CLIPS++: Embedding CLIPS into C++ *

Lance Obermeyer and Daniel P. Miranker

Tactical Simulation Division
Applied Research Laboratories
The University of Texas at Austin
Austin, TX 78713
lanceo@arlut.utexas.edu

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
miranker@cs.utexas.edu

Abstract

This paper describes a set of C++ extensions to the CLIPS language and their embodyment in CLIPS++. These extensions and the implementation approach of CLIPS++ provide a new level of embeddability with C and C++. These extensions are a C++ include statement and a defcontainer construct; `(include <c++-header-file.h>)` and `(defcontainer <c++-type>)`.

The include construct allows C++ functions to be embedded in both the LHS and RHS of CLIPS rules. The header file in an include construct is the same header file the programmer uses for his/her own C++ code, independent of CLIPS. The defcontainer contruct allows the inference engine to treat C++ class instances as CLIPS deftemplate facts. Consequently, existing C++ class libraries may be transparently imported into CLIPS. These C++ types may use advanced features like inheritance, virtual functions, and templates.

The implementation has been tested with several class libraries, including Rogue Wave Software's Tools.h++, GNU's libg++, and USL's C++ Standard Components. The execution speed of CLIPS++ has been determined to be 5 to 700 times the execution speed of CLIPS 6.0 (10 to 20 x typical).

1 Introduction

CLIPS++ is a reimplementation of NASA's CLIPS 6.0 [3] that has been tailored to support applications with large data and performance requirements or applications that must coexist with C++. This reimplementation has the following features

- Rules may directly access C++ objects. No need to reformat C++ objects to CLIPS representations or vica versa[1].
- Simple integration with existing C++ code.
- Compatible with C++ development tools.
- Execution time is reduced from 5 to 700 times (10 to 20x typical).
- Scalabile with respect to data and throughput requirements. See Section 3.2.
- Matching technology that eliminates the problems of volatile match time; resolving a critical problem for real-time applications.

CLIPS++ is compatible with NASA CLIPS 6.0 except that the COOL object system has been replaced with C++ objects, and only a single LEX-like conflict resolution strategy is supported. Nearly all publically available CLIPS programs available on the Internet have been compiled and correctly executed by CLIPS++.

2 Language

The CLIPS++ system includes minor language extensions that allow CLIPS rules to operate transparently on C++ object instances. These extensions comprise just two new constructs and a semantic extension to the use of deftemplate.

*This effort was funded in part by the State of Texas Advanced Technology Program, the Applied Research Laboratories Internal Research and Development Program DABT63-92-0042.

[1] CLIPS++ is a true integration with C++, not a simple wrapper like RETE++

Data Size	CLIPS 6.0	CLIPS++	Speed-Up
16	3	<1	>3.00
32	64	<1	>64.00
64	271	9 3	906.33
128	n.a	12	
256	n.a	66	

Table 1: Execution Time of Manners, (seconds), CLIPS 6.0 vs. CLIPS++

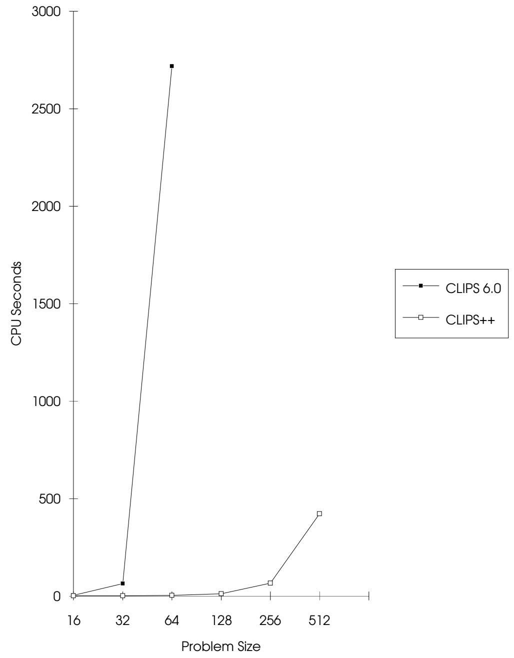

Figure 1: Relative Performance of Manners, CLIPS 6.0 vs. CLIPS++

2.1 Declarations

2.1.1 include

```
; salary.clp
(include "decls.h")
```

The include construct is equivalent to a C/C++ #include. It makes all declarations in a legal C/C++ header file visible to CLIPS++. For example, assume the C++ header file decls.h declares a type `employee_type`. That file may in turn include definitions from third party class libraries. The Rogue Wave string and date classes are used in the running example in this paper.

Consequently, listing the include statement (above) at the beginning of a CLIPS++ source file makes declarations in the file decls.h, including `employee_type`, visible to the CLIPS++ program.

```
#include<rw/cstring.h>
#include<rw/rwdate.h>


class   last_raise_type {
public:
        RWDate          date;
};

class   employee_type   {
public:
        RWCString       name;
        RWCString&      get_department(void)
                                {return dept;}
        last_raise_type last_raise;
private:
        RWCString       dept;
};
```

2.1.2 defcontainer

```
; salary.clp
(defcontainer employee (type employee_type))
```

The defcontainer construct is the primary language addition. Defcontainer is equivalent to the CLIPS deftemplate, except that it is used to declare that a C++ type can be referenced in a rule's LHS. Note that the slots of a defcontainer are precisely the slots of the C++ object defined in the include file. Thus, the arguments to defcontainer are limited to the containers name and the C++ type that is stored in that container.

Any C++ type can be used provided that the type overloads the operators `==` and `>` in the obvious way. Advanced C++ features like inheritance, multiple inheritance, virtual functions, and templates may be used. Most notably the C++ data type need not inherit from a CLIPS++ provided base class. This allows application developers great flexibility in designing class hierarchies, and reusing existing code.

2.2 Rule Syntax

There is no real change between CLIPS++ rule syntax and CLIPS rule syntax. There is a semantic extension to the meaning of the slot names for unordered facts. When the compiler recognizes that a template name is C++ container name, declared by defcontainer, rather than a template name declared by deftemplate, then the slot identifiers are allowed to be any legal C++ expression that returns a value from an object. The expressions may include the C++ dot (.) and dereference (-¿) operators. As a result LHS's can be formulated to traverse complex object structures. For example, the following is a legal CLIPS++ rule that operates on the class defined above. The compiler recognizes `employee` as the name of a container of instances of the C++ class `employee_type`. The slot defined by the accessor function `(get_department())` returns values from the instances. The slot defined by the expression `(last_raise.date)` returns a value from a nested object instance.

```
; give a raise to everyone in r&d who has not
; had a raise since the beginning of 1994
(defrule give-a-raise-to-r&d
  ?e <- (employee
            (get_department() "r&d")
            (name ?name))
            (last_raise.date ?d
                 &:(< ?d (RWDate 1 1 1994)))
  =>
...
```

In the RHS, member functions may be called for objects that are matched in the LHS. In both the RHS and LHS, arbitrary C/C++ objects may be accessed or called.

The tight integration with C++ has performance benefits as well. C/C++ functions are directly called. CLIPS introduces a level of indirection by activating C functions through a name to address mapping scheme.

3 Architecture and Environment

3.1 Development Environment

The CLIPS++ system is based on an optimizing compiler that accepts CLIPS or CLIPS++ programs as input and outputs C++. Output code is compiled by the host system's C++ compiler, and linked with a runtime library of CLIPS++ support routines.

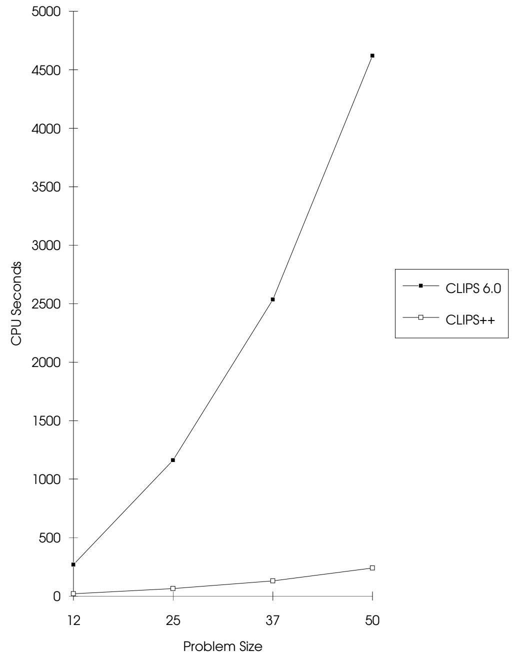

Figure 2: Relative Performance of Waltz, CLIPS 6.0 vs. CLIPS++

The system includes a debugger capable of monitoring system execution and inspecting data, and a profiler capable of guiding the user through the program optimization sequence.

3.2 Performance and Matching Technology

The CLIPS++ system employs both published and unpublished optimization techniques developed over the last 12 years[7, 6, 4, 1, 5, 2]. The CLIPS++ system features the LEAPS matching technique asymptotically better than RETE or TREAT. Consequently the performance of CLIPS++ scales with problem size

data size	CLIPS 6.0	CLIPS ++	Speed-Up
12	269	19	14.16
25	116	1 63	18.43
37	253	6 130	19.51
50	462	0 240	19.25

Table 2: Execution Time of Waltz, (seconds), CLIPS 6.0 vs. CLIPS++

as measured by data and throughput requirements[2].

Rather than computing an entire conflict set and then applying a conflict resolution strategy to determine a single rule instantiation to fire, LEAPS folds the conflict resolution strategy into the matcher such that the first instantiation that it discovers on each cycle is the same instantiation that a RETE or TREAT implementation would fire. LEAPS has been formally proven to produce the same execution sequences as the RETE match.

Real-time applications benefit substantially. Since it is much faster, and more predictable to compute only the fired instantiation, CLIPS++ eliminates much of the volatility in match times developers have come to expect from rule systems. The combination of the improved algorithm and the optimization techniques often result in provably optimal code.

3.3 Integration with C++ Class Libraries

Integration with C++ class libraries is a simple matter of including the correct header files and linking with the correct libraries. CLIPS++ can inference over class library objects if they are declared using a defcontainer construct. CLIPS++ can call library functions wherever a standard CLIPS function would be used.

In the above example, the line

```
(last_raise.date ?d &:(< ?d (RWDate 1 1 1994)))
```

uses RWDate objects, which are the date objects from Rogue Wave's Tools.h++ class library. The statement binds an object of type RWDate to the variable ?d, constructs a temporary object with the date 1/1/94, and compares the bound object with the temporary object. The comparison will automatically call the Rogue Wave supplied function

```
operator <(const RWDate& d1, const RWDate& d2)
```

Nearly all of the complexities of C++ class libraries are hidden from the programmer.

The CLIPS++ system has been tested with several class libraries, including the Tools.h++ library from Rogue Wave Software, the C++ Standard Components from USL, and the libg++ library from GNU.

[2] Nearly all claims of scalable performance are based on increasing the size of the rule base, *not* by increasing the size of the working memory

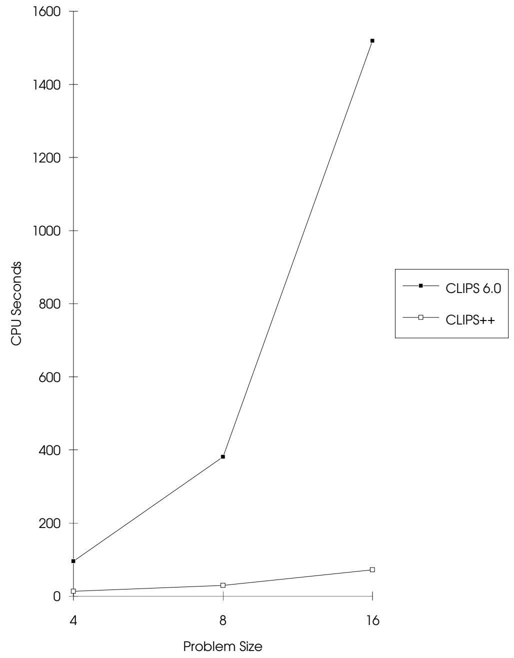

Figure 3: Relative Performance of Waltzdb, CLIPS 6.0 vs. CLIPS++

4 Benchmark Results

We detail the performance of 4 of the 5 programs in the Texas Benchmark suite, Waltz, Waltzdb, Manners and ARP[1][3] Performance results are for CLIPS++ vs. CLIPS 6.0 on standard benchmark programs. All times reported are user + system cpu seconds. The test platform was a Sun Sparcstation 2 running SunOS 4.1.3. Both the test programs and the baseline CLIPS 6.0 were compiled using GNU's gcc version 2.5.8, with highest optimization.

We clearly demonstrate both absolute improvements in speed and very substantial improvement in scalability with respect to data size size. The asymptotic improvement due to the LEAPS match reveals important speed improvement for small data set sizes in the range of 3 to 7. As data set sizes increase, increases in speed are measured by orders of magnitude (i.e. 10x, 100x even 1000x).

[3] Available by ftp from anonymous@cs.utexas.edu, connect to pub/ops5-benchmark-suite. CLIPS versions are also there.

data size	CLIPS 6.0	CLIPS++	Speed-Up
4	95	13	7.31
8	38	1 29	13.14
16	151	9 72	21.10

Table 3: Execution Time of Waltzdb, (seconds), CLIPS 6.0 vs. CLIPS++

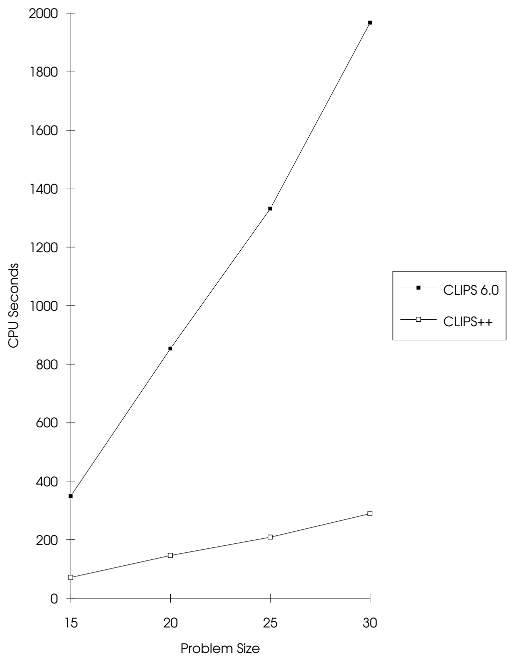

Figure 4: Relative Performance of ARP, CLIPS 6.0 vs. CLIPS++

Data size	CLIPS 6.0	CLIPS++	Speed-Up
15	349	71	4.92
20	853	146	5.84
25	1332	209	6.37
30	1967	289	6.81

Table 4: Execution Time of ARP, (seconds), CLIPS 6.0 vs. CLIPS++

5 Conclusion

The CLIPS++ system is an advanced production system that successfully integrates declarative CLIPS rules with object oriented C++ data types. This integration extends from simple user defined types to complicated class libraries from commercial vendors.

Additionally, the CLIPS++ system is based on the LEAPS algorithm, and contains many published and unpublished performance optimizations. The combination of the asymptotically superior LEAPS algorithm and the optimizations results in a production system of unprecedented performance.

References

[1] D.A. Brant, 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 Data Bases (VLDB)*, 1991.

[2] D.A Brant and D.P. Miranker. Index Support for Rule Activation. In *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, 1993.

[3] J.C. Giarratano. *CLIPS User's Guide, Version 6.0.* Artificial Intelligence Section, Lyndon B. Johnson Space Center, 1994.

[4] D. P. Miranker, 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*, pages 685–692. AAAI, July 1990.

[5] D. P. Miranker, 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.

[6] D. P. Miranker, B.J. Lofaso, G. Farmer, A. Chandra, and D. Brant. On a TREAT Based Production System Compiler. In *Proceedings of the 10th International Conference on Expert System, Avignon, France*, pages 617–630, June 1990.

[7] D.P. Miranker and B. J. Lofaso. The Organization and Performance of a TREAT Based Production System Compiler. *IEEE Trans. on Knowledge and Data Engineering*, pages 3–10, March 1991.