The development of OCPL, object conceptual prototyping language

A.E. James*, S.D.E. Shipley

School of Mathematics and Information Sciences, Coventry University, Priory Street, Coventry CV1 5FB, UK

Received 31 March 1999; revised 8 May 2000; accepted 25 May 2000

Abstract

This paper describes the development of OCPL (object conceptual prototyping language), an object–knowledge representation language. The language is based on CPL, conceptual prototyping language, developed at the Free University of Amsterdam. CPL has been extended to allow for the explicit representation of object-oriented constructs. These constructs include facilities for application system definition, generation and usage. A restricted use of the constraint model of CPL allows for systematic representation of events from which appropriate user interfaces can be generated. The paper describes OCPL and its relationship to CPL and related work. It also illustrates how the constraint model can be used to represent dynamics and provide intelligent user support.

Keywords: Object-oriented; Knowledge-based; Systems representation; Systems generation

1. Background

Object conceptual prototyping language (OCPL) is an object–knowledge representation language developed at Coventry University [9,12,13]. Its purpose is to represent both objects and knowledge within the same conceptual framework. It is based on a language called CPL, conceptual prototyping language, which was developed at the Free University of Amsterdam for representing knowledge [4]. CPL contains useful representation mechanisms for expressing modal logic (constraints that are necessarily true), deontic logic (constraints that ought to be true), dynamic logic (the interpretation of actions) and temporal logic. CPL is based on the linguistic theory of functional grammar [5] and uses the semantic basis of predicate calculus.

The problem with CPL is that it does not explicitly capture an object-oriented perspective of a domain of discourse. As a language that was developed with knowledge representation in mind, the problem of representing large amounts of similarly structured data was not a major priority. It is rich in constructs and very powerful in terms of knowledge definition. It allows the expression of practically anything that can be expressed in natural language. However, it lacks a structure that facilitates efficient systems definition and easy translation to an implementable system. Furthermore, the definition of a typical system in CPL is usually very verbose with much repetition of key words and structures. It would be beneficial to have a language that allows for powerful, effective and efficient expression of both knowledge, which has the characteristics of diversity in structure but smallness in number, and objects, which typically are similar in structure but great in number. A solution to these issues is to introduce object-oriented abstractions into CPL [6]. The use of such abstractions alleviates the verbosity of CPL and allows for systems to be structured in a way that promotes good system qualities like maintainability, comprehensibility, reusability and systems generation. This was the main motivation in extending CPL to create OCPL.

CPL's developers see it primarily as a language that can be used for prototyping and which can be generated directly, with the help of a lexicon and text analyser, from system scenarios written in natural language. Since CPL’s development another project has been undertaken by the originators, called COLOR-X, which stands for conceptual linguistically based object-oriented representation language for information and communication systems [2,3]. COLOR-X remedies some of the problems of CPL, in that an object model graphical layer has now been included, from which CPL can be generated. The object model used is based on OMT [11] but does not make use of the concept of operations, nor does it allow link attributes, complex objects (through composition) or the possibility to model associations as classes. OCPL’s object model which was developed independently differs in that the definition of operations and the possibility that classes should be able to have other classes as attributes are seen as very important features of
the approach. COLOR-X includes a separate event model. In OCPL, events are seen as special types of operation which may be defined in terms of lower level operations and which are associated with special objects known as system objects. Thus they are defined within the same model. We believe that this is a better approach, as events can be elegantly defined as and in terms of specified operations. An important philosophical difference in the two approaches lies in the fact that the COLOR-X project sees the tools mainly as an aid for analysing and verifying natural language definition and thus producing a requirements specification. We see OCPL as a definition language, which will not be directly generated from natural language but will allow analysts to unambiguously and naturally define and implement knowledge-based systems.

As well as the CPL and COLOR-X projects, there has been much work in requirements and system specification that address some of the aspects addressed by this work. For instance, there is the well-known UML [1] approach which covers object-oriented modelling and systems definition. Good examples of rich approaches that cover similar areas are TEMPORA [10] and SOMA/SOMATIK [8]. The main ways in which OCPL differs from these approaches are firstly in its basis in functional grammar and approach to constraint modelling and secondly in its ability to support and integrate an object model and systems model definition thus allowing for simple and seamless application generation. Both of these aspects provide a more powerful means of knowledge representation than other approaches. At the end of this paper, in the discussion, more detail is given on the relationship between OCPL and the other work.

The paper is organised as follows. Section 2 provides an overview of CPL. Section 3 describes OCPL’s constructs and illustrates how an OCPL model is defined. Section 4 describes OCPL’s manipulation language and Section 5 discusses how an interface can be generated from an OCPL definition. Section 6 shows how knowledge defined in the model can be used to support user interaction with the generated system. Section 7 provides a discussion and some conclusions.

2. CPL

In order to describe the structure of OCPL, let us first consider the basic structure of CPL, as this is the language on which OCPL is based.

2.1. A CPL specification

A CPL specification has the following structure:

\[(\text{destination-clause-set})(\text{situation-clause-set})\]

The \textit{situation-clause-set} is optional. A specification with just a \textit{destination-clause-set} represents a fact or facts. A specification with a \textit{situation-clause-set} represents a constraint or rule. The \textit{situation-clause-set} is made up of one or more situation clauses \textit{(sit:…)} which describe a situation under which the destination predication set holds. It is an essential part of constraint expression. Situation clauses can be joined by the logical connectives \textit{and} and \textit{or} and by the sequence connective \textit{followed by}. Both destination and situation clauses break down to the following structure.

\[
\text{Modality: Tense; Predication} \\
(id:…)
\]

The identification clause \textit{(id:)} is optional and can occur more than once in a specification.

Examples of both a CPL fact and a CPL constraint are given below.

FACTUAL: \textit{PERF: sell}(ag = S \text{ in salesperson})(\text{200}) \text{go = computer}(\text{tmp = T in date})

\textit{(id: T = “05-12-98”)(id: S = “Fred Smith”)}

The above expression states that the salesperson Fred Smith sold 200 computers on “05-12-98”.

MUST: \textit{ACTION}

\textit{pay}(ag = C \text{ in customer})(go = I \text{ in invoice})(\text{tmp = T2 in date})

\textit{(id: T2 <= T1 + 28)}

\textit{(sit: PERF: receive}(rec = C \text{ in customer})(go = I \text{ in invoice})(\text{tmp = T1 in date})

The above specification states that if a customer receives an invoice then, they must pay it within 28 days.

CPL is based on functional grammar [5], which uses the idea of semantic relations. A relation corresponds to a verb which has roles associated with it. For instance, in the above example \textit{pay} is a relation with roles \textit{agent}(ag), \textit{goal}(go), \textit{time}(tmp), and \textit{receive} is a relation with roles \textit{recipient}(rec), \textit{goal(go)} and \textit{time}(tmp). A relation thus holds over a set of terms which refer to objects playing particular roles. In the above example, \textit{pay} is a relation between the terms, \textit{C} referring to a particular customer, \textit{I} referring to a particular invoice and \textit{T2} referring to a particular time expressed in days. A central notion in functional grammar representations is that the semantics of an expression can be represented independently of its syntax. For instance ‘a customer has just bought a computer’ and ‘a computer has just been bought by a customer’ would both be expressed in CPL as

FACTUAL: \textit{DONE:buy}(ag = customer)(go = computer)

Notice therefore that the role \textit{agent} does not mean the same as the syntactic element \textit{subject}, rather it means the object doing the action. This characteristic is useful for applications of natural language processing.

Let us now consider some of the other features in a CPL specification. The central part of the CPL specification is the predication which comprises a relation over a set of relevant
terms as described above. The modality describes the type of the predication which can be FACTUAL; MUST; NEC; or PERMIT. A predication of type FACTUAL is simply a fact about the UoD that needs to be represented. A predication of type MUST is a deontic constraint and is used to express something that ought to happen but might not. A predication of type NEC is a necessary constraint and expresses something that is necessarily true within the domain of discourse or system being described. A predication of type PERMIT expresses a relation that is permissible within the system.

The tense of a predication can be one of PRET (past tense); PERF (perfect tense); DONE (perfect tense but action described was last action); ACTION (present tense) or PROSP (future). These tenses may be useful for natural language representation but we have found that for systems description, such a full tense classification is unnecessary. The identification clause \((id:\ldots)\) is used to further identify a term used in the predication. There can be more than one identification clause for each predication. An identification clause can take the form of a CPL clause or a relative expression. Consider the example below.

\[
\text{MUST: ACTION} \\
\text{pay}(ag = C \text{ in customer})(go = I \text{ in invoice}) \\
(id: \#T2 \leq T1 + 28) \\
(sit: \text{PERF}: \text{receive}(ag = C \text{ in customer}) \\
(go = I \text{ in invoice})(id: \text{PERF}: \text{send}(ag = A \text{ in invoice-team}) \\
(id: \text{has-a}(zero = A \text{ in invoice-team})(pat = B \text{ in name})) \\
(id: B = \text{“Computing 1”})
\]

The above specification states that if a customer receives an invoice from the “Computing 1” invoice-team, then that invoice must be paid within 28 days. Notice that the third \(id\) clause is a predication with relation \(\text{has-a}\). This is a special relation as discussed in Section 2.2.

2.2. Special relations

CPL has a number of special relations. These are predefined relations rather than user-defined relations and represent common relations such as \(\text{has-a}\). Zero is used to describe the use of a term with no semantic role in a relation. \(\text{Patient}(pat)\) is used for a term that takes a passive role. In the case of the \(\text{has}\) special relation the roles defined are therefore always \(\text{zero}\) and \(\text{pat}\). An example of this special relation is given above. Other special relations include \(\text{is-a}\) (for generalisation relationships), \(\text{exists}\) (to specify cardinalities) and \(\text{value-of}\) (to specify data types). For example to state that there are 100 salespersons, the following expression might be used.

\[
\text{exists}((100); \text{zero} = \text{salesperson})
\]

2.3. Events in CPL

Specifications that imply actions can be regarded as expressing events. Events can be logically combined into process models by using the connectives \text{and}, \text{or} and \text{followed by} to connect primitive actions in terms of choice, concurrency and sequence. For instance, consider the following to describe the circumstance under which the event of sending a first reminder letter occurs.

\[
\text{NEC: ACTION: send}(ag = \text{invoice-clerk}) \\
(id: \text{L in letter})(rec = \text{C in customer}) \\
(id: \text{D} = \text{‘first-reminder’}) \\
(sit: \text{receive}(rec = \text{C in customer}) \\
(go = I \text{ in invoice})(id: \text{TMP} = \text{T1 in date}) \\
\text{followed by} \\
(sit: \text{not pay}(ag = \text{C in customer}) \\
(go = I \text{ in invoice})(id: \text{TMP} = \text{T2 in date}) \\
(id: \text{T2} \leq \text{T1} + 28)
\]

3. OCPL

OCPL is an object-oriented adaptation of CPL based on a significant extension and amendment of the special relation set. In this section we therefore initially describe the extended special relation set and then illustrate how it can be used to model systems.

3.1. The special relation set

CPL does not explicitly have the facility for representing objects and their behaviour, nor does it have the facility for describing the systems in which the objects interact in an abstract way. In OCPL, we have developed the CPL special relation set to include special relations for defining objects and associated characteristics. OCPL includes the following special relations. The relations are divided into three groups: the first group defines the basic object structure; the second defines the object behaviour in terms of operations; and the third group defines the system of which the objects will be a part.

3.1.1. Definition of object structure

The following relations allow class composition and hierarchy to be defined together with classical attribute characteristics.

- \text{is-a-class(class-name)}
- \text{has-subpart(class-name)(attribute-name)}
- \text{has-attribute(class-name)(attribute-name)}
- \text{is-a-subclass(class-name)(class-name)}
- \text{has-type(class-name)(attribute-name)(value type)}
- \text{has-unique(class-name)(class-name or attribute-name)}
- \text{has-key(class-name)(class-name or attribute-name)}
3.1.2. Definition of object behaviour

The following special relations allow operations to be defined, together with their input and output parameters.

- `has-accessor-operation(class-name)(operation-name)`
- `has-transformer-operation(class-name)(operation-name)`
- `has-constructor-operation(class-name)(operation-name)`
- `has-input-parameter(class-name)(operation-name)`
- `has-output-parameter(class-name)(operation-name)`
- `is-whole-set(class-name)(operation-name)`
- `is-single-instance(class-name)(operation-name)`
- `is-run-time-restricted(class-name)(operation-name)`
- `has-description(class-name)(operation-name)`

3.1.3. Definition of systems

The following are special relations to allow for systems to be defined in terms of subsystems, interfaces, user-groups, users, objects and events. Square brackets indicate optionality. In order to allow for business process modelling at a high level, the event definition includes facilities to break events down into subevents or operations and also facilities to specify sequences, selections and iterations of events.

- `is-system(system-name)`
- `has-system-subpart(system-name)(class-name)`
- `is-subsystem(system-name)(system-name)`
- `has-event(system-name)(event-name)`
- `is-subevent(event-name)(event-name)`
- `is-selection(event-name)(condition clause)`
- `is-selection-part(selecting event name)(event-name or operation name)(triggering condition clause)`
- `is-iteration(event-name)(terminating condition clause)`
- `calls(event-name)(event-name or operation name)`

- `is-in-user-group(user-name)(user-group-name)`
- `may-access(user-group-name)(system-name)`
- `may-read-access(user-group-name)(system-name)`
- `may-fully-access(user-group-name)(system-name)`

3.2. Defining an OCPL model

3.2.1. Object structure

Object-classes are identified using an `is-a-class` relation. Class hierarchies are defined using an `is-a-subclass` relation. Class composition is represented with two special relations, `has-subpart` and `has-attribute`. `Has-subpart` is used when the component object is itself a complex object, `has-attribute` is used when the component object is a basic system-defined type (e.g. integer, character, etc.). Such system-defined types can be thought of as pre-defined basic building block classes. A `has-unique` relation is used to specify those attributes or subparts which uniquely identify an entity. A `has-key` relation is used to define for an object, a single user-defined key which may either be an attribute class or a subpart class. A `has-default` relation is used to allow defaults to be specified, e.g.

- `is-a-class(computer)`
- `is-a-subclass(micro-computer)(computer)`

- `has-subpart(micro-computer)(peripherals)`
- `has-attribute(micro-computer)(overall-weight)`
- `has-attribute(micro-computer)(year)`
- `has-unique(micro-computer)(serial-no)`
- `has-type(micro-computer)(year)(integer)`
- `has-default(micro-computer)(year)(1998)`

3.2.2. Object behaviour

Object behaviour is defined in terms of the operations that may be applied to particular object-classes. These can be divided into three groups: constructor (and destructor), accessor and transformer. Constructor operations create and delete objects, accessor operations simply retrieve information and transformer operations update objects. Operations will therefore be identified and attributed to classes using three relations: the `has-constructor-operation`, the `has-accessor-operation` and the `has-transformer-operation` relations, e.g.

- `has-accessor-operation(computer)(retrieve-components)`
- `has-transformer-operation(computer)(amend-details)`
- `has-constructor-operation(computer)`

Any operation needs input and output and so we have special relations `has-input-parameter` and `has-output-parameter` where the parameters specified may be a complex class or a basic system-defined class. Let us consider an
example. We might have an operation that costs parts within a particular buildsheet between two dates, e.g.

\[
\text{has-accessor-operation(buildsheet)(cost-parts)}
\]

\[
\text{has-input-parameter(buildsheet)(cost-parts)(buildsheet-no)}
\]

\[
\text{has-input-parameter(buildsheet)(cost-parts)(start-date)}
\]

\[
\text{has-input-parameter(buildsheet)(cost-parts)(end-date)}
\]

\[
\text{has-output-parameter(buildsheet)(cost-parts)(total-cost)}
\]

\[
\text{has-description(buildsheet)(cost-parts)("This operation costs parts within a particular buildsheet between two dates")}
\]

It is assumed that objects from the class itself will also be input to an operation. This input might be restricted certain instances, a single instance or might include the total population for the class. In order to distinguish between these types of operation, we have the relations \text{is-whole-set}, \text{is-single-instance} and \text{is-run-time-restricted}. Input and output classes need not always correspond to the basic object-classes defined. They may be specially defined classes for specific input and output. Such specially defined classes are described in greater detail in Section 3.2.3. Notice also that the underlying logic of the operations is not defined here. The OCPL definition provides just the structure of the knowledge base system. The procedural definition of the operations would be defined and eventually coded separately using an appropriate programming language. The \text{has-description} relation allows a description to be attached to an operation.

3.2.3. Definition of systems

The core of a system is the knowledge base defined in terms of objects and operations. A system also includes users and appropriate interfaces. Some of the interfaces may be defined and generated by the objects themselves. Others need to be defined as system components. For any system therefore, we will define the objects and operations and this will be our foundation. Apart from this, however, we will need to define users, system and subsystem entry and exit screens, user-tailored screens (as different users will have different access rights) and we need to define the system events which are made up of lower level event or object operations. Let’s consider an example. Assume we are modelling a computer configuration and sales business that is divided into three sections: sales, technical and administration. We might have an overall system called \text{Computer Configuration and Sales Business} and three subsystems called \text{Sales}, \text{Technical} and \text{Administration}. Each section would be concerned with a subset of the underlying knowledge base objects and a set of appropriate events. The sales section might have three teams, each with its own set of customers. Customers, who are grouped in terms of region, make orders for computers. The following definition would represent this situation.

\[
\text{is-class(customer)}
\]

\[
\text{has-attribute(customer)(region)}
\]

\[
\text{may-fully-access(sales-manager)(Sales)}
\]

\[
\text{may-access(sales-teamA)(Sales)(place-order)}
\]

\[
\text{may-fully-access(sales-teamA)(Sales)(order)(customer.region \^ North)}
\]

\[
\text{may-read-access(sales-teamA)(Sales)(customer)(customer.region = North)}
\]

We may wish to define screens to be used for input and output to operations. These will be defined as virtual object-classes as they constitute interfaces to the underlying data and do not represent actual objects that exist in the database. The relation \text{is-virtual-class} will be used for this purpose. Virtual classes can then be linked to operations using the \text{has-input-parameter} and \text{has-output-parameter} relation. Consider the example given earlier regarding the \text{cost-parts} operation for the class \text{buildsheet}. This might now be defined as follows.

\[
\text{is-system(Computer Configuration and Sales Business)}
\]

\[
\text{is-subsystem(Sales)(Computer Configuration and Sales Business)}
\]

\[
\text{is-subsystem(Technical)(Computer Configuration and Sales Business)}
\]

\[
\text{is-subsystem(Administration)(Computer Configuration and Sales Business)}
\]

\[
\text{is-event(Sales)(place-order)}
\]

\[
\text{is-user-group(sales-manager)}
\]

\[
\text{is-user-group(sales-teamA)}
\]

\[
\text{is-user-group(sales-teamB)}
\]

\[
\text{is-user-group(sales-teamC)}
\]

\[
\text{is-class(customer)}
\]

\[
\text{has-attribute(customer)(region)}
\]

\[
\text{may-fully-access(sales-manager)(Sales)}
\]

\[
\text{may-access(sales-teamA)(Sales)(place-order)}
\]

\[
\text{may-fully-access(sales-teamA)(Sales)(order)(customer.region \^ North)}
\]

\[
\text{may-read-access(sales-teamA)(Sales)(customer)(customer.region = North)}
\]

The advantage of using virtual classes, which may be made up of many attributes or subparts, rather than just listing attributes as parameters is that it allows more structure to be defined to the input and output required of operations. This is helpful to the users in terms of analysis and provides the system greater scope for generating user-friendly screen layouts.
3.2.4. Representing dynamic knowledge in OCPL for user support

So far the description given of OCPL does not look much like CPL aside from the use of special relations. Central to CPL is the definition of constraints. As well as the special relations, OCPL includes a subset of the CPL language that allows for constraints to be specified. OCPL uses the MUST and NEC modality from the CPL language for expressing constraints. PERMIT is not necessary because of the use of special relations for defining access rights and the association of operations to object-classes. Modality in general is not considered to be necessary for the special relations because the meaning of each of these is implicitly NEC as far as the system definition is concerned.

For the expression of dynamic knowledge MUST and NEC can be used to build up event chains in terms of processes as shown below. The purpose of representing event chains is two-fold. On the one hand, the representation can serve for good documentation of the semantics of the application. On the other hand the representation can be used to provide automatic user support. Depending on the purpose, different OCPL definitions may be used. Let's consider an event chain. Let us suppose that if a customer places an order and that order is filled, then the order should be despatched within a week and the customer should receive an invoice within a fortnight. That invoice should be paid within 28 days and if it is not, the customer receives a first reminder letter. Given a free OCPL hand, the event chain might be represented as follows. (Other representations are also possible just as in natural language we are able to say the same thing in more than one way.)

NEC: despatch (ag = Co in company) (go = O in order)(tmp = T2 in date)(id:T2 ≤ T1 + 7) followed by
NEC: receive (rec = C in customer) (go = invoice)(tmp = T3 in date)(id: T3 ≤ T1 + 14) sit: makes (ag = C in customer)(go = O in order) (tmp = T1 in date) followed by
fills (ag = Co in company)(go = O in order)

MUST: pay (ag = C in customer)(go = I in invoice) (tmp = T2)(id: T2 ≤ T1 + 28) sit: receive (rec = C in customer)(go = I in invoice) (tmp = T1 in date) followed by

not pay (ag = C in customer)(go = I in invoice) (tmp = T2)(id:T2 ≤ T1 + 28)

However, although the development of the above might be useful for analysis and requirements specification or documentation, it is not very helpful with respect to providing automated support for events as there is no formal connection between the terms used in the above specification and those that would have been used in the definition of the knowledge base. If we are to use the event chain representation for automated support we need to limit the use of terms to those that describe events or objects that have been defined in the knowledge base. In fact we have to take a much more systems-oriented, abstract view and think in terms of objects and attributes. We would therefore represent the above event chain in measurable linkable terms as follows.

NEC: despatch (ag = dispatcher)(go = O in order) (tmp = T2 in date)(id:T2 ≤ T1 + 7) followed by
NEC: send-invoice (ag = accounts-clerk)(go = invoice) (rec = customer)(ret = O in order)(tmp = T3) (id:T3 ≤ T1 + 14) sit: has-attribute (O in order)(T1 in date) and
has-attribute (O in order)(X in status)(id:X = “filled”)

MUST: accept-payment (ag = accounts-clerk) (ret = I in invoice)(tmp = T2 in date) (id: T2 ≤ T1 + 28) sit: send-invoice (ag = accounts-clerk)(go = I in invoice) (rec = C in customer)(ret = order)(tmp = T1 in date)

NEC: send (ag = accounts-clerk)(rec = C in customer) (go = first reminder letter) sit: send-invoice (ag = accounts-clerk)(go = I in invoice) (rec = C in customer)(ret = O in order) (tmp = T1 in date) followed by
not accept-payment (ag = accounts-clerk)(ret = I in invoice) (tmp = T2 in date)(id:T2 ≤ T1 + 28)

Despatcher and accounts-clerk from the above definition would have been defined as user-groups and despatch, send-invoice and send-reminder as events. Customer, order and invoice will have been defined as object-classes. Thus we have in OCPL some restrictions on how roles may be used and formal links between the static and the dynamic definition. These can be used for integrity checking. In Section 5 we will see how this representation may be used to define knowledge-based support.
3.3. A comment on the use of semantic roles

Functional Grammar [5] defines a complete set of roles and structures that can be used to represent any type of natural language expression. In the development of CPL, it was decided that that level of completeness was unnecessary for the purpose of requirements elicitation within a confined domain. The roles adopted by CPL, defined in terms of functions and satellites have therefore been limited [4]. However, it does seem that by omitting a set of roles known as relating roles in functional grammar, the language is handicapped in its expressive power. It is proposed that these roles are included in OCPL.

Also when it comes to representing a natural language sentence in CPL, there is often more than one way to express the same meaning depending on which roles are used. For example, the giving of a book by John to Mary might be expressed in any of the following ways depending on how the translator interprets the sentence and the semantic roles.

\[
\text{give}(ag = \text{John})(\text{rec} = \text{Mary}) \\
\text{give}(ag = \text{John})(\text{ben} = \text{Mary}) \\
\text{receive}(\text{rec} = \text{Mary})(\text{go} = \text{book})(\text{src} = \text{John})
\]

This seems not to bode well for creating an unambiguous computer-processable, intelligent definition, especially if a translation approach from natural language scenarios is intended as in COLOR-X [2]. However, provided that the human translators start with some common understanding of how the roles should be used and how relations should be set up, the problem can be overcome. An automated translation approach has greater difficulties, although the use of a lexicon can overcome this problem to some extent. The philosophical position makes this a significant problem for the COLOR-X approach but much less so for the OCPL approach, given its philosophical stance. Since OCPL is seen as a tool for defining systems and is not intended to be automatically generated, the analyst has discretion over which roles to use and can use them in a consistent way. Furthermore, OCPL provides support in this respect by including some integrity rules over how the roles are used.

4. Manipulation language for OCPL

The OCPL manipulation language is an interactive language, which basically has the following underlying command-line structure although the mode of interaction is forms-based.

\[(\text{operation}) \ (\text{class-name}) \ [(\text{parameter-clause})]\]

The parameter-clause will contain statements that assign values to the input parameters that have been defined for the particular operation. Since the parameters might be complex objects, a facility is needed to represent subparts and lists. Round brackets are used for subparts and square brackets for lists. Commas are used to separate parameter assignment statements and nesting can be used to represent object complexity. Let us consider an operation \textit{find-part-quantities} for object-class \textit{supplier} that gives quantities of particular parts supplied over a certain time period and takes as input, a \textit{supplier name}, an \textit{address} which consists of a \textit{number}, \textit{street} and \textit{town}, a list of part-numbers and a pair of dates consisting of a start-date and end-date. This operation might be called as follows.

\[
\text{find-part-quantities} \ \text{supplier} = \ldots, \\
\text{address} = (\ldots, \ldots), \text{part-number} = [\ldots, \ldots], \\
\text{dates} = (\ldots)
\]

or with example values

\[
\text{find-part-quantities} \ \text{supplier} = \ldots, \\
\text{address} = (\ldots, \ldots), \text{part-number} = [\ldots, \ldots], \\
\text{dates} = (\ldots, \ldots)
\]

Although the language can be command-line driven, in most cases the input parameters are given using an appropriate interactive form. The output is provided as a screen-based report in accordance with the output parameters specified in the operation definition.

In the command-line input format, actual names of attributes may be omitted from the clause. In this case, assignment of values to attributes is based on the order in which the parameters are declared for the operation. The \textit{find-part-quantities} operation example might therefore be expressed as follows.

\[
\text{find-part-quantities} \ \text{supplier} = \ldots, \\
\text{address} = (\ldots, \ldots), \text{part-number} = [\ldots, \ldots], \\
\text{dates} = (\ldots, \ldots)
\]

As well as user-defined operations, every object-class has four standard operations: \textit{insert}, \textit{delete}, \textit{update} and \textit{select}, which follow the general syntactic pattern for operations as shown above, the input and output parameters are taken as the object-class itself with partial instantiation being acceptable input. In the case of \textit{update}, the input is considered to be two separate object instantiations, the first to find the objects for update and the second to specify the update.

The syntax for the operations is therefore as follows.

\[
\text{insert} \ | \ \text{delete} \ \text{| select} \ (\text{class-name}) \ ((\text{parameter-clause})) \\
\text{update} \ (\text{class-name}) \ ((\text{parameter-clause}))\((\text{parameter-clause})
\]

Some examples are shown below.

\[
\text{insert} \ \text{computer} \ (\text{serial-no} = "S132415T"), \ \text{overall-weight} = 45, \ \text{peripherals} = [(\text{part-number} = "P1268",}
\]
type = “keyboard”), (part-number = “M7895”,
technology = “monitor”)]
delete computer (serial-no = “S132415T”)
update computer (serial-no = “S567238Y”)
(overall-weight = 50)

The representation of subparts and group items means that we need to use set operators like includes (for set membership), subsumes (for subsets), is-subsumed-by (for supersets) and intersects as comparators in update and select statements.

update computer (peripherals subsumes
[(part-number = “P1268”),
(part-number = “M7895”)], peripherals not includes
[(part-number = “M7895”)], peripherals includes
[(part-number = “M9841”)])

The above statement finds any computer whose peripheral list includes both part-no. P1268 and part-no. M7895 and updates it so that part-no. M7895 is no longer included but part-no. M9841 is included.

5. Generating an interface from an OCPL definition

A windows-based application interface can be generated from OCPL. We can consider the interface generation in terms of a static part and a dynamic part.

5.1. The static part

A window is generated as a frame for each object-class. Any attribute of the object-class is a slot in the frame. Any subpart of the object is a button, which will expand to an additional window when clicked. The window generated for the subpart then follows the same form as that for the main object-class; attributes are slots and subparts are buttons within an object-class frame. Fig. 1 shows a sample generated screen for object-class customer-order and a generated window for subpart customer within customer-order. If an object-class or subpart is a superclass for some other objects, its generated screen or window frame will contain buttons for each of its subclasses. When clicked, the buttons will generate new windows and frames for the subclasses. The generated screens for the subclasses automatically inherit the attributes and subparts of the superclass.

5.2. The dynamic part

5.2.1. Standard and application-defined operations

As part of the window for each object-class, there are two tool bars. The first contains buttons for the four basic operations for all object-classes: insert, delete, update and select. The second contains buttons for each application-defined operation. Consider the following OCPL definition for an object-class buildsheet, which contains all the parts required in a particular computer specification.

is_a_class(buildsheet)
has_unique(buildsheet)(serial-no)
has-subpart(buildsheet)(processor parts)
has-subpart(buildsheet)(peripheral parts)
has-subpart(buildsheet)(accessory parts)
has-accessor-operation(buildsheet)(count-part)
has-accessor-operation(buildsheet)(cost-parts)

The sample generated screen for buildsheet is shown in Fig. 2. If the insert button is pressed, a blank buildsheet-window appears showing just attribute names and subpart
buttons. The user then has to fill in details and, when ready, commit the insert. If the delete button is pressed the same window appears. The user needs to complete enough details to distinguish the instance(s) that need to be deleted. If the user presses the update button the same window appears, for the user to specify the instances to be updated, then another screen will appear for input of the update values. If the select button is pressed, the same window appears and the user inserts selection details in the form of example values. If a user clicks on an application-defined operation, a window appears, which allows the user to input the necessary parameters and then the operation is performed. The answer is given in a new window. Fig. 3 shows this for the operation cost-parts (the definition for cost-parts is given in Section 3.2.2).

5.2.2. Events or system operations
A window is generated for each system class, which contains buttons for each of its subparts and a single toolbar with buttons for each of its events. If a system is divided into subsystems, it will contain buttons for each subsystem, which when clicked will produce an appropriate system

5.2.2. Events or system operations
A window is generated for each system class, which contains buttons for each of its subparts and a single toolbar with buttons for each of its events. If a system is divided into subsystems, it will contain buttons for each subsystem, which when clicked will produce an appropriate system

5.2.2. Events or system operations
A window is generated for each system class, which contains buttons for each of its subparts and a single toolbar with buttons for each of its events. If a system is divided into subsystems, it will contain buttons for each subsystem, which when clicked will produce an appropriate system

Fig. 4. System screen for computer configuration and sales business.

Fig. 5. Subsystem screen for sales.

Fig. 6. Knowledge-based system support showing that orders are ready for despatch.

5.2.2. Events or system operations
A window is generated for each system class, which contains buttons for each of its subparts and a single toolbar with buttons for each of its events. If a system is divided into subsystems, it will contain buttons for each subsystem, which when clicked will produce an appropriate system

5.2.2. Events or system operations
A window is generated for each system class, which contains buttons for each of its subparts and a single toolbar with buttons for each of its events. If a system is divided into subsystems, it will contain buttons for each subsystem, which when clicked will produce an appropriate system

5.2.2. Events or system operations
A window is generated for each system class, which contains buttons for each of its subparts and a single toolbar with buttons for each of its events. If a system is divided into subsystems, it will contain buttons for each subsystem, which when clicked will produce an appropriate system

5.2.2. Events or system operations
A window is generated for each system class, which contains buttons for each of its subparts and a single toolbar with buttons for each of its events. If a system is divided into subsystems, it will contain buttons for each subsystem, which when clicked will produce an appropriate system

6. The use of knowledge for user support
OCPL allows us to express various rules for an application. The application generator uses these rules to provide the user with additional support in the form of guiding the user through a business or technical procedure. Let us consider the definitions we saw earlier regarding handling despatches, invoices and late payments. Knowledge-based support can be provided as follows. When the user responsible for despatching (i.e. the despatcher) logs in, the system can highlight the despatch button if any orders are ready for despatch and have a list of those orders ready for despatch. Similarly when the accounts clerk logs in, the send-invoice button will be highlighted if any invoices should be sent out and the relevant invoices can be ready. If any payments are overdue, the system would also highlight the send-reminder button and have to hand a list of late payers who should be
sent first reminder letters. Fig. 6 shows an example of this type of screen-based support.

The way in which the rule base is activated is based on the events with which a user is associated. For instance, the despatcher will have been given access rights to a number of events within the Administration subsystem, one of which would have been the despatch event. Therefore when the despatcher logs in, the system will check to see if any relevant events are part of the destination clause in any rule. If so, backward chaining is used, to see if the rule fires. If it does, the event is highlighted on the despatcher’s screen for activation.

7. Discussion and conclusions

OCPL is a generalised modelling language that can be used to represent many types of application. It is intended to be used by experienced system modellers as part of a systems development life cycle. It is most likely to be used after some initial analysis of the problem domain has been undertaken. It provides an object and knowledge framework and can be used for prototyping or final system generation. The present toolset for OCPL is written in Prolog++ from Logic Programming Associates and generates a Prolog++ data/knowledge base as well as a user-oriented application interface. Other underlying platforms could be used as a target system but at present Prolog++ is the only one that has been developed. Examples of real applications, which OCPL has been used to model for evaluation purposes, are the design of electricity generating turbines and a computer configuration and sales business.

The OCPL examples in this paper have been taken from the latter model. The modelling of the applications mentioned has indicated that OCPL fulfils its objectives of being able to support object-oriented modelling in an efficient and relatively concise manner whilst also supporting knowledge representation in the form of constraints. In the case of the turbine application, the knowledge base held rules regarding legitimate design of various turbines (for instance, the circumstances under which particular turbine parts could be combined). In the case of the computer configuration and sales business model, the knowledge base was concerned with the business process (for instance, modelling event chains).

The OCPL model can be thought of as consisting of two parts: an object model; and a constraint model. The constraint model can hold static and dynamic constraints. The dynamic constraints may be held in the form of event chains thus representing business processes. In OCPL, following CPL, constraints can be thought of as necessary or deontic. Necessary rules are always true within the domain of discourse. Deontic rules ought to be true but sometimes are broken. This distinction, which is not often seen in other rule-based models, allows for more precise modelling of constraints and makes the application semantics more explicit, thus providing for a more maintainable and powerful system.

Let us now consider how OCPL relates to other work. In Section 1, some prime examples of related work were given as the COLOR-X project, UML, SOMA/SOMATIK and the TEMPORA project. Of these, COLOR-X is the most relevant. It is a continuing project by the original developers of CPL, with some similar goals to those of the OCPL project. As mentioned in Section 1, the COLOR-X model omits some object-oriented features, which are considered by the authors of this paper to be essential to meet the goals of the OCPL project, namely efficient and effective systems modelling and generation. The COLOR-X object model is thus considered to be a weaker object model than that of OCPL. It is widely recognised now in modern systems specification and development that the behaviour, as well as the structure of a system, should be captured and supported. The object-oriented approach provides a useful way of modelling behaviour through the concepts of encapsulation and operations. Furthermore, the object approach provides a useful means of capturing real-world structures through complex objects and composition. In traditional systems, based on the relational model, these aspects could not be captured. Another useful way of modelling behaviour is through the use of rules. In general, operations can model behaviour at a micro or class level and rule sets at a macro or system level by representing event chains, which in turn represent business processes. OCPL has been developed as an approach that supports both types of behaviour modelling, whilst also supporting rapid system generation and prototyping. The lack of support for operations in COLOR-X is considered to be a major weakness; similarly, the lack of support for object composition. These aspects are considered to be essential in a system that captures behaviour and generates appropriate user interfaces. The strength of COLOR-X over OCPL is in its handling of scenarios and initial requirements capture. COLOR-X generates an extended CPL model from natural language scenarios. It uses a lexicon to help in this task. It also allows for an object-oriented graphical system description, which also generates extended CPL. The models generated can then be cross-checked for consistency and a prototype generated. The strength of OCPL over COLOR-X is in the stronger object model and more powerful system generation. Operations can be explicitly represented, likewise events, systems and users. GUIs can be generated and used that directly mirror the structure inherent in the OCPL model. Not only can the static structure be represented, the dynamic aspects also can be covered in the GUI and intelligent support can be provided and tailored for users on the basis of the business constraints provided. It is not surprising that the two projects have strengths in different areas since the relative strengths are commensurate with the original aims of the two projects. COLOR-X aims to facilitate natural-language-based requirements capture whilst OCPL aims to generate a system from an object-knowledge-based language.
description. It is therefore considered that the achievements of the two projects are complementary to each other. The inroads made by COLOR-X in the field of natural-language-based requirements capture can be combined with the advances made by OCPL in system generation to create a more powerful toolkit comprising various modules to be used by different users and developers as appropriate to their needs. Where the projects meet is in the extension of CPL to an object model. Although the OCPL model is regarded as a stronger object model the techniques used to achieve the extension in both projects are compatible and thus integration of the projects is feasible.

The major advantage OCPL has over the other work mentioned above, is its conceptual richness because of its basis in functional grammar and its refined characterisation of constraints. Both these aspects make for powerful knowledge representation and a more explicit representation of the model semantics. This leads to a more understandable system and the provision of better user support. Another way in which OCPL differs from UML and SOMA/SOMATIK is in the support of time. Neither UML nor SOMA/SOMATIK support a full temporal logic for reasoning about events. SOMA/SOMATIK includes a time facility in the sense of task sequence simulation but here the time representation is a background for the task sequence rather than a logic with which temporal constraints can be formulated. TEMPORA does include a full temporal logic but the TEMPORA model lacks other features that OCPL provides, namely the power of representation mentioned earlier and an object model (TEMPORA is based on an entity-relationship model). On the minus side, OCPL does not have a strong emphasis on methods or methodology. UML contains a number of techniques for supporting analysis and design. SOMA/SOMATIK offers a methodology and a well-developed CASE tool, although code generation is not part of it. In CASE tool terms the OCPL approach might be thought of as lower CASE and the SOMA/SOMATIK approach as upper CASE. In OCPL, there is little support for analysis and design except through prototyping from the OCPL model. At present, that part of the life cycle is assumed, either through existing techniques or through the skill of the modeller. This assumption is justified given the work being done by approaches such as SOMA/SOMATIK and UML/Rational ROSE. Many of the techniques proposed there could be adopted by OCPL as support for obtaining the initial model. However, OCPL could benefit from more tailored support for the requirements definition stage, not least so that its unique modelling features can be effectively utilised. As it stands the OCPL toolkit consists of a data and knowledge definition interface (DKDI), a syntax checker (SC), a natural language generator (NLG) and two definition translators (DT1 and DT2). Whilst OCPL is an extremely powerful modelling language, it is textual and although much more concise than CPL from which it is derived, it can still be laborious. Thus, it is essential that tools be provided to allow a model to be constructed more rapidly than simply typing in the predications long-hand. This is the primary role of the DKDI. Integrated into the DKDI is an OCPL syntax checker and a tool that translates OCPL statements into English for verification and validation purposes. The definition translators, DT1 and DT2, enable the automatic translation of a set of OCPL predications into a working knowledge base system. DT1 produces a knowledge management system accessible only through the Prolog++ console window. DT2 adapts and extends DT1, in order to automatically produce a suitable application interface on top of the knowledge base. The definition translators therefore accomplish one of the main goals of the project, namely appropriate system generation.

The above aspects aside, OCPL has a number of features that are comparable to those developed in the SOMA/SOMATIK approach. Like SOMA, the OCPL object model is designed for modelling associations within object-classes rather than having a separate concept of relationship or association. Referential integrity can be checked as part of the constraint model. The use of user-defined relations in OCPL is restricted to the constraint model. Here relations correspond to defined events and terms within relations either to users, user-groups or object-classes. One of the main ways in which the SOMA approach differs from other object-oriented models is in the definition of rule sets, which can be associated with object-classes. In OCPL, the constraint model or rule set is apart from the object-classes. One might say, therefore, that SOMA follows the object-oriented paradigm more closely in that rules can be encapsulated within rather than outside objects. The constraint model of OCPL could be subdivided so that particular rule sets could be encapsulated with particular object-classes. However, the rules that represent event chains typically work across many object-classes and thus could not be encapsulated. In SOMA, the agent and task object models deal with business events. SOMA adopts a multiple model approach for manageability. OCPL has a single model to represent object statics, object behaviour and systems. OCPL thus might be considered to be more concise and seamless but the fact that OCPL concentrates on lower CASE and SOMA/SOMATIK on upper, makes such a distinction invalid. OCPL could adopt a multi-model approach for its upper levels of system engineering. SOMA operation specifications allow for pre- and post-conditions. This is useful for modelling dynamics and the semantics of operations. OCPL does not support pre- and post-conditions explicitly but can support them implicitly in the constraint model. SOMA supports the idea of wrappers to subdivide systems and make them more manageable. Subsystems are thus specified as objects with an interface. In OCPL this is achieved through the definition of systems, subsystems and events. It may be noted that whilst SOMA makes everything an object (e.g. subsystems, users and tasks, as well as more traditional application structures),
OCPL uses distinct concepts for system features such as for systems, subsystems, user-groups and users. The object concept could be used for such features but the chosen demarcation is considered to be more natural.

To sum up, OCPL has a number of features in common with other approaches and may in its further development also benefit from ideas prevalent in other approaches. On the other hand, OCPL has a number of features that are unique and which should provide for better systems modelling and development, namely its stronger knowledge representation and system generation. Further work planned for OCPL is the development of the toolkit to support graphical definition, facilities for customisation of the generated interface, more work on a methodology for OCPL and further evaluation through the modelling of different application areas. It might be noted that the powerful representation of knowledge through functional grammar is in some ways a double-edged sword. As with natural language, the same thing can be stated in many different ways using functional grammar. This is not helpful for computer representation. Part of the work in developing the OCPL methodology will be an investigation on support for the appropriate use of roles within relations in the OCPL constraint model. Non-functional requirements were originally considered to be outside the scope of the OCPL model apart from those concerning the look and feel of the system, which are to some extent covered through the prototype system generation. The representation of other non-functional requirements is an area of possible future work. A further area of research would be to look at the scope of OCPL for other information systems applications. For instance, the growth of networking requires intelligent support for computer supported co-operative working. OCPL, which provides means for defining event chains, relations and various roles as well as objects, looks to be a good candidate for defining workflows. Another area where OCPL would be useful is that of knowledge banks. OCPL with its wealth of constructs offers a valuable means of representing knowledge of company processes. OCPL could be used to represent information in data warehouses. Its predication structure and constraint modelling would allow for ad hoc as well as routine information to be held, thus providing the warehouse with more intelligence. A further area is the area of database integration and interoperability. Given the richness of conceptual modelling power in OCPL and its sophisticated support of constraints, it would be a good candidate for a canonical model in this area. Previous work has looked at the use of CPL as a canonical model for database integration [7]. OCPL would provide even more representational power.

This paper has outlined the background and development of OCPL. The distinguishing features of OCPL are the use of functional grammar combined with the formalisms of object-orientation and constraint-based logic to define knowledge-based systems. It differs from related work in that operations and events are explicit and integrated fully into a constraint-based syntax. A restricted use of the OCPL ontology and syntax provides for class integrity checking and more complete and elegant system generation.

Acknowledgements

Thanks to Richard Gatward of Coventry University for initially suggesting the use of CPL as a basis for this work and many discussions throughout the project. The project was supported by Coventry University.

References