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

and

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 embodiment 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 construct 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.
- Simple integration with existing C++ code.
- Compatible with C++ development tools.
- Execution time is reduced from 5 to 700 times (10 to 20x typical).
- Scalable 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 publicly 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.

1CLIPS++ is a true integration with C++, not a simple wrapper like RETE++.
Table 1: Execution Time of Manners, (seconds), CLIPS 6.0 vs. CLIPS++

<table>
<thead>
<tr>
<th>Data Size</th>
<th>CLIPS 6.0</th>
<th>CLIPS++</th>
<th>Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>3</td>
<td>&lt;1</td>
<td>&gt;3.00</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>&lt;1</td>
<td>&gt;64.00</td>
</tr>
<tr>
<td>64</td>
<td>271</td>
<td>93</td>
<td>906.33</td>
</tr>
<tr>
<td>128</td>
<td>n.a</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>n.a</td>
<td>66</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>data size</th>
<th>CLIPS 6.0</th>
<th>CLIPS ++</th>
<th>Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>269</td>
<td>19</td>
<td>14.16</td>
</tr>
<tr>
<td>25</td>
<td>116</td>
<td>1 63</td>
<td>18.43</td>
</tr>
<tr>
<td>37</td>
<td>253</td>
<td>6 130</td>
<td>19.51</td>
</tr>
<tr>
<td>50</td>
<td>462</td>
<td>0 240</td>
<td>19.25</td>
</tr>
</tbody>
</table>

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

\[
\text{(last_raise.date} \text{?d &:(< ?d (R_/Date 1 1 1994))}
\]

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

\[
\text{operator <(const RWDate& d1, const RWDate& d2)}
\]

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

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

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\(^1\) 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).
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


---

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

<table>
<thead>
<tr>
<th>Data size</th>
<th>CLIPS 6.0</th>
<th>CLIPS++</th>
<th>Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>95</td>
<td>13</td>
<td>7.31</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td>129</td>
<td>13.14</td>
</tr>
<tr>
<td>16</td>
<td>151</td>
<td>972</td>
<td>21.10</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Data size</th>
<th>CLIPS 6.0</th>
<th>CLIPS++</th>
<th>Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>349</td>
<td>71</td>
<td>4.92</td>
</tr>
<tr>
<td>20</td>
<td>853</td>
<td>146</td>
<td>5.84</td>
</tr>
<tr>
<td>25</td>
<td>1332</td>
<td>209</td>
<td>6.37</td>
</tr>
<tr>
<td>30</td>
<td>1967</td>
<td>289</td>
<td>6.81</td>
</tr>
</tbody>
</table>