pCG: An Implementation of the Process Mechanism and an Extensible CG Programming Language

David J. Benn[1]
Dan Corbett[2]

Abstract

Mineau (1998) has developed a state-transition based extension to Sowa's Conceptual Structures Theory (1984) called *processes*, which permits the dynamic update of a knowledge base of conceptual graphs. This formalism is a generalisation of Delugach's (1991) *demons*, and Sowa's *actors* or *dataflow graphs*, although the process mechanism also utilises the latter in practice. This paper presents a general-purpose programming language, *pCG*, which has been developed for the purpose of embodying the process formalism. A pCG implementation of Mineau's 1998 iterative factorial process is given, along with a simple pCG program which addresses the ICCS 2001 CGTools workshop entry criteria. The paper ends with conclusions and possible future work.

1 Introduction

Mineau proposed the notion of *processes* to overcome the fact that Conceptual Structures Theory (CST) does not explicitly cater for the dynamic retraction and assertion of graphs in a conceptual graph (CG) knowledge base (KB) [5]. A number of authors use the term *process* either to describe what their formalism is simulating, or in the case of [5], as the name of a formalism. Sowa remarks that processes can be classified as continuous and discrete [12]. An example of the former is a physical process such as the world weather system. Continuous processes are best simulated on analog computers, although they can be approximated by sufficiently fine-grained discrete processes. A discrete process consists of a series of events and states over time, which is the kind of process that is catered for by Mineau's formalism.

Mineau's processes are one kind of executable conceptual graph formalism. Other work has been carried out in this domain by the CG community and [1] reviews the executable CG literature in considerable detail. Some authors have focussed upon the execution of a graph by special status being given to particular nodes, while others have taken a state transition based approach. Some mechanisms manipulate only concept referents, while others utilise concepts or graphs. A recurring theme is the apparent utility of state transition based systems, of which [5] is an example. Mineau's claim is that his process formalism minimally extends CST. It calls for the following: CGs are grouped into rules consisting of pre and post conditions, an attempt is made to specialise the former to graphs in a KB, and if this succeeds, that KB is updated with arbitrary post-condition graphs. This is repeated for as many times as there are precondition matches.

Mineau suggests that actors and demons are specialisations of processes [5]. Actors take concepts of a predetermined type as input and yield a specialisation (by referent) of a predetermined concept type as output, by means of some computation based upon the input concept referent [10]. Demons consume input concepts and assert other concepts as output [2]. Processes generalise demons by taking CGs as input and asserting or retracting CGs as the result of their processing. The relationship is not really this simple however, since as will be seen, Mineau's 1998 factorial example uses actors within preconditions. So, on the one hand we have the suggestion that there exists a generalisation hierarchy between actors, demons and processes, ie `actor < demon < process`. On the other hand, there is at least one example of processes using actors as part of their definition, essentially a *uses* or *has-a* relationship. It is arguable that without actors, process preconditions of even moderate complexity would not be possible since arbitrary

[1] School of Computer and Information Science, University of South Australia, Adelaide, 5095, e-mail: David.Benn@motorola.com
[2] School of Computer and Information Science, University of South Australia, Adelaide, 5095, e-mail: corbett@cs.unisa.edu.au

referent values could not be easily computed or manipulated. A concept is a singleton graph, so a case can be made for a process being a generalisation of a demon as Mineau has suggested, especially since assertion and retraction is the primary mechanism for both.

Correspondence with Mineau [7] confirmed that there had been no implementation of the process mechanism, "…only a theory which needs to be refined, implemented and applied." That is exactly what the work described herein entails. A major motivation for any implementation of the process mechanism is to prove whether it works, and if so, in what ways it is deficient or requires improvement. Delugach remarks that: "For the most part, implementations are not considered research. My experience with implementation (both with CGs and elsewhere) shows that these efforts help validate a theory." [3]

Someone wishing to use processes should be able to do so by expressing them directly in a suitable source language, so as to be able to work with the fundamental entities as first class objects. Mineau provides the beginnings of such a language in [5], and that paper's stated long term goal of the development of a Conceptual Programming Environment combining logic and imperative programming suggests a direction. Such a language would also provide a means to embody the process engine. An alternative is to develop an Application Programming Interface (API) in a popular language, such as C++ or Java. One problem with this approach is that the complex details of using an API can quickly obscure the problem domain. For the purpose of understanding the likely use of processes and for ease of their application to a particular problem, it was decided that a language (pCG) and an interpreter for it be developed. The pCG distribution is available on the web[3], and is available under the Gnu Public Licence. We consider pCG to be an experimental language and believe that feedback from the CG community will be useful at this point in time.

2 The Design of pCG

One ought to approach the design of a programming language with some trepidation as it is fraught with dangers. One strategy to mitigate the risk of failure is to decouple syntax from semantics to the extent possible, so that the source language can change easily if desired. The focus then shifts to getting the semantics right. The following design goals have guided the development of pCG: 1) Making those entities of primary interest to the developer first class; 2) easy extensibility; 3) rapid development; 4) portability; 5) minimality.

The first goal is satisfied by making concepts, graphs, actors, and processes first class in the language. Numbers, booleans, strings, lists, files, and functions are also first class types. Values of all types may be passed as parameters to functions or returned from them.

The second goal is made possible by exposing the run-time system as a set of Java classes to which attributes and operations may be added by way of methods following simple conventions. After recompilation of the relevant class, the new attributes or operations become available in the pCG language. Further, new types may be added to the language by creating subclasses of a particular Java class and creating instances using pCG's `new` operator. In both cases, no changes need to be made to the interpreter.

The third goal is realised in a number of ways: a) Since the core functionality of the language lies in the run-time system, there is little to be gained from compiling pCG to bytecodes or machine code. Like Perl, pCG's interpreter executes abstract syntax trees, an efficient intermediate representation. So, pCG programs can be executed with acceptable speed, removing "compile" from the edit-compile-run cycle. b) Memory allocation in pCG is garbage collected. c) The built-in types provide the essential features required for working with CGs, Mineau's processes, and for general programming. The developer can spend more time focussing upon the problem itself, rather than upon how to represent it. This is also a consequence of the first design goal.

[3] See http://www.adelaide.net.au/~dbenn/Masters/ for the archive and other information pertaining to pCG.

The fourth goal of portability is achieved by virtue of pCG's basis in Java, and the Java-based compiler construction tool, ANTLR[4].

The fifth goal is that the language should be *minimal.* The basic philosophy of pCG is that there should be few special statements and functions and that computation should proceed through interaction with the fundamental objects of the language. Where a statement or function does appear in isolation from an object, it is because the internal state of the core interpreter itself must be modified, there is no object with which to associate it, or it eases use of the language.

CG tools (eg [4]) have been developed which support purely visual programming. We have taken the approach of acknowledging that traditional programming languages have a role to play. In short, CGs are used where knowledge representation is required, while traditional constructs are used where imperative or functional programming is most appropriate. Languages such as Perl (lists, `foreach`), Lisp (dynamic typing, lambda), Java (objects), and Prolog (assert, retract, pattern matching) have influenced pCG.

3 Key Semantic Features

The pCG language is characterised by a few key features. It is *dynamically typed* since values not variables determine type. pCG is *lexically scoped* since functions and processes introduce a new lexical scope when invoked. Wegner (1987) says that an object is an entity that has a set of operations and a state which remembers the effect of those operations. He defines an object-based language as one which "…supports objects as a language feature" but cautions that the support of objects "…is a necessary but not sufficient requirement for being object-oriented. Object-oriented languages must additionally support object classes and class inheritance." [13]. On these grounds, pCG can be characterised as an *object-based* language since its fundamental types have state and operations on that state. However, pCG does not support the creation of arbitrary object types within the language, but as noted above, it is possible to add new types to the language using Java, and to modify intrinsic types. All values in pCG are objects with associated *attributes* (eg length), *operations* (like methods in Java), and *operators* (eg +). Some attributes may appear on the left hand side of an assignment statement. All attributes may appear in expressions. Objects have a type attribute indicating the name of a value's type, and the *is* operator can be used to determine this type for any value at run-time.

The pCG language is *multi-paradigm*, since apart from its object-based characteristic, pCG supports *imperative* (variables, assignment, operators, selection, iteration), *functional* (higher order functions, values, recursion), and *declarative* styles of programming. The latter is supported in the form of processes, since one specifies rules containing pre and post conditions representing knowledge. The details of testing preconditions against the KB, and the assertion/retraction of post-conditions in the KB are left to the process execution engine.

4 Actors

Consider the following function and actor definitions in pCG:

```
function sqr(n,m)
  nVal = n.designator;
  if not (nVal is number) then
    exit "Input operand to " + me.name + " is not a number!";
  end
  m.designator = nVal*nVal; // designator is an example of an attribute
end

actor SQR(x) is `<sqr [Num:'*x'] | [Num:'*y']>`;
```

The actor defines a dataflow graph which contains an actor node or *executor*. This executor is defined in terms of a pCG function, `sqr`, which takes a source and a sink concept as parameters. The sink concept's designator is mutated according to the square of the source concept's designator, which is first tested to

[4] See http://www.antlr.org/.

ensure it is a number. In the actor definition, `*a` and `*b` are defining variables, `?a` and `?b` are bound variables for identifying concepts, while `*x` and `*y` are variables representing designator values. Such variables derive from the original actor notation of [10].

One way to invoke this actor is to write: `g = SQR(4)` which results in a mutated copy of the defining graph being assigned to `g` thus: `<sqr [Num:4] | [Num:16]>.`

A point of departure in pCG's implementation of actors compared to [10] is that only the input parameter is specified in the actor definition's parameter list for the purpose of binding. The actor mechanism can otherwise determine the correct order in which to pass concepts to a particular executor, so long as the arcs are correctly ordered [5]. Assuming the executor (a function or other dataflow graph) performs correctly, the returned mutated graph copy will have appropriately mutated sink concepts. The above invocation method is not useful when an actor appears in a process rule's precondition. An invocation graph may be constructed and the graph activated directly, e.g.

```
n = 4;
mySqrGrStr = "<sqr [Num:" + n + "] | [Num:'*y']>";
g = activate mySqrGrStr.toGraph();
```

Notice that no actor definition is required here. The graph is constructed as a string, the source concept's referent is bound in that string, and the string is converted to a graph which is then activated, returning the mutated graph copy. This is akin to actor invocation that occurs during process execution (see next section), except that the process engine implicitly activates precondition graphs containing actors.

5 Processes

An abstract process is defined in [5] as:

process p(in g1, out g2,...) is $\{ r_i = \langle pre_i, post_i \rangle, \forall_i \in [1, n] \}$

In pCG, the syntax for a process is:

```
'process' name '(' in | out parameter [, …]')'
    ['initial' block]
    ('rule' ident
       [option-list]
       'pre'
          ['action' block] (['~'] pre-condition)*
       'end'
       'post'
          ['action' block] (post-condition [option export])*
       'end'
    'end')*
'end'
```

A process has a name and a list of `in` and `out` parameters, followed optionally by a block of code for miscellaneous initialisation purposes, followed by a set of zero or more rules. Each rule consists of an arbitrary identifier, optionally followed by a list of one or more options for the rule, and a pre and post condition section. A precondition section consists of an optional action block and zero or more possibly negated graph expressions. A post-condition section consists of an optional action block and zero or more possibly exported contexts.

[5] Tools such as Delugach's CharGer can help here by permitting explicit arc numbering, ensuring the correct ordering in the generated CGIF. Mineau acknowledges the importance of arc ordering in process invocation graphs also [5].

An initial code block consists of arbitrary pCG code. The intent of such blocks is to aid in the debugging of processes, to provide useful output during process execution, or to combine procedural and declarative programming styles. Such code may also be used to construct graphs which are subsequently used in precondition graph matching, or post-condition graph assertion or retraction.

A precondition is an arbitrary graph expression. If preceded by a '~' character, the sense of the match for that graph is reversed[6]. A post-condition is a context with one of the following concept types: PROPOSITION or ERASURE, corresponding to assertion or retraction.

An input parameter must be a context and have a type of PROPOSITION or CONDITION. The descriptor of a PROPOSITION context is asserted before process execution starts, the intention of which is that such a graph will act as a trigger which (along with possibly other asserted graphs) causes a suitable rule to fire. A CONDITION context's descriptor graph is added to the precondition list of the first rule of a process. Output parameters are appended to the post-condition list of the last rule. When this rule is reached, the process is terminated after asserting or retracting the specified graphs in the caller's KB depending upon whether the context's concept type is PROPOSITION or ERASURE, respectively. [5]

Contexts with concept types PROPOSITION, ERASURE, and CONDITION are essentially used as a packaging mechanism. PROPOSITION and ERASURE context types are also used in the body of post-conditions to distinguish between assertions and retractions. An alternative would be to have each post-condition block subdivided into assertion and retraction blocks.

When a process is invoked, the first rule is retrieved, and each graph of the rule's precondition is tested in turn. If one does not match, matching for that rule is discontinued and the next rule is retrieved. If no matching rule is found, the process terminates. If one is found, the post-conditions of the matched rule are retrieved. For each post-condition, its descriptor graph is either asserted or retracted — depending upon the type of its context — from the local KB, or the caller's KB if an export option is active. The process engine then starts again at the top of the ordered rule collection, and attempts to find a matching rule during the next cycle.

A process invocation introduces a new KB scope. So, there are two run-time stacks in pCG: one for variables and another for KBs. There is one of each at the top-level for global code execution. KBs in pCG store concept and relation type hierarchies, and a set of graphs. The contents of a process caller's KB are copied to the invoked process's KB. Look-up and assertion/retraction is confined to the KB which is top-most on the stack. This prevents anything except a process's parameterised output graphs from mutating the caller's KB[7], as per [5]. The alternative is a proliferation of graphs in the caller's KB, meaningless outside of a given process, which would need to be retracted by some other means upon exit from that process.

Projection forms the basis of many practical CG systems [8]. The projection operation is central to the matching algorithm of pCG's process execution engine, and the algorithm used by pCG is similar to that given in Theorem 3.5.4 of [10] except that relation type subsumption is also permitted.

6 Mineau's Iterative Factorial Example in pCG

The partially specified example of an iterative factorial process from [5] has been fully specified in pCG and is shown below. The most noteworthy aspect of this is that it proves that Mineau's process formalism works substantially as laid out in his paper. The iterative factorial C program for which Mineau defined some process rules is shown below. Three functions are used by actor executors in the preconditions of the process, the first of which is `LTorEQ` (shown below). Definitions for `Multiply` and `Add` executors are similar.

[6] This is a potentially confusing overloading of the tilde which is already used in the negated context syntax, and so may be changed in the future.

[7] The `option export` directive —an addition to [5] — provides a way to circumvent this restriction.

```
L0: int fact(int n)
L1: { int f;
L2:   int i;
L3:   f = 1;
L4:   I = 2;
L5:   while (I <= n)
L6:   {  f = f * i;
L7:      i = I + 1;
L8:   }
L9:   return f; }
```

The C program from [5].

```
function LTorEq(a,b,result)[8]
  first = a.designator;
  second = b.designator;
  if not (first is number) or
     not (second is number) then
    exit "The first 2 arguments to " +
         me.name + " must be numbers.";
  end
  result.designator = first <= second;
end
```

The actor executor function LTorEQ.

The following code defines the iterative factorial process, consisting of 8 rules, compared to Mineau's original 12. Line concept referents refer to the lines in the C program that the process simulates. Mineau's original 1998 rule numbers are preserved for ease of comparison, but no rules equivalent to those in [5] numbered 3, 5, 9, or 11 appear in the definition below as they are unnecessary. See [1] or the distribution example code for further discussion of the omitted rules.

Note that the style of CGIF in this example corresponds to the draft proposed ANSI Standard[9] of August 1999, a predecessor to [11], but pCG is capable of reading and writing both this and the newer June 2001 CGIF syntax. See the pCG distribution for a version of the code[10] which is compliant with the June 2001 CG Standard, the target of the CGTools workshop[11]. The main differences between the standard versions with respect to this example are that: 1) defining labels must now either appear before the colon (i.e. before the referent) or after the referent, and 2) all numeric literals in a referent's designator must be preceded by a sign. There are a variety of other differences between the old and new standards, such that pCG now affords an option to programmatically switch between CGIF parsers and generators. For the purpose of discussing this program, the exact CGIF version is not important however. For example, three well-formed CGIF syntaxes (based upon the latest CG standard) for rule r1's first precondition graph are:

```
[Integer*a:'*x'][Variable*b:'#n'](val?b?a)

[Integer:'*x'*a][Variable:'#n'*b](val?b?a)

(val [Variable:'#n'] [Integer:'*x'])
```

Note that the single-quoted locator strings here are name designators, not indexicals or defining labels, although the former *could* have been used. pCG treats a string such as `'*x'` as a special kind of variable which may be bound to a particular value during precondition matching. See [1] or the user manual supplied with the pCG distribution for more regarding this.[12]

Now we move on to the iterative factorial process in earnest. The first rule simply establishes whether a variable `n` has a value, while rules 2 and 4 assert graphs representing the variables `f` and `i`.

```
process fact(in trigger, out result)
  rule r1
```

---

[8] Identifiers in pCG may only contain alphanumeric characters or underscores, hence an executor name like `<=` is not possible, since a function with that name cannot be defined.

[9] http://users.bestweb.net/~sowa/cg/cgdpansw.htm

[10] `cgp/examples/CGTools01/ItFactProcess.cgp`
This version of the program also makes less use of coreference labels, enhancing readability.

[11] http://www.cs.nmsu.edu/~hdp/CGTools/cgstand/cgstandnmsu.html

[12] As with their use in actors, such special variables might be confused with defining labels, and so may be changed in the future. One possibility is to use a Perl-like syntax, eg `[Integer*a: '$x']`. Since such a name designator (locator) is an implementation-specific means by which to locate a designator, this would not be a violation of the standard syntax.

```
  pre
    `[Integer:*a'*x'][Variable:*b'#n'](val?b?a)`;
    `[Line:*a'#L0'](to_do?a)`;
  end

  post
    `[ERASURE:[Line:*a'#L0'](to_do?a)]`;
    `[PROPOSITION:[Line:*a'#L3'](to_do?a)]`;
    `[PROPOSITION:[Line:*a'#L4'](to_do?a)]`;
  end
end // rule r1

rule r2
  pre
    `[Line:*a'#L3'](to_do?a)`;
  end

  post
    `[PROPOSITION:[Integer:*a 1][Variable:*b'#f'](val?b?a)]`;
    `[ERASURE:[Line:*a'#L3'](to_do?a)]`;
  end
end // rule r2

rule r4
  pre
    `[Line:*a'#L4'](to_do?a)`;
  end

  post
    `[PROPOSITION:[Integer:*a 2][Variable:*b'#i'](val?b?a)]`;
    `[ERASURE:[Line:*a'#L4'](to_do?a)]`;
    `[PROPOSITION:[Line:*a'#L5'](to_do?a)]`;
  end
end // rule r4
```

Rule 6 compares the values of `i` and `n`, asserting the number of the line for which execution is to be simulated next, thus determining whether the body of the simulated while loop should be executed. If `[Line:*a'#L6'](to_do?a)` is asserted, rule 8 fires on the next round, corresponding to the start of the loop body.

```
rule r6
  pre
    `[Line:*a'#L5'](to_do?a)`;

    `[Integer:*a'*first'][Variable:*b'#i']
     [Integer:*c'*second'][Variable:*d'#n']
     [Boolean:*e"true"]
       (val?b?a)
       (val?d?c)
       <LTorEq?a?c|?e>`;
  end

  post
    `[ERASURE:[Line:*a'#L5'](to_do?a)]`;
    `[PROPOSITION:[Line:*a'#L6'](to_do?a)]`;
  end
end // rule r6
```

Rule 7 (not shown) is identical to rule 6 except that `[Boolean:*e"false"]` is used in the actor, and `[PROPOSITION:[Line:*a'#L9'](to_do?a)]` is asserted to exit the loop. The following rule retracts a graph representing a binding of `f`, and replaces it with another graph which represents a new binding to f.

Rule 10 (not shown) is very similar to rule 8, except that the assignment being represented is `i=i+1` which corresponds to line 7 of the C program.

```
rule r8
    pre
      `[Line:*a'#L6'](to_do?a)`;

      `[Integer:*a'*z4'][Variable:*b'#f']
       [Integer:*c'*x'][Variable:*d'#i']
       [Integer:*e'*y4']
         (val?b?a)
         (val?d?c)
         <Multiply?a?c|?e>`;
    end

    post
      `[ERASURE:[Integer:*a'*z4'][Variable:*b'#f'](val?b?a)]`;
      `[PROPOSITION:[Integer:*a'*y4'][Variable:*b'#f'](val?b?a)]`;
      `[ERASURE:[Line:*a'#L6'](to_do?a)]`;
      `[PROPOSITION:[Line:*a'#L7'](to_do?a)]`;
    end
  end // rule r8
```

Rule 12 results in the final assignment to f being exported as a graph to the caller's KB.

```
rule r12
    pre
      `[Line:*a'#L9'](to_do?a)`;
      `[Integer:*a'*z5'][Variable:*b'#f'](val?b?a)`;
    end

    post
      `[ERASURE:[Line:*a'#L9'](to_do?a)]`;[13]
    end
  end // rule r12
end // process fact
```

A graph representing the variable `n` for which the factorial is to be computed is next constructed and asserted in the caller's KB:

```
n = 7;
varN = "[Integer:*a " + n + "][Variable:*b'#n'](val?b?a)";
assert varN.toGraph();[13]
```

The next few lines create a process invocation graph. The first parameter to the process is a trigger graph which is asserted by the process engine to indicate that the line to start on is the zeroth. The second parameter is an integer result concept that will be specialised to a particular value by the final rule of the process.

```
s = "[PROPOSITION:*a[Line:*b'#L0'](to_do?b)]" +
    "[PROPOSITION:*c[Integer:*d'*z5']]" +
    "<fact?a|?c>";
g = s.toGraph();
```

These three lines show the top-level KB contents before and after activating the process invocation graph:

```
println "Before process 'fact'. Graphs: " + _KB.graphs;
x = activate g;
```

[13] A graph may be parsed from a string value. A graph literal is enclosed in grave accents.

```
println "After process 'fact'. Graphs: " + _KB.graphs;
```

The result of executing this code is:

```
Before process 'fact'. Graphs: {[Integer:*a 7.0] [Variable:*b'#n']
(val?b?a)}
After process 'fact'. Graphs: {[Integer:*a 7.0] [Variable:*b'#n']
(val?b?a), [Integer: 5040.0]}
```

7 Differences from Mineau's Example

The pCG version of the factorial process differs in some ways from Mineau's paper-based example. In general rules have been optimised, omitting unnecessary actors and post-condition graphs. See [1] for more details. Differences with respect to graph format and graph negation are briefly discussed here.

7.1 Graph Format

In the pCG code, CGIF is used to represent graphs rather than LF. A future version of pCG may also support LF[14]. For now, complex graphs may be drawn with a tool such as CharGer [3] and stored in a file for later use in a pCG pre or post-condition. As suggested in [5] page 68, locators have been used for line numbers, eg #L2, instead of the use of referent variables such as `*L2` found elsewhere in Mineau's example, since in all cases, particular individual line numbers are being referred to. Similarly, locators have been used to represent C variable names, rather than defining names (eg `#f` vs `*f`) to avoid match ambiguity, eg see `pre6` actor on page 71 of [5].

7.2 Erasure vs Negation

While pCG permits negated precondition graphs, a simple model of assertion and retraction (via graph erasure, not negation) is employed for post-conditions. In [5], the terms retraction and negation are sometimes used interchangeably. For example, in [5] we find the following (my italics):

> …all graphs in a post-condition are asserted, except those preceded by the *negation sign*, which are *retracted*. Of course, assertions and retractions may be without effect as asserted and negated graphs may already be known to be true or false respectively.

Were negation used, every time a new value is assigned to `f`, a graph with the new value would be asserted, and the graph representing the previous value would be negated. This would have the effect of saying increasingly more about what the value of f is *not*, rather than what the value of f actually *is*, an example of the Frame Problem [12]. As another example of the difference between the two approaches, the precondition of rule 2 in [5] checks that the graphs `[Line:#L3]<-(to_do) and [Line:#L0]<-(done)` have been asserted, while rule 2's post-condition negates `[Line:#L3]<-(to_do)` and asserts `[Line:#L3]<-(done)`. The approach in pCG of just asserting and retracting graphs of the form `[Line:#Ln]<-(to_do)` is simpler and more efficient.

8 pCG in conjunction with the CGTools Workshop

This simple but complete pCG program reads one CGIF graph from each file specified on the command-line, prints equivalent CGIF where each concept has a defining variable and relations use corresponding bound variables, and prints each relation and its arguments. Any comments embedded in the graph (eg by tools such as CharGer) are omitted from the output.

```
foreach path in _ARGS do
  f = file path;
  g = f.readGraph();
  f.close();

  println g.nocomments();
```

[14] Concept and graph parsing relies upon Notio's functionality [9].

```
    println "";
    foreach rel in g.relations do
      relType = rel[1]; inArgs = rel[2]; outArgs = rel[3];
      println relType + "(" + (inArgs.length + outArgs.length) + ")";
    end
  end
```

The program could easily be extended to read multiple graphs per file by reading each line with `f.readline()`, treating blank lines as graph delimiters, concatenating each non-blank line as a string with the + operator, and applying the `toGraph()` operation to the resulting string. An example input graph and corresponding program output (italicised) are shown below.

```
(provide[primary_market*x0][investment*x1])(attr?x1[new])
  (operate_with?x0[stock*x3])(attr?x3[newly_issued])

[primary_market:*a][investment:*b][new:*c][stock:*d][newly_issued:*e]
  (provide?a?b)(attr?b?c)(operate_with?a?d)(attr?d?e)

provide(2.0)
attr(2.0)
operate_with(2.0)
attr(2.0)
```

A more complex version of this program (`cgp/examples/CGTools01/basic-info.cgp`) can be found in the pCG distribution, which parses and processes the CGTools basic graph set, while another program (`cgp/examples/CGTools01/final.cgp`) parses the final CGTools graphs. When applied to the second final graph file, the latter program also projects a filter graph onto the parsed graphs, looking for those containing a particular relation. Similar code could be written to explore semantic aspects of the other final graphs, rather than just parsing and re-generating them from their internalised forms. See also Appendix A re: example programs.

9 Conclusions and Future Work

An implementation of Mineau's processes now exists in the form of pCG. The value of this work is that consideration of processes need no longer be a paper-based exercise. Instead, processes may be constructed, then executed, allowing the limitations and strengths of the mechanism to be explored. The pCG language has been applied to some problems where dynamic knowledge base update is required, including Mineau's 1998 example factorial process and the Sisyphus-I room allocation problem. [1] Given other executable CG systems and appropriate tools, researchers will be able to pose and answer questions by way of comparison to the current work, such as: Which is more powerful or expressive? Which is simpler to work with? To which domains is a particular formalism best suited? In addition to solving problems directly in the pCG language, it could also be treated as a target language for more complex systems.

The pCG language represents process and actor invocation graphs using a single (standard) actor node rather than the special double-diamond node suggested by [5] and [2]. Also, pCG can interoperate with tools such as Delugach's CharGer graph editor with which complex graphs can be drawn and saved as CGIF [11], a format that is readable by pCG programs.

In the conclusion of [5], the author points out that processes would have problems relating to soundness, completeness, and consistency. For example: Do two post-conditions conflict? Are they independent? This is in part a truth maintenance issue, i.e. preserving the consistency of propositions in a KB as it changes over time. [6] proposes another use for contexts which involves imposing constraints on processes, and a logical next step would be to add the constraint functionality specified in that paper to pCG. This would require adding a step after precondition matching to ensure that applying the corresponding rule's post-conditions would not lead to an invalidated KB. Other work remains. The conjunction of graphs in a precondition should be treated as a goal, rather than as isolated conditions. The conformity relation should be implemented [10]. The process algorithm needs to be reconciled with the CG Rules of Inference of [11]

and incorporated into pCG. Some aspects of pCG's syntax are likely to be changed for the sake of readability. Finally, aspects of the process algorithm bear re-examination, eg stopping criteria, graph matching morphology and efficiency.

Appendix A: Installing pCG

Installing pCG is simply matter of unzipping the supplied archive, for example using WinZip under Windows or zip under Unix or Linux. The archive may be unzipped in a location of your choosing, leaving you with a directory called `cgp` (Conceptual Graph Processes). A `README` file at the top-level of the cgp directory suggests that you add the pCG invocation script to your path and guides you through running an example pCG program.

The pCG invocation script is platform dependent: a Bourne Shell script for Unix, and a batch file for Windows. The batch file looks like this:

```
@echo off
rem DOS batch file to invoke Conceptual Graph Processes interpreter.

rem pCG root directory. Change "c:\cgp\" to your installation directory.

java -classpath
c:\cgp\lib\antlr.jar;c:\cgp\lib\Notio.jar;c:\cgp\lib\cgp.jar;c:\cgp
cgp.CGP %1 %2 %3 %4 %5 %6 %7 %8 %9
```

If `java.exe` is not on your path, you can either add it to the `PATH` variable in `c:\autoexec.bat` under Win9x, or instead fully qualify `java` in the batch file, eg

```
c:\jdk1.2.2\bin\java -classpath …
```

Similarly, under Win9x, you can add `pCG` to your path by adding to the `SET PATH` command in `c:\autoexec.bat`, or fully qualify it upon invocation, ie

```
root\cgp\pCG …
```

where *root* is the directory in which the pCG archive was unpacked.

Here are the details of running two such examples after the pCG invocation script has been tailored for a particular system:

- On a Windows machine, open a MS-DOS shell and change directory to *root*`\cgp\examples\`
- Run pCG with `final.cgp` as the sole argument.
- Read that program's comments for more information about optional command-line arguments and program function.
- A minimal pCG program is also present in that directory, `tiny.cgp`, which opens a single file, reads a June 2001 CG Standard compliant CG stream into a list, and prints each graph in it. Here is the complete program:

```
option CGIFparser = "cgp.translators.CGIFParser";
option CGIFgen = "cgp.translators.CGIFGenerator";

if _ARGS.length == 1 then
  f = file (_ARGS[1]);
  graphs = f.readGraphStream();
  f.close();
  foreach g in graphs do
    println g;
  end
```

```
end
```

- Assuming pCG is on your path and the above directory is your working directory, you could type the following to run `tiny.cgp`:

```
pCG tiny.cgp final/final_graphs_level1.cgf
```

Consult the aforementioned README file for more detailed information regarding platform-specific installation, example programs, documentation, and other useful notes.

Appendix B: pCG Requirements

pCG *requires* Java 2, in particular JDK/JRE 1.2.2. The authors make no guarantees about whether pCG will work with any other version of Java at this time.

pCG comes with the extra Java class libraries it requires in `cgp/lib`, specifically those for Antlr (2.7.0) and Notio (0.2.2).

pCG has been tested under Linux 2.2 (RedHat 6.0), Windows 98 (550 MHz Pentium III in both cases), Solaris, and SunOS, but primarily the first two. Despite being Java software, the authors make no claims about whether pCG will run on other operating systems, but would be interested to know of anyone using an operating system apart from the ones mentioned, e.g. Macintosh.

The pCG archive is slightly less than 1.5 megabytes in size and when expanded, occupies approximately 4 megabytes.

References

1. Benn, D., An Implementation of Conceptual Graph Processes. Master of Computer and Information Science Minor Thesis, December 2000, University of South Australia.

2. Delugach, H., Dynamic Assertion and Retraction of Conceptual Graphs, *Proceedings of the 6th Annual Workshop on Conceptual Graphs*, Eileen C. Way, editor, SUNY Binghampton, New York, pp. 15-26, July 1991.

3. Delugach, H., CharGer: Some Lessons Learned and New Directions., In *Proceedings of the 8th International Conference on Conceptual Structures*, Darmstadt, Germany, pp. 306–309, August 2000.

4. Kabbaj, A., Synergy: A Conceptual Graph Activation-Based Language, In *Proceedings of the 7th International Conference on Conceptual Structures*, Blacksburg, VA, USA, pp.198–213, July 1999.

5. Mineau, G., From Actors to Processes: The Representation of Dynamic Knowledge Using Conceptual Graphs. In *Proceedings of the 6th International Conference on Conceptual Structures*, Montpellier, France, pp. 65–79, August 1998.

6. Mineau, G., Constraints on Processes: Essential Elements for the Validation and Execution of Processes. In *Proceedings of the 7th International Conference on Conceptual Structures*, Blacksburg, VA, USA, pp. 66–82, July 1999.

7. Mineau, G., E-mail correspondence from Guy Mineau to David Benn, December 1st, 1999.

8. Mineau, G., The Engineering of a CG-Based System: Fundamental Issues, In *Proceedings of the 8th International Conference on Conceptual Structures*, Darmstadt, Germany, pp.140–155, August 2000.

9. Southey, F. & Linders, J.G., Notio — A Java API for developing CG tools. In *Proceedings of the 7th International Conference on Conceptual Structures*, Blacksburg, VA, USA, pp. 262–271, July 1999.

10. Sowa, J.F., Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley, 1984.

11. Sowa, J.F. et al, Conceptual Graph Standard, American National Standard NCITS.T2/98-003, 2000. [Accessed Online: April 2000], URL: http://www.bestweb.net/~sowa/cg/cgstandw.htm[15]

12. Sowa, J.F., Knowledge Representation: Logical, Philosophical, and Compuational Foundations, Brooks/Cole, 2000.

13. Wegner, P., Dimensions of Object-Based Language Design, In OOPSLA '87 Proceedings, pp. 168–182, October 4–8, 1987.

[15] A more recent version of this document can be found at http://www.jfsowa.com/cg/cgstand.htm