, it will be run using system models instead of DF files.

3.3.1 The Top *model*

The Release Master maintains a model with one component for each component of the release. This list (called the *Top* model) defines, for every model named in the list, a file server and directory where it can be found. While a release is being developed, this model refers to objects on their working directories, e.g., the top model might be

```
Top ~ [
  BTree~ @[Indigo]<Int>BTree.Model!H --ReleaseAs [Indigo]<Cedar>--,
  Runtime ~ @[Indigo]<Int>Runtime.Model!H --ReleaseAs [Indigo]<Cedar>--
]
```

The *Top* model is used during the development phase as a description of models that will be in the release; it gives the locations of these objects while they are being developed. The *Top* model provides the list of models that will be released. Models not mentioned in the *Top* model will not be released.

3.3.2 Release mechanics—client

Every model *M* being released must have a LET statement at the beginning that makes the components in the *Top* model accessible in *M*. Thereafter, *M* must use the names from *Top* to refer to other models. Thus, *M* must begin

```
LET @[Indigo]<Int>Top.Model!H IN [
  ...
  RTTypes: INTERFACE ~ Runtime,
  ....
]
```

Clients of a release component (e.g., *RTTypes*) are not allowed to refer to its model by @-reference, since there is no way to tell whether that model is part of the release. Aside from the initial reference to *Top*, a release component may have @-references only to sub-components of that component.

3.3.3 Release mechanics—implementor

A model *M* being released must also have a comment that gives its object name in the *Top* model (e.g. *BTree*), and the working directory that has a copy of the model, e.g.

```
-- ReleaseName BTree
-- WorkingModelOn [Indigo]<Int>BTree.Model
```

These comments are redundant; they allow a check that *Top* and the component (and hence the Release Master and the implementor) agree about what is being released.

M must also declare the release position of each file, by appending it as a comment after the filename in the model, e.g.

```
@[Ivy]<Work>XImpl.Mesa!11 -- ReleaseAs [Indigo]<Cedar>XPack>-- [ ]
```

A global *ReleaseAs* comment can define the default release position of files in the model (which may differ from the release position of the model itself). Thus if the model contains a comment

```
-- DefaultReleaseAs [Indigo]<Cedar>BTrees>--
```

then the user may omit the

```
-- ReleaseAs [Indigo]<Cedar>BTrees>--
```

clauses.

4. Implementation

The Modeller must be able to analyze large collections of modules quickly, and must provide interfaces to the compiler, loader, debugger, and other programs. This section describes first the basic algorithms used, and then the caches which greatly improve performance in the normal case of incremental changes to a large system. It ends with a description of the algorithms used for releases.

4.1 Evaluation

In order to build a program, the Modeller must *evaluate* the model for the program. The model is an expression

written in SML, which is a strongly typed, applicative language. Evaluating an SML expression is done in three steps:

1) The standard β-reduction evaluation algorithm of the typed lambda calculus converts the expression into one in which all the applications are of primitive objects, namely Cedar modules. Each such application corresponds to compilation or loading of a module. β-reduction works by simply substituting each argument for all occurrences of the corresponding parameter. SML operations such as selecting a named component of a binding are executed as part of this process. Thus in the example,
   LET *Instances*~@*CedarInstances.model* IN *Instances.Rope*
   evaluates to
   @[Indigo]<Cedar>*RopeImpl.cedar*!(July 10, 1982, 17:10:24)[...] [...]
   where the arguments of *RopeImpl* are filled in according to the defaulting rules.

2) Each application of a *.cedar* object is evaluated by the compiler, using the interface arguments computed by (1). The result is a *.binary* object. Of course, each interface argument must itself be evaluated first; i.e., the interfaces on which a module depends must be compiled before the module itself can be compiled.

3) Finally, each application of a *.binary* object computed in (2) is evaluated by the loader, using the instance arguments computed by (1). Cedar permits mutual recursion between procedures in different modules, so it is not always possible to fully evaluate the instance arguments. Instead, for each instance of an interface a record is allocated, with space for all the components of the interface. A pointer to the record is passed as an argument, rather than the record itself. Later, when the *.binary* object application which defines the interface has been evaluated by loading the object, the record is filled in with the results, namely the procedures and other values defined by that module.

Once everything has been loaded, the result is a runnable version of the program.

We now proceed to examine (1) in more detail. This step is done when the user pushes the *StartModelling* button, or on the affected subtree whenever the current model is modified by a *Notice* operation. For *StartModelling*, the Modeller reads the model from its source file, parses the source text and builds an internal parse tree. For *Notice*, the parse tree already exists, and is simply modified by substituting the new version for each occurrence of the old one. The leaves of this parse tree are the Cedar modules referenced with @ from the model. If another model is referenced, it does not become a leaf; instead, its parse tree is computed and becomes a sub-tree of the containing model.

After the parse tree is built, it is evaluated to produce a value tree. The evaluation applies functions (by substituting arguments for parameters in the function body), looks up names in bindings, does type checking, and supplies defaulted arguments. The first two operations have already been discussed. Typechecking requires knowing the type of every value. For a value which is a Cedar module, the Modeller obtains its type by examining the first few lines of the module, where the interfaces and instances imported by the module are declared (in DIRECTORY and IMPORTS clauses), together with the instances exported (in an EXPORTS clause).

For example, a module *M* which uses interfaces *A* and *B*, imports an instance of *A*, and exports an instance of *B*, begins

```
DIRECTORY A, B;
M: PROGRAM
  IMPORTS A;
  EXPORTS B;
...
```

and has the type

[INTERFACE *A*, INTERFACE *B*]→[[*A*]→[*B*]]

I.e., it is a function taking two interface arguments and returning (after it is compiled) another function that takes an instance of *A* and returns an instance of *B*. The Modeller checks that the arguments supplied in the model have these types, and defaults them if appropriate. SML typechecking is discussed in detail in [6].

After the entire model has been evaluated, the Modeller has determined the type of each module, and has checked that every module gets arguments of the types it wants. Any syntactic or type errors discovered are reported to the user. If there are none, then wherever a value is defined in one module and used in another, the two modules agree on its type. Note that nothing has yet been compiled or loaded.

After step (1) the value of the model is a tree with one application for each compilation or loading operation that must be done. The compilation dependencies among the modules are expressed by the arguments: if module *A* is an argument to module *B*, then *A* must be compiled first, and if *A* changes, *B* must be recompiled. Because of the level of indirection in the implementation of loading, it is not necessary to reload a module when other modules change.

To get from this tree to a fully compiled program, each application of a source module must be evaluated by the compiler, as described in (2). During this evaluation, the compiler may find errors within the module. This step is done when the user pushes the *Compile* or *Begin* button.

After step (2), the value of the model is a tree in which each application of a source object has been replaced by the binary object that the compiler produced. To get from

this tree to a runnable program, each binary object must be loaded, and each instance record filled in with the procedures exported from the modules that implement it. The details of how this is done are very dependent on the machine architecture and the runtime data structures of the language.

4.2 Accelerators

It is impractical to repeat the entire procedure just described whenever any change is made to a system; among other things, this would imply recompiling every module. Since the entire system is applicative, however, and the value of an object never changes, the results of any computation can be saved in a cache, and reused instead of repeating the computation. In particular, the results of the type analysis of objects and the results of compilations can be saved. To this end, the Modeller keeps two tables that record the results of computations that are expensive to repeat. These tables serve as accelerators for the Modeller and are stored as files on the local disk.

Object Type Table: A list of objects that are referenced by models and have been analyzed for their types. For example, a Cedar source module is listed along with the implied procedure type used by the Modeller to compile and load it. The unique name of an object is the key in this table, and its type is the value.

Projection Table: A list of entries that describe the results of running a compiler (or other program) that takes a source object and any needed parameters (such as interfaces) and produces an binary object. Before invoking a compiler to produce a binary file, the Modeller consults this table to see if such a file is already available. The key in this table is all the information that affects the result: the name of the source object, the names of all the parameter objects, the compiler switches, and the compiler version. The value of a table entry is the name of the binary object that results. This name is constructed from the user-sensible name of the source object, plus the version stamp, a 48-bit hash code of all the other information. An entry is added to the projection table whenever the compiler is run successfully.

In summary, the object type table speeds the analysis of files, and the projection table speeds the translation of objects into derived objects. These tables are illustrated in Example 2.

It is possible for these tables to fill up with obsolete information. Since they are just caches and can always be reconstructed from the sources, or from information in the *.modelBinary* objects (see § 4.3), they can be purged by any convenient method, including deleting them completely. As information is needed again, it will be recomputed and reentered in the tables.

The projection table is augmented by a different kind of cache provided by the file system. Whenever the result of a needed compilation is not found in the projection table, the Modeller constructs the 48-bit version stamp that the resulting binary object will have (by hashing the source name and parameters), and searches for this object in the file system, as described in § 5. If it is found, the compilation need not be redone; the result is put into the projection table so that the file system need not be searched again. This search of the file system is suppressed for source files that have just been edited, since it would never succeed in this case.

Object type table

Source object	*Type*
BTree.cedar!(Sept 9, 1982, 13:52:55)	[INTERFACE *Ascii*]→[INTERFACE *BTree*]
BTreeImpl.cedar!(Jan 14, 1983, 14:44:09)	[*Rope*: INTERFACE *Rope*, *IO*: INTERFACE *IO*, *Space*: INTERFACE *Space*, *BTree*: INTERFACE *BTree*]→ [[*RopeInst*: *Rope*, *IOInst*: *IO*, *SpaceInst*:*Space*]→[*BTreeInst*: *BTree*]]

Projection table

Source object	*Parameter values*	*Result object*
BTree.cedar!(Sept 9, 1982, 13:52:55)	[*Ascii.binary*!23ACD904EFFA]	*BTree.binary*!43956A3C32F0
BTreeImpl.cedar!(Jan 14, 1983, 14:44:09)	[*Rope.binary*!AC9023E76FA6, *IO.binary*!23843396A24f, *Space.binary*!8348823F1761, *BTree.binary*!43956A3C32F0]	*BTreeImpl.binary*!2045FFD283C

Example 2: Object type and projection tables

4.3 Compiled models

The modeller keeps its caches on each machine. It is also desirable to include this kind of precomputed information with a stored model, since a model is often moved from one machine to another, and some models are shared among many users, who refer to them in their own models by using the @-notation. An example is the model *CedarInterfaces.model*, which returns a large number of commonly-used interfaces that a Cedar program might need. Furthermore, even with the caches it is still quite expensive to do all the typechecking for a sizable model.

For these reasons, the Modeller has the ability to create and read back compiled models. A compiled model contains

a tree which represents a parsed and typechecked version of the model;

object type and projection tables with entries for all the objects in the model;

a version map (see § 5) with entries for all the objects in the model.

When the user pushes the *MakeModelBinary* button in Figure 1, the Modeller makes this binary object for the current model, much as a compiler makes a binary file from a source file. In a *.modelBinary* object any parameters of the model which are not instances may be given specific argument values. This is much like the binary objects produced by the compiler, in which the interface parameters are fixed. The *.modelBinary* object acts merely as an accelerator, since it is always possible to work from the sources of the model and the objects it references, to derive the same result as is encoded in the *.modelBinary*.

4.4 Releases

The *Release* operation described in § 3.3 is implemented in three phases.

Phase one: Check

The *Check* phase of *Release* checks the *Top* model and all its sub-models for problems that might prevent a successful release. Each model is parsed and all files listed in the model are checked. *Check* ensures that the versions listed in the models exist and that their parameterization is correct. The directory containing each source file is checked to make sure it contains a valid object file. This guards against compilation errors in the source files. Common blunders are caught, such as a reference to a model that is not in the *Top* model. The Release Master contacts implementors and asks them to fix any errors caught in this phase.

Phase two: Move

The *Move* phase moves the files of the release onto the release directory and makes new versions of the models that refer to files on the release directory instead of the working directory. For each model listed in the release position list, *Move*:

1) reads in the model from the working directory,
2) moves each file explicitly mentioned in the model to its release position,
3) writes a new version of the source file for the model in the release directory.

This release version of the model is like the working version except that

a) all working directory paths are replaced by paths on the release directory,
b) a comment is added recording the working directory that contained the working version of the model, and
c) the LET statement referring to the *Top* model is changed to refer to the one on the release directory.

Phase three: Build

The *Build* phase takes the new model computed during the *Move* phase and uses it to traverse all the objects in the release. For each model:

1) All models on incoming edges must have been examined.
2) For every source file in the model, its object file (known to exist from the *Check* phase) is moved from the working directory to the release directory.
3) A *.modelBinary* file is made for the version of the model on the release directory.
4) If a special comment in the model is given, a runnable object file is produced for the model.

After this is done for every model, a version map of the entire release is stored on the release directory.

At the conclusion of phases *Check*, *Move* and *Build*, *Release* has established that:

1) *Check*: All reachable source objects exist, and derived objects for all but the *Top* object exist. This means the files input to the release are statically correct.
2.) *Move*: All objects are on the release directory. All references to files in these models are by explicit create time (for source files) or version stamps (for object files).
3) *Build*: The system has been built and is ready for execution. All desired accelerators are made (*.modelBinary* files and a version map for the entire release).

5. Files

A model refers to objects by their unique names. Given the conventions for objects and their names described in § 2, this is unambiguous. In order to build a system from a model, however, the Modeller must obtain the representations of the objects. Since objects are represented by files, the Modeller must be able to deal with files. There are two aspects to this:

1) Locating the file which represents an object, starting from the object's name.
2) Deciding where in the file system a file should reside, and when it is no longer needed and can be deleted.

5.1 Locating files

It would be nice if an object name could simply be used as a file system name. Unfortunately, file systems do not provide the properties of uniqueness and immutability that object names and objects must have. Furthermore, most file systems, including ours, require a file name to include information about the machine that physically stores the file. Hence a mapping is required from object names to the full path names that unambiguously locate files in the file system.

To locate a file, the Modeller uses a location hint in the model. The object reference @[Ivy]<Schmidt>*BTreeImpl.cedar*!(Jan 14, 1983, 14:44:09) contains such a hint, [Ivy]<Schmidt>. To find the file, the Modeller looks on the file server *Ivy* in the directory *Schmidt* for a file named *BTreeImpl.cedar*. There may be one or more versions of this file; they are enumerated, looking for one with a creation date of *Jan 14, 1983, 14:44:09*. If such a file is found, it must be the representation of this object. Otherwise this method of finding the representation has failed.

The distributed environment introduces two types of delays in access to objects represented by files:

1) If the file is on a remote machine, it has to be found.
2) Once found, it has to be retrieved.

Since retrieval time is determined by the speed of file transfer across the network and the load on the file server, the Modeller tries to avoid retrieving files when the information it wants about a file can be computed once and stored in a database. For example, the type of an object, which is the information needed to compute its compilation dependencies, is small compared to the object itself. The object type table stores the types of all objects of current interest; a source object in the table does not have to be examined, or even retrieved, unless it actually needs to be recompiled.

In cases where the file must be retrieved, determining which machine and directory has a copy of the version desired can be very time-consuming. Even when a file location hint is present and correct, it may still be necessary to examine several versions of the file to find the one with the right creation date. The Modeller minimizes these problems by keeping another cache, which maps an object name into the full path name in the distributed file system of a file which represents the object. This cache is a table called the *Version Map*, illustrated in Example 3. Note that both source objects, whose unique identifiers are creation dates, and binary objects, whose unique identifiers are version stamps, appear in the version map. The full path name includes the version number of the file (the number after the !). This version number makes the file name unique in the file system, so that a single reference is sufficient to obtain the file.

Thus the Modeller's strategy for minimizing the cost of referencing objects has three parts:

1) Consult the object type table or the projection table, in the hope that the information needed about the object is recorded there. If it is, the object need not be referenced at all.
2) Next, consult the version map. If the object is there, a single reference to the file system is usually sufficient to obtain it.
3) If there is no entry for the object in the version map, or if there is an entry but the file it mentions doesn't

Version map

Object name	*File location*
BTree.cedar!(Sept 9, 1982, 13:52:55)	[Ivy]<Schmidt>BTree.cedar4
BTreeImpl.cedar!(Jan 14, 1983, 14:44:09)	[Ivy]<Schmidt>BTreeImpl.cedar!9
BTree.binary!43956A3C32F0	[Ivy]<Schmidt>BTree.binary!2
BTreeImpl.binary!2045FFD283C	[Ivy]<Schmidt>BTreeImpl.binary!5
Ascii.binary!23ACD904EFFA	[Indigo]<Cedar>Ascii.binary23

Example 3: Version map

exist, or doesn't actually represent the object, then use the file location hint to identify a directory, and enumerate all the versions of the file to find one which does represent the object. If this search is successful, make a new entry in the version map so that the search need not be repeated.

Like the other caches, a version map is maintained on each machine and in each *.modelBinary* object. A *.modelBinary* version map has an entry for each object mentioned in the model. A machine version map has an entry for each object which has been referenced recently on that machine. In addition, commonly-referenced objects of the Cedar system are added to the machine version map as part of each Cedar release.

Since the version maps are hints, a version map entry for an object does not guarantee that the file is actually present on the file server, Therefore each successful probe to the version map delays the discovery of a missing file. For example, the fact that a source file does not exist may not be discovered until the compilation phase, when the Modeller tries to compile it. This means that the Modeller must be robust in the face of such errors. The release process, however, guarantees that the files are present as long as the release remains active.

5.2 Moving files

The Modeller was originally implemented before the Cedar facilities which provide network-wide naming for files. Consequently, it includes procedures for transferring files automatically between a remote machine and the local machine on which compilations are done and programs are loaded. In addition, the Modeller records (in the local machine's file type table) whether a file has been edited and not saved on a remote file server. A *StoreBack* button stores all such changed files on the proper remote servers. Thus the Modeller can provide a substitute for uniform access to files throughout a network, at least for files which represent objects in a model.

6. Status and plans

The system described above is not yet integrated into Cedar; we expect to complete it over the next year. Although some users are using the Modeller to compile their systems, most use manual techniques. The existing Modeller has been used by five or six programmers over the last year; two have used it heavily. The debugger interface, *.modelBinary* files, and *Release* are not yet implemented.

Databases are naturally suited to storing modules and dependency relationships between modules, as well as object types, projections, and version maps. When this research was started there was no database system in Cedar that could handle the amount of data involved. We envision many programs that process data about modules in systems, such as sophisticated browsers and cross-reference tools like Interlisp's MasterScope [9] or PIE's Browser [4]. Query facilities of the database system could easily be used to answer questions about programs that require specialized analysis programs if no database is used, such as "Who depends on module *X*?"

Acknowledgements

System modelling began with ideas developed jointly with Charles Simonyi, and worked out in detail by Rich Johnsson and John Wick in the Mesa Binder. They evolved further in a working group which included Bob Ayers, Phil Karlton, Tom Malloy, Ed Satterthwaite and John Wick, and in later discussions with Andrew Birrell and Jim Horning. *Release* was originally Roy Levin's idea. Ed Satterthwaite is the implementor of the SML evaluator.

References

[1] Cristofor, E., *et al.*, Source control + tools = stable systems. *Proc. 4th Computer Software and Applications Conf.*, Oct. 1980, 527-532.

[2] Deutsch, L.P., and Taft, E.A. (eds.), *Requirements for an Experimental Programming Environment.* CSL-80-10, Xerox PARC, 1980.

[3] Feldman, S.I. Make – A program for maintaining computer programs. *Software – Practice and Experience*, **9**, 4, April 1979.

[4] Goldstein, I.P., and Bobrow, D.G. A layered approach to software design. CSL-80-5, Xerox PARC, 1980..

[5] Habermann, A.N., *et al.*, *The Second Compendium of Gandalf Documentation.* Computer Science Dept., CMU, May 1982.

[6] Lampson, B.W. and Schmidt, E.E. Practical use of a polymorphic applicative language. *Proc. 10th Symp. Principles of Programming Languages*, Austin, Texas, Jan. 1983.

[7] Mitchell, J.G. *et. al. Mesa Language Manual.* CSL-79-3, Xerox PARC, May 1981.

[8] Schmidt, E., *Controlling Large Software Development in a Distributed Environment.* PhD Thesis, EECS Dept., Univ. of Calif. Berkeley, Dec. 82 and CSL-82-7, Xerox PARC, Dec. 1982.

[9] Teitelman, W. and Masinter, L. The Interlisp programming environment. *Computer* **14**, 4, April 1981, 25-34.

Appendix: A real example

This model describes the *BringOver* program, which is a substantial component in the Cedar system. We present the model with its environment aggregated into separate models, and with defaults for all the parameters.

There are seven implementation modules within this model (*CWFImpl*, *ComParseImpl*, *SubrImpl*, *STPSubrImpl*, *DFSubrImpl*, *DFParserImpl*, *BringOverImpl*). All the rest are interfaces.

First we define the two environment models. One is a big binding for the system interfaces on which *BringOver* and many other parts of Cedar depend. The other is a declaration for the instances of these interfaces. This declaration is rather repetitive, but it is needed to provide the proper names for defaulting the instance arguments of the *BringOver* models. It might be possible to avoid this declaration by passing the entire interface binding, and a corresponding binding for the instances, as two big arguments to the client modules. Cedar currently does not permit this, however, and we do not show it here.

System.model

```
[ Ascii ~ @Ascii.cedar*[];
  CIFS ~ @CIFS.cedar*[];
  ConvertUnsafe ~ @ConvertUnsafe.cedar*[];
  Date ~ @Date.cedar*[];
  DCSFileTypes ~ @DCSFileTypes.cedar*[];
  Directory ~ @Directory.cedar*[];
  Environment ~ @Environment.cedar*[];
  Exec ~ @Exec.cedar*[];
  File ~ @File.cedar*[];
  FileStream ~ @FileStream.cedar*[];
  Heap ~ @Heap.cedar*[];
  Inline ~ @Inline.cedar*[];
  KernelFile ~ @KernelFile.cedar*[];
  LongString ~ @LongString.cedar*[];
  NameAndPasswordOps ~ @NameAndPasswordOps.cedar*[];
  Process ~ @Process.cedar*[];
  Rope ~ @Rope.cedar*[];
  RopeInline ~ @RopeInline.cedar*[];
  Runtime ~ @Runtime.cedar*[];
  Segments ~ @Segments.cedar*[];
  Space ~ @Space.cedar*[];
  Storage ~ @Storage.cedar*[];
  STP ~ @STP.cedar*[];
  STPOps ~ @STPOps.cedar*[];
  Stream ~ @Stream.cedar*[];
  String ~ @String.cedar*[];
  System ~ @System.cedar*[];
  SystemInternal ~ @SystemInternal.cedar*[];
  Time ~ @Time.cedar*[];
  Transaction ~ @Transaction.cedar*[];
  TTY ~ @TTY.cedar*[];
  UserTerminal ~ @UserTerminal.cedar*[] ]
```

SystemInstancesDecl.model

```
LET @System.model IN
  [ CIFSImpl: CIFS,
    ConvertUnsafeImpl: ConvertUnsafe,
    -- 23 declarations are omitted for brevity--
    TTYImpl: TTY,
    UserTerminalImpl: UserTerminal ]
```

We also need instances for the system interfaces. We can get them from the following model; its type is *@SystemInstancesDecl.model*.

SystemInstances.model

```
LET Interfaces~@System.model IN
  [CIFSImpl: CIFS ~ CIFSImpl.cedar*[],
   ConvertUnsafeImpl: ConvertUnsafe ~ ConvertUnsafeImpl.cedar*[],
   DateImpl: Date ~ DateImpl.cedar*[],
   -- 23 bindings are omitted for brevity--
   TTYImpl: TTY ~ TTYImpl.cedar*[],
   UserTerminalImpl: UserTerminal ~ UserTerminalImpl.cedar*[] ]
```

The models above are part of the working environment of a Cedar programmer; they are constructed once, as part of building the Cedar system.

Now we can write the model for *BringOver*. It picks up *System.model* and *SystemInstancesDecl.model*, and then gives a single binding of *BringOverProc* to a function which takes the instances as an argument, and returns two interfaces and an instance of each. The body of the function has

one LET to make all the system interface and instance names directly accessible for defaulting;

a second LET to bind all the internal interfaces and instances of *BringOver*;

a binding to construct the two interfaces and two instances which are the result of applying *BringOverProc*.

BringOver.model

```
LET [Interfaces~@System.model,
 InstancesDecl~@SystemInstancesDecl.model IN
  [BringOverProc ~ λ [Instances: InstancesDecl]⇒
   [BringOver: INTERFACE, BringOverImpl: BringOver,
    BringOverCall: INTERFACE, BringOverCallImpl: BringOverCall] IN
     --Make the system interface and instance names accessible
     LET Interfaces+Instances IN
     LET [ -- These are the internal interfaces and instances
      CWF ~ @CWF.cedar*[];;
      CWFImpl ~ @CWFImpl..cedar*[];;
      ComParse ~ @ComParse.cedar*[];
      ComParseImpl ~ @ComParseImpl.cedar*[];
      Subr ~ @Subr.cedar*[];
      SubrImpl ~ @SubrImpl.cedar*[];
      STPSubr ~ @STPSubr.cedar*[];
      STPSubrImpl ~ @STPSubrImpl.cedar*[];
      DFSubr ~ @DFSubr.cedar*[];
      DFUser ~ @DFUser.cedar*[];
      DFSubrImplA ~ @DFSubrImpl.cedar*[];
      DFSubrImplB ~ @DFParserImpl.cedar*[];
      DFSubrImpl ~ DFSubrImpl+DFSubrImplB ]
     IN [ -- These are the exported interfaces and instances
      BringOver~ @BringOver.cedar*[];
      BringOverCall ~ @BringOverCall.cedar*[];
      [BringOverImpl: BringOver, BringOverCallImpl: BringOverCall] ~
          @BringOverImpl.cedar*[] ] ]
```

Using *BringOverProc*, we can compute the exported interfaces and instances of *BringOver*.

```
[BringOver, BringOverImpl, BringOverCall, BringOverCallImpl] ~
 BringOverProc[@SystemInstances.model]
```