An Application of the Process Mechanism to a Room Allocation Problem using the pCG Language

David Benn[1], Dan Corbett[2]

[1] School of Computer and Information Science, University of South Australia, Mawson Lakes, 5095
`David.Benn@motorola.com`
[2] School of Computer and Information Science, University of South Australia, Mawson Lakes, 5095
`corbett@cs.unisa.edu.au`

Abstract. The Sisyphus-I initiative consists of a constraint satisfaction problem in which a group of people in a research environment must be allocated rooms. Numerous constraints are detailed by the problem description which together impose a partial ordering on any solution. A solution to Sisyphus-I is presented in order to illustrate pCG, a CG-oriented programming language which embodies Mineau's (1998) state-transition based process mechanism. We consider pCG to be an experimental language and believe that feedback from the CG community would be useful at this stage of development. A non-trivial application with which the community is already familiar is an effective means by which to accomplish this. The solution involves automatic extraction of most of the information required to represent the problem from the Sisyphus-I web page, and a pCG program which produces suitable room allocations via a process. The means by which the presented solution could be further constrained to increase its robustness is briefly discussed, as is the likely future development of pCG.

1 Sisyphus-I

Sisyphus-I was used at ICCS'99 as an example of a non-trivial problem that could be represented using conceptual graphs (CGs) [9]. In this paper, we show a solution to the problem using Mineau's process mechanism [3], [13]. The problem statement [9] calls for "…traces of the problem-solving processes...This means that…we want descriptions of operational problem solvers."

The Sisyphus-I room allocation problem is a constraint satisfaction problem in which a group of people in a research department each with particular needs must be allocated rooms. It is a means by which to test knowledge acquisition and reasoning strategies. Numerous constraints are detailed by the problem description and are intended to impose a partial ordering on any solution. One constraint is that the head of the YQT research department must be allocated alone to a large room. Another is that smokers and non-smokers may not share rooms. Yet another is that researchers should share rooms with people from different projects to prevent insularity. There are two slightly different problem statements given in [9]. The difference between them is that Katharina, the head of a project and a smoker is replaced by Christian, a researcher and smoker. Katharina is immediately eligible for a single room by virtue of her lofty status as a project lead. However, Christian ought to share a room with another person due to his more lowly status, unless doing so would place him with a non-smoker, or unless there are only single rooms remaining.

Many techniques were used in the 1999 solutions of the original Sisyphus-I initiative. Agents were used to represent the knowledge in [19] while other authors used a rule base, either for the meta-knowledge [5] and [1] or for the domain facts [12]. Mineau's solution in [15] was to represent the constraints and goals on graphs, and then use subsumption to compare graphs and find falsehoods, which represent invalid states. A similar method for finding invalid states, using projection, was employed by [1].

While our solution, on the surface, seems only to represent yet another rule-based solution, the rules are represented by dynamic CGs with processes. The significance of our solution is that the graphs are used to represent both the declarative facts of the current context and the process knowledge of the domain. Projection is the basis for graph matching in the solution presented here, with type subsumption employed in one of the rules, and restriction on concepts used extensively elsewhere in the process definition. Matched graphs representing rooms and people are retracted from a knowledge base (KB) as constraints are satisfied, with room allocation graphs being asserted into the KB. The frame problem[1] [18] is avoided since graphs are retracted, not negated, decreasing the burden for future KB searches.

2 Processes

Mineau proposed the notion of *processes* to overcome the fact that Conceptual Structures Theory (CST) does not cater for the dynamic retraction and assertion of graphs in a CG-based KB [13]. Mineau's processes are one kind of executable conceptual graph formalism. Other work has been carried out in this domain by the CG community, for example [2], [4], [6], [7], [8], [10], [11], [16], [17]. 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 approach, or both. Some mechanisms manipulate only concept referents, while others utilise concepts or graphs.

A number of authors use the term *process* either to describe what their formalism is simulating, or in the case of [13], as the name of a formalism. In [18] Sowa remarks that processes can be classified as continuous and discrete. 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.

A recurring theme is the apparent utility of state transition based systems, of which [13] is an example. Some authors use state-based systems to solve particular problems, such as [2], [11], and [16]. Sowa remarks that State Transition Diagrams are a common way of representing discrete processes. Sowa discusses Petri nets as a powerful and generic state transition mechanism. [18]

Mineau suggests that actors and demons are specialisations of processes [13]. 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 [17]. Demons consume input concepts and assert other concepts as output [6]. 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, for example, Mineau's 1998 factorial example uses actors within preconditions. So, on the one hand we have the suggestion that there exists a

---

[1] Increasingly more is asserted about what is *not* the case, eg that a room is no longer vacant.

generalisation hierarchy between actors, demons and processes, ie `actor < demon < process`. On the other hand, there is at least one example of a process using an actor as part of its 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 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.

Mineau's claim is that his process formalism minimally extends CST. Essentially, [13] calls for the following: CGs are grouped into rules consisting of pre and post conditions, an attempt is made to specialise a rule's preconditions to graphs in a KB, and if this succeeds, the KB is updated using post-condition graphs. This is repeated as many times as there are precondition matches. The first rule's preconditions and the final rule's post-conditions may be added to via parameters.

Lukose describes Executable Conceptual Structures (ECS) in [10], an executable conceptual modelling language. Components of the system include: concept types (differentia graphs defined in terms of a lambda expression); actor types (scripts with a main method); precondition and post-condition graph lists; lists of graphs to be deleted; a notion of short and long term working memories. ECS also provides special concepts representing predicates (eg `[LTEQ]`) and statements (eg `[ASSERT]`), along with constructs to express temporal relationships, and others to package problem solutions. When taken together, complex systems can be built for problem solving. A complete description of ECS cannot be provided here. What can be said however is that ECS provides a similar capability to Mineau's process formalism, at the expense of a greater number of parts.

Cyre asks the question: given an arbitrary CG, what would be required to execute it? He points out that complex, possibly hierarchical graphs may be difficult to reason about, so may require simulation to understand, and believes that simulation can be useful for predicting the consequences of behaviour described by a CG. A ball-throwing example is given in [4] involving a CG containing a `Throw` action concept. Cyre laments that most researchers have extended CST with special nodes (eg actors or demons), and suggests that instead, `Throw` be associated with some procedure which "knows" what kinds of concepts it can work with, and how `Throw` affects them. Here, concepts rather than relations are being used as the active elements in a graph. What this amounts to, however, is that certain types must be marked as having special roles by some interpreting system, instead of explicitly using actor nodes. Moreover, special relations are added to express inclusive and exclusive logical disjunction [4].

These and other executable CG mechanisms are reviewed in detail in [3]. Mineau's executable CG mechanism strikes a balance between simplicity and expressive power. Correspondence with Mineau at the time [3] was embarked upon confirmed that there had been no implementation of the process mechanism, "…only a theory which needs to be refined, implemented and applied."[2] That is exactly what the work described herein entails. A major motivation for any implementation of the process mechanism is to determine whether it works, and if so, in what ways it requires improvement.

---

[2] E-mail correspondence from Guy Mineau to David Benn, December 1st, 1999.

3 The pCG Language

We believe that 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 closely with the fundamental entities. Mineau provides the beginnings of such a language in [13], and that paper's stated long term goal of the development of a Conceptual Programming Environment (CPE) 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 primary motivation for pCG is to make available a concrete implementation of Guy Mineau's process mechanism [3], [13] for empirical investigation of its strengths and weaknesses. A secondary motivation is to provide an easily extensible general-purpose language with CG processing functionality which makes the primary entities of interest in the domain (eg CGs, actors, and processes) first class values, and permits object-based, functional, and declarative programming styles, which is consistent with Mineau's idea for a CPE.

The pCG distribution is available on the web,[3] and is available under the Gnu Public Licence. The complete pCG solution to the Sisyphus-I problem described in this paper is also provided in the distribution, along with many other example programs. We consider pCG to be an experimental language and believe that feedback from the CG community will be useful at this point in time. A non-trivial application seems to be an effective means by which to accomplish this.

3.1 Design

CG tools such as [8] have been developed to 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 all influenced pCG. The choice of a new language is a deliberate one, to clearly state that something different is being represented, and to avoid confusing similarities with existing languages. Given that major new constructs are required (eg actor type definition, process statement) which will take the form of new syntax, and that significant new types must be introduced (eg concept, graph), an existing language would require significant extension. CG languages and platforms other than Synergy are considered in [3]. One drawback of creating a new language is that no pre-existing utility source code in that language exists, but since pCG has an extensible type system, utility code written in Java can be used.

3.2 Key Features

The pCG language is interpreted for rapid development. It is *dynamically typed* since values not variables determine type. pCG is *lexically scoped* since functions and

---

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

processes introduce a new lexical scope when invoked. pCG can also 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 it is possible to easily add new types or modify existing ones using Java without change to the interpreter itself. All values in pCG are objects with associated *attributes* (eg length), *operations* (like methods in Java), and *operators* (eg +). For example, CG operations such as *join* and *project* are supported as operations on graph objects. Some attributes may appear on the left side of an assignment statement (eg `myConcept.designator=2`). 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 for run-time type identification. Objects in pCG are garbage collected when they can longer be reached.

The pCG language is *multi-paradigm*, since apart from its object-based nature, 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. Apart from the essential iteration and selection constructs found in any modern language, three control constructs are central to pCG: functions, actors, and processes. There is currently no provision for concurrent execution in pCG.

3.2.1 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; // example of setting an attribute
end

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

The actor defines a dataflow graph which contains a single 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 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 [17].

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: `[Num:*a  4][Num:*b 16]<sqr?a|?b>`.

A point of departure in pCG's implementation of actors compared to [17] 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. 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, for example:

```
n = 4;
mySqrGrStr = "[Num:*a " + n + "][Num:*b'*y']<sqr?a|?b>";
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, except that the process engine implicitly activates precondition graphs containing actors. The process definition of section 4 contains no actors, but [3] gives an implementation of Mineau's iterative factorial process [13] which does use actors in preconditions.

3.2.2 Processes

An abstract process is defined in [13] as: *process p(in g1, out g2,...) is { $r_i = <pre_i, post_i>$, $\forall_i \in [1, n]$ } .* 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 graphs (in contexts — see below) to be asserted or retracted.

A 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 otherwise 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. Only retraction of whole graphs from a KB, not partial graphs is possible in pCG.

Contexts with concept types PROPOSITION, ERASURE, and CONDITION are essentially used as a packaging mechanism for passing parameters to a process. PROPOSITION and ERASURE context types are also used in the body of post-conditions to distinguish between assertions and retractions. An alternative to the latter is

discussed in section 6. A precondition is an arbitrary graph expression. If preceded by a '~' character, the sense of the match for that graph is reversed.

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 corresponding graph acting as a trigger which (along with other asserted graphs from the caller's KB) 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 — contexts of type PROPOSITION or ERASURE — are appended to the post-condition list of the last rule. When this rule is reached, the specified graphs are asserted or retracted in the caller's KB.

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 (see section 4.3) 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.

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 [17] except that relation type subsumption is also permitted. A match may be partial. For example, a process rule precondition such as `[Person:*a'*name'][Role:*b'Researcher'](Chrc?a?b)` would be sufficient to retrieve the complete graph of Michael T. shown in section 4.2 if present in a KB, and if it were encountered first during search.

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 content of a process caller's KB is copied to the invoked process's KB. Look-up and, by default, assertion and retraction are confined to the KB which is top-most on the stack. 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.

4 A Sisyphus-I Solution in pCG

What follows is a detailed description of how the Sisyphus-I problem is represented and solved with a pCG program, the bulk of which consists of a process definition.

4.1 Assumptions and Interpretations

This section briefly documents assumptions made and interpretations of [9] for the purpose of problem representation and solution.

The criterion to be used for determining room centrality is unclear from [9]. The criterion used for the current solution is physical connection to room C5-118 (The Tower), implying that rooms C5-113 to C5-119 are central. An alternative interpretation is that any room which is *near* C5-119 could be considered to be central. For example, C5-120 could be argued to be central by virtue of physical proximity to C5-119. Another reasonable interpretation is given by [1] which suggests that C5-116, C5-117, and C5-119

are central. It is assumed that the black square on the room graphic in [9] corresponds to room C5-123.

In section 2.1.1 of [9], "true" and "yes" are both used to indicate boolean true values. The problem representation converts all occurrences of "true" to yes".

4.2 Representing the Problem

All the information required to solve the problem is contained in section 2.1.1 of the HTML found at the URL of [9]. A Perl script was applied to the HTML to generate simple CGs each of which represents some aspect of each YQT member. One such graph set is shown below in CGIF:[4]

```
[Person:*a'Michael T.'][Role:*b'Researcher'](Chrc?a?b)
[Person:*a'Michael T.'][Project:*b'BABYLON Product'](Member?a?b)
[Person:*a'Michael T.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Michael T.'][Hacker:*b'Yes'](Chrc?a?b)
[Person:*a'Michael T.'][Person:*b'Hans W.'](Coworker?a?b)
```

The following graphs for Katharina N. apply to the first problem statement, with almost identical graphs for Christian I. in the second.

```
[Person:*a'Katharina N.'][Role:*b'Researcher'](Chrc?a?b)
[Person:*a'Katharina N.'][Project:*b'MLT'](Member?a?b)
[Person:*a'Katharina N.'][Smoker:*b'Yes'](Chrc?a?b)
[Person:*a'Katharina N.'][Hacker:*b'Yes'](Chrc?a?b)
```

It is inefficient to process many small graphs, so the graphs for each person are joined into a single graph by pCG code. These multi-relation graphs are then asserted into the top-level knowledge base (KB). As an example, Michael T's graphs are joined into the single graph shown below in LF and CGIF, respectively:

```
[Person:'Michael T.'] –             [Person:*a'Michael T.']
   (Chrc) -> [Role: 'Researcher']   [Role:*b'Researcher']
   (Member) -> [Project: 'BABYLON   [Project:*c'BABYLON
                      Product']               Product']
   (Chrc) -> [Smoker: 'No']         [Smoker:*d'No']
   (Chrc) -> [Hacker: 'Yes']        [Hacker:*e'Yes']
   (Coworker) -> [Person: 'Hans W.'] [Person:*f'Hans W.']
                                         (Chrc?a?b)
                                         (Member?a?c)
                                         (Chrc?a?d)
                                         (Chrc?a?e)
                                         (Coworker?a?f)
```

The following pCG code joins and asserts the graphs into the KB:

```
yqt = file inFilePath;
lines = yqt.readall();
yqt.close();
g = "";
foreach line in lines do
```

---

[4] The Conceptual Graph Interchange Format is used in pCG examples throughout this paper.

```
    if line.length > 0 then
      if g is graph then
        g = g.join(line.toGraph()); // add to current graph
      end else
        g = line.toGraph(); // new person
      end
    end else
      assert g; // add current person's graph to KB
      g = ""; // prepare for next person
    end
end
```

Although most information can be automatically extracted, a few facts can not, so these are joined to existing graphs and reasserted, thus:

```
function addTo(addition)
  newConcepts = addition.concepts;
  foreach g in _KB.graphs do
    gConcepts = g.concepts; // are the head concepts identical?
    if newConcepts[1] == gConcepts[1] then
      retract g; assert g.joinAtHead(addition); return;
    end
  end
end

addTo(`[Person:*a'ThomasD.'][Head:*b'YQT'](Chrc?a?b)`);
addTo(`[Person:*a'Hans W.'][Head:*b'BABYLON Product'](Chrc?a?b)`);
addTo(`[Person:*a'Joachim I.'][Head:*b'Other'](Chrc?a?b)`);
addTo(`[Person:*a'Katharina N.'][Head:*b'Other'](Chrc?a?b)`);
```

The additional information above concerning Katharina N. ensures that her room allocation will have a higher priority than it would otherwise, which will become apparent in the discussion of process rules to come.

   Information about the available rooms is next asserted in the top-level KB with large central rooms being C5-117 and C5-119, small central rooms being C5-113..116, and large non-central rooms being C5-120..123.

```
rooms={"[LargeRoom:*b'C5-117'][Location:*c'Central']
          [Integer:*d 2](Chrc?b?c)(Vacancy?b?d)",
        ...
       "[LargeRoom:*b'C5-120'][Location:*c'NonCentral']
          [Integer:*d 2](Chrc?b?c)(Vacancy?b?d)",
        ...
       "[SingleRoom:*b'C5-116'][Location:*c'Central'](Chrc?b?c)"};
foreach room in rooms do
  assert room.toGraph();
end
```

The top-level KB now contains graphs representing information such as that there exists a large room whose location is central and whose vacancy count is 2, which corresponds to the first graph above. While the characteristic of largeness and the vacancy count may seem to be redundant information, the latter is used in the process of the next section to differentiate between full and partially tenanted large rooms.

4.3 The Rules

This section details a pCG process definition which yields a solution to the first and second statements of the Sisyphus-I problem. Action blocks are shown for the first rule, but omitted thereafter to save space. In each rule, the precondition action block prints a question that asks whether a particular kind of person needs to be allocated to a room. The post-condition action block calls functions which direct information to standard output and to a graphical room display regarding a room allocation if the rule of which it is a part fires.

The process definition that follows is punctuated with commentary regarding particular rules where deemed necessary. The process is dependent upon other user-defined functions. For the complete code listing see the pCG distribution.

The Sisyphus process is first named for invocation purposes, and in this case, no input or output parameters are passed to it, but [3] and [13] provide examples of their use. The first rule simply allocates any large central room to the head of department.

```
process Sisyphus()
  rule allocateHeadOfYQT
    pre
      action
        println "Need to allocate room for the head of YQT?";
      end
      `[Person:*a'*name'][Head:*b'YQT'](Chrc?a?b)`;
      `[LargeRoom:*a'*roomLabel'][Location:*b'Central']
         (Chrc?a?b)`;
    end
    post
      action
        label = getRoomLabel(); // uses _MATCHES array
        name = getPersonName(); // also uses _MATCHES array
        showAllocation("Head of YQT", name, label);
        plotName(name, label, 1, headColor);
      end
      `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel']
         (Occupant?a?b)]`; option export;
      mkErasureGraph(_MATCHES[1]); // erase person
      mkErasureGraph(_MATCHES[2]); // erase room
    end
  end // rule allocateHeadOfYQT
```

The PROPOSITION context in the post-condition block directs pCG to assert its descriptor graph into the KB. The caller's KB is targeted rather than the local KB of the process by the `export` directive. Any modifications to the process's local KB will not affect the caller's KB, except when "`option export`" is applied to a post-condition graph, or when the final rule fires. The former applies here with the result that the selected post-condition graphs are asserted in the top-level KB.

The _MATCHES list contains each complete graph that was matched by the rule's precondition graphs via projection onto the graphs in the current KB. The list is created by the pCG interpreter during rule invocation. The getter functions in the action block are invoked to iterate over another special list, `_KB.corefvars`, which stores name-value pairs of concept designator variable bindings (such as *name and *roomLabel) from the matching operations. The only reason for invoking `mkErasureGraph()` is to yield

an ERASURE context.[5] The fact that this is necessary at all suggests that the generic `post..end` syntax ought to be replaced by something more specific. See also section 6.

The next rule *must* come before `allocateFirstSecretary`, otherwise the latter will fire twice, yielding allocation into two different large rooms. The rule `allocateFirstSecretary` allocates one secretary to an empty large central room and decrements the vacancy count of that room by the retraction and assertion of room graphs. The `allocateSecondSecretary` rule looks for a large central room with one vacancy. Both secretaries are thus allocated to the same room, due to a particular rule sequencing.

```
rule allocateSecondSecretary
  pre
    `[Person:*a'*name'][Role:*b'Secretary'](Chrc?a?b)`;
    `[LargeRoom:*a'*roomLabel'][Location:*b'Central']
        [Integer:*c 1](Chrc?a?b)(Vacancy?a?c)`;
  end
  post
    `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel']
        (Occupant?a?b)]`; option export;
    mkErasureGraph(_MATCHES[1]); // erase person
    mkErasureGraph(_MATCHES[2]); // erase room
  end
end // allocateSecondSecretary

rule allocateFirstSecretary
  pre
    `[Person:*a'*name'][Role:*b'Secretary'](Chrc?a?b)`;
    `[LargeRoom:*a'*roomLabel'][Location:*b'Central']
        [Integer:*c 2](Chrc?a?b)(Vacancy?a?c)`;
  end
  post
    `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel']
        (Occupant?a?b)]`; option export;

    mkErasureGraph(_MATCHES[1]); // erase person

    `[ERASURE:[LargeRoom:*a'*roomLabel'][Location:*b'Central']
        [Integer:*c 2](Chrc?a?b)(Vacancy?a?c)]`;

    `[PROPOSITION:[LargeRoom:*a'*roomLabel'][Location:*b
        'Central'][Integer:*c 1](Chrc?a?b)(Vacancy?a?c)]`;
  end
end // allocateFirstSecretary
```

The third post-condition expression in `allocateFirstSecretary` could be replaced by `"mkErasureGraph(_MATCHES[2])"` in order to erase the corresponding room graph, but the current code displays a symmetry between erasure of the old room graph and assertion of the new room graph.

---

[5] Code such as `("[ERASURE:" + _MATCHES[1] + "]").toGraph()` is used.

The next two rules (`allocateManager` and `allocateAHead`) are similar to `allocateHeadOfYQT` except that different constraints are applied in the person and room preconditions.

Rules for allocating first and second researcher pairs are similar to those for secretary allocation shown above. In a similar manner, `allocateSecondResearcher` must come before `allocateFirstResearcher`, otherwise the latter will fire twice, yielding allocation into two different large rooms, making the final 3 pairs of allocations (see actions 8, 9, and 10 in [9]) impossible without a smoker/non-smoker conflict. This ordering is akin to a degenerate recursive step. The lesson here is that the more constrained rule should appear first. We allocate non-smoking researchers with this and the next rule, handling smokers later.

Once again, `allocateSecondSmoker` must precede `allocateFirstSmoker`, otherwise the latter will fire twice, yielding allocation into two different large rooms. Here we are trying to pair off smokers into large rooms. If we have an *odd* number of smokers or non-smokers, we may have no choice but to allocate smokers with non-smokers, or to allocate a smoker to a single room.

The final catch-all rule is designed to cope with the Second Problem Statement (section 2.2 of [9]). It simply says that any remaining researcher should be allocated to *any* remaining room. There should in fact be three rules: two to handle a large room in the manner above, and one to handle a single room (eg allocate{First, Second}RemainingResearcher, and allocateRemainingResearcher). It just so happens that we know from the problem statement that there is only one small room remaining by the time this rule applies, and that the process will terminate on the following cycle since no room or people graphs will be left in the local KB of the process.

```
rule allocateRemainingResearcher
    pre
      `[Person:*a'*name'][Role:*b'Researcher'](Chrc?a?b)`;
      `[Room:*a'*roomLabel'][Location:*b'*somewhere']
         (Chrc?a?b)`;
    end
    post
      `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel']
         (Occupant?a?b)]`; option export;
      mkErasureGraph(_MATCHES[1]); // erase person
      mkErasureGraph(_MATCHES[2]); // erase room
    end
  end // allocateRemainingResearcher
end // process Sisyphus
```

4.4 The Results

The pCG program described above is invoked via the following code: `"Sisyphus()"`. The result of this invocation is text on the standard output and a window with a room graphic, both of which are updated during the course of the program run.[6] A sample of the text output is shown for the first problem statement below.

```
Asserting YQT member graphs.
Asserting more information about certain YQT members.
```

---

[6] This takes approximately 10 seconds on a 550 MHz Pentium III Linux workstation.

```
Need to allocate room for the head of YQT?
--> Head of YQT 'Thomas D.' allocated to C5-117
Need to allocate room for the head of YQT?
Need to allocate room for second secretary?
Need to allocate room for first secretary?
--> First secretary 'Monika X.' allocated to C5-119
Need to allocate room for the head of YQT?
Need to allocate room for second secretary?
Second secretary 'Ulrike U.' allocated to C5-119
...
Need to allocate room for the head of YQT?
Need to allocate room for second secretary?
Need to allocate room for first secretary?
Need to allocate room for the manager?
Need to allocate room for a head?
Need to allocate room for a second researcher?
Need to allocate room for a first researcher?
--> First researcher 'Michael T.' allocated to C5-122
Need to allocate room for the head of YQT?
Need to allocate room for second secretary?
Need to allocate room for first secretary?
Need to allocate room for the manager?
Need to allocate room for a head?
Need to allocate room for a second researcher?
Second researcher 'Harry C.' allocated to C5-122
...
```

The text output reflects the pattern of rule firings. The final several lines of code iterate over the top-level KB once the process has terminated, sending to standard output the final room allocation graphs. This code and the output for the first problem statement is shown below, in that order.

```
println "Room allocation graphs:";
filter = `[Person:*a'*x'][Room:*b'*y'](Occupant?a?b)`;
foreach g in _KB.graphs do
  h = g.project(filter);
  if not (h is undefined) then println " " + h; end
end

[Person:*a"Thomas D."][Room:*b"C5-117"](Occupant?a?b)
[Person:*a"Monika X."][Room:*b"C5-119"](Occupant?a?b)
[Person:*a"Ulrike U."][Room:*b"C5-119"](Occupant?a?b)
[Person:*a"Eva I."][Room:*b"C5-113"](Occupant?a?b)
[Person:*a"Hans W."][Room:*b"C5-114"](Occupant?a?b)
[Person:*a"Joachim I."][Room:*b"C5-115"](Occupant?a?b)
[Person:*a"Katharina N."][Room:*b"C5-116"](Occupant?a?b)
[Person:*a"Werner L."][Room:*b"C5-120"](Occupant?a?b)
[Person:*a"Jurgen L."][Room:*b"C5-120"](Occupant?a?b)
[Person:*a"Marc M."][Room:*b"C5-121"](Occupant?a?b)
[Person:*a"Angi W."][Room:*b"C5-121"](Occupant?a?b)
[Person:*a"Michael T."][Room:*b"C5-122"](Occupant?a?b)
[Person:*a"Harry C."][Room:*b"C5-122"](Occupant?a?b)
[Person:*a"Andy L."][Room:*b"C5-123"](Occupant?a?b)
[Person:*a"Uwe T."][Room:*b"C5-123"](Occupant?a?b)
```

5 Discussion

The key difference in the second problem statement is that Christian is allocated last after all higher priority constraints have been satisfied, whereas in the first problem statement Katharina is allocated much earlier, due to her higher status. This is reflected in the static rule orderings of the process. [9] suggests the general approach that should be followed, such as allocating the head of the YQT department, followed by secretaries, then heads of projects, then smokers, and other researchers. Examples of who should be allocated to which rooms are also given. The current solution departs from the one suggested in that smokers are allocated after all non-smoking researchers, which still produces the desired outcome. The rationale is that the non-smoking mandate is a high-priority constraint.

In a number of cases, two rules are required for correct allocation to large rooms that can hold two occupants. The ordering of these rules is important, as others have found (eg [1], [5]). For example, the `allocateSecondResearcher` rule looks for a large room in which there is already an occupant. The immediately following rule is `allocateFirstResearcher` which will fire first in the absence of such half-tenanted rooms, assigning a non-smoking researcher and decrementing the room's vacancy count. Such rule pairs have already been noted. An earlier version of the program used an actor to obtain a decremented vacancy count in a rule's precondition block, which was utilised in a post-condition graph of that rule. Since the vacancy count is first 2 then 1, actors are overkill in this context, hence the current solution which simply retracts a room graph with a vacancy count of 2 and asserts an otherwise identical room graph with a vacancy count of 1. This is also faster since actor execution involves additional execution steps.

The final rule in the Sisyphus process (`allocateRemainingResearcher`) uses type subsumption on the `Room` concept. Concept and relation type hierarchies are declared before the process definition, but only two actually matter:

```
concept Room > SingleRoom;
concept Room > LargeRoom;
```

This permits the final rule to ask for any room, rather than a single or large room. These types were inspired by [15]. An earlier version of the pCG program instead had a size relation in each room graph. While using a size concept such as `[Size: *dontCareHowBig]` and an associated relation in precondition graphs works, having one less node in each room graph permits more efficient precondition matching, and a more elegant solution based upon type subsumption rather than referent specialisation.

As already mentioned, the program solves both problem statements, however *no rule* imposes a constraint which ensures that people on the same project are not allocated to the same room. The process works without such a constraint given the supplied ordering of YQT members in [9], but if the order of graphs originally asserted in the KB were to change, this constraint could be violated. One approach is for graphs representing two different individuals to be compared in terms of the coworker or project relations, perhaps via a `NotSame` actor. However, identical precondition graphs in a single pCG rule will yield identical matches, for example:

```
[Person: *a '*name1'][Project: *b '*proj1'](Member?a?b)
[Person: *a '*name2'][Project: *b '*proj2'](Member?a?b)
```

This suggests that process rules ought to employ backtracking, essentially treating the conjunction of precondition graphs as a goal, to ensure that different `*proj1` and `*proj2` combinations are tried. It is not currently possible in a single precondition block to find all combinations of graphs in the KB which match the graphs in that precondition. One can only do this in pCG via explicit iteration and projection over a KB's graphs. [14] and [15] also point to a possible solution in terms of the application of constraints. This problem requires further investigation.

Information that is not used in the current solution is the project, hacker, and coworker concepts and their associated relations. An interesting question is whether the current solution could be made more robust by making use of some or all of these relations. However, coworker information sometimes does not correspond to project membership as obtained from the information provided in [9].

6 Conclusions and Future Work

We believe that pCG shows promise as a general CG language, and in particular as a means by which to explore Mineau's process mechanism, but improvements are of course possible. An operational solution to the Sisyphus-I room allocation problem was presented. The pCG language was used as the basis for this, and while successful in solving the problem, the attempt revealed shortcomings in both the solution and the language. The requirement that two researchers on the same project should not share an office might be solved by an application of constraints (eg [14], [15]), or by treating the conjunction of preconditions in a rule as a goal and employing backtracking if necessary. If such functionality were implemented in pCG, the language's problem solving power would be increased.

The generic `post..end` syntax in rules ought to be replaced by `assert..end` and `retract..end`, each of which would contain a sequence of simple graphs instead of ERASURE and PROPOSITION contexts. The content of `_KB.corefvars` could be made available to process action block code on the local stack frame of the executing code.

Where complex pre or post-condition graphs are required, a CG graph editor capable of generating CGIF can be used, and later read from a file for use in a process definition. Perhaps a graphical front end could be used to draw and generate complete process definitions.

The pCG language has the advantage of simplicity, and being interpreted, "compile" is removed from the edit-compile-run cycle, speeding up development of pCG programs. It is also extensible by the addition of Java code, as has been noted. However, it is not a refinement of a widely used existing language.[7] Given that the implementation of pCG cleanly separates the parser (around 500 lines of code) from the core interpreter, the source language for pCG could just as easily be Lisp, if so desired. Feedback from the CG community would be of benefit with respect to the usability of pCG.

Other future work relating specifically to pCG is detailed in [3]. Such work includes the reconciliation of the graph assertion and retraction mechanism with the rules of inference [17], [18], improvements to graph matching efficiency, implementation of the conformity relation, and the addition of further attributes and operations to pCG objects to make the intrinsic types more useful.

---

[7] For example, Prolog+CG (http://www.insea.ac.ma/CGTools/CGTools.htm).

References

1. Baget, J.F., Genest, D., and Mugnier, M.L., A Pure Graph-Based Solution to the SCG-1 Initiative. In *Proceedings of the 7ᵗʰ International Conference on Conceptual Structures*, Blacksburg, VA, USA, pp. 355–376, July 1999.
2. Bos, C., Botella, B., and Vanheeghe, P., Modelling and Simulating Human Behaviours with Conceptual Graphs. In *Proceedings of the 5ᵗʰ International Conference on Conceptual Structures*, Seattle, Washington, USA, pp. 275–289, August 1997.
3. Benn, D., Implementing Conceptual Graph Processes. Master of Computer and Information Science Minor Thesis, April 2001, University of South Australia. URL: http://www.adelaide.net.au/~dbenn/Masters/
4. Cyre, W., Executing Conceptual Graphs, In *Proceedings of the 6ᵗʰ International Conference on Conceptual Structures*, Montpellier, France, pp. 51–64, August 1998.
5. Damianova, S. and Toutanova, K., Using Conceptual Graphs to Solve a Resource Allocation Task. In *Proceedings of the 7ᵗʰ International Conference on Conceptual Structures*, Blacksburg, VA, USA, pp. 355–376, July 1999.
6. Delugach, H., Dynamic Assertion and Retraction of Conceptual Graphs. In *Proceedings of the 6ᵗʰ Annual Workshop on Conceptual Graphs*, Eileen C. Way, editor, SUNY Binghampton, New York, pp. 15–26, July 1991.
7. Ellis, G., Object-Oriented Conceptual Graphs. In *Proceedings of the 3ʳᵈ International Conference on Conceptual Structures*, Santa Cruz, CA, USA, pp. 144–157, August 1995.
8. 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.
9. Linster, M., Documentation for the Sisyphus-I Room Allocation Task, [Accessed Online: November 2000], URL: http://ksi.cpsc.ucalgary.ca/KAW/Sisyphus/Sisyphus1/
10. Lukose, D., Executable Conceptual Structures. In *Proceedings of the 1ˢᵗ International Conference on Conceptual Structures*, Quebec City, Canada, pp. 223–237, August 1993.
11. Mann, G.A., A Rational Goal-Seeking Agent using Conceptual Graphs. In *Proceedings of the 2ⁿᵈ International Conference on Conceptual Structures*, College Park, Maryland, USA, pp. 113–126, August 1994.
12. Martin, P. and Eklund, P., WebKB and the Sisyphus-I Problem. In *Proceedings of the 7ᵗʰ International Conference on Conceptual Structures*, Blacksburg, VA, USA, July 1999.
13. 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.
14. 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.
15. Mineau, G., Constraints and Goals under the Conceptual Graph Formalism: One Way to Solve the SCG-1 Problem. In *Proceedings of the 7ᵗʰ International Conference on Conceptual Structures*, Blacksburg, VA, USA, pp. 334–354, July 1999.
16. Nenkova, A. and Angelova, G., User Modelling as an Application of Actors. In *Proceedings of the 7ᵗʰ International Conference on Conceptual Structures*, Blacksburg, VA, USA, pp. 83–89, July 1999.
17. Sowa, J.F., Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley, 1984.
18. Sowa, J.F., Knowledge Representation: Logical, Philosophical, and Computational Foundations, Brooks/Cole, 2000.
19. Thanitsukkarn, T. and Finkelstein, A., Multiperspective Analysis of the Sisyphus-I Room Allocation Task Modeled in a CG Meta-Representation Language. In *Proceedings of the 7ᵗʰ International Conference on Conceptual Structures*, Blacksburg, VA, USA, July 1999.