The ROCK & ROLL System User Manual

Release 4.0
December 3, 1998

N.W. Paton, M.L. Barja, A. Dinn, A.A.A. Fernandes, O. Liew
Database Research Group
Dept of Computing and Electrical Engineering
Heriot-Watt University

Contents

1 Introduction

The purpose of this document is to introduce the ROCK & ROLL database system. ROCK & ROLL is a deductive object-oriented database system which supports an imperative database programming language, ROCK, and a deductive query language, ROLL. The ROCK component can be used for schema definition and data manipulation, while ROLL can be used to express queries and to define declarative methods. Static type checking is supported throughout.

The current version of ROCK & ROLL has been implemented as an interpreter. There are two versions of the system: the main memory version *rnr* and the persistent version *prnr*. There is a top level shell which accepts commands to compile/execute a program, define persistent environments etc. The commands available at this level are described in Section 2. In Section 3 the general structure of a program is outlined briefly. ROCK is described in section 4 as the conjunction of data definition and data manipulation sublanguages. The facilities available for querying the database using ROLL are described in Section 5. The extensions that provide built-in support for vector spatial data handling are described in section 6. The constructs for defining persistent schemas and manipulating persistent data are described in Section 7. Section 8.1 contains a grammar for ROCK & ROLL.

2 Getting Started

2.1 Main Memory ROCK & ROLL

To invoke the main memory version of ROCK & ROLL type `rnr` at the Unix prompt. This brings you to the system prompt `rock&roll>`.

```
%rnr


ROCK & ROLL System main memory prototype version 4.0        November 98


rock&roll>
```

From here, it is possible to execute different commands. There is a help command which provides information on the available possibilities.

```
rock&roll> help
                      Help facility of the ROCK & ROLL system

The available commands are:


     run <prog>          -- typechecks and executes a program
     run <prog> <envs>   -- typechecks and executes a program
                            which requires environments as parameters
     define_env <env>    -- defines a new PERSISTENT environment
     delete_env <env>    -- deletes a PERSISTENT environment
     list_envs           -- lists the names of all PERSISTENT environments
     list_classes <env> -- lists the names of all classes in the PERSISTENT
                            environment
```

```
quit                -- quits the ROCK & ROLL system commiting changes
abort               -- quits the ROCK & ROLL system aborting changes
compile             -- compiles a ROLL method
list_compiled       -- lists details of a compiled ROLL method
```

The command `run` takes as parameters the name of a file containing a program written in ROCK & ROLL, followed by a number of environment names if the program is to use or populate one or more environments (the nature and use of environments is discussed in section 7). As an example of the creation and use of an environment, the following two commands create the environment `E`, and execute the program `parts.rnr` with respect to this environment:

```
rock&roll> define_env E
rock&roll> run parts.rnr E
```

An environment can be deleted with the command `delete_env`. This causes all the classes in that environment to be deleted. The command `list_envs` gives the list of existing environments, and `list_classes` can be used to display the names of classes in an environment.

`quit` and `abort` can both be used to exit the ROCK & ROLL session. All classes and instances created during the session are lost when the main memory version is exited.

2.2 Persistent ROCK & ROLL

Persistent versions of ROCK & ROLL make use of persistent C++ systems as a repository for ROCK & ROLL classes and instances – ROCK & ROLL is implemented in C++ [1]. Before running persistent ROCK & ROLL you must have installed the relevant storage manager software. This is not supplied with the ROCK & ROLL system.

To invoke the persistent version of ROCK & ROLL type `prnr` at the Unix prompt. This brings you to the system prompt `rock&roll>`.

```
%rnr


ROCK & ROLL System persistent prototype version 4.0         November 98


rock&roll>
```

The same commands are available as for the main memory version. In fact the persistent and main memory versions are identical in functionality with one exception: if the persistent ROCK & ROLL is exited using the quit command then classes and objects created during a persistent run will be commited to disk. This means they will still be resident in the system when persistent ROCK & ROLL is restarted. If the abort command is used to exit then all classes and instances created during the session are deleted and the database is left unchanged.

3 General Program Structure

The general structure of a program is:

[1] Early versions of ROCK & ROLL used EXODUS, later versions ObjectStore.

```
program(< list of parameters >)
begin
    < definitions >
    < declarations >
    < statements >
end
```

$<$ *list of parameters* $>$ corresponds to the list of persistent environments that the program has access to.

$<$ *definitions* $>$ is a list of type and class definitions. These types will be added to the system when the program is run. Class and type definitions are described in section 4.1.

$<$ *declarations* $>$ is a list of declarations for variables used in the main body of the program. These variables are accessible from the main program body while it is executing. Variable declarations are described in section 4.2

$<$ *statements* $>$ is a list of statements which constitute the main program body. These statements are executed when the program is run. Statements are described in section 4.2.

4 Definition and Manipulation of Data in ROCK

ROCK can be regarded as the conjunction of a data definition language for schema declarations, and a data manipulation language that allows operations to be performed on the objects stored in the database. ROCK also supports interaction with the user, and file-based I/O.

The data definition language provides support for the definition of an object oriented schema for the application, and distinguishes between types and classes thus:

- A *type* declaration describes the structural characteristics of a set of instance objects, and describes the operations that can be applied to these objects.
- A *class* definition specifies how the operations associated with a type are implemented. The structure of a class is specified by its type.

The data manipulation language is the vehicle used to implement the operations of a class through imperative method definitions, and for the development of application code in the form of top-level programs. The data manipulation language supports the creation, retrieval and manipulation of objects. The deductive aspects of ROCK & ROLL are discussed in section 5.

4.1 Data Definition Language

The data definition language is the sublanguage for declaring data structures. It provides means for specifying the schema of the database by supporting the semantics of the modeling constructs defined by the data model. The database schema defines the structure to which the objects in the database conform and the operations associated with these structures.

4.1.1 Type system

The structures defined by the data model are realised in the language as *object-type definitions*. Each object in the system can be categorised as belonging either to the set of primary objects or to the set of secondary objects. Primary objects are instances of primitive types and secondary objects are instances of classes whose structure is defined by an object-type. A *type declaration* associates a type name with an object-type definition, and takes the form:

type t : *typeexp* **end-type**

This declaration introduces the type t into the current scope, where t is defined by the type expression *typeexp*.

Primitive Types

Primitive types are the built-in types supported by the language. The primitive types are *int, real, string* and *bool*. The domains of values with which the primitive types are related are:

- The type *int* is associated with the subset of integer constants from the smallest to the largest integer defined by the implementation. The constant associated by default with objects of type *int* is `0`.
- The type *real* is associated with the subset of floating point constants defined by the implementation. The constant associated by default with objects of type *real* is 0.0.
- The type *string* is associated with the domain of all possible combinations of characters forming strings. The constant associated by default with objects of type *string* is the empty string represented as `""` (empty string).
- The type *bool* is associated with the domain of the two truth values *true* and *false*. The constant associated by default with objects of type *bool* is *false*.

Type aliasing

The programmer can define new types by aliasing existing types. Both the original type and the new type correspond to the same actual type, that is, they share the same instances. An alias type declaration takes the form:

type t_1 : t_2 **end-type**

where t_2 is the type for which the alias is being created, and t_1 the new type name. For example, the following declarations introduce the types `dd`, `mm` and `yy` as aliases for the type `int`:

```
type dd:int end-type
type mm:int end-type
type yy:int end-type
```

Object-types

The programmer can define new types by combining a number of primitive and/or user defined types to build an *object-type definition* or *type expression* which is then associated with a type name. An object type definition can have a number of different components, specifically, an *attributive expression*, an *inheritance expression*, a *construction expression*, and an *interface expression*, where any of these may be empty.

The *attributive expression* declares the properties of the objects of the type being defined. A type definition can have at most one attributive expression which describes one or more attributes. Each property or attribute is either a type name or else it is given by specifying a property name and a type. In the latter case, the property definition corresponds to an implicit type alias declaration. Attributes have an associated visibility which is either *private* or *public*, which determines the rights of access to the values of the property.

For example, the type `part` can be defined thus:

```
type part:
    properties:
      public:
          name: string;
    ...
end-type
```

In this example, the type `part` has a single property `name`, which is a type alias for `string`.

The *inheritance expression* allows for the sharing of structure and behaviour definitions between types. For example, it is possible to envisage a special kind of `part` known as a `base_part`, which represents atomic components. Every `base_part` is also a `part`, but is associated with additional properties, such as a `cost` and a `mass`. Such a type can be defined as follows:

```
type base_part:
    specialises: part;
    properties:
      public:
        mass    : real,
        cost    : real;
    ...
end-type
```

The type `base_part` is said to `specialise` the type `part`; every instance of `base_part` is also an instance of `part`, and every property of `part` is also a property of `base_part`.

Multiple inheritance is also supported in ROCK & ROLL. Multiple superclasses are specified by supplying a comma-separated sequence of class names after the `specialises` keyword.

The *construction expression* allows one type to be built from others. Three constructors exist, namely *association*, *sequentiation* and *aggregation*. A construction is associated with a *visibility* which takes as values one of *private* or *public*. This visibility determines whether the access and manipulation of the construction is supported from outside the associated class definition.

An *association* is effectively a set whose members all share the same primitive or object-type. The construction of an object constructed by association is a set of object identifiers of the members of the association. Associations are represented by specifying the type of the elements enclosed by the symbols "{" "}". For example, the type `composite_part` is defined below to be a subtype of `part` which is constructed from an association of `part` objects:

```
type composite_part:
    specialises: part;
    properties:
      private:
        assembly_cost : real,
        mass_increment: real;
    public {part};
    ...
end-type
```

The above definition allows for the construction of hierarchies of parts, where one part is made from a set of other parts, which may themselves be composite. Since the construction is `public` any code provided with a `composite_part` instance may access its members. If the construction was `private` then access to members of a `composite_part` would only be allowed from within methods defined on class `composite_part` or its subclasses.

A *sequentiation* plays both the role of a list and an array whose elements share the same primitive or object-type. The built-in operations associated with sequentiations allow members to be retrieved in order or according to their position. The lower and upper bounds of a sequentiation need not be specified when it is created, and the indexes specifying the positions of elements in the sequentiation need not be contiguous. A sequentiation is represented by specifying the type of the elements enclosed by the symbols "`[`" "`]`". For example, if a `composite_part` is to be represented as an ordered collection of `part` objects, then it can be defined thus:

```
type composite_part:
    specialises: part;
    properties:
      private:
        assembly_cost : real,
        mass_increment: real;
    public [part];
    ...
end-type
```

An *aggregation* is akin to a record or tuple. Each field of the aggregation is defined by a type name, or alternatively, a field can be given by specifying its name and its value range. In this case, as with attributes, the field definition corresponds to a type alias. Aggregations are represented by specifying the type of each coordinate enclosed by the symbols "`<`" "`>`". For example, the following declaration introduces the type `date`, which is an aggregate of a day, a month and a year:

```
type date:
    public <dd:int, mm:int, yy:int>;
end-type
```

Constructions have been introduced into the model to make explicit a distinction between the fundamental structural characteristic of a concept and other stored properties. For example, the principal structural feature of `composite_part` described above is the fact that it is constructed from a collection of other `part` objects. Thus a `composite_part` is declared to be an `association` (or `sequentiation`) of `part` objects. Other stored properties of `composite_part` are principally descriptive, and relate to such features as the cost of building the composite part from its components. Because the construction of a type is intended to capture its principal characteristics, *it is not possible for a single type to have more than one construction, either directly or by inheritance.*

The *interface expression* in an object-type definition consists of declarations of the public user-defined operation interfaces. Objects conforming to the structure given by the type being defined can be manipulated through the operations implemented for these interfaces. An operation interface consists of a name and a signature, and is prefixed by the name of the language which is to be used to implement the interface. The signature specifies the formal input parameters, their types, and optionally the type of the result that might be returned when the operation is invoked[2]. For example, the type declaration for `part` can be extended to introduce operations for computing the cost and mass of its instances thus:

[2]The signatures of ROLL methods are somewhat different from those in ROCK; for details see section 5.3.

```
type part:
    properties:
      public:
        name: string;
    ROCK:
        compute_mass(): real,
        compute_cost(): real;
end-type
```

In addition to user-defined operations which are applicable to instances, it is also possible to declare an operation called `new` for initialising instances of a type:

```
type base_part:
    specialises: part;
    properties:
      public:
        mass    : real,
        cost    : real;
    ROCK:
        new (n: string, m: real, i: bool, s: string, c: real);
        ...
end-type
```

Attributive, constructive, inheritance and interface expressions may make forward reference to types defined later in the current program. Reference to the type currently being defined is also allowed in attributive and interface expressions. Such reference is disallowed in constructive and inheritance expressions due to a more general well-foundedness requirement for constructions and inheritance. Constructive expressions must not give rise to cyclic chains of nested constructions. Inheritance expressions must not give rise to cyclic chains of subtypes.

Example 1 Object type declarations This example presents the type declarations for the bill of materials application which corresponds to the database schema in Figure 1. In schema diagrams, secondary object types are represented using rectangles, primary object types using ellipses, and operations using rectangles with rounded corners (none are shown in figure 1). Labeled directed edges represent modeling features thus: ◯ – attribution, △ – specialisation, ⊗ – aggregation, ⊛ – association, and ⊚ – sequentiation.

```
type part:
    properties:
       public:
         code: string;
     ROCK:
         compute_mass(): real,
         compute_cost(): real;
         print();
end-type

type base_part:
     specialises: part;
     properties:
       public:
         mass    : real, imported: bool, supplier: string,  cost: real;
```

```
    ROCK:
        new (cd: string, m: real, i: bool, s: string, c: real);
        compute_mass(): real;
        compute_cost(): real;
        print();
end-type

type composite_part:
     specialises: part;
     properties:
       private:
         assembly_cost : real,  mass_increment: real;
     public { part };
     ROCK:
          new(cd: string, a: real, m: real),
          compute_mass(): real,
          compute_cost(): real,
          print();
end-type
```

■

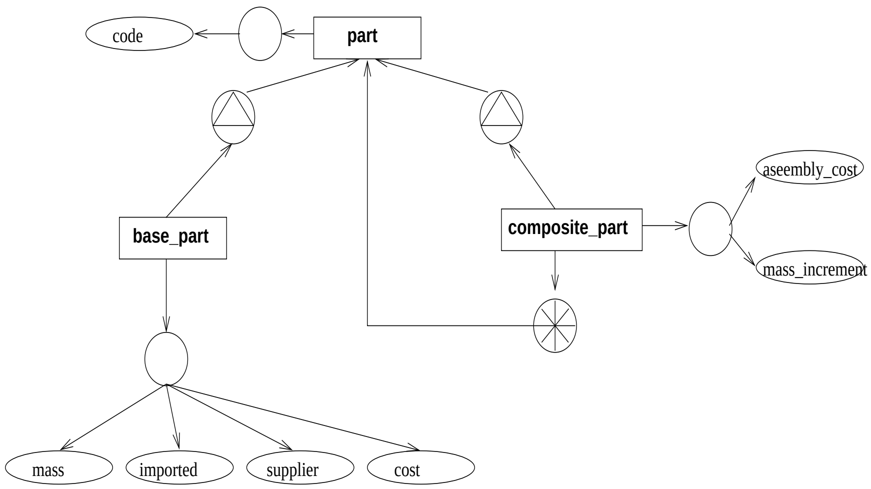


Figure 1: Diagrammatic representation for the *bill of materials* application database schema

4.1.2 Classes

A class definition provides implementations for the operations associated with a type. This type defines the structural information associated the class. Operations are implemented using either ROCK or ROLL. The relationship between types and classes is one-to-one, that is, each class is associated with a unique type and vice-versa. The definition of both public and private methods for a class constitutes the class definition or *class expression*.

The rationale for this partitioning of functionality between types and classes is that the type specifies the *interface* of a concept and the class specifies the *implementation*. The reason that the structural aspects of a type are considered part of the *interface* is that operations are generated by the system to enable access to and manipulation of the structural parts of an object. Thus the complete set of operations associated with a type can be deduced from the interface part of the type definition plus the public structural features of the type.

The visibility of a method is either `private` or `public`. The interfaces of public methods are specified in the definition of the type associated with a class. A class can, however, define private methods which are not mentioned in the corresponding type. Public methods are accessible from outside the methods defined on the same class, while private methods are not.

A class declaration associates a class name with a class expression, and takes the form:

class c: *classexp* **end-class**

This declaration introduces the class c into the current scope with definition *classexp*. A type with the same name must have been defined previously. For example, `date` can be extended with an interface section and then associated with a class as follows:

```
type date:
    public <dd, mm, yy>;
    ROCK:
        print();
end-type

class date
    public:
        print()
        begin
            write get_dd()@self, ":", get_mm()@self, ":", get_yy()@self;
        end
end-class
```

Further examples of class definitions are given in Example 2.

4.2 Data Manipulation Language

The data manipulation language ROCK is used in conjunction with ROLL (see section 5) for developing application code. Application code can be implemented as methods or in top level programs. This section presents an informal description of both the syntax and the semantics of ROCK.

4.2.1 Variable Declarations

A variable declaration is a binding between a name, a type and (optionally) an initial value for the variable. The standard syntax for variable declaration is:

var v: t

where v is the variable being declared and t is the name of the type of the variable v. Variables can be declared which store primary objects or object identifiers (*oids*) of secondary objects. The storage of the identifier of an object rather than the object itself in the variable means that an object is essentially *referenced by* the variable rather than *stored in* the variable.

As examples of variable declarations, an integer variable `i` is declared thus:

```
var i: int;
```

A variable that can store the object identifier of a `part` is declared thus:

```
var p: part;
```

Variables of primary type hav an initial default value at the point of declaration. `0` in the case of an int, `0.0` for a real, `""` for a string and `false` for a boolean. Variables of secondary type do not normally have a value until they are assigned.

An alternative form of variable declaration combines the introduction of a new variable with the specification of its initial value. This results in an implicit binding of an variable to a type which is inferred by the system. The general structure of this form of declaration is:

var v:= *objexp*

where *objexp* is an expression which is evaluated to yield an object. As a simple example, the following statements declare the variable *one* as an *int* with the initial value `1` and the variable p with a newly created part as its inital value:

```
var one := 1;
var p := new part();
```

A variable can be declared using an anonymous collection type of the form {c} or [c], where c is a type name. For example, in the following `intset` is declared to be a variable which references an association of integers and `intseq` is declared to be a variable which references a sequence of integers.

```
var intset: {int};
var intseq: [int];
```

A variable with an anonymous collection type can be initialised with a new empty collection object. The type of the elements of the relevant *association/sequentiation* must be given since the system cannot otherwise infer this type from the declaration. Examples are:

```
var v:= {}: int;
var p := []:part;
```

A variable can be initialised with a non-empty collection containing a definite set of initial values. In this case, the type of the elements of the *association/sequentiation* can be omitted and inferred by the system. For example, from the declaration :

```
var intset := {2,3,1};
```

the system infers that `intset` is an object constructed as an *association* of integers.

An initialiser for an anonymous sequence may also specify the indices at which the initial values are to be inserted into the sequence. For example, the following two declarations initialise variables with sequence objects containing the values `"one"`, `"two"` and `"three"` in positions `1`, `2` and `3`, respectively:

```
var aseq := ["one", "two", "three"];
var bseq := [(3,"three"),(2,"two"),(1,"one")];
```

Note that every time an initialiser for an variable of anonymous type is executed a new instance of the collection is created. Since the anonymous collection classes are not persistent these instances will be deleted at the end of execution of the program in which they are created.

Note also that collection initialisers which contain a list of values may only employ primitive object constants.

4.2.2 Method Invocation

In object-oriented systems, changes to the state of an object are generally performed through invocation of the operations defined for that object, usually referred to as methods. Methods are invoked by sending messages to objects. At run-time, the message is dynamically bound to a method depending on the class of the object receiving the message, and thus dynamic binding is supported.

A method invocation is a request sent to a secondary object for an operation to be performed, and takes the form:

$$methname(objexp_1,...,objexp_n)@objexp$$

where *methname* is the name of the method being invoked. The arguments are given by a, possibly empty, list of object expressions whose types must be compatible with those of the parameters defined for the interface of the method. The message sending operator is @, the second argument of which is an object expression (i.e. an expression which evaluates to yield a secondary object). As an example of a message-send, the following code fragment invokes the method `compute_mass` on the part `p`:

```
var p:part;
...
var theMass :=  compute_mass@p;
```

Message-sends can be nested, e.g. `print@get_thing@obj`.

Method invocations are regarded as object expressions for the purposes of type checking. So, a method invocation can be used wherever a value is expected e.g. as an operand to a builtin operation or an argument to another method invocation. If the method invocation is for a method which does not return a value then it has void type and the type checker will notify an error if it is used in an expression context.

An object can send messages to itself (*self reference*) and is in this way able to activate its own methods. In fact, *self* can be used as an object expression within method implementations. Its type is that associated with the class in which the method is defined. A message sent to *self* takes the form:

$$methname(objexp_1,...,objexp_n)@self$$

An object can also send messages to itself in the rôle of an instance of a superclass. This allows invocation of a method attached to a superclass when that method has been overridden, and is particularly useful when the method provided by the superclass is to be enhanced incrementally by addition of special behaviour supplied by a subclass. This kind of message takes the form:

$$methname(objexp_1,...,objexp_n)@super$$

and implies that the search for the operation starts in the superclass of the class in which the construct is used. If there is ambiguity as to the class of the most specialised definition of a method which can respond to a message sent to `super`, then the class at which search for the operation starts must be given explicitly:

$$method\text{-}name(objexp_1,...,objexp_n)@super(c)$$

4.2.3 System-generated methods

For each property (and for each coordinate in an aggregation) given in the type expression of a type declaration, the system generates a pair of methods whose names are the name of the corresponding attribute with the prefixes *get_* and *put_* respectively. These methods allow the value referenced by a property to be retrieved or updated. The visibility of these methods is determined by that of the property (or construction) from which they are derived. Overriding of these methods is not allowed.

The method *get_* returns the object referenced by the property name in the recipient object. The method *put_* updates the corresponding property of the object to which the method is sent with the property reference passed as parameter. For example, assuming the type definitions in Example 1, the following program updates the cost of every `base_part` in the database by 10%:

```
foreach b in base_part do put_cost(get_cost()@b*1.1)@b;
```

4.2.4 Built-in operators

Built-in operators are those provided by the language. There are two groups of built-in operators: built-in operators on primary objects and built-in operators secondary on objects. There follows a description of the operators in each of these groups.

Built-in operators on primary objects

Built-in operators on primary objects together with their signatures are summarised in Table 1. Five groups of operators are defined on primary objects: *arithmetic operators*, *comparison operators*, *string* *operators*, *boolean operators* and *math operators*. The comparison operators and most of the arithmetic and math operators are overloaded. When an overloaded operator is applied to an integer and a real, the integer is coerced to real, and if the operator is arithmetic or math, the result is a real. The first four signatures in the table are valid for any of the first four arithmetic operators. The first signature given for comparison operators is valid for only the first two comparison operators. The remaining five signatures for comparison operators are valid for all six comparison operators. Each operator is associated with a precedence level which determines the order of evaluation when more than one operator is involved in an expression. Parenthesis can be used in order to alter the order in which composite object expressions are evaluated.

	OPERATOR	SIGNATURE	PRIORITY
ARITHMETIC OPERATORS	+	int × int → int	3
	-	int × real → real	
	⋆	real × int → real	2
	/	real × real → real	
	mod	int × int → int	2
COMPARISON OPERATORS	=	t × t → bool	4
	<>		
	<	int × int → bool	4
	>	int × real → bool	
	<=	real × int → bool	
	>=	real × real → bool	
		string × string → bool	
STRING OPERATORS	ˆ	string × string → string	3
	first	string → string	1
	rest		
BOOLEAN OPERATORS	not	bool → bool	1
	and	bool × bool → bool	2
	or		3
MATHS OPERATORS	sin cos	int → real	2
	tan atan	real → real	
	abs	int → int	2
		real → real	
	⋆⋆	int × int → real	1
		real × int → real	
	sqrt	int → real	2
		real → real	
	rtoi	real → int	2

Table 1: Operations on primary objects

Built-in operators on secondary objects

For secondary objects the following built-in operators are defined:

1. **Creation of objects**

   By default, the system provides a *blank constructor* method **new** associated with each class. This method takes no arguments and performs no initialisation of the newly created object. So, it merely allows instances of the relevant class to be created. After this constructor has been invoked primary attributes will have default values and secondary attributes will be undefined.

The default blank constructor for *aggregation/sequentiation* instances leaves the collection empty. *aggregation* instance fields are initialised as per attributes with primary fields having the default value for the type and secondary fields left undefined. So, it is normal after invoking the default blank constructor to explicitly set attributes and field members.

For example, the following code creates and initialises an instance of the class `date`:

```
d := new date();
put_dd(10)@d;
put_mm(2)@d;
put_yy(95)@d;
```

It is possible to define alternative constructors and, unlike other user methods, more than one definition of `new` may be provided so long as each constructor has a different number of parameters. The constructor will be presented with a blank object and it is up to it to initialise the object using whatever arguments it is called with. Assuming that a constructor **new** is defined on the class base_part with the arguments given in example 1, the following statement:

```
bp := new base_part ("B067",34.6,true,"ACME",56.9)}
```

makes the variable `bp` reference a new object whose associated class is `base_part` and whose property values are the ones specified as parameters (on the assumption that the user-defined constructor assigns the values given to it as parameters to the attributes of `base_part`).

Note that if any user-defined constructor is provided then the system will not generate a default blank constructor.

2. **Destruction of objects**

   Explicit deletion of objects is supported. The following statement

   ```
   delete v
   ```

   deletes the object referenced by the variable `v`, as well as all references to this object. This avoids dangling references from attributes, aggregations fields or associations/sequentiations and other variables or method parameters currently in scope.

3. **Operators on constructions**

   Three constructors, namely *association*, *sequentiation* and *aggregation* have been defined which allow the construction of new types from previously defined ones. Aggregations are accessed/updated using system generated methods, as described in section 4.2.3.

   A set of polymorphic built-in operators has been defined to allow for the manipulation of objects constructed by association and sequentiation. These operators are summarised in Table 2. For each operator its signature is given, where t stands for a type name and *nat* refers to the subset of *int* containing only positive integers.

Construction operators can be invoked in a message sending style, using the names under *Form 1* in Table 2, or using an infix notation with operator names listed under *Form 1* in Table 2. Any transformation from the message sending style into the non-message sending one is purely syntactic and does not convey any added semantics.

Examples of the use of some of these operations in conjunction with other language constructs are given in section 4.2.7.

CONSTRUCTOR	OPERATOR		SIGNATURE
	Form 1	Form 2	
ASSOCIATION	insert	++	$\{t\} \times t \rightarrow void$
	remove	−−	$\{t\} \times t \rightarrow void$
	is_in	is_in	$t \times \{t\} \rightarrow bool$
	get_member		$\{t\} \rightarrow t$
	empty		$\{t\} \rightarrow bool$
SEQUENTIATION	insert	++	$[t] \times t \rightarrow void$
	insert_at	++	$[t] \times nat \times t \rightarrow void$
	remove	−−	$[t] \times t \rightarrow void$
	remove_at	−−	$[t] \times nat \rightarrow void$
	is_in	is_in	$t \times [t] \rightarrow bool$
	get_member_at	[ ]	$[t] \times nat \rightarrow \{t\}$
	get_member		$[t] \rightarrow t$
	get_indexes		$[t] \times t \rightarrow \{nat\}$
	indexes		$[t] \rightarrow \{nat\}$
	indexes_list		$[t] \rightarrow [nat]$
	lower		$[t] \rightarrow nat$
	upper		$[t] \rightarrow nat$
	empty		$[t] \rightarrow bool$

Table 2: Operations on association and sequentiation constructions

For aggregations, the retrieval/update operators are generated by the system in a similar fashion as for properties (see Section 4.2.3).

4.2.5 User defined operations on objects

The user can define operations on objects by writing methods, implementations for the operation in class definitions. Methods defined for a class are applicable to objects in the extension of that class. Each method is characterised by its *visibility*, which can be either *public* or *private*, and which establishes whether the operation can be invoked from outside the class. All methods declared in the type definition must have a public implementation in the class and no other public methods may be defined.

Example 2 Class declaration

As an example of class declaration, the class `base_part` corresponding to the type with the same name presented in Example 1 is given. Note that `new` is not a method but an object generator function or constructor, and therefore is not used in the same way as a method (for an example of its use, see section 4.2.4).

```
class base_part
    public:
```

```
        new (cd: string, m: real, i: bool, s: string, c: real)
        begin
           put_code(cd)@self;
           put_mass(m)@self;
           put_imported(i)@self;
           put_supplier(s)@self;
           put_cost(c)@self;
        end

        compute_mass(): real
        begin
          get_mass()@self
        end

        compute_cost(): real
        begin
          get_cost()@self
        end

        print()
        begin
           write "NAME: ", get_code()@self, nl;
           write "MASS: ", compute_mass()@self, nl;
           write "IMPORTED: ", get_imported()@self, nl;
           write "SUPPLIER: ", get_supplier()@self, nl;
           write "COST: ", compute_cost()@self, nl;
        end

    end-class
```

■

The code specifying the implementation of an operation is commonly referred to as *method*. The inheritance mechanism supported by ROCK is based on subtyping, i.e. only subclasses which are subtypes are allowed [CHC90]. In the presence of inheritance, it is possible to have different signatures, and hence different implementations, for the same interface name (*overloading*). This results in the same interface name being bound to different implementations (*overriding*). The method selected to respond to a message sent to an object does not depend on the compile-time type of the object expression playing the role of the message recipient, but rather on the type of the object which is the actual recipient of the message at run-time (*late-binding*).

If t_1 is a subtype of t_2, $t_1 \leq t_2$, all objects which are instances of t_1 may be used in place of instances of t_2. According to [Car84], in order to comply with subtyping, an overriding method must obey *contravariance* of argument types and *covariance* of result type with respect to the overridden method. That is, a method $methname(t'_1, ..., t'_n) : t'_r$ is allowed to override $methname(t_1, ..., t_n) : t_r$, represented as $methname(t'_1, ..., t'_n) : t'_r \leq_{meth} methname(t_1, ..., t_n) : t_r$ iff:

- $\forall i : 1 \leq i \leq n, \quad t_i \leq t'_i$
- $t'_r \leq t_r$

These constraints restrict inheritance to satisfy the requirements of subtyping, and are supported for ROCK methods.

4.2.6 Expressions

ROCK & ROLL expressions are evaluated to yield an object. They have an associated return type which is identified during type checking of the expression. An expression is either a

- primary object constant e.g. `1`, `''abc''`, etc
- variable e.g `x`, `long_variable_name`
- self or super expression i.e. the keywords `self` or `super`
- composite object expression formed by combining other expressions using a non-void builtin method e.g `get_member_at(5)@colln`
- operator expression formed by combining other expressions using a builtin operator e.g. `x < 5.0`
- new method invocation
- user method invocation

4.2.7 Statements

In contrast with object expressions, the execution of a statement does not (normally) return an object but provokes a change in the *state of computation.* Statements are classified into *expression* statements, *assignment* statements, *block* statements, *control* statements and *built in* statements.

Expression statements

An expression statement is an expression followed by a semi-colon:

$$expression\ ;$$

Expression statements are normally evaluated by evaluating the associated expression and disregarding the result. However, in some contexts i.e. where an expression statement occurs as a component of a block or control statement, the value of the expression may be used as a value for the enclosing statement.

Assignment statements

An assignment is composed of a *target* or left hand side and a *source* or right hand side.

$$variable := expression$$

The target must be a variable, while the source can be any object expression. The assignment operator ":=" is interpreted as *identity copy* and can be applied to two variables or to an object name (target) and an object expression (source). When applied to two object-names, it results in both object-names referencing the same object. In what follows, whenever we refer to the *value* of a variable, it should be interpreted as the object referenced by that variable.

Note that assignment of sequentiation members can be performed using a similar syntax to the assignment statement:

$variable[int\_expression] := expression$

Although this looks similar to an assignment operation it is actually syntactic sugar for an invocation of the builtin add_member_at operation:

$\textbf{add\_member\_at}(int\_expression, expression)@expression$

Block statements

Block statements enable a sequence of statements to be combined to form one large statement. Blocks may also include declarations for variables which are local to the block. A block statement has the format

begin [*declarations*] *statements* **end**

where statements is a sequence of one or more statements.

A block statement normally has no value. However, a block which forms the body of a functional method (i.e. a ROCK method with a return type) has as its value the value of the last statement in the *statements* sequence.

Control statements

The following categories of control statement provide means to specify the order in which other statements are executed:

- A *conditional statement* implies the execution of one of two specified statements depending on the result of evaluating the specified condition. The structure of a conditional statement is:

  if *objexp* **then** *stat* **else** *stat*

  or

  if *objexp* **then** *stat*

  For example, the following `if` statement informs the user as to whether or not the collection `c` is empty:

  ```
  if (empty@c) then write "empty";
  else write "not empty";
  ```

  If a conditional statement occurs at the end of a block statement which requires a value the result of evaluating the then or else statement will be used as the value for the block. Use of a conditional statement without an else part or with incompatible types for the then and else parts in a context where a value is required is a type error.

  For example, the following method returns an int value from the conditional statement

```
class month
    ...
    last_date() : int
    begin
        if (get_name@self = "February" and is_leap_year()@self) then
            get_number_of_days@self + 1;
        else
            get_number_of_days@self;
    end
    ...
```

- An *iteration statement* involves the conditional execution of a non-empty set of statements repeatedly. The number of times that the set of statements is executed can be indefinite or definite.

  - *Indefinite iteration statement*, which causes the execution of a set of statements zero or more times depending on the condition given by *objexp*. Indefinite iteration is expressed with the *while* construct whose structure is defined as follows:

    while *objexp* **do** *stat*

    The termination test of the *while* construct is performed before each execution of *stat*. As an example, the following loop prints out the values from `j+1` to `k`:

    ```
    while (j < k) do begin
        j := j + 1;
        write "j=",j, nl;
    end
    ```

  - *Definite iteration statement*, which allows iteration over the elements of an association or a sequentiation for retrieval and/or manipulation. The number of times that the loop is executed is the number of elements in the particular association or sequence being traversed. The structure of this form of definite iteration statement is:

    foreach v **in** *objexp* **do** *stat*

    If the result of the evaluation of *objexp* is an association, the loop control variable `v` takes, in each step of the loop, the value of one of the members of the association in turn (in no particular order). For example, the following loop prints out the members of the set containing the values `6`, `7`, `8` and `9`:

    ```
    var intset := {9, 8, 7, 6};
    foreach i in intset do write "i=",i, nl;
    ```

    If the evaluation of *objexp* yields an object whose construction is a sequentiation, `v` takes in each step of the loop the value of the next item in the sequence, starting from the first. For example, the following loop prints out the members of the sequence consisting of the elements `9`, `8`, `7` and `6`, in that order:

```
var intseq := [9,8,7,6];
foreach i in intseq do write "i=",i, nl;
```

A similar syntactic structure is defined to iterate over the instances of a class:

foreach v **in** c **do** *stat*

In this case, the loop control variable v refers, in each step of the loop, to an instance of the class c. For example, the following code fragment invokes the method `print` on each instance of the class `date`:

```
foreach v in date do print()@d;
```

If an iteration statement occurs at the end of a block statement which requires a value then the results obtained from each evaluation of the body of the iteration will be collected and the collection returned as a value for the block, either as an association or a sequentiation depending upon the type required for the enclosing block. Use of an iteration statement to terminate a block requiring either a non-collection value or a collection value whose member type is incompatible with the type of the loop body is a type error.

For example, the following method returns a sequence of integers collected from evaluation of the expression `i` at the end of the while statement:

```
class month
    ...
    make_dates_list() : [int]
    begin
        var last := last_date()@self;
        var i := 0;

        while (i < last)
        begin
            i := i + 1;
            i;
        end
    end
    ...
```

- An *input statement* has the effect of reading a primary object from an input device and assigning it to a previously declared variable. An input statement takes the form:

  read v

  where the variable v must have been previously declared. By default, the value to be read is taken from the standard input. However, if a file has been previously opened on read mode using **openr**(filename), values are read from this file. A file which has been opened for read can be closed using `closer`.

- An *output statement* has the effect of writing on an output device each of the primary objects resulting from the evaluation of the corresponding object expression. The object expressions are evaluated and written in order of appearance. An output statement has the form:

  write *objexp*,...,*objexp*

  By default, values are written to the standard output. However, if a file has been previously opened on write mode using **openw**(filename), values are written to this file. A file which has been opened for write can be closed using `closew`.

  A newline can be written to the output by inserting `nl` as an argument to the `write` statement. The current time can be written to the output by inserting `current_time` as an argument to the `write` statement.

  As an example of input/output and file manipulation, the following program writes to the file `test_output` the number of words in the file `test_data`:

```
program wordcount
begin

    var word:string;
    var numwords := 0;

    write "Read commenced at ", current_time, nl;
    openr(test_data);
    while (not eof()) do begin
        read word;
        numwords := numwords + 1;
    end
    closer;
    write "Read completed at ", current_time, nl;

    openw(test_output);
    write "number of words = ", numwords, nl;
    closew;
end
```

A number of code fragments are now given of the use of ROCK statements for processing collections, which exploit the built-in operations on collections introduced in section 4.2.4. For example, the following code fragment populates the variable `dc`, which is declared to be an association of `date` objects, with every instance of `date` stored in the database:

```
var dc := {}:date;
foreach v in date do dc ++ v;
```

The following code fragment is equivalent, but uses the message-sending form of the operation to add a member to a set:

```
var dc := {}:date;
foreach v in date do insert(v)@dc;
```

The following code fragment removes the members from `dc`:

```
while (not empty@dc) do begin
    var tmp := get_member@dc;
    dc -- tmp;
end
```

As an example of the creation and manipulation of sequences, the following code declares `intseq` as a sequence of integers, initialises it to an empty sequence and installs the values `101`, `102`, etc. at positions `1`, `2` etc. respectively.

```
var intseq := []:int;
int i := 100;

while (i < 105) do begin
    i := i + 1;
    intseq ++ i;
end
```

The operators `lower` and `upper` can be used to obtain the minimum and maximum indices for elements of the sequence:

```
var i := lower@intseq;
while (i <= upper@intseq) do begin
    write i," -> ",intseq[i], nl;
    i := i + 1;
end
```

The operator `get_indexes`, which takes an argument, returns an association containing all indices at which the argument value occurs in the sequence.

The following program removes all occurences of value 102 from the sequence:

```
foreach i in get_indexes(102)@intseq do begin
    remove_at(i)@intseq;
end
```

Note that the removal will leave a gap in the sequence. It will not move entry 103 down to index 2 etc.

The following program iterates over the members of the sequence, increasing the value of each item in the sequence by `100`, and printing out the relationship between the position in the sequence and the value stored:

```
foreach i in indexes@intseq do begin
    intseq[i] := intseq[i] + 100;
    write i," -> ",intseq[i], nl;
end
```

Note the use of the operation `indexes`, which returns an association containing the index of each entry in the sequence, allowing for the fact that index values need not be contiguous. The related operator `indexes_list` retruns a sequentiation containing the indices rather than an association.

5 Query and Method Definition in ROLL

Section 4 presented the facilities of ROCK for defining and manipulating object-oriented databases. In itself, ROCK can be seen as an imperative object-oriented database programming language. This section shows how the logic language ROLL is used within the database programming language framework described in section 4, to complement the facilities provided by ROCK. For further information on logic languages, see [CGT90].

5.1 Querying

The database can be queried from within a program by embedding a query expression written in ROLL into a ROCK program. Queries can be applied to either persistent or transient data. A query expression belongs to the syntactic class of object expressions, and can therefore be used in any context in which an object expression is appropriate. Queries are expressed using a modified version of comprehension notation [Tri90], where the following conventions are adopted:

- Queries are enclosed between square brackets.
- The symbol "|" is used in queries to separate the projection part from the body of the query.
- Logic variables are any combination of characters starting either with a lowercase or a capital letter.
- String constants are given between double quotes.
- The infix operator "==" is used to denote unification. The operands are object expressions which must be evaluated before they are unified. The infix operator "=\=" denotes failure to unify.
- The symbol "~" prefixing a subgoal denotes (stratified) negation as failure.
- ROCK variables used from within a ROLL goal are prefixed by the symbol "!".

A query expression, takes one of three forms:

1. [| *goal_set*]

   This form of query has an empty projection list to the left hand side of the | and results in a boolean value being returned, indicating whether the query has succeeded or failed. For example, the following query against the parts database of figure 1 returns *true* if any of the base parts in the database have a mass of less than `5.0`:

```
[|get_mass@P == M, M < 5.0]
```

ROLL is associated with a type inference system which infers types for the logic variables in embedded queries or methods. In this example, the variable `P` is inferred to be of type `base_part`, as `base_part` is the only type that supports a method called `get_mass`. If several types have methods of the same name, then hints can be given to the type inference system by suffixing the ROLL variable name with a ":" and the name of the type. For example, the following query means the same as that given above:

```
[|get_mass@P:base_part == M, M < 5.0]
```

Such a query can appear in a program as a boolean expression. For example:

```
if [|get_mass@P:base_part == M, M < 5.0] then
    write "There is a base part of mass < 5";
else
    write "There are no base parts with mass < 5";
```

2. [**any** *projection* | *goal_set*]

   This is a non-deterministic form of query where one solution is sought for the goals in goal_set. The projection specifies a single variable or tuple of variables. The result of such an embedded query is either a single variable binding or a single aggregation of variable bindings. The interest is in *one* output object which satisfies the goals in goal_set. For example, the following query selects one of the instances of `base_part` of mass less than `5.0`:

   ```
   [any P |get_mass@P:base_part == M, M < 5.0];
   ```

   The result of such a query can be assigned to a variable of type `base_part`, and then manipulated from ROCK. For example:

   ```
   thePart := [any P |get_mass@P:base_part == M, M < 5.0];
   write "The cost of the part is ",get_cost@thePart,
         " and the mass is ",get_mass@thePart, nl;
   ```

3. [**all** *projection* | *goal_set*]

   All the projected objects which satisfy the goals in goal_set are of interest, and are returned as a collection which is empty if no solutions exist. The projection specifies either a variable or a tuple of variables. The result of such a query is an association of variable bindings, or an association of aggregations of variable bindings. For example, the following query expression evaluates to an association of all the instances of `base_part` with a mass less than `5.0`:

```
[all P |get_mass@P == M, M < 5.0]
```

As with the previous example, the association of part objects which is the result of this query can then be manipulated from ROCK in the standard way:

```
var partSet:{part};
partSet := [all P |get_mass@P == M, M < 5.0];
foreach p in partSet do put_cost(get_cost@p*1.1)@p;
```

This example shows the projection of a collection of single values. However, it may be required for a collection of aggregations to be retrieved as the result of an embedded query. For example, there might be an interest in the name and the mass of the light base parts. If so, then the following query can be used, which retrieves name-mass pairs:

```
[all <N,M> | get_mass@P == M, M < 5.0, get_name@P == N]
```

The data in the resulting set of aggregate objects can be accessed from ROCK thus:

```
var nmSet := [ all <N,M> | get_mass@P == M, M < 5.0, get_name@P == N];
foreach v in nmSet do
    write get_name@v, " ",get_mass@v, nl;
```

In this example, the variable `nmSet` is inferred to have a type that is an association of objects which are aggregates of types `name` (an alias for `string`) and `mass` (an alias for `real`).

If the query is used in a context where a sequentiation is expected then the result will be constructed as a sequentiation rather than an association. The following query retrieves a sequence pairs of composite parts and base parts such that the composite parts have light base parts as direct components. The aggregation type `PartSubPart` is used as the type of each retrieved pair and a sequentiation of `PartSubPart` instances is constructed as the result of the query expression:

```
type PartSubPart:
    public <cp:composite_part,bp:base_part>;
end-type
...
var partAgg: [PartSubPart];

partAgg := [ all <CP,BP> | get_mass@BP:base_part == M, M < 5.0,
                           BP is_in CP:composite_part];

write "There are ", upper@partAgg, " solutions", nl;
foreach pp in partAgg do
    write "CP Name = ", get_name@get_cp@pp,
          " BP Name = ", get_name@get_bp@pp, nl;
```

In this example, the operator `is_in` tests for membership of `BP` in the construction of `CP`. Since the query expression is used in an assignment to a variable whose type is `[PartSubPart]` the type of the result is inferred to be a sequentiation and the member type of the sequentiation is inferred to be `PartSubPart`. Note that the order and type of the variables in the projection corresponds to the order and type of the fields in the aggregation type `PartSubPart`.

As an alternative, the type inference system can be used to infer the complete type of the query expression, as in the following version of the same task:

```
var tryout := [all <CP,BP> | get_mass@BP:base_part == M, M < 5.0,
                           BP is_in CP:composite_part];
foreach p in tryout do
    write "CP Name = ", get_name@get_composite_part@p,
         " BP Name = ", get_name@get_base_part@p, nl;
```

In the above example the variable tryout is inferred to be an association whose members are aggregations. The aggregation has a `composite_part` as its first field and a `base_part` as its second field. The system generated methods `get_composite_part` and `get_base_part` are used in the for loop to access these fields.

The "," separator between goals in a goal set denotes conjunction so that any solution must satisfy all goals in the goal set. A goal set may also use the disjunction operator, ";" to specify alternative sequences of goals for the query. For example, the following query retrieves all parts with mass outside the interval [1.0, 5.0]:

```
var nmSet := [ all <N, M> | get_name@P == N, get_mass@P == M, M > 5.0;
                            get_name@P == N, get_mass@P == M, M < 1.0 ];
foreach v in  do
    write get_name@v, " ",get_mass@v, nl;
```

A query must satisfy the condition that all variables which occur in the projection also occur in each disjunct in the goal set that constitutes the body of the query. Otherwise, it is not clear which values the query has to return and it may not be possible to infer a type for the query.

5.2 System-generated methods and built-in operators in ROLL

System-generated methods and built-in operations which do not modify objects or have side-effects may all be invoked from ROLL queries or methods. The operation of these methods or operators is generalised to support deductive evaluation. For example, in the following query

```
[|get_mass@P == M, M < 5.0]
```

the system-generated method `get_mass` is used to retrieve the mass of each instance of type `base_part`.

Although the above method invocation uses a functional syntax it operates declaratively when invoked from ROLL. So, in this case it *generates* pairs of bindings for the variables `P` and `M` by scanning the class `base_part` and retrieving the `mass` associated with each instance.

If the method is invoked with a different binding pattern it operates accordingly. For example in the query

```
[ {P} | get_name@P:part == "flange grommet", get_mass@P == 0.5]
```

the call to `get_name` yields bindings for `P` which are instances of class `part` with name `"flange grommet"`. The call to `get_mass` *tests* these bindings, rejecting bindings for `P` to instances of part whose mass attribute does not have value `0.5`. In practice, there is no guarantee that this order of evaluation is chosen. It may be that the call to `get_mass` is evaluated first to generate bindings for `P` and that the `get_name` call is used to test for objects with the desired name.

Table 3 lists the built-in operations available for use in ROLL methods and queries.

CATEGORY	NAME		SIGNATURE
	Infix Form	Message Form	
ASSOCIATION METHODS	`is_in` `empty`	`is_member` `empty`	$t \times \{t\}$ $\{t\}$
SEQUENTIATION METHODS & OPERATORS	`is_in` `empty` [ ]	`is_member` `empty` `get_element`	$t \times [t]$ $[t]$ $[t] \times nat \rightarrow t$
AGGREGATION METHODS		get_*xxx*	$t \rightarrow t'$
ATTRIBUTE METHODS		get_*xxx*	$t \rightarrow t'$
UNIFICATION OPERATORS	$==$ $=\backslash=$		$t \times t$
INEQUALITY OPERATORS	$<$ $<=$ $>$ $>=$		$int \times int$ $real \times int$ $int \times real$ $real \times real$
ARITHMETIC OPERATORS	$+$ $-$ $*$ $/$		$int \times int \rightarrow int$ $real \times int \rightarrow real$ $int \times real \rightarrow real$ $real \times real \rightarrow real$
NEGATION OPERATOR	$\sim$		

Table 3: ROLL system generated and built-in methods and operators

Most methods have a predicate syntax (indicated in the table by omission of the $\rightarrow$ symbol in the signature). In some cases e.g. `get_name@P` an expression syntax is employed in order to conform with the ROCK syntax. Only invocations of methods with predicate syntax may appear as atoms in a clause or query body. Expressions may occur either as an operand to a unification, e.g. `get_name@P == "flange grommet"` or as an argument to some other predicate e.g `get_mass@P1 > get_mass@P2 + 5`. The latter is equivalent to `get_mass@P1 == M1, get_mass@P2 == M2, M3 == M2 + 5, M1 > M3`. The standard restrictions are imposed on variable bindings used in arithmetic expressions.

The negation operator $\sim$ is a unary prefix operator which applies to ROLL method invocations. It may be prefixed to an invocation of an association or sequentiation method with predicate syntax or to a user-defined method invocation e.g

```
[{P2} | get_name@P1 == "flange_grommet", ~ P1 is\_in P2]
```

This query would retrieve all composite parts which do not contain some component part with name `"flange_grommet"`.

5.3 Method definition in ROLL

As with ROCK methods, ROLL methods are attached to classes. A ROLL method has a signature, which specifies the types of the parameters to the method, and a body. Note that, unlike ROCK, there is no distinction drawn in the definition of a method between parameters which are to be used for input and those which are to be used for output. The body of a ROLL method is defined by a set of one or more rules expressed in clausal form. The body of each clause is a goal with one or more subgoals. The order in which these subgoals are written has no semantic import. The types of logic variables, other than parameters occurring in a subgoal, may be specified, but in general the system is able to infer their types. A subgoal is either an object expression, a unification expression or a ROLL method invocation. A ROLL clause must satisfy the condition that all variables which occur in the head of the clause also occur in the body of the clause.

As with ROCK methods, the signature of a ROLL method is declared with the type, while the code which mades up the body of the method is defined in the class. The interface of a ROLL method has the following form, where t_i represents the type of an argument:

type type_name
    ...
    ROLL:
        method_name(t_1,...t_n),
        ...
        method_name(t_1,...t_n);
end-type

As an example, the definition of the type `composite_part` can be extended with a rule `is_component` with a single argument thus:

```
type composite_part:
    specialises: part;
    properties:
        ...
    public {part};
    ROCK:
        ...
    ROLL:
```

```
        is_component(composite_part);
end-type
```

The definition of a clause within a ROLL method can take one of two forms:

- *method_name(*P_1*,...,*P_n*) :- goallist*

  When a rule whose head is defined in this form is executed, the recipient of the message which caused the rule to be executed is assigned to `self`, and can be used later within the body of the rule. When a method is invoked with a non-bound variable as message recipient, this variable gets bound in each step of execution to a different instance of the class which defines the method responding to the invocation.

  As an example of this form, the definition of `is_component`, which searches transitively for the subparts of a `composite_part`, is as follows:

  ```
  class composite_part
      ...
      is_component(composite_part)
          begin
              is_component(P1) :-
                  P1 is_in self;
              is_component(P1) :-
                   P2 is_in self, is_component(P1)@P2;
          end
  end
  ```

  Where the method has no arguments, it can be written as *method_name :- goallist.*

- *method_name(*P_1*,...,*P_n*)@*P_0 *:- goallist*

  The head of the rule takes the form of a method invocation, and pattern matching is applied between the invocation and the head of the rule. The recipient of the message is assigned to P_0, that is the message recipient is effectively *renamed* as P_0 by the the rule definition. Therefore, `self` cannot be used within the body of the rule and P_0 may be used instead.

  As an example of this form, the definition of `is_component`, which searches transitively for the subparts of a `composite_part` is defined as follows:

  ```
  class composite_part
      ...
      is_component(composite_part)
      begin
          is_component(P1)@This :-
              P1 is_in This;
          is_component(P1)@This :-
  ```

```
                P2 is_in This, is_component(P1)@P2;
        end
    end
```

Note that since the rule may be invoked with the message recipient as an unbound variable, during execution it gets bound to each instance of composite_part in turn.

ROLL methods can only be invoked from within query expressions, and input (ROCK variables) and output parameters (unbound logic variables) are specified with the invocation of the method rather than when the method is defined. Overloading and overriding of ROLL methods is allowed, and the process of binding is handled in a similar fashion as for methods defined in ROCK. Note, however, that when overriding takes place in ROLL, the contravariance rule is not applied, as it is not known at compile time which arguments are for input, and which for output. As a result of this, all argument types in an overriding ROLL method must be covariant with those for the method being overridden.

The following query checks, given two parts in the form of the ROCK variables `x` and `y`, whether one is a component of the other:

```
boolVar := [ | is_component(!x)@!y ];
```

A method can, however, be invoked using different binding patterns. For example, the following query retrieves all the components of the composite part referenced by the ROCK variable `x`:

```
compSet := [{P} | is_component(P)@!x];
```

Note that the message recipient in a query need not be bound. For example, the following query retrieves all the objects of which the part referenced by the ROCK variable `x` is a component:

```
compSet := [{P} | is_component(!x)@P];
```

When a method is invoked with a non-bound variable as message recipient, this variable gets bound in each step of execution to a different instance of the class which defines the method responding to the invocation. Overriding is still respected in such cases, so the solutions for subclass instances are derived using the method associated with the subclass.

6 Spatial Extension

The ROCK & ROLL system provides built-in support for vector spatial data types, specifically the ROSE algebra [GS95]. While it would be possible to implement spatial data types on top of the general purpose information management facilities of ROCK & ROLL, incorporating them into the kernel of the system provides a tighter integration of spatial and aspatial data handling facilities, and better performance. This section gives an overview of the ROSE algebra, and describes how the spatial extension to ROCK & ROLL works.

6.1 The ROSE Algebra

The ROSE algebra provides three spatial data types – `points`, `lines` and `regions`. A `points` value is a set of x-y coordinate pairs. A `lines` value is a set of connected line segments. A *regions* value is a set of polygons, each of which may contain holes.

The underlying geometric domain of the ROSE algebra is a finite discrete grid (built from integer values) called a *realm*. All spatial values in a ROSE algebra application are mapped onto a single underlying realm. The fact that the realm is a finite resolution structure means that ROSE algebra types have a clean semantics and are amenable to efficient implementation.

Figure 2 illustrates a realm with a number of spatial values mapped onto it. The realm is the underlying grid structure (which is not directly accessible to programmers), and the line segments and filled grid points represent components of ROSE algebra values.

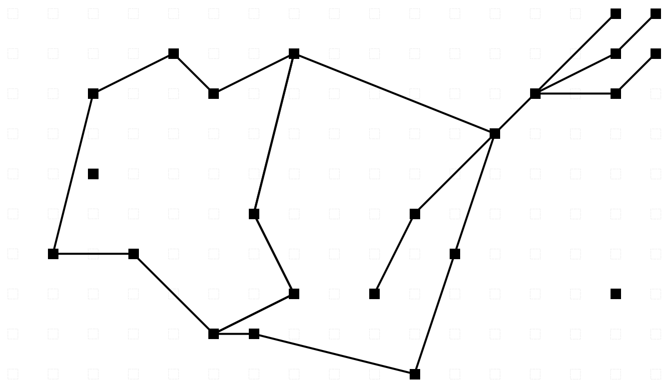


Figure 2: Example realm.

Figure 3 identifies the ROSE algebra values from the realm in figure 2. The spatial values are one `regions` value whose single element is A, another whose single element is B, the `lines` value that is the singleton containing C and the `points` value whose elements are D and E. Note that the same realm components could be associated with different ROSE algebra values. For example, both A and B could have been components of a single `regions` value.

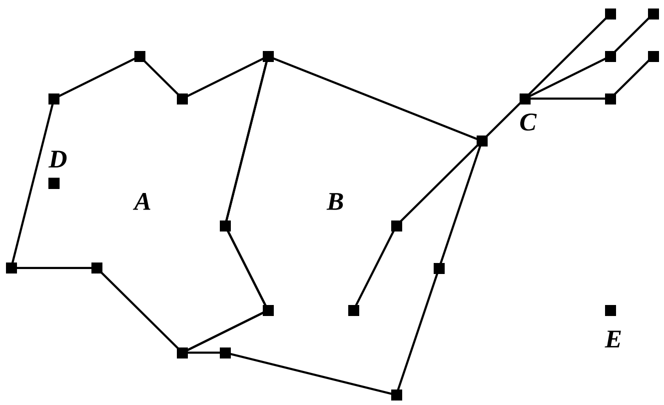


Figure 3: Example ROSE algebra values.

6.2 Spatial Types in ROCK & ROLL

The types `points`, `lines` and `regions` are supported as primitive types, henceforth referred to as Spatial Data Types (SDTs). This decision has a number of consequences:

- SDTs can neither be refined nor redefined within the system's data defintion language.
- Instances of SDTs persist only if they occur as values of the properties or construction in an instance of a user-defined persistent class.
- Instances of SDTs are neither created nor destroyed using ROCK & ROLL constructs explicitly.
- It is not possible to iterate over the extension of an SDT in ROCK & ROLL.

While these consequences may seem quite severe, they are familiar from other literals, and we believe the treatment of spatial values as literals generally turns out to be natural in practice.

As with other types, the SDTs can be used as the type of a variable in ROCK or ROLL, or to provide the type for an property or construction. For example, the following program fragment introduces two new types with spatial attributes:

```
type GEO.State:
    properties:
        public:
            Name: string,
            Number: int;
            RegionsRep : regions;
    ...
end_type

type GEO.Road:
    properties:
        public:
            Name, Type, Width, Lanes: int;
            LinesRep: lines;
    ...
end_type
```

Variables can be declared in ROCK as would be expected:

```
var aPointsValue: points;
var aLinesValue: lines;
var aRegionsValue: regions;
```

6.3 Spatial Literals

In both programs and external files, spatial values are representable by spatial literals. A `points` literal is an expression of the form

$$\texttt{points}\{p_1, \ldots, p_n\}$$

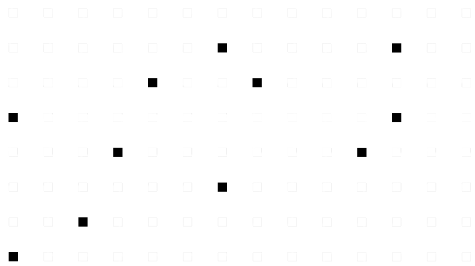
where each $p_i, 0 \leq i \leq n$, is represented as a pair of integer literals separated by a colon. Examples of `literals` are given in Figure 4(a).

```
points{}

points{0:0}

points{
   0:0,  0:4,  2:1,  3:3,
   4:5,  6:2,  6:6,  7:5,
  10:3, 11:4, 11:6
}
```

(a)



(b)

Figure 4: `points` Spatial Literals

The first literal in Figure 4(a) denotes the empty `points` value. The second denotes a singleton `points` value. The third is depicted in Figure 4(b).

A `lines` literal is an expression of the form

$$\texttt{lines}\{b_1, \ldots, b_m\}$$

where each $b_i, 0 \leq i \leq m$ is a connected graph, expressed in the form

$$[s_1, \ldots, s_n]$$

where each $s_i, 1 \leq i \leq n$, is a line segment represented as a pair of points (as previously described) separated by an underscore. Examples of `lines` literals are given in Figure 5(a).

The first literal in Figure 5(a) denotes the empty `lines` value. The second denotes a singleton `lines` value. The third is depicted in Figure 5(b).

A `regions` literal is an expression of the form

$$\texttt{regions}\{f_1, \ldots, f_m\}$$

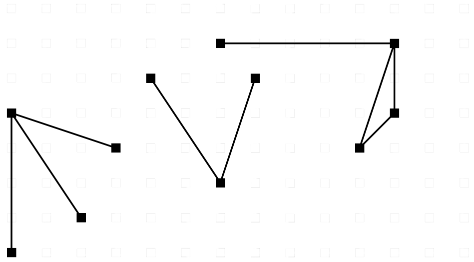
where each $f_i, 0 \leq i \leq m$, is represented as a pair of components separated by the keyword `inner`. The left-hand component represents the boundary of an area and the right-hand component is a set representing inner areas, if any, within the boundary. Each left-hand component is an expression of the form

$$[s_1, \ldots, s_n]$$

```
lines{}

lines{[0:0_0:4]}

lines{
  [ 0:0_ 0:4,  2:1_ 0:4,  3:3_ 0:4 ],
  [ 6:2_ 4:5,  6:2_ 7:5            ],
  [ 6:6_11:6, 11:6_11:4, 11:4_10:3,
    10:3_11:6]
}
```



(a) (b)

Figure 5: `lines` Spatial Literals

where each $s_i, 1 \leq i \leq n$, is a line segment (as previously described). The right-hand component is an expression of the form

$$\{c_1, \ldots, c_p\}$$

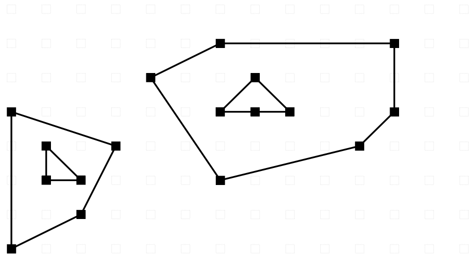
where each $c_i, 0 \leq i \leq p$, has the same form as the left-hand component. If there are no inner areas then only the left-hand component needs to be written and the keyword `inner` is omitted. Examples of `regions` literals are given to the left of Figure 6.

```
regions{}

regions{[ 0:0_2:1, 2:1_3:3, 3:3_0:4, 0:4_0:0 ]}

regions{
  [  0:0_2:1, 2:1_3:3, 3:3_0:4, 0:4_0:0 ]
  inner {[ 1:2_2:2, 2:2_1:3, 1:3_1:2 ]}
}

regions{
  [ 0:0_2:1, 2:1_3:3, 3:3_0:4, 0:4_0:0 ]
  inner {[ 1:2_2:2, 2:2_1:3, 1:3_1:2 ]},
  [ 6:2_ 4:5,  4:5_6:6, 6:6_11:6, 11:6_11:4,
   11:4_10:3, 10:3_6:2 ]
  inner {[ 6:4_7:4, 7:4_8:4, 8:4_7:5, 7:5_6:4 ]}
}
```



(a) (b)

Figure 6: `regions` Spatial Literals

The first literal in Figure 6(a) denotes the empty `regions` value. The second denotes a singleton `regions` value, the only component in which has no inner regions (and hence only the left-hand part is written). The third denotes another singleton `regions` value, but this time there is a hole inside it (and hence the keyword `inner` and the right-hand component are also written). The last `regions` literal has two regions, each with a hole, and is depicted in Figure 6(b).

6.4 Spatial Operations

There are a substantial number of operations supported on `points` (Table 4), `lines` (Table 5) and `regions` (Table 6). Definitions of the ROSE algebra operations are provided in [GS95]. Spatial operations are written as functions or operators.

Operator	**Result Type**
`count`($points_i$)	integer
`diameter`($points_i$)	real
$points_i$ `disjoint` $points_j$	boolean
`dist`($points_i, points_j$)	real
`dist`($points_i, regions_j$)	real
$points_i$ `equal` $points_j$	boolean
$points_i$ `inside` $regions_j$	boolean
$points_i$ `intersection` $points_j$	points
$points_i$ `minus` $points_j$	points
$points_i$ `not_equal` $points_j$	boolean
$points_i$ `on_border_of` $lines_j$	boolean
$points_i$ `on_border_of` $regions_j$	boolean
$points_i$ `plus` $points_j$	points
$points_i$ `vertex_disjoint` $points_j$	boolean

Table 4: Operations on `points`.

6.5 Declarative Methods and Queries

As with non-spatial operations, there are centain restrictions on the invocation of spatial operations from ROLL for safety reasons – in essence, all ROLL programs must be stratified and range restricted. An important benefit of the treatment of STDs as values rather than objects is that the creation of a new spatial value is not an update to the database, so new spatial values can be created in ROLL.

All spatial operations such as `plus(lines)@lines: lines` that construct spatial values have a functional syntax. The value returned by the operation can be regarded as an output, while the recipient and arguments are considered inputs to the operation. For evaluation to be safe, every input variable must have a binding resulting from the evaluation of another positive atom in the body of the method or query.

As an example of a program exploiting the spatial data types, the following program, assuming the data types `State` and `Road`, identifies all streams inside Scotland:

```
var sr: {Road};
```

Operator	Result Type
$lines_i$ `border_in_common` $lines_j$	boolean
$lines_i$ `common_border` $lines_j$	lines
`count`($lines_i$)	integer
`diameter`($lines_i$)	real
$lines_i$ `disjoint` $lines_j$	boolean
`dist`($lines_i$, $points_j$)	real
`dist`($lines_i$, $lines_j$)	real
`dist`($lines_i$, $regions_j$)	real
$lines_i$ `equal` $lines_j$	boolean
$lines_i$ `inside` $regions_j$	boolean
`interior`($lines_i$)	regions
$lines_i$ `intersection` $lines_j$	points
$lines_i$ `intersects` $lines_j$	boolean
$lines_i$ `intersects` $regions_j$	boolean
`length`($lines_i$)	real
$lines_i$ `meets` $lines_j$	boolean
$lines_i$ `meets` $regions_j$	boolean
$lines_i$ `minus` $lines_j$	lines
$lines_i$ `not_equal` $lines_j$	boolean
$lines_i$ `plus` $lines_j$	lines
`vertices`($lines_i$)	points

Table 5: Operations on `lines`.

```
sr:= [ all Rd | get_Name@S:State == "Scotland",
                get_RegionsRep@S == SR,
                get_LinesRep@Rd == RdR,
                RdR inside SR ];

foreach l in sr do begin
    write "Road ", get_Name@l, " lies inside Scotland", nl;
end
```

7 Model of Persistence

The unit of persistence in ROCK & ROLL is the class, and instances of persistent classes persist. Thus, the model provides for the persistence not only of data in the form of class instances, but also of code in the form of methods within class definitions.

An environment is a collection of classes. Environments are used to make classes persist between

Operator	Result Type
$regions_i$ `adjacent` $regions_j$	boolean
`area`($regions_i$)	real
$regions_i$ `area_disjoint` $regions_j$	boolean
$regions_i$ `border_in_common` $lines_j$	boolean
$regions_i$ `common_border` $lines_j$	lines
`contour`($regions_i$)	lines
`count`($regions_i$)	integer
`diameter`($regions_i$)	real
$regions_i$ `disjoint` $regions_j$	boolean
`distance`($regions_i, regions_j$)	real
$regions_i$ `edge_disjoint` $regions_j$	boolean
$regions_i$ `edge_inside` $regions_j$	boolean
$regions_i$ `encloses` $regions_j$	boolean
$regions_i$ `equal` $regions_j$	boolean
$regions_i$ `inside` $regions_j$	boolean
$regions_i$ `intersection` $lines_j$	lines
$regions_i$ `intersection` $regions_j$	regions
$regions_i$ `intersects` $regions_j$	boolean
$regions_i$ `meets` $regions_j$	boolean
$regions_i$ `minus` $regions_j$	regions
$regions_i$ `not_equal` $regions_j$	boolean
`perimeter`($regions_i$)	real
$regions_i$ `plus` $regions_j$	regions
$regions_i$ `vertex_inside` $regions_j$	boolean
`vertices`($regions_i$)	points

Table 6: Operations on `regions`.

program runs. By default, classes are located in the runtime environment which is explicitly purged of all classes and instances after each run of a ROCK & ROLL program. If a program wishes to preserve a class for use by other programs then it must explicitly locate the class in an environment. Other programs which use that environment can mention the class by name and invoke its methods to operate on its instances. When using the persistent version of the system `prnr`, environment classes are stored on disk and so persist across invocations of the ROCK & ROLL system. When using the non-persistent version of the system `rnr`, classes stored in environments persist across program runs during a single invocation of ROCK & ROLL but are lost when the ROCK & ROLL session is quit or aborted.

References between persistent environments or from a persistent environment to the runtime environment are disallowed. Classes stored in an environment can only refer to built in classes or other classes in the same environment. This implies that instances located in the environment can only store references to instances in the same environment.

References from the runtime environment to persistent environments are allowed. Runtime classes can freely reference classes located in the environments appearing in their defining program's environment list. Similarly, instances of runtime classes can reference instances of environment classes.

7.1 Persistent class declaration

A type and its related class can be associated with an environment when they are declared (and only then) by prefixing the name with an environment name:

type e.c : *classexp* **end-class**

class e.c *classexp* **end-class**

`e` must be one of the environments specified as parameters to the program in which `c` is being declared.

A type name can appear in a program in a number of different contexts (e.g. variable declarations, foreach constructs, etc.) other than as the name in a type or class declaration. A persistent type name may be referenced by indicating the formal environment in which it lives:

$$e.\mathrm{c}$$

When a plain type name is found within a program, four different situations are possible:

1. The class was declared as transient within the same program.
2. The class was declared as persistent within the same program. Therefore, it is regarded as being in scope.
3. The class is persistent in at most one of the persistent environments associated with the program. Therefore, it is regarded as being in scope.
4. A class with the same name persists in more than one of the persistent environments associated with the program. If this is the case, the name of the persistent environment must be used to disambiguate between environments.

As an example of the use of persistence, the following programs create and manipulate the persistent class `date`. The first program (assumed to be stored in the file `edate1.rnr`) defines the class `date` and creates some instances:

```
program date(E)
begin

type E.date:
    public <dd:int, mm:int, yy:int>;
    ROCK:
        print();
end-type

class E.date
```

```
    public:
        print()
        begin
            write get_dd()@self, ":", get_mm()@self, ":", get_yy()@self, nl;
        end
end-class


    begin
        var i := 0;

        while (i < 10) do begin
            var d := new date();
            i := i + 1
            put_dd(i)@d;
            put_mm(2)@d;
            put_yy(95)@d;
        end

        foreach v in date do print()@v;
    end
end
```

The program has the environment name `E` as a formal parameter. When the program is run this parameter must be associated with an actual environment:

```
rock&roll> define_env E1
rock&roll> run edate1.rnr E1
Type checking...
End of type checking
Compiling...
End of compilation
Execute? (Y/N): y
1:2:95
2:2:95
3:2:95
4:2:95
5:2:95
6:2:95
7:2:95
8:2:95
9:2:95
10:2:95
```

```
rock&roll>
```

When the program is run as depicted above, the class `date` and its instances are stored in the environment `E1`. When using the persistent version of the system, this environment is stored in the database between sessions. Once the data has been stored in the environment, it can be accessed there by other programs. For example, the following program (which is assumed to be stored in the file `edate2.rnr`) prints information on the dates stored in the environment:

```
program date(E)
begin
    foreach v in date do print()@v;
end
```

This program can be run against the environment `E1` thus:

```
rock&roll> run edate2.rnr E1
Type checking...
End of type checking
Compiling...
End of compilation
Execute? (Y/N): y
1:2:95
2:2:95
3:2:95
4:2:95
5:2:95
6:2:95
7:2:95
8:2:95
9:2:95
10:2:95

rock&roll>
```

Further programs can change the date stored in the environment. For example, the following program creates some additional instances of `date`:

```
program date(E)
begin
    var i := 0;

    while (i < 10) do begin
```

```
        var d := new date();
i := i + 1;
        put_dd(i)@d;
        put_mm(2)@d;
        put_yy(96)@d;
    end

    foreach v in date do print()@v;
end
```

The instances created by this program can then be accessed by other programs, such as `edate2.rnr` described above.

The current version of ROCK & ROLL does not support schema evolution.

8 Grammar and Tokenising Rules

8.1 Grammar in BNF Notation

The syntax of the language is described using a BNF grammar. The following notational conventions are used:

- The non-bold symbol : separates a defined term and its definition.
- The non-bold symbol | separates alternative definitions
- The non-bold symbol ; terminates a definition.
- Bold, regular strings represent a terminal symbol of the language.
- Non-bold lower case italic strings represent defined non-terminal symbols.
- Non-bold upper case italic strings represent undefined non-terminal symbols (numeric/string constants and identifiers).
- The symbol ϵ represents an empty token sequence.

The distinguished symbol is *program*. In order to resolve ambiguity in parsing, alternative forms of each right-hand side of a rule are in order of decreasing precedence. This grammar does not include the spatial extensions.

program	:	**program**
		programid opt_envid_list program_block
	;	
opt_envid_list	:	ϵ
	|	**(** *envid_list* **)**
	;	
envid_list	:	*envid*
	|	*envid_list* **,** *envid*
	;	
program_block	:	**begin** *opt_definition_statements*
		opt_declaration_statements opt_statements **end**
	|	**begin** *opt_definition_statements error* **end**
	;	
block_statement	:	**begin** *opt_declaration_statements opt_statements* **end**
	|	**begin** *error* **end**
	;	
opt_definition_statements	:	ϵ
	|	*definition_statements*
	;	
opt_declaration_statements	:	ϵ
	|	*declaration_statements*
	;	
opt_statements	:	ϵ
	|	*expr*
	|	*statements*
	|	*statements expr*
	;	
definition_statements	:	*definition_statement*
	|	*definition_statements definition_statement*
	;	
definition_statement	:	*type_defn*
	|	*class_defn*
	;	

declaration_statements	:	*declaration_statement*
	|	*declaration_statements declaration_statement*
	;	
declaration_statement	:	*var_decl* ;
	;	
programid	:	*IDENTIFIER*
	;	
envid	:	*IDENTIFIER*
	;	
otypeid	:	*IDENTIFIER*
	;	
classid	:	*otypeid*
	|	*envid* . *otypeid*
	;	
varid	:	*IDENTIFIER*
	;	
rollvarid	:	*IDENTIFIER*
	;	
fileid	:	*IDENTIFIER*
	|	*STRINGCON*
	;	
msgid	:	*IDENTIFIER*
	;	
parmid	:	*IDENTIFIER*
	;	
classid_list	:	*classid*
	|	*classid_list* , *classid*
	;	
varexpr_list	:	*varid*
	|	*varexpr_list* , *varid*
	;	
rollvarid_list	:	*rollvarid*
	|	*rollvarid_list* , *rollvarid*
	;	
typeid_list	:	*typeid*
	|	*typeid_list* , *typeid*
	;	

var_decl	:	**var** *varid* **:** *typeid*
	|	**var** *varid* **:** *constr_expr*
	|	**var** *varid* **:=** *initialiser*
	;	
type_defn	:	**type** *classid* **:**
		type_defn_sub **end_type**
	|	**type** *classid error* **end_type**
	;	
type_defn_sub	:	*opt_inh_expr opt_att_expr*
		opt_constr_expr opt_interface_expr
	|	*typeid*
	;	
opt_inh_expr	:	ϵ
	|	*inh_expr* **;**
	|	*inh_expr error*
	;	
opt_att_expr	:	ϵ
	|	**properties :** *att_expr* **;**
	;	
opt_constr_expr	:	ϵ
	|	**public** *constr_expr* **;**
	|	**private** *constr_expr* **;**
	;	
opt_interface_expr	:	ϵ
	|	*interface_spec_list*
	;	
inh_expr	:	**specialises :** *classid_list*
	;	
att_expr	:	**private :** *props_list*
	|	**public :** *props_list*
	|	*att_expr* **private :** *props_list*
	|	*att_expr* **public :** *props_list*
	;	
interface_spec_list	:	*imp_interface_spec*
	|	*decl_interface_spec*
	|	*interface_spec_list imp_interface_spec*
	|	*interface_spec_list decl_interface_spec*
	;	

```
imp_interface_spec      :   ROCK : imp_interface_list ;
                        |   ROCK : imp_interface_list
                        ;
decl_interface_spec     :   ROLL : decl_interface_list ;
                        |   ROLL : decl_interface_list
                        ;
imp_interface_list      :   imp_interface
                        |   imp_interface_list , imp_interface
                        |   imp_interface_list ; imp_interface
                        ;
decl_interface_list     :   decl_interface
                        |   decl_interface_list , decl_interface
                        |   decl_interface_list ; decl_interface
                        ;
imp_interface           :   func_interface
                        |   proc_interface
                        |   new_interface
                        ;
func_interface          :   msgid ( ) : typeid
                        |   msgid ( signature_args ) : typeid
                        |   msgid ( ) : constr_expr
                        |   msgid ( signature_args ) : constr_expr
                        ;
proc_interface          :   msgid ( )
                        |   msgid ( signature_args )
                        ;
new_interface           :   new ( )
                        |   new ( signature_args )
                        ;
decl_interface          :   msgid ( typeid_list )
                        |   msgid
                        ;
constr_expr             :   {  typeid }
                        |   [ typeid ]
                        |   < props_list >
                        ;
```

primitive_type	:	**int**
	|	**bool**
	|	**string**
	|	**real**
typeid	:	*primitive_type*
	|	*classid*
	;	
signature_args	:	*arg_spec*
	|	*signature_args* , *arg_spec*
	;	
arg_spec	:	*parmid* **:** *typeid*
	;	
props_list	:	*prop*
	|	*props_list* , *prop*
	;	
prop	:	*typeid*
	|	*typeid* **:** *typeid*
	;	
class_defn	:	**class** *classid*
		opt_meth_group **endclass**
	|	**class** *classid error* **endclass**
	;	
opt_meth_group	:	ϵ
	|	*meth_group*
	;	
meth_group	:	**public :** *pub_meth_def_list*
	|	**private :** *priv_meth_def_list*
	|	*meth_group* **public :** *pub_meth_def_list*
	|	*meth_group* **private :** *priv_meth_def_list*
	;	

```
pub_meth_def_list      :   pub_meth_def
                       |   pub_meth_def_list pub_meth_def
                       ;
priv_meth_def_list     :   priv_meth_def
                       |   priv_meth_def_list priv_meth_def
                       ;
pub_meth_def           :   func_meth_def
                       |   proc_meth_def
                       |   new_meth_def
                       |   decl_meth_def
                       ;
priv_meth_def          :   func_meth_def
                       |   proc_meth_def
                       |   decl_meth_def
                       ;
func_meth_def          :   func_interface block_statement
                       ;
proc_meth_def          :   proc_interface block_statement
                       ;
new_meth_def           :   new_interface block_statement
                       ;
decl_meth_def          :   decl_interface clause_block
                       ;
clause_block           :   begin clause_list end
                       ;
clause_list            :   clause ;
                       |   clause_list clause ;
                       ;
clause                 :   clause_head :- goal_list
clause_head            :   msgid
                       |   msgid @ rollvar
                       |   msgid ( roll_term_list )
                       |   msgid ( roll_term_list ) @ rollvar
                       ;
```

statements	:	*statement*
	|	*statements statement*
	;	
statement	:	*assign_statement* ;
	|	*control_statement*
	|	*block_statement*
	|	*builtin_statement* ;
	|	*expr_statement* ;
	;	
assign_statement	:	*varid* **:=** *expr*
	;	
control_statement	:	**foreach** *varid* **in** *expr* **do** *statement*
	|	**while** *expr* **do** *statement*
	|	**if** *expr*
		then *statement*
	|	**if** *expr*
		then *statement* **else** *statement*
	;	
builtin_statement	:	*non_oper_expr* **insert_at** *expr*
	|	**insert** (*expr*) **@** *non_oper_expr*
	|	**insert** (*expr*) **@** *non_oper_expr*
	|	*non_oper_expr* **remove_at** *expr*
	|	**remove** (*expr*) **@** *non_oper_expr*
	|	**remove_at** (*expr*) **@** *non_oper_expr*
	|	*varid* [*expr*] := *expr*
	|	**insert_at** (*expr* , *expr*) **@** *non_oper_expr*
	|	**insert** (*expr* , *expr*) **@** *non_oper_expr*
	|	**remove_at** (*expr*) **@** *non_oper_expr*
	|	**remove_at** (*expr*) **@** *non_oper_expr*
	|	**read** *varexpr_list*
	|	**write** *write_expr_list*
	|	**delete** *varid*
	|	**openr** (*fileid*)
	|	**openw** (*fileid*)
	|	**closer**
	|	**closew**
	;	

expr_statement	:	*expr*
	;	
expr_list	:	*expr*
	|	*expr_list* , *expr*
	;	
non_oper_expr	:	*varid*
	|	**self**
	|	*constant*
	|	*builtin_expr*
	|	*user_method_inv*
	|	*new_method_inv*
	|	*query_expr*
	|	**(** *expr* **)**
	;	
expr	:	*non_oper_expr*
	|	*oper_expr*
	;	
recipient	:	*non_oper_expr*
	|	**super**
	|	**super (** *classid* **)**
	;	
write_expr_list	:	*write_expr*
	|	*write_expr_list* , *write_expr*
	;	
write_expr	:	*expr*
	|	**nl**
	|	**current_time**
	;	

builtin_expr	:	**empty** **(** **)** **@** *non_oper_expr*
	|	**is_class** **(** *classid* **)** **@** *non_oper_expr*
	|	**is_in** **(** *expr* **)** **@** *non_oper_expr*
	|	**get_indexes** **(** *expr* **)** **@** *non_oper_expr*
	|	**indexes** **@** *non_oper_expr*
	|	**indexeslist** **@** *non_oper_expr*
	|	**lower** **@** *non_oper_expr*
	|	**upper** **@** *non_oper_expr*
	|	**getmember_at** **(** *expr* **)** **@** *non_oper_expr*
	|	**getmember** **@** *non_oper_expr*
	|	**empty** **@** *non_oper_expr*
	|	**empty** *non_oper_expr*
	|	*non_oper_expr* **is_in** *non_oper_expr*
	|	**get_indexes** **(** *expr* **,** *non_oper_expr* **)**
	|	**indexes** **(** *non_oper_expr* **)**
	|	**indexeslist** **(** *non_oper_expr* **)**
	|	**lower** **(** *non_oper_expr* **)**
	|	**upper** **(** *non_oper_expr* **)**
	|	*varid* [*expr*]
	|	**getmember** **(** *non_oper_expr* **)**
	|	**eof** **(** **)**
	|	**rtoi** **(** *expr* **)**
	;	
oper_expr	:	*math_expr*
	|	*comp_expr*
	|	*string_expr*
	|	*bool_expr*
	;	

```
math_expr       :  expr + expr
                |  expr - expr
                |  expr * expr
                |  expr / expr
                |  expr mod expr
                |  expr pow expr
                |  sqrt ( expr )
                |  sin ( expr )
                |  cos ( expr )
                |  tan ( expr )
                |  atan ( expr )
                |  abs ( expr )
                ;
comp_expr       :  expr < expr
                |  expr <= expr
                |  expr > expr
                |  expr >= expr
                |  expr <> expr
                |  expr = expr
                ;
string_expr     :  first non_oper_expr
                |  rest non_oper_expr
                |  non_oper_expr ^ expr
                ;
bool_expr       :  not expr
                |  expr and expr
                |  expr or expr
                ;
query_expr      :  [ projection | goal_set ]
                ;
projection      :  ε
                |  any proj_vars
                |  all proj_vars
                |  { proj_vars }
                ;
proj_vars       :  rollvarid
                |  < rollvarid_list , rollvarid >
                ;
```

```
goal_set            :  goal_list
                    |  goal_set ; goal_list
                    ;
goal_list           :  goal
                    |  goal_list , goal
                    ;
goal                :  roll_builtin_goal
                    |  roll_expr == roll_expr
                    |  roll_meth_inv
                    |  ~ roll_builtin_goal
                    |  roll_expr =\= roll_expr
                    |  ~ roll_meth_inv
                    ;
roll_builtin_goal   :  empty ( ) @ roll_expr
                    |  is_class ( classid ) @ roll_expr
                    |  is_in ( roll_expr ) @ roll_expr
                    |  roll_expr < roll_expr
                    |  roll_expr <= roll_expr
                    |  roll_expr > roll_expr
                    |  roll_expr >= roll_expr
                    |  empty roll_expr
                    |  roll_expr is_in roll_expr
                    ;
roll_meth_inv       :  msgid @ roll_expr
                    |  msgid ( ) @ roll_expr
                    |  msgid ( roll_expr_list ) @ roll_expr
roll_expr_list      :  roll_expr
                    |  roll_expr_list , roll_expr
                    ;
roll_expr           :  roll_term
                    |  roll_builtin_expr
                    |  roll_math_expr
                    |  roll_meth_expr
                    |  ( roll_expr )
                    ;
```

roll_term_list : *roll_term*
| *roll_term_list* , *roll_term*
;
rollvar : *rollvarid*
roll_term : *rollvar*
| *rollvarid* : *typeid*
| ! *varid*
| *roll_constant*
| **self**
;
roll_constant : *string_val*
| *int_val*
| *real_val*
| *bool_val*
roll_builtin_expr : **getmember_at** (*roll_expr*) @ *roll_expr*
| *rollvarid* [*roll_expr*]
;
roll_math_expr : *roll_expr* + *roll_expr*
| *roll_expr* - *roll_expr*
| *roll_expr* * *roll_expr*
| *roll_expr* / *roll_expr*
;
roll_meth_expr : *msgid* @ *roll_expr*
| *msgid* ( ) @ *roll_expr*
| *msgid* (*roll_expr_list*) @ *roll_expr*
;
user_method_inv : *msgid* @ *recipient*
| *msgid* ( ) @ *recipient*
| *msgid* (*expr_list*) @ *recipient*
;
new_method_inv : **new** *classid* (*expr_list*)
| **new** *classid* ( )
;
constant_list : *constant*
| *constant_list* , *constant*
;
string_val : *string_con*
int_val : *int_con*
;

```
real_val                 :  real_con
                         ;
bool_val                 :  bool_con
                         ;
string_con               :  STRINGCON
                         ;
int_con                  :  INTCON
                         ;
real_con                 :  REALCON
                         ;
bool_con                 :  BOOLCON
                         ;
constant                 :  string_val
                         |  int_val
                         |  real_val
                         |  bool_val
                         ;
initialiser              :  expr
                         |  collection_initialiser
                         |  sequence_initialiser
                         ;
collection_initialiser   :  {  } : typeid
                         |  {  constant_list }
                         ;
sequence_initialiser     :  [ ] : typeid
                         |  [ constant_list ]
                         |  [ indexed_constant_list ]
                         ;
indexed_constant_list    :  indexed_constant
                         |  indexed_constant_list , indexed_constant
                         ;
indexed_constant         :  ( int_con , constant )
                         ;
```

8.2 Tokenising and Reserved Words

Comments are stripped from input during tokenising. Comments are introduced by the character # which comments out all trailing text on the line.

Numeric and integer constants have the usual representation. Integers are a non-empty sequence of decimal digits optionally preceded by a minus sign. Reals are a non-empty sequence of decimal digits optionally preceded by a minus sign followed by a . character and a (possibly empty) empty sequence of decimal digits. Note that when tokenising negative numbers there must be no space between the sign and the digits. Strings constantsmust be enclosed in " characters. Identifiers are an alphabetic character followed by a (possibly empty) sequence of alphanumeric or _ characters.

For compatibility with the previous ROCK & ROLL release variant spellings are allowed for certain reserved. Also, certain words have been reserved for use by the spatial and active extensions to ROCK & ROLL. A list of reserved words is given below followed by a table of equivalences between new canonical terms and older variants.

The reserved words of the language in alphabetical order are:

`abs`, `action`, `add_member`, `adjacent`, `all`, `and`, `any`, `area`, `area_disjoint`, `atan`, `begin`, `bool`, `border_in_common`, `class`, `closer`, `closew`, `common_border`, `completed`, `condition`, `contour`, `coupling_mode`, `cos`, `count`, `current_time`, `deferred delete`, `diameter`, `different`, `disjoint`, `dist`, `do`, `earliest`, `edge_disjoint`, `edge_inside`, `else`, `empty`, `encloses`, `end`, `end-class`, `end-coupling`, `end-priority`, `end-type`, `end-rule`, `end_class`, `end_coupling`, `end_priority`, `end_type`, `end_rule`, `eof`, `equal`, `false`, `first`, `foreach`, `get_element`, `get_indexes`, `get_member`, `get_member_at`, `if`, `immediate`, `in`, `indexes`, `indexes_list`, `inner`, `insert`, `insert_at`, `inside`, `instead`, `int`, `interface`, `interior`, `intersection`, `intersects`, `is_class`, `is_in`, `isin`, `latest`, `length`, `lines`, `lower`, `meets`, `minus`, `mod`, `mytime`, `new`, `nl`, `not`, `nothing`, `on_border_of`, `openr`, `openw`, `or`, `perimeter`, `plus`, `points`, `priority`, `private`, `program`, `properties`, `public`, `r-interface`, `read`, `real`, `regions`, `remove`, `remove_at`, `remove_index`, `remove_member`, `repeated`, `rest`, `return`, `ROCK`, `ROLL`, `rtoi`, `rule`, `self`, `sent`, `sin`, `specialises`, `sqrt`, `string`, `super`, `tan`, `then`, `times`, `true`, `type`, `upper`, `var`, `vertex_inside`, `vertices`, `when`, `while`, `with`, `within`, `write`

Note that user defined identifiers cannot be reserved words.

Canonical form	**Variants**
insert	add_member
current_time	mytime
end-class	end_class
end-type	end_type
end-rule	end-rule
get_member_at	get_element
ROCK	interface
ROLL	r-interface
is_in	isin
remove	remove_member
remove_at	remove_index

References

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G.Kahn, editor, *Proceedings of the International Symposium in Semantics of Data Types*. Springer-Verlag, June 1984.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990. ISBN 3-540-51728-6, ISBN 0-387-51728-6.

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance Is Not Subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, January 1990. ACM.

[GS95] R.H. Guting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4(2):243–286, 1995.

[Tri90] Phil Trinder. Comprehensions, a Query Notation for DBPLs, 1990. Draft.