RACSAM
Rev. R. Acad. Cien. Serie A. Mat.
VOL. **98** (1), 2004, pp. 127–152
Ciencias de la Computación / Computational Sciences

Z Specification of Object Oriented Constraint Programs

Laurent Henocque

Abstract. Object oriented constraint programs (OOCPs) emerge as a leading evolution of constraint programming and artificial intelligence, first applied to a range of industrial applications called configuration problems. The rich variety of technical approaches to solving configuration problems (CLP(FD), CC(FD), DCSP, Terminological systems, constraint programs with set variables, ...) is a source of difficulty. No universally accepted formal language exists for communicating about OOCPs, which makes the comparison of systems difficult. We present here a Z based specification of OOCPs which avoids the falltrap of hidden object semantics. The object system is part of the specification, and captures all of the most advanced notions from the object oriented modeling standard UML. The paper illustrates these issues and the conciseness and precision of Z by the specification of a working OOCP that solves an historical AI problem : parsing a context free grammar. Being written in Z, an OOCP specification also supports formal proofs. The whole builds the foundation of an adaptative and evolving framework for communicating about constrained object models and programs.

Especificación en Z de programas orientados a objetos con restricciones

Resumen. Los programas orientados a objetos con restricciones (OOCPs) surgen como una evolución trascendental de la programación con restricciones y de la inteligencia artificial, aplicados, en primer lugar, a una variedad de aplicaciones industriales que se denominan problemas de configuración. La dificultad reside en la rica variedad de aproximaciones técnicas para la resolución de problemas de configuración (CLP(FD), CC(FD), DCSP, sistemas terminológicos, programas con restricciones con variables sobre conjuntos,...). No existe ningún lenguaje formal universalmente aceptado para la comunicación acerca de los OOCP, lo que dificulta la comparación entre sistemas. En este trabajo se presenta una especificación de OOCPs basada en Z, que evita caer en la trampa de la semántica de objetos ocultos. El sistema objeto forma parte de la especificación y capta todas las nociones más avanzadas de la modelización orientada a objetos estándar UML. Este trabajo ilustra estas cuestiones y la concisión y precisión de Z al especificar un OOCP operativo que resuelve un problema histórico de la IA, concretamente, el análisis sintáctico de una gramática libre de contexto. Al estar escrito en Z, una especificación OOCP también soporta demostraciones formales. El trabajo forma la base de un marco adaptativo y evolutivo para la comunicación de modelos y programas de objetos con restricciones.

Presentado por Luis M. Laita.
Recibido: December 14, 2003. *Aceptado*: October 13, 2004.
Palabras clave / Keywords: constraint programming, constraint modeling, formal specification, Z language, configuration, knowledge representation.
Mathematics Subject Classifications: 68T27, 68T30, 68Q60, 68Q65, 08A02, 03C13.

Introduction

From Configuration to Object Oriented Constraint Programs

Rule based systems, logic programming, and a recent evolution of constraint programming have been applied to a category of problems called *configuration* problems. Configuring means simulating the construction of a composite and complex product, based on a library of elementary components. Components are subject to relations (this is the *partonomic* information), and participate to inheritance relationships (this is called the *taxonomic* information). Given an input in the form of a partial product and specific constraints, the goal of configuration is to pick up or generate, then interconnect the necessary components, for finally deciding upon their exact type and attribute values. Configuration output is a complex interconnected product respecting well formedness rules stated by various constraints. This combinatorial problem is explicitly formulated as a finite model generation problem.

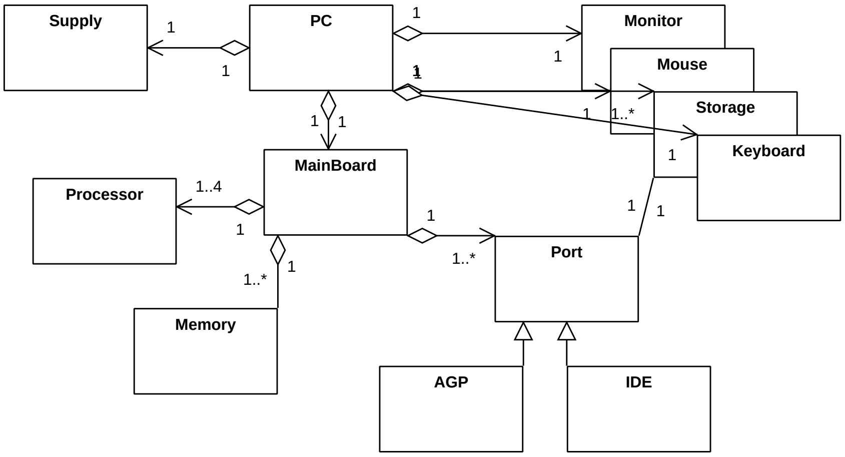

Figure 1. A simplified object model for a personal computer

The UML class design [14] in figure 1 illustrates this with a classical example. Configuring a personal computer (PC) consists in picking components from a catalog of component parts (e.g. processors, hard disks in a PC), using known relations between types (motherboards can connect up to four processors), and instantiating object attributes (selecting the ram size, bus speed, . . .). Constraints apply to such configuration problems that define which products are valid, or well formed. For example in a PC, the processors on a motherboard all have the same type, the ram units have the same wait times, the total power of a power supply must exceed the total power demand of all the devices. Configuration applications deal with such constraints, that bind variables occurring in the form of variable object attributes deep within the object structure.

We suggest to abandon the term configuration, bound to a very specific application area (even if it is broadly distributed in the economy), in favor of a more general purpose and AI related denomination : object oriented constraint programs (OOCP for short). OOCP has many potential AI applications, ranging from context free language parsing (we use this example), to image recognition, or distributed agent intelligence and planning.

Existing approaches

The industrial need for configuration is widespread, and has triggered the development of many applications, as well as generic configuration tools or configurators, built upon all available technologies. For instance, configuration is a leading application field for rule based expert systems. As an evolution of R1 [19], the XCON system [5] designed in 1989 for computer configuration at Digital Equipment involved 31000 components, and 17000 rules. The application of configuration is experimented or planned in many different industrial fields, electronic commerce (the CAWICOMS project [12]), software [32], computers [23], electric engine power supplies [17] and many others like vehicles, electronic devices, customer relation management (CRM) or even software [32, 13].

The high variability rate of configuration knowledge (parts catalogs may vary by up to a third each year) makes configuration application maintenance a challenging task. Rule based systems like R1 or XCON lack modularity in that respect, which encouraged researchers to use variants of the CSP formalism (like DCSP [20, 26, 2], structural CSP [21], composite CSP [24]), constraint logic programming (CLP [16], CC [13], stable model semantics for logic programs with disjunctions [27]), or description logics/object oriented approaches [29, 31, 18, 22].

Because of the variety of approaches to this problem (CLP(FD), CC(FD), CP and extensions, Expert Systems), no common language is available for researchers to exchange problem statements and compare their results. Each of the above cited articles uses an ad-hoc description of its working example. Some UML [14] models are presented from time to time, which never allow to overcome the ambiguities inherent to this exercise, even though the UML is far more formal than people usually think. The usefulness of (a subset of) UML plus the constraint language OCL as a language of choice for the specification of configuration problems has been advocated in [11, 10] and used for instance in [30].

Paper objectives

The main objective of this paper is to propose to the growing community working on configuration problems and applications a common language having more formal grounds but potentially higher expressive power than the UML for exchanging models and problems. To those ends, this paper presents a general use object oriented constraint system for the specification of object oriented constraint problems. The object system is not predefined, or provided as an object oriented extension to some specification language. Instead, we have chosen to make the object system specification explicit, using the Z language [28]. There are several reasons for both the choice of Z and this approach:

- we feel that in order to be widely accepted, the underlying semantics of an object system must be questionable, commented, improved, and formally established,
- the Z language has very simple and clear semantics, and offers an extremely rich range of relational operations, a crucial issue in object oriented constraint programming,
- the Z language was shown to have the favor of the industry over other specification languages [4], essentially because of its structure (the grouping concept introduced by *schemas*),
- we have succeeded in specifying in Z the most advanced class modeling constructs from the UML [14], which has become the standard in object oriented modeling. This guarantees that modeling cannot be biased, or tweaked by arbitrary limitations in the object language, and that any object oriented model can be specified
- Z specifications can be type checked (we used *f*UZZ[1] extensively to type check this paper), which offers a first level of formal verification,
- Z specifications are subject to formal proofs, possibly assisted or automatized by theorem provers,
- Z offers built in extensibility features, that allow to formally define, then use, any operator or relation using any possible syntax (prefix, infix, postfix). We exploit this feature to improve the readability of constraints.

[1] available at http://spivey.oriel.ox.ac.uk/ mike/fuzz/

To summarize this, we found that Z is the simplest logic offering both structure (via schemas) and support for the formal definition of complex structural constraints or expressions. This last issue is crucial to object oriented constraint programs, for instance to allow the statement of a constraint relating, in a personal computer, the sum of powers used by all elementary electrical devices, to the power made available by the supply. It was furthermore argued [15] that coalgebras support most of the notions required to deal with object state and class invariants. Relational algebraic languages like Z provide the capacity of specifying both algebras (types and their operations) and coalgebras (states and their transitions).

There have been many attempts to capture object orientation within specification languages, either viewed as Z extensions (Object Z [25], OOZE [1]) or based upon other mathematical grounds (the FOOPS [6] extension to OBJ). A logical approach to object orientation is also put to work in terminological knowledge representation languages [7, 3]). Also, constraint programming has been introduced in object oriented knowledge representation languages (as e.g in CLAIRE [8]).

We are not presenting the latest object oriented language, or system, or extension to whichever existing approach. Our aim is to capture rich enough object oriented semantics in a simple and unmodified logic (hence Z), rather than to rely upon the inherent semantics of an object oriented extension of some logic. By doing so, we cirvumvent both the potential expressiveness limitations of any given object system, and the possibility that its semantics are improperly defined, or questionable. Our approach allows to document an object oriented constraint program, by simultaneously specifying both the object system semantics, and the problem itself. This task is made simpler because we do not need to specify state transitions (coalgebra operations), but reason exclusively about state. Essential issues in object oriented *programming* like polymorphism, or concurrency are irrelevant here. Our goal is to express valid object system states, using constraints, which altogether describe an object oriented constraint program. This simplifies the use we make of Z, because in our case decorations are useless.

The paper is organized as follows : section 1 briefly introduces essential aspects of Z. Section 2 specifies the class and type features of an object system, illustrating how all essential object oriented modeling concepts can be captured. Section 3 specifies relations and roles. Section 4 details how object structure constraints can be formulated, and introduces useful auxiliary constructs. Section 5 presents the specification of an artificial intelligence application of object oriented constraint programs to context free grammars parsing. Section 6 is the conclusion. It can be a possible reading strategy to first take a glance at section 5, since it illustrates the essential motivation of this work.

1. Introducing Z

For space reasons, it is impossible to make this paper self contained, since this would suppose a thorough presentation of both the UML notation [14], and the Z specification language [28]. The reader, if novice in these domains, is kindly expected to make his way through the documentation, which is electronically available. For clarity however, we provide a brief description of several useful Z constructs. More advanced notations or concepts will be introduced when necessary.

1.1. Data types as named sets

Z data types are possibly infinite sets, either uninterpreted :

$$[DATE]$$

or axiomatically defined as finite sets:

$$\mid\ dom : \mathbb{F}\,\mathbb{N}$$

or declared as explicitly initialized free types.

$$colors ::= red \mid green \mid blue$$

From now on, all possible relation types can be built from cross products of other sets.

1.2. Axiomatic definitions

Axiomatic definitions allow to define global symbols having plain or relation types. For instance, a finite group is declared as

$$
\begin{array}{|l}
\mathit{zero} : \mathit{dom}\\
\mathit{inverse} : \mathit{dom} \longrightarrow \mathit{dom}\\
\mathit{sum} : \mathit{dom} \times \mathit{dom} \longrightarrow \mathit{dom}\\
\hline
\forall x : \mathit{dom} \bullet \mathit{sum}(x, \mathit{inverse}(x)) = \mathit{zero}\\
\forall x : \mathit{dom} \bullet \mathit{sum}(x, \mathit{zero}) = x\\
\forall x, y : \mathit{dom} \bullet \mathit{sum}(x, y) = \mathit{sum}(y, x)\\
\forall x, y, z : \mathit{dom} \bullet \mathit{sum}(x, \mathit{sum}(y, z)) = \mathit{sum}(\mathit{sum}(x, y), z)
\end{array}
$$

The previous axiomatic definition illustrates cross products and function definitions as means of typing Z elements. Now axioms or theorems are expressed in classical math style, involving previously defined sets. For instance, we may formulate that the inverse function above is bijective (this is a theorem) in several equivalent ways as e.g.:

$$\mathit{inverse} \in \mathit{dom} \mathrel{\rightarrowtail\kern-1.2em\twoheadrightarrow} \mathit{dom}$$

where the $\mathrel{\rightarrowtail\kern-1.2em\twoheadrightarrow}$ operator defines a bijection, or explicitly using an appropriate axiom :

$$\forall y : \mathit{dom} \bullet \exists_1 x : \mathit{dom} \bullet \mathit{inverse}(x) = y$$

1.3. Schemas

The most important Z construct, *schemas*, occur in the specification in the form of named axiomatic definitions. A schema $[D \mid P]$ combines one or several variable declarations (in the declaration part D) together with a predicate P stating validity conditions (or constraints) that apply to the declared variables.

$$
\begin{array}{|l|}
\hline
\mathit{SchemaOne}\\
\hline
a : \mathbb{N}\\
b : 1 \mathrel{..} 10\\
\hline
b < a\\
\hline
\end{array}
$$

The schema name hides the inner declarations, which are not global. A schema name (as *SchemaOne* above) is used as a shortcut for its variable and predicate declarations that can be universally or existentially quantified at will. Schemas are *true* or *false* under a given *binding*. For instance, *SchemaOne* is *true* under the binding $\langle 4 \Rrightarrow a, 3 \Rrightarrow b\rangle$ and *false* under the bindings $\langle 3 \Rrightarrow a, 234 \Rrightarrow b\rangle$ or $\langle 3 \Rrightarrow a, 4 \Rrightarrow b\rangle$. The latter violates the explicit constraint stated in the predicate part of the schema, while the former also violates the implicit constraint carried by the interval definition $1 \mathrel{..} 10$ (a subset of $\mathbb{N}$). In some contexts, a schema name denotes the set of bindings under which it is true.

Z allows Boolean schema composition. Two schemas can be logically combined (e.g. ”anded”) by merging their declaration parts provided no conflict arises between the types of similarly called variables, and by applying the corresponding logical operator (e.g. the conjunction) to the predicates. For instance, given the schema *SchemaOne* above, and another schema called $\mathit{SchemaTwo} \mathrel{\widehat=} [b : \mathbb{N}; c : \mathbb{N} \mid b < c]$ [2], we may form the schema *SchemaThree*, as

$$\mathit{SchemaThree} \mathrel{\widehat=} \mathit{SchemaOne} \land \mathit{SchemaTwo}$$

[2] This illustrates another syntax for simple schema declarations

Incidentally, the variable declarations b in both schemas collide, but not for their types since b is a member of $\mathbb{N}$ in both cases. The first declaration of b bears a built in constraint, which can be moved to the predicate part. Hence the schema *SchemaThree* would list as :

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SchemaThree}} \\
a, b, c : \mathbb{N} \\
\hline
1 < b < 10 \\
b < a \\
b < c \\
\hline
\end{array}$$

2. Classes

We wish to describe Z specified object oriented constructs so as to reach an expressive power comparable to that allowed by the UML [14] class (and state) diagrams, hence allowing to model in a purely formal way the static properties of an object constraint system. In order to sit our definitions on clean formal grounds, we propose a generic Z specification that captures all required concepts. In defining classes, we specially need to cover two essential notions : multiple inheritance and multiple discriminator specialization. While the former is attained by existing object oriented Z extensions, the latter is not, which partly justifies this work. An essential contribution of this work is that the object system specification is explicit, and can be discussed, adapted, or extended at will. Essential in this respect is the clarity and soundness of the notion of ”object references”, made explicit here.

The *schema* notation can be understood as an heterogeneous aggregate of mathematical variables, subject to built in constraints. In other words, schemas can be seen as mathematical variables representing all the possible states of Pascal records, or of C structs. From an object oriented point of view, the predicate part of a schema forms the essence of what is called in OO programming a *class invariant* : a property that must be true of object instances at all times (i.e. before and after any method call).

2.1. Preliminary definitions

The object oriented vocabulary involves common and rather vague words. We wish to make things precise, and to avoid difficulties in the sequel. A *class definition* holds the description of *class specific attributes* and *class specific invariant* together with *inheritance* relationships. Accounting for inheritance yields a description of the *(full) class structure* and *(full) class invariant* which together form the *class specification*. A class *instance*, or *object*, is a binding of values to all attributes in the class structure that satisfies the class invariant. Such a binding is often referred to as *state*. The set of all object instances bijectively maps to a set of object *references*. The bijection between object references and class instances allows for a precise definition of *class* and *type*. We call *class* the set of object references mapped to all the instances of a given class structure. A *type* recursively defines as the union of a given class, and of all its subclass types down the inheritance directed acyclic graph. By defining types as sets, we stay respectful of Z’s terminology, which identifies sets and types. All these definitions will be illustrated and made understandable by examples in the sequel. Z provides enough constructs to account for classification mechanisms. We first define a set *ObjectReference* of object references as an uninterpreted data type.

$$[\textit{ObjectReference}]$$

ObjectReference would be interpreted on the set of natural numbers (or a finite subset) by an automated theorem proving approach based on finite model generation. Practical implementations of object systems typically use pointers or integers as object references. We define *ReferenceSet* as an abbreviation for finite sets of object references, later used to model object *types*.

$$ReferenceSet == \mathbb{F}\ ObjectReference$$

We define class names as global names using Z's free type declaration syntax. For practical reasons, if a class should have the name *Engine*, we reserve the symbol *Engine* to denote the corresponding type. The global symbol denoting the class definition is obtained by prepending the string "Class" to the actual name. In our example, the class name thus writes as *ClassEngine*. Depending upon the context, the declaration of class names may look like :

$$CLASSNAME ::= ClassPC \mid ClassPrinter \mid ClassMonitor \mid \ldots$$
$$CLASSNAME ::= ClassCar \mid ClassWheel \mid ClassEngine \mid \ldots$$

We now declare *ObjectDef* as a predefined super class for all future classes. Object definitions will be used to bijectively map each object to a unique individual from the set *ObjectReference*. Object references are needed in addition to object state since in object oriented modeling two distinct objects may share the same attribute values (whereas in Z two "bitwise equal" bindings represent the same logical entity). Also, since two distinct Z schemas may have the same Z type, we need to integrate the actual class name in objects.

$$\begin{array}{|l}
\hline
ObjectDef \\
ref : ObjectReference \\
class : CLASSNAME \\
\hline
\end{array}$$

We define a function *instances* mapping class names to the set of instances of that class.

$$\begin{array}{|l}
instances : CLASSNAME \longrightarrow ReferenceSet
\end{array}$$

2.2. Defining class structures using inheritance

An essential aspect of object oriented modeling is that objects are associated with *state*. On the one hand, inheritance relations allow to restrict the possible values of attributes declared in superclasses (this phenomenon is called inheritance for *specialization*). On the other hand, classes may extend the list of attributes defined in superclasses by their own (this is called inheritance for extension). Most situations where inheritance occurs combine both cases in a single inheritance relation. Z offers built in representation of state in the form of schemas. We now show a way to associate such schemas to individual types in a standardized way, so as to bind state to the types as declared previously. We illustrate this through a simple three class example : class *B* inherits *A*, extending it with an extra attribute, and class *C* specializes *A* with an extra constraint.

$$CLASSNAME ::= ClassA \mid ClassB \mid ClassC$$

Each class *X* is implemented via two constructs. First, the class definition occurs as a schema called *ClassDefX* (we prepend "ClassDef" to the desired class name to form the schema name). This schema defines both the class attributes and inheritance relationships, as would any class definition do in object oriented modeling or programming languages. The predicate part of the schema offers room for the specification of class invariants.

$$\begin{array}{|l}
\hline
ClassDefA \\
a : 1 \mathrel{..} 10 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
ClassDefB \\
ClassDefA \\
b : \mathbb{N}_1 \\
\hline
\end{array}$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ClassDefC}} \\ \hline
\mathit{ClassDefA} \\ \hline
a \geq 5 \\ \hline
\end{array}
$$

Class definitions as seen above account for inheritance by simply copying the definition schemas of the inherited (super) classes. Doing so allows to state constraints involving attributes pertaining to super classes (this is *specialization*). All the predicates present in the inherited classes are conjoined (i.e. logically ”anded”) to the predicate part of the resulting schema. This formulation hence adequately accounts for both types of inheritance : extension and specialization.

2.3. Multiple inheritance

Multiple inheritance is achieved simply by importing the schema definitions of all inherited superclasses into a new one.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ClassDefD}} \\ \hline
\mathit{ClassDefA} \\
\mathit{ClassDefB} \\ \hline
b = 5 \\ \hline
\end{array}
$$

Note that the types of all inherited attributes must match. If attributes having the same name are inherited from two distinct superclasses, either or both can be renamed to prevent clashes, as e.g. in :

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ClassDefE}} \\ \hline
\mathit{ClassDefB}[d/b] \\ \hline
d = 5 \\ \hline
\end{array}
$$

where the constraint $d = 5$ actually binds the attribute originally declared as b in *ClassDefB*. A Z type checker can detect errors in the formulation of such class definitions, specially when type conflicts occur for attributes having the same name.

2.4. Object and class references

To implement a working object system, we need to add some extra technical information to class definition schemas : object and class references. Like before, schema composition with the logical operator $\wedge$ offers the expected semantics of extension by combining the schema types of the two schemas and of specialization by conjoining their predicate parts. Assuming the same toy example as before, (*A* is a toplevel class that *B* and *C* inherit), we write the following :

$$
\begin{array}{l}
\mathit{ClassSpecA} \mathrel{\widehat{=}} \mathit{ClassDefA} \wedge [\mathit{ObjectDef} \mid \mathit{class} = \mathit{ClassA}] \\
\mathit{ClassSpecB} \mathrel{\widehat{=}} \mathit{ClassDefB} \wedge [\mathit{ObjectDef} \mid \mathit{class} = \mathit{ClassB}] \\
\mathit{ClassSpecC} \mathrel{\widehat{=}} \mathit{ClassDefC} \wedge [\mathit{ObjectDef} \mid \mathit{class} = \mathit{ClassC}]
\end{array}
$$

It must be understood that the schema types corresponding to *ClassSpecA* and *ClassSpecC* are the same (this schema type is noted $\langle\!|\, i : \mathit{ObjectReference};\ a : \mathbb{N};\ \mathit{class} : \mathit{CLASSNAME} \,|\!\rangle$ in Z), even though the schema names differ, because class *C* specializes *A* but does not extend it. Hence a specific workaround is needed to make sure that the set of bindings that satisfy *ClassSpecC* is not included in *ClassSpecA*, and more generally that no two sets of bindings satisfying two distinct class definition schemas can intersect. This goal is achieved thanks to the *class* attribute inserted via the schema *ClassDef*, that takes a distinct value for each class.

2.5. Defining class types

We can now make our toy *ABC* model more complete, and define what the types $A,\ B,\ C$ represent. We use an axiomatic definition of three sets $A,\ B,\ C$ as finite sets of object references:

$$
\begin{array}{|l}
A, B, C : ReferenceSet \\
\hline
A = instances(ClassA) \cup B \cup C \\
B = instances(ClassB) \\
C = instances(ClassC) \\
\\
instances(ClassA) = \{o : ClassSpecA \mid o.class = ClassA \bullet o.i\} \\
instances(ClassB) = \{o : ClassSpecB \mid o.class = ClassB \bullet o.i\} \\
instances(ClassC) = \{o : ClassSpecC \mid o.class = ClassC \bullet o.i\} \\
\\
\forall\, i : instances(ClassA) \bullet (\exists_1\, x : ClassSpecA \bullet x.ref = i) \\
\forall\, i : instances(ClassB) \bullet (\exists_1\, x : ClassSpecB \bullet x.ref = i) \\
\forall\, i : instances(ClassC) \bullet (\exists_1\, x : ClassSpecC \bullet x.ref = i)
\end{array}
$$

The declaration part in this axiomatic definition declares the type sets corresponding to all the classes in our toy model. The properties of these sets are stated by several axioms :

- The types of sub classes are subsets of a class type. A type is the union of (the object references of) all the corresponding class instances, and of the types of its subclasses. Note that by making our example more complete the types *B* and *C* might intersect, because of multiple inheritance.

- The set $instances(ClassX)$ holds the schema bindings having the ”class” attribute set to *ClassX*. These sets are pairwise disjoint by construction

- The other set, *X*, corresponds to the classical notion of a type.

- The same object reference cannot be used for two distinct objects in the same class.

These axioms ensure that each object reference is used at most once for an object. Alternatively stated, no two distinct ”object” bindings share the same object reference. The preceding type definitions make the set *ObjectReference* the most general type in the model. Based upon these definitions, any class located in the middle of the inheritance tree is *concrete* : in an interpretation, we may have an instance of *A* that is neither an instance of *B* nor *C*. Finally, this specification makes clear the distinction between :

- a class *definition* : this is the schema *ClassDefX*, which declares inheritance,

- a class *specification* : this is *ClassSpecX*, which accounts inheritance, and for object and class references,

- a *class* : this is represented by the set $instances(ClassX)$,

- an object's *type* : any set *X* to which the object's reference belongs (if an object is an instance of class B, and B inherits A, it is accepted to say that this object is a ”B”, and also that it is an ”A”).

2.6. Semantics, interpretations, objects

An *interpretation* of a Z specification is a set of bindings with types corresponding to the schema types that occur in the specification. A *model* in the logical sense is an interpretation that satisfies all the axioms. An *object* is a binding satisfying the schema type and properties of some class specification schema (*ClassSpecX*). Note that such a binding may satisfy the schema types and properties of several distinct class specification schemas, because in Z the schema name isn't part of the schema type). This does no harm however, since the *class* attribute in class specification schemas sorts things apart. Also note that the final axioms in the axiomatic definition of types ($\forall\, i : instances(ClassX) \bullet (\exists_1\, x : ClassSpecX \bullet x.i = i)) \ldots$) constrain the valid interpretations so that each object reference occurs only once among the whole set of object bindings, which to the best of our knowledge cannot be formulated more concisely.

2.7. Creating objects

Although we do not focus here on the dynamics of the object systems, but rather on the mathematical properties of their valid states, it is of some interest at this point to mention that we are modeling a system that could be specified further to model a practically usable application, where object instances can be created, and destroyed. To achieve this requires to get a hold over the global system state. This is achieved by placing the type definitions within a schema, instead of keeping them as global axiomatic definitions :

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;ObjectSystemABC\;}} \\
A, B, C : ReferenceSet \\
ObjectsA : \mathbb{P}\, ClassSpecA \\
\hline
A = instances(ClassA) \cup B \cup C \\
B = instances(ClassB) \\
C = instances(ClassC) \\
\\
instances(ClassA) = \{o : ClassSpecA \mid o.class = ClassA \bullet o.i\} \\
instances(ClassB) = \{o : ClassSpecB \mid o.class = ClassB \bullet o.i\} \\
instances(ClassC) = \{o : ClassSpecC \mid o.class = ClassC \bullet o.i\} \\
\\
\forall\, i : instances(ClassA) \bullet (\exists_1\, x : ClassSpecA \bullet x.ref = i) \\
\forall\, i : instances(ClassB) \bullet (\exists_1\, x : ClassSpecB \bullet x.ref = i) \\
\forall\, i : instances(ClassC) \bullet (\exists_1\, x : ClassSpecC \bullet x.ref = i) \\
\hline
\end{array}
$$

Now, the following schema defines how the system gets updated because of object creation :

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;NewA\;}} \\
\Delta ObjectSystemABC \\
n? : ClassSpecA \\
\hline
ObjectsA' = ObjectsA \cup \{n?\} \\
\hline
\end{array}
$$

Notice how we added to the schema ObjectSystemABC an attribute $ObjectsA : \mathbb{P}\, ClassSpecA$. This paragraph and the associated schemas should be taken as a parenthesis since our goal here is just to specify the global properties of an object system. We will hence continue using axiomatic definitions instead of schemas for the global object system, which makes most descriptions lighter and easier to read, as long as we do not plan to model how the system state can change.

2.8. Dereferencing attributes

An essential operation in object systems is to obtain the information held by the data structure pointed at by an object reference. This operation, called ”dereferencing” can be modeled in our case on a per attribute basis. We prefix the attribute name by the string ”get”, and promote the first attribute letter to upper case to name the accessor (”power” becomes ”getPower”). Following is an example in our ABC toy problem :

$$
\begin{array}{|l}
getA : A \longrightarrow \mathbb{N} \\
\hline
\forall\, i : A \bullet \\
\qquad getA(i) = (\mu\, v : \{s : ClassSpecA \mid s.ref = i \bullet s.a\} \cup \{s : ClassSpecC \mid s.ref = i \bullet s.a\} \bullet v)
\end{array}
$$

This definition uses Z's *mu* construct $(\mu\, x : T \mid C \bullet E)$ that yields the value of E on the unique x from T matching C. Again, it can be seen as a little verbose, as a result of Z's non object orientedness. However, it is easily specified, and such definitions can be generated automatically from shortcut descriptions.

2.9. Making a class abstract

Now, based on the same example, if we expect the class A to be abstract (i.e. we forbid an individual to be created as an A) we simply need to add a constraint stating that $instances(ClassX)$ is empty : $instances(ClassA) = \{o : ClassSpecA \mid class = ClassA \bullet o.i\} = \emptyset$.

2.10. Unused objects

The specification made so far accepts that elements of *ObjectReference* are members of none of the subtypes. Depending upon the situation (e.g. whether a constraint programming tool using the specification must try giving a type to all the elements in *ObjectReference* or not), we may force objects to belong to types. This is obtained by adding the axiom

$$\langle instances(ClassA), instances(ClassB), instances(ClassC)\rangle \text{ partition } ObjectReference$$

2.11. Specializing across several discriminators

An important concept in object oriented specification is the possibility to specialize a class across two different discriminators, each corresponding to different viewpoints over a class. For instance, a traditional real life example is the class *Vehicle*. It can be specialized in one discriminator, called ”energy”, related to the energy used to power the vehicle. We may imagine the subclasses *Human*(powered), *Wind*(powered), *Gas*(powered) in that discriminator. Each subclass in this case brings its own data attributes : number of humans, number of sails, tank capacity. The *Vehicle* class can also be specialized across another discriminator : the element it moves on. We can imagine here the classes : *Water*, *Ground*, *Air*. Again, each of these classes may carry some data, in isolation from the others. In the declaration of types, it suffices to state the following (where everything irrelevant has been omitted):

$$\begin{array}{|l}
Vehicle, Human, Wind, Gas, Water, Ground, Air : ReferenceSet \\
\hline
Vehicle = Human \cup Wind \cup Gas \\
Vehicle = Water \cup Ground \cup Air \\
instances(ClassHuman) = \{o : ClassSpecHuman \mid o.class = ClassHuman \bullet o.ref\} \ldots \\
\forall\, i : instances(ClassHuman) \bullet (\exists_1\, x : ClassSpecHuman \bullet x.ref = i) \ldots
\end{array}$$

The rule in the UML is that whenever such a multiple discriminator specialization occurs, the main class (here *Vehicle*) and all its child classes (i.e. $Human, \ldots Air$) are abstract, and that any concrete class underneath must inherit at least one class from each discriminator. This is so because since vehicle is partitioned in two discriminators, any ”Vehicle” must belong to some type among each discriminator. Obtaining this requires that each subclass inherits a class from each discriminator. The predicate stated in the axiomatic definition above ensure this: any object reference in a ”sub”subclass of Vehicle must be a member of at least one set among $Human, Wind, Gas$, and of at least one set among $Water, Ground, Air$. Membership to those sets is acquired through inheritance.

2.12. Shortcut notation for class specifications

Z being non object oriented in any way, the previous class and type declarations are verbose. For simplicity and readability, although not sacrificing rigor, we propose the following shortcut definition for classes and types, which makes use of the keywords *class*, *abstract*, *discriminator*, *inherit*. The syntax for this can be presented using simple examples, which must be understood as a shortcut for the corresponding specifications, as was previously described.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{class} - A : \mathit{abstract}} \\
-\mathit{discriminators} : \mathit{default} \\
a : \mathbb{N} \\
\hline
a < 10; \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{class} - B : \mathit{concrete}} \\
-\mathit{inherit} : A - \mathit{default} \\
b : \mathbb{N}_1 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{class} - C : \mathit{concrete}} \\
-\mathit{inherit} : A \\
\hline
a \geq 5; \\
\hline
\end{array}$$

A preprocessor can very easily parse such definitions, or take its input from an UML class design, so as to produce a listing identical to what was built step by step for the *ABC* example in the previous pages. Hence, a byproduct of these declarations is the declaration in the Z specification of the schemas : *ObjectDef*, *ClassDefA*, *ClassSpecA*, *ClassDefB*, . . ., and of the sets $instances(ClassA)$, A, $instances(ClassB)$, ...

In the case of the *Vehicle* class hierarchy, since it has two discriminators, we would declare (all irrelevant information being hidden):

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{class} - \mathit{Vehicle} : \mathit{abstract}} \\
-\mathit{discriminators} : \mathit{powermode}, \mathit{element} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{class} - \mathit{Human} : \mathit{abstract}} \\
-\mathit{inherit} : \mathit{Vehicle} - \mathit{powermode} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{class} - \mathit{Ground} : \mathit{abstract}} \\
-\mathit{inherit} : \mathit{Vehicle} - \mathit{element} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{class} - \mathit{Bicycle} : \mathit{concrete}} \\
-\mathit{inherit} : \mathit{Human} \mid \mathit{Ground} \\
\hline
\end{array}$$

The concrete *Bicycle* class inherits from both the *Human* and *Ground* classes.

3. Relations

Z provides a rich toolkit to define relations and reason about them. This feature is inherent in relational languages, where all common mathematical concepts, like functions, bags, sequences derive from relations through composition and constraints. For instance, a *function* is a relation bound by an axiom of unicity. Also, a *sequence* is a function from a subset of natural numbers $\mathbb{N}$ to a given set.

3.1. A simple example

Having defined the structure and inheritance relations between classes, we must now describe their relations. Like before, we will study this through a concrete example, based on two classes *Person* and *Company*.

$$\mathit{class} - \mathit{Person}$$

$$\mathit{class} - \mathit{Company}$$

This specification implicitly defines schemas:

$$\begin{array}{l}
\mathit{ClassDefPerson} \mathrel{\widehat{=}} [\ldots \mid \ldots] \\
\mathit{ClassDefCompany} \mathrel{\widehat{=}} [\ldots \mid \ldots] \\
\mathit{ClassSpecPerson} \mathrel{\widehat{=}} [\ldots \mid \ldots] \\
\mathit{ClassSpecCompany} \mathrel{\widehat{=}} [\ldots \mid \ldots]
\end{array}$$

as well as the appropriate constraints on $\mathit{instances}(\mathit{ClassPerson}), \mathit{instances}(\mathit{ClassCompany})$ and also the type sets:

$$\mathit{Person}, \mathit{Company} : \mathit{ReferenceSet}$$

3.2. Relations and roles

A relation is declared between types, no matter what the creation type of the objects is. In our example, we may think about these three relations :

$$\begin{array}{l}
\mathit{worksFor} : \mathit{Person} \leftrightarrow \mathit{Company} \\
\mathit{owns} : \mathit{Person} \leftrightarrow \mathit{Company} \\
\mathit{manages} : \mathit{Person} \leftrightarrow \mathit{Company}
\end{array}$$

In standard object oriented modeling [14], relation names are complemented by role names, associated with each extremity of a class relation. Each role name denotes the target class role wrt. the particular relation. Role names must be specified by the object model. When not ambiguous (i.e. when only one relation binds two given classes), the target class name is implicitly accepted as a role name. Roles of binary relations can be axiomatically defined as follows :

$$\begin{array}{l}
\mathit{employees} : \mathit{Company} \longrightarrow \mathbb{P}\, \mathit{Person} \\
\mathit{employer} : \mathit{Person} \longrightarrow \mathbb{P}\, \mathit{Company} \\
\hline
\forall\, c : \mathit{Company} \bullet \mathit{employees}(c) = \{p : \mathit{Person} \mid (p \mapsto c) \in \mathit{worksFor}\} \\
\forall\, p : \mathit{Person} \bullet \mathit{employer}(p) = \{c : \mathit{Company} \mid (p \mapsto c) \in \mathit{worksFor}\}
\end{array}$$

Note that the Z syntax allows more compact definitions for the roles *employer* and *employees* :

$$\begin{array}{l}
\mathit{employees}(c) = \mathrm{dom}(\mathit{worksFor} \rhd \{c\}) \\
\mathit{employer}(p) = \mathrm{ran}(\{p\} \lhd \mathit{worksFor})
\end{array}$$

or also

$$\begin{array}{l}
\mathit{employees}(c) = \mathit{worksFor}^{\sim} (\!| \{c\} |\!) \\
\mathit{employer}(p) = \mathit{worksFor} (\!| \{p\} |\!)
\end{array}$$

where $\mathit{worksFor}^{\sim}$ denotes the relational inverse of *worksFor*, $\mathit{worksFor} \rhd \{c\}$ denotes the range restriction of *worksFor* wrt. $\{c\}$ (which is still a relation), $\{p\} \lhd \mathit{worksFor}$ denotes the domain restriction of *worksFor* wrt. $\{p\}$, $\operatorname{dom} R$ denotes the domain of R, and $\_(\!|\_|\!)$ is the relational image operator. We may ease the pain of declaring roles for all the relations in a model by generically defining the *lrole* and *rrole* Boolean functions as follows.

$$
\begin{array}{l}
[C, D] \\
\hline
lrole : \mathbb{P}((C \leftrightarrow D) \times (D \longrightarrow \mathbb{P}\, C)) \\
rrole : \mathbb{P}((C \leftrightarrow D) \times (C \longrightarrow \mathbb{P}\, D)) \\
\hline
\forall R : C \leftrightarrow D;\ l : D \longrightarrow \mathbb{P}\, C \bullet (R, l) \in lrole \Leftrightarrow (\forall d : D \bullet l(d) = R^{\sim}(\!|\{d\}|\!)) \\
\forall R : C \leftrightarrow D;\ r : C \longrightarrow \mathbb{P}\, D \bullet (R, r) \in rrole \Leftrightarrow (\forall c : C \bullet r(c) = R(\!|\{c\}|\!)) \\
\hline
\end{array}
$$

The previous axiomatic definition is *generic*, parameterized with types (this is the first time we use this). Now, the declaration of the roles associated to the relation *worksFor* can be simplified :

$$
\begin{array}{|l}
employees : Company \longrightarrow \mathbb{P}\, Person \\
employer : Person \longrightarrow \mathbb{P}\, Company \\
\hline
(worksFor, employees) \in lrole \\
(worksFor, employer) \in rrole
\end{array}
$$

It is also possible to generically (pre)define two roles for any arbitrary binary relation as follows :

$$
\begin{array}{l}
[p, c] \\
\hline
leftRole : (p \leftrightarrow c) \times c \longrightarrow \mathbb{P}\, p \\
rightRole : (p \leftrightarrow c) \times p \longrightarrow \mathbb{P}\, c \\
\hline
\forall R : p \leftrightarrow c;\ vc : c \bullet leftRole(R, vc) = R^{\sim}(\!|\{vc\}|\!) \\
\forall R : p \leftrightarrow c;\ vp : p \bullet rightRole(R, vp) = R(\!|\{vp\}|\!) \\
\hline
\end{array}
$$

These definitions illustrate the amazing power of Z for defining builtin syntax extensions, as well as the richness of the relational operator toolkit of Z, later useful for specifying object constraints. It must be noted that from our viewpoint, Z offers a clear advantage over other object oriented [18], or terminological languages [3, 29] for object oriented constraints wrt. a potential broad acceptance, since relation definition is not role centered, but relations, functions and roles can freely coexist.

3.3. Composition, aggregate relations

Object modeling leads to a clear separation between two broad categories of relations. General relations are unconstrained, meaning that every tuple can be accepted, regardless of the number of times an object appears on either side. For instance, in modeling a network of PCs and printers, any PC can view any number of printers (even though it may not see all of them), and any printer can be accessed by any number of PCs. No limitation stems from the nature of the relation itself.

Other relations are more constrained. For instance, no PC can share its mainboard. This is an example of a *composition* relationship. To distinguish between both just involves changing the type of the relation to make it a function of a special kind. If a relation stated between the composite type and the component type (in this sequence) is a composition one, it means that its relational inverse is an injective partial function (each component occurs in at most one composite). If no component can be left aside, the relational inverse is injective. Z provides various notations for constrained functions : injections start with an arrow (↣, ⤔), surjections end with a double arrow (↠, ⤀), partial functions have a bar in the middle (⤔, ⤀), bijections are both injective and surjective (⤖), whereas unconstrained functions are denoted with a simple arrow (⟶) and standard relations have two opposed arrows (⟷).

$$\begin{array}{|l}
\mathit{uses} : \mathit{PC} \leftrightarrow \mathit{Printer} \\
\mathit{hasMainBoard} : \mathit{PC} \leftrightarrow \mathit{MainBoard} \\
\hline
\mathit{hasMainBoard}^{\sim} \in \mathit{MainBoard} \mathrel{\rightarrowtail\!\!\!\!\!\!\!+} \mathit{PC}
\end{array}$$

If components cannot be optional, the injection becomes non partial

$$\begin{array}{|l}
\mathit{hasDVDWriter} : \mathit{PC} \leftrightarrow \mathit{DVDWriter} \\
\hline
\mathit{hasDVDWriter}^{\sim} \in \mathit{DVDWriter} \rightarrowtail \mathit{PC}
\end{array}$$

In the most constrained case, of a strict one to one relation between types, the relation becomes a bijection, which can be formulated as follows :

$$\begin{array}{|l}
\mathit{hasMainBoard} : \mathit{PC} \mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} \mathit{MainBoard}
\end{array}$$

More generally, any constraint can be stated upon a relation using general quantified formulas and all of Z's constructs. The distinction made in the UML between aggregate and standard relations is conceptual, and does not relate to constraints in our sense here (aggregate relations model associations where a dynamic, not structural dependency exists among between objects[3]).

3.4. Multiplicities

Relation multiplicities can be naturally stated as well. Object models often constrain for a given relation the number of related target objects for each source object. For instance, a PC has at most four memory units (the $\#$ operator denotes set cardinality) :

$$\begin{array}{|l}
\mathit{hasMemory} : \mathit{PC} \leftrightarrow \mathit{Memory} \\
\hline
\forall \mathit{pc} : \mathit{PC} \bullet \#(\ \mathit{hasMemory}\llparenthesis\{\mathit{pc}\}\rrparenthesis\ ) \leq 4
\end{array}$$

3.5. Ordered relations

Object models sometimes require that the tuple ordering is significant. For instance, should we model the relation between polygons and points, it is clear that we need a list, not a set, of points to describe the Polygon. The concept available in Z to model this is the *sequence*.

$$\begin{array}{|l}
\mathit{builds} : \mathit{Polygon} \longrightarrow (\mathrm{seq}\,\mathit{Point})
\end{array}$$

To restrict the multiplicity in this case (for instance to describe all the pentagons) requires a little different work than before

$$\begin{array}{|l}
\mathit{builds} : \mathit{Polygon} \longrightarrow (\mathrm{seq}\,\mathit{Point}) \\
\hline
\forall p : \mathit{Polygon} \bullet \#\mathit{builds} = 5
\end{array}$$

To ensure that an object does not occur twice or more in a sequence, we need to make the sequences injective:

$$\begin{array}{|l}
\mathit{builds} : \mathit{Polygon} \longrightarrow (\mathrm{iseq}\,\mathit{Point})
\end{array}$$

To decide that a Point in our example does not occur in the definition of two or mode different Polygons, we state :

$$\begin{array}{|l}
\mathit{builds} : \mathit{Polygon} \longrightarrow (\mathrm{iseq}\,\mathit{Point}) \\
\hline
\forall p_1, p_2 : \mathit{Polygon} \bullet \mathit{items}\,(\mathit{builds}(p_1)) \cap \mathit{items}\,(\mathit{builds}(p_2)) = \varnothing
\end{array}$$

[3] For instance, the relations between a paragraph and a text is of that kind. Translating a paragraph in a text amounts to translating all its characters

3.6. Reified associations

An important feature in object oriented modeling is the possibility to attach extra information to associations in the model. This added information is carried by an *association class*, which can be a standard class (i.e. with a name) or be anonymous. We can however assume the existence of a name since the association class for a given relation *R* can be named automatically (e.g. as *R_DATA*)).

We thus expect the association class used in coordination with a given relation to be properly defined as a class according to the former framework. If we return to the *worksFor* example, we see that an obvious related information can be the salary (the salary can be different if a person works for two different companies, hence it cannot be an information carried by the Person itself).

$$\begin{array}{|l}
\hline
\mathit{class} - \mathit{EnrolmentInfo} : \mathit{concrete} \\
\mathit{salary} : \mathbb{N} \\
\hline
a > \mathit{MIN\_SALARY}; \\
\hline
\end{array}$$

This definition yields as usual two schemas : *ClassDefEnrolmentInfo*, *ClassSpecEnrolmentInfo*, and two sets : $\mathit{instance}(\mathit{ClassEnrolmentInfo})$ and *EnrolmentInfo*. The latter is the type associated with objects built as members of *EnrolmentInfo* itself or subclasses. Binding the enrolment information to the *worksFor* relation is the fact of a function from *worksFor* to *EnrolmentInfo*.

$$\begin{array}{|l} \mathit{mapEnrolmentInfo} : \mathit{worksFor} \longrightarrow \mathit{EnrolmentInfo} \end{array}$$

If the attached information is optional, the function is partial :

$$\begin{array}{|l} \mathit{mapOptionalEnrolmentInfo} : \mathit{worksFor} \rightarrow\!\!\!\!\!+\; \mathit{EnrolmentInfo} \end{array}$$

4. Constraints

4.1. Structural constraints: example

Constrained object oriented problems abound with constraints spanning across the object structure, traversing relations to gather information. The Z notation again offers many possible ways to state such constraints. Now let's study an example, classical in the configuration community : the model describes all valid PC's, composed from standard components, in a simplified form.

We declare the following classes : *PC*, *PowerSupply*, *MainBoard*, *Monitor*, *Processor*. Except for *PowerSupply* and *PC* all the classes inherit an abstraction called *Device*, with an attribute called *powerUsed*. *PowerSupply* has an attribute called *power*. We also have the relations *PC_PowerSupply*, *PC_MainBoard*, *PC_Monitor* and *MainBoard_Processor*. The shortcut definitions for these classes are :

$$\begin{array}{|l}
\hline
\mathit{class} - \mathit{Device} : \mathit{abstract} \\
\mathit{powerUsed} : \mathbb{N} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{class} - \mathit{PC} : \mathit{concrete} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{class} - \mathit{PowerSupply} : \mathit{concrete} \\
\mathit{power} : \mathbb{N} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{class} - \mathit{MainBoard} : \mathit{concrete} \\
-\mathit{inherit} : \mathit{Device} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{class} - \mathit{Processor} : \mathit{concrete} \\
-\mathit{inherit} : \mathit{Device} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{class} - \mathit{Monitor} : \mathit{concrete} \\
-\mathit{inherit} : \mathit{Device} \\
\hline
\end{array}$$

We also declare the composition relations (assuming default role names) *PC_PowerSupply*, *PC_MainBoard*, *PC_Monitor* and *MainBoard_Processor*[4]

$$\begin{array}{|l}
\mathit{PC\_PowerSupply} : \mathit{PC} \leftrightarrow \mathit{PowerSupply} \\
\mathit{PC\_Monitor} : \mathit{PC} \leftrightarrow \mathit{Monitor} \\
\mathit{PC\_MainBoard} : \mathit{PC} \leftrightarrow \mathit{MainBoard} \\
\mathit{MainBoard\_Processor} : \mathit{MainBoard} \leftrightarrow \mathit{Processor} \\
\hline
\mathit{PC\_PowerSupply}^{\sim} \in \mathit{PowerSupply} \rightarrowtail \mathit{PC} \\
\mathit{PC\_Monitor}^{\sim} \in \mathit{Monitor} \rightarrowtail \mathit{PC} \\
\mathit{PC\_MainBoard}^{\sim} \in \mathit{MainBoard} \rightarrowtail \mathit{PC} \\
\mathit{MainBoard\_Processor}^{\sim} \in \mathit{Processor} \rightarrowtail \mathit{MainBoard}
\end{array}$$

Now, we wish to state the constraint that the total power delivered by a PowerSupply must exceed the total power demand by all the devices in the PC. This is a classical example of structural constraint. To achieve this, we must define several utilities.

4.2. Structural constraints utilities

We need several intermediate definitions useful to declare constraints. For instance, some integer arithmetic functions generalize to the case of bags of natural numbers, or numerals. Some of the properties of object systems require to gather some information over the structure (like the *price*, or the *power* used by electrical units for instance). Such information is best represented in bags, which allow repeated occurrences of the same value. Given a bag, we may ask for its min, max, or sum for instance. We detail these three functions, which may serve as a template for possible others. The concept of ”gathering” as implemented by the forthcoming definitions corresponds to the ”*select*” and ”*collect*” operators in the OCL constraint language defined in the UML ([11] details how OCL can be used to describe configuration constraints).

In the rest of the sub section, we also provide a function that can be used to generate a sequence from a set. Since converting from bags to sets is immediate, and converting from sequences to bags is predefined in Z by the function *items*, this allows to convert any structure type into any other.

computing the min and max over a bag of naturals

$$\begin{array}{|l}
\mathit{bagmin} : \mathrm{bag}\,\mathbb{N} \longrightarrow \mathbb{N} \\
\mathit{bagmax} : \mathrm{bag}\,\mathbb{N} \longrightarrow \mathbb{N} \\
\hline
\forall\, b : \mathrm{bag}\,\mathbb{N} \bullet \mathit{bagmin}(b) = \mathit{min}\,(\mathrm{dom}\, b) \\
\forall\, b : \mathrm{bag}\,\mathbb{N} \bullet \mathit{bagmax}(b) = \mathit{max}\,(\mathrm{dom}\, b)
\end{array}$$

summing up the elements in a bag of naturals

$$\begin{array}{|l}
\mathit{bagsum} : \mathrm{bag}\,\mathbb{N} \longrightarrow \mathbb{N} \\
\hline
\mathit{bagsum}(\varnothing) = 0 \\
\forall\, b : \mathrm{bag}\,\mathbb{N} \mid (\mathrm{dom}\, b) \neq \varnothing \bullet \\
\qquad (\mathbf{let}\ x == \mathit{bagmin}(b) \bullet \mathit{bagsum}(b) = b(x) * x + \mathit{bagsum}(b \setminus \{(x, b(x))\}))
\end{array}$$

[4] we intentionally continue to read the relations from composite to components to emphasize the fact that composition is a constraint

To understand this definition of *bagsum*, it suffices to recall that a bag is a partial function from a set to strictly positive integers (the number of times an element is counted).

conversion functions

We define a conversion function from totally ordered finite sets to sequences. This function *asSeq* converts a finite set to a sequence, which may be further converted to a bag using the *items* operator on sequences. This provides full possibilities of converting from a container type to another. We need a function to select a member from a set. This is possible deterministically for totally ordered sets, as are ℕ or the set of rational numbers. We present the specification of the conversion function *asSeq* in the case of natural numbers. This gives a template for the definition of similar conversion functions applying to totally ordered sets of non integral elements.

$$\begin{array}{|l}
\mathit{asSeq} : \mathbb{F}\,\mathbb{N} \longrightarrow \mathrm{seq}\,\mathbb{N} \\
\hline
\mathit{asSeq}(\varnothing) = \langle\rangle \\
\forall\, S : \mathbb{F}_1\,\mathbb{N} \bullet (\mathbf{let}\ x == (\mathit{max}\,S) \bullet \mathit{asSeq}(S) = \mathit{asSeq}(S \setminus \{x\}) \cup \{\#S \mapsto x\})
\end{array}$$

building a bag of attribute values

Together with any attribute, we know that we can specify an accessor function which, given an object reference *i*, will return the attribute value held by the object structure mapped to *i*. We have established the convention of naming *getXyz* the accessor function for attribute *xyz*. We generalize this concept to sets of object references. We want, given a set of object references, to build the bag of corresponding attribute values. In our example, we expect the following accessor functions to be implicitly defined from the class declaration as follows :

$$\begin{array}{|l}
\mathit{getPower} : \mathit{PowerSupply} \longrightarrow \mathbb{N} \\
\hline
\forall\, i : \mathit{Device} \bullet \mathit{getPower}(i) = (\mu\, v : \{s : \mathit{ClassSpecPowerSupply} \mid s.\mathit{ref} = i \bullet s.\mathit{power}\} \bullet v)
\end{array}$$

$$\begin{array}{|l}
\mathit{getPowerUsed} : \mathit{Device} \longrightarrow \mathbb{N} \\
\hline
\forall\, i : \mathit{Device} \bullet \mathit{getPowerUsed}(i) = \\
\qquad (\mu\, v : \{s : \mathit{ClassSpecMonitor} \mid s.\mathit{ref} = i \bullet s.\mathit{powerUsed}\} \cup \\
\qquad \{s : \mathit{ClassSpecProcessor} \mid s.\mathit{ref} = i \bullet s.\mathit{powerUsed}\} \cup \\
\qquad \{s : \mathit{ClassSpecMainBoard} \mid s.\mathit{ref} = i \bullet s.\mathit{powerUsed}\ \} \bullet v)
\end{array}$$

We further make the assumption that the set ObjectReference is totally ordered[5], which allows to define a function called *pickFirst* yielding the first element of any finite set of ObjectReferences :

$$\begin{array}{|l}
\mathit{pickFirst} : \mathbb{F}\,\mathit{ObjectReference} \longrightarrow \mathit{ObjectReference}
\end{array}$$

From these accessors and the function *first*, we may form their generalized counterpart as (we use *bag* as a prefix to form the function names) :

$$\begin{array}{|l}
\mathit{bagPower} : \mathbb{F}\,\mathit{PowerSupply} \longrightarrow \mathrm{bag}\,\mathbb{N} \\
\hline
\mathit{bagPower}(\varnothing) = [\![\,]\!] \\
\forall\, d : \mathbb{F}_1\,\mathit{PowerSupply} \bullet (\mathbf{let}\ x == \mathit{pickFirst}(d) \bullet \\
\qquad \mathit{bagPower}(d) = (\mathit{bagPower}(d \setminus \{x\}) \uplus (\{\mathit{getPower}(x) \mapsto 1\})))
\end{array}$$

[5] This is realistic, since object references generally will be interpreted as integers (machine pointers *are* integers).

In the same spirit, we could define *bagPowerUsed* by simply replacing *PowerSupply* by *Device*, and *getPower* by *getPowerUsed* in the previous statement.

$$
\begin{array}{|l}
\mathit{bagPowerUsed} : \mathbb{F}\ \mathit{Device} \longrightarrow \mathrm{bag}\ \mathbb{N} \\
\ldots
\end{array}
$$

As in the case of association roles, it is possible to elaborate a generic definition for these :

$$
\begin{array}{|l}
[X] \\
\mathit{bagOf} : (\mathit{ObjectReference} \longrightarrow X) \longrightarrow (\mathbb{F}\ \mathit{ObjectReference} \longrightarrow \mathrm{bag}\ X) \\
\hline
\forall f : \mathit{ObjectReference} \longrightarrow X \bullet \mathit{bagOf}(f)(\varnothing) = [\![\,]\!] \\
\forall f : \mathit{ObjectReference} \longrightarrow X \bullet \\
\quad \forall d : \mathbb{F}_1(\mathrm{dom}\, f) \bullet (\mathbf{let}\ x == \mathit{pickFirst}(d) \bullet \\
\quad \mathit{bagOf}(f)(d) = (\mathit{bagOf}(f)(d \setminus \{x\}) \uplus (\{f(x) \mapsto 1\})))
\end{array}
$$

The function *bagOf* hence maps every function from *ObjectReference* to X to a function from sets of *ObjectReference* to bags of X.

4.3. Inter relation constraints

The basic constraint existing between relations is the subset constraint. A simple example is given by the two relations *worksFor* and *manages*, between the types *Person* and *Company*. The manager obviously works for the company, which is expressed as $\mathit{manages} \subset \mathit{worksfor}$. Hence the proper declaration for these relations becomes :

$$
\begin{array}{|l}
\mathit{worksFor} : \mathit{Person} \leftrightarrow \mathit{Company} \\
\mathit{manages} : \mathit{Person} \leftrightarrow \mathit{Company} \\
\hline
\mathit{manages} \subset \mathit{worksFor}
\end{array}
$$

4.4. Structural constraints

We wish to state the constraint that the total power delivered by the power supply suffices to feed all of the PC's devices. This can be stated as follows :

$$
\begin{array}{l}
\forall p : \mathit{PC} \bullet \\
(\mathbf{let}\ R == \mathit{PC\_Monitor} \cup \mathit{PC\_MainBoard} \cup \mathit{MainBoard\_Processor} \bullet \\
\quad \mathit{bagsum}(\mathit{bagOf}(\mathit{getPower})(\mathit{PC\_PowerSupply}(\!|\{p\}|\!))) \geq \\
\quad \mathit{bagsum}(\mathit{bagOf}(\mathit{getPowerUsed})(R^{+}(\!|\{p\}|\!) \cap \mathit{Device})))
\end{array}
$$

where R^{+} denotes the transitive closure of the relation R, obtained as the union of three relations, and $R^{+}(\!|\{p\}|\!)$ denotes the relational image of p, the PC composite, by R^{+}, hence the component objects of p, at any structural level.

4.5. Notational shortcuts for relations and roles

Most often, specifications require to make the structure traversal more explicit. To illustrate the possibilities offered by Z in that respect, we assume that all previous relations have roles named using a standard prefix ”the” followed by the distant class name (we use no *s* at the end, even when there can be several), as e.g. :

$$
\begin{array}{|l}
\mathit{theMonitor} : \mathbb{F}\ \mathit{PC} \longrightarrow \mathbb{F}\ \mathit{Monitor}
\end{array}
$$

Now, we define the operators $\rightarrow$, $\rightharpoonup$, $\cdot$, $\rightsquigarrow$ as shortcuts for the previous explicit definitions.

$$
\begin{array}{l}
[X] \\
\hline
\_ \to \_ : \mathbb{F}\ ObjectReference \times (ObjectReference \longrightarrow X) \longrightarrow \mathrm{bag}\, X \\
\_ \rightharpoonup \_ : \mathbb{F}\ ObjectReference \times (ObjectReference \longrightarrow X) \longrightarrow X \\
\_ \cdot \_ : \mathbb{F}\ ObjectReference \times (\mathbb{F}\ ObjectReference \longrightarrow \mathbb{F}\ X) \longrightarrow \mathbb{F}\ X \\
\_ \rightsquigarrow \_ : ObjectReference \times (\mathbb{F}\ ObjectReference \longrightarrow \mathbb{F}\ X) \longrightarrow \mathbb{F}\ X \\
\hline
\forall\, s : \mathbb{F}\ ObjectReference;\ r : ObjectReference \longrightarrow X \bullet s \to r = bagOf(r)(s) \\
\forall\, s : \mathbb{F}\ ObjectReference;\ r : ObjectReference \longrightarrow X \bullet s \rightharpoonup r = (\mu\, t : bagOf(r)(s) \bullet first\, t) \\
\forall\, s : \mathbb{F}\ ObjectReference;\ r : \mathbb{F}\ ObjectReference \longrightarrow \mathbb{F}\ X \bullet s \cdot r = r(s) \\
\forall\, o : ObjectReference;\ r : \mathbb{F}\ ObjectReference \longrightarrow \mathbb{F}\ X \bullet o \rightsquigarrow r = r(\{o\})
\end{array}
$$

Given a single object reference, ”$\rightsquigarrow$” dereferences a role, hence returning a set of target objects. The ”$\cdot$” dot operator does the same, given a set of object references. Hence, to denote the set of all processors connected to the motherboard of a given PC p, we write : $p \rightsquigarrow theMainBoard \cdot theProcessor$. Given a set of object references and an attribute ”$\to$” produces the bag of attributes values. ”$\rightharpoonup$” does the same, assuming the set contains a of object references contains a unique element. Now, the previous constraint relating the total power used to the power available can be reformulated as:

$$
\begin{array}{l}
\forall\, p : PC \bullet \\
\quad p \rightsquigarrow thePowerSupply \rightharpoonup getPower \geq \\
\quad p \rightsquigarrow theMonitor \rightharpoonup getPowerUsed + \\
\quad p \rightsquigarrow theMainBoard \rightharpoonup getPowerUsed + \\
\quad bagsum(p \rightsquigarrow theMainBoard \cdot theProcessor \to getPowerUsed)
\end{array}
$$

5. An AI Example

We wish to illustrate the use of the specification utilities presented so far with a simple yet very general artificial intelligence problem. [9] defines the problem of analyzing both the syntax and semantics of a context free language using a constraint object system. The language chosen is the archetypal language $\mathcal{L} = a^n\, b^n$ consisting in sequences of a's followed by the same number of b's. $aaabbb \in \mathcal{L}$, but $abbb \notin \mathcal{L}$. [9] proposes the object model and constraints described below to represent valid sentences of $\mathcal{L}$ together with their semantic. The object model is illustrated in figure 2. The program used to solve the problem is Ilog JConfigurator, an object oriented configurator. Given an input made of a sequence of words, some of them not being classified as a's or b's, the system can generate all the valid word sequences compatible with that input together with the correct syntax structure. The system works equally well when chunks of syntactical structure, or elements of the semantic, are fed in.

In other words, the system produces the following results (where inputs and outputs are sequences $\langle string, syntax\ tree, semantic \rangle$) (we use the dot character ”.” do denote an unknown word (a, or b), and the character ”?” to denote an unknown :

$$
\begin{array}{l}
\langle aaabbb, ?, ? \rangle \mapsto \langle aaabbb, S(SA, S(SA, S(SA, null, SB), SB), SB), 3 \rangle \\
\langle abbb, ?, ? \rangle \mapsto false \\
\langle .\, a\, .\, b, ?, ? \rangle \mapsto \langle aabb, S(SA, S(SA, null, SB), SB), 2 \rangle \\
\langle ?, ?, 2 \rangle \mapsto \langle aabb, S(SA, S(SA, null, SB), SB), 2 \rangle
\end{array}
$$

We propose here a rigorous, type checked specification of the object model and its constraints, that illustrate the power of the method. We start with the definition of several classes. *Word*[6] is an abstraction for *SA* and *SB* (representing a and b), *Cat* is an abstraction for both *Word* and *S*. *S* is the only syntactic construct, made of an *SA*, an optional enclosed *S*, and an *SB* in that order. The *Phrase* consists of a list of

[6] Following the terminology of natural language theories, we use ”phrase” to denote a valid sentence for the grammar, called a ”word” or a ”string” in formal language theory

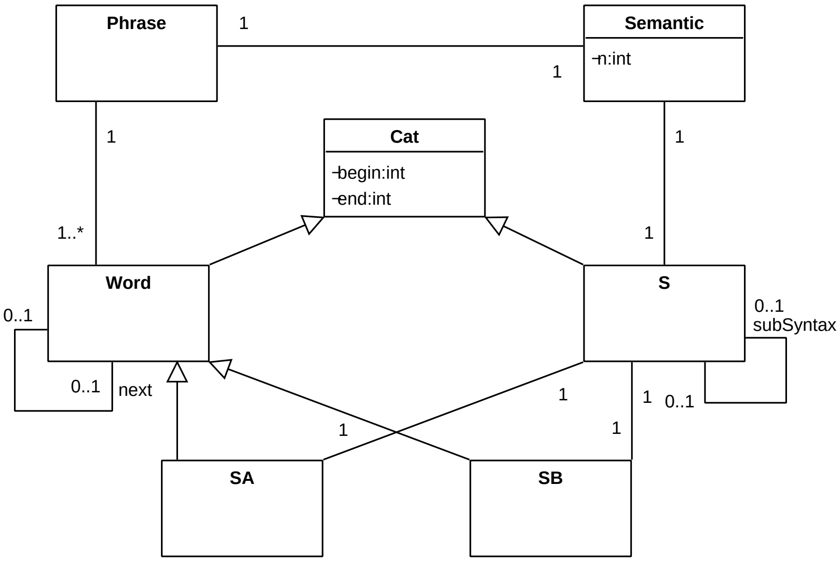

Figure 2. An object model for the $a^n b^n$ parsing problem

Word, a syntax *S*, and a *Semantic*. The semantic chosen is as simple as the example : it describes the count of a's and b's in the sentence.

$class - Phrase : concrete$

$class - Cat : abstract$
$begin : \mathbb{N}$
$end : \mathbb{N}$

$class - Word : abstract$
$-inherit : Cat$

$class - S : concrete$
$-inherit : Cat$

$class - SA : concrete$
$-inherit : Word$

$class - SB : concrete$
$-inherit : Word$

$class - Semantic : concrete$
$n : \mathbb{N}$

Several relations exist in this problem. They can be described very naturally by their most used roles. Whenever the opposite role is needed, the inverse of the relation can be computed. Each phrase maps to a unique first word. Each word maps to a unique phrase. Each word has an optional next word. Each phrase bijectively maps to a semantic. It also maps to a unique syntax S. Each SA (and SB) bijectively maps to an S. Each S has an optional enclosed S (we use a partial injection here). Each S is in a one to one with its semantic. We also know that the first word is a member of the phrase words. All these elements can be formulated as :

$$
\begin{array}{|l}
\mathit{firstWord} : \mathit{Phrase} \longrightarrow \mathit{Word} \\
\mathit{phrase} : \mathit{Word} \longrightarrow \mathit{Phrase} \\
\mathit{next} : \mathit{Word} \pinj \mathit{Word} \\
\mathit{phraseSemantic} : \mathit{Phrase} \bij \mathit{Semantic} \\
\mathit{phraseSyntax} : \mathit{Phrase} \longrightarrow S \\
\mathit{SASyntax} : \mathit{SA} \bij S \\
\mathit{SBSyntax} : \mathit{SB} \bij S \\
\mathit{subSyntax} : S \pinj S \\
\mathit{semantic} : S \bij \mathit{Semantic} \\
\hline
\mathit{firstWord} \subset \mathit{phrase}^{\sim}
\end{array}
$$

In some constraints below, we expect the following functions to be implicitly defined :

$$
\begin{array}{|l}
\mathit{theSA} : \mathbb{F}\, S \longrightarrow \mathbb{F}\, \mathit{SA} \\
\mathit{theSB} : \mathbb{F}\, S \longrightarrow \mathbb{F}\, \mathit{SB} \\
\mathit{theSubS} : \mathbb{F}\, S \longrightarrow \mathbb{F}\, S \\
\mathit{thePhraseSyntax} : \mathbb{F}\, \mathit{Phrase} \longrightarrow \mathbb{F}\, S
\end{array}
$$

The following accessor functions are also implicitly defined :

$$
\begin{array}{|l}
\mathit{getBegin} : \mathit{Cat} \longrightarrow \mathbb{N} \\
\mathit{getEnd} : \mathit{Cat} \longrightarrow \mathbb{N} \\
\mathit{getN} : \mathit{Semantic} \longrightarrow \mathbb{N}
\end{array}
$$

Using these definitions, we may formulate the following axioms, necessarily verified by the object system. Of course, some or all of these axioms must be implemented as constraints in a working system. The length of words is one

$$\forall\, w : \mathit{Word} \bullet \mathit{getBegin}(w) + 1 = \mathit{getEnd}(w)$$

The start position of the first word in a phrase is 0

$$\forall\, p : \mathit{Phrase} \bullet \mathit{getBegin}(\mathit{firstWord}(p)) = 0$$

The first word in a phrase is the SA of its syntax (S)

$$\forall\, p : \mathit{Phrase} \bullet \mathit{firstWord}(p) = \mathit{SASyntax}^{\sim}(\mathit{phraseSyntax}(p))$$

Consecutive words have corresponding end/begin

$$\forall\, w : \mathrm{dom}\, \mathit{next} \bullet \mathit{getEnd}(w) = \mathit{getBegin}(\mathit{next}(w))$$

All a's are located left of all b's

$$\forall\, a : \mathit{SA};\ b : \mathit{SB} \bullet \mathit{getBegin}(a) < \mathit{getBegin}(b)$$

The beginning of an S is the beginning of its SA (and respectively with SB's and "ends").

$$
\begin{array}{l}
\forall\, s : S \bullet \mathit{getBegin}(s) = s \rightsquigarrow \mathit{theSA} \rightharpoonup \mathit{getBegin} \\
\forall\, s : S \bullet \mathit{getEnd}(s) = s \rightsquigarrow \mathit{theSB} \rightharpoonup \mathit{getEnd}
\end{array}
$$

The enclosed S is between the SA and the SB.

$$\begin{array}{l}
\forall s : \operatorname{dom} subSyntax \bullet getBegin(s) < s \leadsto theSubS \rightharpoonup getBegin \\
\forall s : \operatorname{dom} subSyntax \bullet getEnd(s) > s \leadsto theSubS \rightharpoonup getEnd
\end{array}$$

The end position of the SA plus the length of the enclosed S equals the start position of the SB

$$\begin{array}{l}
\forall s : \operatorname{dom} subSyntax \bullet \\
\qquad s \leadsto theSA \rightharpoonup getEnd + s \leadsto theSubS \rightharpoonup getEnd - s \leadsto theSubS \rightharpoonup getBegin = \\
\qquad s \leadsto theSB \rightharpoonup getBegin \\
\forall s : S \mid s \notin \operatorname{dom} subSyntax \bullet s \leadsto theSA \rightharpoonup getEnd = s \leadsto theSB \rightharpoonup getBegin
\end{array}$$

The semantic of a phrase is the semantic of its syntax

$$\forall p : Phrase \bullet phraseSemantic(p) = semantic(phraseSyntax(p))$$

The ”value” of the semantic of an ”S” is the integer division of the its length by two

$$\begin{array}{l}
\forall s : S \bullet getN(semantic(s)) = (getEnd(s) - getBegin(s)) \mathsf{~div~} 2 \\
\forall s : S \bullet getN(semantic(s)) \mathsf{~mod~} 2 = 0
\end{array}$$

Not all these axioms are independent of course. However, they formally describe all the valid object configurations that are instances, or solutions, of this object model. Provided the class definitions are properly expanded, or this expansion is simulated, all the constraints can be fully type checked by a Z type checker. Furthermore, these axioms can be input to a theorem prover, with the possibility of generating automatic or user assisted proofs for conjectures about the properties of the problem, or proofs that some constraints are mutually incompatible.

The same specification may also be converted automatically to a valid input for any practical configurator or object constraint program.

5.1. Formal proofs

All object models so specified naturally allow formal proofs to be made about the axiom set. Essential in that respect are redundancy proofs. In constraint systems, any axiom that can provably be inferred from the rest of the axioms can be safely ignored by an implementation, which hence remains correct. Also, redundant axioms can be added when their implementation as a constraint has a better propagation efficiency than the axioms it can be derived from. In this case, redundancy ensures that the resulting system remains complete.

We illustrate the possibility to establish formal proofs for our example. $\forall s : S \bullet getN(semantic(s)) \mathsf{~mod~} 2 = 0$ can be proved by induction on the height of the syntactical structure (or the value of the semantic ”n”). A sequent proof for height 1 (ie. for an ”S” having no enclosed ”S”, or in other words for the ”S” corresponding to the central ”AB”) is :

$$\begin{array}{l}
\forall s : S \mid s \notin \operatorname{dom} subSyntax \bullet s \leadsto theSA \rightharpoonup getEnd = s \leadsto theSB \rightharpoonup getBegin \\
\forall w : Word \bullet getBegin(w) + 1 = getEnd(w) \\
\hline
\forall s : S \mid s \notin \operatorname{dom} subSyntax \bullet getEnd(s) - getBegin(s) = 2
\end{array}$$

$$\begin{array}{l}
\forall s : S \mid s \notin \operatorname{dom} subSyntax \bullet getEnd(s) - getBegin(s) = 2 \\
\forall s : S \bullet getN(semantic(s)) = (getEnd(s) - getBegin(s)) \mathsf{~div~} 2 \\
\hline
\forall s : S \mid s \notin \operatorname{dom} subSyntax \bullet getN(semantic(s)) \mathsf{~mod~} 2 = 0
\end{array}$$

Now, we can formally prove that if the induction hypothesis is true for height n, it holds for height $n + 1$, hence for all n. We can first establish as a lemma that the length of an S equals 2 plus that of its subSyntax :

$$\begin{array}{l}
\forall s : \mathrm{dom}\, \mathit{subSyntax} \bullet \\
\quad s \leadsto \mathit{theSA} \rightharpoonup \mathit{getEnd} + s \leadsto \mathit{theSubS} \rightharpoonup \mathit{getEnd} - s \leadsto \mathit{theSubS} \rightharpoonup \mathit{getBegin} = \\
\quad s \leadsto \mathit{theSB} \rightharpoonup \mathit{getBegin} \\
\forall w : \mathit{Word} \bullet \mathit{getBegin}(w) + 1 = \mathit{getEnd}(w) \\
\hline
\forall s : \mathrm{dom}\, \mathit{subSyntax} \bullet \mathit{getEnd}(s) - \mathit{getBegin}(s) = \\
\quad 2 + s \leadsto \mathit{theSubS} \rightharpoonup \mathit{getEnd} - s \leadsto \mathit{theSubS} \rightharpoonup \mathit{getBegin}
\end{array}$$

which makes the proof of the induction step obvious.

6. Conclusion

We have presented the entire specification of an object oriented constraint system, which can be used to document and exchange constrained object models. We used Z as an underlying formal system which offers many advantages:

- Z has very simple and clean first order semantics.
- as a relational language, Z offers the richest possible ways of reasoning about relations, which is an essential aspect of constrained object systems.
- Z is freely extensible by introducing new operators, always backed by rigorous axiomatic definitions. This allows to attain the flexibility and readability of existing object oriented approaches.
- the Z language comes with a freely available type checker *f*UZZ, that allows to control both the syntax and the type conformance of specifications (this article is fully type checked using *f*UZZ).

Our goal was to capture as much of object oriented modeling as possible, using as a basis the widely accepted standard UML, while avoiding to produce a new avatar of an object oriented language. We also rejected the idea of using an existing one, since all existing formal object languages have their pros and their cons, which might have interfered with the general objective of producing a tool for communicating and discussing constrained object systems. Even though some of our choices may still be discussed, this can be made formally. Furthermore, the Z language being formally extensible at will, all the proposed generic operators can be viewed as a template, rather than a rule.

References

[1] Antonio Alencar and Joseph Goguen. Ooze: An object-oriented Z environment. In *Pierre America, editor, European Conference on Object Oriented Programming, Springer, Lecture Notes in Computer Science, Volume 512*, pages 180–199, 1991.

[2] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps–application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[3] F. Baader and B. Hollunder. A terminological knowledge representation system with complete inference algorithms. In *Proceedings of the First International Workshop on Processing Declarative Knowledge*, volume 572, pages 67–85, Kaiserslautern (Germany), 1991. Springer–Verlag.

[4] Rosalind Barden, Susan Stepney, and David Cooper. The use of Z. In J. E. Nicholls, editor, *Proceedings of the 6th Z User Meeting, York, UK, 1991*, Workshops in Computing, pages 99–124. Springer, 1992.

[5] Virginia Barker, Dennis O'Connor, Judith Bachant, and Elliot Soloway. Expert systems for configuration at digital: Xcon and beyond. *Communications of the ACM*, 32:298–318, 1989.

[6] P. Borba and J. Goguen. An operational semantics for FOOPS. In R. Wieringa and R. Feenstra, editors, *Working Papers of the International Workshop on Information Systems - Correctness and Reusabilility, IS-CORE'94. Technical Report IR-357*, Amsterdam, 1994.

[7] R. J. Brachman and J. G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(2):171 – 216, 1985.

[8] Yves Caseau. Constraint satisfaction with an object-oriented knowledge representation language. *Applied Intelligence*, 4(2):157–184, 1994.

[9] Mathieu Estratat and Laurent Henocque. Parsing languages with a configurator. In *proceedings of the European Conference for Artificial Intelligence ECAI 2004*, pages 591–595, Valencia, Spain, August 2004.

[10] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, Markus Stumptner, and Markus Zanker. Transforming uml domain descriptions into configuration knowledge bases. In *Knowledge Transformation for the Semantic Web*, pages 154–168, 2003.

[11] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. Configuration knowledge representation using uml/ocl. In *Proceedings of the conference UML 2002*, pages 49–62, 2002.

[12] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. Semantic configuration web services in the cawicoms project. In *Proceedings of the Configuration Workshop, 15th European Conference on Artificial Intelligence*, pages 82–88, Lyon, France, 2002. http://www.cawicoms.org/.

[13] Markus P. J. Fromherz, Vijay A. Saraswat, and Daniel G. Bobrow. Model-based computing: Developing flexible machine control software. *Artificial Intelligence*, 114(1-2):157–202, October 1999.

[14] Object Management Group. *UML v. 1.5 specification.* OMG, 2003.

[15] B. Jacobs. Coalgebras in specification and verification for objectoriented languages, 1999.

[16] Joxan Jaffar and Jean Louis Lassez. Constraint logic programming. In *in ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[17] Ulrich John and Ulrich Geske. Reconfiguration of technical products using conbacon. In *Proceedings of AAAI'99-Workshop on Configuration*, pages 48–53, Orlando, Florida, July 1999.

[18] Daniel Mailharro. A classification and constraint-based framework for configuration. *AI in Engineering, Design and Manufacturing, (12)*, pages 383–397, 1998.

[19] John P. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88, 1982.

[20] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI-90*, pages 25–32, Boston, MA, 1990.

[21] Alexander Nareyek. Structural constraint satisfaction. In *Papers from the 1999 AAAI Workshop on Configuration, Technical Report, WS-99-0*, pages 76–82. AAAI Press, Menlo Park, California, 1999.

[22] Harald Meyer nauf'm Hofe. Construct: Combining concept languages with a model of configuration processes. In *Papers from the 1999 AAAI Workshop on Configuration, Technical Report, WS-99-0*, pages 17–22, 1999.

[23] Kevin R. Plain. Optimal configuration of logically partitionned computer products. In *Proceedings of the Configuration Workshop, 15th European Conference on Artificial Intelligence*, pages 33–34, Lyon, France, 2002.

[24] Daniel Sabin and Eugene C. Freuder. Composite constraint satisfaction. In *Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[25] Graeme Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, in Advances in Formal Methods, 2000.

[26] Timo Soininen, Esther Gelle, and Ilkka Niemela. A fixpoint definition of dynamic constraint satisfaction. In *Proceedings of CP'99*, pages 419–433, 1999.

[27] Timo Soininen, Ilkka Niemela, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring Symp. on Answer Set Programming: Towards Efficient and Scalable Knowledge*, pages 195–201, March 2001.

[28] J. M. Spivey. *The Z Notation: a reference manual*. Prentice Hall originally, now J.M. Spivey, 2001.

[29] Markus Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2), June 1997.

[30] Markus Stumptner. Configuring web services. In *Workshop notes of the Configuration Workshop, European Conference on Artificial Intelligence ECAI'04*, 2004.

[31] Markus Stumptner, Gerhard Friedrich, and Alois Haselbck. Generative constraint-based configuration of large technical systems. *Artificial Intelligence in Engineering, Design, Analysis and Manufacturing (AI EDAM)*, 12(4), Special Issue on Configuration, December 1998.

[32] Katariina Ylinen, Tomi Mnnist, and Timo Soininen. Configuring software products with traditional methods - case linux familiar. In *Proceedings of the Configuration Workshop, 15th European Conference on Artificial Intelligence*, pages 5–10, Lyon, France, 2002.

Laurent Henocque
Laboratoire des Sciences de l'Information et des Systèmes
LSIS (UMR CNRS 6168)
Campus Scientifique de Saint Jérôme
Avenue Escadrille Normandie Niemen
13397 MARSEILLE Cedex 20
`laurent.henocque@lsis.org`