reprinted from
IEEE EXPERT
Intelligent Systems and their Applications

Imbedding Rule Inferencing in Applications

David W. Franke
Microelectronics and Computer Technology Corporation

Many opportunities exist to integrate rule-based problem solving with "conventional" application code. For example, rule-based implementations of consistency and completeness checking, style and method critics, or data querying can enhance applications and, at the same time, provide an easily extensible (checking, critic, or query) facility. However, the preferred languages for applications and for rule-based problem solvers present a barrier to their integration and, consequently, a barrier to the widespread use of rule-based problem solvers embedded in applications.

The current scenarios used to achieve this integration are (1) to write the application in the rule language, or (2) to transfer (and hence translate) relevant data between the application and the rule-based problem solver. This article describes how features of object-oriented languages can facilitate the integration of rule inferencing and application code. The inferencing capability can remain general rather than application specific, and the application code requires only minor changes to accommodate the inference engine. The domain of electrical CAD constitutes our example application area.

Numerous applications of rule-based problem solving exist in the electrical CAD domain.[1] Researchers and commercial developers have implemented rule-based design critics that examine aspects of circuit design — for example, testability or adherence to electric or logic design rules. However, a barrier hinders further realizing the utility of rule-based problem solving in CAD tools. This barrier arises from the goal to create CAD tools with maximum performance. Often, this goal has been realized by implementing these tools in C or C++,[2] with data structures optimized for size (that is, as small as possible) and for tool operations. For example, the data organization most suitable for schematic capture or circuit display may not be the most suitable for simulation or timing analysis. This barrier has impeded the use of rule-based techniques in CAD tools.

To apply rule-based problem solving to CAD problems, the tool must be implemented in the rule language (for example, Prolog,[3] Proteus,[4] or OPS5[5]), or the design representation must be translated first from the tool representation to the rule language and subsequently back into the tool representation (if the rule-based problem solver modifies the design in any way). Consequently, rule-based problem solvers occur as independent steps in the design process, as opposed to being integrated in existing design activities (in schematic capture, for instance).

0885/9000/90/1200-0008 $1.00 © 1990 IEEE

The CAD Inference Engine (CADIE) implements a rule-inferencing capability intended to be embedded in CAD tools. The primary motivation behind CADIE is the ability to tightly integrate a rule-based inferencing capability with CAD tools developed in an object-oriented language, specifically C++. CADIE examines tool data directly, thereby avoiding translations and enabling the rule-based problem solver to be integrated into a tool that supports an existing design activity. CADIE accomplishes this access without additional data fields in tool-defined data structures. The program achieves integration via features of object-oriented languages that (1) enable the inference engine to remain application independent, and (2) require only minor changes to application code and data structure definitions. This article describes the object-oriented design of the inference engine and the interface between the inference engine and tool-defined data structures.

Rules and assertions

An assertion represents ("asserts") a fact to be considered during inferencing. While assertions sometimes represent other information (for example, how to destructure information represented as a list, or the termination condition of a recursive rule), the primary role of assertions is representing facts. For example, the assertions in Figure 1 state that a signal *clock* and a transition *t1* exist. Further, *t1* represents a transition of *clock* at time 110 to the value 1. CADIE uses a rule and assertion syntax adopted from Proteus.[4]

A forward rule specifies facts to add to the database when a set of preconditions becomes true. For example, the forward rule

```
((data  ?d)
 (clock ?c)
 (connected ?d ?c)
 —>
 (connection-error ?d ?c))
```

states, "If a data signal and a clock signal are connected, then add to the database the fact that a connection error exists between these two signals." The syntax *?x* specifies a variable.

A backward rule specifies a set of conditions that, if proved true, support the truth of its predicate (consequent). For example, the backward rule

```
((signal-value ?signal ?time ?value)
 <—
 (transition ?t)
 (signal ?t ?signal)
 (time   ?t ?time)
 (value  ?t ?value))
```

```
(signal clock)
(transition t1)
(signal  t1 clock)
(time    t1 110)
(value   t1 1)
```

Figure 1. Example assertions.

states that a signal value and the associated transition time can be determined from a transition involving that signal.

Objects as assertions

CADIE's unique feature lies in its integration of tool data structures with rules and assertions. The object-centered representation of tool data is simply an alternative organization for information that can be expressed as assertions.[6] Knowledge representation systems often employ the frame representation paradigm, integrated with rules and assertions. The term frame is not used here, since explicit frame representations usually include uniform inference mechanisms (for example, inheritance). While implicit inference mechanisms can be provided in tool data implementations, they need not be uniform across all tool data structures and will be hidden (via data abstraction) from the inferencing mechanism of CADIE.

Consider a tool data structure for a signal transition with data members *signal*, *time*, and *value* (the sidebar at the end of this article defines C++ terminology). Consider a specific instance of the transition data structure, *t1*, with these data member values:

signal: <*clock*1>
time: 110
value: 1

If represented explicitly by CADIE, this data would be expressed by the assertions in Figure 1. CADIE enables the tool-defined data structure (presumably optimized for tool use) to be considered in inferencing without translating the data into the assertion format.

An object-oriented implementation

To understand the integration technique, we first describe the main concepts of the inference engine design. In examining an object-oriented design, one must consider the (conceptual) objects of the discourse, or the ontology; the objects of the implementation expressed in an object language; and the operations supported by those objects.

The first obvious conceptual objects are rules and assertions, and a database in which they reside. In decomposing these objects, we need to describe predicates (known as structures in Prolog), constants, and variables. We also define several specializations of constant, including symbol, string, and integer. In addition to these objects, which capture static concepts, we add the object

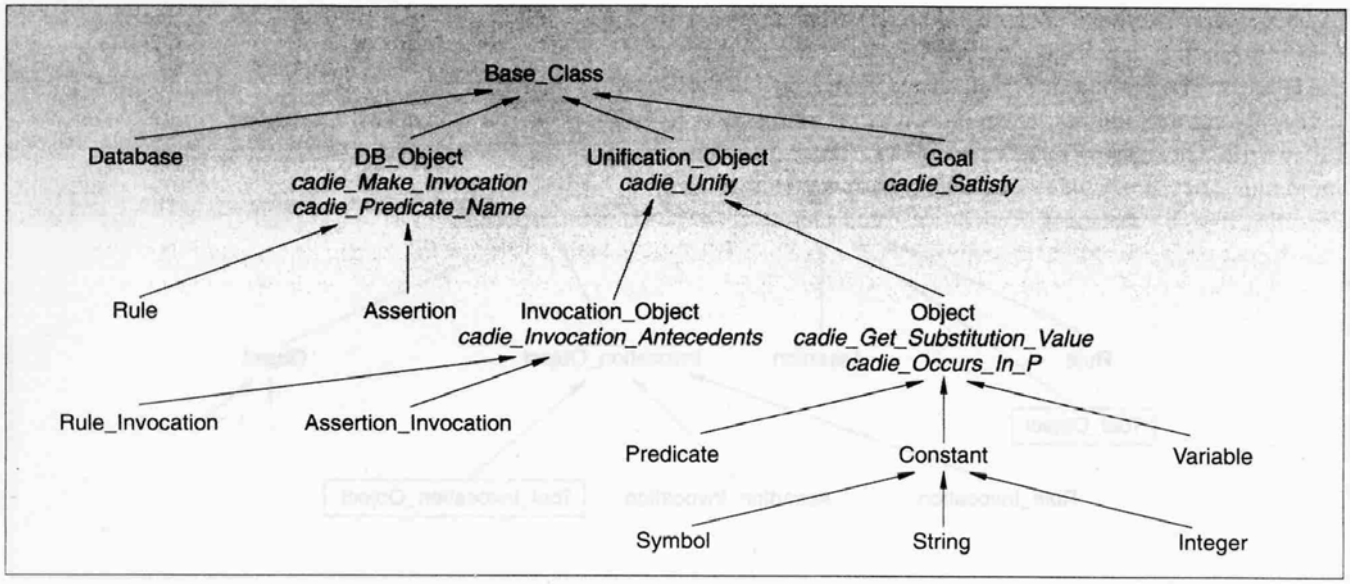

Figure 2. The CADIE class hierarchy.

While candidates (rules and assertions) remain in the database
1. Attempt unification of the goal predicate with the next candidate (via the member function *cadie_Unify*)
2. If unification is successful and the candidate is an assertion
   —Return variable bindings and report success
3. If unification is successful and the candidate is a rule
   —Recursively process subgoals (backward-rule antecedents)
   —If all subgoals are proved, return variable bindings and report success

Figure 3. The basic goal-proving algorithm.

```
class tool_Transition : public cadie_Tool_Object {
  tool_Signal *signal;
  int         time;
  int         value;
 public:
  /* Tool-defined member functions ... */
  virtual char **cadie_One_Place_Predicate_Names();
  virtual char **cadie_Two_Place_Predicate_Names();
  virtual int    cadie_Unify_Two_Place_Predicate(cadie_Object*, cadie_Substitution*, int);
};
```

Figure 4. An augmented class definition.

goal, which represents both a top-level goal or query and the intermediate goals (subgoals) developed in satisfying that top-level goal. Figure 2 shows the class hierarchy defined for these objects in the CADIE implementation. Class names in the code examples use these names with the prefix *cadie*. In this implementation, the objects expressed in the object language represent one-for-one the conceptual objects described above.

In the context of this class hierarchy, we define functions required to implement the inference engine. For example, the function *cadie_Unify* is declared for class *Unification_Object*, and class-specific definitions of this function occur in each class derived from this class. Figure 2 also presents several other functions of CADIE.

With these objects and functions, we can now define the basic algorithm for satisfying a goal via backward chaining. The algorithm shown in Figure 3 has been simplified to facilitate understanding how integration occurs, and is not the complete algorithm used in the CADIE implementation.

With respect to integrating instances of tool-defined objects with the inference engine, the interesting step of the algorithm to examine is step (**1**). A unification candidate (a rule, an assertion, or a tool-defined object) must support unification. Further, a unification candidate must be able to create invocations of itself, since it may be used more than once in proving a goal. (An invocation comprises a reference to the unification candidate and associated variable bindings that result from unification). For a tool-defined object, CADIE accomplishes this by providing the class *Tool_Object*. The tool developer includes this class in the class definitions of the tool-specific

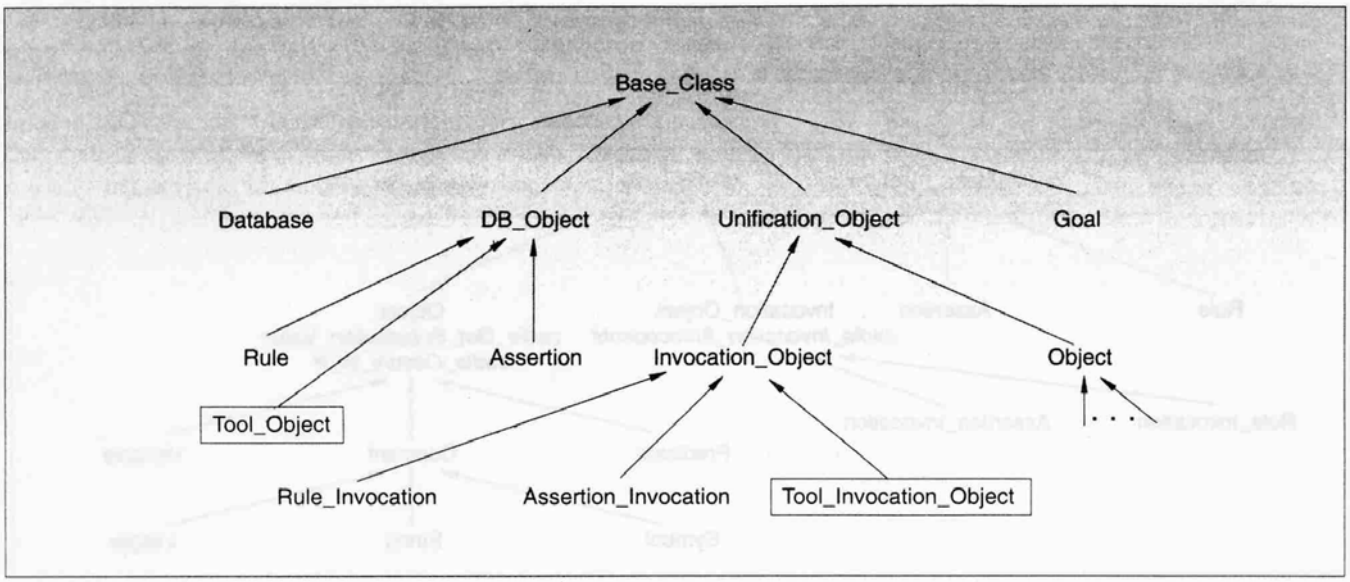

Figure 5. A class hierarchy with tool support classes.

objects that are of interest in the rule-inferencing application. That is to say, the tool-specific class is derived from the CADIE-supplied class *Tool_Object*. This derivation can be at the base of the tool's class hierarchy, or introduced at some intermediate level of the tool's class hierarchy via multiple inheritance. Figure 4 illustrates an example derivation of a tool-defined class from the CADIE-supplied class.

Object/inference-engine integration

To integrate tool-defined objects with CADIE, we add the classes *Tool_Object* and *Tool_Invocation_Object* to the class hierarchy as shown in Figure 5. These classes define member functions required by the inference engine, thereby providing transparent access to tool-defined classes derived from *Tool_Object*. These classes enable the inference engine to access tool-defined objects as if they were expressed as assertions. This interface or protocol definition capability, along with object-type-specific function invocation, constitute the essential features of an object-oriented language that enable the integration described here. In the context of object-oriented programming, the ability of different objects to respond to the same interface (protocol) is called polymorphism.[7] CADIE's object-oriented-protocol approach is similar in spirit to the "protocol of inference" approach in Joshua,[8] although CADIE does not attempt to span the same range of operations.

The object/inference-engine interface. To support unification, the *Tool_Invocation_Object* class redefines the function *cadie_Unify*. Class-specific behavior for the *cadie_Unify* function is achieved via the following virtual-member functions declared by the class *Tool_Object*:

- *cadie_One_Place_Predicate_Names*
- *cadie_Two_Place_Predicate_Names*
- *cadie_Unify_Two_Place_Predicate*

These functions are utilized by the *cadie_Unify* function of the class *Tool_Invocation_Object*, and class-specific definitions of these functions occur in each tool-defined class derived from class *Tool_Object*. The first two functions define the predicate names that objects of the derived class will recognize (that is, the predicates with which they can potentially unify). One-place predicates have the form

(<predicate-name> <object>)

where <predicate-name> is a type for <object>. Two-place predicates have the form

(<predicate-name> <object> <value>)

where <predicate-name> is an attribute name of <object>, whose value is <value>. Unification of a tool-defined object with a one-place predicate is straightforward, and verifies that

(1) The object recognizes the predicate name; and

(2) <object> is the specific tool-defined object, or a variable to which the tool-defined object can be bound.

Unifying a tool-defined object with a two-place predicate is also straightforward, requiring verification that

(1) The object recognizes the predicate name;

(2) <object> is the specific tool-defined object, or a variable to which the tool-defined object can be bound; and

```
(tool_Transition
  (transition)
  ((signal cadie_Object* 'signal')
   (time   integer       'time')
   (value  integer       'value')
  ))
```

Figure 6. *tool_Transition* predicate declarations.

(3) <value> matches the appropriate attribute value of <object>, or is a variable to which the appropriate attribute value of <object> can be bound.

A class declaration example. Consider the tool class *tool_Transition* representing a transition in simulation output. The class definition, after being augmented with information to be used by CADIE, is shown in Figure 4.

Possible predicate names for the *tool_Transition* class are *transition* for one-place predicates, and *signal*, *time*, and *value* for two-place predicates. The value associated with the predicates *time* and *value* would be integers denoted by the corresponding members of the class instance. The value of the predicate *signal* could be a pointer to an object representing a signal from which additional information can be obtained, or possibly a string denoting the signal's name.

Inference engine commands. To complete the integration example, we need inferencing commands that the tool can issue to initiate inference engine activity. Initiating inferencing can occur as a result of an explicit request by the tool user, or an internal (implicit) action by the tool. For example, a schematic-capture tool could provide consistency and completeness checks once a design has been entered. At the request of the designer, the tool could verify that all ports of all components are connected to signals or other ports, that clock signals are not connected to data signals, and that no direct connections exist from power to ground. In addition to this explicitly initiated checking, the tool could be performing implicit inferences after each interaction with the user. For example, a designer could specify constraints for the design, such as total space available or maximum delay allowed. If a constraint is violated at any point while the designer is entering or modifying the design, the tool can notify the designer at that point — rather than after the complete design has been entered. A rule-based approach to constraint checking, as opposed to an algorithmic approach, may be necessary for incomplete designs or for designs in which component size or speed have not been completely characterized.

To this end, CADIE defines the following functions:

- *cadie_Load* — Load the source file of rules and assertions;
- *cadie_Add_Object* — Add a tool-defined object (that is, an assertion) to the database;
- *cadie_Make_Goal* — Create a goal or query;
- *cadie_Query* — Search for the next solution of the goal, via backward chaining; and
- *cadie_Fire* — Fire a forward rule.

Forward inferencing. When tool developers or users introduce forward rules into the database, the forward-rule antecedents are unified with assertions in the database. Hence, any tool-defined objects in the database will also be considered in forward inference. If CADIE can satisfy all antecedents of a forward rule with consistent variable bindings, the rule is "ready to fire." All such rules make up the conflict set, from which the tool can select and fire specific rules. Firing a forward rule asserts all the rule consequents into the database and removes the rule from the conflict set.

When considering tool-defined objects in the context of forward-rule consequents, a one-place consequent (that is, a predicate) of a forward rule specifies a (possibly new) type name for the corresponding object. Hence, if a forward rule fires and has the consequent

(<type-name> <object>),

then <object> should now recognize one-place predicates with the predicate name <type-name>. A two-place consequent (a predicate) of a forward rule specifies a value for the corresponding attribute of the object. Hence, if a forward rule fires and has the consequent

(<predicate-name> <object> <value>),

this results in modifying the attribute denoted by <predicate-name> of the object <object>. This requires additional functions to be supported by the tool-defined classes, namely

- *cadie_Apply_One_Place_Predicate*
- *cadie_Apply_Two_Place_Predicate*

The implementation for a specific attribute dictates the appropriate implementation of the function. For example, the consequent

(*time* <transition-object> *100*)

can result in setting 100 as the value of member *time* of object <transition-object>, while the consequent

(*signal* <transition-object> *"carry"*)

can result in setting *"carry"* as the value of the *name* member of the *tool_Signal* instance referenced in the *signal* member of <transition-object>. This is a consequence of the data abstraction capabilities of the object language. Class- and attribute-specific code can also handle multivalued attributes, enabling new attribute values (asserted as the result of forward-rule firing) to be

added to a set of values as opposed to replacing a single value.

Forward rules can also specify negated consequents (predicates). For one-place predicates, a negation can remove a specified type name from the set of type names recognized by the specific object. For two-place predicates, a negation can remove the specified value from the set of values of the specific object's attribute, or it can remove a single value of the attribute. If the single value or the last value of the set is removed, the object will no longer unify with a two-place predicate whose predicate name signifies that attribute. For example, a tool maintaining delay information on a circuit under design might represent delay values as unknown (no value), an upper and lower bound (a list or a pair of values), or a single value. If the circuit is modified so that a previously estimated delay interval is no longer valid, the corresponding upper and lower bounds must be removed, where the absence of a value indicates that the delay value is unknown. A forward rule realizing this action might resemble the following rule fragment:

```
( ...
 (delay ?path ?delay-value)
 —>
 (~ (delay ?path ?delay-value)))
```

Additional member functions that handle negated forward-rule consequents include

- *cadie_Apply_Negated_One_Place_Predicate*
- *cadie_Apply_Negated_Two_Place_Predicate*

Code generation

The tool and inference engine integration described to this point requires tool developers to write the member functions used by CADIE. To avoid errors and to ease the requirements placed on developers, CADIE provides a preprocessing utility that takes a concise description of the predicates that a class instance recognizes and generates the appropriate C++ code. Developers can include this code at the appropriate location in the tool code.

The preprocessor requires the following information:

- The class name;
- One-place predicate names;
- Two-place predicate names;

```
char *cadie_One_Place_Predicate_Names_For_tool_Transition[ ]=
{"transition," NULL};
char *cadie_Two_Place_Predicate_Names_For_tool_Transition[ ]=
{"signal," "time," "value," NULL};

char **tool_Transition::cadie_One_Place_Predicate_Names()
{ return cadie_One_Place_Predicate_Names_For_tool_Transition;}

char **tool_Transition::cadie_Two_Place_Predicate_Names()
{ return cadie_Two_Place_Predicate_Names_For_tool_Transition; }

int tool_Transition::cadie_Unify_Two_Place_Predicate(
 cadie_Digested_Object *candidate,
 cadie_Substitution *substitution,
 int predicate_Index)
{
 switch (predicate_Index) {
 case 0: { /* signal */
  cadie_Arbitrary_Object *cadie_Value_Of_Predicate_signal = signal;

  if (cadie_Value_Of_Predicate_signal == NULL) return FALSE;
  return candidate -> cadie_Unify(cadie_Value_Of_Predicate_signal,
                substitution); }
 case 1: { /* time */
  cadie_Digested_Integer_Constant *cadie_Value_Of_Predicate_time;

  cadie_Value_Of_Predicate_time = cadie_Make_Integer_Constant (time);
  return candidate -> cadie_Unify(cadie_Value_Of_Predicate_time,
                substitution); }
 case 2: { /* value */
  cadie_Digested_Integer_Constant *cadie_Value_Of_Predicate_value;

  cadie_Value_Of_Predicate_value = cadie_Make_Integer_Constant (value);
  return candidate -> cadie_Unify(cadie_Value_Of_Predicate_value,
                substitution); }

 }
}
```

Figure 7. The preprocessor output.

- For each two-place predicate name
  - — The type of value returned; and
  - — The expression that retrieves the value for the attribute.

Consider the class *tool_Transition* in Figure 4. A preprocessor input specification for this class is shown in Figure 6, and Figure 7 presents the preprocessor output.

Suppose that the tool developer wanted to retrieve the signal name and type via the transition object itself, as opposed to retrieving the signal object and then extracting the name and type. Figure 8 presents the preprocessor input denoting this object description.

```
(tool_Transition
 (transition device)
 ((signal cadie_Object* 'signal')
  (time   integer       'time')
  (value  integer       'value')
  (name   string        'signal -> name')
  (type   symbol        'signal -> type')
 ))
```

Figure 8. Alternate *tool_Transition* predicate declarations.

The integration approach provided in CADIE enables a rule-inferencing capability that operates directly against tool-defined data structures with no additional data fields required in the tool-defined classes. The expense of integration occurs in the number of member functions associated with the tool-defined classes. This integration capability encourages the use of rule-based problem solving in conventional applications that are implemented in object-oriented languages, including C++. It does so by overcoming performance and size penalties imposed by alternative integration techniques such as translation.

Acknowledgments

I would like to thank Ramon Acosta and David Burgess for their comments and suggestions, and Bill Read and Jerry Sullivan for their support in this effort. I also thank Merrill Cornish for his insights into the concepts of object-oriented program organization.

C++ terminology

As is the case with many languages, C++ has language-specific terminology for concepts that are known under different names in other languages. The C++-specific terminology for object-oriented concepts is described here and is related to terminology used in other object-oriented languages.

C++ uses the term class in the same manner as other object languages — as a description or template for similar objects. In C++, class specialization is called derivation, with a derived class specializing a base class. The C++ derived class corresponds to the term subclass. C++ class derivation provides single and multiple inheritance.

The C++ term data member refers to the data elements of a class instance, or object. The C++ (data) member corresponds to the instance variable, slot, or attribute term used in other object languages. Class variables can be defined in C++ and are called static data members. The visibility of data members can be controlled in C++ code by the specifications public, protected, and private.

A protocol is a set of messages (that is, an interface) supported by one or more classes. Protocols for C++ classes are defined via member function declarations of the classes. The C++ member function implementation corresponds to a method, as defined in other object languages. As with data members, the visibility of member functions can be specified as public, protected, or private. To achieve class-specific implementation of a C++ member function declaration (that is, a message), the member function is declared as virtual. This class- specific behavior of object-oriented languages is called polymorphism, and is an essential feature of these languages. Declaring a member function as virtual in a base class simply states to the compiler that the details of the member function implementation may not be known until the compiler processes classes derived from the base class. At runtime, the object type determines the specific member function implementation to execute.

References

1. T.J. Kowalski, "At the Cross Roads: Looking Both Ways," *Proc. IFIP TC 10/WG10.2 Working Conf. on CAD Systems Using AI Techniques*, Organizing Committee of CAD Systems Using AI Techniques, Tokyo, 1989, pp. 3-10.
2. S.B. Lippman, *C++ Primer*, Addison-Wesley, Reading, Mass., 1989.
3. W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1984.
4. C.J. Petrie, Jr. et al., "Proteus 2: System Description," Tech. Report AI-136-87, MCC, Austin, Texas, May 1987.
5. C.L. Forgy, "The OPS5 User's Manual," Tech. Report CMU-CS-81-135, Computer Science Dept., Carnegie Mellon University, Pittsburgh, 1981.
6. N.J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing, Palo Alto, Calif., 1980.
7. M. Stefik and D.G. Bobrow, "Object-Oriented Programming: Themes and Variations," *AI Magazine*, Vol. 6, No. 4, Winter 1986, pp. 40-62.
8. S. Rowley, H. Shrobe, and R. Cassels, "Joshua: Uniform Access to Heterogeneous Knowledge Structures, or Why Joshing is Better than Conniving or Planning," in *Proc. Sixth AAAI*, MIT Press, Cambridge, Mass., 1987, pp. 48-52.

David Franke is a senior member of technical staff for the MCC CAD program. Prior to joining MCC, he worked on operating systems, computer architecture, and AI applications at Texas Instruments. Having received his BS in mathematics from the University of Oklahoma and his MS in computer science from Pennsylvania State University, he is now completing the PhD program in computer science at the University of Texas. His research interests include design knowledge representation, design reuse, and model-based reasoning. He is a member of the IEEE Computer Society, AAAI, ACM, and Upsilon Pi Epsilon.

The author can be reached at MCC, 3500 W. Balcones Center Drive, Austin, TX 78759; email franke@mcc.com.