Table of Contents

List of Figures

List of Tables

Source Code Listings

Abbreviations and Symbols

Abbreviation	Meaning
ANSI	American National Standards Institute.
API	Application Programming Interface.
AST	Abstract Syntax Tree; an intermediate form generated by a parser.
CFR	Canonical Formation Rules; fundamental CG operations.
CG	Conceptual Graph; basic entities in Sowa's knowledge representation formalism.
CGIF	Conceptual Graph Interchange Format.
CPE	Conceptual Programming Environment; Mineau's proposed imperative/logic language.
CST	Sowa's Conceptual Structures Theory.
DF	Display Form of CGs.
EBNF	Extended Backus-Naur Form; a formal grammar notation.
GUI	Graphical User Interface.
ICCS	International Conference on Conceptual Structures, an annual conference.
IDE	Integrated Development Environment.
JDK	Java ™ Development Kit.
JRE	Java ™ Run-time Environment.
KB	Knowledge Base.
KIF	Knowledge Interchange Format.
LF	Linear Form of CGs.
OO	Object-Oriented; programming and design paradigms.
pCG	Conceptual Graph Processes language and interpreter.
STD	State Transition Diagram.

Symbol	Meaning
∀	For all, e.g. ∀x:Cat purr(x) means that all cats purr.
∃	There exists, e.g. ∃x:Cat name(x, Foo) means that there exists a cat named Foo.
δ	Denotation of a type. For example, δMyFamily = {David, Karen, Nicholas}.
⊤	Top or the Universal type. The top of a concept or relation type hierarchy.
⊥	Bottom or the Absurd type. The bottom of a concept or relation type hierarchy.
≥, ≤, >, <, =	Operators on concept and relation types, or numbers. Explained in context. For example, `t ≥ t'` says that `t` is a supertype of `t'` or the same as `t'`.
$\geq_g$	Graph subsumption. For `g` $\geq_g$ `h` to be true, the graph g must be a generalisation of another graph.
ℜ	The set of real numbers.
∈	Set membership operator.
{…}	A set, or a list literal in pCG.
Ø	The empty set.
∧	Logical conjunction (and).
∨	Logical disjunction (or).
⊃	Logical implication.
|	Choice, e.g. `(a | b | c)`.
[…]	Optional component, e.g. `[foo]bar` could be `foobar` or just `foo`.
(…)*	Kleene Closure: zero or more, e.g. `(a)*` could be nothing, `a`, `aa`, `aaa`, and so on.
⇒	Returns. For example, a pCG operation returns some type `t`.
™	Trade Mark.
©	Copyright.

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. The current work surveys the literature regarding executable and dynamic conceptual graph formalisms and tools, then presents a general-purpose programming language, *pCG*, which has been developed by the author to embody Mineau's process formalism. Experiments carried out with pCG are detailed, and future directions are suggested.

Results

The development of pCG, a general-purpose programming language which embodies a concrete form of Mineau's (1998) process formalism via a process definition and invocation mechanism. The language provides capabilities for working with conceptual graphs, processes, actors, and functions as first class entities. It is possible to extend pCG's type system in arbitrary ways without change to the core interpreter.
Progress toward Mineau's (1998) suggested Conceptual Programming Environment, in the form of pCG.
The refinement of Mineau's process formalism in the course of developing pCG and applying it to problems.
The application of pCG to examples other than Mineau's (1998) iterative factorial example process, e.g. the Sisyphus-I room allocation problem [Linster 1999].
The uniform representation of actor and process invocation via the standard actor node found in the conceptual graph notation of the proposed ANSI CG Standard [Sowa 1999], instead of the non-standard syntax of [Mineau 1998] and [Delugach 1991].
The development of recursive anonymous actors in pCG, obviating the need for an actor type definition in the special case of a recursive actor.

Declaration

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

Acknowledgments

The pCG language is a concrete realisation of the ideas in Guy Mineau's 1998 ICCS paper. I am grateful to my supervisor, Dan Corbett, who has been enthusiastic about my work from the start, and has always made the time to meet with me. I engaged in discussions with some members of the Intelligent Systems Group[1] regarding the process formalism, pCG, and Conceptual Structures Theory in general. I appreciate the time Dan, Csaba, and Conn took to participate in those face-to-face and e-mail discussions.

I am fortunate to work for a company that permits its employees to take regular time out for postgraduate studies. For more than a year, the Motorola Australia Software Centre permitted me to commit each Wednesday to working on this thesis.

Finally, my wife Karen has tolerated the "long strange trip" that has seen me move from nurse to software engineer and would-be computer scientist for what seems like a very long time now. For that and many other things I am indebted to her, especially all the times she asked whether I needed another cup of coffee! She also helped in proof reading. More recently, when I should have been helping Karen look after our son Nicholas, I was usually to be found tending to Microsoft Word's foibles or writing Java code in XEmacs.

[1] This is a research group of the School of Computer and Information Science of the University of South Australia.

1 Background

It takes considerable knowledge just to realise the extent of your own ignorance.

Thomas Sowell

1.1 Introduction

Sowa's Conceptual Structures Theory (CST) provides a means to represent knowledge as concepts and relationships between concepts in structures known as Conceptual Graphs (CGs) [Sowa 1984]. CGs are derived from Charles Sanders Peirce's 19th Century existential graphs with influences from linguistics, and AI, such as semantic nets. [Sowa 2000] [Shinghal 1992]

The original motivation for CGs according to [Sowa 2000] was for the representation of natural language semantics. Sowa contends that CGs "…can help form a bridge between computer languages and the natural languages that everyone reads, writes, and speaks." [Sowa 2000]. Indeed, some authors have developed tools for converting CGs to natural language, for example [Delugach 1999].

A simple example of a CG is:

```
[Baby] -> (On) -> [Mat]
```

which expresses the notion that there is a baby on a mat of some kind, or more specifically:

```
[Baby: Nicholas] -> (On) -> [ChangeMat: Blue]
```

which says that a baby, Nicholas, is on a blue change mat. The square-bracketed text represents *concepts*, while the parenthesised text represents a *conceptual relation* over two concepts.

More formally, a CG is a bipartite graph meaning that concepts are connected to conceptual relations (or just *relations*) by directional arcs. Concepts can never be directly connected, nor can relations, either of which would be meaningless. Arcs may be numbered to make argument ordering explicit. The terms CG and graph will be used interchangeably in this thesis. Special cases of CGs are the blank graph, containing no concepts or relations, and a singleton graph consisting only of one concept. The blank graph has special meaning in Sowa's Rules of Inference [Sowa 1999] and this will be briefly considered below. Conceptual relations take concepts as arguments and may be monadic, dyadic, triadic, or more generally, n-adic. [Sowa 2000]

1.2 Concepts

A concept consists of a *type* and a *referent*. A referent consists of a *quantifier* and a *designator*, either of which may be absent. A concept's quantifier defaults to the existential quantifier: ∃. The universal quantifier: ∀ denotes all, e.g.

```
[Baby: ∀] -> (Attr) -> [Small]
```

declares that all babies have the attribute [2] of being small. This set could be reduced to state that particular babies are small:

```
[Baby: {Nicholas, Joshua}] -> (Attr) -> [Small]
```

or simply that there is some set of small babies:

[2] Conceptual relations used in the example CGs of this chapter are consistent with the sample ontology in Appendix B of [Sowa 2000]. An ontology is a set of things which are considered to exist in a domain. These things (types) are ordered in a particular way so as to indicate their relationship to one another.

```
[Baby: {*}] -> (Attr) -> [Small]
```

or 5 babies:

```
[Baby: {*}@5] -> (Attr) -> [Small]
```

In the concept `[Baby]`, the type is `Baby` and the referent is blank. The concept: `[Baby: Nicholas]` has a string *literal* designator: "Nicholas". In both cases, the quantifier defaults to ∃ indicating that there is some baby. In the case of `[Baby: Nicholas]`, all that can be said is that there exists some baby called Nicholas, but not some specific baby Nicholas. The latter requires a *locator* designator such as `#Nicholas` which specifies uniqueness. In practice, one might need to say `[Baby: #NicholasBenn]`, since there are many babies with Nicholas as their first name. There may well be more than one Nicholas Benn, in which case a designator such as `#8754926` may be required to uniquely identify the baby in question in say, a database. This could be used in conjunction with a relation, e.g.

```
[Baby: #8754926] -> (Chrc) -> [Name: 'Nicholas Benn']
```

which declares that there exists a baby who can be uniquely identified by the number 8754926, and that this baby has the characteristic of having the name shown. The third kind of designator is a *descriptor* which is a CG that provides an arbitrarily complex description. Descriptors will be considered along with contexts below.

1.3 Notations

The examples so far have been shown in the Linear Form (LF) notation which requires no special display capability, except for an appropriate font for quantifiers. Indeed, even this is not required by the ANSI CG standard [Sowa 1999] since `@forall` can be used instead of ∀. A CG notation intended primarily for machine translation and processing is the Conceptual Graph Interchange Format (CGIF) [Sowa 1999]. The first example CG above written in CGIF is:

```
[Baby: *a] [Mat: *b] (On?a?b)
```

A more complex example graph shown by Figure 1-1 illustrates the advantages of the CG Display Form (DF):

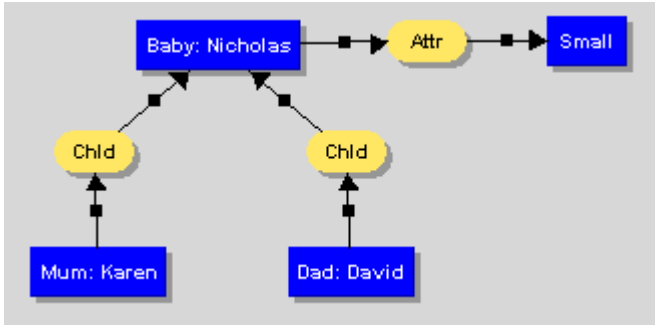

Figure 1-1 An example of CG Display Form. [3]

The graph of Figure 1-1 says there is a small baby with the specified name who is a child of a Mum and Dad each of whom have the names shown. Compared to the LF representation of the same graph, Figure 1-1 is more readable:

[3] All Display Form CGs in this thesis have been drawn using Harry Delugach's excellent CharGer software [Delugach 1999].

```
[Small] <- (Attr) <- [Baby: Nicholas] -
                          <- (Chld) <- [Mum: Karen]
                          <- (Chld) <- [Dad: David]
```

1.4 Signatures and Types

A conceptual relation has a signature. For example, the `Chld` (child) relation in Figure 1-1 has as its signature the tuple: `<HumanBeing, HumanBeing>`. What this means is that each concept which is an argument to this relation must be a subtype (which includes the type itself) of `HumanBeing`. A suitable partial concept type hierarchy for a world consisting of babies, mothers, fathers, and so on, is shown in Figure 1-2:

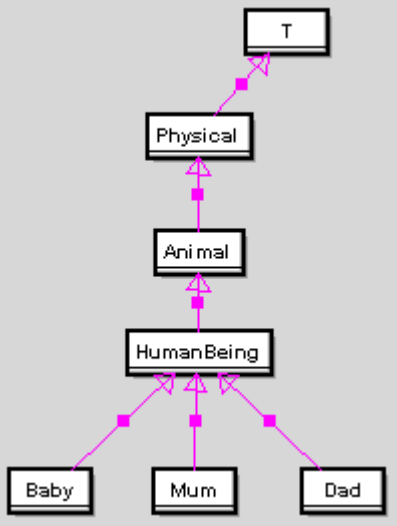

Figure 1-2 Partial concept type hierarchy

This can also be written in a linear format as:

```
T > Physical > Animal > HumanBeing
HumanBeing > Baby, Mum, Dad
```

`t > t'` indicates that `t` is a proper supertype of `t'`, i.e. `t` is more general than `t'`. `t ≥ t'` means that `t` is a supertype of `t'` which also permits `t = t'`, while `t < t'` indicates that `t` is a proper subtype of `t'`.

The special type ⊤ is the `Universal` type or `Entity` from which all other types descend. Although not shown here, the `Absurd` type (⊥) is the default proper subtype of any type with no explicit proper subtype [Sowa 2000]. For example, the proper subtype of `Dad` is ⊥, the proper supertype of `Dad` is `HumanBeing`, the proper supertype of `Physical` is ⊤, and the proper subtypes of `HumanBeing` are `Baby`, `Mum`, and `Dad`. Just as a type may have more than one proper subtype, a type may also have more than one proper supertype. For example, the concepts `HumanBeing` and `CanonBall` would be reasonable supertypes for the concept `HumanCanonBall`. If the following types were added to the hierarchy:

```
Physical > CelestialObject > Asteroid
```

We could ask a question such as: what is the common subtype of `Asteroid` and `Baby`? The answer would be ⊥.

If the following additions were made to the type hierarchy:

```
Mum > Parent
Dad > Parent
```

the common subtype of Mum and Dad would be Parent [4]. There are separate type hierarchies for relations and concepts, both of which are important for certain CG operations. For a detailed discussion of one such operation (projection), see 'The Implementation of pCG'. In essence, what such an operation permits is the specialisation of a graph, e.g.

```
[HumanBeing] -> (Agnt) -> [Action]
```

to:

```
[Baby: Nicholas] -> (Agnt) -> [Eat: 'Sweet Potato']
```

where the first is a template or filter graph, and the second is a graph in a KB matching the first, to yield: some baby Nicholas is the agent of an eating of Sweet Potato. This would require that:

```
Action > Eat
```

appears somewhere in the type hierarchy of Figure 1-2. Note that there is no relation specialisation here, only concept specialisation by type and referent restriction, both canonical operations. See also the 'Canonical Formation Rules and Rules of Inference' section below.

1.5 Actors

A special kind of conceptual relation called an actor will be considered in the next chapter. Regarding actors, Sowa says that they "…may have side effects that are not represented in the abstract syntax; formally however, actors are treated like other conceptual relations." [Sowa 2000]

1.6 Lambda Expressions

A formalism known as a *lambda expression* [5] can enhance the power of types [Sowa 2000]. A simple example is:

```
MaleBaby = [Baby: λ] -> (Chrc) -> [Gender: Male]
```

This can also be written as:

```
type MaleBaby(*x) is [Baby: ?x] -> (Chrc) -> [Gender: Male]
```

One can then have a graph such as:

```
[MaleBaby: Nicholas] -> (Attr) -> [Small]
```

which expands to:

```
[ [Baby: λ] -> (Chrc) -> [Gender: Male]: Nicholas ] -> (Attr) –
                                                    <- [Small]
```

Here, the lambda expression can be thought of as being applied to the designator `Nicholas`. Such types are implicitly added to the concept type hierarchy, for example we could have:

```
Baby > MaleBaby
```

`MaleBaby` is now a proper subtype of `Baby` by virtue of specialising the latter. Relation types can also be constructed using lambda expressions.

[4] Given that the notion of parent is more general than the other two, it would actually make more sense to have: `HumanBeing > Parent > Mum, Dad`.

[5] As we shall see in the next chapter and elsewhere, lambda expressions and actors are not unrelated.

1.7 Translation to Predicate Calculus

A CG can be translated to typed predicate calculus using the formula operator φ [Sowa 2000]. Conceptual relations become predicates, arcs translate to arguments, and concepts to typed variables. For example, here is a graph first in LF, then in CGIF, followed by its translation to predicate calculus:

```
[Baby: Nicholas] -> (On) -> [Mat]

[Baby: *x Nicholas] [Mat: *y] (On?x?y)

(∃x:mat)(baby(Nicholas) ∧ on(Nicholas, x)
```

From this example, it can be seen that existentially quantified concepts become existentially quantified typed variables, while concepts with non-blank referents become monadic predicates. This formula says that there exists a mat x, a baby Nicholas, and Nicholas is on x. Other formalisms of equal expressive power can be the target of such translations, e.g, the Knowledge Interchange Format (KIF). [Sowa 2000]

1.8 Contexts and Coreference

A context is a concept whose designator is a non-blank CG, i.e. a descriptor. Figure 1-3 is an example of a CG with a context:

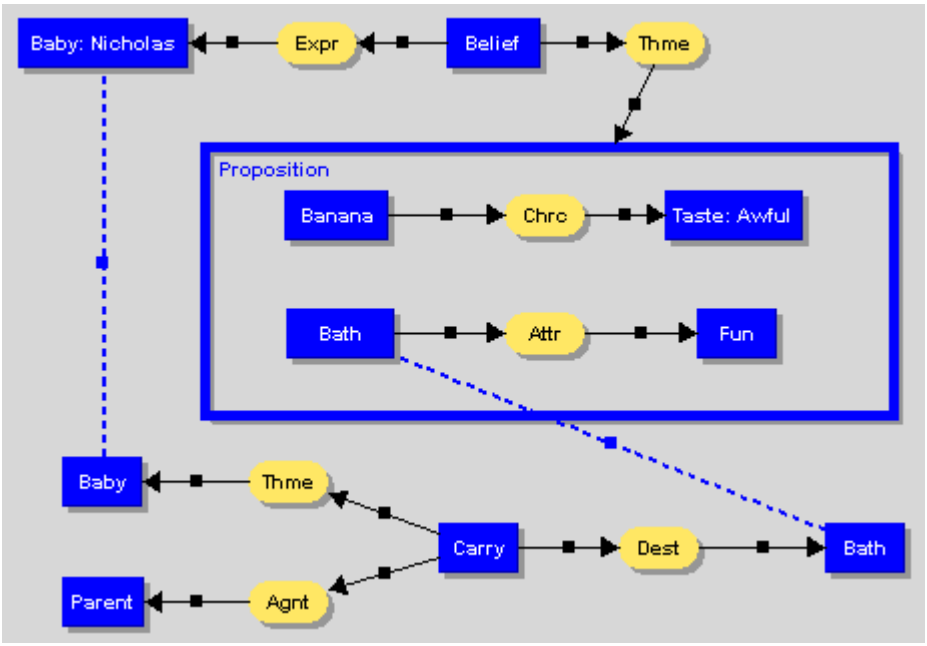

Figure 1-3 Baby Nicholas's beliefs

The top part of Figure 1-3 says that some baby, Nicholas, is the experiencer of a belief, the theme of which is two propositions: that some banana tastes awful, and some bath is fun. The important thing to notice here is that the two beliefs are bounded by a context box that is a concept of type `Proposition`, whose referent is a conjunction of two graphs. Of course, it is very unlikely to be the case that all babies hold these particular beliefs, and it is possible to create a context for an arbitrary baby's beliefs. 'See also 'Appendix A — Contexts'.

The dashed lines between the `Bath` and `Baby` concepts in Figure 1-3 is called a *coreference link* [Sowa 2000] or *line of identity* [Esch 1994], and the paired concepts are considered to be coreferent, which means that they refer to the same concept instance or individual. In this example, what is meant is that the bath that is believed by the baby to be fun, is the *same bath* to which the parent is carrying him, and that the baby with the specified beliefs is the same baby who is on the way to the bath.

1.9 Knowledge Bases

A Knowledge Base (KB) consists of a concept type hierarchy, a relation type hierarchy, a set of individuals conforming to particular concept types (e.g. `Nicholas` conforms to the type `Baby`), and a top-level context whose type is `Assertion`. In this context exists some set of graphs [Sowa 2000]. In the current work, the notion of a KB is slightly different: the graphs of the KB are not contained within an outermost context, but simply in a set. See 'The pCG Language' regarding the details of knowledge bases in pCG.

1.10 Canonical Formation Rules and Rules of Inference

CST's Canonical Formation Rules (CFRs) permit the modification of a KB of CGs via such operations as copy, join and simplify [Sowa 1984] [Sowa 1999]. Specific rules will be discussed in the chapters documenting the pCG Language. The CFRs ensure that only syntactically correct graphs will be produced, given a well-formed graph as a starting point. The rules are a CG-specific basis for the Rules of Inference.

When given a true graph the Rules of Inference will yield another true graph. There are three kinds of Rules of Inference: those which generalise a graph, those which specialise a graph, and those which produce an equivalent graph. One example is the rule known as erasure in which any graph may be replaced by a generalisation of itself so long as this takes place in a context which is not negated. In a negated context, a graph may be replaced by a specialisation of itself, an operation known as insertion. The names erasure and insertion derive from the fact that the blank graph is considered to be the generalisation of all graphs, such that a graph can be replaced by the blank graph (erased) or a graph can replace the blank (be inserted). A graph may not be erased from a negated context since the blank graph cannot be false[6] [Sowa 1984] [Sowa 1999] [Sowa 2000]. A full discussion of these and other rules of inference is beyond the scope of this thesis. Aspects of the CG Rules of Inference will however be discussed in the chapters documenting the pCG Language. See also 'Future Work'.

However one example which is an outcome of the rules is of use as a CG reasoning mechanism:

```
¬ [p ¬ [q]]
```

This *double negation* is a means by which logical implication may be carried out via CGs and negated contexts. Inside the nested contexts are two graphs: `p` and `q`, although there could be more than one in conjunction, in either or both contexts. If `p` is a graph representing some true fact, `q` is implied. This CG is equivalent to the following propositional logic formula:

```
~(p ∧ ~q)
```

which can be transformed via de Morgan's Laws [Shinghal 1992] to:

```
~p ∨ q
```

The truth table for this formula (Table 1-1) is equivalent to logical implication, $p \supset q$:

p	**q**	**~p**	**~p ∨ q**
T	T	F	T
F	T	T	T
T	F	F	F
F	F	T	T

Table 1-1 Truth Table for Logical Implication

[Sowa 1984] contains a proof that double negation yields implication, in terms of the Rules of Inference. The implication CG form is a common enough idiom that the CG standard [Sowa 1999] has reserved a special nested context for it:

[6] This is taken to be an axiom. [Sowa 2000] p 300.

```
[If: p [Then: q]]
```

There are however other ways to achieve implication, which will be discussed in 'The pCG Language'.

1.11 Dynamic Knowledge Base Update

CGs represent declarative information. By themselves they are insufficient for carrying out computation or simulating events and processes. While actors may suffice for some forms of computation, the simulation of processes requires some kind of truth maintenance engine, permitting assertion and retraction of graphs over time, which CST does not cater for [Lukose & Mineau 1998]. The next chapter reviews attempts to do this by the CG community.

This thesis is concerned with the refinement, implementation, and application of one such attempt, that of Guy Mineau's state-transition based process formalism, which permits arbitrarily complex knowledge-based precondition matching and subsequent dynamic KB updates [Mineau 1998]. Along the way, the broader topic of executable CGs will also be considered.

The remainder of this document consists of the following sections: literature review, objectives, description of developed software, experiments using the latter, and future work.

2 Literature Review

2.1 Introduction to Dynamic Conceptual Graphs

[Mineau 1998] proposed the notion of *processes* to overcome the fact that CST does not cater for the dynamic retraction and assertion of graphs in a CG-based KB. Mineau's processes are one kind of executable conceptual graph formalism. Other work has been carried out in this area by the CG community, and this chapter reviews that work in an effort to show how processes fit in.

Lukose and Mineau (1998) point out that knowledge representation formalisms such as Frames, Conceptual Dependency Graphs, and CGs represent declarative information and are by themselves "…insufficient for doing computation, solving problems, and simulating events and processes in such a way as to induce change onto the description of a system. Some form of truth maintenance inference engine is required for this purpose." [Lukose & Mineau 1998]. They cite PLASMA and Petri nets as more traditional graph notations for representing procedural information. PLASMA for example employs a network of actors each of which can be thought of as some kind of process that responds to a message, performs a service, and generates a message which is passed to another actor. Of course, this very broad description could correspond just as easily to an object in SmallTalk or Java as a computational entity in PLASMA or a CG system. [Lukose & Mineau 1998] restate four common themes of "graphs for doing computation" — broadly referred to as actors — from [Sowa 1984]. Such a graph:

1. Is a network of active elements.
2. Passes messages between actors or actor components.
3. Supports modularity, permitting an actor component to be replaced by another with no effect upon function. Assuming no side effects between actor components, this means that a component's output is dictated solely by its inputs. Such referential transparency [Louden 1993] also makes concurrency possible.
4. Provides definitional mechanisms permitting actors to be defined in terms of primitives (e.g. procedural functions) or other actors.

This list more closely corresponds to formalisms such as Sowa's actors [Sowa 1984] than to Mineau's processes, as this chapter will illustrate. Nevertheless, it provides a useful starting point for exploration. In the following sections, actors and other executable CG systems will be considered. Actors, demons, and processes will be given more emphasis given that the first two are ancestors of Mineau's processes.

Before continuing, a number of authors use the term "process" either to describe what their formalism is simulating, or in Mineau's case, as the name of a formalism. It is important therefore to explore briefly just what is meant by "process".

Sowa (2000) comments that Peirce distinguished between processes and the entities taking part in those processes and that Whitehead [7] made them primary in his ontology. 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 [Sowa 2000], which is the kind of process that is under consideration in this thesis.

[Sowa 2000] distinguishes between a procedure, a process, and a history: In Sowa's words:

- A *process*, is an evolving sequence of states and events, in which one of the states or events is marked *current* at a context-dependent time called #now.
- A *procedure* is a pattern or *script* that determines the types of states and events that may occur in an entire family of processes. Each process in the family is called an *activation* of the procedure.

[7] Co-author of *Principia Mathematica* (1913) with Bertrand Russell. [Shinghal 1992]

- A *history* is a record of the sequence of states and events that existed in the evolution of some process. Each state and event of a history may be marked with a *time stamp* that records its point in time or duration.

So, a *procedure* is analogous to its imperative programming counterpart class, a *process* is an invoked procedure, and a *history* is the log of effects of a process.

[Sowa 2000] discusses the means by which time can be represented and reasoned about in predicate logic and CGs, and such temporal logic will be briefly considered when comparing Mineau's processes and Delugach's demons.

[Sowa 2000] remarks that State Transition Diagrams (STDs) are a common way of representing discrete processes. The notion of state transition appears in a number of the formalisms in this review. Only one such formalism — Petri nets — is not a CG formalism, but is considered here due to its generality and to avoid being completely CG-centric.

In what follows, the immediate predecessors of Mineau's process formalism shall be considered, followed by Mineau's work, then the work of others, and finally some concluding remarks.

2.2 Actors

According to [Mineau 1998], actors take concepts of a particular type as input and compute individuals of a predetermined concept type.

In [Lukose & Mineau 1998], actors are compared with other executable CG systems (Mineau's processes and Lukose's MODEL-ECS — see elsewhere in this chapter). They report that Sowa undertook the development of a formalism for graphs of actors in which actor nodes are attached to CGs in order to form a dataflow graph [Sowa 1984] [Sowa 2000]. An actor is characterised as a finite, connected, bipartite graph with a set of concept and actor nodes. The latter are primitive computational elements ultimately defined in terms of functions or other actors (referred to as *executors* below), the combined effects of which yield a result, as discussed below. Directional arcs are drawn between concept and actor nodes, and actor nodes are drawn as diamonds rather than ellipsoids.

Curiously, in [Lukose and Mineau 1998] and as originally specified in [Sowa 1984], arcs between actor and concept nodes are drawn as dotted lines to distinguish actor graphs from CGs. This seems strange since actor graphs are essentially CGs with special kinds of relations. Indeed, in other places throughout [Lukose & Mineau 1998] actor nodes appear together with ordinary relation nodes in a single graph. Since actor nodes are themselves visually distinguishable from ordinary relations, the special arc notation seems redundant. Interestingly, ordinary arcs are used in Sowa's later work: [Sowa 2000].

[Lukose & Mineau 1998] distinguishes the different kinds of concept in an actor graph, as originally specified by [Sowa 1984]:

1. *Source*: concept nodes which are inputs to actor nodes, but take no outputs from other actor nodes;
2. *Sink*: concept nodes which are outputs from actor nodes, but provide no input to other actor nodes;
3. *Intermediate*: concept nodes taking input from actor nodes and outputting to other actor nodes.
4. *Conflicting*: sink or intermediate concepts, which take output from more than one actor.

A recursive factorial example actor graph in [Lukose & Mineau 1998] has a conflicting concept but no explanation is given as to whether this is considered valid. See the 'Experiments' section for a similar actor. This will not cause a problem unless two or more executors mutate a single concept's referent within a given actor invocation, an undesirable side effect, since the order of executor activations may not be deterministic.

As described by [Sowa 1984] and restated in [Lukose & Mineau 1998], control marks on the graph trigger actions to compute the referents of previously generic concepts. Control marks direct the flow of the computation and are of three kinds:

1. *Request marks ("?")* which initiate goal-driven computation starting at the concept node where the desired result is to go.
2. *Assertion marks ("!")* which trigger data-directed computation starting at the input concept node.

3. *Neutral marks ("o")* which have no effect upon computation, and represents an unmarked concept.

[Sowa 1984] gives an algorithm for a dataflow graph based upon these control marks which is illustrated via a recursive factorial example. The use of such assertion marks appears to be an implementation detail of the dataflow algorithm, the basic idea of which is that computational elements within an actor graph cannot proceed (or "fire") until all their input concepts have bound referents. The combined action of potentially many intermediate actors finally results in one or more bound sink concepts. For one such implementation see "The Implementation of pCG".

An actor type definition, as presented in [Sowa 1984] and [Lukose & Mineau 1998], declares which named referents correspond to input concepts and which to output concepts, suggesting that an actor's output referents may be easily identified. For example:

```
actor Sqr (in x,out y) is [Number: *x] -> <sqr> -> [Number: *y]
```

However, it is possible for an actor graph to have no accompanying definition statement, and examples of this are found in [Lukose & Mineau 1998] and [Mineau 1998], implying that source and sink concepts must be extracted from the graph by the actor implementation in that circumstance. If invocation originates with a graph matching or join operation as is the case with the use of actors in [Mineau 1998], the referents of generic concepts may be bound to specific values, obviating the need for explicit source concept parameter identification and binding. For example:

```
[Number: *x] -> <sqr> [Number: *y]
```

and

```
[Variable: #n] -> (Value) -> [Number: 4]
```

could be joined to form:

```
[Number: *y] <- <sqr> <- [Number: 4]-
                                  <- (Value) <- [Variable: #n]
```

such that when the actor is invoked, `*y` will be bound to the value `16`. See the chapters detailing the pCG language for further treatment of such issues, and the 'Experiments' chapter for examples.

There is a brief discussion in [Lukose & Mineau 1998] about the use of role concept types such as `Quotient` and `Sum` which are subtypes of `Number`, but it is unclear whether these are mandated when dealing with data of a particular kind. This raises issues regarding type, in particular the correspondence between concept type and referent data type [8]. The factorial example uses concepts which are subrange types, e.g. `[Number>0]` and the action of the `Identity` executor — which copies its input to its output — is said to be blocked if the referent of such a concept does not conform to the concept's subrange type [8]. An alternative is to have an executor, which only mutates its output concept's referent if its input is within a particular numeric range. If another executor takes as input the aforementioned output concept, it will be blocked. See the 'Experiments' chapter for an adaptation of the factorial example, which demonstrates this. See also 'The Implementation of pCG' and 'Future Work'.

One problem with actor relation types is that they do not participate in any type hierarchy, although whether they cannot *in principle* do so is unclear. This *may* account in part for Sowa's contention that actors are unnecessary since (he claims) CGs without actors provide a version of logic which is Turing complete [Sowa 2000b]. On the other hand, [Sowa 2000] points out that dataflow graphs which permit the definition of recursive functions and application of functions, and provide a conditional operator are Turing complete in the functional sense [Louden 1993]. From a computational perspective, both systems ought to be equivalent.

It is important to note that actors play a supporting role in Mineau's processes, as discussed in the 'Processes' section of this chapter.

[8] An intrinsic *conformity relation* of a given CG system maps a concept type `t` to a set of individuals (designators), yielding the denotation of `t`, δ`t`. For example, for the concept type `Number>0`, δ`Number>0` = {`x` | `x`$\in\Re \wedge$ `x>0`}, where $\Re$ is the set of real numbers. [Sowa 1984]

2.3 Demons

[Delugach 1991] presents an extension to CST to represent the temporal notions of a process or the transformations of a system. This involves the addition of a new CG node type: a *demon* node. Demons were suggested by Prolog's dynamic assertion and retraction capability, which Delugach believes are part of any usable logic system. Delugach points out that with the added semantics, it becomes possible to express notions of non-monotonic logic [Delugach 1990] [Delugach 1991].

Demons began as an attempt to convert STDs to CGs [Delugach 1990] requiring a dynamic structure. While use of a `(transition)` relation was considered, there is no ability in CST to represent change of state explicitly.

Previous work on temporal logic by the CG community had been concerned largely about working with tense in sentences. Delugach's focus is upon how one graph can be transformed into another, yielding a new assertion, i.e. change over time. [Delugach 1991]

Instead of a single-line bordered diamond in the display form, a double-line bordered diamond is used to represent a demon node, while in the linear form, `<<d>>` is used, where d is the name of the demon. Dashed arrows are used for argument arcs, similar to Sowa's original actor notation. It is interesting to note that in Delugach's later work of creating the CG editor CharGer, no distinction is made in the appearance of arcs, just in the shape of actor nodes (diamonds) [Delugach 1999]. A CG containing one or more demon nodes is referred to as a temporal graph, in a similar sense to the way a CG with actor nodes is referred to as a dataflow graph [Sowa 2000].

Whereas an actor uses for input the referent of its input arguments, and modifies the referents of its output arguments, a demon consumes its input concepts, and creates output concepts, doing so only when all of its input concepts are available [Delugach 1991].

A concept argument may be a context, and if so, the complete context including the associated descriptor graph will be asserted or retracted. If an input concept must still exist after the demon's activity, it may immediately be reasserted. This is achieved by having bidirectional arcs between a concept and demon node, indicating that the former is both an output and an input concept. Retraction and assertion of concepts is considered an atomic action, although internally, the retractions necessarily precede assertions [Delugach 1991].

The temporal graph `[B] -> <<d>> -> [C]` means: "if B is ever true, then C will be true at some future time". A temporal graph represents a series of states over time, with each state a present moment on the sheet of assertion [Delugach 1991].

There is some discussion in [Delugach 1991] regarding what to do with relations attached to a concept which is being transformed. The conclusion is that any relations not explicitly shown in a (temporal graph) are preserved. For example, consider the definition:

```
<<Burns>>-
         <-[Wood]->(Attr)->[Colour: Brown]
         ->[Ashes]->(Attr)->[Colour: Black]
```

Here, wood is being transformed to ashes. The concept wood may also have a relation: `(Attr) -> [Location: Somewhere]` which is not shown here, but would be preserved in the output graph, i.e. it would be attached to the resulting ashes concept. The key thing here is that essentially only the wood and colour concepts are changed. One observation that can be made in the case of the colour concept, is that an actor could have produced that transformation, although it would have to be directly attached to the actor node. There is some further discussion in [Delugach 1991] regarding retraction of attached relations, which would be dealt with if instead of retracting and asserting concepts, whole graphs were used instead. As will be seen, this is exactly what Mineau's process formalism does. Delugach considers the possibility of asserting or retracting relations without regard to their associated concepts, but as he freely admits, there are syntactic and semantic problems with this, for one, that a demon node is a special kind of relation and it would contradict CST to link relations directly.

[Delugach 1991] goes on to say that an input concept such as `[Time: *ignition]` and an output such as `[Time: *burnout]` could be attached to the burns demon. These concepts could then be attached via lines of identity (coreference) to concepts in a time line such as:

```
[Time: #t1] -> <Flies> -> [Time: #t2]
```

in order to represent the temporal dependencies in the system, i.e. that ignition precedes burnout, even though this is implied by the internal ordering of input retraction and output assertion.

[Delugach 1991] states that "a demon captures the notion of an entire subgraph being changed to another entire subgraph". It points out that an actor could represent this by the descriptor of an input concept (a context) becoming null to indicate retraction, and an output concept's descriptor being set to some graph. However, Delugach believes that this is stretching the intention of actors and that temporal graphs provide a more straightforward representation of assertion and retraction.

What is not made clear in [Delugach 1991] is exactly how a demon's input and output concepts should in general be related. In the wood to ashes example, it was assumed that these were so related as to be able to transfer any unspecified relations untouched, while modifying others (the `Color` relation) in specified ways. Ordinarily in CST, this implies some kind of specialisation operation. Delugach states that a demon must have some kind of canonical definition that specifies which concept types may be permitted as arguments. He also realises that this could lead to large definitions or a proliferation of sub-definitions. Perhaps an appropriate concept type hierarchy could solve this problem. But could one really consider wood and ashes to be in a subtype relationship in a concept type hierarchy?

Delugach admits that there are unresolved issues, such as what the semantics of negated or nested contexts linked to a demon should be. Nevertheless, the notion of demons seems fairly well-defined and useful. One intriguing possibility suggested in [Delugach 1990] is that a more powerful demon might also be able to modify the type hierarchy, although the utility and implications of such a capability are not explored in that paper, and the idea is not mentioned in [Delugach 1991].

2.4 Processes

[Mineau 1998] points out that the representation of dynamic knowledge does not fall within the current scope of CST — but believes it should. He uncovered a need for this in a major R&D project which aimed at capturing knowledge about corporate processes. The representation of such business processes using CGs is briefly discussed in [Gerbe´ 1998]. They give as an example, the steps involved in the fabrication of a window.

In this activity there are certain dependencies, inputs, and outputs. For each step there are preconditions and post-conditions. An example of a precondition here is that before a window can be assembled, a fabrication order must have been updated with the specification of frames and panes which conform to a client order. An example of a post-condition is that when a frame is built, the fabrication order is updated, likewise when a window pane is cut. [Gerbe´ 1998] outlines a definition of processes in terms of activities and events but says nothing about mechanism.

In [Mineau 1998] it was proposed that dynamic knowledge could be represented by processes which Mineau defined in terms of transitions between states; previous and new states can be characterised by the pre and post conditions associated with them. The triggering of a transition to a new state depends upon the truth of the preconditions. Mineau describes these state changes in terms of the assertion and retraction of graphs in a CG system. [Lukose & Mineau 1998] refers mainly to assertion of graphs or negated graphs in post-conditions. [Mineau 1998] somewhat confusingly refers to negation *and* retraction in post-conditions. The assertion of a negated graph is quite different to the retraction or erasure of a graph which can viewed as a generalisation operation involving replacement of a graph by the blank graph [Sowa 1999]. As [Delugach 1991] says of concepts or contexts, but which applies equally well to whole graphs:

> Negation of a concept (or context) is not the same as the retraction of the concept (or context). Explicit negation means that an assertion is false; mere non-appearance says that we no longer assert the fact — its truth or falsity is not known.

As an aside, this helps to explain why erasure of a graph from a negated context is not permitted in Sowa's Rules of Inference [Sowa 1999]: one cannot say declare the falsity of *nothing* (a blank graph).

Regarding retraction vs negation, it is worth making the point that the former reduces the number of graphs in a KB, while the latter increases it. This has obvious implications for a system in which a KB will be searched often. See the chapters describing the pCG language elsewhere in this thesis for more.

Mineau [1998] suggests that an algorithm — such as iterative factorial — can be automatically translated into a set of pre and post condition pairs. A context-free language like C or Horne Clauses (Prolog) could plausibly be, but what of more complex knowledge, e.g. business processes? This is a non-trivial problem which is barely treated in [Mineau 1998] or [Lukose & Mineau 1998]. See 'Future Work'.

[Mineau 1998] proposes a process statement as an extension to CST, which is used as the basis of the process statement in the pCG language. Formally, it is as follows:

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

The statement specifies a *rule set* comprising pairs of pre and post-conditions — $r_i = \langle pre_i; post_i \rangle$ — such that each process has *n* transition rules (where $1 \leq i \leq n$), and it is this which Mineau suggests can be automatically generated from an analysis of the algorithm. Input parameters are CGs which are incorporated into the precondition of the first transition rule when the process is triggered. Output parameters are incorporated into the final transition rule. Whereas the effect of a post-condition is normally localised to a process, the output parameters induce changes in the caller's KB. These mechanisms are outlined as is the invocation and clean up of a process [Mineau 1998]. The pCG language chapters detail this mechanism.

There is a lengthy example which shows the iterative factorial algorithm being converted manually to a rule set for a process, the means by which that process would be triggered by an asserted CG — `[Line:#L0] <- (to_do)` — and the pre and post conditions corresponding to each line of the algorithm. Mineau briefly mentions the use of coreference within the process to capture the output parameter's value which will be asserted in the KB when the process exits. The example does not show the dynamic execution of the process, just its definition. The definition of a process shows only generic concepts (e.g. `[Line: *x]`) not individuals, in much the same way that a C program only shows variable definitions, not their run-time values. At run-time, this would correspond to a specialisation from generic concept to individual. Mineau notes that the execution path (i.e. via transitions) is dynamically generated due to the assertion and retraction of CGs in the post-conditions of each rule in the rule set. He also points out that — except when iteration is required — rules don't refire once executed [Mineau 1998]. See the 'Experiments' chapter for a working example in the pCG language.

Mineau [1998] gives a synchronisation graph to illustrate how each part of the factorial algorithm is dependent upon other parts. See Figure 2-1 and Figure 2-2. For example, before the first line of the while loop's body can begin, certain variables must have been initialised and the loop condition must have been checked. The example definition shows how these dependencies are incorporated into the preconditions of rules. A similar notion is found in graphical dataflow programming languages such as Prograph [Pictorius 2000] [Steinman 1995] such that all inputs to a particular rule must be computed before that rule will fire. Interestingly, actors themselves essentially dataflow-based expressions, are used in [Mineau 1998] to represent such imperative constructs as relational and arithmetic operations (e.g. >=, !=, *) within processes.

```
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; }
```

Figure 2-1 The iterative factorial algorithm from [Mineau 1998]

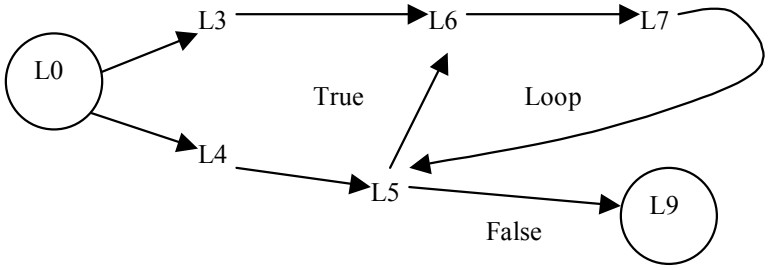

Figure 2-2 The synchronisation graph for [Mineau 1998]'s iterative factorial algorithm

In the conclusion of [Mineau 1998], it is suggested that a process should be considered in terms of its input and output rather than its internal structure, where the only side effects will be assertion or retraction of CGs in the KB. However, since the defined variable of an output parameter (e.g. `[Integer: *result]`) must coincide with some variable which has been generated within the process, such a black box approach is not entirely possible. See the 'Experiments' chapter for an implementation and discussion of the iterative factorial process proposed by [Mineau 1998].

Nevertheless, the notion of processes would seem to require minimal modification of CST itself, while at the same time providing considerable expressive power.

In another paper [Lukose & Mineau 1998], five objectives for Mineau's representation of processes were set out:

1. Allow automatic execution of processes. It is claimed that all that is needed for the user to know is the entry node of the synchronisation graph in order to engage a process. As pointed out above, this is not quite true since one also must know the names of the generic variables in output graphs.
2. Incorporate declarative and dynamic knowledge to support the implementation of a Conceptual Programming Environment (CPE).
3. Avoid human intervention in process execution.
4. Be as simple as possible to facilitate application development.
5. Allow auto-validation of processes.

As will be seen, 2. is a significant motivation for the current work which is a step along the road to a CPE. It does not seem unreasonable to suggest that the CPE should also facilitate objectives 1., 3., and 4. It is not entirely clear what auto-validation means, but at the very least, a process can be developed and executed, and the outcome observed. If what is meant instead is some kind of formal verification, then this is somewhat more difficult, but having a formal notation in the form of a programming language would at least make it easier to embark upon such an activity.

[Lukose & Mineau 1998] claims that the process mechanism is more general than Sowa's original actors, and that it provides a framework which greatly improves the representational power of CST. Further, Mineau [1998] suggests that actors and demons are specialisations of processes. Processes generalise demons by taking CGs as input and asserting or retracting CGs as the result of their

processing. It is questionable as to whether the relationship is really this simple however, since as has been mentioned, Mineau's factorial example uses actors in preconditions. So, on the one hand we have the suggestion that there exists a generalisation hierarchy between actors, demons and processes, i.e.

```
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 the relationships between concept referents and arbitrary values could not be determined, although there exist alternatives in the form of extensions to CST such as those presented in [Corbett 2000a]. Sowa and others suggest that actors are not required however — see the 'Actors', 'Executable Conceptual Graphs', 'Synergy', and 'Special nodes vs direct execution of CGs' sections. On the other hand, since a concept is a singleton graph, a case could be made for a process being a generalisation of a demon as Mineau has suggested, since in both cases, assertion and retraction is the primary mechanism.

Both [Lukose & Mineau 1998] and [Mineau 1998] suggest CG notations for processes but don't mandate them. [Lukose & Mineau 1998] proposes the use of contexts such as `[CONDITION: g]` to pass a graph as an input parameter to a process and `[STATEMENT: h]` to pass an output parameter from a process to the caller. Here, `g` would be added to the first rule's precondition, while `h` would be added to the final rule's post-condition and asserted in the caller's KB when that rule fires.

[Mineau 1998] adopts the double-diamond notation from [Delugach 1991] for process nodes. See 'Special nodes vs direct execution of CGs' for an alternative.

[Lukose & Mineau 1998] points out that the assertion of some graph `g` may either:

1. Not cause a change in the current state of the system if `g` has already been asserted or retracted, or
2. Necessitate knowledge revision, e.g. `g` may not be able to be asserted if it would conflict with some other graph `g'`.

In the second case, Mineau thinks that the system should block and the user should be notified, presumably so corrective action can be taken before proceeding or terminating. [Mineau 1999a] deals with constraints on processes which could yield a complete truth maintenance system such that KB consistency is preserved. See also 'Future Work'.

In [Lukose & Mineau 1998], it is suggested that coreference is global such that a concept in an actual parameter graph can be represented in the output. For example, as mentioned above, a singleton graph such as `[Integer: *result]` could be passed as an output graph (final rule's post-condition addition) to the factorial process, and the activity of the process results in the variable `*result` being bound to a particular value. What the second paper does not spell out, but the first paper does is that the graphs and coreference variables associated with the activity of a process should go out of scope when the process has ended. The alternative is the pollution of the caller's KB with unwanted graphs and generic variable bindings. It should also be acknowledged that the identification of generic concept variables with coreference is not the same as specified in [Sowa 1999] and elsewhere. It is instead very much like the notion of generic variables to be found in Sowa's original description of actors in [Sowa 1984]. See the discussion of the pCG language elsewhere in this thesis for more details.

[Lukose & Mineau 1998] claims that the process sequencing mechanism is simpler than Sowa's original actor token passing mechanism in [Sowa 1984]. This claim is debatable given the relative complexity of the algorithm set out in [Mineau 1998], and the experience of this author in the development of the pCG language. Moreover, the encoding of particular problems as processes vs actors is often non-trivial. This author contends that in simple cases, such as the factorial algorithm, processes lead to substantial overheads in terms of the size of the rule set and the time taken for computation, when compared with an actor. See 'Experiments' for examples of this. Of course, Mineau has only used factorial as an example to illustrate the process mechanism. The Sisyphus-I room allocation problem [Linster 1999] is a fairer example with which to judge the scalability of processes, in addition to being a more likely use for them. See 'Experiments' for an implementation of this problem as a process.

In both process papers, errors may be found in the examples, e.g. missing intermediate concepts in an actor graph, incorrect generic variable names, inconsistent arc directionality in process graphs, and so on. This is to be expected in a paper-based exercise. What is important is that Mineau has provided sufficient details for others to attempt an implementation of the process mechanism. See 'The pCG Language' and 'The Implementation of pCG' for one such attempt.

One quite appealing aspect of Delugach's temporal graphs is that like dataflow (actor) graphs, they are CGs, unlike processes which consist of multiple CGs and require additional scaffolding (e.g. the process statement). Again, see the chapters detailing the pCG language for the practical implications of this.

One of the major motivating factors of demons is the representation of temporal notions, and the ability to carry out temporal logic. While Mineau does not explicitly tout this as a motivation, the nature of a process and the process formalism is consistent with it. Further, given the claim that Mineau's processes are a generalisation of demons, for processes to be at least as expressive as demons requires them to be capable of handling temporal logic. Figure 9 of [Delugach 1991] is a table describing fundamental temporal logic operators and how aspects of Delugach's temporal graphs correspond to them. Table 2-1 is a similar table which adds correspondence to Mineau's processes in an effort to show how, in principle, Mineau's formalism ought to be capable of supporting temporal logic. The first 3 columns have been taken directly from Delugach's table.

Operator	**Meaning**	**As interpreted in Temporal Graphs**	**As interpreted in Processes**
F *A*	*A* is true at some future time; "Eventually".	*A* is input to a demon that has not been enabled.	*A* is a post-condition graph which has not been asserted since the corresponding rule has not fired, but will be when its precondition graphs all match.
P *A*	*A* was true at some past time.	*A* is input to a demon that has already been enabled.	*A* was retracted by a process by virtue of being a (retraction) post-condition graph in a rule which fired.
G *A*	*A* will be true at all future times; "Henceforth".	*A* is not an input concept to any demon.	*A* has been asserted but is never targeted for retraction by any process rule's post-condition.
H *A*	*A* has always been true in the past.	*A* is not an output concept to any demon.	*A* is asserted, but not by a process.

Table 2-1 Temporal logic as supported by demons and processes

These in-principle correspondences must of course be tested by an implementation of the formalisms. Further exploration of the foundations of temporal logic is beyond the scope of this review.

The distinction between actors and demons or processes can be viewed in terms of the fact that actor activity is confined to concept referent mutation whereas the latter two formalisms concern themselves with dynamic assertions and retractions. However, it is important to note that the distinction between demons and processes is not as clear cut as one may initially think given that a retracted or asserted concept may be a context, e.g. a `PROPOSITION`, the referent of which is a graph. Some points of difference are that:

1. A process permits arbitrary graph matching via projection [Mineau 1998]. This may be permitted in temporal graphs, but [Delugach 1991] does not say.
2. Process inputs (preconditions) are not retracted.
3. Since complete graphs are being asserted or retracted, the problem of what to do with relations found in demons is not present for processes.

Greater detail regarding key aspects of the process mechanism is given in 'The pCG Language' and 'The Implementation of pCG'.

2.5 MODEL-ECS

[Lukose & Mineau 1998] discusses actors as originally envisaged by Sowa, processes by Mineau, and MODEL-ECS by Dickson Lukose. MODEL-ECS as an acronym is not explained in this paper. One can hazard a guess that ECS means Executable Conceptual Structures, and [Lukose 1993] confirms this.

A key feature of MODEL-ECS according to Lukose is that it is actor-based, and is an executable conceptual modelling language. An actor in this context differs considerably from Sowa's meaning, being more like a class definition in or to an Object-Oriented (OO) languages (but see also 'Object-Oriented CGs'), defining an interface to methods. An *actor graph* is distinguished from an *actor* and consists of:

- Concept type (a differentia graph defined in terms of a lambda expression)
- Actor type (a script with a main method)
- Precondition (a list of graphs)
- Post-condition (a list of graphs)
- Delete list (a list of graphs to be deleted)
- Short and Long Term memories

[Lukose & Mineau 1998] [Lukose 1993]

In commenting upon Lukose's mechanism, [Cyre 1998] makes the statement that:

> An actor graph consists of the type definition of an *act* concept supplemented with an actor whose methods are invoked by messages. Execution of an actor graph terminates in a goal state which may be a condition for the execution of other actor graphs.

[Lukose 1993] states that "All concepts which are subtypes of the concept "ACT" are considered to involve an activity". It continues:

> An activity warrants a "script" to represent an appropriate sequence of actions in order to realise the particular "act". The type definition graph represents the type definition of an *act*, where all the concepts and the relations represent the essential or obligatory properties that must hold for the type, while the actor enables the representation of appropriate scripts. Thus, a new form of abstraction called the "Actor Graph". The actor graph definition links both the type definition graph and the actor for a particular "act". With these definitions in place, a particular "act" can be executed by simply sending an appropriate message to it.

MODEL-ECS also defines the following special modelling constructs:

- Problem maps: `[PM]`;
- Knowledge sources: `[KS]`;
- Special conceptual relations to express temporal relationships, such as `(FBS)` for sequence (finish-before-start), and `(SWS)` for concurrent execution (start-when-start);
- Special actor graphs (Lukose's term; concepts in fact) to represent such notions as:
  1. Predicates such as `[LTEQ]`, `[TRUE_TEST]`, `[FALSE_TEST]`;
  2. Statements such as `[ASSERT]`, `[RETRACT]`, and `[INCREMENT]`;

[Lukose 1997b]

The special (built-in) actor graphs are quite different from the user-defined ones mentioned earlier, since there is no explicit script or graph lists. It may have been better for different terminology to have been used so as to make clear the principles and basic entities in the formalism. One could of course always think of new special constructs to add, but many can be replaced by a more general notion such as Sowa's actors or dataflow graphs, with underlying arbitrary functions. Lukose does support such a capability in the form of user-defined actor graphs, as has been mentioned already and will be discussed further below.

A problem map is described in [Lukose 1993] as being composed of a number of actor graphs in a particular sequence. Each actor graph's pre and post conditions impact upon the run-time execution sequence of a problem map. This is a similar notion to the order in which rules fire being determined by each rule's preconditions in one of Mineau's processes. Also, like processes, [Lukose 1993] claims that all state information takes the form of CGs in the system.

In [Lukose 1997b], the notion of a problem map incorporates the aforementioned special constructs, including nested problem maps. A common idiom in Lukose's papers is:

```
[PM: α] -> (FBS) -> [PM: β]
```

where α and β are knowledge sources or arbitrarily nested problem maps. The special (FBS) relation imposes a temporal constraint upon the problem maps. Suppose that α and β were defined respectively to be:

```
[KS: [TRUE_TEST: [DESIGN] -> (OBJ) -> [HOUSE: *x]]]
```

and

```
(FBS)-
      <- [KS: [RETRACT: [DESIGN] -> (OBJ) -> [HOUSE: *x]]]
      -> [KS: [ASSERT: [BUILD] -> (OBJ) -> [HOUSE: *x]]]
```

The first problem map would search for a particular house design in a KB [9] and if found, TRUE_TEST would permit the second problem map to execute, otherwise nothing further would happen. The second problem map retracts the stage of house design, then asserts the stage of house building. This is not dissimilar to the functionality of assertion and retraction in Mineau's process formalism. It is not difficult to imagine however, that such graphs when shown as fully nested (e.g. with β replaced by the second graph above) can quickly become unwieldy.

Using such constructs, more complex ones can in turn be built, such as: `if`, `while`, `repeat`, `case`, and `for`. Some, especially the `for` construct are quite complex and in [Lukose & Mineau 1998] the reader is referred to four other papers, including [Lukose 1993] and [Lukose 1997b]. The latter gives a lucid account of these primitive modelling constructs, as Lukose calls them. The `if` construct in [Lukose 1997b] looks like this:

```
[PM: {[KS: [TRUE_TEST: ϕ]] -> (FBS) -> [PM: α],
      [KS: [FALSE_TEST: ϕ]] -> (FBS) -> [PM: β]}]
```

where ϕ is a CG acting as a condition, and α and β are nested problem maps or knowledge sources. Instead of simple CGs or special constructs, ϕ, α, and β could instead be user-defined actor graphs.

Loops are possible by defining the final problem map in a `(FBS)` sequence as being coreferent with the outermost context of one of these constructs. One curious aspect of the complex modelling constructs of [Lukose 1997b] is that they make use of sets, as can be seen from the two [KS] concepts in the above `if` construct [10], but sets have no internal ordering. Ordered lists would be a more accurate representation. While sequence is not important for `if`, it may be important for a `case` construct since it may be necessary for one case to be evaluated before another.

[Lukose & Mineau 1998] defines a `factorial` problem map in terms of the special modelling constructs, but first, actor graph concept types and associated actor scripts such as `ADD`, `SUBTRACT` and `IDENTITY` are defined. In each, an `[ACT]` concept is used, and apart from the use of a concept as the active element in the graphs instead of a relation, these actors have a similar appearance to Sowa's actors. An example of such a graph is:

[9] [Lukose 1997b] calls this "working memory", but the relationship to long or short term memory as used by an actor graph is not clear.

[10] [Lukose 1997b] uses {…} in the referent of concepts to indicate a set of graphs, and describes [PM: {[PM: α], [PM: β]}] as non-deterministic execution of α and β.

```
actor SUBTRACT(*x, *y, *z) is

      [ACT]-
           (ARG1) -> [NUMBER: *x]
           (ARG2) -> [NUMBER: *y]
           (RSLT) -> [NUMBER: *z]
```

The factorial example of [Lukose & Mineau 1998] is not simple to comprehend without some explanation, and considerably more complex than the equivalent function in a procedural language. The same criticism can of course be levelled at the factorial process example of [Mineau 1998]. [Lukose 1993] admits that a problem map is "…a highly nested conceptual graph, thus a complex form of abstraction." But abstractions *should* simplify a system, not make it more complex.

Like Sowa's actors (dataflow graphs), there is for user-defined actor graphs ultimately a reliance upon some underlying function or executor. In MODEL-ECS, such functions are written as scripts in a language which appears, from an example given in [Lukose 1993], to be Prolog. Such scripts are responsible for more than just the execution of a simple function however. Each has a main method which is responsible for checking preconditions which if satisfied leads to execution of other code in the script involving the manipulation of CGs, and finally the instantiation of post-conditions. What Lukose calls "skeleton" pre and post-condition graphs are stored in a long-term memory which a script's main method retrieves. An actor graph's short-term memory contains current graphs which are compared against the retrieved preconditions. Long-term memory seems analogous to Mineau's process rules, while short-term memory plays a similar role to a process's temporary local knowledge base.

[Lukose 1993] acknowledges the influence of the STRIPS problem solving formalism, key features of which are preconditions, delete lists and add lists. Upon the satisfaction of particular preconditions, the system's world has graphs added and/or deleted [Sowa 2000]. It seems likely that MODEL-ECS's actor graph post-condition and delete lists have a similar use. The relationship between the post-condition graph list and the delete list is not explained in the papers cited however.

While one can appreciate the building blocks of MODEL-ECS and the general trend of building more complex constructs from simpler ones, the essential ideas cannot apparently be concisely explained in a single comparative paper such as [Lukose & Mineau 1998]. To be fair, Sowa's actors and Mineau's processes are devoted considerable space in the latter paper. However, Mineau's process formalism does appears to have less "moving parts" than MODEL-ECS leading to less perturbation of CST.

[Lukose 1993] claims that problem maps permit knowledge fusion from multiple experts, but no reasons are given as to why problem maps are especially good at this. Another claim is that a collection of problem maps ought to be able to synthesise a new problem map. While new graphs can be asserted as post-conditions in an actor graph script's main method, what of all the other components of the actor graph, more than one of which must be created to form a problem map? This and the reliance upon scripts casts doubt upon the earlier claim ([Lukose 1993]) that all state information in this formalism is CG-based.

Further analysis of MODEL-ECS is beyond the scope of this review. However, the final section of [Lukose & Mineau 1998] makes various claims about the similarities and strengths of MODEL-ECS and processes, which will be considered next.

2.6 Relationship between MODEL-ECS and Processes

Both MODEL-ECS and processes are characterised as permitting the execution of processes, and both permitting parallelism, claims which while not unreasonable, must be tested.

The following deserves comment:

> Lukose's work focuses on the graphical aspect of process modelling, and pinpoints implementation details such as memory organization, which would support such a system. [Lukose & Mineau 1998]

It is not clear how the MODEL-ECS linear form graphs presented offer advantages over those of processes. The memory organisation required for MODEL-ECS is not elucidated in [Lukose & Mineau 1998], although [Lukose 1993] gives more detail. It continues:

> Mineau's approach focuses more on the conceptual representation of a process in a state transition machine. It advocates automatic translation of a process description from a linear algorithmic modelling language to a logical representation language….In effect, the breakdown of a process into a set of transition rules results in the loss of structure of the process itself…it may be beneficial to the application modeller to use a graphical notation which explicitly shows the structure of the process and its relation to other processes. [Lukose & Mineau 1998]

Here, the claim is that MODEL-ECS is superior to Mineau's processes for process definition and exploration, but this claim is not supported by arguments. One might instead want to advocate the use of a more abstract notation such as Petri nets for this task. It is certainly true (see 'Experiments') that a process rule set can make the process which is being modelled difficult to understand, but MODEL-ECS appears to be just as complex on paper.

The final conclusion of the paper is that MODEL-ECS and processes are complementary approaches such that the former could be used for obtaining an initial model of a process, subsequently using the latter to create a set of transition rules, possibly in an iteratively refining manner. Again, this claim while not outside the realm of possibility, requires supporting evidence. One could just as easily claim that successive refinement of a process rule set alone is sufficient, and indeed, this approach was taken in 'Experiments' for the Sisyphus-I process.

Lukose's MODEL-ECS and Mineau's processes provide a similar capability, but the former does so at the cost of greater complexity and arbitrariness. In short, Mineau's is a more minimal formalism than MODEL-ECS. A minimal solution means on the one hand, less modification to CST, but on the other hand a given construct may be less succinct.

2.7 Executable Conceptual Graphs

[Cyre 1998] asks the question: given an arbitrary CG, what would be required to execute it? Cyre 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.

[Cyre 1998] gives a ball-throwing graph example, and considers what would be required to execute it. The graph contains no actor nodes, but a concept of type `Throw`, which could be considered as having an effect such as a change of position. Figure 2-3 reproduces Cyre's ball throwing example graph:

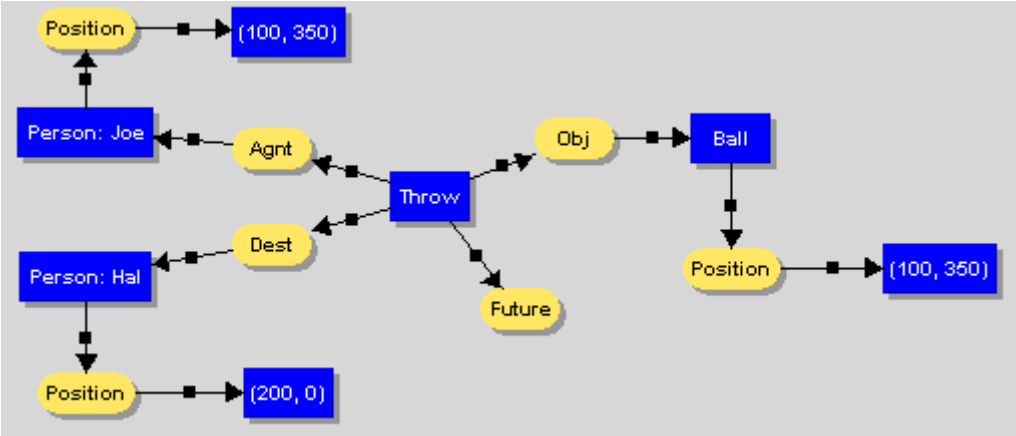

Figure 2-3 Cyre's ball throwing example

This graph says that there will be a future throw action, the object of which is a ball located at a particular two-dimensional coordinate. In reality there would be a third coordinate of course. The agent

of the throw is Joe who is at the same position as the ball, and the destination is another person, Hal, at a different location. The intention is that an underlying action associated with the `Throw` concept would ultimately cause the ball's coordinate concept to change to that of Hal's (either by referent mutation or the addition of a coreference link), and the tense of `Throw` would be changed to the past, i.e. `Threw`.

[Cyre 1998] laments that most researchers have extended CST with special nodes (e.g. 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. It is interesting to note that concepts rather than relations are being used as the active elements in a graph. This is a departure from actors, demons, and processes. See also 'MODEL-ECS' and 'Synergy', which has a similar departure.

Some unresolved issues arise from Figure 2-3:

- The `Throw` concept must not only be aware of other concepts and "…how the execution of the throw action affects these adjacent concepts" [Cyre 1998], it must also be aware that the concept in question is at the end of some string of relations. In this case the concept is an argument to a position relation attached to the concept representing Hal, and the latter is connected via a destination relation to the `Throw` concept. Later in the paper, Cyre *does* say that his system uses conceptual relations such as `Position` and `Object` to "…describe the interaction among these concept types…" and that they direct "…the simulator in assigning attributes and generating events during simulation." [Cyre 1998] The latter suggests a shift in emphasis from what a particular concept is aware of to what a set of relations must be aware of.

- The suggestion that the `Throw` concept's type would be changed to the past tense, i.e. `Threw`, is significant for a few reasons:

    1. Either the `Throw` concept modifies itself, or the `Future` relation is also an active element contrary to Cyre's claims that only "…actions need to have 'personalized' procedures" and that only concept "behaviour types" actively participate in the simulation [Cyre 1998]. If the `Future` relation is not an active element, then what *is* its role?

    2. What are the semantics associated with modifying the type label of a concept in this way, and how does this impact upon CST?

    3. How is it possible to execute the graph more than once, after `Throw` becomes `Threw`? Is there some way to re-initialise the graph?

Such issues relating to this simple graph are not considered before the paper moves on to a more complex example in which active elements have effects upon other active elements, the semantics of which are also not discussed in detail.

The modification to graphs in Cyre's system is limited to what he refers to as concept attributes (referents), which is of course computationally equivalent to actors. Rather than a multitude of primitive procedures, Cyre suggests a modest set of operations such as copy, replace, combine and so on, defining procedures such as those needed for the `Throw` concept in terms of these. But this is also true of Sowa's actors whose executors are defined in terms of other actors, or functions defined in terms of primitive operations in the language.

[Cyre 1998] outlines related work, including actors [Sowa 1984], demons [Delugach 1991], and MODEL-ECS [Lukose 1993]. Cyre also mentions work in CG based state transition systems inspired by human behaviour [Bos 1997] (see also 'Applications' section). However, Cyre considers that his approach treats states and events more generally, considers interaction between behaviour concepts (events, states, actions) and other entities, and differs by taking inspiration from digital logic simulation techniques and examination of computer system modelling notations rather than human behaviour.

Digital logic simulators are based upon signal change and simulation time. Statecharts are consistent with this, and permit representation of nested finite state machines. Cyre's algorithm is based upon the

statechart formalism, but no indication is given as to whether an implementation has been attempted. He goes on to say more about this formalism:

> Each transition can be triggered by an event and predicated on a condition…When a transition occurs, an action may be stimulated…Actions and events may generate new events. [Cyre 1998]

To develop a general CG simulator, the concept types that will take part in the simulation must be defined. For example, *behaviour* types actively take part in the simulation (e.g. `Throw`), while *object* types are passive, but their attributes may be modified (e.g. ball coordinates). State and event concept types are also defined. Cyre maps his simulator types to Sowa's traditional concept types from [Sowa 1984] to show the generality of his approach [Cyre 1998].

What this amounts to is that certain types must be marked as having special roles by some interpreting system, instead of having special actor nodes. It is important to realise that there must always be *some* fixed point, some anchor which permits a system to identify the active elements in an executable CG system. Which kind of anchor is more valid? See also 'Special nodes vs direct execution of CGs'.

[Cyre 1998] points out that CST permits concept types to be CGs (lambda expressions), such that graphs can be executed recursively in his system. This is rather like recursive actor invocation in which an executor in a dataflow graph is not defined in terms of a function, but another actor graph. See 'Experiments' section.

[Cyre 1998] describes the simulation cycle in some detail. He first defines the notion of an *incident* which is associated with a specific concept and has attributes such as: concept type, scheduled simulation time, and type of operation to be performed. Various lists in the system keep track of these incidents and the state of the CG during simulation, e.g. a queue of pending activities, lists of current and future activities, currently true states, and so on. The simulation cycle involves:

1. Getting the incidents to be processed during the current cycle (based upon simulation time);
2. Updating attributes of affected object concepts; This may affect the true states list;
3. Processing of remaining current incidents: states, actions, events with consequences placed in the future list;
4. Action concept execution, if slated for the current cycle;
5. Appending the future list to the queue for the next cycle.

Cyre points out that concept attributes (step 2) must be processed before action or event incidents (step 3) since they can impact on the truth of preconditions for execution of actions and the firing of events.

[Cyre 1998] goes on to discuss the generation of new incidents *by* the simulation, for use *in* the simulation. For example, a `Generator` relation whose signature is `<Event,Action>` will result in the generation of some action incident corresponding to the types of the actual event and action concept arguments. This would appear to take place in step 3 above. Cyre points out that without such new incident creation, the simulation would soon stop. This is comparable to saying that without the assertion and retraction of CGs during process execution (see 'Processes'), a process would not continue for very long.

Within the discussion of new incident generation, Cyre makes a surprising remark: "Disjunction of relations is not conveniently represented in conceptual graphs. For this purpose we define new relations *or* and *xor* to synthesise artificial disjunctive concepts." [Cyre 1998]. Surprising, since Cyre has taken great care to permit execution of a CG without special constructs. Such relations are arguably no more special constructs than specially marked concept types are, but one has to wonder at the advantage of this over an as-needed creation of *or* or *xor* actors. The issue of disjunction is also an issue for Mineau's process formalism, whose preconditions and post-conditions consist of conjunctions of CGs. See also the chapters documenting the pCG language.

Discussion about the generation of the time component of incidents hints at the possibility of support for temporal logic in Cyre's formalism, but this is not explicitly discussed in [Cyre 1998].

While Cyre's proposal appears promising, it does seem, like MODEL-ECS, to be complex and containing numerous contingent aspects. It would be interesting to see a working implementation however.

2.8 Object-Oriented CGs

[Mineau 1998] cites other work on process-related primitives and believes that a primary motivation in the CG community for processes being added to the CST is for the creation of an OO architecture on top of a CG-based system, such as [Ellis 1995], but does not discuss this further. In [Mineau 1998], the emphasis is on simple primitives for modelling processes, which could be extended in arbitrary ways.

[Cyre 1998] claims that Lukose's actor graphs [Lukose & Mineau 1998] and problem maps are OO extensions of CGs which provide executability, at least in part because methods are invoked via messages. To qualify as object-oriented, there are other criteria which must be satisfied by any formalism however [Wegner 1987], and this issue will be raised again later in this section.

[Kabbaj 1999b] describes a CG-based OO visual programming system, outlined elsewhere in this review (see the 'Synergy' section).

[Ellis 1995] starts by reinforcing the usual motivations for OO programming, such as encapsulation, inheritance, information hiding and the reuse these permit, and says that Sowa applied OO ideas to CGs in a 1993 paper. Ellis proposes an OO model for CGs which is state-based, claiming it to be simpler and more illustrative than Sowa's approach, and more efficient for programming and theorem proving since fewer operations are required to achieve the same effect. Ellis uses Sowa's example of car ignition. Sowa's system requires an 18 step proof to show that a car engine will be in a running state after an ignition message is sent to a car object, while Ellis's formalism yields a 4 step proof.

In Sowa's notation (shown in Figure 2-4), an object consists of a concept that is coreferent with another concept in a context, the latter concept being at the head of a graph which describes the first concept. For example, `[Car]` is coreferent with `[Automobile]` in a context whose type label is `Automobile`, and where `[Automobile]` is an argument to relations such as `(kind) -> [Model]`, `(part) -> [Ignition]` and so on. A particular Car instance, e.g. `[Car: PCX999]`, may be created which inherits this description and to which new relations may be attached, as for any CG.

In this system [Ellis 1995] explains, messages are passed to such an instance by drawing a concept into the `Car` context, e.g. `[TurnRight]` which corresponds to an underlying method [11], and is supported by Sowa's extended *OO Rules of Inference* which includes import and export of messages to and from contexts. Ellis instead defines a method as a relation on some pre and post states of an object.

[11] The term *method* is used in some OO languages (e.g. Java) to refer to a function which is present within an object and which has access to the state of that object.

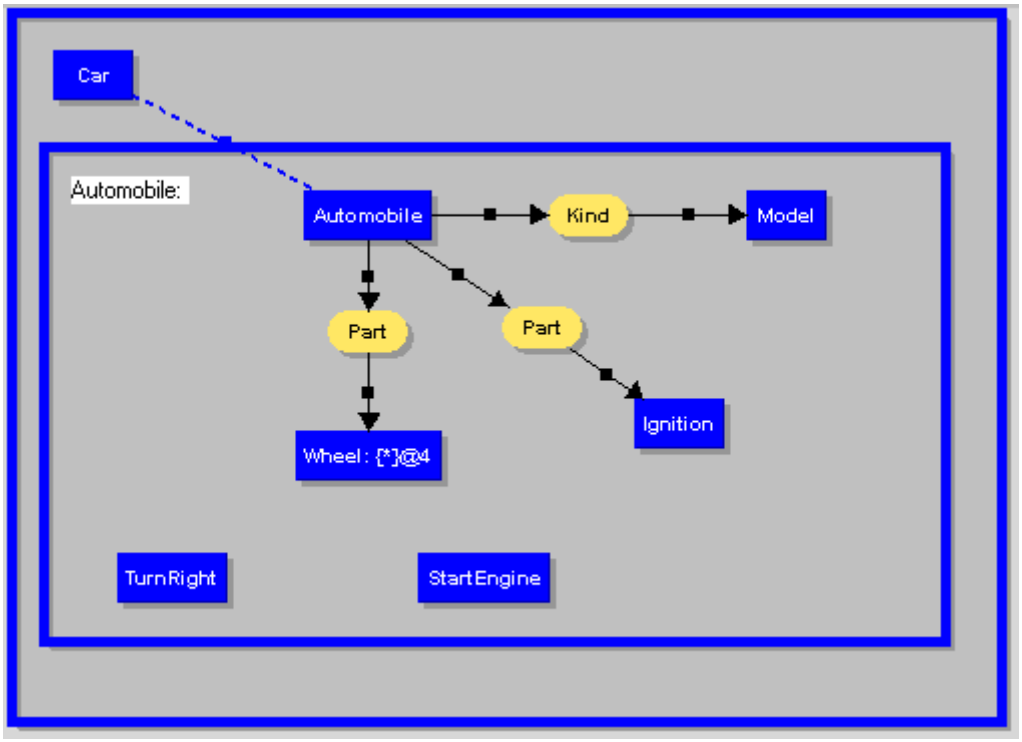

Figure 2-4 A Sowa-style object

[Ellis 1995] goes on to explain that Sowa also defines an associated `StartEngine` *process* which "…describes a state transition, from the time *t1 to the time *t2 5 seconds later." This involves a `Car` *c and an `Ignition` *i. This StartEngine process is itself an object whose coreferent context describes the state transition using CGs and implication — `[If: g [Then: h]]` — to represent an engine which isn't running at time *t1, but is at time *t2. Sowa's 18 step proof is given in terms of his OO extensions to the Rules of Inference. Ellis's opinion is that this is a long and clumsy proof.

[Ellis 1995] presents an informal mapping from Object-Z (an OO formal verification language) to CGs and provides a 4 step proof of car ignition based upon method (relation) expansion which Ellis states is a canonical equivalence operation, modus ponens, time constraint computation, and erasure of previous state information and links. The appearance of an object in Ellis's formalism is not dissimilar to Sowa's. A method however consists in a relation which may have an arbitrary number of concept arguments, but the first and last of these are coreferent with (represent) the pre and post states, respectively, of the object with which the method is associated. For the car example, Ellis's method is similar to the following (where `". . ."` represents coreference in Ellis's notation — see below):

```
[Car: *x descr] . . . [Car] -> (StartEngine) -> [Car] . . . [Car: *y descr]
[?x] -> (succ) -> [Event: [TurnOn] -> (ptnt) -> [Ignition]] -> (succ) -> [?y]
```

where *descr* is a descriptor graph similar to Sowa's Automobile context of Figure 2-4, and *x and *y are defining variables and ?x and ?y are a standard LF notation called defining and bound variables providing a way to refer to a concept elsewhere in a LF graph [Sowa 1999]. The **[Car]** concepts (in bold) are the first and last formal method parameters — there are no others in this example — while the `Car` concepts containing descriptor graphs represent the desired pre and post states of the car object in question. The latter concepts are linked via lines of object identity (explained below), but this is not shown in the LF above. What *is* shown is that the formal arguments are linked to the desired pre and post states of the car object in question.

In Ellis's formalism, concepts "…can only refer to one object, but an object identifier can have many conceptualisations", such as particular pre and post states. He uses dotted lines to show concept identity (coreference), and dashed lines for object identity [12]. In the former case, graphs can be joined since the

[12] This conflicts with the LF and DF notation of [Sowa 1999] in which coreference links are represented by dashed lines.

concepts are identical, whereas this may not be true of objects since they are contexts containing descriptor graphs that may differ in arbitrary ways. For example, before the `StartEngine` message is sent, an object may contain a relation such as `[Engine] -> (stat) -> ¬[Running]` whereas the post-`StartEngine` object would contain `[Engine] -> (stat) -> [Running]`. Ellis suggests that such objects represent different states of the same `Car`.

In Sowa's original example, a message such as:

```
[StartEngine: [TurnOn] -> (ptnt) -> [Ignition]]
```

is imported into the car instance's context to invoke the `StartEngine` method. In [Ellis 1995], this message becomes an event:

```
[Event: [TurnOn] -> (ptnt) -> [Ignition]]
```

which corresponds to the `Event` concept in the core graph of the `StartEngine` method defined above, which essentially says that some car state is succeeded by some event which is succeeded by some other car state. The complete transition is overlaid onto a CG implication, `¬[p ¬[q]]`, where `[Car] -> (succ) -> [Event]` is the value of p while `(succ) -> [Car]` is the value of q. So, essentially the state transition mechanism itself *is* an implication and it is this which constitutes the dynamic invocation of the method. The method's formal `Car` arguments are linked via lines of identity to the actual car object instances.

[Ellis 1995] makes the observation that it ought to be possible to change the formalism such that any description graphs that are not affected by a method could be omitted in that method's state pre and post objects. In his words:

> The post-state of these subobjects is assumed to be identical to the pre-state: that is, every subobject of the Car state has a line of identity between the occurrence in the pre-state and the post-state.

This is a very similar notion to that suggested in [Delugach 1991] regarding the fate of attached relations after a demon invocation. Ellis does not cite Delugach however so it is difficult to ascertain whether his mechanism was influenced by demons [13].

[Ellis 1995] outlines the promised 4 step proof for car ignition, showing a "movie" of the CG changes for each step. The first two steps are the method invocation and implication (modus ponens) as described above. The third step utilises a `TimeIncrement` relation attached via two `Time` concepts and `Ptim` relations to each car state concept. Ellis says that relation expansion is used here, but does not detail this, referring the reader to another paper for the details. The graph is similar to the following:

```
[Car] -> (Ptim) -> [Time] -> (TimeIncrement) –
                               <- [Interval: @5sec]
                               -> [Time] <- (Ptim) <- [Car]
```

It is not difficult to see how the `[TimeIncrement]` relation could be replaced by an actor which would modify the referent of the second `[Time]` concept. The extent to which this step in Ellis's proof process is generalisable to all problems is an interesting question. Without further analysis of the relation expansion mechanism, this cannot be answered.

The fourth step of Ellis's proof is to erase the original `Car` state object which existed before the method invocation.

Ellis shows how it would be possible to compose messages by "…joining the post-states of messages to the pre-states of other messages, thus "pipelining" the state information in the transition." [Ellis 1995]

[13] This is, of course, not intended to suggest some notion of *deus ex machina.*

While Ellis's proof is apparently simpler than Sowa's, the latter *might* counter that his proof is grounded in the canonical formation rules and rules of inference of CST — albeit extended to cater for the OO semantics — and that Ellis is trying to take a shortcut. Further analysis is beyond the scope of this thesis.

It seems that the primary motivation for OO CGs in [Ellis 1995] is to facilitate state transition systems, rather than developing OO systems for their own sake. Ellis seems to be focussed on objects as a means to an end, except for his introductory remarks about OO programming. Mineau acknowledges the influence of Ellis's state transition based representation on his process formalism. [Mineau 1998]

A criticism regarding use of the term object-oriented is perhaps in order here. Many formalisms have claimed to be object-oriented over the years, but many do not satisfy all the criteria, such as inheritance and the attendant non ad hoc polymorphism. Many of these *might* be characterised as object-based however [Wegner 1987]. So, claims of OO CG implementations must be carefully examined.

Indeed, misconceptions abound in the CG literature. In [Lukose 1993] one finds the statement: "There seems to be a one-to-one relationship between the notion of *class* in objected-oriented formalism and the notion of *type* in the conceptual graph formalism." Again, in [Sowa 2000] one finds: "There is a close correspondence between the hierarchy of theories in the ontology of Figure 2.14 and the hierarchy of types or classes in O-O systems", where Figure 2.14 is a type lattice showing "a generalisation hierarchy of theories". If one considers types as differentia graphs (lambda) in the type hierarchy, there is *some* similarity in terms of attributes. [Ellis 1995] points out that `[Car]` inherits the characteristics of `[Automobile]`, which may then be further specialised, so some form of inheritance does seem possible. But what of methods and data hiding? [Lukose 1993] cites a study which maps ideas such as class and instance variables onto a single construct in CGs — concepts — resulting in a loss of information regarding accessibility.

The dataflow graphs of [Sowa 2000] could be argued to have some analogue with class and object since the defining graph (class) is copied (instance or object) and concept referents modified (instance variables) for each actor invocation. A dataflow graph also contains an analogue of class or object methods in the form of actor nodes (executors) which map to functions or other dataflow graphs. On the other hand, one could adopt an imperative view instead. On this view, the defining graph is a like a function definition, its copy is a like a function invocation, referents are like function parameters, and executor nodes are like other functions. In neither the OO or imperative analogy is there a complete fit —for example, one does not copy a function in order to execute it — which highlights the danger of reading too much into analogies with other formalisms.

[Sowa 2000] presents a good case for contexts as vehicles for encapsulation, and [Lukose 1997b] argues that nested problem maps provide information hiding. Nevertheless, as has just been emphasised, to argue for too much equivalence between CGs and OO languages may at the very least be forcing the issue somewhat.

In [Ellis 1995], Sowa's example involves the creation of a process, which seems to bear little resemblance to Mineau's processes since it is based upon double-negation as implication, but given that [Mineau 1998] mentions OO CGs, it is possible that he borrowed the term from Sowa.

However, neither Ellis's nor Sowa's realisations of state transition systems appears to be simpler than Mineau's or Delugach's.

2.9 Petri Nets

Petri nets generalise STDs and were developed by Carl Adam Petri in 1962 for representing concurrent processes. STDs consist of states and transitions between states. In Petri nets, states are called *places*, while arcs between places are bisected by bars known as *transitions*, and represent events permitting a transition between states. Transitions may optionally be labelled with events to indicate the trigger for the transition. Figure 2-5 shows a simple Petri net, and is very similar to Figure 4.5 in [Sowa 2000]. Sowa believes that Petri nets are well suited to representing cause and effect, and remarks that systems which execute Petri nets on a computer (e.g. see [Mortensen 2000] for such tools) permit the simulation of processes and causal dependencies. [Sowa 2000]

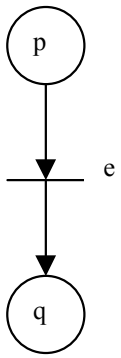

Figure 2-5 A simple Petri net with two places *p* and *q* and a transition *e*.

[Sowa 2000] discusses the mapping of Petri nets to predicate logic and CGs. Figure 2-5 can be mapped to a CG such as this:

```
[State: p] -> (Next) -> [Event: e] -> (Next) -> [State: q]
```

where e, p, and q are descriptor graphs detailing the nature of the event and states.

In terms of the language used in the section 'Introduction to Dynamic Conceptual Graphs' of this review, a Petri net is a procedure which can be instantiated to describe some process. During the course of this process, an instance of each state (e.g. p or q) is created which is represented by a *token* (a dot) drawn inside a particular place. [Sowa 2000]

[Sowa 2000] describes in some detail the flow of tokens through a Petri net. A transition becomes *enabled* when each input place has one or more tokens. An enabled transition becomes *active* when a token is removed from each input place. An active transition is *finished* when a token is moved to each output place. This constitutes a transition firing.

In a variation on Petri nets, called coloured or typed Petri nets, colours can be used to represent different types of transitions, places, and the type of data associated with tokens. Such types can be translated to CG concept types or typed predicate calculus variables. A transition may be associated with some primitive procedure specified in a programming language, or defined in terms of another Petri net. Tokens may refer to state information earlier in the history of a process. [Sowa 2000]

These characteristics of Petri nets make them equivalent to Sowa's actors (dataflow graphs). In principle, state transition based systems such as Mineau's processes and the OO state-based systems of Ellis and Sowa would also appear to be within the grasp of the Petri net formalism since events, states, memory of earlier states, and arbitrary data (referred to by tokens) are the essential features of such formalisms. By extension Delugach's demons should also be representable by Petri nets given that they are a specialisation of processes. This seems plausible when one reads Sowa's description of the erasure of tokens from input places and the addition of tokens to output places upon the firing of a transition — not dissimilar to the retraction of input concepts and assertion of output concepts when a demon fires. The Petri net formalism even caters for a token which is erased and then immediately added again to a particular place so as to provide persistence of conditions, something also permitted by Delugach's demons. [Sowa 2000] [Delugach 1991]

Of course, if the data referred to by tokens in a coloured Petri net were actually CGs, it could be argued that the Petri net formalism was simply acting as the implementation platform for some executable CG mechanism such as demons or processes. Indeed, it is possible that this *could* have been adopted as the basis for what is described in 'The Implementation of pCG'.

It is interesting to note that Sowa has considered the use of Petri nets to describe the dataflow graph (actor) mechanism in a forthcoming version of the draft ANSI standard for CGs:

> My original reason for using actors was inspired by Petri nets and the token passing mechanisms used in Quillian's semantic networks (which are a subset of the functionality of

> Petri nets). Since I have elaborated on Petri nets in Ch 4 of [Sowa 2000], I am beginning to think that the formal definition of how actors are executed could be defined to be equivalent to Petri nets. [Sowa 2000b]

In [Sowa 2000b], Sowa also remarks that Petri nets are more concise and readable than CG or predicate logic solutions. In particular [Sowa 2000] points out that Petri nets are equivalent to a special kind of logic called *linear logic*:

> …which introduces a kind of sequencing that is useful for representing processes. Although it is possible to define that same kind of sequencing in CGs or predicate calculus, the result is less readable, harder to reason about, and less efficient to compute. [Sowa 2000b]

The claims regarding readability and efficiency are subject to opinion and implementation, respectively, while the claim regarding ease of reasoning requires evidence. Certainly, the suggestion that Petri nets be used to describe the execution mechanism for dataflow graphs seems reasonable and can be easily verified by comparison against Sowa's descriptions in [Sowa 1984] and [Sowa 2000], and against actual implementations, e.g. see 'The Implementation of pCG'.

2.10 Synergy

[Kabbaj 1999a] describes a CG-based programming language (Synergy) resulting from a previous proposal by the author of an activation-based executable CG mechanism. Activation-based computation is used in domains such as simulation and visual programming. When coupled with CGs, key notions are concept lifecycle, relation propagation rules, and referent instantiation. A concept has a state and a lifecycle defined in terms of state transitions. Kabbaj also says that relation propagation rules are similar to the underlying mechanisms in dataflow graphs or Petri nets.

Development of Synergy started in 1995. It is a multi-paradigm visual programming language, supporting functional, procedural, process, reactive, OO, and concurrent OO paradigms. The language is without actors or other external notation. CGs are the fundamental knowledge structure. Kabbaj reports that some possible and actual applications are: intelligent tutoring systems and an intensive care unit visual agent-oriented model.

Synergy's components are: an IDE including CG editor, engine for dynamic formation of KBs, and an information retrieval mechanism.

The basic elements of Synergy [Kabbaj 1999a] are:

- Applications defined in terms of long and short term memory contexts. Long term memory is essentially a KB consisting of type hierarchy, CG type definitions, and individuals. Working memory stores requests in terms of CGs.

- Concepts with the syntax: `[Type: Referent =Description #State]` which while non-standard with respect to [Sowa 1999], is recognisable except for `#State`. Kabbaj asserts that this is an enhancement of Sowa's original control marks from Sowa's dataflow graphs [Sowa 1984]. State is a member of the set { steady, trigger, check-precondition, wait-precondition, wait-value, in-activation, wait-affectation, wait-end-affectation }.

- A Graphical User Interface (GUI) which permits graphs to be manipulated and applications run and observed. Concepts are represented by boxes in a similar manner to standard display form [Sowa 1999], but relations are not surrounded by circles, for conciseness, giving CGs a non-standard appearance. Colours are used to indicate whether a concept's description is simple (primitive) or defined (a graph), which state a concept is in, and so on.

Another non-standard aspect of Synergy graphs is that a concept's descriptor — Kabbaj refers to this as description — may be a primitive value, not just a graph. From the syntax above, it is possible that Kabbaj means something entirely distinct from a descriptor which is — in the standard [Sowa 1999] — part of the referent. If Kabbaj is not equating description with descriptor, he has added yet another component apart from `#State` to the formal notion of a concept.

A variety of referent types (even multi-media) and CG operations are supported. Operations are represented by concepts rather than actor nodes, and more complex operations may be built atop these primitives, in a similar manner to the way dataflow graphs can be built upon other actors. At run-time, operations map to activation records as in procedural programming. Primitive relations also exist such as: in, out, and grd (guard) which are control/data relations. The grd relation represents a precondition (C) for the execution of some other concept (C'). The sp relation indicates specialisation between concepts. [Kabbaj 1999a] The in and out relations could be replaced by simple arc directionality, except that special modifiers, mentioned below, can be applied to these relations.

Synergy permits an efficient and tidy KB organisation via CG definitions within contexts. Coreference variables can be simple identifiers or composed (e.g. `x.y.z`) for accessing embedded contexts.

The CG activation mechanism is focussed upon context activation, starting with the working memory containing a user request. Context activation corresponds to description interpretation for that context, and CG activation. Since there can be no interaction between contexts during execution, parallel activation is possible, although not supported in the implementation described in [Kabbaj 1999a]. Data and control relations propagate activation through a graph according to a set of relation propagation rules. Each concept within a graph has a lifecycle during this activation, which can be thought of as a state transition diagram. For a concept C in the graph segment: `C' - R -> C` where `R` is the relation in, out, or grd, Figure 2-6 shows the kind of state transitions that may occur.

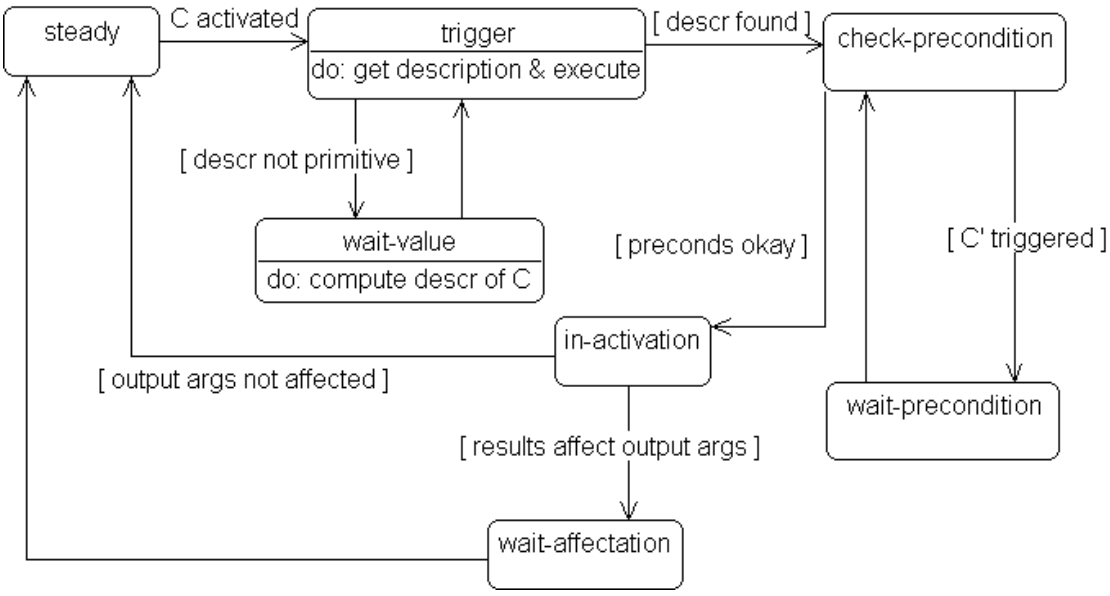

Figure 2-6 Example of state transitions in a concept lifecycle; derived from Kabbaj's descriptions

Note that in the wait-value state, C's description (e.g. a CG representing a compound operation) may be determined by backward propagation until ultimately a primitive is found. In the case of the wait-precondition state, activation of another concept may be triggered — C' in this case — before a precondition — of C here — can be checked.

Relation rules can be propagated forward or backward. Forward propagation can be cancelled by appending "/" to the relation name, while backward propagation can be cancelled by appending "\", reminiscent of Prolog's cut capability. Examples of when a relation may propagate forward are:

- If an active context has a graph `C1-R->C2` where C1.state is "trigger" and R ∈ {in, out, next} then R forward propagates the trigger state to C2.

- In `C1-R->C2`, `if` R is "grd" and C1.desc is not false then forward propagation occurs. Here, C1 is essentially a concept based precondition, although since a concept may be a context, the true or false value of C1's description might be defined in terms of a CG to be evaluated first.

One example of when backward relation propagation occurs is:

- If `C1-R->C2` and C2.state is "trigger" and R ∈ {in, out, grd } then R will backward propagate to C1, if one of a number of conditions holds, e.g. R is "grd" and C1.state is "steady" and C1.desc is undefined. In the latter case, the system will backward propagate in order to find the description for C1, ultimately resulting in execution of primitives.

[Kabbaj 1999a] gives a project schedule example in which earliest start and latest finish times for activities are calculated by two rounds of forward relation propagation through a network of nodes representing activities in a schedule. In order to do this, Kabbaj creates a definition that he calls a *process*. This is achieved by adding `[Process: super]` to the context in which the graph for the definition exists. There appears to be no other difference between this and any other kind of Synergy graph except that a process P's description is not destroyed after the first execution, so that it can be used — in this case — for a second round of forward propagation. P is also declared to be lazy by adding `[Lazy: super]` to the defining context. What this means is that P may ignore particular input concepts if they are not required for a given activation of P, which in the schedule example is true: only part of P's definition is required in each propagation round.

As can be seen, Synergy has a rich set of control constructs embodied in the concept lifecycle and relation propagation rules, but fundamentally Synergy applications appear to be sophisticated actors, only without actor relation nodes. In their place one finds concepts representing primitive or compound operations, and relations coordinating input and output concepts. One simple notation has been replaced by a fairly complex, interesting, but non-standard one. While Synergy has a notion of processes, they bear no resemblance to Mineau's. Kabbaj cites [Mineau 1998] in [Kabbaj 1999a] but makes no explicit mention of its content. Concepts in Synergy graphs may have preconditions, but they are themselves concepts (see Figure 2-6 Example of state transitions in a concept lifecycle), not graphs as in Mineau's process formalism, although as admitted above, a concept may be a context containing a graph.

[Kabbaj 1999a] gives an example of a primitive operation, SetType, which can modify the type of a concept at run-time, essentially permitting self-modifying code, resulting in greater difficulty for formal verification. This is similar to [Cyre 1998], in which the throw concept was changed to the past tense, and raises similar problems, as does the suggestion in [Delugach 1990] of powerful demons changing the type hierarchy.

[Kabbaj 1999b] describes the use of Synergy as an OO CG language, and introduces the role of methods and *daemons* in invoking functionality on such graphs. There is no relationship however between these daemons and Delugach's demons, quite apart from the fact of their different spellings. Synergy's daemons are procedural attachments to concepts which can be used to compute the value of a referent when a concept is triggered, rather like actors.

[Kabbaj 1999a] states that the original version of Synergy is implemented in Visual C++, and [Kabbaj 2000b] indicates that a Java version of Synergy is under construction, but there were no executables to download at the time of writing, so the tool could not be evaluated. Screenshots in [Kabbaj 1999a] give a good indication of Synergy's use however.

2.11 Prolog++ and Prolog+CG

[Kabbaj 2000a] describes the transition from Prolog++ to Prolog+CG, both enhanced forms of the Prolog programming language which include CGs as a data type, permitting CGs to form the basis of objects, and to be included within Horn clauses with appropriate extensions to Prolog's unification algorithm.

In Prolog++, CGs were not incorporated fully into the language, object inheritance was not implemented, generated C-Prolog code, and its portability was limited. [Kabbaj 2000a] says that Prolog++ "…has been reviewed and reformulated into a more expressive, efficient, simple, and orthogonal language." Prolog+CG is built atop a Java version of Prolog (J-Prolog) so is portable to any machine harbouring a Java 2™ Virtual Machine, implements object inheritance, and permits a CG to be in the head or tail of a rule. Kabbaj gives examples like this one:

```
goodSister(x) :- employee(x), [Woman: x]-attr->[Classical].
```

The merits of such a description of sisterhood shall not be discussed here. It is however interesting to point out that the syntax for CGs used in Prolog+CG does not conform to [Sowa 1999] or [Sowa 1984] in a number of ways, one of which can be seen in the above example in the relation "attr". Another is in the syntax of concepts:

```
[Type: Referent =Value]
```

Graph syntax appears to have been influenced by Kabbaj's earlier work on Synergy [Kabbaj 1999a].

Type hierarchies may be specified in the typical way, e.g. `Universal > Person, Action`, but this is limited to concept types and an examination of the extension to the unification algorithm reveals that graph subsumption does not include relation specialisation.

Prolog+CG has a feature called instantiation rules which allows one to specify that an individual should be mapped to a concept type, e.g.

```
Universal > Person
Person > Man

Man = Mark  // instantiation rule

[Work]-agnt->[Person: Mark]
```

This seems to correspond to Sowa's conformity relation [Sowa 1984], something currently missing in pCG. See footnote 8 in the 'Actors' section, 'The pCG Language', and 'Future Work'.

Future plans for Prolog+CG include an expert system mode, a constraint-based programming capability, support for concurrent programming, connection with database systems, and extension of the Integrated Development Environment (IDE) and graphical capabilities of the language in various ways. One other proposed extension is what Kabbaj refers to as a "knowledge base dynamic formation algorithm" [Kabbaj 2000a] which is also referenced in [Kabbaj 1999a] in connection with Synergy. [Kabbaj 2000c] refers to this under a distinct category as a tool under construction, with no further details available at the time of writing. On the surface at least, this notion seems to suggest dynamic KB update.

Why consider Prolog+CG here? It is a CG-based programming language, like pCG. It has similar fundamental data types to pCG, which is not too surprising given the need to support basic CG referents in both languages. The underlying philosophy of the two languages is quite different however. Prolog+CG is an attempt to enhance the expressiveness of a well-established and successful language (especially in AI). pCG on the other hand is a hybrid of many languages whose purpose is to give concrete form to a particular CG formalism, and because of this supports whatever else is needed by processes, e.g. functions, actors (dataflow graphs) and so on. See the chapters documenting the pCG language elsewhere in this thesis.

2.12 Other Tools

Two tools have already been mentioned in this survey: Synergy and Prolog+CG. A variety of CG tools have been created in the last several years for editing CGs, performing graph operations (eg. the CFRs and projection).

In some tools, inference engines are present, employing forward and/or backward chaining. For example: The Peirce Workbench [Ellis 1994] [Ellis 1997], The Deakin Toolset [Garner 1997] and its core [Tsui 1997], CGKEE [Lukose 1997a], CGLex [Angelova 1997], CoGITo/CoGITaNT [Genest 1998], WebKB/WebKB-GE [Pollitt 1998], CharGer [Delugach 1999], to name a few. MODEL-ECS [Lukose & Mineau 1998] is part of the Deakin Toolset.

An examination of these tools is beyond the scope of this review but it is important to acknowledge their existence so as not to suggest that the current work is without precedent, and at least, is implemented in their tradition.

2.13 Applications

The R&D work of capturing business processes described in [Gerbe´ 1998] is cited by Mineau as a motivation for the development of his process formalism. Other authors have made use of dynamic CG notions to solve particular problems or classes of problem. This section briefly considers a few of these. The emphasis in these cases is upon solution rather than formalism.

[Mann 1994] documents the development of an agent that forms the basis of a natural language understanding navigation system intended for traversing maps of the physical world. An agent prepares an action plan based upon "observation" of a simulated physical world by a simulated mobile robot. This plan is acted upon to effect "motion" within this world. Simulated sensory inputs and effectors are included. Natural language sentences such as "Go east on this road" — with perhaps additions relating to stopping when a building is reached — are translated to CGs such as:

```
[Go] -
      (Agnt) -> [Robot: #Self]
      (Inst) -> [Road: #3]
      (Axis) -> [East]
      (Dest) -> [Building]
```

[Mann 1994] says that this is "…a case frame organised around the act [Go]" and that the "…[Go] act implies physical movement in the world." A KB contains assertions regarding the robot's whereabouts and so on, e.g.

```
[State: [Robot: #Self] -> (Loc) -> [Fountain: #1]]
```

The [Go] concept must have an attached action *demon*, in Mann's system, a LISP function called TRAVEL-FN. This demon is added to the above action graph via what Mann calls a "decomposition" relation:

```
[Go] -
      (Agnt) -> [Robot: #Self]
      (Inst) -> [Road: #3]
      (Axis) -> [East]
      (Dest) -> [Building]
      (Decn) -> [Demon: #TRAVEL-FN] -
                        (Arg) -> [Robot: #Self]
                        (Arg) -> [Road: #3]
                        (Arg) -> [Building]
```

Note that these demons bear no resemblance to Delugach's, being more like actors.

The action graph when successfully invoked deletes the assertion of the robot being at its original position, and asserts the fact of the robot's new location, resulting in KB update. Relations to handle this assertion and retraction are also added to the action graph:

```
[Go] -
    (Agnt) -> [Robot: #Self]
    (Inst) -> [Road: #3]
    (Axis) -> [East]
    (Dest) -> [Building]
    (Decn) -> [Demon: #TRAVEL-FN] -
                      (Arg) -> [Robot: #Self]
                      (Arg) -> [Road: #3]
                      (Arg) -> [Building]
    (Adds) -> [State: [Robot: #Self] -> (Loc) -> [Building]]
    (Dels) -> [State: [Robot: #Self] -> (Loc) -> [Fountain: #1]]
```

[Mann 1994] also documents the use of preconditions, for example to check whether a pathway is available before attempting to move. To this end, a preconditions relation is added to the action graph, where the argument is a context containing a conjunction of preconditions. Mann also indicates that the

projection operation is used for graph matching. Finally, Mann says that all concepts of type `Act` must have available to them an appropriate demon. MODEL-ECS is not cited by [Mann 1994], so presumably, this `Act` concept type is unrelated to Lukose's. It can be inferred from the context that `Act` is the supertype of concepts such as `Go`.

In essence, [Mann 1994] describes a CG system which makes use of state-transition and an actor-like facility for invoking primitive functions. There are similarities to Mineau's formalism, which appeared four years after Mann's work, although Mann is not cited in [Mineau 1998].

[Nenkova 1999] describes a work in progress which attempts to do user modelling for the purpose of natural language explanation generation. Explanations are dynamically tailored, based upon knowledge of previous user requests in a domain. For example, if a particular explanation has been generated three times in a session, the same level of detail should not be required in future responses to similar queries. The user model and the KB are combined, permitting a single inference engine to process all graphs. The collection of information about assumed user expertise level based upon previous queries and responses results in modification of information retrieval patterns. These patterns are CGs which are projected onto the KB of graphs in order to retrieve responses.

The differences between Sowa style actors and Mineau's processes are acknowledged, i.e. that actors rely upon procedural attachments to do computation while processes perform assertions and retractions. The authors have a notion of assertion and retraction events, *from*-graphs (preconditions), and consequent graphs. Coreference links between these graph groups provides access to individuals. An *actor* is considered in [Nenkova 1999] to be a relation between two states that is triggered when certain conditions are satisfied, with a resulting consequent graph.

For example, if questions which would extract information corresponding to particular relations have been invoked too many times, the extraction pattern graphs containing those relations are retracted from the list of projection graphs. "The retract operators are actors, activated when predefined conditions about the query sequence within a single session are satisfied." [Nenkova 1999]. The authors also say that an actor asserts the results of queries in the user's KB. An example of such an actor called `<<ASS_FAM>>` is given, and is passed graphs in contexts (albeit type-less) as parameters in a manner not unlike parameters are passed to Mineau's processes.

What is puzzling about [Nenkova 1999] is not the state-based mechanism, but that the generic term *actor* is used here. [Lukose & Mineau 1998] initially takes this approach, referring for example, to a process as just a more powerful actor. Although a special node for actors vs processes or demons is of debatable value (see 'Special nodes vs direct execution of CGs'), the combination of the double-diamond notation and the term actor here is confusing, given that the special notation in the literature is specific to demons and processes.

[Bos 1997] is concerned with the representation of stereotyped human behaviours, for example, following an emergency procedure. Such behaviours involve plans associated with a particular role, consisting of ordered actions. The authors believe that the development and refinement of behavioural models is preferable to traditional knowledge engineering which suffers from two problems:

1. Difficulty in having an expert describe behaviours completely.
2. Semi-formality which permits an implementor to take too many liberties.

The proposed system of [Bos 1997] is reported to be built atop the CoGITo platform [Genest 1998], and defines a formal language based upon CGs for the modelling of human behaviour. The approach to this representation is claimed to be similar to Lukose's MODEL-ECS. One difference is that MODEL-ECS models reasoning processes, while [Bos 1997] models acting (behavioural) processes. Actually, it appears to be simpler than MODEL-ECS.

The system provides an ontology of types for modelling domain-independent human behaviour, and it is intended that this should be understandable by an expert of any domain. Bos et al believe that the use of CGs will be appealing to experts who will be able to draw them, given a suitable tool [Bos 1997]. The ease with which people unfamiliar with CGs will be able to work with them is far from certain however, since to understand the correct use of CGs, including a disciplined use of an ontology requires some learning.

In [Bos 1997], behaviour associated with a role must only be applied if certain conditions are satisfied. For example, when moving rooms, one must first be *in* the originating room. Such conditions can be expressed in terms of *states* and *events*. A state is defined as being a world property which is true during a particular temporal period. Such a graph is nested in a `State` context. Events are described as being "punctual" changes in the world. Once the conditions for a behaviour are satisfied, the action associated with that behaviour can be applied via transition from a *pre-state* to a *post-state*. For example, consider a move from room A to room B. For movement from room A, the pre-state is:

```
[Person] -> (In) -> [Room: A]
```

and the post-state is:

```
¬[Person] -> (In) -> [Room: A]
```

each of which appears in a `State` context. For movement into room B, the pre-state is:

```
¬[Person] -> (In) -> [Room: B]
```

and the post-state is:

```
[Person] -> (In) -> [Room: B]
```

The progress from one room to another also has an implicit temporal aspect represented by the two transitions: out of one room and into another, the latter succeeding the former. The CG projection operation is used in testing conditions.

[Bos 1997] suggests that a particular ordering of "instantaneous descriptions" of the world, or *situation-points*, represents one possible world evolution. Such a sequence results from the repeated matching of certain preconditions and the consequent update of the current situation-point with the associated action.

The main reason for including [Bos 1997] in this review is that [Mineau 1998] acknowledges its influence with respect to the use of contexts (e.g. `State`), pre and post-conditions. [Bos 1997] also discusses the fact that projection may result in instantiation of generic concepts and that the resulting concepts may be used in, for example, an action's graphs via coreference links. This idea is also important in [Mineau 1998]'s examples, and consequently in pCG.

2.14 Special nodes vs direct execution of CGs

From the foregoing, it can be seen that some authors believe that CGs should be executed without embellishment (e.g. Sowa, Cyre, Kabbaj). Here are some reasons why this stance is unnecessary:

- An actor node clearly indicates that a relation is present which will in some way, modify the attached arguments, or cause some modification to a KB. This may aid understanding of a graph.

- The actor node is a standard part of CST, introduced by [Sowa 1984] and reinforced in [Sowa 2000] and elsewhere. The ANSI standard [Sowa 1999] includes this node as a part of the CGIF standard, making CGs containing actor nodes parsable by conformant tools. The specification of actors in [Sowa 1999] will be augmented with a semantic specification based upon Petri nets, as mentioned in the 'Petri Nets' section.

- There is no need to introduce a new node for demons or processes. The standard actor node will do. The interpreting system can determine *from the context* exactly what kind of active element or executor is represented by the node, and then act accordingly. This has the advantage that standard CGIF can be used to represent actors, demons, and processes. pCG has done exactly this for actors and processes. See 'The pCG Language'.

Of course, one could argue that if the same special node can be used for actors, demons, and processes, why not just use a normal relation node and still rely upon the system to figure out the active elements

in a graph from the relation type? While a valid criticism, this author believes that the above points justify the use of the special node, especially the first in relation to this matter. Moreover, the CGIF representation of actors [Sowa 1999] enforces separation of input from output parameters, something that is required for actors, demons, and processes, and which strengthens the case for use of the standard notation.

2.15 Concluding Remarks

If one thing is clear, it is that even in a specialised domain such as that represented by the conceptual graphs community, imprecise and conflicting use of terminology abounds. A primary task of philosophy is conceptual analysis, which endeavours to understand how language is used within a group, to find common understandings in order to resolve conflict. It is ironic that a research community whose work frequently involves fundamental aspects of ontologies, has permitted such a Tower of Babel to develop. The foregoing survey has shown that the number of denotations of terms such as process, demon, and actor is not small. The use of similar words with different spellings (e.g. daemon) only contributes to the problem. While people sometimes do not agree with his choices, Sowa has at least attempted to keep the CG community on track via his books ([Sowa 1984] [Sowa 2000]), the CG mailing list, and most importantly, the establishment of an ANSI standard [Sowa 1999], and a standard ontology [Sowa 2000]. The field of dynamic CGs requires some attention however, and Sowa's proposed use of Petri nets as a formal description of actors is one step in this direction. Perhaps what is needed for dynamic CGs is something akin to what [Corbett 2000b] has done for unification: a framework within which formalisms may be compared. Formalisms could be divided into categories according to whether they involve:

- State-transition
- Knowledge Base update
- Dataflow
- Actor nodes in the Sowa tradition
- Concepts or graphs as arguments

and so on.

A recurring theme is the apparent utility of state transition based systems, of which [Mineau 1998] is an example. Mineau's claim is that his process formalism minimally extends CST. When compared with some of the formalisms considered in this review, the claim seems reasonable. It is clear however, that Mineau has not been operating in vacuo, that there exists a rich body of literature upon which he has been able to draw in developing his formalism.

Essentially, [Mineau 1998] 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 using post-condition graphs. This is repeated for as many times as there are precondition matches. First rule preconditions, and final rule post-conditions may be parameterised.

But is this the complete picture? While a minimalistic implementation could be developed, its utility might be limited. [Mineau 1998] hints at this with the mention of a CPE. This accords with Mineau's more recent comment that:

> The implementation layer of a CG system should be devoted attention from the CG community if a large scale CG tool platform is to be developed, which would help disseminate the theory. [Mineau 2000]

The current work is an attempt to develop just such a platform, although by no means the first attempt to produce a generic CG development environment, as acknowledged in the 'Other Tools' section of this review.

3 Objectives

3.1 The Gap

In e-mail correspondence with Mineau (1999b) it was confirmed that there has been no implementation of the process mechanism described in [Mineau 1998] and [Lukose & Mineau 1998]. Mineau only showed an example of process definition, and discussed part of its execution. In his words, "…there is only a theory which needs to be refined, implemented and applied." That is exactly what the current work entails.

A major motivation for any implementation of the process mechanism therefore is to prove whether or not 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.
> [Delugach 2000]

This is certainly true for the work carried out in this thesis.

3.2 Purpose

Accordingly, the primary purpose of the current work is to refine Mineau's theory and provide an implementation of Mineau's dynamic CG-based KB update mechanism. This will be achieved via a concrete implementation of processes as described in [Mineau 1998] which extends CST in a minimal way.

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 1998] provides the beginnings of just such a language, 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) and an underlying framework or library in a popular language, such as C++ or Java. But which? 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 and an interpreter for it be developed.

See 'The pCG Language' and 'The Implementation of pCG' for more detail regarding the interpreter, including its design goals.

A secondary — but important — purpose of the current work is to apply the developed interpreter to some problems where dynamic knowledge base update is required. At the very least, Mineau's example factorial process [Mineau 1998] [Lukose & Mineau 1998] should be tested to ensure that the basic process mechanism works. More complex problems should then be implemented, e.g. Sisyphus-I [Linster 1999] to discover what limitations the mechanism has. See 'Experiments' for implemented examples.

3.3 Significance

The value of the current work is that consideration of Mineau's processes need no longer be a paper-based exercise. Instead, processes may be constructed, then executed (see 'The pCG Language'), allowing the limitations and strengths of the mechanism to be explored.

Given other executable CG systems and appropriate tools — see 'Literature Review', 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?

The pCG language is compliant with CGIF, representing process and actor invocation graphs using a single (standard) actor node rather than the special node suggested by [Mineau 1998] and [Delugach 1991]. Also, pCG can interoperate with other [Sowa 1999] compliant CG tools such as Delugach's CharGer graph editor with which complex graphs can be drawn and saved as CGIF, a format that is readable by pCG programs.

In addition to solving problems directly in the pCG language, it can also be treated as a target language for more complex systems, a requirement for Mineau's automatic translation idea.

4 The pCG Language

4.1 Introduction

This chapter describes in detail the pCG language and its interpreter. It details the language in terms of its design goals, informal semantics, and major features, in order to make the reader familiar enough with it to be able to understand pCG programs in the 'Experiments' chapter and appendices. The documentation for actors and processes occurs near the end of this chapter after the essentials of pCG have been dealt with, much of which is supporting infrastructure.

After reading this chapter, or even before completing it, the reader may wish to skip to the 'Experiments' chapter for more substantial examples of working with pCG, before delving into more detail in 'The Implementation of pCG'. The content of the various tables in this chapter may be skimmed on a first reading and consulted later for reference.

The distribution (see 'Availability and Requirements') contains additional documentation such as JavaDocs[14] for pCG's run-time class library, and Extended Backus-Naur Form (EBNF) for pCG's grammar.

The primary motivation for the development of pCG is as a means to express Mineau's process formalism [Mineau 1998] in a concrete form. Accordingly, the abbreviation, *pCG*, is meant to indicate a *CG* being passed to or operated upon by some process *p*. The language is also this author's interpretation of Mineau's CPE.

4.2 Availability and Requirements

The pCG interpreter and its source code are available as a ZIP archive from http://www.adelaide.net.au/~dbenn/Masters/index.html with the executables and source code being redistributable under the GNU Public Licence. Simply unzip the downloaded archive and view the README file for usage information.

Java ™ 2 (in particular JDK or JRE 1.2.2 or higher) is required to run pCG. Earlier versions of Java will not suffice as pCG uses the Collections component of the Java Foundation Classes. Although pCG was developed under Linux, it can in principle be executed on any machine with a Java 2 run-time environment. It has been used by the author under Red Hat Linux 6.0 and Windows 98.

The pCG interpreter as currently implemented is invoked from the command-line, interprets a single source file, and has no interactive mode. pCG is not yet production quality software, but is quite usable. See also 'Future Work'.

4.3 Design Goals and Influences

One ought to approach the design of a programming language with some trepidation as it is fraught with dangers [15]. One strategy to mitigate the risk of failure is to decouple syntax from semantics to the extent possible, so that the nature of the source language can change easily if desired. The focus then shifts to getting the semantics right. The means by which this decoupling has been achieved shall be discussed in 'The Implementation of 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 (e.g. actor type definition, process statement) which will take the form of new syntax, and that significant new types must be introduced (e.g. concept, graph), an existing language would require significant extension. 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 (see 'User-defined Types'), utility code written in Java can be used.

[14] HTML documentation automatically extracted from Java source code.
[15] Other adventures in programming language design and implementation by the author can be found at http://www.adelaide.net.au/~dbenn/docs/projects.html

The author agrees with the following intuitions regarding language design, particularly the second:

> [s]mall languages tend to be better designed than large ones, showing fewer signs of ad hoc compromise between conflicting aesthetic principles and design goals.
>
> (Stone 1993), cited in [Gabriel 1996]

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today. [16]
>
> (Clinger and Rees 1993), cited in [Gabriel 1996]

There is a distinct tension between purity and utility insofar as language design is concerned. Very little has been documented regarding the Conceptual Programming Environment in [Mineau 1998] or elsewhere, except what has already been alluded to in the 'Literature Review' and 'Objectives' chapters. The following design goals have guided the development of pCG:

- Making those entities of primary interest to the developer, first class.
- Easy extensibility.
- Rapid development.
- Portability.

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. All types may be passed as parameters to or returned from functions.

The second is made possible by exposing the run-time system as a set of Java classes to which attributes and operations (see the next section) 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. See 'Objects', 'Built-in Types', and 'User-defined Types' for more details.

The third goal is realised by:

- The interpretive nature of pCG. 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. In the same way as Perl's key features such as regular expressions come in the form of pre-compiled library code, pCG's graph, actor, and process features are handled by pre-compiled Java classes. What this amounts to is that pCG programs can be executed with acceptable speed without the need for compilation, removing "compile" from the edit-compile-run cycle.

- The fact that all memory allocation in pCG is garbage collected[17] means that the programmer does not need to worry about chasing memory leaks.

- 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 how to represent it. This is a consequence of the first design goal.

'The Implementation of pCG' describes how pCG achieves the fourth goal of portability by virtue of its basis in Java.

[16] Given its beauty and utility, Scheme could have been used as the basis for pCG, but not everyone is comfortable with Lisp dialects.
[17] This is courtesy of the Java implementation.

pCG has been influenced by such languages as:

- Perl: first-class strings, dynamic arrays, the `foreach`, `system`, and `die` commands, but not by its purity since it has none;
- OO languages such as Java: objects and class libraries, the `instanceof` and other operators;
- Lisp: lists, dynamic typing, first class functions (lambda), `apply`;
- Prolog: assert, retract, variable binding as a result of unification;
- NewtonScript: Algol-like syntax combined with dynamic typing, *self*.

pCG does not make the statement that CGs are the best way to represent all aspects of a problem, instead permitting a hybrid approach based upon more familiar programming paradigms, not enforcing any particular one. Sowa has this to say regarding logic and procedural programming [Sowa 2000]:

> …logic can represent the same kinds of procedures as a programming language. The primary difference is that logic requires explicit relations or predicates to express the sequence, while procedural languages depend on the implicit sequence of the program listing. Ideally, programmers should use whatever notation they find easiest to read and write.

CG tools such as Synergy [Kabbaj 1999a] have been developed (see 'Literature Review') which support purely visual programming, but as Lukose remarks of his own formalism:

> …one must not be mislead into believing that the conceptual graphs and the executable conceptual structures are all good, and all encompassing, representational scheme that solves all modelling problems. [Lukose 1997b] [18]

In [Kremer 1997] a simple, general-purpose, visual flow-charting notation, Annotated Flow Chart (AFC), is mapped to MODEL-ECS in order to provide a knowledge modelling capability to non knowledge engineers in order to avoid the requirement to understand MODEL-ECS, a fairly complex language. Kremer et al show how selection and iteration constructs in AFC map to MODEL-ECS. In turn, MODEL-ECS itself implements selection and iteration constructs using CGs and special intrinsic constructs (see 'MODEL-ECS' section in the 'Literature Review' chapter).

The pCG language takes a different approach, by acknowledging that traditional programming languages and their constructs have a role to play. As a consequence, pCG has ordinary selection and iteration constructs and simple but powerful containers (lists), all of which are found in many modern languages, leaving to CGs the role of representing knowledge. If one wishes to apply a procedural operation to a KB of CGs, a `foreach` loop can be used to iterate over the set of graphs in the KB, invoking operations on each one. In short, CGs are used where knowledge representation is required, while traditional constructs are used where imperative or functional programming is most appropriate.

4.4 Informal Semantics

The pCG language is characterised by a few key semantic features. It is:

Dynamically typed: Values not variables determine type. Variables are not declared. The *is* operator can be used to determine the type of a value at run-time.

Lexically scoped: Functions and processes introduce a new lexical scope when invoked.

Object-based: 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." [Wegner 1987].

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 new arbitrary

[18] Grammatical errors have been copied verbatim.

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.

Objects in pCG have documented public *operations* (functions in objects) and *attributes*. In addition, each object "knows" its capabilities with respect to the standard in-built *operators* (`+`, `-`, `*`, `/` etc), an implementation detail which simplifies the design of the interpreter. See also 'Objects', 'Built-in Types', 'User-defined Types', and 'The Implementation of pCG'.

Minimal: The basic philosophy of pCG is that there should be few special statements and functions and that most 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.

Multi-paradigm: Apart from its object-based characteristic, pCG supports *imperative*, *functional*, and *declarative* styles of programming.

- *Imperative* programming is supported since there exists variables, assignment, operators, selection, iteration, and sequential execution.

- *Functional* programming is possible since functions are first class values and may be anonymous, closures may be created, and an `apply` operator is provided — in short, higher order functions are part of the language. Dataflow graphs (actors) may also be anonymous, and like functions anonymous actors may be recursive.

- *Declarative* programming is supported in the guise of process definitions and invocations, 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. Processes could also be seen as supporting *constraint-based* programming (as found in languages like CLIPS [Sowa 2000]) since precondition graphs essentially specify constraints on the set of graphs in the KB.

The remainder of this chapter describes specific pCG features[19].

4.5 Lexical Conventions

Identifiers and reserved words in pCG programs are case sensitive (but see the `option` directive in 'Ad hoc Statements'). This includes concept and relation type names[20]. Whitespace (tabs, newlines, spaces) is ignored by the interpreter. Single and multiple line comments are supported, for example:

```
# This is a single-line comment.

// So is this.

/*
 * This is not.
 * The second line of this comment.
 */
```

The first form (`#`) was included so that environments supporting command shells with interpreter invocation lines[21] could specify pCG as the interpreter in the first line of a program, e.g.

```
#!/home/david/bin/pCG
```

[19] pCG code snippets appear in a fixed-width `Courier` font.
[20] Whether concept and relation types should be case-insensitive is debatable. In pCG, this has been done for consistency.
[21] For example, Unix shells, third party shell implementations for Win32.

The ASCII character set is supported in comments, strings, and graphs. String literals are double-quoted. Graph and concept literals are delimited by grave accents, e.g. `` `[Baby: *a Nicholas] [Small *b] (?a?b)` ``. For the syntax of CGIF and LF graphs, see [Sowa 1999]. Identifiers may contain alphanumeric characters and underscores, but must begin with a letter or underscore. Numeric values take the form: `n[.m]`, e.g. `2`, `45.789`, but exponential notation is not currently supported [22]. Numbers are stored internally as IEEE double-precision floating-point values.

4.6 Program Structure

A pCG program consists of statements, each of which is followed by a semi-colon [23]. Functions, and selection and iteration constructs consist of statement blocks, containing zero or more statements. Such blocks also appear within the definition of processes. See the relevant sections below for specific details.

Certain statements may only appear at the top-level:

- Concept and relation type lattice declarations;
- Named function definitions;
- Lambda (of the CG variety), actor and process definitions;
- Certain option directives (`LF` [24], `TRACE`).

There is no main function in a pCG program, and code outside of functions and processes is executed in the order it appears in the text. A pCG program may currently only span a single source file[25].

4.7 Run-time Environment

Before a pCG program runs, a syntax check is performed leaving only semantic errors to be detected at run-time. Graph and concept literals are not parsed until run-time however[26]. The interpreter currently stops after encountering the first error.

Each function and process invocation introduces a new lexical scope. A process invocation also 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.

Variables are not declared to have a particular type, and a given variable may take on different types of values in various parts of the same program. The first lexical appearance determines the scope of a variable. So, if a variable is first used at the top-level (i.e. outside of a function or process definition) it will have global scope, while a variable which appears only within a function or process definition has a scope which is local to that function or process. Local variables shadow those with the same name further up the scope stack, as would be expected from a lexically scoped language. A variable that is used before being assigned a value has a default value of `undefined`.

Variables and other named entities (e.g. functions; see 'Built-in Types') are stored in a symbol table on a per scope basis. All such entities are first class, i.e. they are values that can be passed to functions, returned from functions, assigned to variables, have attributes and associated operations and/or operators. All types implement the `is` and `+` operators, the latter for string concatenation.

KBs in pCG store concept and relation type hierarchies, and a set of graphs. Look-up is confined to the KB which is top-most on the stack. The KB contents of a process caller are copied to the invoked process's KB. This prevents anything except a process's parameterised output graphs from mutating

[22] This would be trivial to add to the grammar, but a number of features have not yet been added due to time constraints. These are documented in the appropriate sections. However, see the `string.toNumber()` operation in Table 4-2.
[23] An isolated semi-colon is the *empty statement*.
[24] This option currently has no effect but is intended to specify that all CG output should be in linear form rather than CGIF.
[25] However, a Perl-like `require` directive is planned.
[26] Concept and graph parsing relies upon Notio [Southey 1999] functionality.

the caller's KB, as per [Mineau 1998] [27]. 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. See 'Experiments' and 'The Implementation of pCG' for more details.

4.8 Objects

All values in pCG are *objects* with associated *attributes*, *operations*, and *operators*. The term *value* and object will be used interchangeably. Some attributes may appear on the left hand side of an assignment statement. All may appear in expressions. All objects have an associated type attribute with a string value (itself a type) indicating the name of a value's type. For example, the following code prints the word `number`:

```
x = 42;
println x.type;
```

An operation is the equivalent of a method in Java or a member function in C++. An operator may be monadic or dyadic (unary or binary). Given a string assignment such as:

```
s = "Hello, world!"; [28]
```

the next two lines of code show an operation with two parameters and a dyadic operator, respectively:

```
s = s.substring(1,4);
s = s + "Take me to your leader.";  // string concatenation
```

Operations may be overloaded, which is to say that two or more operations may have the same name, but different formal parameter lists [29].

4.9 Built-in Types

4.9.1 Attributes and Operations

The pCG language has the following intrinsic types: number, boolean, string, list, file, concept, graph, and a special value: `undefined`. Function, lambda, actor, and process definitions also yield values with particular types. There is also a special type for Knowledge Bases, an instance of which is made available by the interpreter in the current scope. This is considered to be an internal type only. See 'Ad hoc Variables' for details.

Table 4-1 shows the attributes associated with each built-in type. An attribute name is followed by a colon, a type, and an optional "+" if the attribute is mutable, i.e. can appear on the left hand side of an assignment statement.

Type	Attributes	Comments

actor	`type: string`
	`name: string`
	`defgraph: graph`
	`sourceconcepts: list`
	`sinkconcepts: list`
		• May be "anonymous".
		• Defining graph.

boolean	`type: string`	

concept	`type: string`
	`label: string`
	`designator: number, string, boolean +`
		• pCG value type.
		• Concept type.

[27] The `option export` directive provides a way to circumvent this restriction. See 'Ad hoc Statements'.
[28] In pCG, there is only one kind of double or single quote character for strings and referents, respectively, contrary to what is displayed in this document.
[29] Currently pCG differentiates only on parameter list length, not types, but this can easily be rectified.

	`descriptor: graph +`	
file	`type: string` `kind: string`	• "reader" or "writer"

function	`type: string`
	`name: string`
	`argcount: number`
		• May be "anonymous".

graph	`type: string`
	`concepts: list`
	`relations: list`
	
	`actors: list`
		• List of concept type values.
		• List of lists of relation type label (string), input, and output argument lists. This includes actors which are a special kind of relation.
		• Same as above, but only for actor relations.

lambda	`type: string`
	`name: string`
	`defgraph: graph`
		• Defining graph.

list	`type: string` `length: number`	
number	`type: string`	
process	`type: string` `name: string`	
string	`type: string` `length: number`	

Table 4-1 Attributes for intrinsic types.

Table 4-2 shows the operations associated with each built-in type. An operation name is followed by parenthesised ordered parameter types (if any), an arrow (⇒), and a return type. The symbol "|" between return types indicates options. If "+" is appended, the object on which the operation is invoked will be mutated, i.e. its state will change. A type of `t` or `t'` indicates that any type may be passed to or returned from the operation in question.

Type	**Operation**	**Comments**
actor	`None`	
boolean	`None`	

concept	`restrict(concept)⇒boolean +`
	`copy()⇒concept`
	`nocomments()⇒concept`
	
	`isGeneric()⇒boolean`
	`isContext()⇒boolean`
		• By type and/or referent.
		• Omits comments from a concept, e.g. when created with a tool such as CharGer.
		• Has no referent?
		• Has a descriptor?

file	`readline()⇒string | undefined`
	`readall()⇒list | undefined`
	
	`readGraph()⇒graph | undefined`
	
	`write(string)`
	`writeln(string)`
	`close()`
		• Reads all lines from a file. Text files are assumed in pCG.
		• Reads a CGIF or LF graph from a file.
		• Writes a string to standard output.
		• Adds a linefeed after writing.

function	`None`	

graph	`copy()⇒graph`
	`nocomments()⇒graph`
	
	`project(graph)⇒graph | undefined +`
		• Omits comments from a CGIF graph, e.g. after creation with a tool such as CharGer.
		• Projection operator. See 'The

	`join(graph)⇒graph | undefined +`
	
	`joinAtHead(graph)⇒graph | undefined +`
		Implementation of pCG' for details.
		• Joins this graph to specified graph on first compatible concepts found in each graph.
		• Joins this graph to specified graph at the head concept (first argument of first relation) of each.

lambda	None	

list	`hasMember(t)⇒boolean`
	
	`member(t)⇒t'`
	
	`prepend(t)⇒list +`
	`append(t)⇒list +`
	`merge(t)⇒list +`
		• Looks for a value in the current list and returns true or false, but does not into recurse list members.
		• Looks for a value in this list, or in each sub-list of this list, recursively. If found, the sub-list within which it is embedded is returned. Note that this may be the outermost list.

number	`pow(number)⇒number`
	`sqrt()⇒number`
	`sin()⇒number`
	`cos()⇒number`
	`tan()⇒number`
	`floor()⇒number`
	`ceil()⇒number`
	`round()⇒number`
	`inc()⇒number +`
	`dec()⇒number +`
	`chr()⇒number`
		• Standard math functions.
		• Increment.
		• Decrement.
		• Return a string which this number represents in ASCII.

process	None	

string	`substring(number, number)⇒string | undefined`
	`substring(number)⇒string | undefined`
	
	`index(string) ⇒number`
	`toBoolean()⇒boolean`
	
	`toNumber()⇒number | undefined`
	
	`toGraph()⇒graph | undefined`
	
	`replace(string, string)⇒string +`
		• Start and end parameters are ≥ 1 and ≤ length of string.
		• Start parameter is ≥ 1 and ≤ length of string. Sub-string from start value to end of string is returned.
		• If string not found, -1 is returned.
		• Anything but string "true" (case insensitive) is considered false.
		• If the string contains a valid number, a number object will result, otherwise `undefined` will be returned [30].
		• If the string contains a valid graph, a graph object will result, otherwise undefined will be returned.
		• Replaces all occurrences of the first single character string with the second and returns the result.

Table 4-2 Operations for intrinsic types.

[30] Indeed, this is a way to sneak an exponential format number in the back door since underlying Java class library code handles this, not pCG's lexical analyser.

Operators are considered below. Note that undefined attributes evaluate to `undefined`, just as uninitialised variables do. Operation or operator failure may also give rise to the `undefined` value.

The means by which attributes, operators, and operations may be added to intrinsic objects is discussed in 'The Implementation of pCG'.

Only those attributes, operations, and operators required for the current thesis work have been implemented at the time of writing.

4.9.2 Concept and Graph Values

In pCG, concepts are existentially quantified by default, and no other explicit form of quantification is currently supported. Concept referents (see the draft CG ANSI standard [Sowa 1999]) may otherwise be:

- *Literal designators* of type number, string (single or double quoted) , or boolean[31];
- *Locator designators*, e.g. `#123658` or `#Nicholas`;
- *Variable designators*. These must currently be quoted and are distinct from CGIF defined and bound variables, for example in the CGIF graph:

  ```
  [Number: *a '*n'][Number: *b '*result']<sqr?a|?b>;
  ```

  `*a` and `*b` are defining variables, `?a` and `?b` are bound variables for identifying concepts, while `*n` and `*result` are variables representing designator values. Such variables derive from the original actor notation of [Sowa 1984] and have no basis in the proposed ANSI standard of [Sowa 1999]. The terms *variable designator,* and *referent variable* are used interchangeably in this document.

- *Descriptor graphs*, especially useful in contexts passed as process parameters.

Graphs and concepts may be specified as literals or read from a file in the CGIF or LF[32] formats. Complex graphs may be made available to pCG by using a graph editor such as CharGer [Delugach 1999] and saving them as CGIF to a file.

4.10 User-defined Types

By creating a subclass of the abstract Java class `cgp.runtime.Type`, invoking a setType() method in the new class's constructor, and optionally overriding any or all of a number of the base class's methods, a new type is made available to pCG. The class `cgp.runtime.Type` provides default behaviour for all pCG operators. The means by which this is accomplished is described in 'The Implementation of pCG'. If a new type overrides `java.lang.Object`'s `toString()` method, string concatenation is automatically available in the form: `t + s or s + t`, where `t` is the new type and `s` is a string value. One can for example say:

```
x = "This is a graph: " + g;
```

where `g` is a graph value, the result being a string value assigned to `x` with the shown string literal preceding a CGIF graph.

The distribution (see 'Availability and Requirements') contains examples of new types, e.g. `Window` and `Util` [33]. The first adds to pCG the ability to open windows, draw on a window's canvas via Turtle Graphics [Abelson 1992], and display text or arbitrary GIF or JPG images. The second class is a

[31] Boolean values (`true`, `false`) must be double-quoted strings;
[32] LF handling relies upon Notio's functionality and is currently unreliable.
[33] See the `cgp/cgp/runtime/newtypes` directory.

starting point for additional miscellaneous useful functions such as `sleep` and `random`. Both were used in a solution of the Sisyphus-I room allocation problem [Linster 1999], described in the 'Experiments' chapter. Table 4-3 details the operations in these classes. Neither contains any attributes other than the default `type` attribute. It is important to realise that once implemented, these types are no different from pCG's intrinsics except that they must be created using pCG's `new` operator [34].

Type	Operation	Comments

Util	`sleep(number)`
	
	`random(number)⇒number`
		• Sleep for the specified number of seconds or fraction thereof.
		• Return a random integer (as a number type) in the range `-n-1 ≤ 0 ≤ n-1`. This uses a random number generator which is seeded by default from the system's time.

Window	`open(string, number, number, number, number)`
	
	`close()`
	
	`setColor(list)`
	
	`drawLine(number, number, number, number)`
	
	`drawText(string, number, number)`
	
	`drawImage(string, number, number)`
	
	`moveTo(number, number)`
	
	`lineTo(number, number)`
	
	`turn(number)`
	
	`walk(number)`
		• Opens a window with the specified title, left and top coordinates, width, and height (in that order).
		• Closes the window associated with this `Window` object.
		• Sets the current window's colour using the specified red, green, and blue components.
		• Draws a line in the current window from `(x1,y1)-(x2,y2)`, using the parameters in that order.
		• Draws a string in the current window at the specified coordinates.
		• Draws an image in the current window at the specified coordinates. The first parameter is the URL of the image.
		• Turtle graphics. Move to the specified location in the current window.
		• Turtle graphics. Draw a line to the specified location in the current window.
		• Turtle graphics. Turn the turtle by the specified number of degrees (left is negative).
		• Turtle graphics. Walk the turtle in the current direction [35]. If a negative value is specified the turtle will move in reverse.

Table 4-3 Sample user-defined types supplied with the distribution.

[34] Currently, no parameter may be passed at object creation time, so the appropriate attributes must be set later if necessary.

[35] The turtle's current direction is specified by the `turn` operation, but defaults to 270 degrees, or north.

4.11 Operators

Unlike attributes and operations, the set, precedence, and associativity of operators is fixed in pCG. Table 4-4 lists these operators for built-in types from lowest to highest precedence, specifying permissible operand types, with `t` indicating any type. All pCG operators are left associative, e.g. `8-4-2` is 2 not 6. These operators are a subset of those found in Java.

Operator	Operand types	Comments
`or`	`boolean`	• Logical or. Not short-circuit.
`and`	`boolean`	• Logical and. Not short-circuit.
`> < >= <=`	`number, string`	• Relational operators.
`== !=`	`number, string, boolean, concept, list`	• More relational operators.
`is`	`t is string` [36]	• Type equivalence operator.
`+`	`number + number, string + t, t + string`	• Numeric addition and string concatenation.
`-`	`number`	• Numeric subtraction.
`* div mod`	`number`	• Numeric multiplication, division, modulus.
`-`	`number`	• Unary negation.
`not`	`boolean`	• Logical complement.
`[ ]`	`list`	• Array indexing.
`.`	`t`	• Attribute and operation access. [37]

Table 4-4 pCG operators

Although not strictly an operator, one comment needs to be made regarding assignment: it does not copy what is being assigned to the variable on its left hand side. The assigned variable merely references the object on the right hand side of the assignment operator. While some types have a `copy` operation, e.g. `concept`, `graph`, this is not intrinsic to pCG.

4.12 Selection and Iteration

The pCG language has the following basic control constructs: `if`, `while`, and `foreach`, where ellipsis represents one or more statements, and square-bracketing indicates optional parts.

```
if…then…end [else…end]

while…do…end

foreach…do…end
```

for example:

```
foreach n in {1,2,3,4,5} do
  println sqr(n);
end

foreach con in myConceptList do
  println con.designator;
end
```

[36] For intrinsic types, the type name string does not have to be quoted.
[37] Due to a grammar bug in pCG, a statement such as: `c = g.concepts[1]` must currently be written: `c = (g.concepts)[1]`.

Examples of these familiar constructs abound in the supplied examples.

4.13 Ad hoc Statements

As mentioned earlier, pCG has several statements and functions that either:

- Do not fit neatly into one of the intrinsic objects, although arguably some could be shoe-horned into one or more;
- Or, must have special access to or modify the internal state of the interpreter.

Table 4-5 describes these statements. The notation `(...)*` is a Kleene Closure, indicating zero or more occurrences of some pattern. The text "`id`" means that an identifier is expected, `expr` means that an arbitrary expression is expected. In some cases, specific types are specified for values.

Syntax	**Description**
`activate`	Actually an intrinsic function which activates a dataflow graph or process invocation graph. See also the 'Lambda and Actors' and 'Processes' sections in this chapter.
`apply function list`	Applies the function object to the list of arguments. This can also be invoked in an expression, permitting a return result to be captured.
`assert graph`	Asserts a graph in the currently active KB.
`exit [number|string]`	Terminates a program with or without a numeric or string value. The former is a result code that becomes available in the invoking shell (assuming one exists). If instead a string is supplied, it is sent to standard error before the program exits with a result code of 1. If no value is supplied, the program is terminated with a result code of zero.
`last`	Breaks out of the enclosing `while` or `foreach` loop.
`new`	This is discussed in the section 'User-defined Types', and is in fact an intrinsic function rather than a statement since it returns a new object instance.
`option id (, id)*`	An arbitrary interpreter directive. In this statement, the case of identifiers is *ignored*. The scope of an option depends upon where it occurs, e.g. at the top-level, an option's scope is global, but if associated with a process rule, the scope is limited to that rule. An option's scope may even be limited to a single post-condition graph. Five options are currently implemented, the first two at the top-level, the next two in process rules, and the final one in process rules or associated with single post-condition graphs: • `option LF`: string representation of graphs in LF not CGIF. As indicated earlier, this is currently inactive. • `option trace`: provides verbose output from the interpreter, starting with a Lisp-style syntax tree of the parsed program; useful for debugging or reporting errors to the author. • `option exportretract`: all post-condition graphs which correspond to retractions will affect the caller's KB, not the local KB. • `option exportassert`: all post-condition graphs which correspond to assertions will affect the caller's KB, not the local KB. • `option export`: all post-condition graphs will affect the caller's KB, not the local KB.
`print expr`	Prints the expression on the standard output.

`println expr`	Same as `print` but adds a newline [38].
`retract graph`	Retracts a graph from the currently active KB.
`return [expr]`	Exits functions and processes with or without a value.
`system string`	Executes an arbitrary external command and is therefore operating system dependent. This may either be invoked as a statement or as a function (part of an expression) if the exit code for the invoked process is required.

Table 4-5 Ad hoc Statements in pCG.

4.14 Ad hoc Variables

In addition to user-defined entities being entered into the symbol table for the current scope, a number of special variables are also entered by the interpreter. By convention, these special variables begin with an underscore and consist of upper-case letters. The "me" variable is an exception to this rule, because it is considered to be a special, silent parameter to each invoked function, somewhat like `this` in OO methods, but representing the function object itself. Table 4-6 details these special variables.

Variable and Type	Description
`_ARGS: list`	The command-line arguments passed to the pCG program.
`_ENV: list`	Environment variables, or more particularly, Java system properties, such as home directory, platform-specific path delimiter, and so on.

`_KB` [39]	This is inserted into the local scope of the currently executing process, and represents the local KB for the process. It evaluates to a special type with the following attributes:
	• `graphs: list`
	• `concepttypes: string`
	• `relationtypes: string`
	• `corefvars: list`
	One can either obtain these values individually, or print the value of _KB as a whole. The purpose of this variable is to provide information about updates to the currently active KB, and to provide access to its contents. One could for example, iterate over the graphs in the current KB thus, where target is some graph:
	`foreach g in _KB.graphs do`
	`  p = target.project(g);`
	`  if p != undefined then println p; end`
	`end`
	which prints successful projections of the graphs in the current KB onto a target graph.
	Every pCG KB has some default concept types such as `Number`, `String`, and `Boolean`, although they are not really necessary since such types can be added as needed in a program, and there is currently no conformity relation in pCG.
	The `corefvars` attribute is a list of bound variable designators as described in 'Concept and Graph Values' in the 'Built-in Types' section. These are bound as a result of actor activity in processes, and concept restriction (e.g. during a projection operation in a

[38] Since pCG does not yet support escape sequences such as "\n" for newline, this is necessitated. A future revision may replace `println "foo bar"` with print `"foo bar\n"`. Once again, time constraints have not permitted this at time of writing.
[39] See section 'Built-in Types' regarding type.

	process). They may be used by subsequent actors, assertions, or retractions. See 'The Implementation of pCG' for details of the algorithms [40]. Since the conformity relation is not implemented in pCG, the KB's catalog of individuals is unused.
`_MATCHES: list`	For each rule in an executing process, the list of successfully matched precondition graphs is made available via this locally scoped variable.
`me: function`	This silent parameter to every invoked function, is the function object itself. It is useful for obtaining a function's name or arity in generic error messages via the function's `name` or `argcount` attributes. It may also be used for referring to an anonymous function. See 'Functions' section.

Table 4-6 Ad hoc Variables in pCG.

4.15 Functions

Functions in pCG may be named or anonymous. The syntax of a named function definition is:

```
'function' name parameter-list block
```

An example of a named pCG function definition is:

```
function sqr(n)
  return n*n;
end
```

The definitions of functions, lambda, actors, and processes must appear before their first invocation.

Parameters may take on any value during function execution, just as variables can. A function may be invoked as a statement or as part of an expression, as in C. In the former case, any return value is ignored, e.g.

```
sqr(4);
n = sqr(4);
```

Functions may of course, be recursive. Here is the stereotypical factorial function in pCG.

```
function fact(n)
  if n < 1 then
    return 1;
  end else
    return n*fact(n-1);
  end
end
```

A pCG function may also be anonymous, e.g.

```
println apply function (n)
                if n < 1 then
                  return 1;
                end else
```

[40] The need for such a capability is a result of the assertion in [Mineau 1998] that coreference should be global in the process mechanism. That paper also suggests that the activity of a process must be cleaned up when the process exits. Accordingly, this mechanism in pCG is *local* to a process invocation, but the variables can be captured by the concepts of output parameter graphs, thus satisfying Mineau's requirement. It is worth noting that this notion of coreference is not the same as that of [Sowa 1999], but derives from the dataflow graph domain.

```
    return n*me(n-1);
  end
end       {7};
```

The difference here is that the function definition has no name following the `function` keyword, and appears as part of an expression, modifying the above syntax to:

```
'function' parameter-list block
```

This example illustrates the use of the built-in `apply` function which takes a function and list as arguments. It also shows that anonymous functions may still be recursive, courtesy of the silent `me` parameter which is passed to all functions. This idea followed the author's refinement of the actor invocation mechanism which led to recursive anonymous actors (see 'Lambda and Actors'). Given that the `me` parameter had previously been added to pCG, anonymous recursive functions were a natural outcome. Other languages that support this capability were subsequently found. For example, the Joy and R programming languages [von Thun 2000] [Bates 1997]. [Tierney 1997] shows techniques for anonymous recursive functions in a Lisp dialect, including the lambda calculus's Y combinator [Friedman 1989] [Louden 1993] [Tierney 1997]. None of these mechanisms appear to be simpler than pCG's.

As an aside, since named function definitions are entered into the top-level symbol table, the following is also possible:

```
println apply fact {7};
```

By itself this is not useful, but a function such as `map` may be written to apply a function to a sequence of values, yielding another sequence (i.e. a list). This `map` function would take a function and a list of lists as an argument, so for example, one could map the factorial function to a list of values thus:

```
myList = map(fact, {{0,1,2,3,4,5,6,7,8,9}});
```

to yield a list of factorials of the first n positive integers. Listing 1 in Appendix B shows an implementation of `map` in pCG.

Functions in pCG are also *closures* [Louden 1993] since the environment in which anonymous functions are defined is carried with the definition, e.g.

```
function mkCounter(n)
  count = n;
  return function() count=count+1; return count; end;
end

c1 = mkCounter(3);
c2 = mkCounter(10);
println c1 + " is a closure.";
println c2 + " is a closure.";
foreach i in {1,2,3,4,5} do
  println c1() + " " + c2();
end
```

This code creates two different "counter" closures, one of which has an initial value of `n` of 3, and the other of 10. Each counter function is invoked several times in order to show that they have separate copies of `n`. The following output results:

```
function anonymous; arity 0 is a closure.
function anonymous; arity 0 is a closure.
4.0 11.0
5.0 12.0
6.0 13.0
7.0 14.0
```

```
8.0 15.0
```

pCG's anonymous functions are the equivalent of `lambda` [Louden 1993] in such languages as Scheme, making functional programming possible in pCG.

4.16 Lambda and Actors

The 'Background' chapter's 'Lambda Expressions' section described Sowa's use of Alonzo Church's lambda calculus [Louden 1993] for types in CST. Recall the following example given in that section:

```
MaleBaby = [Baby: λ] -> (Chrc) -> [Gender: Male]
```

This can also be written as:

```
type MaleBaby(*x) is [Baby: ?x] -> (Chrc) -> [Gender: Male]
```

In pCG one can express this as:

```
lambda MaleBaby(x) is
             `[Baby: *a '*x'][Gender: *b "Male"](Chrc?a?b)`;
```

which has elements of the notation used in [Sowa 1999] and [Sowa 2000]. In this example, CGIF is being used to specify the defining graph. One can then say:

```
g = MaleBaby("Nicholas");
println g;
```

to yield the graph:

```
[Baby: *a "Nicholas"][Gender: *b "Male"](Chrc?a?b)
```

The essential point of similarity between the kind of lambda expression facility described in the previous section 'Functions', and the mechanism described above is that in both cases, there is β-reduction taking place which binds parameters to free variables [Louden 1993]. In pCG, `MaleBaby` is not added to the concept type lattice, making this implementation of CG lambda expressions of limited utility except as a kind of macro capability. However, it forms the basis for actors in pCG.

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;
end

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

The actor defines a graph (a dataflow graph) with an active element — actor node or executor — as discussed in the 'Actors' section of the 'Literature Review' chapter. This executor is ultimately defined in terms of a pCG function, `sqr`, which takes two concepts as parameters, a source and a sink concept. The sink concept's referent (designator) is mutated according to the square of the source concept's designator, which is first tested to ensure it is a number. 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:

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

A point of departure in pCG's implementation of actors compared to [Sowa 1984] and [Lukose & Mineau 1998] is that only the input parameter is specified in the actor definition's parameter list, and then only 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 [41]. Assuming the executor (a function or other dataflow graph) performs correctly, the returned mutated graph copy will have appropriately mutated sink concepts [42]. The defining graph in the SQR example still has a sink concept referent variable to indicate that the designator is unbound prior to actor invocation.

The above invocation code is not useful for 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 = "[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 actor node — `<sqr?a|?b>` — refers to an executor called `sqr` which takes as input a concept referred to by the bound variable `?a`, and mutates another concept, `?b`.

What if the actor is recursive? To what entity should the executor responsible for the recursive step refer? The nature of a recursive dataflow graph is that it must invoke another actor, itself in the case of direct recursion. A reasonable solution is to supply an actor definition which names the actor to be invoked. Another solution is to employ a special self-referential actor node such as `self` [43], avoiding the need for a definition. The 'Experiments' chapter gives two versions of a recursive factorial dataflow graph. The first uses an explicit actor definition, the second uses a `self` node.

Although an explicit definition works, why should recursive actors require a special case? Further consideration of this problem led the author to revisit the Y combinator. This function permits recursive functions, without explicit definition of the recursive function, by repeatedly generating the body of the next recursive invocation of the function, then invoking it for each step. Since each invocation of a dataflow graph involves copying the defining graph, before binding its source concepts and executing it, there is a distinct similarity to the Y combinator. This is more so than for recursive anonymous function invocation in pCG (or any language of which the author is aware) since there is no sense in which the body of a function is copied at invocation time.

For more substantial actor examples (recursive, multiple executors per graph), see the 'Experiments' chapter. The role of actors in process rule preconditions is also revealed in that chapter. For details of the dataflow algorithm, see 'The Implementation of pCG' chapter.

4.17 Processes

4.17.1 Definition

A process definition in pCG is similar to that proposed in [Mineau 1998]. Here is Mineau's formal definition again:

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

[41] CharGer can help with this by permitting explicit arc numbering, ensuring the correct ordering in the generated CGIF. [Mineau 1998] acknowledges the importance of arc ordering in process invocation graphs also.

[42] In order to make the mechanism more robust, it may be prudent to add an attribute to the concept type which indicates whether a concept is a sink or a source, so that the underlying function could check that it was not mutating a source concept, for example.

[43] `me` and `self` were defined independently and for different purposes, but arguably their names should be reconciled for consistency.

This translates to the following syntax in pCG:

```
'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'
```

What this essentially says is that 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 (and hopefully descriptive) identifier, optionally followed by a list of one or more options (see 'Ad hoc Statements') 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.

4.17.2 Code Blocks

An initial or action 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.

4.17.3 Pre and Post Conditions

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

4.17.4 Parameters

An input parameter must be a concept with a descriptor graph, i.e. a context, and have one of following concept types: PROPOSITION or CONDITION. The descriptor of a PROPOSITION context is asserted before process execution starts and the intention is that such a graph will act as a trigger which (along with possibly other asserted graphs) causes a suitable rule to fire. The alternative is to assert a trigger graph prior to invoking a process, but this may pollute the caller's KB unnecessarily. A CONDITION context's descriptor graph is added to the precondition list of the first rule of a process, as per [Mineau 1998]. Also in accordance with [Mineau 1998], 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 graphs in the caller's KB (rather than the local KB) depending upon whether the context's concept type is PROPOSITION or ERASURE, respectively. Arguably, this should be refined such that all final rule post-conditions modify the caller's KB, not just the output parameters, since once that rule is finished, the process will terminate, resulting in the loss of any updates to the local KB. Currently however, only the final rule's output parameters update the caller's KB, unless a directive such as `option export` (see 'Ad hoc Statements') is used.

Contexts with concept types PROPOSITION, ERASURE, and CONDITION are essentially being used as a packaging mechanism (see 'Appendix A — Contexts'), as a means by which to transport graphs to a different KB. Sowa's notion of import and export of concepts in the 'Literature Review' OO CGs section is not dissimilar.

PROPOSITION and ERASURE context types are also used in the body of post-conditions to distinguish between assertions and retractions. An alternative would have been to have each pre-condition block subdivided into assertion and retraction blocks.

4.17.5 Invocation

A process may be invoked in a similar manner to an actor. What follows is a trivial but complete process definition:

```
process p(in trigger, out finalAssertion)
  rule r1
    pre
      `[Line:*a'#1'](to_do?a)`;
    end

    post
      `[ERASURE:[Line:*a'#1'](to_do?a)]`;
      `[PROPOSITION:[Line:*a'#2'](to_do?a)]`;
    end
  end // rule r1

  rule r2
    pre
      `[Line:*a'#2'](to_do?a)`;
    end

    post
      `[ERASURE:[Line:*a'#2'](to_do?a)]`;
      `[PROPOSITION:[Line:*a'#3'](to_do?a)]`;
    end
  end // rule r3

  rule r3
    pre
      `[Line:*a'#3'](to_do?a)`;
    end

    post
    end
  end // rule r3
end
```

The following invocations of this process are equivalent.

```
// Explicit invocation. Note that while concepts are
// being passed as parameters, they are only acting as a
// vehicle (contexts) for the graphs of interest.
x = p(concept `[PROPOSITION:[Line:*a'#1'](to_do?a)]`,
      concept `[PROPOSITION:[Foo:'on you']]`);
```

and

```
// Invocation by graph activation.
g = `[PROPOSITION:*a[Line:*b'#1'](to_do?b)]
     [PROPOSITION:*c[Foo:'on you']]
     <p?a|?c>`;
x = activate g;
```

Notice that the result of process invocation is assigned to a variable in both cases. This is because it is possible for a code block in a process to contain a return statement. If this code block is reached, the process will terminate and possibly return a value to the caller. The actor node in this graph — `<p?a|?c>` — refers to a process `p` with an input parameter referred to by the bound variable `?a`, and

an output parameter, referred to by `?c` [44]. Note that the names of parameters in the formal parameter list of a process are not used anywhere within the body of the process, serving only as useful mnemonics.

4.17.6 Further Comments Regarding Preconditions, Post-conditions, and Contexts

A rule's preconditions consist of a conjunction of graphs, all of which must be matched against the local KB for the rule to fire, i.e. for the post-conditions to be acted upon.

When a graph is preceded by a tilde ("~"), it must not be possible for that graph to be matched against the local KB, if further precondition processing is to occur for the rule in question. So, the purpose of this special case of negation is not to assert a negated graph in a KB — to assert the falsity of some fact — but to reverse the sense of a graph match [45]. Such "negation as failure" is discussed in [Bos 1997].

pCG's model of KB update is a simple one of assertion and retraction, and utilises ERASURE and PROPOSITION concept types in post-condition contexts to determine which operation to apply to the KB. pCG's erasure is akin to the *erasure* Rule of Inference of [Sowa 1999] which generalises a graph to the blank graph in a positive context. Conversely, a proposition post-condition is akin to the *insertion* Rule of Inference of [Sowa 1999] which specialises the blank graph in a negative context.

The use of ERASURE seems theoretically safe, given that there are no explicit negative contexts in pCG [Sowa 1999]. pCG's use of PROPOSITION has the desired effect of asserting new facts on the sheet of assertion, i.e. in the current KB's graph set, but since the creation of a negated context is not recognised in pCG, such assertion may be theoretically questionable.

In [Esch 1994] we find the suggestion that the "…basic thing to remember about contexts and coreference is that it closely models scope of variables in block structured languages." One interpretation of a local process KB in pCG is that its graph set represents a positive outermost context. But this is an imperfect comparison, since a process KB, like a process or function scope is ephemeral and additionally, contains a copy of the caller's KB as a basis from which to begin process execution. Alternatively, one might argue that pCG's stack of KBs itself represents nested contexts.

These issues relating to assertion, retraction, and contexts in pCG and the process mechanism (as specified in [Mineau 1998]) require closer examination to determine the extent to which they are consistent with CST. See also 'Future Work'.

There are precedents for pCG's notion of assertion and retraction, in the Prolog and CLIPS languages for example [Sterling 1986] [Sowa 2000] [Giarratano 1989]. Sterling and Shapiro provide pause for caution in the use of assertion and retraction in Prolog however:

> Asserting a clause is justified, for example, if the clause already logically follows from the program. In such a case adding it will not affect the meaning of the program…Similarly, retracting a clause is justified if the clause is logically redundant. In this case retracting constitutes a kind of logical garbage collection, whose purpose is to reduce the size of the program.

While these comments probably apply to pCG, the author contends that this is a non-trivial issue, further consideration of which is beyond the scope of this thesis. See also the 'Future Work' chapter.

[44] While concept argument ordering and actual parameter context type are critical in order for formal and actual parameters to match, pCG currently ignores arc directionality. Such a check should be added for completeness. CGIF enforces that at least one output argument exist in an actor node.
[45] Note that this is currently only available in the graphs defined in rule preconditions, not in CONDITION input parameters.

4.17.7 The Process Engine

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. This is a kind of short-circuit feature, analogous to C's `&&` and `||` operators. If no matching rule is found, the process terminates. If one is found, the post-condition contexts of the matched rule are retrieved. For each post-condition, its descriptor graph is either asserted or retracted — depending upon the concept type of the context — from the local KB, or the caller's KB if one of the export options 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.

Assuming the pattern of rule firings does not lead the process into an infinite loop, and it finally reaches the last rule, the process will terminate, updating the caller's KB based upon any output parameters. See 'The Implementation of pCG' for further details of the process algorithm.

At process invocation time, the optional initial code block is executed. Before a rule's precondition matching begins, the optional block of arbitrary pCG code is executed. Another optional block is executed before every rule's post-conditions are applied. These procedural aspects of pCG are pragmatic additions to Mineau's processes.

The core process mechanism of pCG is consistent with the existential conjunctive subset of logic — with some embellishments such as precondition negation — in which only existential quantification and conjunction are required [Sowa 2000].

4.17.7.1 Implication

In the 'Background' chapter, double-negation as implication was introduced, but pCG does not currently support this. However, the precondition of a rule can be considered to be the *antecedent* of a *production rule*, while that rule's post-condition is the corresponding *consequent*, providing a form of *modus ponens*. [Sowa 2000]

4.17.7.2 Disjunction

A process's rule set is like a sophisticated case statement in the sense that when one rule fires instead of another, a choice of one from many has been made, essentially a logical *or* operation, or *disjunction*. This is contrasted with the fact that each rule's precondition constitutes a *conjunction* of graphs.

4.17.7.3 Forward Chaining

The pCG process engine is essentially a forward chaining production rule system. The difference is essentially in the richness of knowledge expression in the form of CGs. After defining a production rule as `pattern => action`, Sowa describes how a forward chaining system processes such a rule:

> When it is executed, the inference engine searches working memory for some combination of data that matches the pattern. If the match succeeds, the action on the right is executed to assert, retract, or modify facts or to call external programs that perform some computation. [Sowa 2000]

[Shinghal 1992] provides a similar description of a forward chaining production rule system. At the start of a cycle, assertions stored in a "working memory" may cause a number of production rules to become "heated" (Shinghal's word) by virtue of a match between asserted facts and rule antecedents. Only one of these is selected, on the basis of priority or specificity or some other criterion, and the consequent of the selected rule is acted upon, i.e. the rule fires. All heated rules are then "chilled". The change to working memory caused by the fired rule may cause another rule to fire on the next round.

> The practice of repeatedly heating prodrules and firing one of them is known in the literature as a recognise–act cycle/loop or as a select–execute cycle/loop. [Shinghal 1992]

The parallels with pCG are clear. pCG adds to this: CGs as patterns and the subjects of KB update, actors in preconditions, and graph export.

See the ‘Experiments’ chapter for non-trivial process examples.

5 The Implementation of pCG

This chapter describes key aspects of the implementation of the pCG language, including the interpreter, objects, and algorithms for dataflow graphs and processes.

5.1 The Interpreter

The current implementation of pCG is portable since its components are either written in Java™ or rely only upon Java-based tools:

- ANTLR [Parr 2000] [Schaps 1999], a modern alternative to lex & yacc permitting OO compiler construction, and generating Java code;
- The Notio API [Southey 1999] which implements ANSI compliant CG operations [Sowa 1999] as a Java OO framework [46];

As of the current writing, pCG consists of more than 4,500 lines of *non-commented* [47] Java and ANTLR code. If the relatively efficient ANTLR-generated Java code is substituted for the ANTLR source code, the line count jumps to almost 11,000. This gives some indication of the saving afforded by ANTLR. The parser and interpreter files are the largest, with concept and graph type run-time code the next largest.

ANTLR encourages the syntax of the language to be decoupled from the generation of the target code. A pCG source file is parsed, giving rise to Abstract Syntax Trees (AST), an efficient intermediate representation for program execution. This representation can be the target of different grammars, lessening the importance of which source language is chosen. Operator precedence and associativity is handled only by the parser which generates a particular pattern of nested ASTs. The parser is 500 lines of ANTLR code, whereas the interpreter is more than 1,800 lines long, making the task of changing pCG's source language less onerous than it would be without the aid of a tool like ANTLR.

The Java based Notio framework [Southey 1999] is used to take care of basic CG functionality, e.g. KBs, type lattices, CG I/O in CGIF or LF, and operations such as restrict and join [48]. This enables the focus to be upon the additional functionality required by actors and processes. Again, without Notio, the source line count for pCG would be considerably higher than it is.

A `makefile` is supplied as a convenience. While especially suited to the Unix environment, if the `make` command is not supported on the target operating system, the `makefile` can be used as the source of commands to be invoked in order to generate the interpreter executable. The same is true for the supplied shell script for invoking the interpreter, but a suitable Windows batch file is also supplied.

5.2 Objects

An object in pCG is represented internally as a Java subclass of `cgp.runtime.Type`, hereafter referred to as `Type`. Attributes correspond to getter and setter methods and associated private instance data in the subclass. For a subclass `S` of `Type` and an attribute `a`, there may be two instance methods on `S` (at least the first):

1. `T`[49] `getA()`
2. `void setA(T)`

The getter method is invoked by pCG when the attribute's value is required as part of an expression, and the setter method is invoked for attribute assignment. If an attribute is not mutable, the setter

[46] As has been mentioned elsewhere, the Notio based CharGer tool is used for non-trivial graph creation as input to pCG programs. [Delugach 1999]
[47] That is, with comments and blank lines stripped. The source is well commented.
[48] Notio does not handle all required CG operations, but assists greatly.
[49] `T` is some subclass of `Type`, or `Type` itself.

method will not exist. An operation on `S` corresponds to a Java method of the same name in that class. The interpreter uses the Java Reflection library to find and invoke the correct method for attributes and operations. This facility permits dynamic discovery of methods in a class, and when a pCG program specifies a particular attribute or operation, the interpreter queries the appropriate class regarding its capabilities, using Reflection. If the required method is not present, pCG generates an error. The `Type` class implements the methods necessary to provide this capability.

Adding methods with particular names to new or existing types provides automatic access to pCG's built-in operators (e.g. `+`,`-`,`*`) so long as the left hand operand's type is that of the type in question. For example, the operator "`+`" corresponds to the method `plusOp()` in `Type`. `Type` provides default method implementations for all pCG operators. If a subclass of `Type` does not override these methods, the default behaviour is invoked, which may be to generate an error or perform string concatenation, for example. Given the pCG expression:

```
x + 2
```

the following method will be invoked:

```
x.plusOp(2)
```

where `x` will be a value whose type is some subclass of `Type`, and 2 will have been converted to a value of type `cgp.runtime.NumberType`.

Regarding the signatures of the aforementioned implementation methods, a pCG type name may be mapped to its underlying implementation type, and found in the package `cgp.runtime`, by taking the former, making its first letter uppercase, and appending "`Type`". So, for example, the pCG type `string` has the corresponding implementation type: `cgp.runtime.StringType`.

The source code for the class `Type`, or its corresponding JavaDocs, should be consulted by anyone wishing to enhance pCG's existing types or to add new ones. See 'Availability and Requirements' in 'The pCG Language'.

5.3 Actors

Actors, like functions are first class values in pCG, and actor definitions are stored in the top-level scope's symbol table (since they may only be defined at the top-level). The underlying `cgp.runtime.ActorType`[50] stores the information required for invocation of an actor such as defining graph and formal parameters, if any. Anonymous actors also correspond to instances of this class but are not stored in a symbol table, and are only generated internally by the interpreter during explicit (`activate` function) or implicit (within process preconditions) dataflow graph invocation.

An implementation-oriented view of pCG actors (dataflow graphs) is that of a run-list of actor nodes or "executors" along with their concept arguments. Each executor attains the *ready* state when all its input arguments are bound. The first ready executor encountered in the run-list is chosen, removed, and executed. Assuming referential transparency, the order of execution is irrelevant. Eventually the run-list ought to be empty (see the introductory section of the 'Literature Review'), so the algorithm can exit, and the mutated graph copy can be returned.

There may be a problem from the viewpoint of dataflow if one executor in an actor blocks for any reason, since this means that some output concept will not have its referent bound to a particular value. If another executor takes this concept as input, it will in turn be blocked. If the concept in question is a sink, then the actor as a whole will have no output, unless the sink also takes output from another executor, providing a valid use for conflicting sinks [51]. This is the case for the recursive factorial example documented in the 'Experiments' chapter, such that for a particular copy (instance) of the defining graph, only one executor fires. If no executor on the run-list is ready, the actor's execution must terminate, irrespective of whether the sinks have yet been mutated, since this implies blocked executors which will never become unblocked.

[50] A subclass of `cgp.runtime.LambdaType`
[51] A conflicting concept is an output concept for more than one executor.

The foregoing accords with Sowa's description of dataflow for a recursive factorial actor, similar to the one mentioned above: "Each recursive call to Facto would cause the label Facto to be replaced by a copy of its defining graph. That new copy which is invoked to compute another value, is erased after it returns the result." [Sowa 2000]

What follows is an algorithm (pseudocode) for pCG's actor execution mechanism. Note that this is recursive. A precondition of calling this function is that the first parameter is a known dataflow graph.

```
Function actorCall(actor, argList, bindArgsByOrder) returns graph
      Copy the actor object. Call this actorCopy.

      // Parameter binding.
      If argList's length is > 0 and equal to the formal list length
      then
        If bindArgsByOrder is true then
            // Assumption: result of a recursive anonymous actor call
            // in which source concept referent variable names have
            // never been associated with an actor type definition.
            // We just assume that the actual argument list and input
            // arc ordering provide a compatible mapping. Any other
            // call from one actor to another implies the existence
            // of an actor type definition, for which the next else
            // block can be used. Note that this is different from
            // the initial activation of an anonymous actor graph
            // whose source concepts have already been bound. The
            // final block of this If statement deals with that
            // scenario.

           Bind each argList element to actorCopy's source concepts
           by source concept arc order.
        Else
              // Normal actor call in which actor is associated with
              // an actor type definition.

              1. Map each formal parameter name in actor's type
                 definition to each element of argList.

              2. Bind each mapped argList element to actorCopy's
                 source concepts by referent variable name.
        End If
      Else
            // No arguments.
            // Assume the initial activation of an anonymous actor
            // graph whose source concepts have already been bound.
            // This will be the result of explicit actor graph
            // invocation via the intrinsic pCG activate function or
            // via its implicit counterpart in process precondition
            // processing.
      End If

      // This is primarily motivated by the fact that within a
      // process some graph projections may not result in all concept
      // designators being bound, and that existing coreference
      // variables should fulfil this need. Indeed, some of
      // [Mineau 1998]'s examples rely upon such coreference.
      // pCG restricts the scope to the current knowledge base.

      Foreach source concept c in actorCopy do
            If c's designator is still not bound then
```

```
                If a suitably named referent variable v is defined
                in the current KB then
                      Bind c's designator to v's value.
                End If
          End If
    End Foreach

    Copy all of actorCopy's executors with their source and sink
    concepts to actorCopy's run-list.

    While executors remaining on run-list do
          Find the next ready executor in the run-list.

          If no executor is ready then
             // The actor is blocked.
             Break out of the loop.
          Else If executor is a function, f, then
                 // Function will mutate output concepts.
                 1. Gather (bound) input concepts and (unbound)
                    output concepts, appending latter to former as
                    a list.
                 2. Apply f to this list.
                End If
          Else
                // The executor is an actor. Invoke it with the
                // current executor's input arguments, then
                // obtain the outputs resulting from execution of
                // this actor.

                1. Gather input concept designator values as args.
                2. If executor is recursive and anonymous then
                      graph = actorCall(executor, args, true)
                   Else
                      graph = actorCall(executor, args, false)
                   End If
                3. Obtain the sink concepts from graph.
                4. Foreach sink concept do
                      Set the designator of the corresponding
                      output concept in the executor to the value
                      of this sink's designator.
                   End Foreach
          End If
    End While

    // Either the run-list is empty and all executors have been
    // invoked, or the actor is blocked because there are no
    // executors in the ready state.

    Return the possibly mutated actorCopy's graph.
End Function
```

5.4 Processes

5.4.1 Representation and Algorithm

Like actors, processes are first class values in pCG, and the remarks made about actors apply here except that the underlying process type is `cgp.runtime.ProcessType`.

While [Mineau 1998] provides a basic algorithm for process execution and details parameter evaluation, it leaves certain details untreated, for example, how actors modify the precondition

matching algorithm. This section provides an algorithm for process execution. This amounts to a refinement of what is provided in [Mineau 1998].

The formally defined contexts of [Mineau 1997] are suggested as the basis for specifying input parameters, and for representing pre and post conditions, "…providing a formal environment for using contexts as state descriptions" [Mineau 1998]. Mineau does not spell out the details or consequences of using contexts in this way, but contexts could be used to implement rules such that preconditions are represented by the intention of a context and post-conditions are represented by the extension of that context. The search operations defined in [Mineau 1997] could be used to find the matching precondition given the current state of the KB. See "Appendix A — Contexts" for an overview of contexts.

As detailed in 'The pCG Language', contexts are used for input and output process parameters, as suggested by Mineau. Contexts are not used in the current implementation for representing rules however. A simpler approach has been taken which is closely aligned with Mineau's process statement specification in which a set of rules is called for. Accordingly, rules are stored internally as a collection of `cgp.runtime.Rule` instances.

5.4.2 Projection

Recently, projection has been considered as an addition to the next version of the CG ANSI standard [Sowa 1999] [Sowa 2000a], given that it forms the basis of many practical CG systems [Mineau 2000]. The projection operation is central to the matching algorithm of pCG's process execution algorithm. The projection algorithm used by pCG is similar to that given in Theorem 3.5.4 of [Sowa 1984] except that relation type subsumption is also permitted. What follows is the algorithm for pCG's projection operation.

```
// Project the specified graph onto this graph, returning the
// projection as a new graph, or null. The second parameter can be
// seen as a filter which will be applied to the first. The
// projection will only be successful if the filter graph is
// a generalisation of the target graph, i.e. all relations and
// concepts in the target are subtypes or the same as those in
// the filter graph, and the filter's concepts are restrictable
// by type and/or referent to the target's. Note that the filter
// graph may have more or less nodes than the target.

Function projection(target, filter) returns graph
      // The filter graph must always be seen as the template for
      // the final projection since it may contain desired actors or
      // extra relations. We copy the filter as the basis of the
      // final projection graph which will be mutated below [52].

      1. Copy the filter graph F. Call this P, the projection
         starting point. This will be mutated later.
      2. Call the target relations TR.
      3. Call the filter relations FR.
      4. Call the projection relations PR.
      5. Create a list of relations of the same size as TR. Matched
         relations will be stored here. Call this MR. MR's elements
         default to null.

      // Find the relations in the filter graph which match those
      // in the target graph. If a target relation has no match,
      // the match array element defaults to null. An important
      // thing to note here is that only the relations in the target
```

[52] Arguably, this approach combines projection and join operations since new sub-graphs (e.g. actors) are sometimes essentially being added to a copy of the target after it is shown that $\exists g\ \exists h$ (sub-graph(g, filter) $\wedge$ sub-graph(h, target) $\wedge$ g $\geq_g$ h).

```
      // graph are matched against. This makes it possible to have
      // extra relations (such as actors as noted above) in the
      // filter[53].

      Foreach TRi in TR do
            Foreach FRj in FR do
                  If FRj.type ≥ TRi.type
                  and ∀c ∈ FRj.sources, ∀c' ∈ TRi.sources, c ≥ c'
                  and ∀c ∈ FRj.sinks, ∀c' ∈ TRi.sinks, c ≥ c' then
                        1. Relations match, so MRi = PRj
                        2. Increment matches.
                  End If
            End Foreach
      End Foreach

      // If we have as many matching relations as there are relation
      // nodes in the smallest graph, proceed to restrict relations
      // and concepts in the projection graph based upon the target.
      // There will be null matches in elements of MR if there are
      // more relations in the filter graph than the target graph.

      If matches ≠ 0 and matches = minimum(F.relations, T.relations)
      then
            Foreach non-null MRk in MR, TRk in TR do
                  1. If MRk.type > TRk.type then
                        Restrict MRk.type to TRk.type
                     End If
                  2. Foreach concept c in MRk, c' in TRk do
                        1. If c.type > c'.type then
                              Restrict c.type to c'.type
                        2. End If
                        3. If c.referent is blank or a variable
                           and c'.referent is bound then
                              Restrict c.referent to c'.referent
                           End If
                     End Foreach
            End Foreach
      Else
            P = null
      End If

      Return P
End Function
```

5.4.3 Process Execution Algorithm

```
Function processCall(process, actualParams) returns t [54]
      1. Get process's formal and actual parameters. Exit if not same
         length.

      2. Push a new scope.
```

[53] If a graph contains more than one compatible conceptual relation, e.g. `[Person: X] -> (Has) -> [Friend: A]` and `[Person: X] -> (Has) -> [Friend: B]`, the current implementation will not distinguish between them. To be able to do so requires concept restriction to occur at this point in the algorithm, rather than later.

[54] `t ≤ cgp.runtime.Type`

```
3. Push a new KB copying the type lattices, corefvars, and
   graphs of the caller's KB[55].

// Arguments must be contexts, i.e. concepts with descriptors.

4. Foreach arg_n in actuals list and formal_n in formals list do
      If arg_n is a context then
            If formal_n is input parameter then
                  If arg_n.type is PROPOSITION then
                        Assert arg_n.descriptor as a trigger.
                  Else If arg_n.type is CONDITION
                        Add arg_n.descriptor to first rule's
                        precondition.
                  Else
                        Error: exit process.
                  End If
            Else If formal_n is output parameter then
                  If arg_n.type is PROPOSITION then
                        Add arg_n.descriptor to final rule's
                        invocation assertion list.
                  Else If arg_n.type is ERASURE
                        Add arg_n.descriptor to final rule's
                        retraction list.
                  Else
                        Error: exit process.
                  End If
            End If
      End If
   End Foreach

   5. Execute code in process's optional initial block.

   6. Do
            Foreach rule Rule_i in process's rule list do
                  1. Execute code in optional precondition
                     block.

                     // To fire this rule and act upon the
                     // post-conditions, all preconditions must
                     // match against the local KB. If a
                     // precondition graph is negated, this
                     // reverses the sense of that graph match.
                     // Such negation is not shown here.

                  2. allMatch = true
                     Foreach precondition graph Pre_j in
                     Rule_i.preconditions do
                        If not match(Pre_j, local KB) then
                           allMatch = false
                           Exit foreach loop
                        End If
                     End Foreach
                  3. If allMatch is true then
                        1.  Execute code in optional post-
                           condition block.

                        // Assert or retract graphs in local or
                        // caller's KB, first binding referent
                        // variables in graph to be asserted or
```

[55] See 'Ad hoc Variables' section of 'The pCG Language' re: KB contents.

```
                                    // retracted using local KB's corefvar
                                    // bindings, which were set (if at all)
                                    // during precondition matching.

                                    2.  Foreach postcondition Post_j in
                                       Rule_i do
                                          If an export option is set
                                          or (Post_j is an output parameter
                                          and Rule_i is the final rule) then
                                                kbase = caller's KB
                                          Else
                                                kbase = local KB
                                          End If

                                          If Post_j is a context then
                                             If Post_j.type is PROPOSITION
                                             then
                                                Assert Post_j in kbase
                                             Else If Post_j.type is ERASURE
                                             then
                                                Retract Post_j from kbase
                                             Else
                                                // Wrong context.
                                                Error: exit process.
                                             End If
                                          Else
                                             // Expecting a context.
                                             Error: exit process.
                                          End If
                                       End Foreach

                                    3. If Rule_i is the last rule in the
                                       rule collection then exit the while
                                       loop, terminating the process.
                                 End If
                           End Foreach
                     While match occurred and Rule_i is not final rule.
End Function
```

The `processCall` function is dependent upon the `match` function, which matches a graph against the graphs in the local KB.

```
Function match(graph, kbase) returns boolean
      // Found a match?
      found = false

      // If projection is used, graph could be modified, so copy it.
      Copy the graph parameter. Call this g.

      If g represents a process invocation graph then
            // If we invoke another process, the expectation is that
            // there will be an update to the current KB.

            Activate the corresponding process.
      Else
            // Try various kinds of matches.

            1. Attempt an exact match of g for every graph g' in the
               KB in turn, i.e. try to satisfy:

                  ∃g ∃g' (g' ∈ KB ∧ g' = g)
```

```
            2. If 1. fails, attempt a projection of g onto every
               graph g' in the KB in turn, i.e. try to satisfy:

                  ∃g ∃g' (g' ∈ KB ∧ projection(g, g'))

            3. If 1. and 2. fail, but g is a dataflow graph, isolate
               each conceptual relation in g, create a graph h from
               it, and attempt a projection of h onto every graph g'
               in the KB in turn[56]., i.e. try to satisfy:

                  (∃g ∀subg
                        (subg ∈ g.relations ∧
                              (∃g'
                                    (g' ∈ KB ∧
                                     projection(subg, g')))))

            4. If 1., 2., or 3. yielded a match then
                  If g is a dataflow graph then
                        1. h = actorCall(make-actor(g), {}, false)

                        2. Check each g.sink against the
                           corresponding h.sink.
                           • If g.sink had a referent variable and
                             h.sink is now bound then the referent
                             has been computed, so found = true.

                           • If g.sink had a default literal
                             designator against which the
                             corresponding h.sink must be compared
                             and h.sink's referent has been mutated
                             and g.sink = h.sink then found = true.
                  Else
                        found = true
                  End If
               End If
      End If

      Return found
End Function
```

5.4.4 Complexity of the Process Algorithm

The highly iterative nature of the above algorithms suggests that given a candidate target graph in the KB, it is possible to check whether a precondition graph matches it in polynomial time. But the number of steps required to find a matching precondition-target pair is unknown for a given process rule set and KB state, and exponential in the worst case, making the process algorithm NP-complete [Dewdney 1993]. The search problem is not noticeable for the small KBs accompanying the process examples of the 'Experiments' chapter — a few tens of graphs — but would quickly become so for non-trivial problems. See also [Willems 1995] on the subject of projection complexity.

[56] The big assumption here is that the actor's arguments are part of single-relation graphs. If they are not, then this approach is invalid since the whole sub-graph in question (attached to an actor argument) should be projected against the KB's graphs. See the pCG implementation of Mineau's iterative factorial algorithm in the 'Experiments' chapter. Rules 7 and 8, for example, rely upon this mechanism to glue together two variable graphs for the purpose of some computation.

6 Experiments

This chapter documents experiments in the form of pCG example programs and discussion thereof. Limitations of processes and the pCG language that arise from these examples are also discussed, particularly in the last example.

The chapter starts with a simple example which demonstrates fundamental pCG features, moves on to actors, then to the iterative factorial process of [Mineau 1998], and finally to a solution of the Sisyphus-I room allocation problem using a process. Of primary interest to the subject of this thesis are the last two. Of these, the most appropriate application of the process formalism is Sisyphus-I, since pattern matching and KB update are key elements.

Throughout this chapter, references are made to code listings, all of which can be found in 'Appendix B — Source Code Listings'. Short code fragments will also appear in the body of the chapter.

Numerous other examples of pCG programs are included in the distribution. See 'Availability and Requirements' in 'The pCG Language'.

6.1 A Recursive Tree

Although not CG-related, this simple pCG program demonstrates the following features, described in 'The pCG Language':

- Interpreter specification line (assumes execution is under Unix);
- Both forms of single-line comment;
- Function definition and invocation;
- An ad hoc variable;
- List creation and element access;
- Object operation invocation;
- User-defined types;
- Coercion of a number to a string and subsequent string concatenation;
- Explicit program termination.

```
#!/home/david/cgp/pCG
#
# A recursively drawn tree using Turtle Graphics. [57]

function abs(n)
  if n < 0 then n = -n; end
  return n;
end

function tree(n, w)
  if n >= 5 then
    w.turn(30);
    w.walk(n);
    tree(n*3 div 4, w);
    w.walk(-n);
    w.turn(-60);
    w.walk(n);
    tree(n*3 div 4, w);
    w.walk(-n);
    w.turn(30);
  end
end
```

[57] The author has a penchant for Turtle Graphics [Abelson 1992] due to its power and simplicity, so sneaks it in wherever possible.

```
# Main program.
depth = (_ARGS[1]).toNumber(); // try 30
colors = {{200,0,0}, {0,200,0}, {0,0,200}}; // red, green, blue
u = new Util;
w = new Window;
w.open("Tree", 100, 100, 300, 300);
w.drawText("A tree of depth " + depth, 100, 25);
w.setColor(colors[abs(u.random(3))+1]);
tree(depth, w);
u.sleep(2);
w.close();
exit;
```

Invoking this program with a command-line argument of `30` yields Figure 6-1, one of the author's favourite class of graphical entities:

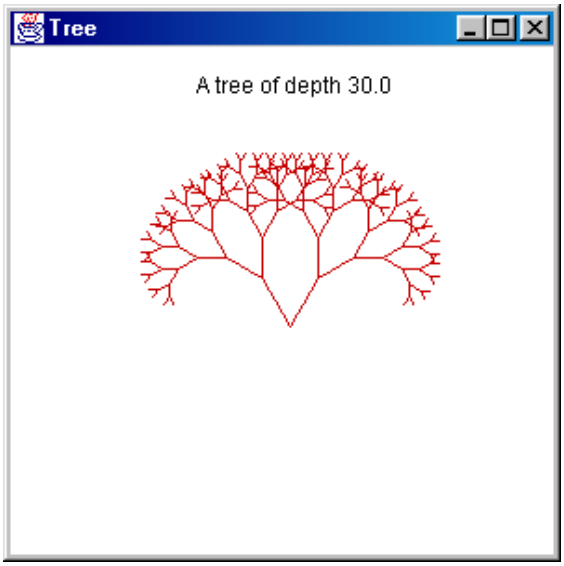

Figure 6-1 A recursively drawn tree.

6.2 Quotient and Remainder Dataflow Graph

This example is an implementation of the first actor (dataflow graph) in [Lukose & Mineau 1998]. Figure 6-2 is that actor recreated using CharGer and saved as CGIF to a file called Figure1.CGF, and is shown in Listing 2 (Appendix B). Notice the considerable non-CG information contained in the CGIF generated by CharGer, which specifies concept and actor positional information.

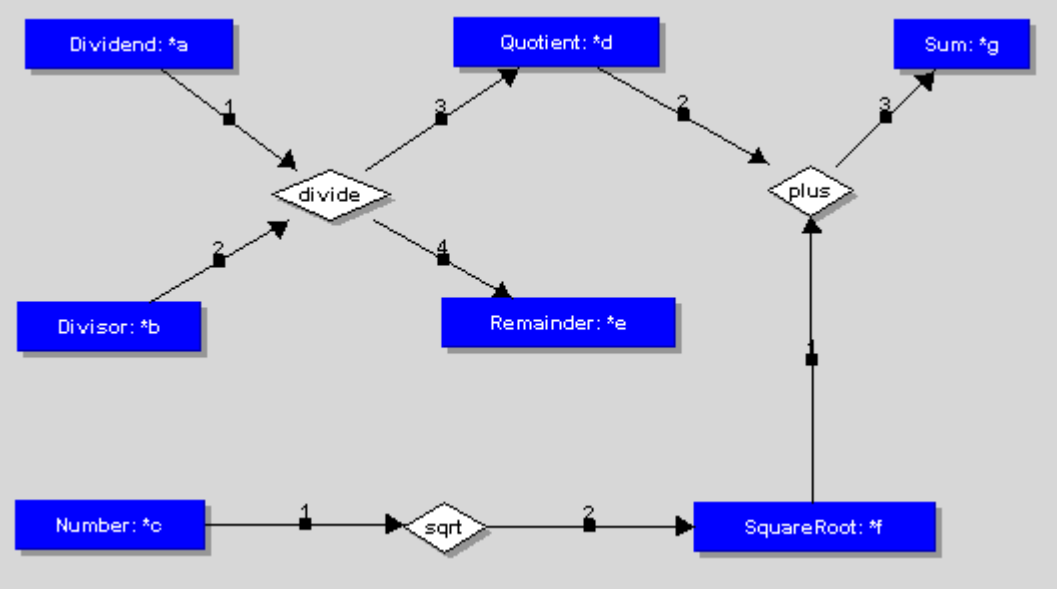

Figure 6-2 The dataflow graph of Figure 1 in [Lukose & Mineau 1998].

Listing 3 contains pCG code which invokes this actor. The first line is an interpreter specification which tells the shell that pCG is the program which will interpreter this source file. Lines 11–21 define functions corresponding to the executors of Figure 6-2. As an aside, lines 33–36 show how pCG supports platform independent file paths, here in the form of the `fs` variable:

```
# ** Examples directory **
home = _ENV.member("user.home");
fs = (_ENV.member("file.separator"))[2];
dir = home[2] + fs + "cgp" + fs + "examples" + fs;
```

The `home[2]` value may not point to the appropriate directory. Instead the value of the "`user.dir`" key (absolute path to the current directory) may be more appropriate. Lines 38 and 39 read the CGIF of Figure 6-2 from the file and associate the resultant defining graph with the actor type, `Figure1`:

```
r = file (dir + "Figure1.CGF");
actor Figure1(a,b,c) is r.readGraph();
```

Lines 49 &50 invoke an instance of the actor thus and show the result on standard output:

```
g = Figure1(9,4,144);
println g.nocomments();
```

Lines 52–56 write a CharGer readable form of the result graph to a file, the content of which is shown in Figure 6-3 as it is seen in CharGer.

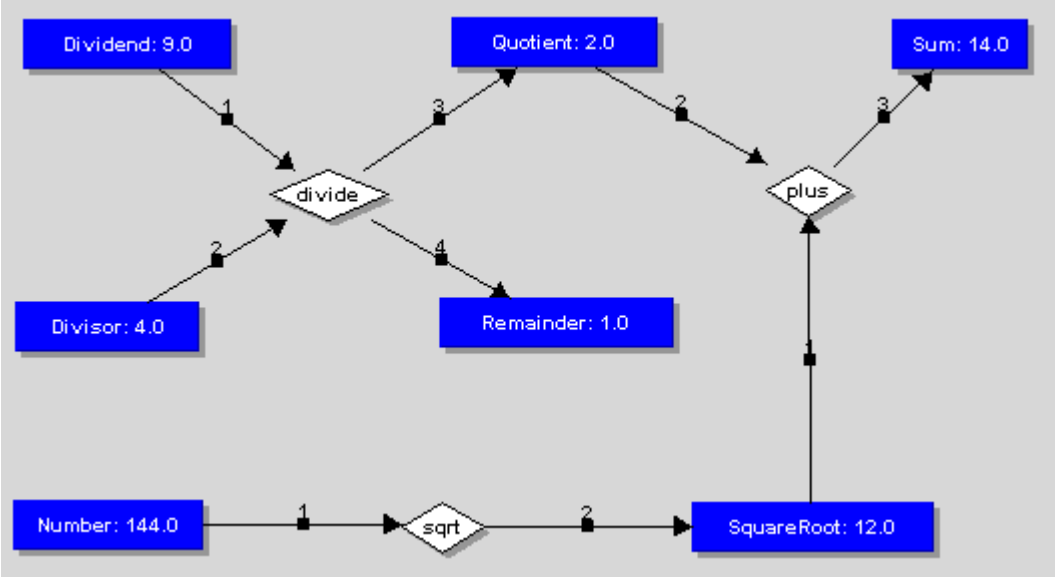

Figure 6-3 The result of executing the first actor of [Lukose & Mineau 1998].

6.3 Recursive Factorial Actor

Listing 4 shows a program that is very similar to the one of Listing 3. It defines an actor for and invokes the graph of Figure 6-4:

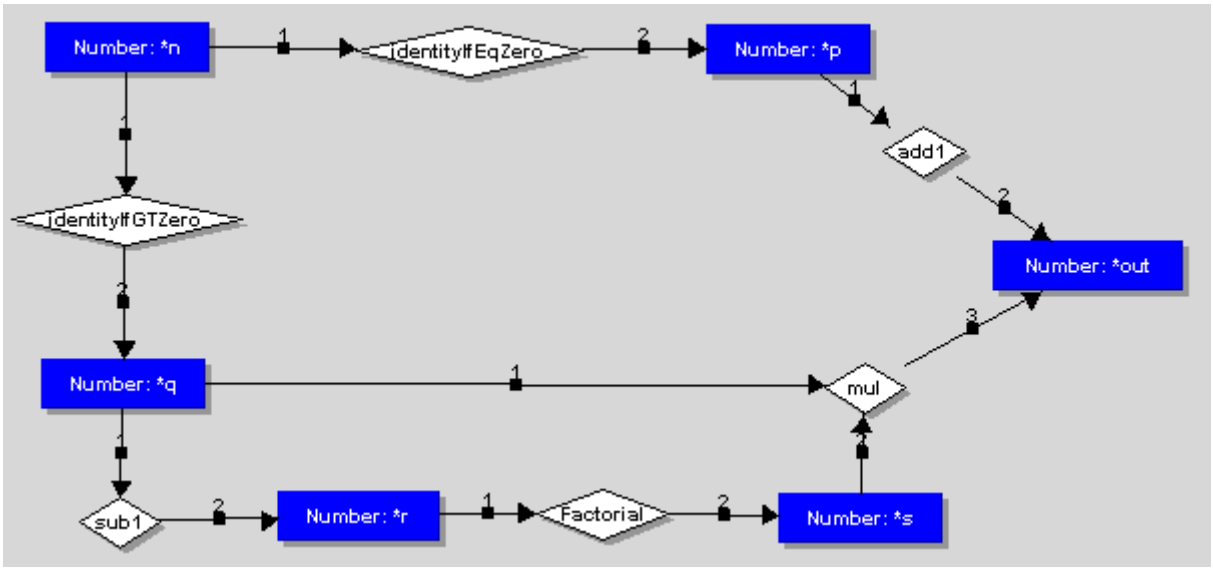

Figure 6-4 Recursive factorial actor adapted from [Lukose & Mineau 1998].

The differences between this graph and that presented by [Lukose & Mineau 1998] are that all concepts have explicit referent variables rather than being generic, and the actors `identityIfGTZero` and `identityIfEqZero` are used in place of a generic `Identity` actor and the subrange types `Number>0` and `Number=0`, respectively. [Sowa 1984] advocates the use of such subrange types, the purpose of which is to constrain the permissible referent values. A footnote in the 'Actors' section of the Literature Review chapter characterises this in terms of the conformity relation and the denotation of a type, such that for the concept type `Number>0`, δ`Number>0` = {x | x∈ℜ ∧ x>0}, where ℜ is the set of real numbers. The idea is that if an operation resulting in referent specialisation would lead to a violation of the conformity relation in a concept, e.g. `[Number>0:-1]`, and that concept is an input to another executor, the latter is said to be blocked. This may result in the whole dataflow

graph becoming blocked. In the following, a source concept's referent is passed unmodified to a sink concept, and whether the generic sink concept can be instantiated depends upon the binding of `*n`.

```
[Number: *n] -> <Ident> -> [Number>0]
```

Since pCG does not currently support the conformity relation, this construct cannot be represented. However, the same effect can be achieved procedurally (with less control) rather than declaratively, by an actor that conditionally mutates the referent of its sink concept, for example:

```
[Number: *n] -> <identityIfGTZero> -> [Number: *out]
```

If the sink concept's referent is not specialised by the `identityIfGTZero` executor, any dependent executor (e.g. `add1`) will be blocked.

The program of Listing 4 yields the graph of Figure 6-5:

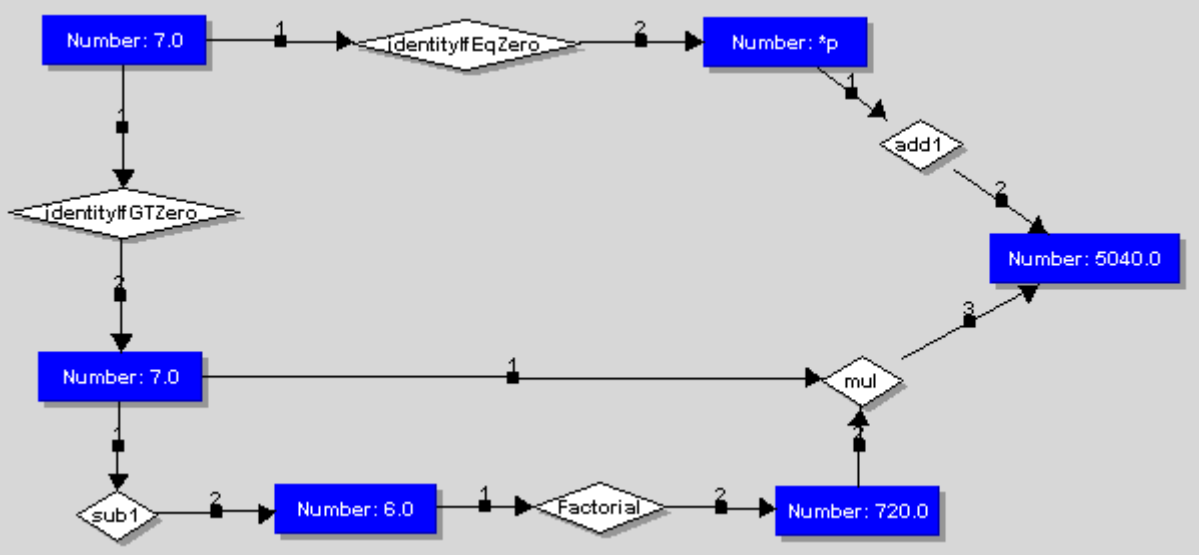

Figure 6-5 The result of executing the recursive factorial actor.

For this to work, there must be an entity called Factorial for the executor of the same name to map to. This is achieved on lines 67 and 68 of Listing 4:

```
r = file (dir + "Factorial.CGF");
actor Factorial(n) is r.readGraph();
```

The need for such a special case can be avoided as described in the 'Lambda and Actors' section of the 'The pCG Language' chapter. Listing 5 and the figure below show how this has been accomplished for the recursive factorial actor, using a *self* node in place of a Factorial node.

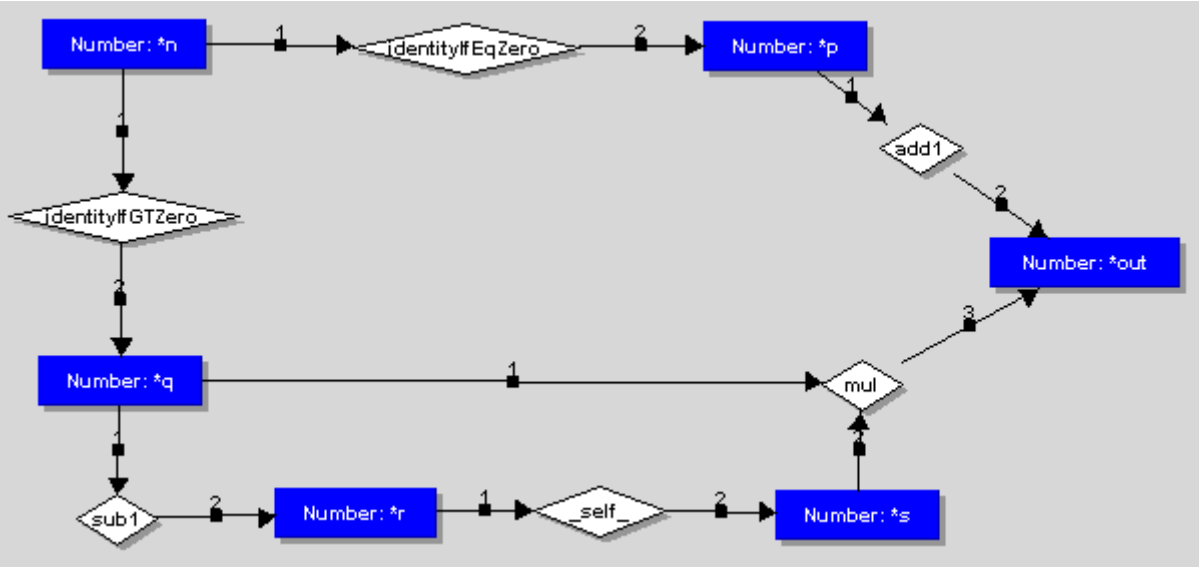

Figure 6-6 Recursive anonyous factorial actor graph.

The following code fragment from Listing 5 reads the defining graph of Figure 6-6 and activates it without the need for an actor type definition. First, the source concept referent must be bound. This could be done via concept restriction, a join operation, or a projection operation (e.g. within the context of a process rule's precondition matching)[58], or explicitly as shown below, by assigning to the source concept's designator attribute:

```
r = file (dir + "FactorialY.CGF");
g = r.readGraph();
foreach c in g.concepts do
  if c.designator == "*n" then c.designator = 7; last; end [59]
end
h = activate g;
```

The resulting graph is identical to the one generated by the named factorial actor above, except of course that the recursive executor is named *self*.

See 'The pCG Language' and 'The Implementation of pCG' for more details regarding the topics of actor blocking and recursive actor invocation.

6.4 Iterative Factorial Process

6.4.1 Description of the pCG Process Definition

The partially specified example of an iterative factorial process from [Mineau 1998] has been fully specified in pCG and is shown in Listing 6. The most noteworthy aspect of this is that it proves that Mineau's process formalism works substantially as laid out in his paper.

Lines 21–49 of Listing 6 define functions used by actor executors. Lines 50–154 define the iterative factorial process, consisting of 8 rules, compared to Mineau's original 12. The following lines (155–158) construct and assert a graph representing the value of `n` for which the factorial is to be computed:

```
// Graph representing the initial value for a variable n.
n = 7;
varN = "[Integer:*a " + n + "][Variable:*b'#n'](val?b?a)";
```

[58] See the 'Built-in Types' section of 'The pCG Language' for operations on concept and graph types.
[59] The `foreach` loop sets the designator of the anonymous factorial actor's source concept.

```
assert varN.toGraph();
```

The next few lines (159–163) create a process invocation graph consisting of:

- A trigger as input which is asserted by the process engine to indicate that the line to start on is line zero;
- An Integer result concept as output that will be specialised to a particular value by the final rule of the process.

Here are those lines:

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

The following 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;

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

What has been achieved is that actor and process *invocations* in pCG:

- Can be represented by a similar invocation graph using standard CGIF [Sowa 1999], as per the 'Special nodes vs direct execution of CGs' section of the 'Literature Review' chapter;

- An arbitrary graph can have the statement `activate` applied to it, and the meaning of an executor node will be determined by the interpreter from the node's underlying definition.

Although the definition mechanisms of actors and processes are quite different, process rule preconditions may themselves contain actors and processes. However, invocation graphs cannot contain both actor and process nodes or more than one process node.

The pCG version differs in some ways from Mineau's paper-based example, and these differences are discussed next.

6.4.2 Graph Format

- CGIF is used to represent graphs, not LF.

- As suggested on [Mineau 1998] page 68, locators have been used for line numbers, e.g. #L2, instead of the incorrect use of referent variables such as `*L2` found elsewhere in Mineau's example. 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 (e.g. `#f` vs `*f`) to avoid match ambiguity, e.g. see `pre6` actor on page 71 of [Mineau 1998]. See also Figure 6-7 for an example of Mineau's use of `*f`. [Mineau 1998] makes a general remark that "…we illustrate the definition of a process, not an actual individual process ready for execution. Consequently, all figures…display generic concepts instead of individuals." While this makes sense for the value of a variable, e.g.

  ```
  [Variable: #i] -> (val) -> [Integer: *x]
  ```

  it does not make sense for the names of the variables since they are known statically.

- pCG identifiers may only contain alphanumeric characters or underscores, hence an executor name like `<=` is not possible, since a function with that name cannot be defined. One must instead use a name such as `LTorEq`.

- Using concepts with blank referents in graphs may cause errors, so referent variables are used where a completely generic concept would be expected. The result is the same[60].

6.4.3 Erasure vs Negation

As discussed in 'The pCG Language' section 'Processes', apart from a special use of negation in preconditions, pCG has a simple model of assertion and retraction (erasure). In [Mineau 1998], the terms retraction and negation are used somewhat confusingly and often apparently interchangeably. For example, in [Mineau 1998] 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.

The 'Literature Review' has commented in more detail upon this confusion between retraction and negation of a graph. To be clear however, pCG uses retraction (graph erasure) in post-conditions, not negation.

The negation symbol is missing from most graphs in the rules of [Mineau 1998], but for example, in the post-condition of rule 8, the variable `f`'s old value must first be negated before the new value is asserted. If negation *were* 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 is a good example of why retraction not negation should be used, otherwise what one ends up with is numerous negated variable `f` assignment graphs. In other words, one is saying increasingly more about what the value of f is *not*, rather than what the value of f actually *is*. This is essentially the Frame Problem [Sowa 2000] [Bos 1997].

In Listing 6, `[Line:#Ln]<-(done)` — where n is an integer corresponding to a C program line — is not used at all. Instead, `[Line:#Ln]<-(to_do)` is asserted in one rule's post-condition and retracted in another's when the latter's precondition matches that "to-do" graph. As an example of the difference, between the two approaches, the precondition of rule 2 in [Mineau 1998] checks that the graphs:

```
[Line:#L3]<-(to_do)  [61]
[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 than Mineau's.

See also the 'Literature Review' regarding negation vs erasure.

---

[60] At the time of writing, this seems to be a bug in pCG.

[61] Although, as mentioned previously, referents of the form `*Ln` rather than `#Ln` are incorrectly used in Mineau's graphs.

6.4.4 Unnecessary Rules and Actors

Since the variable `f` has not been assigned by the time the process engine reaches rules 2 and 3 in [Mineau 1998], no graph of the form:

```
[Variable: #f] -> (val) -> [Integer: *z2]
```

will have been asserted into the process's local KB. Hence these rules will not fire. Figure 6-7 shows the relevant precondition of Mineau's rule 2 as an example. Rule 3 has a similar graph, the only difference being that it represents the false case, hence `[Boolean: False]` replaces `[Boolean: True]`:

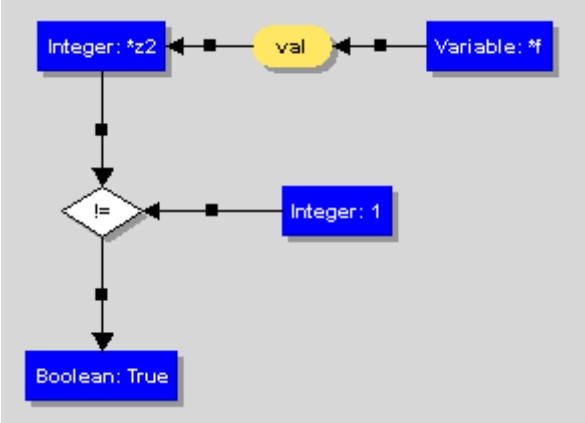

Figure 6-7 A rule 2 graph from [Mineau 1998] with erroneous referent (*f) and actor.

The "`!=`" actors in rules 2 and 3 are not necessary for the same reason, i.e. the variable `f` graph does not yet exist. The same is true for rules 4 and 5 which assign the variable `i`. Rules 2 and 4 have been changed to simply match the line number, and assert the variable value graphs, while rules 3 and 5 have been omitted.

Similar remarks can be made about the "`!=`" actor node in rules 8, 9, 10, and 11. The only purpose of these rules is to compute new values of `f` and `i`. There is no need for example, to test whether the product of `i` and `f` that is about to be assigned to the latter is the same as the current value of `f`. Hence, the "`!=`" actor may be omitted. As in the case of rules 2–5, in the absence of True and False cases, only rules 8 and 10 are necessary.

The pCG distribution (see 'Availability and Requirements' in 'The pCG Language') also has other versions of the iterative factorial process. One has action blocks which provide output regarding rule matching attempts and firings. The other is the same as Listing 6 but rule and referent variable names have been made less obfuscated than in Mineau's example.

6.5 A Process To Solve The Sisyphus-I Room Allocation Problem

6.5.1 The Problem

The Sisyphus-I room allocation problem [Linster 1999] is a constraint allocation problem in which a group of people in a research environment 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" group must be allocated alone to a large room. Another is that smokers and non-smokers may not share rooms. Yet another is that researchers on different projects should share rooms with people from different projects to prevent insularity. There are two slightly different problem statements, and the difference between them is that Katharina, the head of project and smoker is replaced by Christian, a researcher and smoker. The key issue here is that 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 by virtue of his more lowly status, unless doing so would place him with a non-smoker, or unless there are only single rooms remaining.

6.5.2 A Solution in pCG

All the information required to solve the problem is contained in the HTML found at the URL of the problem description [Linster 1999]. Listing 7 is a Perl script which is applied to the HTML [62] to generate the simple CGs of Listing 8. Listing 9 shows the manually created CGs describing Christian which must be substituted for Katharina in the second problem statement.

Listing 10 is approximately 500 lines of pCG code which solves both Sisyphus-I problem statements. The problem statement calls for "…traces of the problem-solving processes…for both problem statements. This means that…we want description of operational problem solvers." [Linster 1999] The program satisfies that criterion by generating output in two forms:

- Text showing what questions are being asked and who is allocated to which rooms as a result. A final output is the room allocations as CGs asserted in the caller's (top-level) KB.

- A graphic display showing names being plotted against an image of the research group's rooms in the "chateau" [Linster 1999]. Observing this display change over 10 seconds or so [63] enhances comprehension of rule firing, and in particular highlights the difference between the first and second problem statement solutions.

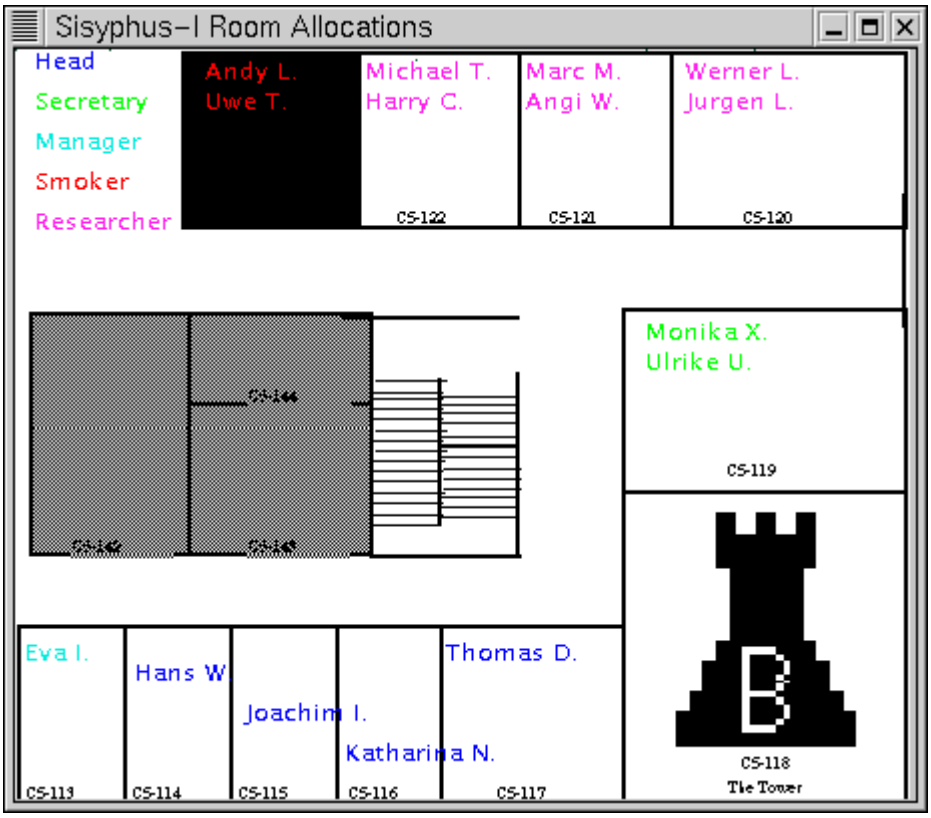

Figure 6-8 The allocations for the first problem statement.

Listing 11 shows the solution to the first problem statement [64], while Figure 6-8 shows the final room allocations [65]. Similarly, Listing 12 shows the solution to the second problem statement, and Figure 6-9

[62] This HTML is also provided in the pCG distribution.
[63] On a 550 MHz Pentium III Linux 6.0 machine.
[64] Captured by the `script` command under Linux.

shows the corresponding final room allocations. Notice in Listing 12 that Christian is allocated last after all higher priority constraints have been satisfied, whereas Katharina is allocated much earlier, as shown by Listing 11, due to her higher status.

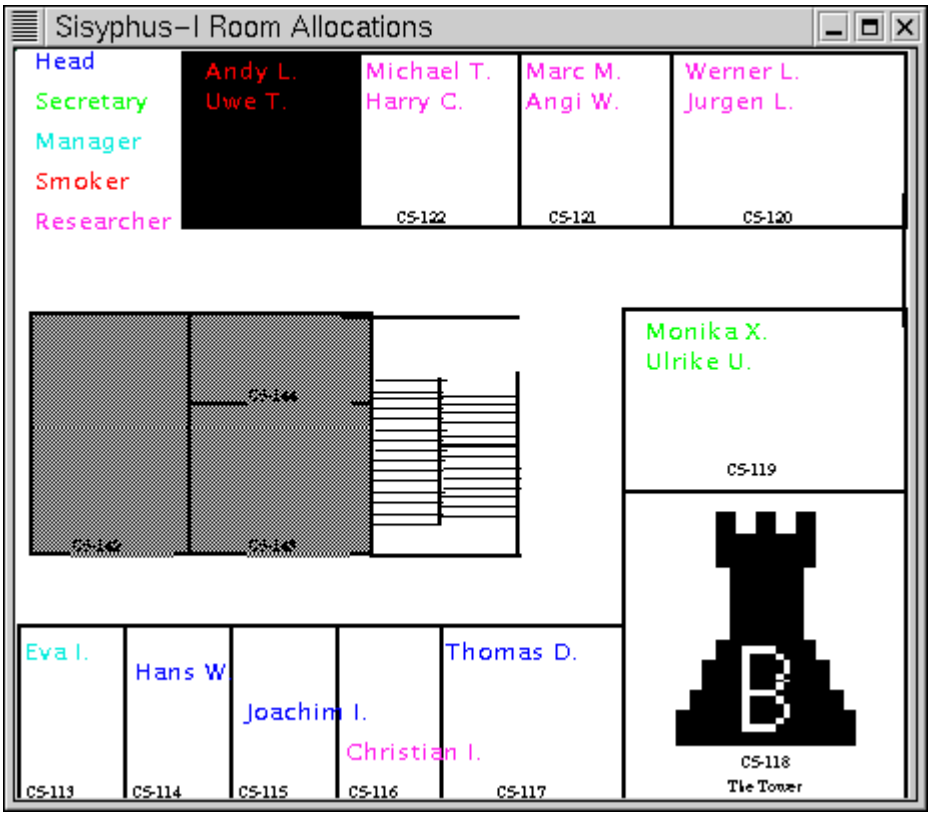

Figure 6-9 The allocations for the second problem statement.

6.5.3 Program Description

Lines 6–20 of Listing 10 create the concept and relation type hierarchies (ontology) for the problem's world. Lines 21–44 define functions to be used by actors in preconditions of rules in the process [66]. Lines 45–102 define functions to be used in the action code blocks of the process, or in the main part of the program. Lines 103–364 consist of the process definition.

Lines 367–408 define YQT member class colours and room coordinates, open a window, and display the rooms image and a legend.

Lines 411–445 open a default (Listing 8) or command-line specified file containing the graphs generated by the Perl script of Listing 7, and read all CGs contained therein. All graphs pertaining to one person are joined into a single graph for that person. This reduces the search space and shaves 2.5 seconds off the execution time of an earlier version which asserted each of the single-relation graphs, rather than a single graph with many relations per person. An example of such a graph in LF and CGIF is:

```
[Person:'Michael T.'] -
        (Chrc) -> [Role: 'Researcher']
        (Member) -> [Project: 'BABYLON Product']
        (Chrc) -> [Smoker: 'No']
```

[65] Note that the black square is assumed to be room C5-123. This is how the image appears in [Linster 1999].

[66] One of these, `NotSame`, is not used in the current solution, but is discussed elsewhere in the 'Program Description' section.

```
            (Chrc) -> [Hacker: 'Yes']
            (Coworker) -> [Person: 'Hans W.']

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

Lines 446–452 assert a few more graphs pertaining to certain YQTmembers. The Perl script of Listing 7 is able to extract most of the required information automatically from a table in [Linster 1999], correcting for a few inconsistencies, e.g. sometimes `Yes` rather than `True` is used to indicate what is essentially a boolean characteristic. A few facts must be extracted manually from the text of [Linster 1999], and these are the subject of lines 446–452. A more general solution would store even these in a file however.

Next, lines 453–482 assert graphs that declare information about what rooms are available, their centrality, size, vacancies and so on. An earlier version of the pCG program had a size relation in each of these graphs. Later, a room type hierarchy was created to make it possible to ask for any kind of room. See lines 6–20 for these types which were inspired by [Mineau 1999c]. While using a size concept such as `[Size: *dontCareHowBig]` (and an associated relation) 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.

Line 486 invokes the `Sisyphus` process. Note that this process takes no parameters, each room allocation graph being asserted in the caller's KB via an `option export` associated with the appropriate post-condition contexts.

Finally, lines 487–497 iterate over the top-level KB projecting a room filter onto each graph, displaying matches as CGs. Listings 11 and 12 show these room allocation graphs.

6.5.4 Discussion

[Linster 1999] 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 suggested one in that smokers are allocated after all non-smoking researchers, which still produces the desired outcome.

In a number of cases, two rules are required for correct allocation to large rooms which can hold two occupants. The ordering of these rules is important. 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. The rule `allocateSecondResearcher` must appear before `allocateFirstResearcher`, otherwise the latter will fire twice, yielding allocation into two different large rooms, making the final 3 pairs of allocations (see steps 8, 9, and 10 in [Linster 1999]) impossible without a smoker/non-smoker conflict. The lesson here is that the more constrained rule should appear first, not unlike recursive rules in Prolog. In accordance with this, we allocate non-smoking researchers with this pair of rules, followed by anyone else who is left and happens to smoke.

Likewise, the rule `allocateSecondSecretary` must appear before `allocateFirstSecretary`, otherwise the latter will fire twice, yielding allocation into two

different large rooms. The same comment can be made regarding `allocateSecondSmoker` and `allocateFirstSmoker`, the aim of which is to pair off smokers into large rooms.

The final rule `allocateRemainingResearcher` is a catch-all rule, designed to cope with the Second Problem Statement. It simply says that any remaining researcher should be allocated to *any* remaining room, ignoring smoker/non-smoker conflicts. There should probably be three rules in fact: two to handle a large room in the manner above (e.g. `allocateSecondRemainingResearcher` and `allocateFirstRemainingResearcher`, in that order) , and one to handle a single room. It just so happens that we know that there is only one single small room remaining by the time we get to this rule with the provided input.

The role of actors in process preconditions may be seen by examining a rule such as `allocateFirstSecretary` (starting on line 150). A `GreaterThan` actor is used to test whether a large room's vacancy count is greater than 1, thus:

```
[LargeRoom:*a'*roomLabel'][Location:*b'Central']
[Integer:*c'*num'][Integer:*d 1][Boolean:*e"true"]
  (Chrc?a?b)(Vacancy?a?c)<GreaterThan?c?d|?e>
```

This could have been accomplished without an actor in fact, by instantiating the `[Integer:*c'*num']` concept to `[Integer:*c 2]`. However, there is a dependency between this variable designator and the next precondition which uses a `Decrement` actor to decrement the vacancy count:

```
[LargeRoom:*a'*roomLabel'][Location:*b'Central']
[Integer:*c'*num'][Integer:*d'*decNum']
    (Chrc?a?b)(Vacancy?a?c)<Decrement?c|?d>
```

The old value `*num` and the new value `*decNum` are used in that rule's post-condition to retract and assert graphs representing the changed information. In fact, neither actor is required for this trivial computation. One could instead have graphs with hard-coded vacancy counts (of 1 and 2) in the graphs in question. Doing so may improve the execution time of this process also.[67] For more elaborate computation, actors would be harder to do without however. Examples are those of Mineau's iterative factorial process, e.g. `Multiply`, `Subtract`, since the initial value of the `n` variable for example, is not statically known.

As already mentioned, the program of Listing 10 solves both problem statements, however *no rule* imposes a constraint which ensures that people on the same project are not allocated the same room. The process works without such a constraint given the supplied ordering of YQT members in [Linster 1999], but if the order of graphs originally asserted (e.g. from Listing 11) in the KB were to change, this constraint could be violated. What is required is for graphs representing two different individuals to be compared in terms of the coworker or project relations, perhaps via a `NotSame` actor, or some other constraint application. The main problem is that in a given rule, identical precondition graphs in a single pCG rule will yield identical matches given the same KB, e.g.

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

This suggests that under some circumstances, pCG processes ought to employ something like backward chaining, essentially treating the conjunction of precondition graphs as a single goal akin to Prolog's approach, in this case to ensure that different `proj1` and `proj2` combinations are tried. Essentially, it is not possible in a single precondition 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. This issue requires further investigation. [Mineau 1999a] and [Mineau 1999c] also point to a possible solution in terms of the application of constraints.

[67] See the pCG distribution for just such an alternative implementation.

6.6 Conclusion

The pCG language examples presented in this chapter prove that Mineau's process mechanism works, but that there are deficiencies.

This chapter also pointed out some ways in which pCG's implementation is different from Mineau's original theory. Such difference is to be expected given that one can rarely consider all the issues by pure reason alone, and that some details were not elucidated in [Mineau 1998].

The 'Future Work' chapter suggests directions for further experiments, investigations, and applications.

7 Future Work

Mineau's mechanism has been implemented and found to work, but there is much else that could be done by way of the refinement and application of pCG and processes. This final chapter indicates some possible directions.

7.1 pCG and the Process Mechanism

[Mineau 1998] claims in his conclusion that the automatic generation of process rule sets from arbitrary input is possible. This claim must be tested, and pCG provides a suitable target language for translators or generators.

In the conclusion of [Mineau 1998], 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. [Mineau 1999a] 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. The Sisyphus-I problem described in the 'Experiments' chapter whereby two researchers on the same project should not share an office might also be solved by an application of constraints as discussed in [Mineau 1999c].

Processes have no backtracking capability as found in Prolog or Prolog+CG [Sowa 2000] [Kabbaj 2000a]. This could be incorporated into pCG such that the conjunction of preconditions in a rule is treated as a goal. This would permit alternative sub-matches to be tried at the level of individual preconditions (including referent variables therein), in order for the conjunction of preconditions in a rule to be satisfied, rather than stopping at the first attempt at matching individual preconditions. Again, the Sisyphus-I room allocation problem solution presented in the 'Experiments' chapter gives one motivation for this.

The process algorithm needs to be reconciled with the CG Rules of Inference in [Sowa 1999], particularly with respect to the use of contexts. See also [Esch 1995].

Certain aspects of the process algorithm bear re-examination. For example, see the footnote in 'The Implementation of pCG' relating to the projection of subgraphs (single relations) of a precondition graph onto graphs in the KB, in the case where a precondition contains an actor that may be attempting to glue together other graphs in the KB for the purpose of some arbitrary computation.

The conformity relation should be implemented in pCG. The instantiation rules of Prolog+CG provide one example of how this could be done [Kabbaj 2000a]. An example of where this would be useful is given in the 'Experiments' chapter when discussing a recursive factorial actor. In general, the conformity relation is important for ensuring that the individual to which an instantiated concept refers can be reasonably denoted by that concept. [Wermelinger 1997] laments that the conformity relation has often been neglected and argues that it should remain a first class aspect of CST.

Projection algorithms as described by [Willems 1995] and others should be examined and incorporated into pCG if more efficient. The context lattices of [Mineau 1997] may be another more efficient alternative to the current implementation. This paper is summarised in 'Appendix A — Contexts'.

pCG's variation on the projection algorithm should be incorporated into Corbett's unification framework [Corbett 2000b].

Graph morphology issues should be considered since pCG's projection algorithm does not currently test for structural equivalence. See for example [Corbett 2000b] and [Mugnier 2000], which indicate that this issue is non-trivial. For example, the latter remarks that "…projection is generally not a one-to-one mapping ($\Pi(G)$ is generally not isomorphic to G)."

Currently, the process execution engine stops when there are no more matching preconditions or the final rule has been reached. One could investigate whether it makes sense for a process to be interactively stopped and restarted, or stopped according to other criteria.

Multiple exit points in a process may be useful. This would mean marking particular rules which, once any one of them fired, would cause the process to exit. What one can *already* do in pCG is to invoke the return statement in a process action block and optionally return an arbitrary value.

Although lambda for CGs is minimally supported in pCG, the addition of lambda-based differentia graphs to concept and relation type hierarchies is not. In pCG, lambda is primarily present to support actors, but is also exposed for use in the language, providing a simple parameter substitution mechanism for graphs.

By a combination of strict concept argument arc ordering and correct arc directionality in a process invocation graph, it would be possible to abolish the formal argument list of a process, permitting something akin to C's *varargs* (variable argument list lengths). Given that a process does not make use of its arguments by name, in contrast to functions and some actors, there ought to be no consequences. A completely opposite possibility is to keep the formal parameter list in place and to permit values other than contexts to be passed to a process, and to be lexically scoped for use in code blocks. This would constitute a fusion of current process and function parameter lists. Which would be more useful? One could of course simulate varargs with a list as a parameter, so perhaps the second possibility provides greater generality.

Can concurrency be supported in pCG (processes or dataflow graphs) or are side effects likely to be rampant? In cases where referential transparency cannot be guaranteed in dataflow graphs (e.g. conflicting sink concepts), a synchronisation mechanism would be required. Given that processes are entirely state based, this issue would come to the fore immediately for them.

An Integrated Development Environment (IDE) could be provided for pCG to cater for platforms where a command line is not typical (e.g. on a Macintosh). Java's Swing GUI framework would be a suitable choice for this purpose due to its richness and platform independence. It ought also to be possible to invoke CharGer from such an IDE.

For Unix afficionados, an Emacs major mode for pCG would be useful[68].

pCG could act as a scripting language for CharGer, to permit, for example, arbitrary actors.

How to make pCG generate CGIF viewable in CharGer when not derived from CharGer drawn graphs, since positional information will be absent? This is essentially a graph drawing problem.

Finally, pCG should be made production quality, widely tested, and enhanced. A bug list and suggested list of improvements and possible new features can be found at:
http://www.adelaide.net.au/~dbenn/Masters/index.html

7.2 Experiments and Applications

[Lukose & Mineau 1998] presents a recursive factorial process instead of the iterative version of [Mineau 1998]. While pCG supports recursive processes by invocation of a process within the precondition of the rule of a process, this feature has yet to be tested. See 'The Implementation of pCG' for details. It would be interesting to see a recursive Towers of Hanoi process also; entertaining, if not useful.

The pCG distribution contains two simple Blocks World processes which query and modify block positions. These could be enhanced, for example to invoke other processes which carry out more complex queries and actions. Changes in the state of the world could be shown graphically.

The Sisyphus-I room allocation problem solution presented in the 'Experiments' chapter needs to be refined as noted in the previous section.

[68] The author would certainly use it.

Sowa's ignition example could be implemented as a pCG process. It would be interesting to compare this to Sowa's and Ellis's approaches. See the 'Literature Review' chapter regarding OO CGs.

As mentioned in the 'Literature Review' chapter, the business processes from [Gerbe´ 1998] were a motivation for [Mineau 1998]. An implementation of this could be attempted in pCG.

Comparison against more traditional production rule systems such as CLIPS might prove instructive from the viewpoint of determining whether a system such as pCG is more or less expressive and usable in practice. [Sowa 2000] and [Giarratano 1989] provide numerous CLIPS examples.

pCG's temporal logic capability, as outlined in the 'Processes' section of the 'Literature Review' chapter, should be explored.

[Calabretto 1998] is concerned with the acquisition and representation of knowledge via CGs in the Health Informatics domain. An extended example of what appears to be a production rule system is presented. The purpose of this is to "…alert the health care provider of new or worsening renal insufficiency based upon the serum creatinine level." A pCG process which captures the required knowledge, and generates the appropriate alerts would very likely be feasible.

In short, anywhere that discrete processes embodying non-trivial information need to be represented, pCG and its process mechanism could in principle be applied.

8 Appendix A — Contexts

[Mineau 1997] begins with the observation that [Sowa 1984] gave no formal definition of the notion of *context*, saying only that a context exists when a proposition is asserted. They suggest that a lack of consensus regarding the use of contexts prior to their paper also hindered any standardisation. They give a formal semantics for contexts when used for information packaging and give an application of these semantics to querying a KB.

A brief survey of the literature by the authors reveals the historical usage of contexts. They point out that while there is common agreement on Peirce's original definition of contexts as sheets of assertions, the same is not true for their semantics. [Sowa 1984] introduced a special concept type, PROPOSITION, an instance of which, *p*, has as a referent a set of CGs. The graphs are said to be true in the context of *p*. They cite an analysis of contexts by Sowa in 1995 which yielded 3 syntactic aspects: an information packaging mechanism, the contents (a set of assertions, i.e. CGs), permissible operations (e.g. import and export of assertions to and from a context). Mineau and Gerbe´ add semantics to Sowa's syntactic aspects in an attempt to bridge the gap between different viewpoints in the CG community.

Two main uses of contexts are identified. The first is the partitioning of a universe of discourse into distinct spaces (e.g. temporal) based upon the cognitive operations used by people to understand discourses. The second use is in representing texts by packaging sentences in contexts. The authors' view of contexts is expressed in terms of *worlds of assertions* that are created when the truth value of a set of assertions (CGs) is dependent upon certain conditions. Formally, a context Ci is defined to be a tuple consisting of an *intention* — I(Ci) — and an *extension* — E(Ci). The former typically consists of a single graph, while the latter is a set of graphs which are also conjunctively true in Ci. A context's intention acts to constrain the conditions under which that context exists and therefore whether the assertions it represents — its extension — hold in a given circumstance. They point out that a graph g in E(C1) could be an implicit member of E(C2) if g is a generalisation of another graph in E(C2) under the CST subsumption relation. Also, if the intention graphs of two contexts are related by subsumption, the extension of one will be a subset of the extension of another.

Since a graph can appear in more than one context, the set of all situations in which a particular graph is asserted is the *scope* of that graph. The authors give the example of Mary's thoughts forming a context in which one or more statements may be true, e.g. that Peter loves Mary. In this context, the intention is that Mary believes certain things to be true, and those things are in the extension — just one in this case: that Peter loves Mary. This may be true in the context of Mary's thoughts as well as Anne's (a friend of Mary's), but not Peter's.

The notion of a *formal context* builds upon the idea of simple contexts to yield, among other things, the scope of particular graphs. The set of all formal contexts forms a structure called a *context lattice* which can be computed automatically, providing an access structure which relates worlds of assertions. A KB can then be structured in terms of the semantics of the CGs contained within it. At the top of this lattice we find a formal context with an intention of Ø, and at the bottom, a context with an extension of Ø. This appears to be analogous to the universal and absurd types of a type lattice. The alternative is a single global context in which all graphs exist and no constraints apply.

Mineau & Gerbe´ [1997] provide a discussion about querying a KB whose contents are structured via formal contexts. They present two query equations for extracting graphs from the intention and extension of a formal context. In both cases, given a formal context and a single query graph, the set of all graphs that are the same as or a generalisation of the query graph is returned. They also give two equations for obtaining a context, given an extension or intention set. A key idea is that queries can be simplified by first identifying a target context, and that a KB's structure can be queried by searching intention and extension sets, optimising and reducing the complexity of KB querying.

9 Appendix B — Source Code Listings

This appendix contains non-trivial source code and other listings, located here so as not to clutter the main body of the document. In some cases, the listings are line numbered [69] in order to provide reference points for the purpose of exposition.

Listings use `Courier New` font. Particularly long or wide listings use 8 or 9 point rather than 10 point font size.

Listing 1 The map function in pCG.

```
function map(f, lists)
  // Sanity checks: n lists for f's n arguments.
  if lists.length != f.argcount then
    exit f.name + " requires " + f.argcount + " arguments.";
  end
  foreach thing in lists do
    if not (thing is list) then
      exit "the second argument to " + me.name +
           " must be a list of lists.";
    end
  end
  len = (lists[1]).length;
  foreach L in lists do
    if L.length != len then
      exit "all lists passed to " + me.name +
           " must be of the same length.";
    end
  end
  // Construct arguments for the function call from each list.
  results = {};
  n = 1;
  while n <= len do
    args = {};
    foreach L in lists do
      args.append(L[n]);
    end
    results.append(apply f args);
    n = n+1;
  end
  return results;
end
```

Listing 2 CGIF of the CharGer drawn dataflow graph in [Lukose & Mineau 1998].

```
[Remainder:*a'*e';CGHSD2.3b&Concept|243|232,190,117,25|&]
[Divisor:*b'*b';CGHSD2.3b&Concept|240|20,192,92,25|&]
[Dividend:*c'*a';CGHSD2.3b&Concept|238|24,51,104,25|&]
[Number:*d'*c';CGHSD2.3b&Concept|234|19,291,95,25|&]
[Sum:*e'*g';CGHSD2.3b&Concept|231|458,51,68,25|&]
[Quotient:*f'*d';CGHSD2.3b&Concept|229|238,50,103,25|&]
[SquareRoot:*g'*f';CGHSD2.3b&Concept|227|358,292,121,25|&]
<divide?c?b|?f?a;CGHSD2.3b&Actor|237|147,126,59,25|&>
<sqrt?d|?g;CGHSD2.3b&Actor|233|213,292,41,25|&>
```

[69] Using the Unix command: `nl -nrz` *path*

```
<plus?g?f|?e;CGHSD2.3b&Actor|226|395,124,42,25|&>/*CGHSD2.3b&Graph|22
5|0,0,40,25|&*/
```

Listing 3 Code to invoke the first dataflow graph in [Lukose & Mineau 1998].

```
000001 #!/home/david/cgp/pCG
000002 #
000003 # A test of Figure 1 from [Lukose and Mineau 1998].
000004 # The actor graph was drawn using CharGer. Note the
000005 # importance of arc ordering in this example.
000006 #
000007 # Note: This shows explicit actor invocation but pCG also supports
000008 #       implicit actor invocation.

000009 # ** Constants **
000010 DBL_QUOTE = 34;

000011 # ** Function definitions **
000012 function plus(x,y,z)
000013   z.designator = x.designator + y.designator;
000014 end

000015 function divide(x,y,q,r)
000016   q.designator = (x.designator div y.designator).round();
000017   r.designator = x.designator mod y.designator;
000018 end

000019 function sqrt(x,y)
000020   y.designator = (x.designator).sqrt();
000021 end

000022 function quoteNumbersForCharGer(g)
000023   # Quote numeric literal referents in a graph for CharGer
000024   # otherwise numbers won't be displayed for some reason.
000025   # These will later have to be changed to single quotes.
000026   foreach con in g.concepts do
000027     value = con.designator;
000028     if value is number then con.designator = "" + value; end
000029   end
000030   return g;
000031 end

000032 # ** Main Program **

000033 # ** Examples directory **
000034 home = _ENV.member("user.home");
000035 fs = (_ENV.member("file.separator"))[2];
000036 dir = home[2] + fs + "cgp" + fs + "examples" + fs;

000037 # Define actor.
000038 r = file (dir + "Figure1.CGF");
000039 actor Figure1(a,b,c) is r.readGraph();

000040 println "";
000041 println "Sinks: " + Figure1.sinkconcepts;
000042 println "Sources: " + Figure1.sourceconcepts;

000043 println "";
000044 println "** Defining graph for " + Figure1 + " is:";
000045 println (Figure1.defgraph).nocomments();

000046 # Invoke actor.
000047 println "";
000048 println "** Resulting graph for " + "Figure1(9,4,144) is:";
000049 g = Figure1(9,4,144);
000050 println g.nocomments();
```

```
000051 # Write resulting graph to a file.
000052 g = quoteNumbersForCharGer(g);
000053 out_path = dir + "Figure1Out.cgf";
000054 w = file (">" + out_path);
000055 w.writeln((g + "").replace((DBL_QUOTE).chr(), "'"));
000056 w.close();

000057 println "";
000058 println "See " + out_path + " for CharGer-readable result graph.";
59     println "";
```

Listing 4 Recursive factorial dataflow graph definition and invocation.

```
000001 // A test of Figure 4 (modified) from [Lukose and Mineau 1998].
000002 // The actor graph was drawn using CharGer. Arc ordering is not
000003 // important here. This example also shows how pCG's run-time
000004 // type testing can be put to good use.
000005 //
000006 // Note: This shows explicit actor invocation but pCG also supports
000007 //       implicit actor invocation.

000008 // ** Constants **
000009 DBL_QUOTE = 34;

000010 // ** Function definitions **
000011 function mul(x,y,z)
000012   operand1 = x.designator;
000013   operand2 = y.designator;
000014   if not (operand1 is number) or not (operand2 is number) then
000015     exit "Operand to " + me.name + " not a number!";
000016   end
000017   z.designator = operand1 * operand2;
000018 end

000019 function sub1(x,y)
000020   operand = x.designator;
000021   if not (operand is number) then
000022     exit "Operand to " + me.name + " not a number!";
000023   end
000024   y.designator = operand - 1;
000025 end

000026 function add1(x,y)
000027   operand = x.designator;
000028   if not (operand is number) then
000029     exit "Operand to " + me.name + " not a number!";
000030   end
000031   y.designator = operand + 1;
000032 end

000033 function identityIfEqZero(x,y)
000034   operand = x.designator;
000035   if not (operand is number) then
000036     exit "Operand to " + me.name + " not a number!";
000037   end
000038   if operand == 0 then
000039     y.designator = operand;
000040   end
000041 end

000042 function identityIfGTZero(x,y)
000043   operand = x.designator;
000044   if not (operand is number) then
000045     exit "Operand to " + me.name + " not a number!";
000046   end
000047   if operand > 0 then
```

```
000048     y.designator = operand;
000049   end
000050 end

000051 function quoteNumbersForCharGer(g)
000052   # Quote numeric literal referents in a graph for CharGer
000053   # otherwise numbers won't be displayed for some reason.
000054   # These will later have to be changed to single quotes.
000055   foreach con in g.concepts do
000056     value = con.designator;
000057     if value is number then con.designator = "" + value; end
000058   end
000059   return g;
000060 end

000061 // ** Main Program **

000062 // ** Examples directory **
000063 home = _ENV.member("user.home");
000064 fs = (_ENV.member("file.separator"))[2];
000065 dir = home[2] + fs + "cgp" + fs + "examples" + fs;

000066 // Define actor.
000067 r = file (dir + "Factorial.CGF");
000068 actor Factorial(n) is r.readGraph();

000069 println "";
000070 println "** Defining graph for " + Factorial + " is:";
000071 println (Factorial.defgraph).nocomments();

000072 // Invoke actor to compute factorial n.
000073 n = 7; // try making this a string
000074 println "";
000075 g = Factorial(n);
000076 println "** Resulting graph for " + "Factorial(" + n + ") is:";
000077 println g.nocomments();

000078 println "";
000079 println "Note: the above graph shows the top-level recursive
invocation";
000080 println "      after unwinding. To see other invocations, add 'option
TRACE'";
000081 println "      to the top of this program.";

000082 // Write resulting graph to a file.
000083 g = quoteNumbersForCharGer(g);
000084 out_path = dir + "FactorialOut.cgf";
000085 w = file (">" + out_path);
000086 w.writeln((g + "").replace((DBL_QUOTE).chr(), "'"));
000087 w.close();

000088 println "";
000089 println "See " + out_path + " for CharGer-readable result graph.";
90     println "";
```

Listing 5 Code to invoke a recursive anonymous factorial actor.

```
// A test of Figure 4 (modified) from [Lukose and Mineau 1998].
// The actor graph was drawn using CharGer. Arc ordering is not
// important here. This example also shows how pCG's run-time
// type testing can be put to good use.
//
// Note: The really interesting thing about this example is that
//       the file FactorialY.CGF contains an anonymous actor such
//       that when the recursive step is required, an executor node
//       called "_self_" is found, causing the current actor's defining
//       graph to be duplicated. This is essentially the CG equivalent
```

```
//       of the lambda calculus Y combinator which permits recursive
//       anonymous functions. Refer to any text on the lambda calculus.

// ** Constants **
DBL_QUOTE = 34;

// ** Function definitions **
function mul(x,y,z)
  operand1 = x.designator;
  operand2 = y.designator;
  if not (operand1 is number) or not (operand2 is number) then
    exit "Operand to " + me.name + " not a number!";
  end
  z.designator = operand1 * operand2;
end

function sub1(x,y)
  operand = x.designator;
  if not (operand is number) then
    exit "Operand to " + me.name + " not a number!";
  end
  y.designator = operand - 1;
end

function add1(x,y)
  operand = x.designator;
  if not (operand is number) then
    exit "Operand to " + me.name + " not a number!";
  end
  y.designator = operand + 1;
end

function identityIfEqZero(x,y)
  operand = x.designator;
  if not (operand is number) then
    exit "Operand to " + me.name + " not a number!";
  end
  if operand == 0 then
    y.designator = operand;
  end
end

function identityIfGTZero(x,y)
  operand = x.designator;
  if not (operand is number) then
    exit "Operand to " + me.name + " not a number!";
  end
  if operand > 0 then
    y.designator = operand;
  end
end

function quoteNumbersForCharGer(g)
  # Quote numeric literal referents in a graph for CharGer
  # otherwise numbers won't be displayed for some reason.
  # These will later have to be changed to single quotes.
  foreach con in g.concepts do
    value = con.designator;
    if value is number then con.designator = "" + value; end
  end
  return g;
end

// ** Main Program **

// ** Examples directory **
home = _ENV.member("user.home");
fs = (_ENV.member("file.separator"))[2];
```

```
dir = home[2] + fs + "cgp" + fs + "examples" + fs;

// Define actor.
r = file (dir + "FactorialY.CGF");
actor Factorial(n) is r.readGraph();

println "";
println "Sinks: " + Factorial.sinkconcepts;
println "Sources: " + Factorial.sourceconcepts;

println "";
println "** Defining graph for " + Factorial + " is:";
println (Factorial.defgraph).nocomments();

// Invoke actor to compute factorial n.
n = 7; // try making this a string
println "";
g = Factorial(n);
println "** Resulting graph for " + "Factorial(" + n + ") is:";
println g.nocomments();

println "";
println "Now, treat the graph as an anonymous actor and execute it...";
println "Hey presto! The Y combinator in CG actors.";
println "";
r = file (dir + "FactorialY.CGF");
g = r.readGraph();
foreach c in g.concepts do
  if c.designator == "*n" then c.designator = 7; last; end
end
h = activate g;
println "** Resulting graph for activation of: ";
println "";
println g.nocomments();
println "";
println "is:";
println "";
println h.nocomments();

// Write resulting graph to a file.
h = quoteNumbersForCharGer(h);
out_path = dir + "FactorialYOut.cgf";
w = file (">" + out_path);
w.writeln((h + "").replace((DBL_QUOTE).chr(), "'"));
w.close();

println "";
println "Note: the above graph shows the top-level recursive invocation";
println "      after unwinding. To see other invocations, add 'option
TRACE'";
println "      to the top of this program.";

println "";
println "See " + out_path + " for CharGer-readable result graph.";
println "";
```

Listing 6 The Iterative Factorial Process of [Mineau 1998].

```
000001 // Iterative factorial process.
000002 //
000003 // This is based upon the factorial process example in
000004 // Mineau's 1998 ICCS paper. The implementation is similar
000005 // although not identical. As much as possible has been kept
000006 // the same including defining variable and marker names,
000007 // and rule numbers.
000008 //
000009 // The C code from page 67 of [Mineau 1998]:
```

```
000010 //
000011 // L0: int fact(int n)
000012 // L1: { int f;
000013 // L2:   int i;
000014 // L3:   f = 1;
000015 // L4:   i = 2;
000016 // L5:   while (i <= n)
000017 // L6:   {  f = f * i;
000018 // L7:      i = i + 1;
000019 // L8:   }
000020 // L9:   return f; }

000021 // Types.
000022 // Not necessary here, since only referent specialisation will occur.
000023 concept Number > Integer;
000024 concept String > Identifier > Variable;

000025 // Function definitions.
000026 function LTorEq(a,b,result)
000027   first = a.designator;
000028   second = b.designator;
000029   if not (first is number) or not (second is number) then
000030     exit "The first 2 arguments to " + me.name + " must be numbers.";
000031   end
000032   result.designator = first <= second;
000033 end

000034 function Multiply(a,b,result)
000035   first = a.designator;
000036   second = b.designator;
000037   if not (first is number) or not (second is number) then
000038     exit "The first 2 arguments to " + me.name + " must be numbers.";
000039   end
000040   result.designator = first * second;
000041 end

000042 function Add(a,b,result)
000043   first = a.designator;
000044   second = b.designator;
000045   if not (first is number) or not (second is number) then
000046     exit "The first 2 arguments to " + me.name + " must be numbers.";
000047   end
000048   result.designator = first + second;
000049 end

000050 // The iterative factorial process definition.
000051 process fact(in trigger, out result)
000052   rule r1
000053     pre
000054       `[Integer:*a'*x'][Variable:*b'#n'](val?b?a)`;
000055       `[Line:*a'#L0'](to_do?a)`;
000056     end

000057     post
000058       `[ERASURE:[Line:*a'#L0'](to_do?a)]`;
000059       `[PROPOSITION:[Line:*a'#L3'](to_do?a)]`; // lines 1 and 2 can be
ignored
000060       `[PROPOSITION:[Line:*a'#L4'](to_do?a)]`; // why L3 and L4 at
once?
000061     end
000062   end // rule r1

000063   rule r2
000064     pre
000065       `[Line:*a'#L3'](to_do?a)`;
000066     end

000067     post
```

```
000068       `[PROPOSITION:[Integer:*a 1][Variable:*b'#f'](val?b?a)]`;
000069       `[ERASURE:[Line:*a'#L3'](to_do?a)]`;
000070     end
000071   end // rule r2


000072   rule r4
000073     pre
000074       `[Line:*a'#L4'](to_do?a)`;
000075     end

000076     post
000077       `[PROPOSITION:[Integer:*a 2][Variable:*b'#i'](val?b?a)]`;
000078       `[ERASURE:[Line:*a'#L4'](to_do?a)]`;
000079       `[PROPOSITION:[Line:*a'#L5'](to_do?a)]`;
000080     end
000081   end // rule r4

000082   rule r6
000083     pre
000084       `[Line:*a'#L5'](to_do?a)`;

000085       `[Integer:*a'*first'][Variable:*b'#i']
000086        [Integer:*c'*second'][Variable:*d'#n']
000087        [Boolean:*e"true"]
000088         (val?b?a)
000089           (val?d?c)
000090         <LTorEq?a?c|?e>`;
000091     end

000092     post
000093       `[ERASURE:[Line:*a'#L5'](to_do?a)]`;
000094       `[PROPOSITION:[Line:*a'#L6'](to_do?a)]`;
000095     end
000096   end // rule r6

000097   rule r7
000098     pre
000099       `[Line:*a'#L5'](to_do?a)`;

000100       `[Integer:*a'*first'][Variable:*b'#i']
000101        [Integer:*c'*second'][Variable:*d'#n']
000102        [Boolean:*e"false"]
000103         (val?b?a)
000104           (val?d?c)
000105         <LTorEq?a?c|?e>`;
000106     end

000107     post
000108       `[ERASURE:[Line:*a'#L5'](to_do?a)]`;
000109       `[PROPOSITION:[Line:*a'#L9'](to_do?a)]`; // exit the while loop
000110     end
000111   end // rule r7

000112   rule r8
000113     pre
000114       `[Line:*a'#L6'](to_do?a)`;

000115       `[Integer:*a'*z4'][Variable:*b'#f']
000116        [Integer:*c'*x'][Variable:*d'#i']
000117        [Integer:*e'*y4']
000118          (val?b?a)
000119          (val?d?c)
000120          <Multiply?a?c|?e>`;
000121     end

000122     post
000123       `[ERASURE:[Integer:*a'*z4'][Variable:*b'#f'](val?b?a)]`;
```

```
000124       `[PROPOSITION:[Integer:*a'*y4'][Variable:*b'#f'](val?b?a)]`;
000125       `[ERASURE:[Line:*a'#L6'](to_do?a)]`;
000126       `[PROPOSITION:[Line:*a'#L7'](to_do?a)]`;
000127     end
000128   end // rule r8

000129   rule r10
000130     pre
000131       `[Line:*a'#L7'](to_do?a)`;

000132       `[Integer:*a'*z4'][Variable:*b'#i']
000133        [Integer:*c 1]
000134        [Integer:*d'*y4']
000135          (val?b?a)
000136          <Add?a?c|?d>`;
000137     end

000138     post
000139       `[ERASURE:[Integer:*a'*z4'][Variable:*b'#i'](val?b?a)]`;
000140       `[PROPOSITION:[Integer:*a'*y4'][Variable:*b'#i'](val?b?a)]`;
000141       `[ERASURE:[Line:*a'#L7'](to_do?a)]`;
000142       `[PROPOSITION:[Line:*a'#L5'](to_do?a)]`;
000143     end
000144   end // rule r10

000145   rule r12
000146     pre
000147       `[Line:*a'#L9'](to_do?a)`;
000148       `[Integer:*a'*z5'][Variable:*b'#f'](val?b?a)`;
000149     end

000150     post
000151       `[ERASURE:[Line:*a'#L9'](to_do?a)]`;
000152     end
000153   end // rule r12
000154 end

000155 // Graph representing the initial value for a variable n.
000156 n = 7;
000157 varN = "[Integer:*a " + n + "][Variable:*b'#n'](val?b?a)";
000158 assert varN.toGraph();

000159 // Construct process invocation graph.
000160 s = "[PROPOSITION:*a[Line:*b'#L0'](to_do?b)]" +
000161     "[PROPOSITION:*c[Integer:*d'*z5']]" +
000162     "<fact?a|?c>";
000163 g = s.toGraph();

000164 println "Before process 'fact'. Graphs: " + _KB.graphs;
000165
000166 x = activate g;

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

Listing 7 Perl script to extract information about YQT members and generate CGs.

```
#!/usr/bin/perl
#
# Extract the YQT members from the HTML description of the
# Sisyphus-I problem and generate CGIF to standard output.
# Assumptions about the file format are made, as can be seen
# from the regular expressions used.
#
# David Benn, October 2000

if (@ARGV == 1) {
  # Read all lines from HTML file.
```

```
  open(YQT, $ARGV[0]) or die "can't open $ARGV[0] for reading.\n";
  @lines = <YQT>;
  close(YQT);

  $i = 0;
  while ($line = $lines[$i++]) {
    # Find next person.
    if ($line =~ /<TD WIDTH=211>(.+)<BR>/i) {
      $name = $1;
      $name =~ s/\&uuml\;/u/; # replace HTML u-umlaut with u

      # Extract key-value pairs for this person.
      while ($line = $lines[$i++]) {
       if ($line =~ /\s*<\/TD>/) {
         # Start of next person, so break out of loop.
         $i--;
         last;
       }

       # Key-value pair.
       if ($line =~ /\s*(.+)\s*=\s*(.+)\s*/) {
         $key = ucfirst($1);
         $value = ucfirst($2);
         $key =~ s/ //; # why necessary, but not for value?
         $value =~ s/<BR>//; # lines other than "Works-with" end with "<BR>"
         $value =~ s/True/Yes/; # all Yes or No

         # Skip over blank line to second colleagues line?
         if ($key =~ /Works\-with/ and $value =~ /\,\s*$/) {
           $i++;
           $line = $lines[$i++];
           if ($line =~ /\s*(.+)/) { # "." won't include linefeed
              $value .= $1;
           }
         }

         $value =~ s/\&uuml\;/u/; # replace HTML u-umlaut with u (Works-with)

         # Generate CGIF for this YQT member.
         if ($key ne 'Works-with') {
           if ($key eq 'Project') {
             $relation = 'Member';
           } else {
             $relation = "Chrc";
           }
           print "[Person:*a'$name'][$key:*b'$value']($relation?a?b)\n";
         } else {
           @coworkers = split(/\s*,\s*/, $value);
           foreach $coworker (@coworkers) {
             if ($coworker !~ /^\s+$/) {
              print "[Person:*a'$name'][Person:*b'$coworker']";
              print "(Coworker?a?b)\n";
             }
           }
         }
       }
      }
      print "\n";
    }
  }
} else {
  die "Sisyphus-I HTML description page required.\n";
}
```

Listing 8 CGs representing information for the first Sysiphus-I problem statement.

```
[Person:*a'Werner L.'][Role:*b'Researcher'](Chrc?a?b)
```

```
[Person:*a'Werner L.'][Project:*b'RESPECT'](Member?a?b)
[Person:*a'Werner L.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Werner L.'][Hacker:*b'Yes'](Chrc?a?b)
[Person:*a'Werner L.'][Person:*b'Angi W.'](Coworker?a?b)
[Person:*a'Werner L.'][Person:*b'Marc M.'](Coworker?a?b)

[Person:*a'Jurgen L.'][Role:*b'Researcher'](Chrc?a?b)
[Person:*a'Jurgen L.'][Project:*b'EULISP'](Member?a?b)
[Person:*a'Jurgen L.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Jurgen L.'][Hacker:*b'Yes'](Chrc?a?b)
[Person:*a'Jurgen L.'][Person:*b'Harry C.'](Coworker?a?b)
[Person:*a'Jurgen L.'][Person:*b'Thomas D.'](Coworker?a?b)

[Person:*a'Marc M.'][Role:*b'Researcher'](Chrc?a?b)
[Person:*a'Marc M.'][Project:*b'KRITON'](Member?a?b)
[Person:*a'Marc M.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Marc M.'][Hacker:*b'Yes'](Chrc?a?b)
[Person:*a'Marc M.'][Person:*b'Angi W.'](Coworker?a?b)
[Person:*a'Marc M.'][Person:*b'Werner L.'](Coworker?a?b)

[Person:*a'Angi W.'][Role:*b'Researcher'](Chrc?a?b)
[Person:*a'Angi W.'][Project:*b'RESPECT'](Member?a?b)
[Person:*a'Angi W.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Angi W.'][Hacker:*b'No'](Chrc?a?b)
[Person:*a'Angi W.'][Person:*b'Marc M.'](Coworker?a?b)
[Person:*a'Angi W.'][Person:*b'Werner L.'](Coworker?a?b)

[Person:*a'Andy L.'][Role:*b'Researcher'](Chrc?a?b)
[Person:*a'Andy L.'][Project:*b'TUTOR2000'](Member?a?b)
[Person:*a'Andy L.'][Smoker:*b'Yes'](Chrc?a?b)
[Person:*a'Andy L.'][Hacker:*b'No'](Chrc?a?b)

[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)

[Person:*a'Harry C.'][Role:*b'Researcher'](Chrc?a?b)
[Person:*a'Harry C.'][Project:*b'EULISP'](Member?a?b)
[Person:*a'Harry C.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Harry C.'][Hacker:*b'Yes'](Chrc?a?b)
[Person:*a'Harry C.'][Person:*b'Jurgen L.'](Coworker?a?b)
[Person:*a'Harry C.'][Person:*b'Thomas D.'](Coworker?a?b)

[Person:*a'Uwe T.'][Role:*b'Researcher'](Chrc?a?b)
[Person:*a'Uwe T.'][Project:*b'Autonomous Systems'](Member?a?b)
[Person:*a'Uwe T.'][Smoker:*b'Yes'](Chrc?a?b)
[Person:*a'Uwe T.'][Hacker:*b'Yes'](Chrc?a?b)

[Person:*a'Thomas D.'][Role:*b'Researcher'](Chrc?a?b)
[Person:*a'Thomas D.'][Project:*b'EULISP'](Member?a?b)
[Person:*a'Thomas D.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Thomas D.'][Hacker:*b'No'](Chrc?a?b)
[Person:*a'Thomas D.'][Person:*b'Jurgen L.'](Coworker?a?b)
[Person:*a'Thomas D.'][Person:*b'Harry B.'](Coworker?a?b)

[Person:*a'Monika X.'][Role:*b'Secretary'](Chrc?a?b)
[Person:*a'Monika X.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Monika X.'][Hacker:*b'No'](Chrc?a?b)
[Person:*a'Monika X.'][Person:*b'Thomas D.'](Coworker?a?b)
```

```
[Person:*a'Monika X.'][Person:*b'Ulrike U.'](Coworker?a?b)
[Person:*a'Monika X.'][Person:*b'Eva I.'](Coworker?a?b)

[Person:*a'Ulrike U.'][Role:*b'Secretary'](Chrc?a?b)
[Person:*a'Ulrike U.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Ulrike U.'][Hacker:*b'No'](Chrc?a?b)
[Person:*a'Ulrike U.'][Person:*b'Thomas D.'](Coworker?a?b)
[Person:*a'Ulrike U.'][Person:*b'Monika X.'](Coworker?a?b)
[Person:*a'Ulrike U.'][Person:*b'Eva I.'](Coworker?a?b)

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

[Person:*a'Eva I.'][Role:*b'Manager'](Chrc?a?b)
[Person:*a'Eva I.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Eva I.'][Hacker:*b'No'](Chrc?a?b)
[Person:*a'Eva I.'][Person:*b'Thomas D.'](Coworker?a?b)
[Person:*a'Eva I.'][Person:*b'Ulrike U.'](Coworker?a?b)
[Person:*a'Eva I.'][Person:*b'Monika X.'](Coworker?a?b)

[Person:*a'Joachim I.'][Role:*b'Researcher'](Chrc?a?b)
[Person:*a'Joachim I.'][Project:*b'ASERTI'](Member?a?b)
[Person:*a'Joachim I.'][Smoker:*b'No'](Chrc?a?b)
[Person:*a'Joachim I.'][Hacker:*b'No'](Chrc?a?b)

[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)
```

Listing 9 These graphs replace Katharina's for the second problem statement.

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

Listing 10 A pCG solution to the Sisyphus-I room allocation problem.

```
000001  // An attempt at a solution to the Sisyphus-I room allocation problem
000002  // using processes. It even works. Room allocations are shown against
000003  // the image on the Sisyphus-I problem description web page.
000004  //
000005  // David Benn, October-November 2000

000006  // ** Type Hierarchies **
000007  concept Person;
000008  concept Role;
000009  concept Project;
000010  concept Smoker;
000011  concept Hacker;
000012  concept Room > SingleRoom;
000013  concept Room > LargeRoom;
000014  concept Location;

000015  relation Attr;
000016  relation Chrc;
000017  relation Member;
000018  relation Coworker;
000019  relation Vacancy;
```

```
000020 relation Occupant;

000021 // ** Actor Executor Functions **

000022 function GreaterThan(x,y,result)
000023   op1 = x.designator;
000024   op2 = y.designator;

000025   if not (op1 is number) or not (op2 is number) then
000026     exit me.name + " expects numeric concept designators.";
000027   end

000028   result.designator = op1 > op2;
000029 end

000030 function NotSame(x,y,result)
000031   op1 = x.designator;
000032   op2 = y.designator;

000033   if not (op1 is string) or not (op2 is string) then
000034     exit me.name + " expects string concept designators.";
000035   end

000036   result.designator = not (op1 == op2);
000037 end

000038 function Decrement(x,result)
000039   op = x.designator;

000040   if not (op is number) then
000041     exit me.name + " expects a numeric concept input designator.";
000042   end

000043   result.designator = op-1;
000044 end

000045 // ** General Purpose Functions **

000046 function addToMatchingPerson(addition)
000047   newConcepts = addition.concepts;
000048   foreach g in _KB.graphs do
000049     // Join 2 graphs at head if head concepts identical.
000050     gConcepts = g.concepts;
000051     if newConcepts[1] == gConcepts[1] then
000052       retract g;
000053       assert g.joinAtHead(addition);
000054       return;
000055     end
000056   end
000057 end

000058 function findMatchingGraph(filter)
000059   foreach g in _KB.graphs do
000060     h = g.project(filter);
000061     if not (h is undefined) then
000062       return g;
000063     end
000064   end
000065 end

000066 function mkPropositionGraph(g)
000067   return ("[PROPOSITION:" + g + "]").toGraph();
000068 end

000069 function mkErasureGraph(g)
000070   return ("[ERASURE:" + g + "]").toGraph();
000071 end

000072 function getCorefVarValue(s)
000073   return s.substring(s.index("=")+2, s.length);
000074 end

000075 function getVarValue(key)
000076   foreach var in _KB.corefvars do
000077     if var.index(key) == 1 then
000078       return getCorefVarValue(var);
000079     end
```

```
000080    end
000081  end

000082  function getRoomLabel()
000083    return getVarValue("*roomLabel");
000084  end

000085  function getRoomNumber(label)
000086    num = label.substring(label.index("-")+1, label.length);
000087    return num.toNumber();
000088  end

000089  function getPersonName()
000090    return getVarValue("*name");
000091  end

000092  function showAllocation(kind, name, label)
000093    println "--> " + kind + " '" + name + "' allocated to " + label;
000094  end

000095  function plotName(name, roomLabel, occupantNum, color)
000096    roomNum = getRoomNumber(roomLabel);
000097    coord = roomXY[roomNum-112];
000098    x = coord[1]+5;
000099    y = coord[2] + 15*occupantNum;
000100    w.setColor(color);
000101    w.drawText(name, x, y);
000102  end

000103  // ** Processes **

000104  process Sisyphus()
000105    initial
000106    end

000107    rule allocateHeadOfYQT
000108      pre
000109        action
000110          println "Need to allocate room for the head of YQT?";
000111        end

000112        `[Person:*a'*name'][Head:*b'YQT'](Chrc?a?b)`;

000113        `[LargeRoom:*a'*roomLabel'][Location:*b'Central'](Chrc?a?b)`;
000114      end

000115      post
000116        action
000117          label = getRoomLabel();
000118          name = getPersonName();
000119          showAllocation("Head of YQT", name, label);
000120          plotName(name, label, 1, headColor);
000121        end

000122        `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel'](Occupant?a?b)]`;
000123         option export;

000124        mkErasureGraph(findMatchingGraph(_MATCHES[1])); // erase person
000125        mkErasureGraph(findMatchingGraph(_MATCHES[2])); // erase room
000126      end
000127    end // rule allocateHeadOfYQT

000128    rule allocateSecondSecretary
000129      pre
000130        action
000131          println "Need to allocate room for second secretary?";
000132        end

000133        `[Person:*a'*name'][Role:*b'Secretary'](Chrc?a?b)`;

000134        `[LargeRoom:*a'*roomLabel'][Location:*b'*somewhere'][Integer:*c 1]
000135           (Chrc?a?b)(Vacancy?a?c)`;
000136      end

000137      post
000138        action
000139          label = getRoomLabel();
```

```
000140          name = getPersonName();
000141          showAllocation("Second secretary", name, label);
000142          plotName(name, label, 2, secretaryColor);
000143        end

000144        `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel'](Occupant?a?b)]`;
000145         option export;

000146        mkErasureGraph(findMatchingGraph(_MATCHES[1])); // erase person
000147        mkErasureGraph(findMatchingGraph(_MATCHES[2])); // erase room
000148      end
000149    end // allocateSecondSecretary

000150    rule allocateFirstSecretary
000151      pre
000152        action
000153          println "Need to allocate room for first secretary?";
000154        end

000155        `[Person:*a'*name'][Role:*b'Secretary'](Chrc?a?b)`;

000156        // Note: "true" must be double-quoted currently. Not an issue for
000157        // non-boolean literals. This is just a peculiarity of the way in
000158        // which pCG handles boolean literals as a special case.

000159        `[LargeRoom:*a'*roomLabel'][Location:*b'Central'][Integer:*c'*num']
000160         [Integer:*d 1][Boolean:*e"true"]
000161           (Chrc?a?b)(Vacancy?a?c)<GreaterThan?c?d|?e>`; // num == 2

000162        `[LargeRoom:*a'*roomLabel'][Location:*b'Central'][Integer:*c'*num']
000163         [Integer:*d'*decNum']
000164           (Chrc?a?b)(Vacancy?a?c)<Decrement?c|?d>`;
000165      end

000166      post
000167        action
000168          label = getRoomLabel();
000169          name = getPersonName();
000170          showAllocation("First secretary", name, label);
000171          plotName(name, label, 1, secretaryColor);
000172        end

000173        `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel'](Occupant?a?b)]`;
000174         option export;

000175        mkErasureGraph(findMatchingGraph(_MATCHES[1])); // erase person

000176        `[ERASURE:[LargeRoom:*a'*roomLabel'][Location:*b'Central']
000177                  [Integer:*c'*num']
000178                    (Chrc?a?b)(Vacancy?a?c)]`;

000179        `[PROPOSITION:[LargeRoom:*a'*roomLabel'][Location:*b'Central']
000180                      [Integer:*c'*decNum']
000181                        (Chrc?a?b)(Vacancy?a?c)]`;

000182      end
000183    end // allocateFirstSecretary

000184    rule allocateManager
000185      pre
000186        action
000187          println "Need to allocate room for the manager?";
000188        end

000189        `[Person:*a'*name'][Role:*b'Manager'](Chrc?a?b)`;

000190        `[SingleRoom:*a'*roomLabel'][Location:*b'Central'][Integer:*c 1]
000191         [Size:*d'Small']
000192           (Chrc?a?b)(Vacancy?a?c)(Attr?a?d)`;
000193      end

000194      post
000195        action
000196          label = getRoomLabel();
000197          name = getPersonName();
000198          showAllocation("Manager", name, label);
000199          plotName(name, label, 1, managerColor);
```

```
000200        end

000201        `[PROPOSITION:[Person:*a'*name']][Room:*b'*roomLabel']](Occupant?a?b)]`;
000202         option export;

000203        mkErasureGraph(findMatchingGraph(_MATCHES[1])); // erase person
000204        mkErasureGraph(findMatchingGraph(_MATCHES[2])); // erase room
000205      end
000206    end // rule allocateManager

000207    rule allocateAHead
000208      pre
000209        action
000210         println "Need to allocate room for a head?";
000211        end

000212        `[Person:*a'*name']][Head:*b'*someProject']](Chrc?a?b)`;

000213        `[SingleRoom:*a'*roomLabel']][Location:*b'Central']][Integer:*c 1]
000214         [Size:*d'Small']
000215           (Chrc?a?b)(Vacancy?a?c)(Attr?a?d)`;
000216      end

000217      post
000218        action
000219         label = getRoomLabel();
000220         name = getPersonName();
000221         showAllocation("Head", name, label);
000222         plotName(name, label, 1, headColor);
000223        end

000224        `[PROPOSITION:[Person:*a'*name']][Room:*b'*roomLabel']](Occupant?a?b)]`;
000225         option export;

000226        mkErasureGraph(findMatchingGraph(_MATCHES[1])); // erase person
000227        mkErasureGraph(findMatchingGraph(_MATCHES[2])); // erase room
000228      end
000229    end // rule allocateAHead

000230    rule allocateSecondResearcher
000231      pre
000232        action
000233         println "Need to allocate room for a second researcher?";
000234        end

000235        `[Person:*a'*name']][Role:*b'Researcher']][Project:*c'*proj1']
000236           [Smoker:*d'No']
000237           (Chrc?a?b)(Member?a?c)(Chrc?a?d)`;

000238        `[LargeRoom:*a'*roomLabel']][Location:*b'*somewhere']][Integer:*c 1]
000239           (Chrc?a?b)(Vacancy?a?c)`;
000240      end

000241      post
000242        action
000243         label = getRoomLabel();
000244         name = getPersonName();
000245         showAllocation("Second researcher", name, label);
000246         plotName(name, label, 2, researcherColor);
000247        end

000248        `[PROPOSITION:[Person:*a'*name']][Room:*b'*roomLabel']](Occupant?a?b)]`;
000249         option export;

000250        mkErasureGraph(findMatchingGraph(_MATCHES[1])); // erase person
000251        mkErasureGraph(findMatchingGraph(_MATCHES[2])); // erase room
000252      end
000253    end // allocateSecondResearcher

000254    rule allocateFirstResearcher
000255      pre
000256        action
000257         println "Need to allocate room for a first researcher?";
000258        end

000259        `[Person:*a'*name']][Role:*b'Researcher']][Smoker:*c'No']
000260           (Chrc?a?b)(Chrc?a?c)`;
```

```
000261        `[LargeRoom:*a'*roomLabel'][Location:*b'*somewhere'][Integer:*c'*num']
000262         [Integer:*d 0][Boolean:*e"true"]
000263           (Chrc?a?b)(Vacancy?a?c)<GreaterThan?c?d|?e>`;

000264        `[LargeRoom:*a'*roomLabel'][Location:*b'*somewhere'][Integer:*c'*num']
000265         [Integer:*d'*decNum']
000266           (Chrc?a?b)(Vacancy?a?c)<Decrement?c|?d>`;
000267      end

000268      post
000269        action
000270         label = getRoomLabel();
000271         name = getPersonName();
000272         showAllocation("First researcher", name, label);
000273         plotName(name, label, 1, researcherColor);
000274        end

000275        `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel'](Occupant?a?b)]`;
000276         option export;

000277        mkErasureGraph(findMatchingGraph(_MATCHES[1])); // erase person

000278        `[ERASURE:[LargeRoom:*a'*roomLabel'][Location:*b'*somewhere']
000279                  [Integer:*c'*num']
000280                    (Chrc?a?b)(Vacancy?a?c)]`;

000281        `[PROPOSITION:[LargeRoom:*a'*roomLabel'][Location:*b'*somewhere']
000282                      [Integer:*c'*decNum']
000283                        (Chrc?a?b)(Vacancy?a?c)]`;
000284      end
000285    end // allocateFirstResearcher

000286    rule allocateSecondSmoker
000287      pre
000288        action
000289         println "Need to allocate room for second smoker?";
000290        end

000291        `[Person:*a'*name'][Smoker:*b'Yes'](Chrc?a?b)`;

000292        `[LargeRoom:*a'*roomLabel'][Location:*b'*somewhere'][Integer:*c 1]
000293           (Chrc?a?b)(Vacancy?a?c)`;
000294      end

000295      post
000296        action
000297         label = getRoomLabel();
000298         name = getPersonName();
000299         showAllocation("Second smoker", name, label);
000300         plotName(name, label, 2, smokerColor);
000301        end

000302        `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel'](Occupant?a?b)]`;
000303         option export;

000304        mkErasureGraph(findMatchingGraph(_MATCHES[1])); // erase person
000305        mkErasureGraph(findMatchingGraph(_MATCHES[2])); // erase room
000306      end
000307    end // allocateSecondSmoker

000308    rule allocateFirstSmoker
000309      pre
000310        action
000311         println "Need to allocate room for first smoker?";
000312        end

000313        `[Person:*a'*name'][Smoker:*b'Yes'](Chrc?a?b)`;

000314        `[LargeRoom:*a'*roomLabel'][Location:*b'*somewhere'][Integer:*c'*num']
000315         [Integer:*d 1][Boolean:*e"true"]
000316           (Chrc?a?b)(Vacancy?a?c)<GreaterThan?c?d|?e>`; // num == 2

000317        `[LargeRoom:*a'*roomLabel'][Location:*b'*somewhere'][Integer:*c'*num']
000318         [Integer:*d'*decNum']
000319           (Chrc?a?b)(Vacancy?a?c)<Decrement?c|?d>`;
000320      end
```

```
000321      post
000322        action
000323          label = getRoomLabel();
000324          name = getPersonName();
000325          showAllocation("First smoker", name, label);
000326          plotName(name, label, 1, smokerColor);
000327        end

000328        `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel'](Occupant?a?b)]`;
000329         option export;

000330        mkErasureGraph(findMatchingGraph(_MATCHES[1])); // erase person

000331        `[ERASURE:[LargeRoom:*a'*roomLabel'][Location:*b'*somewhere']
000332                  [Integer:*c'*num']
000333                    (Chrc?a?b)(Vacancy?a?c)]`;

000334        `[PROPOSITION:[LargeRoom:*a'*roomLabel'][Location:*b'*somewhere']
000335                      [Integer:*c'*decNum']
000336                        (Chrc?a?b)(Vacancy?a?c)]`;
000337      end
000338    end // allocateFirstSmoker

000339    rule allocateRemainingResearcher
000340      pre
000341        action
000342          println "Need to allocate room for a remaining researcher?";
000343        end

000344        `[Person:*a'*name'][Role:*b'Researcher'](Chrc?a?b)`;

000345        `[Room:*a'*roomLabel'][Location:*b'*somewhere'](Chrc?a?b)`;
000346      end

000347      post
000348        action
000349          label = getRoomLabel();
000350          name = getPersonName();
000351          showAllocation("Remaining researcher", name, label);
000352          plotName(name, label, 1, researcherColor);
000353        end

000354        `[PROPOSITION:[Person:*a'*name'][Room:*b'*roomLabel'](Occupant?a?b)]`;
000355         option export;

000356        // Next two are not necessary since this is the final rule, so the
000357        // process will exit after this post-condition block. If another
000358        // rule is added after this one however, these erasures will again
000359        // be necessary, so leave them in.
000360        mkErasureGraph(findMatchingGraph(_MATCHES[1])); // erase person
000361        mkErasureGraph(findMatchingGraph(_MATCHES[2])); // erase room
000362      end
000363    end // allocateRemainingResearcher
000364  end
000365
000366  // ** Main Program **

000367  // Text colours.
000368  red = {255,0,0};
000369  green = {0,255,0};
000370  blue = {0,0,255};
000371  aqua = {0,237,221};
000372  violet = {255,33,237};

000373  headColor = blue;
000374  secretaryColor = green;
000375  managerColor = aqua;
000376  smokerColor = red;
000377  researcherColor = violet;

000378  // Room coordinates.
000379  roomXY = {{0,290},     // C5-113
000380            {55,300},    // C5-114
000381            {110,320},   // C5-115
000382            {160,340},   // C5-116
000383            {210,290},   // C5-117
```

```
000384           {0,0},       // C5-118 -- The Tower, so don't care
000385           {310,130},   // C5-119
000386           {330,0},     // C5-120
000387           {250,0},     // C5-121
000388           {170,0},     // C5-122
000389           {90,0}};     // C5-123

000390 // Open a window containing the Sisyphus-I rooms graphic.
000391 // * Modify URL appropriately. *

000392 u = new Util;
000393 w = new Window;
000394 w.open("Sisyphus-I Room Allocations", 50, 50, 450, 375);
000395 w.drawImage("file:/home/david/cgp/examples/sisyphus-1/Image2.gif", 0,0);

000396 // Wait long enough for image to load before proceeding.
000397 u.sleep(2);

000398 // Legend.
000399 w.setColor(headColor);
000400 w.drawText("Head", 10,10);
000401 w.setColor(secretaryColor);
000402 w.drawText("Secretary", 10,30);
000403 w.setColor(managerColor);
000404 w.drawText("Manager", 10,50);
000405 w.setColor(smokerColor);
000406 w.drawText("Smoker", 10,70);
000407 w.setColor(researcherColor);
000408 w.drawText("Researcher", 10,90);

000409 // Which YQT member file?
000410 // * Modify path appropriately. *

000411 if _ARGS.length == 1 then
000412   // Arbitrary data file, e.g. for
000413   // the second problem description.
000414   inFilePath = _ARGS[1];
000415 end else
000416   home = _ENV.member("user.home");
000417   dir = home[2] + "/cgp/examples/sisyphus-1/";
000418   inFilePath = dir + "yqt.cgif";
000419 end

000420 // Read CGIF created from HTML description taken from web
000421 // (URL: http://ksi.cpsc.ucalgary.ca/KAW/Sisyphus/Sisyphus1/)
000422 // and assert graphs in top-level KB. The table of YQT members
000423 // and information pertaining to them is the source of the
000424 // information.

000425 println "Asserting YQT member graphs.";
000426 yqt = file inFilePath;
000427 lines = yqt.readall();
000428 yqt.close();
000429 g = ""; // really want to say "undefined" here
000430 foreach line in lines do
000431   if line.length > 0 then
000432     if g is graph then
000433       // Add to current person's graph.
000434       g = g.join(line.toGraph());
000435     end else
000436       // New person.
000437       g = line.toGraph();
000438     end
000439   end else
000440     // Assert current person's graph.
000441     assert g;
000442     // "Reset" graph for next person.
000443     g = "";
000444   end
000445 end

000446 // Add YQT member characteristics from descriptive paragraph
000447 // of (section 2.1.2) of above URL.

000448 println "Asserting more information about certain YQT members.";
000449 addToMatchingPerson(`[Person:*a'Thomas D.'][Head:*b'YQT'](Chrc?a?b)`);
```

```
000450  addToMatchingPerson(`[Person:*a'Hans W.'][Head:*b'BABYLON
Product'](Chrc?a?b)`);
000451  addToMatchingPerson(`[Person:*a'Joachim I.'][Head:*b'Other'](Chrc?a?b)`);
000452  addToMatchingPerson(`[Person:*a'Katharina N.'][Head:*b'Other'](Chrc?a?b)`);

000453  // Room graphs asserted in the top-level KB, to be modified
000454  // in the process's local KB. These will be copied to the
000455  // invoked process's local KB.
000456  //
000457  // These will be searched for, possibly vacancies modified, and
000458  // in most cases, retracted from the process's KB when a particular
000459  // allocation sequence is completed.

000460  rooms = { "[LargeRoom:*b'C5-117'][Location:*c'Central'][Integer:*d 2]" +
000461            "(Chrc?b?c)(Vacancy?b?d)",
000462           "[LargeRoom:*b'C5-119'][Location:*c'Central'][Integer:*d 2]" +
000463            "(Chrc?b?c)(Vacancy?b?d)",
000464           "[LargeRoom:*b'C5-120'][Location:*c'NonCentral'][Integer:*d 2]" +
000465            "(Chrc?b?c)(Vacancy?b?d)",
000466           "[LargeRoom:*b'C5-121'][Location:*c'NonCentral'][Integer:*d 2]" +
000467            "(Chrc?b?c)(Vacancy?b?d)",
000468           "[LargeRoom:*b'C5-122'][Location:*c'NonCentral'][Integer:*d 2]" +
000469            "(Chrc?b?c)(Vacancy?b?d)",
000470           "[LargeRoom:*b'C5-123'][Location:*c'NonCentral'][Integer:*d 2]" +
000471            "(Chrc?b?c)(Vacancy?b?d)",
000472           "[SingleRoom:*b'C5-113'][Location:*c'Central'][Integer:*d 1]" +
000473            "(Chrc?b?c)(Vacancy?b?d)",
000474           "[SingleRoom:*b'C5-114'][Location:*c'Central'][Integer:*d 1]" +
000475            "(Chrc?b?c)(Vacancy?b?d)",
000476           "[SingleRoom:*b'C5-115'][Location:*c'Central'][Integer:*d 1]" +
000477            "(Chrc?b?c)(Vacancy?b?d)",
000478            "[SingleRoom:*b'C5-116'][Location:*c'Central'][Integer:*d 1]" +
000479            "(Chrc?b?c)(Vacancy?b?d)" };

000480  foreach room in rooms do
000481    assert room.toGraph();
000482  end

000483  // Invoke a process to solve the Sisyphus-I room allocation problem
000484  // The end result will be a new graph for each member indicating his/her
000485  // room allocation, and a window displaying the result.

000486  result = Sisyphus();

000487  // Iterate over the KB projecting room filter onto each graph,
000488  // displaying matches.

000489  println "";
000490  println "Room allocation graphs:";
000491  filter = `[Person:*a'*x'][Room:*b'*y'](Occupant?a?b)`;
000492  foreach g in _KB.graphs do
000493    h = g.project(filter);
000494    if not (h is undefined) then
000495      println " " + h;
000496    end
000497  end

000498  println "";
499     println "ctrl-c to exit";
```

Listing 11 Solution trace for the first problem statement.

```
Script started on Mon Nov 27 09:32:13 2000
[david@twist ~/cgp]$ pCG examples/sisyphus-1/scg-1.cgp
Asserting YQT member graphs.
Asserting more information about certain YQT members.
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?
--> Manager 'Eva I.' allocated to C5-113
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?
--> Head 'Hans W.' allocated to C5-114
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?
--> Head 'Joachim I.' allocated to C5-115
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?
--> Head 'Katharina N.' allocated to C5-116
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 'Werner L.' allocated to C5-120
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 'Jurgen L.' allocated to C5-120
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 'Marc M.' allocated to C5-121
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 'Angi W.' allocated to C5-121
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
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?
Need to allocate room for second smoker?
Need to allocate room for first smoker?
--> First smoker 'Andy L.' allocated to C5-123
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?
Need to allocate room for second smoker?
--> Second smoker 'Uwe T.' allocated to C5-123
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?
Need to allocate room for second smoker?
Need to allocate room for first smoker?
Need to allocate room for a remaining researcher?

Room allocation graphs:
 [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)

ctrl-c to exit

[david@twist ~/cgp]$
Script done on Mon Nov 27 09:32:38 2000
```

Listing 12 Solution trace for the second problem statement.

```
Script started on Mon Nov 27 09:40:27 2000
[david@twist ~/cgp]$ pCG examples/sisyphus-1/scg-1.cgp
examples/sisyphus-1/yqt2nd.cgif
Asserting YQT member graphs.
Asserting more information about certain YQT members.
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?
--> Manager 'Eva I.' allocated to C5-113
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?
--> Head 'Hans W.' allocated to C5-114
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?
--> Head 'Joachim I.' allocated to C5-115
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 'Werner L.' allocated to C5-120
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 'Jurgen L.' allocated to C5-120
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 'Marc M.' allocated to C5-121
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 'Angi W.' allocated to C5-121
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
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?
Need to allocate room for second smoker?
Need to allocate room for first smoker?
--> First smoker 'Andy L.' allocated to C5-123
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?
Need to allocate room for second smoker?
--> Second smoker 'Uwe T.' allocated to C5-123
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?
Need to allocate room for second smoker?
Need to allocate room for first smoker?
Need to allocate room for a remaining researcher?
--> Remaining researcher 'Christian I.' allocated to C5-116

Room allocation graphs:
 [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"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)
  [Person:*a"Christian I."][Room:*b"C5-116"](Occupant?a?b)

ctrl-c to exit

[david@twist ~/cgp]$
Script done on Mon Nov 27 09:45:18 2000
```

10 References

[Abelson 1992]
Abelson, H. and diSessa, A., 1992, Turtle Geometry: The Computer as a Medium for Exploring Mathematics, MIT Press, Cambridge, Massachusetts.

[Angelova 1997]
Angelova, G., Damyanova, S. Toutanova, K. & Bontcheva, K., Menu-Based Interfaces to Conceptual Graphs: The CGLex Approach, In *Proceedings of the 5th International Conference on Conceptual Structures*, Seattle, Washington, USA, pp. 603–610, August 1997.

[Bates 1997]
Bates, D., R-beta: a recursive anonymous function, April 1997, [Accessed Online: November 2000], URL: http://www.r-project.org/nocvs/mail/r-help/1997/0063.html

[Bos 1997]
Bos, C., Botella, B., 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.

[Calabretto 1998]
Calabretto, J-P., Knowledge Acquisition Models for Health Informatics: Higher Order Conceptual Structures for Decision Support in Medical Information Systems. Master of Computing Studies Research Project and Dissertation, October 1998, Deakin University.

[Corbett 2000a]
Corbett, D., Extending Conceptual Graphs with unification over constraints, unpublished draft, University of South Australia, School of Computer and Information Science, October 2000.

[Corbett 2000b]
Corbett, D., A Framework for Conceptual Graph Unification, In *Proceedings of the 8th Annual Workshop on Conceptual Structures*, Darmstadt, Germany, pp. 162–174, August 2000.

[Cyre 1998]
Cyre, W., Executing Conceptual Graphs, *In Proceedings of the 6th International Conference on Conceptual Structures*, Montpellier, France, pp. 51–64, August 1998.

[Delugach 1990]
Delugach, H., Using Conceptual Graphs to Analyze Multiple Views of Software Requirements, *Proceedings of the 5th Annual Workshop on Conceptual Graphs*, Peter Eklund, Laurie Gerholz, editors, Linkoping University, Boston & Stockholm, pp. 17–22, July 1990.

[Delugach 1991]
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.

[Delugach 1999]
Delugach, H., CharGer: A Conceptual Graph Editor, [Accessed Online: October 2000], URL: http://www.cs.uah.edu/~delugach/CharGer/

[Delugach 2000]
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.

[Dewdney 1993]
Dewdney, A.K., 1993, The New Turing Omnibus: 66 Excursions in Computer Science, Computer Science Press, New York, pp. 276–281.

[Ellis 1994]
Ellis, G. and Levinson, R. (Eds.), In *Proceedings of the 3rd International Workshop on Peirce: A Conceptual Graphs Workbench*, University of Maryland, August 19 1994.

[Ellis 1995]
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.

[Ellis 1997]
Ellis, G., PEIRCE: A Conceptual Graphs Workbench, [Accessed Online: November 2000], URL: http://www.cs.adelaide.edu.au/users/peirce/

[Esch 1994]
Esch, J., Contexts, Canons and Coreferent Types, In *Proceedings of the 2nd International Conference on Conceptual Structures*, College Park, Maryland, USA, pp. 185–195, August 1994.

[Esch 1995]
Esch, J. & Levinson, R., An Implementation Model for Contexts and Negation in Conceptual Graphs, In *Proceedings of the 3rd International Conference on Conceptual Structures*, Santa Cruz, CA, USA, pp. 247–261, August 1995.

[Friedman 1989]
Friedman, D.P. & Felleisen, M., The Little Lisper: Trade Edition, MIT Press, Cambridge, Massachusetts, chapter 9.

[Gabriel 1996]
Gabriel, R.P., 1996, Patterns of Software: Tales from the Software Community, Oxford University Press, New York, pp. 107–108.

[Garner 1997]
Garner, B., Tsui, E. & Lukose, D., Deakin Toolset: Conceptual Graphs Based Knowledge Acquisition, Management, and Processing Tools. In *Proceedings of the 5th International Conference on Conceptual Structures*, Seattle, Washington, USA, pp. 587–593, August 1997.

[Genest 1998]
Genest, D. & Salvat, E., A Platform Allowing Type Nested Graphs: How CoGITo Became CoGITaNT, In *Proceedings of the 6th International Conference on Conceptual Structures*, Montpellier, France, pp. 154–161, August 1998.

[Gerbe´ 1998]
Gerbe´, O., Keller, R., Mineau G., Conceptual Graphs for Representing Business Processes in Corporate Memories. In *Proceedings of the 6th International Conference on Conceptual Structures*, Montpellier, France, pp. 401-415, August 1998.

[Giarratano 1989]
Giarratano, J. & Riley, G., 1989, Expert Systems: Principles and Programming, PWS-KENT, Boston, chapter 7.

[Kabbaj 1999a]
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.

[Kabbaj 1999b]
Kabbaj, A., Synergy as an Hybrid Object-Oriented Conceptual Graph Language, In *Proceedings of the 7th International Conference on Conceptual Structures*, Blacksburg, VA, USA, pp.247–261, July 1999.

[Kabbaj 2000a]
Kabbaj, A. & Janta-Polczynski, M., From Prolog++ to Prolog+CG: A CG Object-oriented programming language, In *Proceedings of the 8th International Conference on Conceptual Structures*, Darmstadt, Germany, pp. 536–550, August 2000.

[Kabbaj 2000b]
Kabbaj, A., SYNERGY, [Accessed Online: October 2000], URL: http://www.insea.ac.ma/CGTools/CGTools.htm#SYNERGY

[Kabbaj 2000c]
Kabbaj, A., Dynamic Formation of Knowledge Bases, [Accessed Online: October 2000], URL: http://www.insea.ac.ma/CGTools/CGTools.htm#Dynamic

[Kremer 1997]
Kremer, R., Lukose, D., & Gaines, B., Knowledge Modelling Using Annotated Flow Chart. In *Proceedings of the 5th International Conference on Conceptual Structures*, Seattle, Washington, USA, pp. 213–227, August 1997.

[Linster 1999]
Linster, M., Documentation for the Sisyphus-I Room Allocation Task, [Accessed Online: November 2000], URL: http://ksi.cpsc.ucalgary.ca/KAW/Sisyphus/Sisyphus1/

[Louden 1993]
Louden, K.C., 1993, Programming Languages: Principles and Practice, PWS-Kent, Boston, pp. 11–12, 229, 402–408, 361–362.

[Lukose 1993]
Lukose, D., Executable Conceptual Structures, *In Proceedings of the 1st International Conference on Conceptual Structures*, Quebec City, Canada, pp. 223–237, August 1993.

[Lukose 1997a]
Lukose, D., CGKEE: Conceptual Graph Knowledge Engineering Environment, In *Proceedings of the 5th International Conference on Conceptual Structures*, Seattle, Washington, USA, pp. 598–602, August 1997.

[Lukose 1997b]
Lukose, D., Complex Modelling Constructs in MODEL-ECS, In *Proceedings of the 5th International Conference on Conceptual Structures*, Seattle, Washington, USA, pp. 228–243, August 1997.

[Lukose & Mineau 1998]
Lukose, D. & Mineau, G., A Comparative Study of Dynamic Conceptual Graphs. *In Proceedings of the 11th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW-98)*. Banff, Alberta, Canada. Section VKM-7. April 18–23 1998.

[Mann 1994]
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.

[Mineau 1997]
Mineau, G. & Gerbe´, O., Contexts: A Formal Definition of Worlds of Assertions. In *Proceedings of the 5th International Conference on Conceptual Structures*, Seattle, Washington, USA, pp. 80–94, August 1997.

[Mineau 1998]
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.

[Mineau 1999a]
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.

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

[Mineau 1999c]
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.

[Mineau 2000]
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.

[Mortensen 2000]
Mortensen, K.H., Christensen, S., & Kummer, O., Petri Net Tools on the Web, [Accessed Online: November 2000], URL: http://www.daimi.au.dk/PetriNets/tools/

[Mugnier 2000]
Mugnier, M.L., Knowledge Representation and Reasonings Based on Graph Homomorphism, In *Proceedings of the 8th International Conference on Conceptual Structures*, Darmstadt, Germany, pp.172–192, August 2000.

[Nenkova 1999]
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.

[Parr 2000]
Parr, T., ANTLR, [Accessed Online: November 2000], URL: http://www.antlr.org/

[Pictorius 2000]
Pictorius Incorporated, Prograph, [Accessed Online: November 2000], URL: http://www.pictorius.com/prograph.html

[Pollitt 1998]
Pollitt, S., Burrow, A. & Eklund, P.W., WebKB-GE — A Visual Editor for Canonical Conceptual Graphs. In *Proceedings of the 6th International Conference on Conceptual Structures*, Montpellier, France, pp. 111–118, August 1998.

[Schaps 1999]
Schaps, G.L., Compiler Construction with ANTLR and Java, Dr. Dobb's Journal, March 1999, pp. 84–89.

[Shinghal 1992]
Shinghal, R., 1992, Formal Concepts in Artificial Intelligence, Chapman & Hall, London, pp. 14, 250, 318–331, 646.

[Southey 1999]
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.

[Sowa 1984]
Sowa, J.F., Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley, 1984, chapter 3 and pp. 187–195.

[Sowa 1999]
Sowa, J.F. et al, Conceptual Graph Standard, draft proposed American National Standard (dpANS) NCITS.T2/98-003, 1999. [Accessed Online: November 1999], URL: http://www.bestweb.net/~sowa/cg/cgdpansw.htm

[Sowa 2000]
Sowa, J.F., Knowledge Representation: Logical, Philosophical, and Computational Foundations, Brooks/Cole, 2000, chapter 3, chapter 4, appendix A, appendix B.

[Sowa 2000a]
Sowa, J.F., E-mail message to the CG mailing list: CG standard, 28th August 2000.

[Sowa 2000b]
Sowa, J.F., E-mail message to the CG mailing list: CG standard for Actors, 29th August 2000.

[Steinman 1995]
Steinman, S., Why do I need Prograph CPX?, In *MacTech Magazine, Volume 11, Issue 7*, June 1995, [Accessed Online: November 2000], URL: http://www.mactech.com/articles/mactech/Vol.11/11.07/PrographApp/

[Sterling 1986]
Sterling, L. & Shapiro, E., 1986, The Art of Prolog: Advanced Programming Techniques, MIT Press, Cambridge, Massachusetts, pp. 179, 180.

[Tierney 1997]
Tierney, L., More on Symbolic Derivatives, May 1997, [Accessed Online: November 2000], URL: http://www.stat.umn.edu/~luke/classes/8932/notes/node10.html

[Tsui 1997]
Tsui, E., Garner, B. & Lukose, D., ECG: Extendible Graph Processor. In *Proceedings of the 5th International Conference on Conceptual Structures*, Seattle, Washington, USA, pp. 594–597, August 1997.

[von Thun 2000]
von Thun, M., An informal tutorial on Joy, [Accessed Online: November 2000], URL: http://www.latrobe.edu.au/www/philosophy/phimvt/j01tut.html

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

[Wermelinger 1997]
Wermelinger, M., A Different Perspective on Canonicity, In *Proceedings of the 5th International Conference on Conceptual Structures*, Seattle, Washington, USA, pp. 110–123, August 1997.

[Willems 1995]
Willems, M., Projection and Unification for Conceptual Graphs, In *Proceedings of the 3rd International Conference on Conceptual Structures*, Santa Cruz, CA, USA, pp. 278–292, August 1995.