Software Interconnection Models

Dewayne E. Perry
AT&T Bell Laboratories
Murray Hill, NJ 07974
908.582.2529

published in
The 9th International Conference on Software Engineering
May 1987, Monterey CA

Abstract

We present a formulation of interconnection models and present the unit and syntactic models — the primary models used for managing the evolution of large software systems. We discuss various tools that use these models and evaluate how well these models support the management of system evolution. We then introduce the semantic interconnection model. The semantic interconnection model incorporates the advantages of the unit and syntactic interconnection models and provides extremely useful extensions to them. By refining the grain of interconnections to the level of semantics (that is, to the predicates that define aspects of behavior) we provide tools that are better suited to manage the details of evolution in software systems and that provide a better understanding of the implications of changes. We do this by using the semantic interconnection model to formalize the semantics of program construction, the semantics of changes, and the semantics of version equivalence and compatibility. Thus, with this formalization, we provide tools that are knowledgeable about the process of system construction and evolution and that work in symbiosis with the system builders to construct and evolve large software systems.

1. Introduction

Progress in the management of software evolution has been very slow. While we now have a growing emphasis on programming environments and integrated sets of tools for the software production process, the underlying models used in managing the evolution of software have not changed significantly in the past 20 years. Where formerly we managed the process of software evolution informally, we now have tools to help us with these problems. However, the models we use have changed very little.

One of the primary models that we use to manage the evolution of software systems is an *Interconnection Model (IM)*. An interconnection model is a tuple that consists of two sets: a set of objects that are the components of interconnections and a set of relations that define the kinds of interconnections that exists among the objects in the model.

$$IM = (\{ Objects \} , \{ Relations \})$$

The interconnection models are typically used to construct graph structures in which the nodes are objects in the model and the arcs are the relations. Various sorts of computations are then performed on the graph.

Presently, we use two interconnection models in managing the evolution of software: the unit interconnection model and the syntactic interconnection model. We present each of these models and describe various uses that are made of these models to aid the evolution process. It should be noted that the syntactic interconnection model may either implicitly or explicitly include the unit interconnection model. We then introduce a new model, the semantic interconnection model, and indicate how it provides a finer grain of interconnections than the first two models. We show that, because of the nature of these semantic interconnections and their finer granularity, the semantic interconnection model provides a better model for managing the evolution of software than either the unit or syntactic interconnection models. Please note that, as the syntactic model may include the unit model, the semantic model may include both the syntactic and unit models.

2. Unit Interconnection Model

The unit interconnection model defines relationships between various *units* — typically, either files or modules — that comprise a software system. The basic relationship is one of dependency — for example, unit *a* depends upon units *b*, *c* and *d*. Thus, in generic terms, a unit interconnection model has as its set of objects a single type of object, a *unit*, and has as its set of relations a single relation *depends on*.

$$Unit\ IM = (\{ units \} , \{ "depends\ on" \})$$

We shall see that this model varies slightly with each individual use.

This model is useful because it supports and encourages modular construction of software. It captures both the notion of encapsulation (that is, breaking a program into pieces that isolate different features of the system) and the notion of a single copy of each unit in order to localize the problems of maintenance and evolution.

Among the uses of this model are the following: determining compilation contexts, determining recompilation strategies, change notification, and system modeling. We discuss each of these uses and describe the particular instantiation of the unit interconnection model.

Two examples of using this model for determining the context of a compilation are the C preprocessor [23] and Ada [1]. In C, the context of a compilation is defined by a set of **#include** directives. The C preprocessor then establishes the compilation context by physically including the mentioned files in order to compile correctly the unit (in this case, a file). The model used by the C preprocessor is

IM = ({ files } , { "includes" }).

In Ada, the context is defined by the **with** clause. The Ada language system then establishes the context for a compilation unit (which in this case may be a package, a subprogram, etc.) by incorporating the symbol table information derived from the specification of the named compilation unit into the symbol table for the unit being compiled. The model used for the Ada language system is

IM = ({ compilation-units } , { "with" })

Recompilation strategies make use of a slightly richer form of the unit interconnection model. Ada, for example, uses this information in order to determine compilation dependencies and adds the relation of *changed more recently than* to the model in order to determine its recompilation strategy. Thus, its model for this purpose is

IM = ({ compilation-units }, { "with", "changed more recently than" }).

Make [4] incorporates the explicit definition of dependencies, the relationship *changed since derived object generation*, and the knowledge of how to generate particular derived objects from those they depend on in order to determine recompilation and regeneration strategies. For example, *.o* files are derived by applying *cc* to *.c* files and *.c* files may be derived from *.y* files by applying *yacc*. Thus, Make's unit interconnection model is

IM = ({ .o files, .c files, .y files, ... } , { "depends on", "changed since generation" }).

The **#include** dependencies used for defining the context of a compilation can also be used to help with the problems of making changes consistently throughout a system. For example, an extension to the Software Change Control System (SCCS) [23] uses these dependencies to notify the programmers who are responsible for units that depend on units which have been changed that their unit may be affected by these changes. Whereas SCCS's facilities are static and provided only on demand, DOMAIN Software Engineering Environment (DSEE) [14] provides change notification facilities that are dynamic — monitors are placed on files that are triggered when changes are made to the monitored file. Notices are then sent to the agents responsible for the monitors.

The composition of systems is described by a system model — a description of the components that comprise the system. Cedar's System Modeler [13], DSEE, some versions of SCCS, and RCS [27] provide

system modeling on the basis of files. This information is then combined with facilities for recompilation to provide facilities for system regeneration. The interconnection model for these kinds of system modeling is

IM = ({ systems, files } , { "is composed with" }).

Some of these systems provide facilities for composing subsystems from files and then systems from subsystems. In this latter case, we merely add the appropriate objects to the model.

While the model is useful in supporting encapsulation and modularity, determining compilation contexts, determining useful recompilation strategies, providing change notification, and describing system models, it is useful only with very large-grained objects. Remember that the basic units in this model are either files or packages which usually contain multiple smaller objects such as types, constants, data and functions. Often only a small amount of a context provided in this manner is actually used. More than is really necessary is often recompiled because of the size of the granularity involved. Since only a small amount of a context may be actually used, change notifications may also be sent on too broad a basis as well. Finally, we may want to compose our systems out of smaller pieces than files, modules or packages. Thus, a finer grain of interconnection than is supplied by the unit interconnection model is needed for effective management of evolution in software systems.

3. Syntactic Interconnection Model

The syntactic interconnection model has an advantage over the unit interconnection model in that it focuses on smaller units: it describes the relations among the syntactic elements of programming languages. Thus, in generic terms, a syntactic interconnection model has in its set of objects procedures, functions, types, variables, etc. (the exact list depends upon the objects defined in the particular programming language — for example, some languages support modules, others do not) and has as its set of relations such relations as *calls, is called by, is the parameter of, is an argument of, is used at, is set at,* etc. (again, the exact set of relations that is possible is language dependent).

Syntactic IM = (
{ functions, procedures, types, variables, ... } ,
{ "is used at", "is set at", "calls", "is called by", ... }
)

Again, we shall see that this model varies slightly with each individual use.

This model is useful because it localizes the interconnections to those among objects used in writing the software: types, variables, procedures, parameters and arguments, etc. Further, it captures the basic objects of evolution, the objects of the language in which the programming is done. Where the unit interconnection model only indicates the general location of changes, the syntactic interconnection model indicates the syntactic objects that are changed — a finer grain of locality.

Among the uses of this model are the following: change management, static analysis, smart recompilation, and system modeling. We discuss each of these uses and describe the particular instantiation of the syntactic interconnection model.

The classic manifestation of the syntactic interconnection model is the cross-reference, or set/used, listing (often supplied as a side-effect of compilation). The cross-reference listing is a simplified instantiation of the syntactic interconnection model and has been used as the primary tool for managing the change process for the past 25 years. By consulting a cross-reference listing, programmers can systematically locate the exact places affected by changes and can propagate the effects of these changes manually. The model used by the cross-reference listing combines the syntactic objects of the language with the notion of a location (often the line number in the program listing) to define the relations of definition, use and assignment.

IM = (
{ functions, procedures, types, variables, ..., locations } ,
{ "is defined at", "is set at", "is used at" }
)

This kind of change management information can be automated with the appropriate database built from the system on the basis of this interconnection model. Simple kinds of automation are supplied by existing tools such as compilers. For example, because changes are made to syntactic units, compilers can often be used to detect the effects of certain kinds of changes such as deletions of identifiers and modifications of spelling. Interlisp's Masterscope [26], Cscope [25] and Smile [12, 17] provide a more sophisticated sort of automation, though in differing ways — Masterscope builds a database of relationships and provides change management by querying the database; Smile provides incremental checking as changes are made. An interesting knowledge-based approach is described in [2]. All of these tools use a version of the syntactic interconnection model that is close to the generic model.

Static analysis of programs, such as that found in the semantic checking of compilers and tools like Lint [16], use this model to build the relationships between objects in a program. While most compilers use the generic form of the interconnection model in order to provide there semantic analysis, Lint requires a richer model in order to determine such things as unreachable code, unset variables, etc.

Tichy [28] uses the syntactic model to determine the effects of changes within a module and the necessity of recompilation. As in the unit model, the relations of change, addition and deletion are added to the relations of the syntactic model and used to determine whether a particular change necessitates recompilation. For example, a change within a procedure only requires the recompilation of that procedure; a change to the parameter list of a procedure might necessitate the recompilation of all of its uses, depending upon the nature of the change. The model used by Tichy's smart recompilation is

IM = (
{ functions, procedures, types, variables, ... } ,
{ "is used at", ..., "is changed to", "is deleted from", "is added to" }
).

System modeling, such as that found in Gandalf's SVCE [8, 11], indicates the objects that are needed to compose a particular version of a system or system component. Thus, elements of a system can be composed of separate parts that are of a smaller grain than in the previous model. Two further advantages result: better composition consistency checking and smarter system generation. First, the system model can be married with static analysis to determine the consistency of the components in the system model. Second, because the objects are smaller than the units in the previous model, the process of generating new systems — that is, marrying system modeling with smart recompilation — can be managed with a finer degree of control than in the unit model. System modeling, then, requires a slightly richer model than the generic syntactic interconnection model.

IM = (
{ systems, system-components, functions, procedures, types, variables, ... } ,
{ "is used at", "is set at", "calls", ..., "is composed with" }
)

In all these manifestations of the syntactic interconnection model, we have a finer degree of interconnections and a richer set of relationships than in the unit interconnection model. We have explicit interconnections among the basic objects of program construction. However, we have no notion of *why* these interconnections exist; there is no indication how the objects were intended to be used, nor why the objects were in fact used.

4. Semantic Interconnection Model

While we can derive various kinds of facts about a system from its syntactic interconnections, we gain no information about the semantics of the system, and certainly no information about the intentions of the numerous programmers in building a large system the way it was built. Granted, there are certain assertions that we can derive about data from strong typing, but that is a relatively small amount of information when considered as part of the whole system and as part of what is needed in order to provide effective management of system evolution.

It is this lack of semantic information about the components that are used in building large software systems that leads to the introduction of the semantic interconnection model. We need a way to express how objects that comprise a system are *meant* to be used — that is, what the designers and builders had in mind when the objects were created — and to capture *how* these objects are, in fact, used — what the builders had in mind when they used these objects to build the system.

We found the technology of formal specifications to be a useful source of information about how objects are meant to be used. Algebraic specification approaches (such as OBJ [6] and Larch [7]) and input/output predicate approaches (such as Hoare [10], Alphard [29], Dijkstra [3], and Anna [15]) represent some of the ways in which a system builder might describe the semantics of system objects. Algebraic axioms are particularly apt for describing the relationships between operations and for indicating how these operations are meant to be used. Input/output predicates, while perhaps less elegant for this purpose, are more suggestive as a means of describing how system objects are actually used. In particular, Hoare's input/output predicates [10] viewed as points of interconnections, rather than as elements to be used in proving properties about a program fragment, are particularly suggestive.

It is the thesis of the Inscape Environment research project [18, 19] that this view of input/output predicates provides an extremely fruitful basis upon which to build tools for managing the construction and evolution of large software systems. In the following subsections, we discuss how predicates can be used as points of interconnection and define the semantic interconnection model. We then describe how this model can be used to provide tools to manage the evolution of large systems. In particular, we discuss how this model is used in system construction, in system modification, and in version control.

4.1 Predicates and Interconnections

In the basic input/output predicate approach, the input predicates define the assumptions, or preconditions, that must be satisfied if a sequence of code is to execute successfully. The output predicates define the results, or postconditions, that are guaranteed to be true if the input predicates are satisfied. Each of these predicates, whether they are preconditions or postconditions, represents a fact about the behavior of either the system or a system component for example, *Allocated(*Bufptr)* indicates that a buffer has been allocated as the reference of Bufptr and $0 \leq L \leq MaxLength$ indicates that L has a value between zero and the maximum possible length. Predicates provide the basic vocabulary with which to describe the behavior required and produced by a component in a system. If we view a component of a system and its preconditions and postconditions as analogous to a hardware chip with its input and output pins, we have a metaphor for semantic interconnections. Postconditions are facts that can be used to satisfy our assumptions, i.e., our preconditions. At this point we will be somewhat vague about the term *satisfies* and refine it in the subsequent discussion.

Thus, we extend the syntactic interconnection model to get the semantic interconnection model by adding predicates to the set of objects and the relation *satisfies* to the set of relations.

Semantic IM = *(*
{ functions, procedures, types, variables, ..., predicates } ,
{ "is used at", "is set at", "calls", "is called by", ..., "satisfies" }
)

4.2 Predicates and Program Construction

Whereas formal specification projects in general emphasize the formal properties of specifications and the problems of verification (as for example in the OBJ and Larch projects), the Inscape Environment emphasizes the practical and constructive use of interface specifications. The interface specifications are used as a means of practical program construction, not verification. In order to do this, we extend Hoare's approach in two ways. The first extension is the manner in which the result behavior of an operation is specified. In addition to postconditions, which describe what is known to be true as a result of an operation's execution, there are obligations, which describe conditions that must eventually be satisfied. Obligations are incurred as side-effects of operations — for example, an obligation to deallocate a buffer or close a file may be incurred when allocating a buffer or opening a file. The second extension reflects the fact that exception conditions are a part of the result behavior of an operation. In robust, fault-tolerant software, there are a number of different results depending upon whether or not some of the preconditions are satisfied. It is not always the case that unpredictable behavior results from the falsification of preconditions. In many cases, the resulting behavior is predictable but with only partial results. It is important that these partial results be specified as well. Thus, we provide an extension to the Hoare specification paradigm in which there are multiple sets of results specifications (of postconditions and obligations).

With these additions, we have extended the semantic interconnection model to include objects such as obligations and exceptions and additional relations between these objects. For example, the following are some of relationships that exist among the objects.

- Preconditions and obligations *depend on* postconditions or are *propagated* to the encompassing interface (where eventually they will be satisfied by some postconditions).
- Postconditions *satisfy* preconditions and obligations and may be *propagated* to the encompassing interface (as they are appropriate to the supported abstraction).
- The failure of some preconditions can be *handled* by exception handlers in such a way as to recover from the precondition failure. Note that this is not the only way in which exceptions may be handled (for a discussion of the varieties of exception handling, see [20]).
- Some exceptions can be *precluded* by the *satisfaction* of their related preconditions.

Thus, the semantic interconnection model that we use in Inscape is

IM = ({ ..., preconditions, obligations, postconditions, exceptions } , { ..., "depends on", "satisfies", "propagated", "precludes", "handles", ... }).

The Inscape Environment [19] provides a module interface specification language, *Instress*, in which to describe the properties of and constraints on data and the behavior of operations. The *Inform* program

construction component of the environment uses these interface specifications in symbiosis with the programmer to construct components of the system. As a component is constructed, the environment records these relationships, enforces the consistent use of the interface specifications, and enforces the semantics of program construction with respect to these interfaces.

We present the example below to illustrate how Inscape uses semantic interconnections in the construction of a simple operation and to show the relationship between the implementation of the operation and its automatically propagated interface. In this example, we implement a procedure *ObtainRecord* in the context of a file system of distinctly named files, where each file consists of a set of records denoted by record numbers. *ObtainRecord* obtains a designated record from the specified file. Note that in this example we treat *only* the successful case (for the sake of simplicity); we do not present any of the exceptional conditions that *would* be present in a normal specification and implementation).

The operation *ReadRecord* (figure 1)comes close to providing the functionality that we desire for the target operation *ObtainRecord*. Note, however, that there are a number of preconditions that must be satisfied before it can read the record successfully.

Preconditions :	ValidFilePtr (FP)	R1
	FileOpen (FP)	R2
	LegalRecordNr (R)	R3
	RecordExists (R)	R4
	RecordReadable (R)	R5
	RecordConsistent (R)	R6
ReadRecord (FP, R, &L, &Bufptr)		
Postconditions :	ValidFilePtr (FP)	R7
	FileOpen (FP)	R8
	LegalRecordNr (R)	R9
	RecordExists (R)	R10
	Was (RecordReadable (R))	R11
	Was (RecordConsistent (R))	R12
	Allocated (*Bufptr)	R13
	0 <= L <= Allocated (*Bufptr)	R14
	RecordIn (*Bufptr)	R15
Obligations :	Deallocated (*Bufptr)	R16

Figure 1

Two of these preconditions can be satisfied by the operation *OpenFile* (shown in figure 2): postconditions O5 and O6 satisfy preconditions R1 and R2. Note that in the process of using *OpenFile*, we have incurred an obligation to close the file.

Preconditions :	LegalFileName (F)	O1
	FileExists (F)	O2
OpenFile (F, &FP)		
Postconditions :	LegalFileName (F)	O3
	FileExists (F)	O4
	ValidFilePtr (FP)	O5
	FileOpen (FP)	O6
Obligations :	FileClosed(FP)	O7

Figure 2

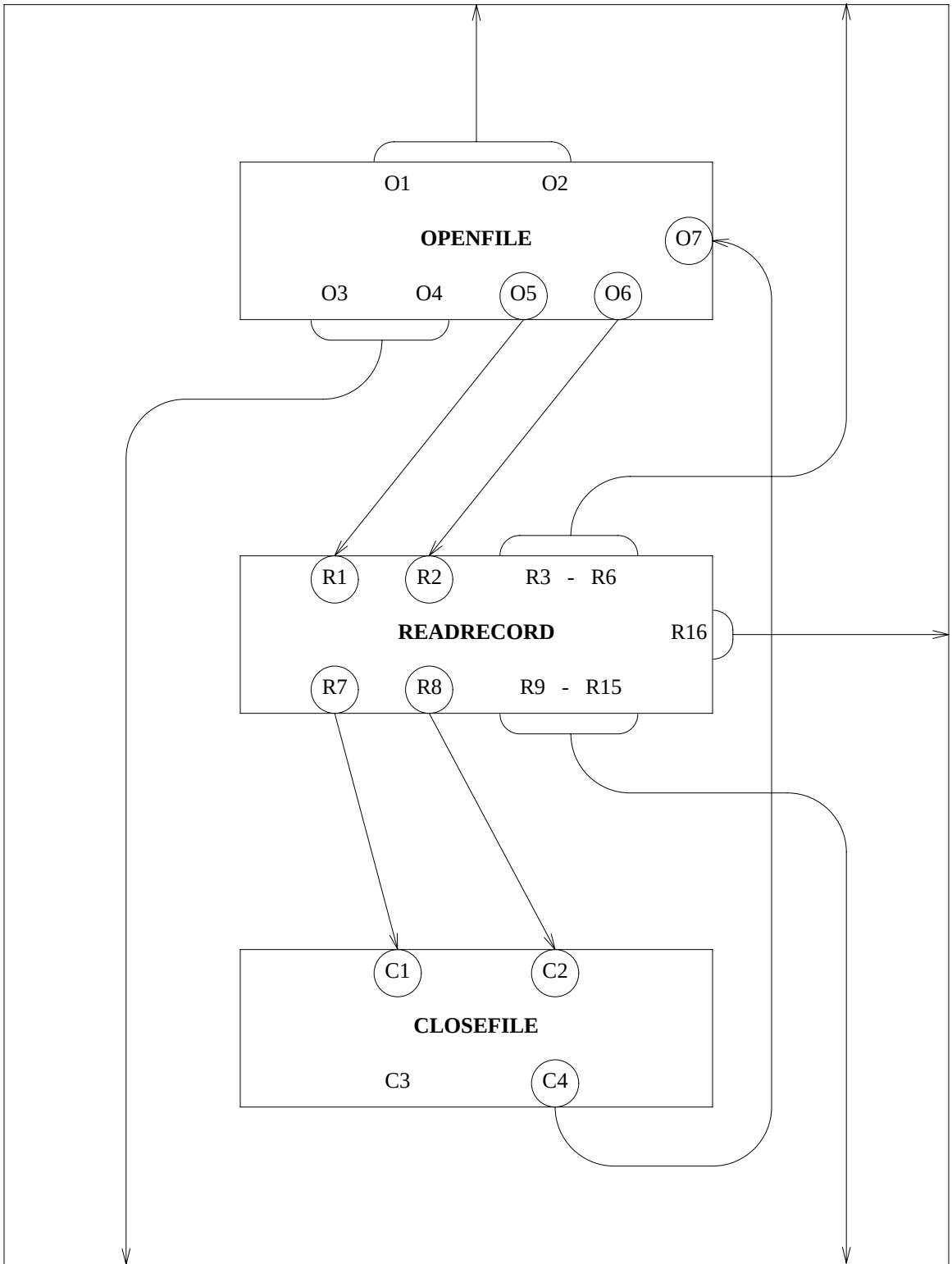
This obligation can be satisfied by the operation *CloseFile* by one of its postconditions (namely C4) as illustrated in figure 3. The preconditions of *CloseFile*, C1 and C2, can be satisfied by the postconditions, R7 and R8, of *ReadRecord.*

Preconditions :	ValidFilePtr (FP)	C1
	FileOpen (FP)	C2
CloseFile (FP)		
Postconditions :	Not (ValidFilePtr (outvalue (FP)))	C3
	FileClosed (invalue (FP))	C4
Obligations :	<none>	

Figure 3

Figure 4 illustrates graphically the implementation of *ObtainRecord* as the sequence of three operation calls: *OpenFile, ReadRecord*, and *CloseFile*. The abbreviations at the top of each box represent each operations preconditions; the abbreviations at the bottom, postconditions; and the abbreviations at the right

side, obligations. Note that not all of the preconditions of *OpenFile* or *ReadRecord* have been satisfied within the implementation. (Those that have been satisfied are depicted as circled abbreviations connected by arrows.) These preconditions must be satisfied somewhere, and since they have not been satisfied within the implementation, they must be satisfied outside the implementation and, hence, are exported to the enclosing operation's interface: O1 and O2; R3 through R6.



Some of the postconditions known to be true after the execution of *CloseFile* are not exported to the encompassing interface because they are not appropriate to the abstraction presented by *ObtainRecord* — for example C3 and C4. The notion of files being opened or closed and the concept of file pointers are not needed by the user of the implemented operation. However, the facts known about the file and specific record are of importance and are thus propagated to the enclosing interface (cf. postconditions O3 and O4, and R9 through R15. Finally, we must propagate the unsatisfied obligation of *ReadRecord*, R16, to the interface of *ObtainRecord*.

Preconditions :	LegalFileName (F)	O1
	FileExists (F)	O2
	LegalRecordNr (R)	R3
	RecordExists (R)	R4
	RecordReadable (R)	R5
	RecordConsistent (R)	R6
ObtainRecord (FP, R, &L, &Bufptr)		
Postconditions :	LegalFileName (F)	O3
	FileExists (F)	O4
	LegalRecordNr (R)	R9
	RecordExists (R)	R10
	Was (RecordReadable (R))	R11
	Was (RecordConsistent (R))	R12
	Allocated(*Bufptr)	R13
	0 <= L <= Allocated (*Bufptr)	R14
	RecordIn (*Bufptr)	R15
Obligations :	Deallocated (*Bufptr)	R16

Figure 5

Figure 5 illustrates *Obtain Record*'s interface as generated from its implementation.

4.3 Predicates and Program Evolution

The details of interconnections recorded while constructing the system form the basis for *Infuse* [21] — the change management component of Inscape — which has the ability to determine the implications and extent of changes. When changes occur, either to interfaces or to implementations, the effects of these changes can be detected by determining the preconditions, postconditions, obligations, and exceptions that are affected by the change and by tracing (recursively) how these modifications affect the components that use them. For example, if predicates are removed from an interface specification, the environment can

determine how this removal affects the components where the changed component is used. Some of the relationships that existed previously may no longer exist — that is, some of the interconnections may no longer be made. The following kinds of situations can be determined by the environment with respect to changes to interface specifications.

- If a precondition or obligation is deleted, or a postcondition is added, it can be determined whether some sequence of code is redundant.
- If a precondition or obligation is added, or a postcondition is deleted, it can be determined whether some sequence of code must be added in order to satisfy the added predicate or supplant the delete one.
- If predicates are added or deleted, it can be determined whether the changes have no effect at all on the implementation.

Conversely, if changes are made to the implementation, it can be determined what effects these changes have on the implementation and on the propagated interface. The environment determines the effects analogously to the changes to interfaces: whether there is redundant code, new code needed, or no effect at all. The same holds true for an implementation change and its effect on the propagated interface.

Given that the environment can determine the implications and extent of changes, we provide the facilities to simulate changes in addition to propagating them. In this way, the developer can determine whether a set of changes might have too adverse an effect or not before committing to those changes.

Finally, Infuse guarantees that the implications of changes will be carried out completely and consistently. As consistency is at the deeper level of semantics, the system builder gains more from the environment than he or she would where consistency is only at the level of syntax.

4.4 Predicates and Version Control

Inscape's version control mechanism, *Invariant* [22], uses the same model as Inform and Infuse use in order to provide a formalization of version control that has a number of significant advantages over version control mechanisms that use either the unit or syntactic interconnection models. Most notable is the notion of *plug-compatibility* which arises from Invariant's concepts of version equivalence and version compatibility. Briefly, versions are equivalent if their interfaces are identical; versions are compatible if they can be used (with various kinds of benign or even deleterious effects — depending upon the particular kind of version compatibility) interchangeably. Since we reason at the level of behavior provided and behavior used in the construction of systems out of components, we exploit the fact that the environment knows what behavior has been used in the construction of the system and the fact the environment can determine the extent to which similar but not necessarily equivalent versions can be substituted into a system model.

Furthermore, as the consistency of the implementations and interfaces is something that Inscape can guarantee (within the level of consistency supported by the implementation of the environment — this is one of the trade-offs made: how weak a logic is needed versus how strong a logic can be supported),

Invariant can check the consistency of the system models with respect to their semantics, not just their syntax.

5. Conclusions

The semantic interconnection model incorporates the advantages of the unit and syntactic interconnection models and provides extremely useful extensions to them. By refining the grain of interconnections to the level of semantics (that is, to the predicates that define aspects of behavior) we provide tools that are better suited to manage the details of evolution in software systems and that provide a better understanding of the implications of changes. We do this by using the semantic interconnection model to formalize the semantics of program construction, the semantics of changes, and the semantics of version equivalence and compatibility. Thus, with this formalization, we provide tools that are knowledgeable about the process of system construction and evolution and that work in symbiosis with the system builders to construct and evolve large software systems.

Acknowledgements

Peggy Quinn provided careful readings and numerous useful suggestions.

References

[1] *Military Standard: Ada Programming Language*, ANSI/MIL-STD-1815A, 22 January 1983.

[2] Robert Balzer. ‘‘Automated Enhancement of Knowledge Representations’’, *IJCAI-85 Conference Proceedings*, Los Angeles, August, 1985.

[3] Edsgar W. Dijkstra. *The Discipline of Programming*. Prentice-Hall, 1976.

[4] S. I. Feldman. ‘‘Make - a program for maintaining computer programs’’, *Software — Practice & Experience*, 9 (1979). pp 255-265.

[5] *Special Issue on the Gandalf Project, The Journal of Systems and Software*, 5:2 (May 1985).

[6] J. A. Goguen and J. J. Tardo. ‘‘An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications’’, *Proceedings of a Conference on Specifications of Reliable Software*, IEEE Computer Society, April 1979, pp 170-189.

[7] J. V. Guttag, J. J. Horning, and J. M. Wing. ‘‘The Larch Family of Specification Languages’’, *IEEE Software*, 2:5 (September 1985), pp.24-36.

[8] A. Nico Habermann and Dewayne E. Perry. ‘‘System Composition and Version Control for Ada’’, in *Software Engineering Environments*. H. Huenke, editor. North-Holland, 1981. pp 331-343.

[9] A. Nico Habermann, et al. *The Second Compendium of Gandalf Documentation*. Department of Computer Science, Carnegie-Mellon University. 24 May 1982.

[10] C. A. R. Hoare. ‘‘An Axiomatic Approach to Computer Programming’’, *CACM* 12:10 (October 1969). pp 576-580, 583.

[11] Gail E. Kaiser and A. Nico Habermann. ‘‘An Environment for System Version Control’’, *Digest of Papers Spring CompCon ’83*, IEEE Computer Society Press, February 1983. pp. 415-420.

[12] Charles W. Krueger. *The SMILE User’s Guide*. Carnegie-Mellon University, Department of Computer Science, The Gandalf Project, Draft, October 1985.

[13] Butler W. Lampson and Eric E. Schmidt. ‘‘Organizing Software in a Distributed Environment’’, *Proceedings of the Sigplan ’83 Symposium on Programming Language Issues in Software Systems. Sigplan Notices*, 18:6 (June 1983).

[14] David B. Leblang and Gordon D. McLean, Jr. ‘‘Computer-Aided Software Engineering in a Distributed Workstation Environment’’, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, 19:5 (May 1984). pp. 104-112.

[15] David Luckham and Friedrich W. von Henke. ‘‘An Overview of Anna, A Specification Language for Ada’’, *IEEE Software*, 2:2 (March 1985). pp. 24-33.

[16] S. C. Johnson. ‘‘Lint, a C Program Checker’’, *Unix Programmer’s Manual*, AT&T Bell Laboratories, 1978.

[17] David S. Notkin. ‘‘The Gandalf Project’’, *The Journal of Systems and Software*, 5:2 (May 1985). pp. 91-105.

[18] Dewayne E. Perry. ‘‘Position Paper: The Constructive Use of Module Interface Specifications’’, *Third International Workshop on Software Specification and Design*. IEEE Computer Society, August 26-27, 1985, London, England.

[19] Dewayne E. Perry. *The Inscape Program Construction and Evolution Environment*. Technical Report. Computer Technology Research Laboratory Technical Report, AT&T Bell Laboratories, August 1986.

[20] Dewayne E. Perry. *The Construction of Robust, Fault-Tolerant Software in the Inscape Environment*. Technical Report. Computer Technology Research Laboratory, AT&T Bell Laboratories, September 1986.

[21] Dewayne E. Perry and Gail E. Kaiser. ‘‘Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems’’, *Proceedings of the 1987 ACM Computer Science Conference*, February 17-19, 1987, St. Louis MO.

[22] Dewayne E. Perry. ‘‘Version Control in the Inscape Environment’’, This proceedings, *Proceedings of the 9th International Conference on Software Engineering*, March 30 - April 2, 1987, Monterey CA.

[23] D. M. Ritchie. *The C Programming Language - Reference Manual*. AT&T Bell Laboratories, September 1980.

[24] M. J. Rochkind. ‘‘The source code control system’’, *IEEE Transactions on Software Engineering*, SE-1 (1975). pp 364-370.

[25] Joseph L. Steffen. ‘‘Interactive Examination of a C Program with Cscope’’, *USENIX Winter Conference Proceedings*, Dallas 1985, pp. 170-175.

[26] Warren Teitelman and Larry Masinter. ‘‘The Interlisp Programming Environment’’, *Computer*, 14:4 (April 1981), pp. 25-34.

[27] Walter F. Tichy. ‘‘RCS — A System for Version Control’’, *Software — Practice and Experience*, 15:7 (July 1985). pp. 637-654.

[28] Walter F. Tichy. ‘‘Smart Recompilation’’, *ACM Transactions on Programming Languages and Systems*, 8:3 (July 1986), pp.273-291.

[29] W. A. Wulf, R. L. London, M. Shaw. ‘‘Abstraction and Verification in Alphard: Introduction to Language and Methodology’’, *IEEE Transactions on Software Engineering*, Vol. SE-2:4 (December 1976). pp 253-265.