Venus: An Object-Oriented Extension of Rule-Based Programming

Daniel P. Miranker, Lance Obermeyer*, Lane Warshaw* and James C. Browne

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

*Applied Research Laboratories
The University of Texas at Austin
Austin, Texas 78713

Abstract

Declarative programming, in the form of forward-chaining rule languages, offers advantages complementary to procedurally based object-oriented programming languages. The Venus rule language addresses certain deficiencies with rule-based programming by adding object-oriented features to the rule paradigm. This differs from most other efforts at hybridizing the two paradigms. Most other efforts seek advantage by starting with an object-oriented language and adding rule facilities. Object-oriented features of the Venus language includes encapsulation of parameterized rule modules such that the behavior of a rule group may be defined using inheritance and improved embeddability via resolving polymorphism.

An implementation of a large subset of Venus has been exploited for the implementation of benchmark programs and applications. Experience with the application of Venus as the basis of rule-based database query optimizers and a financial expert system demonstrates the reusability of Venus-rule modules. Quantitative software metrics suggest the object-oriented structures embodied in Venus yield a significant reduction in life-cycle costs of rule-based program development.

1 Introduction

A forward-chaining rule language consists of a set of "if then" rules, where the "if" part expresses a predicate over the program's state space. The "then" part consists of procedural actions. These actions generally modify the state of the program. Thus, a rule is a transition function from one area of a state space to another. The power of forward-chaining rule-languages is based in large part upon the opportunity to develop programs by defining behaviors piecemeal such that each rule, independently, specifies only a partial state description. In addition an execution framework integrates the set of rule definitions into a single, deterministic, transition function. The execution framework is commonly called an *inference engine*, since it determines the applicability of each rule at every execution step of a program.

The expression of programs as rules where all rules have access to the entire state space is clearly at odds with good programming practices. There is no information hiding between conceptually separate parts of the program. Every rule execution has the opportunity to behave like a *goto*. Since object-oriented programming precisely addresses the issue of encapsulation, there have been many efforts to integrate the rule-programming paradigm and the object-oriented paradigm.

Efforts at integrating the two paradigms date to the first expert-system shells [XER82, ART87]. These shells helped to popularize both the rule and object programming paradigms. In the LOOPS language, a

method within an object could be defined by either a procedural definition or as a collection of rules. Since then, efforts to hybridize the rule paradigm with the object paradigm continue to start with the syntax and semantics of a procedurally-based object-oriented language and add a rule facility [PAC94]. The efforts that have sought to introduce encapsulation directly into rule languages have simply added implicit and explicit procedural constructs with little or no consideration to larger issues in programming languages [BRO94].

This paper presents the Venus rule-based programming environment, with specific emphasis on the Venus language. The Venus system features the Venus rule language, a compiler for the language, a debugger, and an optimizer.

The Venus language is based heavily on C++. The intention is to reduce the learning curve for new programmers and to facilitate the development of embedded systems. In Venus, the basic program statement is the `rule`. A rule represents a single transition from one area of the state space to another. An important contribution of the Venus language is the `module` construct, which encapsulates a set of rules. Modules are parameterized by data collections. The actual parameters passed to a module at runtime determine the start state for its transition function. Since the modules are parameterized with respect to named collections of objects, the modules themselves do not contain state and thus only serve to define a state transition function. Thus a rule module definition is much like a class definition, since it is separate from instances of the modules.

The semantics of module composition in Venus is declarative in nature, not procedural. In Venus the rules of a module fire until the module reaches a fixed point. Control may move from one module to a subordinate module, in which case the semantics embrace a nested model of the state-space. Once a subordinate module begins execution it will not exit until it reach a fixed point within its nesting. Control then returns to the superior module. Thus, there is a close correspondence between the firing rules for Venus modules and a distributed implementation of the state machine model used for specification of flow of control in the popular models for object-oriented analysis[Jim, give us a citation].

The ability to parameterize rules in this fashion is unique to Venus. It also follows from this structure that rule modules may be defined by exploiting inheritance. Specifically, consider that each rule in the module contributes to the definition of the module's behavior similar to the way each method in a class contributes to the definition of the object. Just as methods may be inherited from a base class, Venus modules may be defined by inheriting rules from other modules. Similarly, since modules are defined using formal arguments and the actions taken depend upon the actual arguments there is an opportunity to introduce polymorphism. See section 3.

A significant subset of Venus has been implemented. Venus has been used to implement several applications and benchmark programs. We discuss two of these applications as a concrete basis to illustrate and validate the concepts in this paper. These applications are a financial program, a version of which is in commercial use, and several related implementations of database query optimizers. Development and evaluation of these applications reveals clear opportunities for reusability and other software engineering advantages that are a direct result of Venus' object-oriented features. In the case of the financial program, quantitative software metrics suggests that the Venus implementation resulted in significantly better code than an earlier effort that used a traditional rule language.

2 Venus Overview

The Venus programming system includes all of the elements of a program development environment integrating object oriented and rule based programming. It consists of the Venus language, a compiler for the language that generates C++ code, a precompiled runtime library, and a tool suite.

2.1 C++ Heritage

The Venus language is syntactically modeled on C++, and retains C++ syntax wherever possible without introducing ambiguity or confusion. Data elements in Venus are defined as C++ classes and the data instances are C++ class instances. This eliminates the need for reformatting external data into an internal representation. It also allows inferencing on user defined data types. This is especially useful when adding rule based capabilities to existing programs.

```
class Relation
{
public:
      OID domain();
      OID range();
      void domain(OID);
      void range(OID);
};
```

Figure 1. Example Class Definition

The Venus language uses a C++ based syntax for expressing rules. Although a number of systems, including R++, Ral, RETE++, and CERS, use C or C++ to define data objects, only Venus and R++ embrace C++ syntax for the expression of rules[CRA96, FOR94, HAL96, MIR93]. The other systems continue to adopt syntax derived from LISP-embedded OPS-like languages. (i.e. extensive use of reserved lexical symbols, such as parenthesis and punctuation marks to simplify parsing).

2.2 Data

Venus supports two data types, containers and primitive variables. Containers represent set data, where primitive variables represent individual variables. It is often convenient for an application programmer to define a class such that a container represents the class's extent.

Containers are denoted by square brackets (`[]`), intentionally drawing upon the C++ syntax for arrays. Both arrays and containers associate multiple data elements with a single name. In C++, array elements are accessed directly by placing an index value between the square brackets. In Venus, container elements are accessed indirectly through cursors. As a declarative rule language, Venus manages the cursors, not the programmer. The programmer simply describes the desired cursor behavior based on the cursor type. This is conceptually similar to the behavior of SQL, another declarative language. In an SQL query, the query engine manages iteration over the base relations.

There are two cursor types, existential and universal. The cursor type is selected by inserting a quantifier between the square brackets. An existential cursor is denoted by a question mark (`?`) quantifier, and corresponds to a positive condition element (e.g. `container[?]`). A universal cursor is denoted by an asterisk (`*`) quantifier. Universal quantifiers are always assumed to be within the scope of existential quantifiers. Consequently the use of * usually corresponds to the use of negative condition elements in common expert system languages (e.g. `container[*]` is roughly equivalent to an `OPS5 (container)`).

Primitive variables are similar to containers, except they always contain one element. There is never any search required to find primitive variables. Therefore, their syntax is the same as C++ variables. They are slightly more efficient than containers, both in space and in time. This is because linked list node pointers are not required, and direct accessing can be used.

2.3 Rules

A Venus rule consists of a header, a guard, and an action. The combined guard and action is syntactically equivalent to a C++ `if` expression. The header specifies the rule name, an optional priority, and an optional `from` clause. The `from` clause is a syntactic shortcut borrowed from SQL. It allows the programmer to replace a container name and quantifier with a string. In Figure 2, the keyword `from` and

the asterisk declare `u` to be a universally quantified variable over the relation `r`. Similarly, the question mark declares `e` to be an existentially quantified variable. This style is closer to first-order logic notation than traditional rule languages.

```
// for all u in r, if there does not exist a
// symmetric element e, then create it and
// add to the relation r
module enforce_symmetry(Relation r[])
{
      rule enforce;
      from r[?] e;
      from r[*] u;
      if(!(u.domain() == e.range() &&
          (u.range() == e.domain())) {
            Relation i;
            i.domain(e.range());
            i.range(e.domain());
            r.insert(i);
      }
}
```

Figure 2. Example Rule

A guard is defined by an `if` keyword, an opening "(", an expression, and a closing ")". There are few restrictions placed on the expression. The programmer may call externally defined C++ functions. These functions must be side effect free, and each must return a value. Parameters to these external functions must be passed by value. Note that these functions are directly called. Since Venus is a compiled system, a runtime name registration/lookup scheme is not required.

A guard expression may compare values from different containers. Traditional rule languages compare values using pattern-variables to bind individual attribute values. Venus, however, allows direct comparison of different values using relational operators. This is the familiar style for database languages such as SQL. Multiple tests on a single attribute in this style result in repeated exposition of a path expression.

Venus uses C++ as the action, or right-hand side (RHS) language. Venus places some restrictions on the actions a rule can execute. The compiler parses the RHS and automatically recognizes updates to to the state and inserts runtime calls notifying the inference engine. Externally defined functions may modify Venus variables, but the variables are restricted to ones within the scope of the module, and they must be modified using special Venus supplied functions. The compiler does not process external function definitions.

2.4 Modules

The unit of organization in a Venus program is the `module`. Although the notion of a parameterized subroutine was promptly introduced into procedural languages, some time passed for a rule-based analog to be proposed {tai94}. A module consists a formal parameter list, local variables, and rules. We have adopted C++ functional notation for module constructs, including curly braces to denote nesting program blocks. The scope of a module is limited to its actual parameters and local variables. To ease formal analysis, there are no global variables.

Figure 2 illustrates the code for a module called `enforce_symmetry`. This is from a stylized presentation of a system developed from a specification of a device-structure hardware diagnosis program [CRA96]. The module `enforce_symmetry` contains a rule, parameterized over a type called `relation`, that ensures that for all pairs (a,b) in the relation, a symmetric pair (b,a) is always also present. In the application, this is used to maintain lists of devices in a cluster that can serve as backups to each other. For example, if in the cluster machine *a* can backup machine *b*, then machine *b* can also backup machine *a*. By providing formal parameters to a module, a collection of rules can be "reused" for

slightly different circumstances. By this we mean that the module can be called once with one container as the actual parameter, and another time with a different container. This feature has been extensively used in all developed applications. Its use often leads to a significant reduction in the number of rules and complexity of the rules.

A Venus module will fire rules until a fixed point is reached. The entire RHS is treated as a single atomic action. Rules with the same priority, including multiple instantiations of the same rule, are selected for firing by a fair nondeterministic policy. For historical reasons, the current implementation dictates that fairness be defined such that a rule instantiation is fired at most once, where once has the OPS5 definition [FOR81].

```
module A(int P1[],P2[])
{
      int L[];

      rule r;
      from P1[?] cur;
      if(TRUE) print(cur);
}

module B(int P[])
: A(P,L)
{
      int L[]; // local

      rule s;
      from P[?] cur;
      if(TRUE) print(cur);

      rule t;
      from A::L[?] cur;
      if(TRUE) print(cur);
}
```

Figure 3. Module Inheritance

2.5 Scope

Venus, like most procedural languages, supports multiple binding scopes for variables. The first level is *module scoped*. These are formal parameter to a module or local variables declared within a module. In the above example, the container `r` is module scoped, and can be accessed by all rules in the module. These are visible to all rules in a module. Changes to module scoped variables can trigger rules. The second level is *rule scoped*. These are implicitly declared variables bound to cursors over containers. In the above example, the cursor `e` points to an element from the `r` relation. Elements defined by cursors can be accessed in both the guard and action portions of an individual rule. The final level is *action scoped*. These are variables defined and used only in the RHS of a rule. These variables do not trigger rule firings. In the example, the variable `i` is rule scoped.

Like C++ and unlike Pascal, Venus modules do not inherit variables from their caller. Venus does not support globally scoped variables.

2.6 Execution phases

There is a hierarchy of execution phases for a Venus module. These phases are shown in ???. The base phase is *inactive*. Modules that are out of scope are inactive. The second phase is *active*. In this phase, the module actively fires rules. Entering the active phase suspends the inactive phase. The third phase is *listening*. In this phase, a Venus module monitors changes to variables and queues rules for later firing. This phase begins when the RHS of a rule in the current module calls another Venus module, and ends when the called module returns. It is incorrect to think of a listening module as suspended. Rule firings are suspended, but monitoring and queuing functions are not. Entering the listening phase suspends the active phase. A sequential implementation will always have at most one active module. A parallel implementation may have many.

2.7 Module Semantics

A Venus program may consist of more than one module, and modules may be children of other modules. Module names and their actual parameters are listed in the action portion of a rule. If the rule is satisfied then rules in the embedded module may also fire. Modules may be embedded arbitrarily deep. The same

module with the same or different actual parameters may be activated from multiple places in the code. At this time the graph structure of the module hierarchy must form a static and directed acyclic graph.

```
module A(T t[], int x) {…}

module B(T t[]) {…}

module C(T t[])
{
      int x = 0;

      rule guard_A;
      if(TRUE) {
            A(t,x);
      }

      rule guard_B;
      from
      if(x==1) {
            B(t);
      }
}
```

Figure 4. Module Calls

It is convenient but incorrect to think of a module as being called in the usual sense. An obvious firing of the guard rule is not necessary for execution of rules in the child module. If the guard rule is not satisfied, the module will not be called. If the guard rule is satisfied, the module might be called. In regard to module calls, we extend the fairness policy. Whereas a normal rule will fire at most once on a single instantiation, a rule guarding a module call can potentially fire whenever there is a state change affecting the satisfaction of a rule predicate in the child module. Thus the data driven nature of Venus extends *through* the rule guarding a module call. This explains the behavior of rules with guards of `if(TRUE)`.

Similarly, a guard rule's predicate is **not** distributed over the rules in the guarded modules. As a consequence of the atomicity, even if the predicate is disabled during a child module's execution, that module continues to fire rules until it reaches a fixed point. Again, though this behavior is derived from formal declarative definitions, the ultimate execution is consistent with procedural intuition. If a C++ function is called in the `then` portion of an `if` statement, execution of the function does not stop if the guarding expression becomes false.

Figure 4, for example, illustrates the semantics of modules. Module `C` has a container over elements of type `T` as a formal parameter. Initially, the rule `C::guard_B` is unable to fire since its local variable `x` is initialized to 0. As noted above, evaluation will automatically pass to the `if(TRUE)` rule `C::guard_A`, causing evaluation of the `A` module with actual parameters `t` and `x`. Should that module set `x` to 1, then control will eventually pass to module `B`.

```
module A(int P[])
{
      int L[];
      rule r;
      …
}

module B(int P[])
{
      int L[];
      rule r;
      …
}

module C(int P[])
: A(P), B(P)
{
      rule r;
      from A::L[?] a;
      from B::L[?] b;
      if(a == b) …
}
```

Figure 5. Multiple Inheritance

3 Object Oriented Features

3.1 Inheritance

The Venus language supports inheritance of rules at the module level. Venus' inheritance syntax is modeled after that of C++. Figure 3 shows a simple inheritance example.

The semantics of inheritance is essentially rule and local variable inclusion. Rules defined in the base module are conceptually copied into the derived module. Rule priorities from the base and derived modules are numerically merged into a single priority ordering.

In the case of a parameter to a base module, the inheriting

module must explicitly assign either one of its parameters of the same type or one of its local variables of the same type. This is similar to the initialization of a C++ reference. Local variables of the base module are visible in the derived module. If the name does not collide with a name in the derived module, the name can be directly used. If it collides with a name in the derived module, rules in the derived module may access the variable using the standard `<base name>::<local variable>` notation. In the example, the rules `A::r` and `B::s` are functionally the same, since they both iterate over the container `B::P`. Note that `A::P1` is an alias for `B::P`.

```
module stipulate_relations(Relation r[])
:     enforce_symmetry(r),
      enforce_transitivity(r)
{
}
```

Figure 6. Inheritance Example

Venus also supports multiple inheritance. When multiply inheriting, the derived class can be viewed as consisting of its own rules plus the rules from each base module. Rules in the derived module can access local variables from the base modules directly by name if the name is unique, or by using the scope resolution operator (`::`). Figure 5 demonstrates use of the `::` notation.

To illustrate inheritance semantics, consider the module `stipulate_relations` defined in Figure 6 as well as a module similar to `enforce_symmetry` called `enforce_transitivity`. This example implements a symmetric and transitive relation `r`. As new elements of the relation are discovered, the `enforce` modules ensure that an explicit representation of the relationship between all pairs of objects is stored in the container `same`. Assume the system was initialized with the following pairs:

```
r(a,b), r(b,c)
```

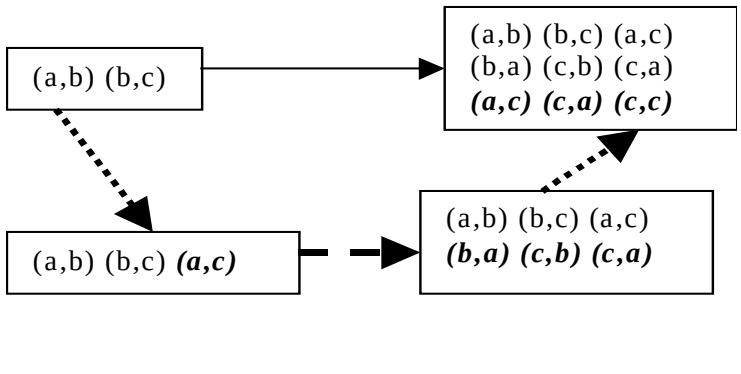
At fixed-point the following literals would be present in the same relation.

```
r(a,b), r(b,a), r(a,a),
r(b,c), r(c,b), r(b,b),
r(a,c), r(c,a), r(c,c)
```

Figure 7 presents a possible execution of this module through the state space. The solid line shows the view from the caller of `stipulate_relations`. From its view, the module call is an atomic transition from the inconsistent input state to the symmetric and transitive output state show the path through the inheritance hierarchy to the rules that assess the state changes. States in italicized type indicate states added. Individual rule firings are not shown. The transitions shown using a dotted line indicate firings of rules inherited from `enforce_symmetry`. The dashed transition shows firings of rules inherited from `enforce_transitivity`.

The most important aspect of the execution is that tangible rule execution moved directly from rules inherited from `enforce_transitivity` and to rules inherited from `enforce_symmetry` and *back to* `enforce_transitivity`.

Figure 7. Flow of Control



There were no procedural RHS actions in the inherited modules that explicitly activated the other. The inference engine determined the locus of control. If a module or rules inherited from a module changes the state such that another enabled module or rules inherited from another module is no longer at a fixed point, control passes *automatically* to the enabled module or rules.

3.2 Polymorphism

Venus is designed to inference over data. Venus supports polymorphism with respect to this data in two ways. First, Venus supports a formalized interface between data stores and the inference engine. This interface is called the Abstract Machine Interface (AMI) [MIR96]. The AMI is discussed in detail in section 4.3. It provides an abstraction allowing a data store to be of nearly any implementation, ranging from main memory linked lists to tables in relational databases such as Oracle.

The formal parameters to a Venus module simply specify the type of object that resides within a container. It does not specify the underlying container implementation. Thus, successive calls to a module may have as actual parameters different containers with different implementations. For example, the module defined in Figure 2 specifies as its formal parameter a container consisting of objects of type `Relation`. The first call to the module might be with a container implemented as a main memory doubly linked list. The second call to the container might be with a container implemented as an Oracle table. In the first case, a call to `r.insert(i)` results in inserting a node into the list, whereas in the second case, it results in adding a tuple into the Oracle database.

The second form of polymorphic behavior is with respect to inheritance within the data stored in containers. In the example rule in Figure 2, the container `r` consists of elements of type `Relation`. The accessor methods `domain()` and `range()` return the object ids of elements. If these were defined as C++ `virtual` methods, then a container of elements inheriting from type `Relation` could be passed to Venus as an actual parameter, and the method calls would be forwarded to the methods of the appropriate derived class.

A final area of polymorphic behavior concerns the behavior of Venus when containers have performance enhancing indexing structures. Rules have a tight correspondence with database queries. A single rule is generally equivalent to a relational query. Therefore, many of the same optimization techniques used in the database community are applicable to rule processing. In Venus, containers can be indexed. As previously stated, different containers can be passed to a Venus module at different times. It is entirely possible that these containers differ in the indices they support, if any. Therefore, Venus is designed to decide, at module execution time, what indices to use out of the available indices. Thus, depending upon the exact scenario at each module call, Venus will react in the appropriate way by choosing an optimized rule evaluation execution plan.

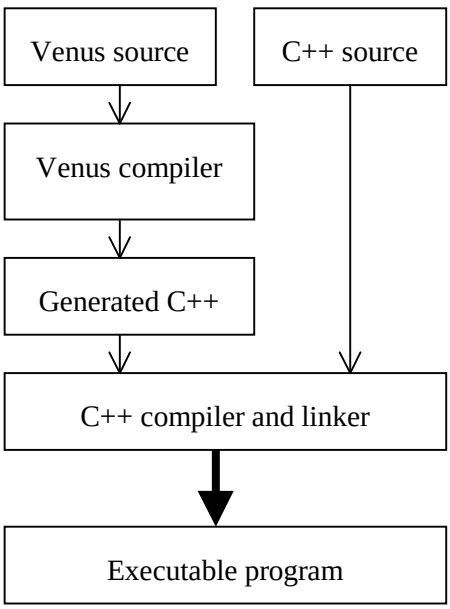
3.3 Encapsulation

Venus modules have no global data. As noted previously, Venus modules also implement usable forms of inheritance and polymorphism. Adoption of an appropriate set of coding conventions where a given module hierarchy is given complete responsibility for updates of the state of a specific set of Venus data classes leads to full encapsulation for these classes. Use of this mode of encapsulation will greatly enhance reusability of Venus modules. Future versions of Venus may formalize and enforce some form of these conventions

4 Implementation

The current Venus implementation is descended from a series of rule-based compilers, each employing more successful execution algorithms and indexing methods. The implementation does not yet directly implement all of the advanced features concerning inheritance and polymorphism. Similar to early C++ implementations, these features can be introduced through a preprocessor. Our discussions of the impact of these features on real programs stem from manual expansion of these conventions.

Figure 8. Venus Compilation



4.1 System Overview

Venus is a realized system, and includes a compiler, a debugger, and a rule profiler. The Venus compiler is the heart of the system. It takes as input rules in Venus format and generates C++. The compiler is organized conventionally, with phases for lexical and syntactic analysis, optimization, and code generation. The generated code is combined with any user written C++ code or third party C++ libraries (in source or archive format) and passed to the C++ compiler and linker. The output of that phase is an executable program. Figure 8 illustrates this process. The use of C++ as the target language enhances embeddability and portability of Venus.

The Venus system also includes a rule debugger, which allows a programmer to monitor an executing program for certain events and associate with those events break points and trace related outputs. At the break points developers may examine internal state of the execution and inference engine. The Venus profiler generates information about the behavior of programs most importantly the selectivity and usage of certain indices. This information may be exploited as input to the optimizer.[OBE95]

4.2 Performance

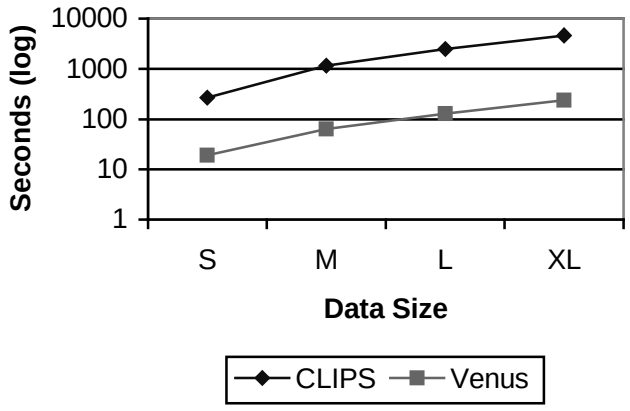
The Venus system is, to the best of the authors' knowledge, the highest performance rule system currently in existence. This is based on three primary features, the use of the LEAPS matching algorithm, support for data indexing, and an advanced technique for filtering updates.

Conceptually, rule environments like Venus work through an iterative cycle of examining the state space, choosing a satisfied rule to fire, and firing the rule. Since the rule firing presumably changes the state space, the cycle begins again. It was noticed that from cycle to cycle, the majority of the state space remains unchanged. This observation led to *incremental matching*, first embodied in the RETE match algorithm.In an incremental matcher rather rule predicates are recast from predicates whose arguments are the current state of the system, to predicates whose arguments are the results of the previous cycle and the changes to the current state. [FOR82]. As consequence of the resulting improvement in execution time, the RETE match promptly became the norm for the implementation of rule evaluation.

The Venus implementation is based on the LEAPS algorithm, not the RETE match. The RETE match embodies a time-space tradeoff and entails space complexity $O(n^c)$, where n is the size of the state space and c is the largest number of cursor variables in a single rule [MIR90]. The LEAPS (Lazy EvAluation for Production Systems) algorithm [BRA91, MIR90] starts with the incremental definition of rule evaluation but suspends evaluation as soon as a satisfied rule is identified. A backtracking stack of suspended evaluations is maintained and the internal sequence of execution carefully developed to maintain correctness The results is both prodigiously faster execution and a collapse of the memory requirements to linear in the size of the state space.

In addition to using the LEAPS matching algorithm, Venus is able to benefit from two other performance enhancing features. First, Venus can reduce rule evaluation time by utilizing indices over data containers, if available. As already mentioned, there is a correspondence between rule processing and database query evaluation. Previous work demonstrates that rule systems can take advantage of two types of indices, sorted attribute indices of the familiar database style and predicate indices (called *alpha memories* in the production system literature). A predicate index is a specialized index contains all of the elements of a container that satisfy a constant test predicate stated in a rule. Due to the fact that rules, like object definitions, express behavior rather than state, the set of rules for a program is relatively static. This allows the Venus compiler to create predicate indices to speed up individual rules, much as an optimizing compiler for a traditional language

Figure 9. Performance Result



The combination of the above optimization techniques provides Venus with excellent levels of performance, both at a single instance and with respect to data set scaling. As evidence, we present previously published benchmarking results [OBE94]. Figure 9 shows the performance of Venus compared to CLIPS version 6.0, a product of NASA. CLIPS is a popular expert system shell that includes COOL, the Clips Object Oriented Library. CLIPS uses the RETE match algorithm. The graphs in the figure compares Venus and CLIPS on the Waltz benchmark program, a typical program from the Texas Benchmark Suite [BRA91]. While this program does not use any of Venus's object oriented features, it fully exercises the rule execution component. As the graph shows, Venus performs in excess of one order of magnitude faster than CLIPS at every data size.

4.3 Data Access

Venus communicates data stores via a well-defined interface, called the Abstract Machine Interface (AMI) [MIR96] . The AMI provides an encapsulation over data stores. The AMI consists of two C++ abstract classes, a container object and a cursor object. A container object is a wrapper around a data store, and consists of methods like `insert(object)`. As such, it provides standard iterator primitives such as `reset()`, `next()`, and `get()`, though with some Venus specific behaviors.

The AMI is implemented as two C++ abstract classes, `AMI_Container` and `AMI_Cursor`. A programmer wishing to implement an interface to Venus for his/her data store need only create two C++

classes, one that inherits from `AMI_Container` and one that inherits from `AMI_Cursor`. These classes must provide implementations for the primitives defined in the AMI base classes.

By virtue of the AMI, a Venus module can inference over data stored in different containers. AMI implementations have been written for reference counted main memory linked lists, doubly linked lists from the Rogue Wave Tools.h++ class library, Sybase relational database tables, and ObjectStore OODB containers. For an example of the utility of the AMI, see section 5.2.4.

5 Application Experience

The following section describes two applications that we developed using the current Venus implementation. In each case, modularity proved beneficial in the design of the programs. We can pinpoint many places in these programs whereupon the implementation and exploitation of inheritance and polymorphism would improve the code of these applications by another factor.

5.1 Extensible Database Query Optimizers

A major focus of database research, circa 1990, was the development of database query optimizers in rule-based form [GRD87, PHH92, GRM93, DAS95, DB95]. A database query optimizer takes as input the definition of a query and returns as output a plan for the execution of the query. For each logical operator expressed by the query, an optimizer must choose an implementing algorithm, termed a *physical operator*, and an ordering of the operators. Operators are selected from a library of operators supported by the particular database, such as the well known join algorithms like nested loops, hash-join, etc. In general, query optimization is NP-Complete. Thus, query optimizers are the subject of extensive investigation with respect to approximate and heuristic methods.

Conceptually, query optimizers operate by applying algebraic identities and rewrite rules derived from relational algebra. This is in essence declarative, and a rule-based representation is attractive since it closes the semantic gap between the specification of an optimizer's capabilities and its implementation. The database community has shown that by virtue of an executable specification, rule-based optimizers are easier to extend than their predecessors. However, each of the demonstration systems mentioned above considered general-purpose rule-languages as the basis for their optimizers and deemed them infeasible. LISP-based syntax, unembeddable data definition languages, slow performance, and rigid search strategies were often cited as the major shortcomings of general-purpose rule languages. As a result, each of these query-optimizer groups developed their own specialized rule environments. These environments contain features that go beyond the convenient formulation of optimizer rules and include ad-hoc elements for controlling search and encapsulating rules. This is in an effort to better manage the search spaces and otherwise improve execution speed. Thus, the development of query optimizers in these systems usually involves elaborate knowledge, at the source level, of both the rule language **and** its underlying execution engine. Development of new query optimizers in these systems often necessitates engineering at both levels.

We believe that the Venus language and its implementation attended to most of the concerns of the database researchers. To date we have implemented four different query optimizers in an effort to prove the hypothesis that Venus meets the necessary requirements for the development of rule-based query optimizers and provides execution speed comparable to these specialized systems. Further, the resulting query-optimizers are much smaller and more extensible than those optimizers developed using special purpose rule environments.

5.1.1 Organization of the Venus Database Query Optimizers

```
rule apply_associative_operator;
from distinct
       // for binding a pair of operators
       OpenList[?] opp_node1, opp_node2;
       // bind to the data catalog
       MetaData[?] rel;
if (   opp_node1.QueryId() == opp_node2.QueryId() &&
       opp_node1.NodeOp() == JOIN &&
       opp_node2.NodeOp() == JOIN &&
       // the operators are adjacent in the tree
       isAssociative(opp_node1.plan, opp_node2.plan, rel)    &&
       // the two joins have relation in common
       // prevents the introduction of cartesian products
       Union (opp_node1.plan, opp_node2.plan, rel))
{
       // Then apply transform
       QUERY_PLAN optimized_node(Associate(opp_node1, opp_node2, rel));
       // Call Venus module which tests post-condition statements
       postConditions(optimized_node, OpenList, ClosedList);
}
```

Figure 10. Query Optimization Rule

The three basic parts of a rule-based query optimizer consists of the *cost model*, the *search space*, and the *search strategy* [DB95]. The Venus rule-based query optimizers are being built to uncover an organization that is extensible in all three dimensions. The first element of the approach was to define a generalized operator tree clearly capable of representing SQL expressions and otherwise heavily influenced by similar definitions used in other extensible query optimizers [PHH92, OZM95]. Since the operator tree is an argument to rule-based transforms, this part of the system is defined starting with a C++ base class. By virtue of a single base class there is a standard method for expressing pattern matching against the tree. Transformation rules themselves are stylized in the form of a triple; a precondition, a pattern predicate on the operator tree, and the side effect of the transformation. A design pattern appeared for the expression of transformation rules. Inheritance is exploited to define a class specialized for each logical and physical operator supported by the database system. Thus, in regard to adding new operators, Venus does no more than exploit the power provided by specifying C++ as the data definition language.

The important object-oriented result is that independent of the search method, the conceptual structure remains the same. For any individual database system, that is for any one library of physical operators, search strategies may be changed without changing the modules that express the transformations. There are groups of transformations specific to the expressive power of the query language and the scope of the search space to be considered. We anticipate reusable modules organized around these groups and for users of the query-optimizer to be able to "plug-and-play" from a library of transformations. Details are beyond the scope of this paper. However, a simple example is the set of logical identities concerning associativity in the relational algebra. Disallowing associativity can disallow so-called bushy query plans. Thus, it becomes a simple matter to implement system requirements concerning the legality of bush query plans. If only by analogy, a documented library of such rule modules for query optimization would have advantages similar to those created by abstract-data type libraries.

5.1.2 Implementations

Our first significant optimizer, a simple relational optimizer, spans a single implementation of the select, project, join, sort and scan operators. The optimizer exploits a simple bounded, best-first search strategy. The entire optimizer was expressed in just 7 rules. Counting rules can be misleading. There were 2,690 total lines of code, of which 325 lines of those were used to express the rule system and 1,067 lines, were used to define the C++ objects needed to represent the operator tree. Line counts include comments and white space.

The second optimizer, a re-implementation of the Open OODB optimizer [DB95], required 4,481 lines of code. Most of the code from the first optimizer was reused. The 1,067 lines used to define the operator tree remained unchanged. The search engine remained unchanged, the transformation aspects of the first optimizer were reused. Quantitatively, of 1,692 lines of code comprising the non-data part of the program, 1,193 lines remained unchanged.

We rebuilt the System R optimizer [SEL79]. System R employs a *bottom-up* strategy, where the previous two optimizers are considered top-down. This was done by first expanding the operators in the first optimizer to include all the operators in System R. An envelope encoding System R's bottom-up search technique was written. Per above, precisely the same modules expressing the transformations work unaltered for both search techniques.

5.2 The Mortgage Pool Allocation Problem.

The mortgage pool allocation problem involves matching buy orders of mortgage-backed securities with sell orders. If this were stocks or bonds this would be a simple bin-packing problem. In mortgage-backed securities, though, buy orders need only be filled within 2.5% of the requested amount and there are government regulations that dictate how sell orders may be cut into smaller pieces and how those pieces may be allocated to the buy orders.

We have implemented a solution to the mortgage pool allocation problem in Venus, coined REALESYS. We then developed quantitative software metrics and compared their values against the same statistics developed for the original implementation in the OPS5 rule language, coined ALEXSYS [STO90, WAR96]. The OPS5 version has also been recoded in C++ and widely licensed by financial institutions. It would clearly be valuable to measure the C++ version as well, but that version is proprietary.

5.2.1 REALESYS

REALESYS was designed top-down, using stepwise refinement. The REALESYS solution used the same basic collection of greedy heuristic methods exploited ALEXSYS. Figure 11 illustrates the program structure of REALESYS. Each vertex in the graph, both circles and squares, represent a module. Vertices with multiple arcs entering them represent parameterized modules[1]. Control flow in the program is clear.

[1]Note that the ALEXSYS graph does have multiple arrows entering single modules. However, these modules are used for only a single purpose since parameters are not available, thus, they are not an instance of code re-use.

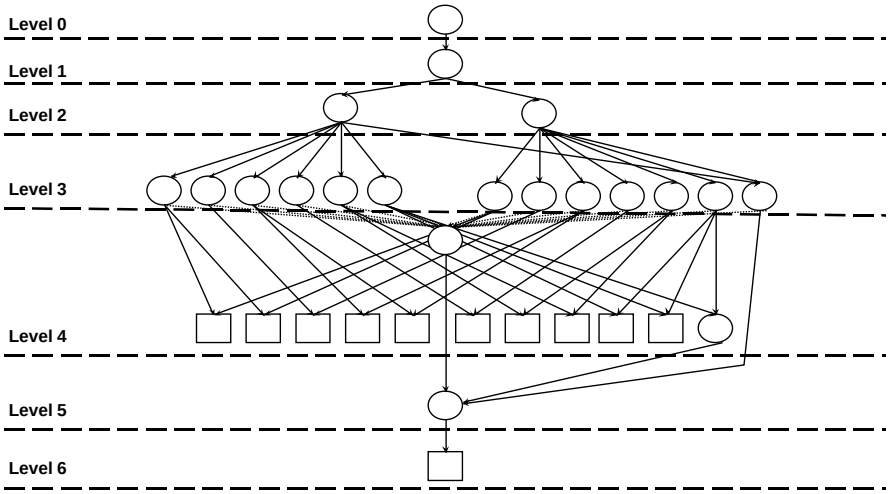


Figure 11. REALESYS Call Graph

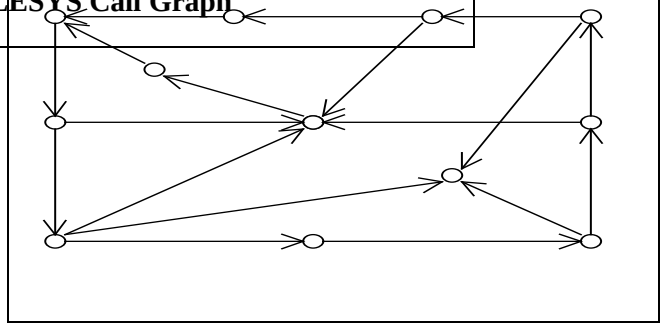


Figure 12. ALEXSYS Call Graph

The length of the longest path in the structure of REALESYS is seven long and corresponds directly to seven levels of stepwise refinement expressed in the implementation. Each level considers one element in the decision processing. The rule predicates are correspondingly simple. Similar to the query optimizers, the organization provided for easy but substantial flexibility in experimenting with different search heuristics as well as encapsulating program requirements within clear syntactic structures. For example, the circles in level three reflect the preconditions for filling an order in a certain way. The boxes in level four perform the corresponding allocation. Thus, should a change in government regulations require program maintenance there is clear place in the code where the changes will have to be made.

5.2.2 ALEXSYS

OPS5 contains no explicit method for structuring a rule set. To create structure, programmers commonly add special data elements into the state of the program and add to each rule a predicate testing those data elements for particular values. Control may then be passed from one subset of rules to another by changing the value of one of those elements. Since this is not distinguished syntactically and programs are rarely well documented, this method of control flow is sometimes called *secret-messages* [MCD93].

After deciphering the secret messages in ALEXSYS, it is possible to expose a modular structure developed by the original authors. Figure 12 illustrates the control flow of ALEXSYS's 44 rules and 12 modules**.** It is this intrinsic modular structure that is reported on in this study.

The longest acyclic path through the call graph is of length nine. Yet, the graph is made up of many cycles, and in fact, the starting node of the graph is ambiguous as drawn and only slightly clearer when reading the OPS5 source code. Each node in the graph can only be used for a specific purpose providing for no code-reuse in this implementation. Thus, the difficulty in determining how control reaches any point in the program provided for confusion in analyzing the effects of a change to the rule base.

5.2.3 Quantitative Results

The ideal test for design and long-term maintenance of the two implementations would be to deploy both, independently, and to carefully account for manpower costs, number of bugs, average time to repair a bug etc. Such large scale testing is impractical and has rarely been performed. However, a number of efforts have resulted in quantitative software metrics that have, in smaller scale studies, been correlated to the life-cycle costs of computer software.

Any one software metric will not give a full picture of a program's quality. There are three measurement domains for software metrics: volume, control flow, and information flow [GRO91]. We compared ALEXSYS vs. REALESYS in each domain and with respect to four different software metrics. These are the volume metrics of conditions per rule and lines-of-code, the control flow metric McCabe's cyclomatic complexity, and an information metric of "fan-out" [MCC76, HEN81]. Figure 13 presents the results.

5.2.3.1 Conditions Per Rule

The volume metric of conditions per rule gives evidence of a rule-based program's guard complexity. This metric is often considered by many rule-based programmers as the most telling metric of the complexity of a rule-based program [GUP87]. Many conditions per rule imply complex guards with a

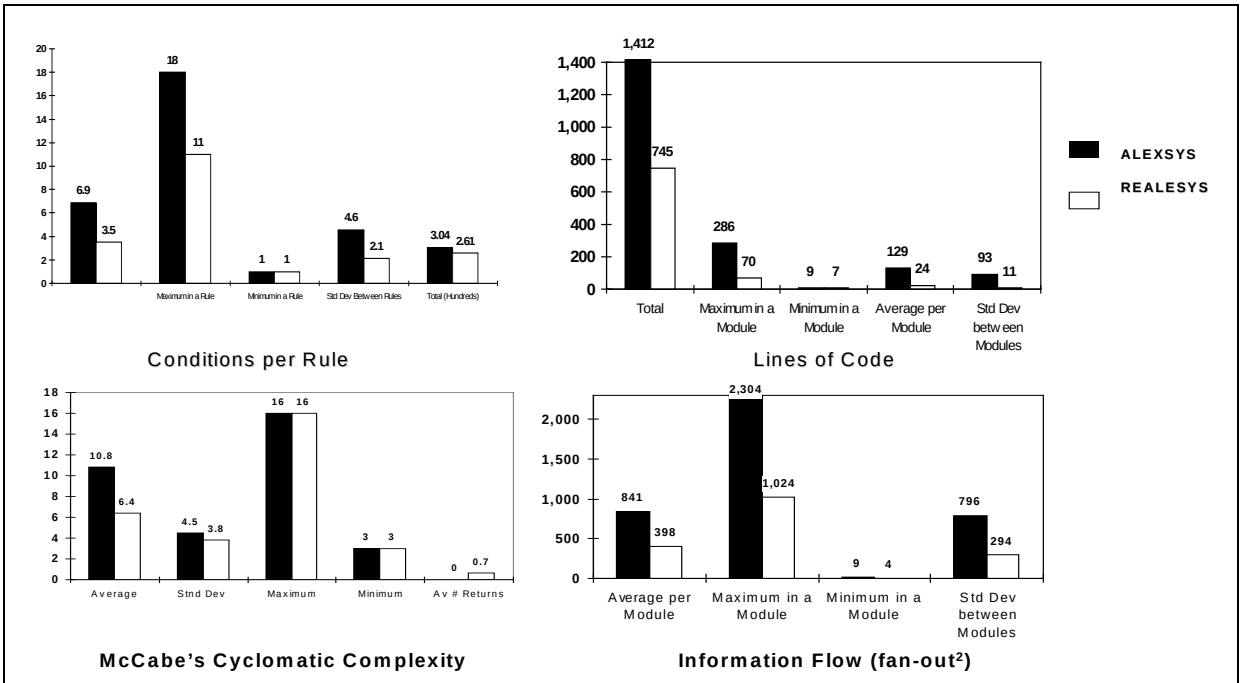


Figure 13. Software Metrics

large potential for error. Few conditions per rule in contrast represent less complex guards with a small potential for error.

5.2.3.2 Lines of Code

Lines of code measure the volume of the total program. This metric is frequently given for iterative programs as a measure of the overall complexity of a program. The more lines of code any program may have, the more apt to typographical errors, code repetition, and complex statements that the program may contain

5.2.3.3 McCabe's Cyclomatic Complexity

Cyclomatic complexity measures the complexity of control flow by extracting the number of possible paths through a program. Cyclomatic complexity, v(G), of a program graph of n vertices, e edges, and p connected components is calculated by:

1) $v(G) = e - n + 2 * p$

2) = number of decision statements + 1

In formula 2, a decision statement, called a predicate, is any conditional branch within a module. For iterative programs, a cyclomatic complexity of less than or equal to 11 for a given module is considered acceptable [MCC76]. Since the control of a rule-based program differs from an iterative program, cyclomatic complexity was slightly adapted to rule-based programming [WAR96].

5.2.3.4 Information Flow (fan-out^2)

Fan-out^2 measures the complexity of information flow between modules. This metric provides evidence of modules with a likelihood of semantic errors [HEN81]. Fan-Out^2 is divided into two parts, fan-in and fan-out. Fan-in is the number of objects passed into a module (read set), while conversely, fan-out is the number of objects that are changed that are visible outside of the module (write set). Thus, fan-out is calculated by

$$\text{Fan-Out}^2 = (\text{fan-in} * \text{fan-out})^2$$

5.2.3.5 Results of quantitative comparison

The metrics consistently yielded around a 2-fold improvement of complexity in favor of REALESYS. This provides strong evidence of the power and usefulness of Venus's parameterized modules. In fact, only one area seemed to swell as a result of implementing the mortgage pool allocation problem in Venus as opposed to OPS5, the number of rules. As is typical in any modular and functional environment, Venus broke up many large complex problems into many smaller and easy to digest rules. The other numbers support this claim. Further, all the measurements are with respect to the current implementation. We can pinpoint a number of places where repeated definitions could be eliminated with an implementation of the inheritance mechanism. The net result would be 11 fewer rules and 84 fewer lines of code.

5.2.4 REALESYS-Active

REALESYS has been re-implemented from a main memory batch processing system into an active database form appropriately named REALESYS-Active. This re-implementation was straight forward because of the object oriented and modular constructs exploited in the Venus implementation of REALESYS. The level 3 - 6 modules presented in Figure 11, which contain the application logic and form the majority of the rule base, was left completely unchanged in this version. The top level modules were modified to move from a batch orientation to an active orientation.

Apart from the batch to active change, the data store changed as well. As an active database application, REALESYS-Active retrieves all its data from containers stored in ObjectStore databases. This is by virtue of the AMI, which abstracts the physical implementation of the containers. As noted in the previous paragraph, the level 3 - 6 modules were unchanged, even though the data they were inferencing over was now coming from a commercial OODB rather than in memory lists.

6 Related Work

The examination of ALEXSYS alone should make it clear that the early, flat-monolithic rule-based programming languages are simply not viable for large-scale software engineering. Indeed commercial offering of proprietary rule-based languages promptly addressed this by introducing syntax for grouping rules and regulating control flow by treating those rule groups as subroutines and/or program blocks reachable with goto statements [ART87]. The rule groups were not parameterized. From a practitioner's perspective this was seen as an improvement. Never the less, the semantics of the resulting languages becomes difficult, at best. Rules could no longer be counted on to fire unless procedural control flowed through the correct rule group. Without parameters and a rich set of iterative constructs the languages were deficient as procedural languages. Thus, these procedural extensions are simply insufficient to repair the difficulties associated using rule systems when programming in the large.

A much more common solution to the lack of modularity in rule-based languages has been to start with an object-oriented language and to augment it with a rule capability. This approach has been called *Embedded Object-Oriented Production Systems*[2] *(EOOPS)* [PAC94] and it dates to the earliest efforts in the development of expert systems shells. The developers of the LOOPS expert-system shell [Bobrow&stefik83] were explicit in their goal to integrate diverse programming paradigms in a single framework. In LOOPS four programming paradigms were distinguished, procedural, object, data and rule oriented. Their definition of data-oriented programming derived from frame systems where a slot, though represented by a label indistinguishable from a variable name, would in fact execute a function and return the function value. The functions could have procedural side effects. In a more contemporary context, this is the desired behavior of expanded database trigger abilities in the form of active and deductive databases [STO2]. The procedural aspects of LOOPS were folded into the other three other paradigms in the obvious ways; methods for objects, the action part of a rule for rules, triggered procedural attachment on data access.

In LOOPS the primary framework for the integration was objects. Everything in LOOPS forms an object. In the special case of rules, a collection of rules could be used to define a method. Of course if the an object had only a single method defined using rules, then that was a rule object. The scope of the rule's predicates was one in the same as the scope of procedural methods within the object. LOOPS advanced the notion of flow-control through message passing. Consequently objects and their behavior could be recursively defined with respect to other objects in the system the paradigms could be intermixed

[2] Production System is a synonym for rule-based program.

arbitrarily. For example, the value of an active data object could be determined by a set of rules. In the course of evaluating a rule, a message could be sent to another object, moving control to that object which might itself be another active data object. Thus, objects in LOOPS formed a basis for syntactic encapsulation, but there was nothing in the system to structure and encapsulate related functional elements.

By virtue of its generality we can review the functionality of other EOOPS systems by citing LOOPS. An element allEOOPS systems have in common with LOOPS is the orthogonal use of object definitions for representing state. Starting with CADIE [FRA90], the vast majority of contemporary EOOPS systems are embedded in C++[PAC94]. Thus, like Venus, C++ class definitions form a data definition language. However, with the exception of Venus, the C++ embedded rule systems require substantial hooks into the class definitions. This usually take the form of requiring objects "seen" by the rule system to inherit methods from a special base class. In Venus scalar values can be manipulated directly. Collections are supported two ways. Either a pointer to each object may be inserted into a special container or a user must declare to the Venus compiler the names of basic cursor functions able to iterate over an existing container. The goal behind the organization of Venus was to enable the system to inference over embedded data without having to alter the structure of that data. In particular to be able to infer over legacy data, especially that already stored in relational database systems.

CADIE, ILOG rules and RAL all embrace the notion of an inference-engine object, such that each instance of that object is parameterized by a particular rule set. In other words, like a LOOPS rule-object, rules could be grouped to form an object instance. Since there is a single entry point to the rule-object, syntactically it would be feasible, similar to Venus, to define rule groups using inheritance mechanisms. This opportunity is not discussed in any of these systems. The CERS system also embraced the notion of an inference-engine object. But rather than casting rule grouping at the level of objects, rules in CERS are used to define methods within an object. Thus, a method could be defined either procedurally or declaritively with rules. Both forms of definition could exist within a class definition. It follows that inheritance could be supported, but at the granularity of a method. That is behaviors could be specialized, but only by replacing entire sets of rules. CERS provides a larger encapsulated scope within which the rules could operate. It enabled a limited form of parameterization by allowing entire containers of objects to be swapped at during runtime by calling inference-engine methods that would exchange pointers. In all these systems, control flow between rule-groups remains procedural. The concept of message-passing as a control mechanism can be spoken to. Nevertheless, the resulting semantics is that the rules only evaluate a program's state after some explicit call.

R++ is the only system, other than Venus, to completely, syntactically, divorce itself from the ancestry of rule-based systems in expert-systems. In R++ rules may be interspersed inside a class definition with ordinary C++ methods. Since rules are named, class definitions may be inherited and individual rules added or refined. The scope of the rule predicates is limited to the scope of data visible within the object instance or by dereferencing a path expression that starts with data within the instance; a safety property known as *access-limited*. The execution model is that of the active-data paradigm in LOOPS, except that access to any value within a class instance results in all R++ rules associated with that instance to become sensitive to the program state and able to fire. The execution model is implemented by a preprocessor that, as identified by special declarations, wraps accesses to active data with calls to an inference engine. Access-limitation has been shown to provide for tighter complexity bounds on the evaluation of the rule predicates. Access-limitation, coupled with the object/rule structuring in R++ is tantamount to enforcing additional encapsulation and further provides a basis for assuring the correctness of the implementation with respect to the insertion of the wrappers which serve to integrate the inference engine with execution.

Given the tight integration of R++ with underlying C++ class definitions, existing legacy systems can not add R++ code without recompilation. Since any method call to an object allows the rule component to evaluate, the paradigm of message-passing as control flow clearly applies and would appear to ameliorate

the deficiencies associated with procedural control of rule groups. The structure of state in an R++ is not as well structured as the nested-state space model of Venus, but it is precisely the same structure of ordinary C++ programs and so, obviously, sufficiently well structured for large-scale programming. The R++ is the system most similar to Venus, syntactically and functionally. Given that it is a rule-based extension of an object-oriented language, rather than vica-verca, a detailed comparative study would appear most worthwhile.

7 Conclusion

This paper presented an overview of the object-oriented features of Venus, a declarative language for rule based programming. Venus adds object oriented features to a rule language, rather than adding rules to an object language, which is the course followed by most prior efforts at integrating the rule and object paradigms. Venus uses C++ for both its basic syntax and its data definition language, which allows it to inherit many object oriented features in a familiar way. Venus builds upon features inherited from C++ by adding inheritance, polymorphism, and encapsulation to the rules themselves.

To demonstrate the utility of the Venus object/rule formulation, we presented our experience with two applications, a set of extensible database query optimizers and a large expert system from the financial industry. The first described application consists of a family of database query optimizers. The object oriented features of Venus allow one optimizer to reuse rules created for other optimizers, resulting in an extensible system. The second application is a Venus rewrite of an existing expert system originally written in OPS5, and the paper presents a comparison of the two programs with respect to several software engineering metrics. While the Venus formulation has a greater number of rules than the OPS5 formulation, Venus provides significant improvements in all other metrics. The Venus formulation displays roughly a two fold complexity simplification in each of the four metrics spanning the three basic measurement domains of software metrics, volume, control-flow and information-flow. Thus, we conclude that the method of encapsulation developed for Venus represent an important step toward reducing the engineering costs of expert-system programs.

8 References

[ALB94] Albert, P. "Ilog Rules, Embedding Rules in C++: results and limits." In *Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems (EOOPS)*, Technical Report LAFORIA 94/24. Laboratoire Formes et Inelligence Artifcielle, Institut Blaise Pascal. Dec. 1994.

[ART87] Inference Corp. *Art Reference Manual*, 1987.

[BRA91] Brant, D., T. Grose, B. Lofaso, and D. P. Miranker. "Effects of Database Size on Rule System Performance: Five Case Studies." In *Proceedings of the 17th International Conference on Very Large Databases*, 1991.

[BRO95] Browne, J. C., and et.al. "Modularity in Rule-Based Programming." *International Journal on Artificial Intelligence Tools*, 4(1), 1995.

[CRA96] Crawford, J, et. al. "*Path based rules in object oriented programming*." In Proceedings of the 13th National Conference on Artificial Intelligence (AAAI), 1996.

[DAS95] Das, D. "*Making Database Optimizers More Extensible*" Ph.D. dissertation, The University of Texas at Austin, May 1995.

[DB95] Das, D. and D. Batory. "Prairie: A Rule Specification Framework for Query Optimizers." In *Proceedings of the 11th International Conference on Data Engineering*, March 1995.

[FOR81] Forgy, C. "OPS5 User's Manual." Technical Report CMU-CS-81-135. Carnegie-Mellon University, 1981.

[FOR82] Forgy, C., "RETE: A Fast Match Algorithm for the Many Pattern/Many Object Pattern Match Problem." Artificial Intelligence, 19, 1982.

[FOR94] Forgy, C. "Embedded Object-Oriented Production Systems." in Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems (EOOPS), Technical Report LAFORIA 94/24. Laboratoire Formes et Inelligence Artifcielle, Institut Blaise Pascal. Dec. 1994.

[FRA90] D.W. Franke. "Imbedding Rule Inferencing in Applications.", *IEEE Expert*, pages 8--14, Dec. 1990.

[GIA88] Giarratano, J. C. CLIPS User's Guide, Version 4.2. Artificial Intelligence Section, Lyndon B. Johnson Space Center, 1988.

[GRD87] G.Graefe, and D. Dewit. "The EXODUS Optimizer Generator." In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, May 1987.

[GRM93] Graefe, G. and W. McKenna. "The volcano Optimizer Generator: Extensiility and Efficient Search." In *Proceedings of the 12th International Conference on Data Engineering*, March 1996.

[GRO91] Grose, Timothy. *The Programming and Functionality of OPS5 Compared to LISP and FORTRAN in an Aeronautical Route Planning System*, Masters of Arts Thesis, The University of Texas at Austin, May 1991.

[GUP87] Gupta, A. *Parallelism in Production Systems*, Pitman/Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[HAL96] The Haley Enterprise. *RETE++ Manual*. 1996.

[HEN81] Henry, S., and D. Kafura. "Software Structure Metrics based on Information Flow." *IEEE Transactions on Software Engineering*, SE 7(5), September 1981.

[LIT97] Litman, D., P.F. Patel-Scheider, A. Mishra. "Modeling Dynamic Collections of Interdepedent Objects Using Path-Based Rules" In the Proc. *of the OOPSLA Conference.* 1997 (to appear.)

[MCC76] McCabe, T. "A Complexity Measure." *IEEE Transactions on Software Engineering*, December 1976, 308-320.

[MCD93] McDermott, John. "R1 ("XCON") at Age 12: Lessons from an Elementary School Achiever." *Artificial Intelligence*, (59): 1993, 241-247.

[MIR90] Miranker, D. P., D. Brant, B. J. Lofaso, and D. Gadbois. "On the Performance of Lazy Matching in Production Systems." In *Proceedings of the 1990 National Conference on Artificial Intelligence*, 1990.

[MIR91] Miranker, D. P., and B. Lofaso. "The Organization and Performance of a TREAT-Based Production System Compiler." *IEEE Transactions on Knowledge and Data Engineering*, 3(1), 1991.

[MIR93] Miranker, D. P., F.H. Burke, J.~J. Steele, J.~Kolts, and D.~R. Haug. "The C++ Embeddable Rule System." *Int. Journal on Artificial Intelligence Tools*, 2(1):33--46, 1993. Also in the *Proc. of the 1991 Int. Conf. on Tools for Artificial Intelligence*.

[OBD94] Obermeyer, L, and D. P. Miranker. "Clips++: Embedding CLIPS into C++." In *Proceedings of the Third CLIPS Conference*, 1994.

[OBE95] Obermeyer, L., D. P. Miranker, and D. Brant. "Selective Indexing Speeds Production Systems." In *Proceedings of the 7th International Conference on Tools with Artificial Intelligence*, 1995.

[OBE96] Obermeyer, L., L. Warshaw, and D. P. Miranker. "Porting an Expert Database Application to an Active Database: An Experience Report." *In Proceedings of the Workshop on Databases: Active and Real-Time*, 1996.

[OZM95] Ozsu, M. T., A. Munoz and D. Szafron. "An Extensible Query Optimizer for an Objectbase Management System." In *Proceedings of the 4th International Conference on Information and Knowledge Management*, November 1995.

[PAC94] Pachet, F. ed. *Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems (EOOPS)*, Technical Report LAFORIA 94/24. Laboratoire Formes et Intelligence Artifcielle, Institut Blaise Pascal. Dec. 1994.

[PHH92] Pirahesh, H., J. Hellerstein and W. Hasan.. "Extensible/Rule Based Query Rewrite Optimization in Starburst tor." In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, May 1992.

[RUM91] Rumbaugh, J., et. al. "Object-Oriented Modeling and Design" (Prentice-Hall, Englewood, Cliffs, NJ, 1991

[SHL92] Shlaer, S. and S. Mellor " Object Lifecycles: Modeling the World in States" (Yourdon Press, Englewood
[STO90] Stolfo, S. et.al. "The ALEXSYS Mortgage Pool Allocation Expert System: A Case Study of Speeding Up Rule-based Programs." Columbia University Department of Computer Sciences and Center for Advanced Technology, 1990.

[SEL79] Selinger et al. Access Path Selection in a Relational Database System." In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1979.

[STO2] Stonebraker; M. "The Integration of Rule Systems and Database Systems.*" IEEE Transactions on Knowledge and Data Engineering*, 4(5), 1992.

[WAR93] Warren, D. H. D. "An Abstract Prolog Instruction Set." Technical Note 309. Artificial Intelligence Center, SRI International, 1993.

[WAR96] Warshaw, L. and D. P. Miranker. "A Case Study of Venus and a Declarative Bases for Rule Modules." In *Proceedings of the 5th Conference on Information and Knowledge Management*, 1996.

[XER82] Xerox Corporation. *LOOPS Reference Manual*. 1992.