Commitment-Based Software Development [1]

William Mark Sherman Tyler
James McGuire Jon Schlossberg
Information and Computing Sciences
Lockheed Palo Alto Research Labs O/96-01 B/254F
3251 Hanover Street
Palo Alto, CA 94034-1191

Abstract

During the development of a system, software modules can be viewed in terms of their commitments: the constraints imposed by their own structure and behavior, and by their relationships with other modules (in terms of resource consumption, data requirements, etc.). The Comet system uses explicit representation and reasoning with commitments to aid the software design and development process – in particular, to lead software developers to make decisions that result in reuse. Developers can examine the commitments that must be met in order to include an existing module, and can explore how commitments change when modules are modified. Comet has been applied to the domain of sensor-based tracker software.

I. Introduction

A major problem for software developers is judging how a change in a module affects and is affected by the rest of the design. Modules need to change for a variety of reasons: an existing system is modified, a change in an ongoing design is proposed, a bug is found, etc.; developers spend much of their time responding to changes. The Comet system provides computational support for developers in understanding – and being influenced by – the ramifications of proposed design modifications. Developers are given feedback about which design decisions will result in modules that can be "coded" (in particular, coded with the aid of synthesis and reuse technologies) with minimal disruption to the rest of the design. This is perhaps the major factor in real life design decision-making, but it has so far received little attention in software automation technology.

Software development is a process of negotiation: decisions are made and changed frequently as specifications change and new implementation ideas are brought forward. Each decision implies a set of constraints that may or may not be compatible with past or future decisions. Constraints are embodied in the modules – procedures and data structures – that make up the system. From the earliest stages of design, modules can be described in terms of the constraints they will place on the other modules in the system via their input, output, control structure, and shared resource requirements.

A software design comprises a large number of interacting constraints that must be met by the set of modules that make up the design. A subset of these constraints – of immediate interest to developers designing (or redesigning) a particular module for a particular system – are those that must be satisfied in order for that module to be included in a particular design.

For example, suppose a search module is required. Binary search has a commitment to sorted input. The decision to use binary search in a design clearly depends on this commitment. If the data to be searched is already in sorted form, binary search is a natural choice. If the data is not sorted and it is inconvenient to introduce a sort routine, using binary search becomes harder, and developers may choose a different kind of search.

The commitment types for any particular domain are defined by the ontology: commitments represent the preselected constraint types that are known to be "interesting" in determining the implications of component descriptions on each other. For example, in a mechanical domain, physical linkages, spatial relationships, and functional roles define commitments. In an electrical domain, commitments are concerned with connectivity, physical configuration, thermal, and radiation characteristics. In software, input/output, data access requirements, and control relationships represent commitments. In any domain, every component has commitments, some of which are intrinsic, "brought on by itself" by the component's own structure and behavior description; while others are extrinsic, "thrust on" the component by the other component descriptions in the design. Commitments are the constraints that bear on whether the component fits into the design, and are derivable from, and meetable by, some component description in the design.

A primary concern for developers is to meet the commitments of a module in a way that will not establish new commitments that will be hard to meet in the current design. Developers making design decisions about a module are thus engaged in "commitment management": determining the existing constraints that impact the module and the new constraints the module would add, and the amount of work required to satisfy them in the design.

A form of commitment management occurs now at the architecture level. An architecture encodes decisions about how the system is to be divided into modules and how these modules should interact. Many of these are represented explicitly (and visually) in various types of architecture diagrams, e.g., data and control flow diagrams.

[1] To appear in *IEEE Transactions on Software Engineering*, October, 1992.

Architecture commitments can be examined and to some extent reasoned about in terms of these explicit representations. Decisions that affect modules described at the architecture level, i.e., decisions that change module interaction, are immediately apparent in terms of the diagram.

In standard software practice, commitments below the level of the system architecture are usually not represented in a way that allows developers to reason about them. With the current code-plus-comments description of modules, commitments are implicit in the design: i.e., they reside only in the heads of the developers (and later, to a much lesser extent, in design documents). It is impossible to maintain an accessible record of the commitments as they continuously evolve. The result is that developers cannot assess the impact of a new decision.

Comet extends the commitment management style of software development that begins (and now ends) at the architecture level throughout the software lifecycle. Developers receive automated support in visualizing and keeping track of commitments during design and development. This implies that software modules must be represented in a way that allows rapid assessment of their commitments within the overall design.

Sections II, III, and IV describe Comet's representation and reasoning technology. Section V presents a "look and feel" overview of what it is like to use Comet. Section VI gives a detailed example, providing a scenario of Comet use and the specific reasoning that goes on behind the scenes to produce what the users see on the screen. The paper concludes with descriptions of related work, current status, and a discussion of some technical issues raised by this work.

II. Overview of Representation and Reasoning

The design knowledge managed by Comet is in the form of "module descriptions": structure and behavior specifications of modules interrelated by commitment constraints. The underlying representation is LOOM, a language and environment for knowledge representation and reasoning [10]. Declarative knowledge in LOOM consists of definitions, rules, facts, and default rules. The LOOM classifier implements forward-chaining, semantic unification, and object-oriented truth maintenance technologies in order to compile the declarative knowledge into a network designed to support efficient deductive query processing.

Comet is readied for a new application domain by building a set of core domain-specific module descriptions. These are built "by hand", i.e., directly in LOOM by Comet developers. Once the core modules are built in, further module descriptions are introduced by developers combining and specializing the domain-specific terms used in the definitions (Comet's facilities for enabling this user interaction with the knowledge base are discussed in Section V). Comet's ability to understand developer-introduced module descriptions depends on being able to automatically "place" new descriptions in its knowledge base, i.e., to understand their relationship to known terms. Because modules are described in precisely defined LOOM terms, LOOM automatically maintains a taxonomy of module descriptions based on the defined interrelationship of their constituent terms. That is, since new modules must be described as well-defined compositions of abstract classes, LOOM can automatically determine their subsumption relationships[2].

Comet's module representation is designed to enable support for developers in modifying existing designs. When a module description is modified, Comet finds the set of more specific module descriptions in the design knowledge base that are consistent with the newly modified description, and computes the new commitments that must be met for each of these alternatives in order for them to be included in the design. Since each commitment can only be met by further module description modifications, this process is recursive: a modification causes the system to compute a set of potentially relevant design alternatives and their commitments; it can in turn compute the alternatives that can meet these commitments, along with the commitments that they introduce, and so on. The computationally intensive reasoning processes within Comet are thus determining the set of modules in the design library that are consistent with a new description, and computing the new commitments they would introduce.

By using the concept of commitments to bound the set of constraints that must be computed at any given design step, and by using description logic representations and reasoning, these reasoning processes can be applied to large scale software knowledge bases at the performance levels required to support human interaction. Commitments represent the preselected constraint types that are known to be "interesting" in determining the implications of module descriptions on each other. Restricting constraint reasoning to commitment management with respect to a single focus module changes the computational support task from "full behavior verification" to "providing a useful service". Description logic representations ensure that the module interrelationships that must be examined for commitment management are described solely as compositions of primitive terms. This allows the construction of reasoning mechanisms than can rely on this rigor. The detailed complexity inherent in large scale software makes this a key requirement: the representers of the software must be able to depend on the reasoning system to perform the same sort of computations on any description that uses an agreed-upon set of terms.

III. Representing Module Descriptions

Throughout the development process, software developers deal with the system by manipulating module descriptions. The initial boxes-and-arrows design for the system is a set of very high level module descriptions. The fully implemented system is a set of very detailed mod-

[2] That is, LOOM can determine these relationships in principle. There are well known tractability issues that limit the operational feasibility of subsumption reasoning [12]; LOOM therefore determines some subsumption relationships, but not others [10].

ule descriptions (annotated code). Software development consists of the elaboration, addition, and modification of module descriptions.

Current programming languages provide module descriptions in only a very limited form. The software can be compartmentalized into packages and modules, and the input and output of the module can be described in terms of generic types. The language for describing types is limited to structural definition (order and substructure specification) in terms of a set of primitives (integer, etc.). The behavior of the module is usually described as a procedure, which is not well suited to design activities (e.g., comparing the behavior of alternative modules, summarizing the behavior of complex modules, assessing the relationship of one module to another).

Object-oriented systems include type taxonomies in which modules can inherit user-specified characteristics according to user-specified inheritance links. Although this helps the developer in understanding the organization and function of the modules in the taxonomy, these systems cannot enforce constraints on the use of modules based on their description. The system does not understand the relationship among modules; it only knows their position in a taxonomy that was defined outside of its purview.

The module description capability in Comet goes beyond that currently offered in both the richness of the descriptive language and the system's ability to make inferences and enforce constraints based solely on the descriptions it is given.

A. Structural Descriptions

As an introduction to Comet's taxonomic reasoning, we present the usual representation of modules in terms of structural descriptions and then show the use of classification to maintain taxonomies of modules represented in this way.

Module descriptions are encoded as LOOM definitions. Definitions bind module names to concepts (an abstract class of individuals) or to a relation (an abstract relationship among individuals). For example, procedures are defined as specializations of the primitive abstract class **Procedure Module**, whose relations describe substructure, high-level input/output ports, and interconnectivity – see Figure 1[3]. Ports have semantic datatype restrictions imposed on their values. Since these datatype restrictions are themselves abstract classes managed within the taxonomy, Comet can reason about the legality of connections and complain about connections between ports with incompatible datatypes (e.g., trying to coerce a set of cardinality greater than 5 into a set of cardinality less than 3).

As an illustrative example, Figure 2 shows specializations of the **Procedure Module** definition to describe

[3]LOOM expressions are shown in bold; abstract class names are capitalized, relation names are lowercase, and LOOM keywords are lowercase preceded by a colon. LOOM constructs are explained as they are introduced; a numbered comment will appear within the LOOM text and a corresponding explanation will appear in the caption of the enclosing figure.

```
(defconcept ProcedureModule
  :is :primitive ;1;
  :constraints
      (:and (:all submodule ProcedureModule) ;2;
            (:the parent ProcedureModule)    ;3;
            (:all input-port-of InputPort)
            (:all output-port-of OutputPort)
            (:all submodule-interconnectity Connection)
            (:the behavior-of Behavior-Sequence)))
```

Fig. 1. Basic LOOM module description definition. (1) A primitive is a LOOM construct that specifies an abstract class that is not defined compositionally in the knowledge base. Primitives can be specialized with compositional definitions. (2) Every submodule must be a **Procedure Module**. (3) Only one parent can exist, and it is a **Procedure Module**.

core procedures used for a "grading" domain, i.e., assigning letter grades on a curve. The specializations are formed by restricting the module types of subcomponents and/or the datatypes of inputs and outputs. Each domain-specific term mentioned in the definition (e.g., **Array Student Scores**) is itself an abstract class with a precise compositional definition in terms of other abstract classes, eventually bottoming out in the set of primitives for that domain.

In this example, a grade threshold is identified for use in partitioning grades into groups. Scores in the "good" group are then assigned either an A or B based upon which half they belong to; scores in the "bad" group are assigned C, D, or F based on which third they belong to. Different alternatives for identifying the threshold are modeled as specializations of **Identify Grade Threshold**.

One such alternative is **Median Grade Threshold**, whose behavior (discussed in Section III.B) is to identify the median score stored in the input array. The other alternative is less precise (and less fair-minded): **Special Circumstances Grade Threshold** posits extra inputs in addition to **Array Student Scores** (which is mandated via inheritance from the definition of **Identify Grade Threshold**). This allows something other than the grades themselves to influence the cutoff grade choice. The essence of this is captured by the cardinality constraint (at-least 2) on the number of values to fill its input-port-of relation.

As a simple example of user-defined specialization, Figure 3 shows a case in which an unscrupulous teacher wishes to guarantee the grades of favorite students. The teacher specializes the more general grading function **Identify Grade Threshold** to take an argument **Student** that identifies the lucky student. LOOM will classify the teacher's new **My Grade Threshold** module as a specialization of **Indentify Grade Threshold**, as it is instructed to in the definition of the concept. Furthermore, LOOM will recognize during this process that, based on their LOOM definitions, **Student** is not compatible with the input type **Array Student Scores** of **Identify Grade Threshold**. Thus, it will conclude that there must be at least two required input datatypes for this module and it will automatically discover that the new module description is a specialization of **Special Circumstances**

```
(defconcept GradeOnCurve :is
  (:and ProcedureModule  ;1;
        (:some submodule IdentifyGradeThreshold) ;2;
        (:some (:compose input-port-of data-type) ;3;
               ArrayStudentScores)
        (:some submodule PartitionGradesByThreshold)
        (:some submodule  AssignGoodGradesAorB)
        (:some submodule AssignBadGradesCorDorF))))
(defconcept IdentifyGradeThreshold :is
  (:and ProcedureModule :primitive
        (:some (:compose input-port-of data-type)
               ArrayStudentScores)
        (:the (:compose output-port-of data-type) Score)))
(defconcept MedianGradeThreshold :is
  (:and IdentifyGradeThreshold
        (:exactly 1 input-port-of)  ;4;
        (:some behavior-of FindMedianGradeBehavior))))
(defconcept SpecialCircumstancesGradeThreshold
  :is (:and IdentifyGradeThreshold
            (:at-least 2 input-port-of)))
```

Fig. 2. Domain specific core module descriptions (1) Inherit all the requirements of a **Procedure Module**. (2) There exists a submodule of type **Identify Grade Threshold**. (3) To impose a type restriction on a linear composition of relations, one can employ LOOM's :compose operator. Thus this restriction says that there exists a data-type of an input-port-of the module which is of type **Array Student Scores**. (4) Cardinality restriction to one value.

```
(defconcept MyGradeThreshold :is
  (:and IdentifyGradeThreshold
        (:some (:compose input-port-of datatype) Student)
        (:some behavior-of FindGradeBehavior))))
```

Fig. 3. A user-defined specialization

Grade Threshold[4].

This automatic classification-based inference is important because there may be software design ramifications of using a **Special Circumstances** kind of thresholding scheme, and Comet can make the teacher/developer aware of it.

B. Behavior Descriptions

The structural implications of existing module descriptions on current design decisions are usually relatively clear to system developers. It is the behavioral implications that most require computational support to be made evident. Comet must therefore be able to represent and reason about the behavioral implications of module descriptions.

In Comet, the behavioral description of a module is specified via its **behavior-of** relation. Behavioral descriptions are compositions of behavior primitives, which are elaborated in each application domain (see Section VI.C) from a predefined set of generic behavior primitives. Generic behavior primitives have so far been defined to represent:

- boolean **Test Condition**;
- **Actions** with side-effects;
- **Sequences** of behavior, with a relation defined for enumerating the **steps**;
- **If Then Else Behavior** control flow whose **if** relation is restricted to be a specialization of **Test Condition** and whose **then** and **else** relations are specializations of a behavior **Sequence**;
- message passing between modules;
- iteration/**Mapping** over collections of elements much like LISP's MAPCAR with a relation for describing the mapped **lambda-expression**; and
- **Filtering** constructs to select elements from collections based upon the **Test Condition** role restriction of the **filtering-criteria** relation.

Relations in the primitive behavior descriptions (e.g., **step**, **lambda-expression**, **then**, **else**, **if**) are filled by other behavior types, and behavior descriptions can be qualified by their input parameter types. Figure 4 shows the behavior definitions for the modules introduced in Figures 2 and 3. For example, to implement **Find Median Grade Behavior**, a requirement is imposed that some step in the behavior sequence must be a call to the primitive function **Access Middle Of Array**. Furthermore, the input parameter must be a sorted array.

The description **Find Grade Behavior** implements the behavior of the module **My Grade Threshold** shown in Figure 3: it mandates that some step in the behavior sequence be a **Student Search**. Via inheritance from **Binary Search**, the **Student Search** behavior is implemented as an array search using an **If Then Else Behavior** construct. The **if** relation uses the **Equal Value** behavior primitive to test whether the target has been found. To handle the case where this does not occur, the **else** relation's implementation is defined to contain a step that is a nested **If Then Else Behavior**. Within this nested structure, the behavior primitive **Less Than Value** determines which half of the array to do a recursive binary search on.

C. Augmenting Behavior Descriptions with Test Runs

Behavior descriptions are used to discriminate among **Procedure Modules**. The intent is not to build executable specifications, but rather to develop a rich enough set of descriptors to allow retrieval of modules based on descriptions of behavioral requirements.

Subsumption checking over behavior descriptions is not always possible because of the limited expressiveness of the language over which subsumption checking is feasible. Consequently the classifier may be powerless to draw distinctions between several modules within the same taxonomic neighborhood, even though the necessary conditions have been specified.

To deal with this modeling problem, Comet allows module descriptions to be augmented with example test runs. Testruns are represented in LOOM via the **test-run** relation (see Figure 5). The values for this relation are instances of the concept **Run**, which has relations defined for capturing each **test-input** and **test-output** of an example run.

Since testruns are grounded instances of actual behav-

[4]The ability to infer cardinality restrictions from attribute-type incompatibilities will be supported in LOOM 2.0.

```
(defconcept FindMedianGradeBehavior :is
  (:and Behavior-Sequence
        (:some step
               (:and AccessMiddleOfArray
                     (:the input-parameter
                           (:and ArrayStudentScores
                                 SortedArray))))))
(defconcept FindGradeBehavior :is
  (:and Behavior-Sequence
        (:some step StudentSearch)))
(defconcept StudentSearch :is
  (:and BinarySearch
        (:some input-parameter ArrayStudentScores)
        (:some input-parameter Student)
        (:some output-parameterScore)))
(defconcept BinarySearch :is
  (:and ArraySearch
        IfThenElseBehavior
        (:some if  EqualValue)  ;1;
        (:some (:compose else step) IfThenElseBehavior) ;2;
        (:some (:compose else step if) LessThanValue)) ;3;
  :constraints
    (:and (:some (:compose else step then step)
                 BinarySearch) ;4;
          (:some (:compose else step else step)
                 BinarySearch) ;5;
          (:some intrinsic-commitment RequireSortCommit)))
```

Fig. 4. Behavior description.
(1,2,3,4,5):
if (EqualValue) then {..}
else { if (LessThanValue) then {BinarySearch} else {BinarySearch}}

```
(defrelation test-run
    :is :primitive
    :domain (:or ProcedureModule Behavior)
    :range Run)
(defconcept Run :is
    (:and :primitive
          (:all test-input Thing)
          (:all test-output Thing)))
```

Fig. 5. The **test-run** relation and the **Run** concept.

ior, they have the nice computational property of being amenable to backward-chaining pattern recognition. Deductive patterns over testruns can be devised to specify necessary and excluding conditions for behaviors that are difficult to capture using only the subset of first-order logic exploitable by LOOM's classifier. To locate likely candidates for module reuse, Comet uses these deductive patterns as filters to discriminate between closely related modules after the classifier has focused attention to a relevant set of concepts.

As an example, let's return to the grading domain. A new teacher wishes to define a grade threshold selection algorithm: grades are placed in a fixed-size priority queue, and the lowest grade left in the queue will be the good/bad grade threshold. The teacher calls this algorithm **Reducing Sort** (see Figure 6) and specifies it as a descending sort algorithm that does screening. **Descending Sort** is a specialization of the **Sort** primitive behavior. The **Screen** behavior is also a difficult requirement to capture in the subset of LOOM that is amenable to classification. The critical aspect of screening is therefore captured in Comet as a deductive pattern over testruns, looking for

```
(defconcept ReducingSort :is
  (:and DescendingSort Screen))
(defconcept Screen :is
  (:and Behavior
        (:some input-parameter Collection)
        (:some output-parameter Collection)
        (:satisfies (?behav)
           (:for-some (?run ?input ?output)
               (:and (test-run ?behav ?run)
                     (test-input ?run ?input)
                     (test-output ?run ?output)
                     (strict-subset ?output ?input)))))
```

Fig. 6. The concepts **Reducing Sort** and **Screen**.

cases where an output data collection is a strict subset of an input data collection. To find potentially reusable behaviors, LOOM performs pattern recognition on all instances of behavior which are a type of **Descending Sort**. Consequently less candidates (i.e., the more relevant ones) will be recommended for reuse than would have been suggested by a mechanism relying only on LOOM's classifier. While not being adequate to capture behavior completely, testruns are still an accurate description of behavior and provide a vehicle to more closely associate LOOM behavior descriptions with actual code.

As will be discussed in the next section, Comet provides a means for modules to impose behavioral requirements on their neighbors. These constraints are expressed via LOOM concepts. Deductive patterns over testruns can be exploited to express more precise behavioral constraints, thereby enriching the class of constraint checking inferences that Comet will provide for its users.

IV. Representing Commitments

Again, commitments are the constraints that must be satisfied for a particular module to be included in a particular design. Some of these are intrinsic (e.g., the commitment of binary search to sorted input); while others are extrinsic (e.g., the commitment to use the message passing protocol used by the other modules). In any case, the only constraints that qualify as commitments are those that bear on whether the module fits into the design, and are derivable from, and meetable by, some module description in the design.

The Comet approach does not claim to determine the complete set of commitments for a given module automatically – indeed that is probably impossible. Instead Comet assumes the responsibility of reliably (and rapidly) managing a well defined subset of the commitments, and of providing visualizations to aid the developers in using them in their development process.

This "well defined subset" of commitments consists of those that Comet can automatically infer from module descriptions, and those that are explicitly represented as **Commitment** annotations on module descriptions. The most straightforward kind of automatically inferable commitments involve implied input/output relationships. For example, the **Send Message** class is the primitive behavior responsible for representing message passing between modules. Any module whose behavior description includes an element of the **Send Message** class has automatically inferable commitments to other modules in the design to ensure that messages are properly received. Similarly, the input datatype restrictions on all procedural modules imply commitments that mandate the existence of upstream modules that are capable of producing output datatypes compatible with the module's input requirements.

Comet's use of declarative behavior descriptions allows automatically inferable commitments arising from type restrictions to be more subtle than module input/output port matching. For example, **Find Median Grade Behavior** in Figure 4 uses the **Access Middle Of Array** behavior primitive and further restricts its input parameter to be both a **Sorted Array** and an **Array Student Scores**. This allows a commitment to be inferred which is assigned to all modules whose behavior can be classified as a specialization of **Find Median Grade Behavior**; either the module's behavior includes an additional prior sorting sub-behavior **step**, or the module posseses an **input-port-of** whose datatype is of the correct sorted array type.

To expand the support offered by Comet, commitments can also be represented explicitly as annotations on core module descriptions. Commitments are represented as the class **Commitment** with the relation **requirement** for specifying criteria for how the commitment can be met within a design. For example, the **Binary Search behavior** in Figure 4 has a requirement that the **Require Sort Commit** (shown in Figure 7) definition be satisfied. **Require Sort Commit** is just an illustrative explicit representation of the automatically inferred commitment already supported by Comet and discussed in the preceding paragraph. The requirement for meeting the commitment is that the behavior's parent module must be classifiable as a **Module Containing Sort Behavior**. This is defined by the module either having a sorted input or containing a sorting sub-behavior within the composed behavior description of the module. The relation **behavior-network-member*** defines the transitive search through the module's behavior description.

So far, we have been concerned with the commitments introduced by a new module – commitments that must be met by other modules in the design. Comet must also recognize which of the other modules' unmet commitments are met by the module being introduced into the design. Furthermore, it would be useful to tell developers when a particular kind of commitment *ought to* be met by a new module, and to give the new module the responsibility of meeting that commitment.

Comet relies on LOOM's deductive capability to determine which outstanding commitments in a design are actually met by each newly introduced module. Each unmet commitment is compared against the structure and behavior description of the new module. If by backward chaining or subsumption checking LOOM can decide that some aspect of the description satisfies the criteria speci-

```
(defconcept RequireSortCommit :is
  (:and Commitment
       (:the requirement
             ModuleContainingSortBehavior)))
(defconcept ModuleContainingSortBehavior :is
  (:and Module
        (:satisfies (?mod)
           (:or (:for-some (?datatype ?port) ;1;
                    (:and (input-port-of ?mod ?port)
                          (datatype ?port ?datatype)
                          (SortedArray ?datatype)))
                (:for-some (?behav)  ;2;
                    (:and (behavior-network-member*
                                (behavior-of ?mod) ?behav)
                          (SortBehavior ?behav)))))))
```

Fig. 7. Explicit commitment annotations. (1) Either the module has an **input-port-of** whose datatype is a **Sorted Array** or (2) some behavior element within the module's behavior decomposition is of type **Sort Behavior**. The relation **behavior-network-member*** recursively descends through the behavior description of a module finding all member sub-behaviors.

fied in the commitment, the commitment is removed from the "unmet" list. To allow for LOOM's incomplete reasoning capability, Comet allows developers to override this procedure and explicitly assert that a module meets a particular commitment – but it maintains a record of where the assertion came from.

To achieve the goal of informing developers when a module ought to meet a commitment, we have added a **responsible-for** relation on the **Commitment** class for specifying what module classes should be meeting the commitment. This "responsibility" is inherited by any user-defined specializations of the module. If a new module is "responsible-for" but is unable to meet an outstanding commitment in a particular design, Comet informs the developers of that fact, and of other module descriptions (if any) that are capable of meeting the commitment.

After the intrinsic commitments of a newly incorporated module have been identified, responsibility for meeting each intrinsic commitment is assigned to those modules satisfying the commitment's **responsible-for** relation criteria. These become extrinsic commitments for those modules: extra responsibilities thrust upon them by the intrinsic commitments of the new module. If no existing module meets the "responsible-for" criteria, then new modules need to be incorporated into the design. If a module meets the **responsible-for** criteria, but does not meet the requirements of a commitment, the developers are warned that it must be replaced.

V. Look and Feel

This representation and reasoning capability is used to provide a direct manipulation-style visual feedback of user interaction with the Comet system. Figure 8 shows a schematic Comet screen[5] representing a snapshot of a development process in progress. The screen consists of three windows: Module Description, for editing diagram representations of modules; Forms-Based Editing, for editing

[5]This schematic is a MacIntosh drawing, but the other "screen pictures" are printed directly from the Comet screen.

text representations of modules; and Design Memory, for displaying existing module descriptions form the Comet knowledge base that could fit into the current design, along with the commitments that must be satisfied in order for them to do so. These windows will be discussed in terms of a typical user interaction sequence, following the numbered arrows in Figure 8.

Developers begin by browsing system diagrams – much as they might look through design documentation from previous systems before embarking on their task. If the job is a modification of an existing system already known to Comet, the developers might begin by examining a fairly specific module. If the job is to build a "whole new system" in the domain, the starting point might be an existing architecture. From Comet's point of view, the starting point does not matter: module descriptions at all levels are expressed externally in terms of diagrams and text, and internally in terms of the LOOM-based structure and behavior representation language.

The developers can examine the structure and behavior of modules to see if they meet the new requirements, or, more usually, to see what has to be changed in order for them to meet the new requirements. The Module Description window in Figure 8 shows a graphic rendering of the module description under consideration. Currently, only data flow diagramming is supported: various levels of detail can be viewed via recursive Module Description windows.

In addition, developers can view English-like descriptions of the module's structure and behavior. The text is generated (in a very limited way) by the system. Parameterized English-like phrase forms are associated with constructs in the module description language. A particular module description is rendered by instantiating the parameters of the appropriate phrase forms using the corresponding elements of the description's constructs.

Comet must allow developers to express new module descriptions to meet new needs. However, given current technology, the system must severely limit the users' flexibility in this task. Comet allows the user to introduce new module descriptions only as specializations of existing module descriptions. In practice this is not much of a restriction, since some of the existing module descriptions are at a very high level of abstraction – virtually any module can be seen as a kind of one of them. The more telling restriction is that Comet allows the user to specialize module descriptions only with a small set of predefined modification alternatives. These are presented to the user in terms of a forms-based interface: essentially the user must pick from the modification options presented on the menus (see section VI.C). In step (1) of Figure 8, the developers are modifying the text version of the module description in the Forms-Based Editing window in order to make it meet new requirements. The modified text is automatically translated back into LOOM, resulting in a new description, shown highlighted in the Module Description window.

In step (2) of Figure 8, the developers request to see any existing, more detailed, module descriptions in the Comet knowledge base that are consistent with the new description they have just created. The goal is to find existing assets that might apply to the new job. The idea is that when the developers propose a module to fulfill some aspect of their design, they should be able to see if the system knows of any existing module descriptions that could possibly fill that role. The Design Memory window [11] presents module descriptions that are potential substitution candidates. The leftmost column lists vertically the alternative module descriptions that are compatible with the current design. A crucial part of the Comet philosophy is that "compatible with" means that their descriptions are subsumed by the developers' description, and their commitments can potentially be met in the current design. This is rather different than the usual notion of compatibility: the candidate modules are not *yet* compatible with the design, but they can be made so if their commitments are not met. This notion is very important to the tractability of the system's reasoning processes (see Section IX.A).

The links emanating from each module description represent its commitments. Each commitment can be met by incorporating the module description it points to (another module known to the system, and thus another asset) into the current design. A dashed link indicates that the commitment has already been met in the current design. The Design Memory window therefore gives developers immediate visual feedback on the ramifications of using known assets: each commitment must be met, which means that the designated other parts of the design must be altered to include the candidate module descriptions. In the figure, after exploring the commitments, the developers have decided that the highlighted module description in the Design Memory window meets their new requirements, and that its commitments are not too hard to meet. In step (3) of Figure 8, they substitute it for the highlighted module in the Module Description window, thus altering the design. Each module description modified in the Comet environment adds to the system's store of reusable assets – reusable because they are described in terms of their commitments to other known assets.

In Comet, choosing substitution modules is deliberately designed as an interactive process. We believe that developers must play an active role in reuse. First, as a practical matter, the system cannot be expected to understand all commitments among modules. Second, we believe that exploring the reuse memory should be a feedback process: if the developers specify a module description that leads to an empty Design Memory window, or that gives rise to candidates that have onerous commitments, the developers may wish to reconsider their proposal. That is, a valid reason for developers' inability to find appropriate modules is that they are looking for the wrong thing; their thinking or their requirements need to change. One of the most powerful reasons to change a design is to make existing solutions applicable.

Fig. 8. Schematic view of a Comet screen. Arrows represent data flow, boxes represent data structures, and ovals represent procedures. Double boxes or ovals indicate that the module description is associated with code. Heavy dashed arrows show numbered steps in the user interaction sequence described here, and do not actually appear on the screen.

VI. Example

We now show this "look and feel" in the context of an actual scenario of Comet use, and explain how the representation and reasoning discussed in Section 2 implements the system's behavior. We first introduce the application domain, and then sketch some highlights from an actual Comet scenario. Finally, we give a "behind the scenes" look at the system, showing some detailed module descriptions from the application domain, and describing specific reasoning activities.

A. Domain

We have tested Comet representation and reasoning by constructing an application in the domain of "trackers". We have examined actual tracker systems, from design documents to code, and created (by hand) Comet module descriptions of significant portions of the system to form the core knowledge base in the tracker domain. In parallel (i.e., without reference to the specifics of existing systems), we developed some informal requirements for a hypothetical new tracker system for construction using Comet. Our purpose was to determine whether Comet's module description language is adequate for representing the intricacies of real software systems, and whether the reasoning mechanisms are sufficient (including sufficiently efficient) to support interactive software development.

Tracker systems take sensor data about vehicles moving through space and resolve the data into individual vehicle tracks. Air traffic control systems are a familiar example. Tracking comprises a variety of functions, including accessing sensor data in the form of "contacts" (probable vehicle positions), screening the data according to the regions that could contain the predicted continuations of known tracks, initiating new tracks, updating tracks, making new predictions, etc.

For any new application domain, Comet's built-in behavior primitives (discussed in Section III.B) must be extended with domain-specific specializations. Figure 9 shows the taxonomy for some of the behavior primitives created for the tracker domain . For example, the generic Volume Containment primitive has been specialized to accommodate various kinds of volumes of space that are relevant to tracker software ("scene gates", "track gates", and "clusters"). This extended set of primitives is then used to build (compositionally) behavior descriptions for the actual software module descriptions that form the core for this domain.

B. Scenario

The core module descriptions for the tracker having been built, one scenario of Comet use begins with software developers being given a requirement to build a multi-hypothesis tracker, i.e., a tracker that can temporarily associate an ambiguous contact with more than one track; later contacts are expected to resolve the ambiguity. The developers have access (via Comet) to descriptions of existing tracker systems. The goal is to use Comet to see if one of the existing systems meets this requirement, or whether one can be easily re-engineered to meet the new requirement. Developers can browse through Comet's knowledge base of tracker designs, examining module descriptions in terms of architecture diagrams and English text as described above. In this case the developers select an existing single hypothesis tracker design[6] to re-engineer to handle multiple hypotheses.

[6]This example is based on an actual Lockheed tracker.

Fig. 9. Part of the behavior taxonomy. Ovals represent concepts (i.e., the constructs created by the **defconcept** operator in LOOM); arrows represent subsumption relationships; the semicircular line connecting arrows below a concept indicates that the linked subsumers form a disjoint covering.

Step 1: Modify Assign Contact to Track to handle multiple hypotheses

The developers focus first on the Coarse Contact Screening module, one of four major high-level modules in the existing design. This module examines incoming contact data from the sensors and assigns the contact to an appropriate existing track. If there is no appropriate existing track, Coarse Contact Screening will initiate a new track, or hold the contact in a temporary data structure (called a cluster) until more information is known. The developers find that the first place that needs to be modified within Coarse Contact Screening is the module that assigns contacts to existing tracks, called Assign Contact To Best Track. To examine the behavior of this module, the developers call up the generated text description of the behavior of this module, shown in Figure 10.

The developers see that the aspect of module behavior represented by the phrase
For the one contact/track gate with the highest score
clearly needs to be modified (after all, they are interested in multiple hypotheses, not just the one with the highest score). They therefore request possible replacement options, and the system presents three, one of which selects all of the contact/track gates rather than choosing only the one with the highest score. This option is chosen, and the developers delete the two lines that refer to track scoring, since the scoring is superfluous given that all contact/track gate assignments are now appropriate.

The developers save this new behavior, causing Comet to replace Assign Contact To Track with a temporary placeholder module called (by the developers) Assign Contact To All Tracks. Comet maps the new behavior into a LOOM concept, created from the lower-level concepts that correspond directly with English phrases (see next section). This concept is classified in the taxonomy.

Next, LOOM is queried for modules whose behavior specializes this concept. For each of these modules, Comet determines all of their commitments relative to the existing design, and presents them in the Design Memory window. In our example scenario, only one such specializer is located, Multi Assign Contact To Track.

Step 2: Add Scene Gates

The developers now begin to browse the Design Memory window (see Figure 11) in order to explore the ramifications of substituting Multi Assign Contact To Track into the design. They find that two data structure commitments (Scene Gate and Contact) and two procedure commitments (Scene Gate Containment Check and One To N Assoc Update Scene History) are currently outstanding.

Focusing on the Scene Gate commitment, the Design Memory window shows that it can be met by introducing an Agglomerate Gates procedure. Agglomerate Gates takes Track Gates as inputs and "agglomerates" them into Scene Gates. Track Gate is tracker terminology for the volume of space that could possibly contain the continuation of a track; Scene Gates are composed of Track Gates that overlap, and are normally introduced to gain efficiency in analysis.

The developer chooses to introduce the Agglomerate Gates module into the design. This can be accomplished directly from the Design Memory window by selecting the Agglomerate Gates icon. Comet allows only legal connections to existing ports; if some of the required ports do not yet exist (modules might be introduced into the design before the data structures they need), Comet allows dangling connections to exist temporarily.

Fig. 10. Forms-based editing of behavior descriptions

Fig. 11. Partially modified design with Design Memory window

Step 3: Replace Test for Track Gate Containment with Scene Gate Containment

The developers then proceed to examine the Scene Gate Containment Check procedure commitment (at the top of the Design Memory window in Figure 11). This commitment arises from the fact that the current design contains a Test For Track Correlation Gate Containment, which functions in terms of Track Gates. However, we already know from the previous commitment that the proposed Multi Assign Contact To Track works in terms of Scene Gates, not Track Gates. Thus the Test For Track Correlation Gate Containment module needs to be replaced with one that handles Scene Gates.

The developers can see how much work it will be to incorporate Scene Gate Containment Check by examining its four commitments. The commitment to Assign Contact To Cluster has already been met (indicated by the dashed line), since an acceptable Assign Contact To Cluster module already exists in the current design. Satisfied that the remaining three unmet commitments will not pose too much of a challenge, the developers substitute the new Test For Scene Gate Containment module for the existing Test For Track Correlation Gate Containment module in the design.

Figure 12 shows all of the commitments in the Design Memory window having been met. Note the corresponding changes in Coarse Contact Screening in the Module Description window.

C. Behind the Scenes

This section highlights some of the representations and reasoning underlying the steps in the scenario described

Fig. 12. Revised design with all commitments met

above.

Supporting Forms-Based Editing

Step 1 of the scenario showed a new LOOM behavior description being created by modification of a text description of an existing behavior. Comet implements this capability by offering behavior alternatives, each described by a text phrase in a menu. These phrases correspond directly with domain-specific behaviors in the core taxonomy. When the developers wish to modify a phrase in a text behavior description, as they did with
For the one contact/track gate with the highest score
in the scenario, Comet creates a menu of behavior alternatives by finding the most specific other domain behavior primitives that are subsumed by the same generic behavior primitive. Figure 13 shows how the phrases in the menu in Figure 9 are associated with behavior primitives in the taxonomy.

Finding Relevant Substitution Candidates

When the developers choose one of the alternative primitive behaviors,
For all contact/track gates
in the scenario, Comet uses it to compose a new behavior description (i.e., it substitutes it into the composed behavior description of the existing module). In the scenario, the modified behavior description forms a LOOM concept named **Placeholder Behavior1**, shown in Figure 14. Comet's goal is to see if this behavior description – the one that the developers want to introduce into the design – subsumes the behavior description of any known module. That is, Comet looks to see whether any of the modules that it knows about are compatible with, but more specific than, the developers' specification. Such modules would be good candidates for reuse in the developers' current design.

In Figure 14, one candidate behavior description, **Assign Contact to All Track Behavior** is found which is subsumed by the newly formed behavior concept **Placeholder Behavior1**. This is the behavior description of the module **Multi Assign Contact to Track**, which appears as the single oval icon in the left-hand column of the Design Memory window (Figure 11) in Step 2 of the scenario.

Computing Commitments

After **Multi Assign Contact to Track** has been selected as a substitution candidate, Comet computes its commitments via inheritance (see Figure 15) and presents them in the Design Memory window. **Multi Assign Contact to Track** has intrinsic commitments to input of types **Contact** and **Scene Gate**, giving rise to the bottom two commitments displayed in the Design Memory window in Figure 11. In addition, the module **Multi Assign Contact to Track** inherits the explicitly represented commitment **Scene Gate Containment Check**, shown in Figure 16. The module **responsible-for** fulfilling this commitment is a module of type **Coarse Test For Containment**, and the module **Test For Track Correlation Gate Containment** in Figure 11 meets the **responsible-for** criteria. The **requirement** which this module must meet is specified in the **Scene Gate Containment Check** description (not shown); this further constrains the **Contact** and **Scene Gate** outputs of the responsible module such that the **Contact** must be contained in the volume of space represented by the **Scene Gate**. This constraint is specified behaviorally, using static test runs (as described in section III.C) to verify that all Contact/Scene Gate output pairs from the responsible module have the primitively defined **contains** relationship between them. This commitment gives rise to the commitment link to the **Scene Gate Containment Check** module at the top of the Design Memory

Fig. 13. The association of behavior alternatives with English phrases

Fig. 14. Finding substitution candidates

window in figure 11. The fact that this commitment link is solid indicates that the existing module within the design meeting the **responsible-for** criteria does not meet the **requirement** criteria, and consequently alternative modules need to be suggested by Comet.

Suggesting Modules to Meet Commitments

For each commitment it discovers, Comet tries to suggest existing modules that can meet it. It looks first for modules that exist in the current design; if it finds any, the commitment is considered to be met, and is shown with a dashed line, as occurred in Step 3 of the scenario. Otherwise, Comet will search for existing modules that *could* be incorporated into the design, and shows the commitments that must be met in order to do so. For an explicit commitment, the characteristics of the modules that could meet it are spelled out in the **requirement** relation, and Comet need only find the subsumees of the criteria defined in it. For other commitments, some special reasoning over module descriptions is required. For example, to meet a commitment mandating an input of a particular datatype, Comet looks upstream in the design for modules of that datatype or modules that produce output of that datatype. If no such modules exist, Comet will try to find a module that can be introduced into the design in the right context that is capable of of *creating* the desired datatype.

In the scenario we saw the **Agglomerate Gates** module presented as a way of meeting the **Scene Gate** datatype commitment. No module in the existing design produced the **Scene Gate** datatype. Thus the taxonomy was searched for all modules capable of creating a **Scene**

Fig. 15. Intrinsic commitments computed via inheritance. (A) Connections must be created to siblings producing **Contact** and **Scene Gate** data structures. (B) Modules exist fulfilling the commitments **Scene Gate Containment Check** and **One to N Assoc Update Scene History**.

```
(defconcept SceneGateContainmentCommit :is
 (:and ContainmentCommit
       (:the responsible-for CoarseTestForContainment)
       (:the requirement SceneGateContainmentCheck)))
```

Fig. 16. An explicit commitment. The module "responsible-for" meeting the commitment is of type **Coarse Test For Containment**. The requirement it must meet is described by the description **Scene Gate Containment Check**, which is (not shown) expressed as a deductive pattern over the stored behavioral testruns of a module.

Gate. For a **Procedure Module** to be considered a "creator", it must possess a behavior capable of producing the desired datatype as output from other input datatypes excluding the desired one. So a behavior that took a **Scene Gate** as input and produced it as output is not considered a creator. As it turned out, the behavior description of the **Agglomerate Gates** module possessed the relevant creator sub-behavior.

VII. Alternative Approaches and Related Work

All knowledge-based approaches for facilitating software development share a common underlying theme of supporting developers by reasoning in terms of explicitly represented knowledge about software. However, the various approaches differ greatly in their emphasis on particular technologies and stages of the development process. Comet's emphasis is as follows:

- detailed, computational treatment of module behavior as well as structure – Comet contains formal descriptions of the behavior of each module, and has the capability of capturing and reasoning about complex behaviors from both their constituent primitive behaviors and stored prototypical test runs;
- primary attention to the interaction of design decisions in an evolving design – Comet provides context-specific guidance on what existing modules may be relevant to include in a design, and what design modifications will be required in order to include them; and
- support for the design phase of system construction, when the system requirements and behavioral specifications are being explored and negotiated in order to determine the internal structure of the software system – Comet's users are system engineers responsible for architecting the system, rather than programmers coding to given requirements. The implications of these choices can be seen by comparison with other knowledge-based software development systems.

The Knowledge-Based Software Assistant is an attempt to develop a knowledge-based paradigm supporting all phases of the software development lifecycle from requirements through code implementation. Notable achievements in this program have included the Knowledge-Based Requirements Assistant (KBRA) of Czuchry and Harris [1] and the Knowledge-Based Specification Assistant of Johnson [5], as well as the combination of these two components into the ARIES system [6].

Like Comet, these systems are aimed at supporting reuse. KBRA, for example, supports the development of system requirements by managing informal information in an intelligent notebook, noting inconsistencies between different parts of this notebook, generating different presentations of information, and critiquing and sometimes completing partial descriptions. To enable requirements reuse, the system relies on users browsing through the domain taxonomy or requesting information using system names. It does not have a facility for retrieving and reasoning about relevant information based on abstract descriptions, as Comet does. Also, while it uses constraints and classification within its knowledge representation language (Socle), the classification is based on special purpose decision tables and the constraints are primarily numeric, derived from formulas relating requirement entities. Thus, KBRA does not exploit general classification or rely on symbolic constraint reasoning to determine suitable reuse candidates or system consistency in the way Comet does. Instead, it focuses more on the issues of multiple presentations of information and maintaining their consistency through a central repository of requirement statements, something not yet pursued in Comet.

The IDeA system of Lubars and Harandi [9] and its successor, Rose-1 [7], were developed to support the incremental and coordinated evolution of requirements and design. In these systems, the user is a requirements analyst and the system serves the role of the software developer. The initial user requirements lead IDeA to select an appropriate design schema from its library and, through continued interaction, this schema is refined into a complete design. The system also notes when there are mismatches to be resolved and further refinements to be made and stores these on a goal agenda. The users are not exploring designs as in Comet – designs are always developed in a top-down fashion, and a requirement cannot be retracted once it has been specified. In Comet, a design choice can always be changed and the user can begin the design process from a very abstract module or an entire implemented system. Later extensions to Rose-1 (see [8]) support this kind of flexibility through truth maintenance and hypertext mechanisms still lack the range of reasoning support for determining the ramifications of system modifications that Comet provides.

Another recent knowledge-based system, LaSSIE [2], promotes software reuse by providing multiple viewpoints of modules, including architectural, domain, and code perspectives. Like Comet, LaSSIE relies on a description logic system, KANDOR for representing knowledge about software. Domain actions are represented in terms of their superclasses, actors, agents, operands, and the state changes they produce. LaSSIE exploits classification to provide candidate responses to user queries: module descriptions in queries are classified within the domain taxonomy, and all subsumees are retrieved for possible further examination. LaSSIE is primarily oriented around a higher level description of domain actions and is used as an information resource which users can query for reuse candidates, examples, or further module details. The system has more detail on the actual code, as well as automated extraction mechanisms [16] for obtaining code knowledge from modules.

LaSSIE differs from Comet in its representation of modules and in the support it provides to users. In LaSSIE, actions are represented by simple slots defining the roles of the action, whereas Comet includes a detailed formal description of behavior. Comet uses its compositional behavior representation to support retrieval of potential reuse candidates in response to edited behavior descriptions. Reasoning in terms of behavior descriptions also enables some of its more subtle feedback about module commitments and inconsistencies in the design. LaSSIE offers the user no explicit assistance when retrieving modules concerning which are most suitable in the current design nor how the module might need to be altered to fit into that design – the same module will always be retrieved for a given query, no matter where in the current design it might be used. There are also differences in the reasoning of the systems due to the KANDOR language, e.g., constraints between roles within a given module cannot be stated. Furthermore, in comparison to Comet, the system is designed primarily to aid the developer rather than the designer; as such, it has more detail on the actual code, as well as automated extraction mechanisms [16] for obtaining code knowledge from modules.

The Programmer's Apprentice [14] is an intelligent computer assistant that can aid the programmer in constructing a software system. This work has relied primarily on the Plan Calculus [13] and the CAKE knowledge representation and reasoning system. The Plan Calculus provides an abstract, plan-based view of a body of software code, while CAKE supplies a layered reasoning system for making inferences at the levels of propositional logic, algebraic reasoning, frames, and the Plan Calculus itself. This approach has resulted in the development of a specialized editor, KBEmacs [17], which allows programmers to develop their programs using both plans and program text. KBEmacs automatically does translation and updating between these two forms of representation. There has also been work on a Requirements Apprentice that supports the generation of formal specifications from informal requirement descriptions based on a library of clichés capturing common concepts employed in a given domain. The clichés are themselves represented in the Plan Calculus. In comparison with Comet, the Programmer's Apprentice work differs in its strong orientation towards programming. It seeks to abstract away the canonical form of programs and enable pseudo-natural language interactions which can refer to parts of a program directly. However, this does not overcome the problem of describing and reasoning about a system in domain-oriented terms. Rather, it focuses more on easing the implementation burden on the programmer (synthesis support) without the kind of high-level design support offered by Comet. One could imagine, however, extending the notion of clichés into the design realm to bridge this gap between requirements and implementation.

VIII. Status

Comet runs on a Sun Sparcstation. It is based on LOOM and on interface software previously developed for the LEAP automatic programming system [4]. A taxonomy of generic behavior primitives has been built, as discussed in Section III.B. These were specialized into some ninety-six primitive behaviors for the tracker domain, which, along with sixty-five datatype descriptions (tracks, contact, etc.), form the core knowledge base for this domain. These primitives were then composed to model actual Lockheed tracker code.

This initial system is currently serving as an experimental prototype, used to guide further development of the representation and reasoning components, as well as to experiment with user interface techniques for presenting detailed guidance to the users.

IX. Discussion

The development and use of the Comet system has caused us to address issues that we believe are of concern to all knowledge-based approaches to supporting software development. This section briefly outlines our "design philosophy" with respect to some issues of efficiency, knowledge acquisition, and associating formal software descriptions with actual code.

A. *Efficient Reasoning*

The key reasoning support offered by Comet, finding relevant module descriptions, is a form of description classification: the system must determine which module description terms are subsumed by the module description term of interest. After much study in this area (e.g., see [12]), we know that any reasonably expressive description representation language will not allow complete, tractable classification reasoning. So we are in the usual bind: the module description language must be expressive enough to encode the fine shades of meaning that can differentiate potential substitute modules from inappropriate candidates, but the system must be able to rapidly discover at least most of the right candidates most of the time.

LOOM follows the tradition [15,18] of allowing the expression of terms whose subsumption relationships cannot be automatically determined by the system. This leaves open the question of how much reasoning is to be done in addition to the subsumption reasoning provided by LOOM, and how it is to be structured with respect to the reasoning provided by LOOM.

Comet uses the idea of commitments to break the overall subsumption reasoning problem into tractable chunks, and to provide a clean interface to LOOM. The LOOM classifier automatically determines the subsumption relationships of each new description to the extent that it can, given (efficiency-based) limitations on its ability to deal with some stated constraints and given that primitive terms play a prominent role in module descriptions.

Comet uses LOOM classification to determine a set of modules that could possibly be substituted for the new module description on the basis of their structure and behavior. Comet then takes this set, determines their commitments, and displays them for the user. This can be seen as a form of "residue" reasoning [3]: each module is appropriate (i.e., can be substituted into the design) if its commitments are met.

Commitments are by definition determined with respect to a single module. They are expressed in terms of relationships with other module descriptions, which in turn may have commitments that must be met. However, Comet does not explore all of these ramifications at once; the user is responsible for choosing to examine each of the module descriptions in turn. The reasoning task is thus broken into chunks, exploiting the modularity of the design. Reasoning is also paced to user interaction, and takes advantage of the users' reasoning capability: some commitments may not be worth exploring for reasons that the user understands but the system does not.

B. *Going Beyond the Built-In*

Except for the built-in core, Comet's knowledge base of module descriptions grows automatically as a side-effect of using the system for software development. Developers use Comet for support in analyzing the effect of design decisions, not as part of a knowledge acquisition scenario. However, as developers modify module descriptions to meet changing requirements, the system automatically acquires knowledge by relating each new module description to the module descriptions it already knows about. The developers do not need to be concerned with this process: they just use the system to do their job.

New module descriptions must be specializations of known ones. As was pointed out earlier, since the descriptions are hierarchically organized at different levels of detail, this is not a serious restriction: it is easy to specialize a very general type. On the other hand, the more specific the description that is specialized, the more the system will know about it, and the more helpful it will be to the developers in determining commitments. We believe that this will encourage specialization at the most specific level possible.

C. *Getting Down to Code*

Comet does not guarantee that the module descriptions it manipulates can be realized as working code. For those module descriptions that are associated with code, Comet does not guarantee that the code correctly implements the module as described. Besides being unavoidable, we do not see this as a serious problem.

The primary goal of Comet is to encourage the use of existing assets in software development. Given the oft-quoted proportions of effort in system development, the most valuable unused assets are undoubtedly previously proven designs, not code. Nonetheless, the ultimate goal of software development is working code. The eventual goal for Comet is to automatically generate code from the module descriptions, and indeed, we are currently working to use Comet module descriptions as input to an automatic programming system [4]. But even in the immediate term we believe that the association of code with descriptions will become quite accurate via the cumulative effects of reuse: module descriptions with incorrect implementations will soon be detected and weeded out. Bugs occur in hardware modules too, but the continual use of the same modules in many designs results in increasingly bug-free modules.

X. Acknowledgements

The authors acknowledge the very significant contributions of Alan Teague in implementing the user interface software, Brian Livezey in designing the module description language, and Sukesh Patel, Lori Ogata, and Rich Baxter in analyzing actual tracker software. We also thank Robert MacGregor for descriptions of LOOM reasoning and the referees for their detailed suggestions.

XI. References

[1] A. J. Czuchry, Jr. and D. R. Harris, "KBRA: A New Paradigm for Requirements Engineering", IEEE Expert, Vol. 3, No. 4, pp. 21-35, 1988.

[2] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard, "LaSSIE: A Knowledge-Based Software Information System", Communications of the ACM, Vol. 34, No. 5, pp. 35-49, 1991.

[3] J. Finger and Genesereth, M., RESIDUE: A Deductive Approach to Design Synthesis, Stanford Heuristic Programming Project Memo HPP-85-1, Stanford, CA, 1985.

[4] H. Graves, "Interactive Design in LEAP", Proceedings of AAAI Workshop on Automated Software Design, Anaheim, CA, pp. 173-182, 1991.

[5] W. L. Johnson, "Overview of the Knowledge-Based Specification Assistant", Proceedings of the 2nd Annual Knowledge-Based Software Assistant Conference, Rome, NY, 1987.

[6] W. L. Johnson and D. R. Harris, "Requirements Analysis Using Aries: Themes and Examples", Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference, Rome, NY, pp. 121-131, 1990.

[7] M. D. Lubars, "A General Design Representation", Technical Report STP-066-89, MCC, Austin, TX, 1989.

[8] M. D. Lubars, "A General Design Representation - Representing Design Dependencies in the Issue-Based Information Style", Technical Report STP-426-89, MCC, Austin, TX, 1989.

[9] M. D. Lubars and M. T. Harandi, "The Knowledge-Based Refinement Paradigm and IDeA: Concepts, Limitations and Future Directions", Proceedings of the 1988 AAAI Workshop on Automating Software Design, 1988.

[10] R. MacGregor, "The Evolving Technology of Classification-based Knowledge Representation Systems", in Principles of Semantic Networks: Explorations in the Representation of Knowledge, John Sowa (ed.), Morgan Kaufmann, San Mateo, CA, 1990.

[11] W. Mark, "Software Design Memory", Proceedings of AAAI Workshop on Automated Software Design, Anaheim, CA, pp. 115-120, 1991.

[12] P. Patel-Schneider, "Undecidability of Subsumption in NIKL", Artificial Intelligence, 38(3), 1989.

[13] R. C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", Proceedings of the Seventh International Joint Conference on AI, pp. 1044-1052, 1981.

[14] R. C. Rich and R. C. Waters, "The Programmer's Apprentice: A Research Overview", IEEE Computer, Vol. 21, No. 11, pp. 10-25, 1988.

[15] J. Schmolze and W. Mark, "The NIKL Experience", Computational Intelligence, 7 (2), pp. 134 - 159, 1991.

[16] P. G. Selfridge, "Integrating Code Knowledge with a Software Information System", Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference, Rome, NY, pp. 183-195, 1990.

[17] R. C. Waters, "The Programmer's Apprentice: A Session with KBEmacs", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, pp. 1296-1320, 1985.

[18] Woods, W. and Schmolze, The KL-ONE Family, Harvard University Aiken Computation Laboratory TR-20-90, Cambridge, MA, 1990.