A Common Process Language Version 1.0

Stephen T. Polyak

Artificial Intelligence Applications Institute (AIAI)
Institute for Representation and Reasoning (IRR)
The University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN United Kingdom
Steve_Polyak@ed.ac.uk
Research Paper 933

November 26, 1998

Contents

1 Purpose **2**

2 Introduction **2**
2.1 Basic Tokens and Syntactic Categories . . . 2
2.2 Lexicons . . . 3
2.3 Grammars . . . 3

3 Core Language **4**
3.1 Commands . . . 4
3.2 Sort Definitions . . . 5
3.3 Assignments . . . 5
3.4 Extensions . . . 5
3.5 Process . . . 6
3.6 Nodes and Activities . . . 6
3.7 Activity-Relatable Objects and Agents . . . 7
3.8 Timepoints . . . 8
3.9 Sets . . . 8
3.10 Domain Levels . . . 8

4 Core Language - Constraints **9**
4.1 Issue Constraints . . . 9
4.2 Node Constraints . . . 10
4.3 Ordering Constraints . . . 10
4.4 Variable Constraints . . . 11
4.5 Auxiliary Constraints . . . 11

5 Example: Three Pigs Building **12**

6 Extensions **14**
6.1 Tool-Based . . . 14
6.2 Rationale . . . 15

1 Purpose

The Common Process Framework (CPF) provides methods, tools, and representations for integrating AI planning technology and plan representations into organisations for the primary purpose of synthesising and managing organisational processes. It meshes and extends past and present University of Edinburgh planning research and infuses it with new work in ontological engineering, knowledge sharing, software/requirements engineering and design rationale. One component of the framework is the Common Process Language (CPL) which is derived from a sorted, first order language. This language is used to express processes and process domain knowledge from a constraint-based view of the world. The elements of this perspective are defined in the Common Process Ontology (CPO) which underpins the terms and concepts in CPL. The language is used by all of the CPF tools for exchanging rich process knowledge. The CPL approach allows for very flexible constraint expressions which are defined via extended grammar specifications. This paper presents the CPL and defines both its core lexicon and grammar along with default sub-language extensions for constraint expressions.

2 Introduction

The language used to describe domains and processes in CPF is CPL. It includes variables, functions and predicates in addition to the standard logical operators, such as negations, conjunction, quantification, etc. The language is strongly typed, where each type is a finite sort. This paper provides the definition of the CPL language using an extended Backus-Naur form (BNF)[1]. The following extended BNF conventions are used

- A vertical bar “|” indicates an exclusive disjunction; thus, for example, if C1 and C2 are two syntactic categories “C1|C2” indicates an occurrence of either an instance of C1 or C2 but not both. The absence of such a bar between two constructs indicates a concatenation.
- An asterisk “*” immediately following a construct indicates that there can be any finite number (including 0) of instances of the construct.
- A plus sign “+” superscript immediately following a construct indicates that there can be one or more instances of the construct.
- Braces “{” and “}” are used to indicate grouping. Thus, “{C1|C2}+” indicates one or more instances of either C1 or C2.
- A construct surrounded by brackets (e.g. “[C1|C2]”) indicates that an instance of the indicated construct is optional.
- Nonterminals, representing categories of CPL expressions, start with “<” and end with “>”.
- Where necessary, the space character is represented by “<space>”.

2.1 Basic Tokens and Syntactic Categories

CPL is built up from a set of basic tokens and certain categories of expressions. This section presents these building blocks which will be used to define complex CPL expressions. They are listed below in the following BNF.

```
<uc-letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<lc-letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
```

[1]This BNF form is borrowed from the style used to describe the enhanced PIF language in “Foundations for Product Realization Process Knowledge Sharing”, Knowledge Based Systems, Inc., Final Report, U.S. Dept. of Commerce, Contract No. 50-DKNB-7-90095

```
   <letter> ::= <uc-letter> | <lc-letter>
    <digit> ::= 0|1|2|3|4|5|6|7|8|9
  <integer> ::= <digit>+
    <float> ::= <digit>*.<digit>+
   <number> ::= {[-]<integer>}|{[-]<float>}
     <oper> ::= -|~|#|$|*|+|/
    <punct> ::= _|-|~|!|@|#|$|%|^|&|*|(|)|+|=|`|:|;|'|<|>|,|.|?|/|||[|]|{|}|<space>
```

An expression is any string of basic tokens. We define four basic categories of expression.

```
     <b-con> ::= {<uc-letter>}{<letter>|<digit>}* {{_|-}{<letter>|<digit>}}*
     <b-var> ::= ?<b-con>
    <b-func> ::= {<oper>|<lc-letter>}{<letter>|<digit>}*{{_|-|.}{<letter>|<digit>}}*
    <b-pred> ::= {<lc-letter>}{<letter>|<digit>}*{{_|-|.}{<letter>|<digit>}}*
    <b-sort> ::= {<lc-letter>}{<letter>|<digit>}*{{_|-|.}{"|<letter>|<digit>}}*
<doc-string> ::= "{<letter>|<digit>|<punct>|\"|\\}*"
   <comment> ::= //{<letter>|<digit>|<punct>}* {{\n}|{\r}|{\r\n}}
```

Thus, a <b-con> (i.e., an expression derived from the nonterminal <b-con>) is a string of alphanumeric characters, dashes, and underscores that begins with an upper case letter and in which every dash and underscore is flanked on either side by a letter or digit. A <b-var> is the result of prefixing a <b-con> with a question mark. A <b-func> is just like a <b-con> except that it must begin with either an <oper>, a <punc>, or a lower case letter and it may have a "dot" separator as well. (Every <b-pred> is thus a <b-func>) A <doc-string> is the result of quoting any string of tokens; double quotes and the backslash can be used as well as long as they are preceded by a backslash. The $\backslash n$ and $\backslash r$ in the single line comment definition are meant to refer to the newline and carriage return characters.

2.2 Lexicons

The first-order CPL language is given in terms of a lexicon and grammar. The lexicon provides the basic "words" of the language, and the grammar determines how the lexical elements may be used to build the complex, well-formed expressions of the language. A CPL lexicon, λ, consists of the following:

- The expressions <space>,SORT,{,},(,),not,and,or,=>,<=>,forall, and exists;
- A denumerable recursive set V_λ of <b-var>s (i.e. expressions derived from the nonterminal <b-var> in the above BNF), known as the variables of λ;
- A recursive set C_λ of <b-con>s known as the constants of λ which includes at least the strings inf-, inf, +max, -max+.
- A recursive set F_λ of <b-func>s, known as the function symbols of λ;
- A recursive set P_λ of <b-pred>s known as the predicates of λ.
- A recursive set S_λ of <b-sort>s known as the sorts of λ.

2.3 Grammars

Given the CPL lexicon λ, the grammar G_λ based on λ is given in the following BNF.

<con> ::= a member of C_λ
<var> ::= a member of V_λ
<func> ::= a member of F_λ

<pred> ::= a member of P_λ
<sort> ::= a member of S_λ
<term> ::= <atomterm> | <compterm>
<atomterm> ::= <var> | <con>
<compterm> ::= <func> (<term>{,<term>}*)
<sentence> ::= <command> | <sortdef> | <assignment> | <atomsent> |
    <boolsent> | <quantsent>
<command> ::= %{define-domain|import-domain}(<pred>{,<pred>}*)
<sortdef> ::= SORT <sort> = **{** [<con> {, <con>*}]} **}**[2]
<assignment> ::= <compterm>={<con> | <doc-string> | <integer>}
<atomsent> ::= <pred> (<term>{,<term>}*)
<boolsent> ::= not (<sentence>) | and (<sentence> <sentence>+) |
    or (<sentence> <sentence>+) | => (<sentence><sentence>+)
<quantsent> ::= <forall|exists> {<var> |(<var>+)} (<sentence>)

3 Core Language

The core Common Process Language L_λ based on a lexicon λ is the set of all expressions that can be derived from the nonterminal <sentence> in the grammar G_λ. The members of L_λ will also be called the sentences of L_λ. Given these basic sentence types we will present the lexical elements from S_λ, P_λ, F_λ under conceptual headings. Within the CPF, there are actually two separate categories of knowledge that can be expressed using CPL: process domains or individual processes. We will use the abbreviation CPD for Common Process Domains when we are referring to knowledge in the former category. Some elements of CPL are only appropriate for CPD knowledge while other parts are restricted only to individual process specifications. These restrictions will be pointed out in the description below. Knowledge specifications expressed in CPL are physically identified as files and thus we will use the term, file, to refer to an entire specification or set of sentences. Throughout this paper we will provide examples of CPL sentences which will be numbered and listed in bold text. Many of these examples are based on a simple house building domain which is summarised in Section 5.

3.1 Commands

All CPL commands, <command>, start with a percentage sign. These are used to tell a CPL compiler various things about a domain or process specification. Currently, the only commands defined involve domain definition (%define-domain) which can only be used in a CPD file and domain dependencies (%import-domain) which can be used to express either the link between an individual process file and its domain(s) or (when expressed in a CPD) to create a lattice of domain dependencies.

For example, the first sentence below states that the content of a CPD file can be referred to as the "three_pigs_building" domain. The second sentence would be used in an individual process file to state that a particular process specification applies to the "three_pigs_building" domain. The third sentence might be used to indicate a dependency between some domain and a generic domain which contains a set of building objects.

%define-domain(three_pigs_building) (1)

%import-domain(three_pigs_building) (2)

%import-domain(general_building_objects) (3)

[2] Note that the bolded {} are actually terminals in this expression, so SORT cpo-action={A1,A2} is legal, and SORT cpo-action=A1,A2 is illegal.

3.2 Sort Definitions

The CPL is strongly typed. The ontology on which CPL is based, CPO [Polyak & Tate 98a], specifies the sorts of function and predicate parameters as well as the sort of the result in the case of functions. A sort definition sentence, <sortdef>, is used to associate a <con> or a set of <con>s with a particular <sort>.

Currently, only a (possibly empty) list of named symbols or <con>s can be specified although future versions of CPL may provide syntactic sugar for expressing ranges of <con>s more succinctly. For example, sentence 4 states that A1, A2 and A3 are of the sort "cpo-action" while 5 illustrates a shorthand which will be supported in the future.

SORT cpo-action={A1,A2,A3} (4)

(future) SORT cpo-action={A1-A3} (5)

3.3 Assignments

In CPL, the way to indicate the result of evaluating a function on a domain element (or a specific tuple of domain elements) is to represent it as an assignment, <assignment>. An assignment is given by a function term with parameters, an assignment operator "=", and the value it should be mapped to. So, for example, we may wish to state that the result of evaluating the function, "activity.begin-timepoint", on the domain element A1 (which is of type "cpo-action", from above) results in the element Tp1, which is defined to be a "cpo-timepoint".

activity.begin-timepoint(A1)=Tp1 (6)

3.4 Extensions

The relationship between the CPO and CPL is completely transparent. Classes in CPO correspond to <sort>s in CPL and the functions and relations from CPO are directly tied to <func>s and <pred>s in CPL. In general, extensions to CPL must be mirrored by and tied to ontological extensions to CPO. There are a couple of very important exceptions to this rule though. The first exception includes those extensions which are only concerned with extended grammar specifications. As we will see, grammar plug-ins are required to define the format of various CPL constraint expressions for a particular application of CPL. The definition of these extensions are examined in Section 6.

Another exception, permits simple generalisations/specialisations to be declared in a file (without being required to be linked to an external ontology or ontological extension). This can be declared using a simple "entity.isa" assertion in a file. Syntactically this simply relates two <doc-strings>, but it is meant to denote the implied sub-sort order relation. For example, given a three pigs building domain, we may wish to simply add a special activity-relatable object sort called "pigs-object-material" which will be used to define three material objects which will represent a store of straw, sticks, and bricks in the domain.

SORT pigs-object-material={Mat1,Mat2,Mat3} (7)

entity.isa("pigs-object-material","cpo-activity-relatable-object") (8)

object.name(Mat1)="straw" (9)

object.name(Mat2)="sticks" (10)

object.name(Mat3)="bricks" (11)

3.5 Process

The first set of CPL constructs (i.e. grouping of sorts, functions, relations) we will present are those related to process. A process sort in CPL is identified by cpo-process.

$$\textbf{SORT cpo-process=\{P1\}} \tag{12}$$

A process has a start timepoint and a finish timepoint.

$$\textbf{process.start-timepoint(P1)=Tp1} \tag{13}$$
$$\textbf{process.finish-timepoint(P1)=Tp2} \tag{14}$$

A process can be associated with an activity specification which defines its "space of behaviour". Activity specifications are examined in more detail below.

$$\textbf{process.activity-spec(P1)=As1} \tag{15}$$

A process may have a pattern which can be unified with an abstract action pattern to form a decompositional link. For example, the pattern specified in 16 unifies with the pattern specified in 17 which means that P1 is a potential expansion for Act1. A process may be specified to expand a particular action. For example, sentence 18 states that P1 does indeed expand Act1.

$$\textbf{process.pattern(P1)=``purchase bricks''} \tag{16}$$
$$\textbf{activity.pattern(Act1)=``purchase ?material''} \tag{17}$$
$$\textbf{process.expands(P1)=Act1} \tag{18}$$

Some processes are considered to be plans which will be identified by a different sort, cpo-plan. Plans carry the additional constraint of being designed for some objectives. This means that a plan will be associated with an objective specification as in sentence 20.

$$\textbf{SORT cpo-plan=\{Pl1\}} \tag{19}$$
$$\textbf{plan.objective-spec(Pl1)=Os1} \tag{20}$$

3.6 Nodes and Activities

The activity specification linked to a process/plan will typically have constraints which state that certain nodes are to be included (or excluded) from a process/plan. For the most part, these nodes will denote actions or events (which are specialisations of cpo-activity). For example, the action introduced in 17 might be declared with

$$\textbf{SORT cpo-action=\{Act1\}} \tag{21}$$

In addition to the pattern introduced in 17, an activity has a counterpart to the process.expands predicate (e.g. 18). This predicate is used to simply state the expansion relationship from the other direction

$$\textbf{node.expansion(Act1)=P1} \tag{22}$$

Activities, as with processes or plans, are bounded by two timepoints.

$$\textbf{activity.begin-timepoint(Act1)=Tp3} \tag{23}$$
$$\textbf{activity.end-timepoint(Act1)=Tp4} \tag{24}$$

In addition to nodes which represent activity, there are other nodes types which may be declared to help provide structure to the process definition. The most common class of these nodes are the start/finish and begin/end nodes. These are sometimes referred to as "dummy" nodes indicating that they do not denote activity but rather provide structure. The following statements declare particular structuring nodes which may be included in an activity specification

$$\textbf{SORT cpo-start=\{Start1\}} \tag{25}$$
$$\textbf{SORT cpo-finish=\{Finish1\}} \tag{26}$$
$$\textbf{SORT cpo-begin=\{Beg1\}} \tag{27}$$
$$\textbf{SORT cpo-end=\{End1\}} \tag{28}$$

All of these node types may be associated with a single timepoint.

$$\textbf{start.timepoint(Start1)=Tp1} \tag{29}$$
$$\textbf{start.timepoint(Beg1)=Tp3} \tag{30}$$
$$\textbf{finish.timepoint(Finish1)=Tp2} \tag{31}$$
$$\textbf{finish.timepoint(End1)=Tp4} \tag{32}$$

The other nodes type can be extended to provide common process structuring elements such as split/join nodes, and/or, iteration, etc.

3.7 Activity-Relatable Objects and Agents

Various objects may be involved in or related to process activities. These objects might be employed in various roles. For example, one role might be informally referred to as resource. A resource could be thought of as some object required in order to perform an activity. The objects introduced in 7 might be used in this role for a building activity. In general, these objects may be introduced with

$$\textbf{SORT cpo-activity-relatable-object=\{Aro1\}} \tag{33}$$

Note that object sort instances can be labelled in order to provide human-readable labels which differentiate their use in the domain (see sentences 9 - 11). These objects may also be produced, modified, destroyed, etc. Constraints in a process' activity specification associate these objects with activities and also indicate the role they are playing.

A special activity-relatable object is distinguished in CPL. This object represents agents who can perform behaviour, hold purposes, and have capabilities. These objects can be associated with the agent sort

$$\textbf{SORT cpo-agent=\{Agt1\}} \tag{34}$$

The performs relation for an agent can be expressed as a constraint which is included in an activity specification. This is discussed in the CPL constraint section. The purpose-holding relation on the other hand can be directly assigned between some objective and an agent. This

purpose-holding definition may be expressed as a requirement (hard constraint) or preference (soft constraint).

$$\textbf{agent.has-requirement(Agt1,Obj1)} \quad (35)$$
$$\textbf{agent.has-preference(Agt1,Obj2)} \quad (36)$$

Capabilities may also be directly assigned to agents. The expression of the capabilities is discussed in the constraints section as well.

$$\textbf{agent.has-capability(Agt1,Cap1)} \quad (37)$$

3.8 Timepoints

In CPL, the concept of time is approached from a timepoint-based perspective. A cpo-timepoint is an entity that represents a specific, instantaneous point along a timeline which is an infinite sequence of timepoints.

$$\textbf{SORT cpo-timepoint=\{Tp1,Tp2,Tp3,Tp4\}} \quad (38)$$

Timepoints may be associated with processes or nodes as illustrated in: 13, 14, 23, 24. The points may be related with ordering constraints which are discussed in Section 4.3. Pairs of these timepoints delineate process and activity intervals. In [Polyak 98b] we discuss the mapping of timepoint-based constraints into interval relationships.

3.9 Sets

A specification is a fundamental CPL structure which is used to express process design information. Generically speaking, the CPL definition of a specification is simply some set of constraints. When the set of constraints are activity constraints we, call the specification a cpo-activity-specification. When the set of constraints are objective constraints, we call the specification a cpo-objective-specification.

$$\textbf{SORT cpo-activity-specification=\{As1\}} \quad (39)$$
$$\textbf{SORT cpo-objective-specification=\{Os1\}} \quad (40)$$

The CPL core supports the most basic set operation which permits a constraint to be specified as a member of the set. For example, 41 illustrates an include-node constraint being added to an activity specification and 42 illustrates an objective being added to an objective specification.

$$\textbf{member(Ic1,As1)} \quad (41)$$
$$\textbf{member(Obj1,Os1)} \quad (42)$$

3.10 Domain Levels

When CPL is being used to describe a domain (i.e. a CPD file) it is often "good practice" to associate a process with a particular domain level. These level considerations encourage domain modellers to be consistent with their use of domain elements at varying degrees of generality or specificity [Polyak 99, Tate *et al.* 98].

CPL permits the declaration of domain levels, along with meaningful labels to identify the role of the level, and a numerical value for hierarchically ordering levels.

$$\textbf{SORT cpo-domain-level=\{D1,D2\}} \quad (43)$$
$$\textbf{domain-level.label(D1)=“Model house level”} \quad (44)$$
$$\textbf{domain-level.label(D2)=“Primitive building level”} \quad (45)$$
$$\textbf{domain-level.number(D1)=1} \quad (46)$$
$$\textbf{domain-level.number(D2)=2} \quad (47)$$

Processes may then be related to a particular domain level.

$$\textbf{domain-level.contains(D1,P1)} \quad (48)$$
$$(49)$$

4 Core Language - Constraints

In this section, we describe various categories of constraints which may be placed between CPO objects. These constraint types are based on the <I-N-OVA> model [Tate 96a, Tate 95] and Tate's plan ontology [Tate 96b]. Tate describes a constraint as “a relationship which expresses an assertion that can be evaluated with respect to a given plan as something that may hold and can be elaborated in some language” [Tate 95].

In order to support a range of constraints we present a flexible “plug-in” syntax method for constraint expressions which is similar to the method described in the SPAR approach [Tate 98]. We describe some default syntax specifications for most of the constraint types, but these may be modified for a particular use.

$$\textbf{constraint.expression(Oc1)=“...”} \quad (50)$$

There is typically a need to recognise which agent added a specific constraint during a design process. At a high-level, we can relate these entities using

$$\textbf{constraint.added-by(Oc1)=Agt1} \quad (51)$$

As we saw in 35 and 36, constraints may either be labelled as soft or hard depending on the type of purpose held by an agent.

$$\textbf{constraint.soft-hard-information(Oc1)=hard} \quad (52)$$

For each constraint type, we will present examples along with a default grammar for expressing the constraint information.

4.1 Issue Constraints

An issue is an outstanding aim, preference, task, flaw or other issue which remains to be addressed by the process. Issues provide implied constraints on the real world behaviour specified by the process. The default expression of issue constraints will be defined by a verb, zero or more noun phrases and zero, one or more qualifiers. The initial set of issues may be populated by the objectives set for a plan. The set of issues may expand or shrink throughout the design process. CPL currently considers the expression of objectives and issues to be defined in the same way.

$$\textbf{SORT cpo-objective-constraint=\{Objc1\}} \quad (53)$$
$$\textbf{SORT cpo-issue-constraint=\{Is1\}} \quad (54)$$
$$\textbf{constraint.expression(Objc1)=“expand Act1”} \quad (55)$$
$$\textbf{constraint.expression(Is1)=“expand Act1”} \quad (56)$$

The default definition of an issue constraint expression is

```
<issue-expression> ::= <rtq-sent> | <rt-sent> | <r-sent>
        <rtq-sent> ::= <issue-relconst> <term>+ <boolsent>
         <rt-sent> ::= <issue-relconst> <term>+
          <r-sent> ::= <issue-relconst> <term>*
  <issue-relconst> ::= achieve | expand | add | resolve
```

4.2 Node Constraints

Node constraints form the backbone of a process design. They define the space of behaviour upon which other constraints seek to refine. The primary purpose of these constraints are to specify which actions are to be included in a process. This constraint type is so common that CPL uses a built-in relation as opposed to a plug-in expression type.

$$\textbf{SORT cpo-include-constraint=\{Inc1\}} \quad (57)$$
$$\textbf{include-node(Inc1)=Act1} \quad (58)$$

Node inclusion is complemented by its counterpart constraint of node exclusion.

$$\textbf{SORT cpo-not-include-constraint=\{Inc1\}} \quad (59)$$
$$\textbf{not-include-node(Inc1)=Act1} \quad (60)$$

Both the node inclusion and node exclusion relations have unique forms which allow them to refer to an entire sort. This is convenient for saying things like, “no transportation action can be included” or “include any drilling action” or even something as extreme as “do nothing”.

$$\textbf{not-include-node(Inc1)=“transport-action”} \quad (61)$$
$$\textbf{include-node(Inc1)=“manu-drilling-action”} \quad (62)$$
$$\textbf{not-include-node(Inc1)=“cpo-action”} \quad (63)$$

4.3 Ordering Constraints

Ordering constraints may be placed between timepoints in order to define the process temporal structure. The default set of ordering constraint expressions include those which state that one timepoint is before another or that two are equal.

$$\textbf{SORT cpo-ordering-constraint=\{Oc1,Oc2\}} \quad (64)$$
$$\textbf{constraint.expression(Oc1)=“before(Tp1,Tp2)”} \quad (65)$$
$$\textbf{constraint.expression(Oc2)=“equal(Tp2,Tp4)”} \quad (66)$$

The default definition of an ordering expression is

```
<ordering-expression> ::= <before-sent> | <equal-sent>
        <before-sent> ::= before(<con>,<con>)
         <equal-sent> ::= equal(<con>,<con>)
```

4.4 Variable Constraints

Co-designation and non-co-designation constraints between variables relate activity relatable objects in the domain and are quite common in plan and process specifications. These variable constraints limit the range of values which may be assigned to particular variables in CPO expressions. For example, some activity labelled "replace drill bit" may be defined with a pattern "replace-drill-bit ?Old ?New". The specification of this activity may include a variable constraint which has an expression that specifies that the old bit cannot be the new bit.

$$\textbf{SORT cpo-variable-constraint=\{Vc1\}} \tag{67}$$

$$\textbf{constraint.expression(Vc1)="not-equal-var(?Old,?New)"} \tag{68}$$

The default definition of an ordering expression is

```
          <varc-expression> ::= <equal-var-sent> | <not-equal-var-sent> |
                                <equal-vartype-sent> | <not-equal-vartype-sent>
           <equal-var-sent> ::= equal-var(<?var>,{<term>|<doc-string>|<number>})
       <not-equal-var-sent> ::= not-equal-var(<?var>,{<term>|<doc-string>|<number>})
       <equal-vartype-sent> ::= equal-vartype(<?var>,<doc-string>)
   <not-equal-vartype-sent> ::= not-equal-vartype(<?var>,<doc-string>)
```

Note that this grammar specification is very flexible. While the first parameter in a variable expression is required to be a variable, the second parameter permits several syntactic categories. The second parameter may be a variable, as in the case described above. Other examples include the ability to constraint a variable to be equal or not equal to a certain value (e.g. string or number) or atomic or complex term. The vartype forms allow constraints to be placed on the possible range of values for a variable.

4.5 Auxiliary Constraints

The auxiliary constraints represent a broad category of constraint types which can be used to detail the design of processes and plans. In this section, we present the current set of core auxiliary constraints which have been defined for CPL.

The first two types provide generic hooks for expressing conditions and effects which may be associated with processes and activities. These are referred to as input and output constraints.

$$\textbf{SORT cpo-input-constraint=\{Ic1\}} \tag{69}$$

$$\textbf{SORT cpo-output-constraint=\{Oc1\}} \tag{70}$$

Input and output constraints are used to connect behaviour and state. The expression of both types of constraints is referred to as a world-state-expression. The default approach for representing this knowledge involves stating (pattern)=(value) associations. For example, the process P1 described in 16 may have an activity specification which includes some primitive action for buying a supply of bricks. We may wish to state that a condition on the performance of this action is that a supply of money is available. So, we might include an input-constraint in P1's activity specification which has the following expression

$$\textbf{constraint.expression(Ic1)="unsupervised \{have money\}=true at Tp8"} \tag{71}$$

The default definition of world-state-expressions is based on the Task Formalism (TF) [Tate *et al.* 94]. Note that TF allows for typing of these expressions (e.g. unsupervised, supervised, etc.). The following grammar structures these expressions

```
<world-state-expression> ::= [<ws-type>] <lbrace> <term>+ <rbrace>
                             [= {true|false|<term>+}]
                             at <term>
               <ws-type> ::= supervised | achieve | unsupervised |
                             only_use_if | only_use_for_query
                <lbrace> ::= {
                <rbrace> ::= }
```

While input and output constraints can be used to associate state assertions at particular points in time there are also cases where we may wish to make some assertion apply for an entire domain (i.e. holds for or is automatically included in any domain activity specification). This type of constraint is referred to in CPL as an "always" constraint, as it is in TF. For example, we may assign a wolf-proof property to bricks in the three pigs domain.

$$\textbf{SORT cpo-always-constraint=\{Ac1\}} \tag{72}$$

$$\textbf{constraint.expression(Ac1)="\{proof_against wolf bricks\}=true"} \tag{73}$$

The default grammar of an always constraint expression is similar to the world-state-expression defined above with the exception that no <ws-type> or {at <term>} may be used.

The resource constraints can be used to describe an activities required allocation of resource objects, producible/consumable resource effects, etc. While it is possible to lump resource constraints into the general notion of input and output constraints it is beneficial to separate them out as many tools are geared toward working with this knowledge (e.g. scheduling tools, etc.) For this purpose, CPL has a defined resource-constraint. For example, we may wish to specify that the conclusion of a purchase brick action entails 50 pounds (money) to be consumed.

$$\textbf{SORT cpo-resource-constraint=\{Rc1\}} \tag{74}$$

$$\textbf{constraint.expression(Ac1)="consumes \{resource money\} = 50 pounds"} \tag{75}$$

The default definition of a resource utilisation expression is

```
<resource-expression> ::= <res-type> <lbrace> resource <term>+ <rbrace>
                          [= {true|false|<number>} term*]
                          at <term>
            <ws-type> ::= consumes | produces | uses
             <lbrace> ::= {
             <rbrace> ::= }
```

Finally, a core auxiliary constraint may be utilised for attaching annotations or documentation to the process artifact. This could also be used to provide links to non-textual or external data related to a process such as CAD and multimedia filenames, web site addresses, or printed policy/standards document references.

$$\textbf{SORT cpo-annotation-constraint=\{Anc1\}} \tag{76}$$

5 Example: Three Pigs Building

The examples in this paper are based on a demonstration building scenario. This building domain is similar to the Task Formalism 3 pigs domain which was created for the O-Plan planner [Currie & Tate 91]. The only task in the domain is concerned with building a house for a pig. As in the TF domain, the main building materials involve straw, sticks, and bricks which each cost different amounts of money. There are also costs for performing the activities and for other house materials such as windows and doors.

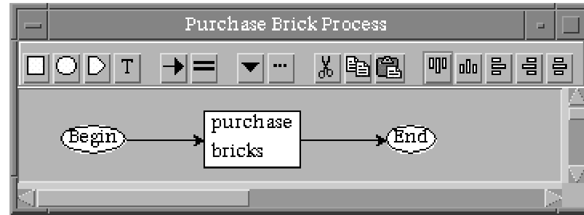

Figure 1: Simple CPL process example

In order to provide a detailed example of a CPL process specification which utilises CPO terms and concepts, we will restrict the content to a rather simplified process. The example "Purchase Brick Process" is part of the larger 3 pigs building domain and represents a particular transaction activity whereby money is consumed to acquire some supply of brick building material. As we can see in Figure 1, it is bounded by a begin/end node pairing and contains only one action, "purchase bricks". Figure 2 provides an example CPL expression of this process.

```
%define-domain{my-building}
SORT cpo-action={A1}
SORT cpo-activity-specification={As1}
SORT cpo-begin={B1}
SORT cpo-end={E1}
SORT cpo-include-constraint={Ic1-Ic3}
SORT cpo-ordering-constraint={Or1,Or2}
SORT cpo-output-constraint={Oc1}
SORT cpo-process={P1}
SORT cpo-resource-constraint={Rc1}
SORT cpo-timepoint={Tp1-TP4}

label(P1)="Purchase Brick Process"
start-timepoint(P1)=Tp1
finish-timepoint(P1)=Tp4
pattern(P1)="{purchase bricks}"
label(B1)="begin"
timepoint(B1)=Tp1
include-node(Ic1)=B1
member(Ic1,As1)
label(E1)="end"
timepoint(E1)=TP4
include-node(Ic2)=E1
member(Ic2,As1)
label(A1)="purchase bricks"
begin-timepoint(A1)=Tp2
end-timepoint(A1)=Tp3
include-node(Ic3)=A1
member(Ic3,As1)
expression(Or1)="before(Tp1,Tp2)"
member(Or1,As1)
expression(Or2)="before(Tp3,Tp4)"
member(Or2,As1)
expression(Oc1)="{have bricks} at A1"
member(Oc1,As1)
expression(Rc1)="consumes
  {resource money} = 50 pounds at A1"
member(Rc1,As1)
```

Figure 2: CPL expression of the purchase brick process

6 Extensions

6.1 Tool-Based

CPO provides a core set of concepts which may be extended to capture specialised process-related knowledge. One class of extensions can be considered to be tool-specific. Tool-specific extensions are used to express new or specialised sorts or relations which address aspects linked to a particular tool's ontology. Two examples are provided here for extensions related to O-Plan's TF and the process/domain editors in CPF.

O-Plan's Task Formalism language [Tate *et al.* 94] encompasses a large set of terms and concepts for expressing plan/process domain knowledge. For this particular TF extension example though, we are simply interested in providing additional support for capturing TF resource-related information. One facet of this information is "resource units". Resource unit statements in TF are used to define unit types for resources such as person/people, gallons, kilograms, etc. These units have their own properties in TF (e.g. type, which could have the values: count; size; weight; or set).

In the TF extension, we define a new sort called resource unit. Two new functions are designated for this sort to express both its label (e.g. pounds) and its type (e.g. count).

SORT tf-resource-unit={Ru1} (77)

ru-label(Ru1)="pounds" (78)

ru-type(Ru1)="count" (79)

The ru-label can be any <doc-string> but the ru-type expression above is syntactically constrained to {count|size|weight|set}. In addition to this, we need to add functions and a relation to the activity-relatable object sort. In particular, we need to be able to express whether an ARO is going to play the role of a TF resource and if so, what its TF resource type is

is-resource(Aro1) (80)

resource-type(Aro1)="consumable_strictly" (81)

unit(Aro1,Ru1) (82)

The resource-type expression above is syntactically constrained to the following forms which are defined in the TF manual.

> {consumable_strictly | consumable_producible_by_agent |
> consumable_producible_outwith_agent | consumable_producible_by_and_outwith_agent |
> reusable_non_sharable | reusable_sharable_independently | sharable_synchronously}

Some tool-specific extensions are related to presentation information or internal state information (e.g. nodes selected, etc.) associated with processes. In both the Common Domain Editor (CDE) and the Common Process Editor (CPE) in CPF, process presentation information is attached to various parts of the domain specification. The CPF tools extension defines additional support for this such as

cpf-proc-xpos(P1)=10

cpf-proc-ypos(P1)=10

cpf-proc-width(P1)=400

cpf-proc-height(P1)=400

cpf-proc-label(P1)="Purchase Brick Process"

$$\textbf{cpf-node-xpos(Act1)=10}$$
$$\textbf{cpf-node-ypos(Act1)=10}$$
$$\textbf{cpf-node-type(Act1)="Act"}$$
$$\textbf{cpf-node-status(Act1)=0}$$
$$\textbf{cpf-node-label="purchase bricks"}$$
$$\textbf{cpf-ann-xpos(Anc1)=20}$$
$$\textbf{cpf-ann-ypos(Anc1)=20}$$
$$\textbf{cpf-proc-top-level(P1)}$$
$$\textbf{cpf-node-selected(Act2)}$$

The cpf-node-type may be {Act|Event|Special} which indicates its presentation style. The node status can be used to attach an executability status to nodes. The cpf-node-status can be {}.

6.2 Rationale

While the extensions discussed in the previous section were labelled tool-specific, we can also have extensions which are tool-independent, or more appropriately, concept-specific. Concept-specific extensions provide terms and definitions which are centred around a general set of closely associated entities and relations. One example of such an extension is the rationale extension we have developed for CPO.

In our work with plan rationale [Polyak & Tate 98b], we explored the epistemological nature of this category of knowledge and described it from the perspectives of dependencies, causal relationships and decisions. While there has been much work done on both plan/process causality and dependencies, there has been correspondingly less research into plan decision rationale.

We proposed a "design space analysis (DSA)" approach to plan decision rationale [Polyak 98a] which was based on research from the design rationale (DR) field. If we envision the <I-N-OVA> approach, which CPO has adopted, as describing a "space of behaviour" we can also consider a "space of decisions" which is navigated in creating this behavioural specification. It is possible then to augment a process description with the rationale that went into designing this artifact.

Both CPE and CDE support this DSA approach (i.e. provide graphical editing of a DSA) and rely on this CPF rationale extension to define the DSA terms and concepts which are expressed in CPL. In this extension, we refer to an entity called a decision rationale which represents the overall "decision space" for a process design. In the CPO core, an activity specification groups the constraints which form the "space of behaviour". Analogously, a rationale specification groups the constraints which form the "space of decisions". So, the CPF rationale extension includes

$$\textbf{SORT dsa-decision-rationale=\{Dr1\}} \tag{83}$$
$$\textbf{SORT dsa-rationale-specification=\{Rs1\}} \tag{84}$$
$$\textbf{dr.rationale-spec(Dr1)=Rs1} \tag{85}$$
$$\textbf{process.decision-rationale(P1)=Dr1} \tag{86}$$

While a plan is described in Tate's plan ontology as "a set of constraints on the relationships between agents, their purposes and their behaviour" a decision rationale can be viewed as "a set of constraints on the relationships between questions (or design issues), options (or answers to these questions), and evaluative criteria. The CPF rationale extension includes the sorts for questions, options and criteria.

Questions pose key issues for structuring the space of alternatives (options). The role of questions is to define local contexts in a design space which help to ensure that certain options are compared with each other. Criteria represent the desirable properties of the process and requirements that it must satisfy. They form the basis against which to evaluate the options. These

elements can be included into a rationale specification and interrelated via a set of defined constraints which represent relationships.

$$\textbf{SORT dsa-question=\{Q1,Q2\}} \tag{87}$$
$$\textbf{SORT dsa-option=\{Opt1,Opt2\}} \tag{88}$$
$$\textbf{SORT dsa-criteria=\{Crt1\}} \tag{89}$$
$$\textbf{dsa-has-option(Q1,Opt1)} \tag{90}$$
$$\textbf{dsa-has-option(Q1,Opt2)} \tag{91}$$
$$\textbf{dsa-selected(Opt1)} \tag{92}$$
$$\textbf{dsa-supports(Crt1,Opt1)} \tag{93}$$
$$\textbf{dsa-detracts(Crt1,Opt2)} \tag{94}$$
$$\textbf{dsa-sub-question(Opt1,Q2)} \tag{95}$$
$$\tag{96}$$

Acknowledgements

The author is sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-96-1-0348 – an AASERT award monitored by Dr. Abe Waksman and associated with the O-Plan project F30602-97-1-0022. The U.S. Government is authorised to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either express or implied, of DARPA,AFOSR or the U.S. Government.

References

[Currie & Tate 91] K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[Polyak & Tate 98a] S. Polyak and A. Tate. A common process ontology for process-centred organisations. In E.A. Edmonds, editor, *Submitted to: Knowledge-Based Systems.* Elsevier Science, 1998.

[Polyak & Tate 98b] S. Polyak and A. Tate. Rationale in planning: Causality, dependencies, and decisions. *Knowledge Engineering Review*, 13(3):247–262, September 1998.

[Polyak 98a] S. Polyak. Applying design space analysis to planning. In *Proceedings of the AIPS-98 workshop on Knowledge Engineering and Acquisition for Planning: Bridging Theory and Practice AAAI Technical Report WS-98-03*, Carnegie-Mellon University, June 1998.

[Polyak 98b] S. Polyak. Mapping timepoint-based constraints into interval relationships. Department of Artificial Intelligence RP 932, University of Edinburgh, Edinburgh, Scotland, 1998.

[Polyak 99] S. Polyak. A common process methodology for engineering process domains. In D. Bustard, editor, *Submitted to: Systems Modelling for Business Process Improvement (SMBPI).* Artech House, 1999.

[Tate 95] A. Tate. Characterising plans as a set of constraints – the < I-N-OVA > model – a framework for comparitive analysis. *ACM Sigart Bulletin*, 6(1), January 1995.

[Tate 96a] A. Tate. Representing plans as a set of constraints – the < I-N-OVA > model. In B. Drabble, editor, *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 221–228, Edinburgh, Scotland, May 1996. Morgan Kaufmann.

[Tate 96b] A. Tate. Towards a plan ontology. *AI*IA Notiziqe (Publication of the Associazione Italiana per l'Intelligenza Artificiale), Special Issue on Aspects of Planning Research*, 9(1):19–26, 1996.

[Tate 98] A. Tate. Roots of SPAR - Shared Planning and Activity Representation. *The Knowledge Engineering Review*, 13(1):121–128, 1998.

[Tate *et al.* 94] A. Tate, B. Drabble, and J. Dalton. The Task Formalism Manual. Artificial Intelligence Applications Institute AIAI-TF-Manual, University of Edinburgh, Edinburgh, UK ftp://ftp.aiai.ed.ac.uk/pub/documents/ANY/oplan-tf-manual.ps.gz, 1994.

[Tate *et al.* 98] A. Tate, S. Polyak, and P. Jarvis. TF Method: An initial framework for modelling and analysing planning domains,. AIPS '98 Workshop on Knowledge Engineering and Acquisition for Planning: Bridging Theory and Practice AAAI Technical Report WS-98-03, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1998.