NeoClassic User's Guide
Version 0.7

Lori Alperin Resnick, Peter F. Patel-Schneider, Deborah L. McGuinness, Elia Weixelbaum, Merryll Herman, Alex Borgida, Ronald J. Brachman, Charles L. Isbell, Kevin C. Zalondek

1 Introduction

The CLASSIC family of Description Logic-based Knowledge Representation Systems represent information about a domain in terms of descriptions, concept, roles, rules, and individuals. This family emphasizes simplicity of the description language along with completeness and tractability of its inference algorithms.

NeoClassic is the most recent member of the CLASSIC family, and features a tight integration with the C++ programming language, allowing close control of its knowledge representation capabilities without compromising their integrity. NeoClassic currently provides three interfaces: a C++ application programming interface, a graphical user interface, and a character stream interface.

In a Description Logic-based Knowledge Representation System such as NeoClassic, a model of a domain consists of a description of the *concepts* (kinds of individuals) that exist in the domain plus a description of each of the *individuals* in terms of these concepts and relationships between individuals (represented as *roles* and their fillers).

The CLASSIC family extends these basic notions of description logics with *rules*, simple forward-chaining rules of inference. NeoClassic introduces the idea of multiple, independent *knowledge bases*, where each knowledge base contains a related collection of concepts, roles, rules, and individuals.

In the CLASSIC family, some individuals belong to the domain being modeled, i.e., particular wines, in a knowledge base that reasons about wines. (These individuals are said to belong to the **CLASSIC** realm, and are called **CLASSIC** individuals.) Other individuals are used to help describe **CLASSIC** individuals, i.e., ages may be represented as integers, and names may be represented as strings. (These individuals are said to belong to the **HOST** realm, and are called **HOST** individuals.)

This user's guide describes the above notions in NeoClassic, and gives examples of their creation and use. Examples and syntax are generally given as in the character interface—for details on the C++ interface see the NeoClassic reference manual.

2 Descriptions

Descriptions are the most important notion in Description Logic-based Knowledge Representation Systems. In NeoClassic, a description is built up using expressions that contain other descriptions (including concepts) as well as roles and individuals. Descriptions cannot use concepts, roles, or individuals from different knowledge bases. A grammar describing all the ways of building descriptions in the character interface is given in Appendix A.

Descriptions are applied to individuals (in the manner that one-place predicates are applied to constants in predicate logic). An individual is said to be an instance of, or satisfy, a description if the information known about the individual implies the description. In the CLASSIC family, descriptions can be in either the **CLASSIC** realm or the **HOST** realm, depending on which type of individuals they describe.

For example, one could describe "people between the ages of 30 and 40 with at most 5 friends and all of whose friends are doctors" as

```
(and Person
     (all age (and (minimum 30)
                   (maximum 40)))
     (atMost 5 friend)
     (all friend Doctor))
```

If `jill` is known to be a `Person` with `age` 35, with at most 2 `friends`, and with `friends` `bill` and `jack`; and both `bill` and `jack` are known to be `Doctors`; then `jill` would satisfy the above description.

2.1 Building CLASSIC Descriptions

The **CLASSIC and** operator forms the conjunction of some number of **CLASSIC** descriptions. Its syntax is[1]

(**and** *ClassicDescription*$^+$)

For example, a `VegetarianPerson` might be someone who is both a `Vegetarian` and a `Person`:

[1]All NeoClassic input here is given in the syntax of the NeoClassic character stream interface. For details on how to build descriptions and perform NeoClassic operations in the C++ application programming interface, see the NeoClassic Reference Manual.

```
(and Vegetarian Person)
```

A **CLASSIC** **oneOf** description enumerates a set of classic individuals, which are the only possible instances of the description. The syntax is

(**oneOf** *ClassicIndividual*$^+$)

For example, (**oneOf** `White Red Rose`), defines a set of three **CLASSIC** individuals.

The four operators **all**, **atLeast**, **atMost**, and **fills** form special types of descriptions known as *role restrictions*, that restrict the fillers of a role. Depending on the kind of role restriction, either the type of the fillers can be restricted (**all** restrictions), the number of fillers can be restricted (**atLeast** and **atMost** restrictions), or some of the actual fillers can be specified (**fills** restrictions).

A value restriction, or **all** restriction, has the syntax

(**all** *Role Description*)

An **all** restriction specifies that all the fillers of a particular role must be individuals described by a particular description. For an individual `I` to satisfy the value restriction (**all** `r C1`), either all the fillers of `r` on `I` must be known and all of them must satisfy `C1`; or there must be a derivable **all** restriction[2] on `I`, (**all** `r C2`), such that `C1` *subsumes* (is more general than) `C2`.

For example, the instances of

```
(all food Plant)
```

must have all their fillers for `food` be instances of `plant`, such as, for example, `LeafLettuce`.

It is possible for the description in an **all** restriction to be *incoherent* (i.e., a description that can have no instances because it contains conflicting information). This means that the role can have no fillers, which is fine as long as there is not a (positive) **atLeast** restriction on the role. For example, suppose that `Male` and `Female` are two disjoint primitive concepts in the same disjoint grouping. Now suppose the concept `IllegalChildren` were defined as

```
(all child (and Male Female))
```

Since no individual can be described by both `Male` and `Female`, this would be equivalent to the concept of someone with no children:

```
(atMost 0 child)
```

An **atLeast** restriction specifies the minimum number of fillers allowed for a given role on a concept or individual. Its syntax is

(**atLeast** *PositiveInteger Role*)

For example, a `Parent` might be defined to have at least `1 child`:

```
(atLeast 1 child)
```

[2]A *derivable* restriction is one that was either stated as part of the definition, or can be deduced from other information.

For an individual `I` to satisfy the **atLeast** restriction (**atLeast** `n1 r`), either at least `n1` fillers for `r` on `I` must be known, or there must be a derivable **atLeast** restriction on `I`, (**atLeast** `n2 r`), such that `n2` is greater than or equal to `n1`.

An **atMost** restriction specifies the maximum number of fillers allowed for a given role on a concept or individual. The syntax is

(**atMost** *NonNegativeInteger Role*)

For example, an `Orphan` might be defined to have no `parents`:

```
(atMost 0 parent)
```

For an individual `I` to satisfy the **atMost** restriction (**atMost** `n1 r`) there must be a derivable **atMost** restriction on `I`, (**atMost** `n2 r`), such that `n2` is less than or equal to `n1`.

The **fills** operator specifies that a particular role is filled by the specified individuals. The syntax is

(**fills** *Role Individual*$^+$)

where an individual is either the name of a **CLASSIC** individual, e.g., (**fills** `brother Fred Sam`) says that the `brother` role is filled with the individuals `Fred` and `Sam`), or a string or an int or a float, e.g., (**fills** `address "123 Main Street"`) says that the `address` role is filled with the given string). A **fills** description requires that an individual satisfying it must have the given role filled with the given value(s) (and possibly other values also).

For example, `MaleFriendOfSue` might be

```
(and Person (fills friend Sue)
            (fills gender Male))
```

where the filler of the `friend` role is the **CLASSIC** individual `Sue`, and the filler of the `gender` role is the **CLASSIC** individual `Male`. The concept `PersonNamedSam` could be

```
(and Person (fills first-name "Sam"))
```

where the filler of the `first-name` role is the string `"Sam"`.

It is possible to extend the description-forming capabilities of NeoClassic by writing C++ code that determines whether a **CLASSIC** individual is an instance of a **CLASSIC** description. The syntax is

(**testC** *ClassicTestGenerate Parameter**)

ClassicTestGenerate is a NeoClassic name for a C++ object that embodies the procedural test. It must have a member function called run, that takes a **CLASSIC** individual and two sets of individuals (that are modified to indicate dependencies—see Section B.3) and returns one of three values:

`testFalse`: the individual is inconsistent with this description;

`testMaybe`: the individual is currently consistent with this description, but if information is added to the individual, the individual may become either inconsistent with the description or definitely described by the description; or `testTrue`: the individual definitely satisfies this description.

The run function also has access to the parameters of the test.

For details on how to write these C++ objects, see Section B.1.

2.2 Building HOST Descriptions

The **HOST and** operator forms the conjunction of some number of **HOST** descriptions. Its syntax is

(**and** *HostDescription*$^+$)

For example, a `PRIME-NUMBER` might be an integer that is prime

(**and** `INTEGER` (**testH** `prime`))

A **HOST oneOf** description enumerates a set of host individuals, which are the only possible instances of the description. The syntax is

(**oneOf** *Individual*$^+$)

For example, (**oneOf** `3 5`), defines a set of two **HOST** individuals.

A **minimum** description specifies that an individual must be a number with a given lower bound. The syntax is

(**minimum** *Number*)

For example, the description (**minimum** `5`) specifies that the value must be greater than or equal to 5.

A **maximum** description specifies that an individual must be a number with a given upper bound. The syntax is

(**maximum** *Number*)

For example, the description (**maximum** `5`) specifies that the value must be less than or equal to 5.

Intervals are often used for **all** restrictions. For example,

```
(all age (and Integer (minimum 18)
                      (maximum 65)))
```

restricts someone's age to be an integer between 18 and 65.

There is a method for extending **HOST** descriptions in the same way as **CLASSIC** descriptions can be extended. Its syntax is:

(**testH** *HostTestGenerate Parameter**)

HostTestGenerate is a NeoClassic name for a C++ object that embodies the procedural test. The run function is similar to the run function for classic tests, but does not have the extra dependency arguments.

For details on how to write these C++ objects, see Section B.1.

2.3 Building Other Descriptions

It is possible to create incoherent descriptions—descriptions that can never have any individuals that belong to them. Usually this happens as a result of specifying conflicting information, such as

(**and** (**atLeast** `3 r`) (**atMost** `2 r`))

However, there is a direct way of stating an incoherent description, and this description is neither a **CLASSIC** nor a **HOST** description. The empty **oneOf** description specifies an empty set. The syntax is

(**oneOf**)

It is sometimes necessary to provide a universal description—a description that allows any individual, **CLASSIC** or **HOST**, as an instance. Such a description is formed using the universal **and** operator. Its syntax is

(**and**)

3 Concepts

A concept in NeoClassic is a named description, possibly with some extra primitiveness information. A concept is part of a knowledge base, and its description must not use concepts, roles, or individuals from other knowledge bases. As descriptions come in both realms, **CLASSIC** and **HOST**, so do concepts.

When a concept is defined, a name, such as `C`, is associated with its defining description. For example, suppose that `Vegetarian` has been defined as "something all of whose `food` is of type `Plant`",

(**all** `food Plant`)

Then `VegetarianPerson` could be defined as "a `Vegetarian` and a `Person`",

(**and** `Vegetarian Person`)

which is equivalent to "a `Person`, all of whose `food` is of type `Plant`".

3.1 Primitive and Defined Concepts

All **HOST** concepts are completely defined by the description associated with them, but **CLASSIC** concepts can be *primitive*, in that their definition includes something beyond the description associated with them. Such concepts are called *primitive concepts*, and other concepts are called *defined concepts*.

Suppose that a `Vegetarian` is defined as someone who eats only `Plants`, and a `GreenPlantVegetarian` is defined as someone who eats only green `Plants`. NeoClassic will infer that `GreenPlantVegetarian` is a more specific concept than (is subsumed by) `Vegetarian`. Also, if an individual `Fred` is known to eat only `Squash`, and `Squash` is known to be a `Plant`, NeoClassic will determine that `Fred` must be a `Vegetarian`.

Consider on the other hand, primitive concepts. If `Mammal` is defined as a primitive concept, no other concept will be deduced to be more specific than `Mammal` unless it directly or indirectly mentions `Mammal` as a superconcept. No other deductions can determine that a concept `C` is subsumed by `Mammal`.

For example, suppose the concepts `Mammal` and `Invertebrate` are defined as primitive concepts under `Animal`, and the concept `Dog` is defined as a primitive concept under `Mammal`. It is possible to deduce that `Dog` is subsumed by both `Mammal` (this was stated) and `Animal` (this was inherited), but there is no way to deduce that `Dog` is subsumed by `Invertebrate`.

Some primitive concepts have further, disjointness information associated with them in the form of a set of disjoint groupings. A disjoint primitive concept is just like a primitive concept, except that all disjoint primitive concepts that belong to the same disjoint grouping are known to be disjoint from each other. Thus, no individual can satisfy two disjoint primitive concepts in the same disjoint grouping.

The following example is depicted as a hierarchy in Figure 1, where the concepts within a given arc represent members of a disjoint grouping: Suppose the concepts `Man` and `Woman` are defined as disjoint primitive concepts under `Person`, in the `gender` disjoint grouping, and the concepts `OldPerson` and `YoungPerson` are defined as disjoint primitive concepts under `Person`, in the `age` disjoint grouping. This says that no individual can be both a `Man` and a `Woman`, and no individual can be both an `OldPerson` and a `YoungPerson`. However, there could be an individual that is both a `Woman` and a `YoungPerson`. Note that no exhaustiveness assumption is made when reasoning with disjoint primitive concepts (i.e., there could be individuals that are described by `Person`, but which are not described by either `OldPerson` or `YoungPerson`).

3.2 Creating Concepts

Concepts are created by means of the various forms of the function **createConcept**. In all cases the concept is not created if the name of the new concept is already in use as the name of a concept or if the definition of the concept is incoherent.

To create a **HOST** concept use:[3]

(**createConcept** *Symbol HostDescription*)

where the symbol is the name of the concept being defined and the description is the concept definition.

To create a defined **CLASSIC** concept use:

(**createConcept** *Symbol ClassicDescription*)

where the symbol is the name of the concept being defined and the description is the concept definition.

For example, to define a `Vegetarian` as a `Person` who eats only `Plants` use:

```
(createConcept Vegetarian
               (and Person (all food Plant)))
```

[3]In the NeoClassic character interface there is a current knowledge base and concepts are created in that knowledge base.

To create a primitive **CLASSIC** concept use

(**createConcept** *Symbol Classic Description* `true`)

where the symbol is the name of the concept being defined and the description is the concept definition.

To define `Mammal` as a primitive concept under `Animal` use:

```
(createConcept Mammal Animal true)
```

To create a disjoint primitive concept use:

(**createConcept** *Symbol ClassicDescription Symbol*$^+$)

where the first symbol is the name of the concept being defined, the description is the concept definition, and trailing symbol(s) specify which disjoint grouping(s) the concept belongs to.

The following example defines the disjoint primitive concepts `Man` and `Woman` under `Person` in the `gender` disjoint grouping, and `OldPerson` and `YoungPerson` under `Person` in the `age` disjoint grouping:

```
(createConcept Man Person gender)
(createConcept Woman Person gender)
(createConcept OldPerson Person age)
(createConcept YoungPerson Person age)
```

3.3 Built-in Concepts

Each knowledge base in NeoClassic comes with a number of built-in concepts.

The concept `Thing` has as its definition a universal description. All descriptions are subsumed by `Thing`. The concept `ClassicThing` (also named `classic-thing`) has as its definition the most-general **CLASSIC** description. The concept `HostThing` (also named `host-thing`) has as its definition the most-general **HOST** description. All **CLASSIC** individuals are instances of `ClassicThing`, and all **HOST** individuals are instances of `HostThing`.

There are four built-in concepts under `HostThing` in NeoClassic. They are `Number`, `Integer`, `Float`, and `String`. The definitions of these concepts are built-in host test descriptions that recognize numbers, integers, floats, and strings, respectively.

4 Individuals

Individuals are specific instances of concepts. Just as in the case of concepts, individuals are divided into two realms: **CLASSIC** and **HOST**. **CLASSIC** individuals are used to represent the real-world objects of a domain, while **HOST** individuals are objects in the C++ language. For example, the **CLASSIC** individual `Fred` might be an instance of the primitive **CLASSIC** concept `PERSON`, while the **HOST** individuals `10000` and `"Harry"` would be instances of the **HOST** concepts `NUMBER` and `STRING`, respectively. **HOST** individuals cannot be created or modified, but they can be used in **fills** and **oneOf** descriptions.

When **CLASSIC** individuals are created, they are given a description. The language of operators for **CLASSIC** individual descriptions is identical to that for

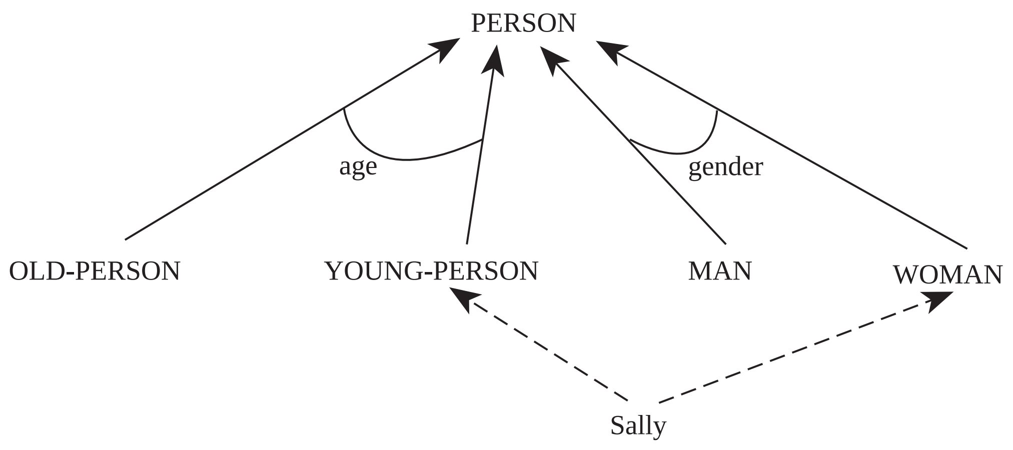

Figure 1: A hierarchy with disjoint groupings.

CLASSIC concept descriptions. In addition, by means of a function call, a role on an individual can be closed. This asserts that the currently known fillers of the role on the individual are provably all the fillers. Just like concepts, **CLASSIC** individuals are part of a knowledge base, and their description must not use concepts, roles, or individuals from other knowledge bases.

The function **createIndividual** is used to create a **CLASSIC** individual.[4] The syntax is

(**createIndividual** *Symbol ClassicDescription*)

where the symbol is the name of the individual being created, and the description is the definition of the individual. For example, to create the individual `Mary` as a `PERSON`, use:

```
(createIndividual Mary PERSON)
```

To create the individual `Sam` as a `PERSON`, both of whose parents are `DOCTORS`, and who has a brother `Fred` and at least 1 sister, say:

```
(createIndividual Sam
        (and Person (all parent Doctor)
                    (fills brother Fred)
                    (atLeast 1 sister)))
```

There is one other thing that can be stated about an individual—that a particular role is closed, i.e., it can have no more fillers. For reasons beyond the scope of this guide[5], there is no **close** operator as part of the language, but instead, there is a function, **closeRole**, which is used for this purpose. Its syntax is

(**closeRole** *Individual Role*)

For example, if `Sue` is `80` years old, has child `Joshua`, and has no other children, create `Sue` as follows:

[4] Just as for concepts, the character interface creates **CLASSIC** individuals in the current knowledge base.

[5] For the interested reader, the reason is roughly that this would make concept descriptions autoepistemic (and thus nonmonotonic).

```
(createIndividual Sue
        (and Person (fills age 80)
                    (fills child Joshua)))
```

and then call (**closeRole** `Sue child`).

Explicitly closing a role is the *only* way a role can be closed. A role becomes *full* on an individual when the **atMost** restriction is reached by the number of fillers for the role, i.e., the role can hold no more fillers, but this is different from the role being closed. For example, if the `age` role is an attribute (with an implicit **atMost** restriction of `1`—see Section 5), and `Mary`'s `age` role is filled with `25`, then the `age` role on `Mary` is full, because it can have no more fillers, but `age` is not closed on `Mary`.

An open-world assumption is used in NeoClassic. Unless information to the contrary is known or deducible, it is assumed that there may be more values associated with that role for the individual. This affects NeoClassic's deductions in many ways.

If `John` has `children Mary`, `Fred`, and `Sam`, all of whom are `LAWYERS`, there is no way of knowing that *all* of `John`'s `children` are `LAWYERS` unless it is possible to independently derive that `John` satisfies this description, or that `John` has no more `children`. For example, if there is a derivable description on `John` of *at most* `3 children`, then NeoClassic deduces that `John` has no more `children` (because he already has `3`), and that his `child` role is full, and thus **all** restriction is satisfied.

By the same principle, suppose that `John` has `friends Jack` and `Jill`. There is no way of knowing that `John` has *at most* `2 friends` unless it is possible to independently derive that `John` satisfies this description (either because the `friend` role had been closed for `John`, or because there was such an **atMost** restriction derivable on `John`).

5 Roles

Roles are entities that represent the properties of **CLASSIC** individuals. They map **CLASSIC** individuals to other (**CLASSIC** or **HOST**) individuals. The roles of a **CLASSIC** individual can either be "filled" by individuals (called the role fillers) or have their potential fillers restricted by certain concepts (i.e., as type descriptions), or both, where the fillers and descriptions can be in either the **CLASSIC** realm or the **HOST** realm. *Attributes* are special types of roles that have an implicit maximum number of fillers of `1`.

The syntax for defining a role is:[6]

(**createRole** *Symbol Boolean*)

where the symbol is the name of the role, and the boolean specifies whether the role is an attribute (and defaults to false if missing).

A role `r` can be thought of as a two-place predicate: `r(individual1, individual2)`, where the role-predicate is TRUE if `r` on `individual1` is known to have the value (filler) `individual2`. For example, `brother(Mary, John)` would be TRUE if the `brother` role on the individual `Mary` is known to have the value `John`. A role on an individual may have any number of values, possibly none.

6 Rules

NeoClassic provides three different types of forward-chaining rules: simple rules, which will be referred to as *rules*; *computed description rules*; and *computed filler rules*.

A (simple) rule consists of an antecedent, which must be a concept, and a consequent, which is a description. Suppose there is a rule with concept `C1` as the antecedent and concept `C2` as the consequent (i.e., `C1` is the left-hand side of the rule, and `C2` is the right-hand side of the rule). Then as soon as an individual `I1` is known to belong under concept `C1`, the rule is "fired", and `I1` is deduced to belong under `C2`. A concept or individual does not need to be described by the consequent `C2` in order to be classified under (described by) the antecedent `C1`. However, once the rule is fired, the individual may be further classified based on the new information provided by the rule.

For example, suppose there is a rule with `Person` as the antecedent and

```
(and (atLeast 1 social-security-number)
     (all social-security-number SSNUM)).
```

as the consequent. If `Mary` is created as a `Person`, then this rule "fires", and she is known to have a `social-security-number` that is of type `SSNUM`. If there happened to be a concept `ThingWith-SocialSecurityNumber`, defined as

```
(atLeast 1 social-security-number),
```

then `Mary` would ultimately get classified under this concept.

As a second example, suppose that in order to determine that someone is a `Vegetarian`, it is enough to know that the only kind of `food` she eats is of type `Plant`. Suppose also that there is a rule stating that anyone who is a `Vegetarian` is known to be a `HealthyThing`. Thus, in order to determine that someone is a `Vegetarian`, NeoClassic is only required to know the kind of `food` she eats, but once she is known to be a `Vegetarian`, NeoClassic can immediately "fire" the rule and infer that she is a `HealthyThing`.

The function **createRule** creates a simple rule.[7] Its syntax is

(**createRule** *Symbol ClassicConcept*
ClassicDescription)

A *computed description rule* is similar to the simple rule described above, except that instead of the rule being defined with a consequent that is specified at rule creation time, the rule is defined with a function and parameters, which, when the rule is fired, are used to generate the consequent of the rule. Thus, the consequent is not specified in advance, but can be based on information that is known about the individual at the time the rule is fired.

A computed description rule is defined with the function **createRule**. The syntax is

(**createRule** *Symbol ClassicConcept*
`(computedDescription` *Fn Parameter**`))`

It takes a name (a symbol), a **CLASSIC** concept for the antecedent, a C++ object representing the function to run for the rule, and, optionally, a list of parameters for the function.

A *computed filler rule* is like a computed description rule, except that in addition to the function and parameters, it takes a role, and the function generates a list of fillers for the role when the rule is fired.

A filler rule is defined with the function **createRule**. The syntax is

(**createRule** *Symbol ClassicConcept*
`(computedFillers` *Fn Role Parameter**`))`

It takes a name (a symbol), a **CLASSIC** concept for the antecedent, a C++ object representing the function to run for the rule, and, optionally, a list of parameters for the function.

7 Knowledge Bases

A knowledge base is simply a collection of concepts, individuals, roles, and rules, and the relationships between them. It is possible to create and destroy knowledge bases by name and to switch between knowledge bases.

In the character interface, functions work on the current knowledge base, so it is possible to ignore this aspect of NeoClassic if only one knowledge base is needed.

[6]As before, the character interface creates roles in the current knowledge base.

[7]Again, the character interface creates roles in the current knowledge base.

8 Updates

Once an individual has been created, it may be updated in one of two ways: additional information can be added to the individual, or existing information specified for an individual can be retracted. Both of these types of updates on individuals are described in this section. The definition of a concept also cannot be modified. The instances of a concept, however, can all get modified indirectly, if a rule is created which has that concept as the antecedent, because the rule will then fire on all instances of the concept.

8.1 Addition

An individual can be asserted to have additional parents or descriptions, or new fillers for a role, with the **addToldInformation** function. The syntax is

(**addToldInformation** *Individual ClassicDescription*)

where the individual is an existing classic individual, and the description is the information to be added to the individual. For example, to add the information that the existing individual `Mary` has `child John`, say:

(**addToldInformation** `Mary` (**fills** `child John`))

It can also be asserted for an individual that the currently known fillers of a role are the only fillers (via **closeRole** in Section 4 above).

8.2 Retraction

Information previously asserted about an individual can be retracted.

removeToldInformation retracts information from an individual. The syntax for `removeToldInformation` is

(**removeToldInformation** *Individual ClassicDescription*)

For example, to remove the child of Mary above,

(**removeToldInformation** `Mary` (**fills** `child John`))

uncloseRole removes the told fact that a particular role is closed on an individual. The syntax for **uncloseRole** is

(**uncloseRole** *Individual Role*)

Note that retraction only removes the information directly told by means of individual creation or adding information to individuals. If the information was independantly derived from other information, the retraction will have no effect.

9 Inference in NeoClassic

In each knowledge base NeoClassic creates a taxonomy of concepts and individuals, based on the subsumption relationship, so it has to be able to (efficiently) answer subsumption questions. Determining subsumption in NeoClassic is a two-part process, consisting first of normalization, and then of checking whether one normalized concept is more general than another.

To perform normalization, NeoClassic takes the input (told) information about a concept or individual, and computes an inferred (derived) version. The derived version contains information from several sources, including rule-firings and propagations (for individuals), and from inheritance, classification, and combining information in a number of ways (for both concepts and individuals). The derived version contains all the information that could be deduced about the concept or individual. Functions that retrieve information about concepts and individuals come in two varieties: one that retrieves told information and one that retrieves derived information.

NeoClassic is normally only concerned with the normalized (derived) version of a concept or individual, except when performing retraction—as only told information can be retracted, and in explanation—where told information is the root cause of all information.

This section presents an informal discussion of the normalization and subsumption process so that users can understand how NeoClassic makes certain inferences. It is not necessary to fully understand this section (especially the parts on classification) in order to be able to use NeoClassic.

9.1 Concepts

Normalization

The normalization of concepts includes :

- inheriting information from parent concepts,
- combining descriptions on the same role,
- eliminating embedded **and** operators,
- deducing fillers when they are implied by **all** restrictions,
- deducing fillers when they are restricted to `Integers`, and they are implied by a fully-specified interval (**minimum** and **maximum** restrictions are both specified),
- inheriting and combining role restrictions up and down role hierarchies,
- checking the consistency of descriptions, disjoint primitive concepts, etc.,
- combining **oneOf** descriptions (taking their intersection),
- filtering **oneOf** descriptions which are restricted to **HOST** individuals, by all applicable test descriptions, since the properties of **HOST** individuals cannot change (this is not done for **CLASSIC** individuals since their properties can change).

For example, suppose a `HealthyVegetarian` is defined as

```
(and Person (all food HealthyThing)
            (all food Plant)),
```

where `Person`, `HealthyThing`, and `Plant` are all primitive concepts. Then the normalized form of `HealthyVegetarian` is

```
(and Person
     (all food (and HealthyThing Plant))).
```

The **all** restrictions on the `food` role were combined, and the embedded **and**, which was redundant, was removed.

Now define an `OldHealthyVegetarian` as

```
(and HealthyVegetarian (all age Old)).
```

Then the normalized form of `OldHealthyVegetarian` is

```
(and Person
     (all food (and HealthyThing Plant))
     (all age Old))
```

It inherits the primitive superconcept `Person`, as well as the **all** restriction on `food`.

Suppose concept `C` is defined as

```
(all age (and Integer
              (oneOf 20 25 30)
              (testH evenp)))
```

where **evenp** is a **HOST** test function that recognizes even integers. The normalized form of `C` is

```
(all age (and Integer
              (oneOf 20 30)
              (testH evenp)))
```

Note that, since `25` is not an even integer, it has been filtered out.

The last part of normalization checks concepts for consistency. Examples of inconsistent concepts are:

```
(and (atLeast 2 child) (atMost 1 child));
```

```
(and (all child Male) (all child Female))
```

where `Male` and `Female` are two disjoint primitive concepts of the same disjoint grouping;

```
(and (fills child Jack)
     (all child (oneOf Mary Sue)))
```

since `Jack` is not a member of the set `(oneOf Mary Sue)`;

```
(and (minimum 3) (maximum 2));
```

and

```
(and (fills age 17) (all age (minimum 18)))
```

since `17` is inconsistent with the specified interval.

Subsumption

A concept `C1` *subsumes* another concept `C2` if `C1` is an equivalent concept to `C2`, or `C1` is a more general concept than `C2`. Once two concepts have been normalized, NeoClassic can easily determine whether one concept subsumes the other. Basically, in order for `C1` to subsume `C2`, for each description on `C1` there must be an equivalent or more specific description on `C2`.

For example, there are the primitive concepts `Mammal` and `Plant` under the concept `ClassicThing`, the primitive concept `Person` under `Mammal`, and the primitive concept `Fruit` under `Plant`. Now define the concept `VegetarianMammal` as

```
(and Mammal (all food Plant)),
```

and the concept `FruitEatingPerson` as

```
(and Person (all food Fruit)).
```

`VegetarianMammal` subsumes (is more general than) `FruitEatingPerson`, because:

- `Mammal` subsumes `Person` (`Person` was defined to be more specific than `Mammal`); and
- the **all** restriction on `food` for `VegetarianMammal`, `Plant`, subsumes the **all** restriction on `food` for `FruitEatingPerson`, `Fruit`.

The Classification Process

Classification of a concept involves finding the parents, children, and direct instances of the concept, and firing any rules with this concept as antecedent on all instances of the concept. NeoClassic finds the parents (most specific subsumers) of `C` as follows: First start at the top of the concept hierarchy (with either `ClassicThing` or `HostThing`, depending on the realm of `C`—assume it is a **CLASSIC** concept for the sake of this example). For each concept `Ci` that is known to subsume `C` (starting with `ClassicThing`), check each of its children `Cj` to see if `Cj` subsumes `C`. The lowest such set of concepts in the concept hierarchy are `C`'s parents.

To find `C`'s children (most general subsumees), start with the parents you have just found for `C`, and for each child of the parents, see if `C` subsumes it. If so, you have found a child; if not, try its children until you have found a child of `C`, or a concept has no children.

To find `C`'s instances (direct instances, not descendant instances), look at each individual which is a descendant instance of one of `C`'s parents, but is not a descendant instance of one of `C`'s children. If `C` subsumes it (see Section 9.2), it is an instance of `C`.

Now any rules with `C` as antecedent are fired on `C`'s instances (descendant instances, i.e., all instances), and these individuals are reclassified (see Section 9.2 below).

9.2 Individuals

Normalization

The normalization of an individual includes all the steps involved in the normalization of a concept. In addition, it involves running any test functions on the individual (for checking its consistency), and propagating information to other individuals when new facts are implied by **all** restrictions.

After inheriting information from all its parents, an individual is checked for consistency. The examples of inconsistent concepts from Section 9.1 would also cause inconsistent individuals. The following are some additional examples of inconsistent individuals:

- `John` contains `(all child Male)` and `(fills child Mary)`, where `Mary` is a `Female`, and `Male` and `Female` are two disjoint primitive concepts of the same disjoint grouping;
- `Jack` contains `(atLeast 2 child)` and `(fills child Jill)` (and no other **fills** statements), and the `child` role is then closed with **cl-ind-close-role**;
- `Sally` has 2 fillers for the `child` role, `Fred` and `Sam`, and contains the description `(atMost 1 child)`;

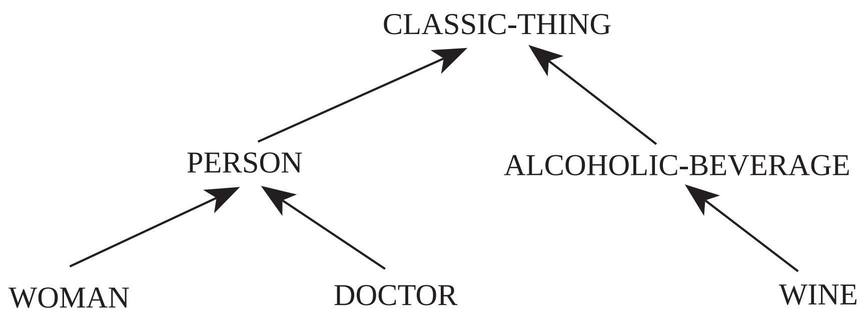

Figure 2: Hierarchy for individual subsumption example.

- `Harry` contains `(all age Integer)` and `(fills age 25.5)`.

Propagation

Whenever an individual, `i`, that has fillers for any role is normalized. These fillers are then given the value restriction for this role on `i`. This process is referred to as *propagation*, since information has been *propagated* from `i` to the filler.

For example, consider:

```
(createConcept ProudParent
   (and Person
        (atLeast 1 child)
        (all child Doctor)))
(createIndividual Ben)
(createIndividual MrsCasey
   (and ProudParent (fills child Ben)))
```

When `MrsCasey` is normalized, NeoClassic recognizes that `MrsCasey` has a `child` filler, `Ben`, and that all `child` fillers must be `Doctor`. Therefore, NeoClassic propagates `Doctor` to `Ben`. If it turned out that `Ben` had other information that was contradictory to `Doctor`, then an error would be generated due to the contradiction, and the entire operation would be cancelled.

Propagation only occurs on individuals that have a value restriction and a filler(s) on the same rule, and not on concepts that have a value restriction and a filler(s) on the same role. For example, suppose that `Doctor` and `Bum` were defined as disjoint primitive concepts:

```
(createConcept Doctor ClassicThing job)
(createConcept Bum ClassicThing job)
```

Now define two concepts, `ProudParentOfBen` and `EmbarrassedParentOfBen`, that would cause `Ben` to be a `Doctor` and a `Bum` respectively.

```
(createConcept ProudParentOfBen
   (and (all child Doctor)
        (fills child Ben)))
(createConcept EmbarrassedParentOfBen
   (and (all child Bum)
        (fills child Ben)))
```

While these two concept definitions seem to be contradictory, they are both legal and allowed to exist at the same time. However, once an individual is defined to be an instance of one of the above two concepts, no individual could be defined as an instance of the other. For example, suppose one tried to define the following two individuals:

```
(createIndividual MrsCasey ProudParentOfBen)
(createIndividual MrCasey EmbarrassedParentOfBen)
```

Then following the first creation, NeoClassic would propagate `Doctor` to `Ben`. Following the second creation, NeoClassic would attempt to propagate `Bum` to `Ben`, which would generate an error since `Bum` and `Doctor` are disjoint concepts. As a result, `MrCasey` would not be added to the knowledge base.

Subsumption

An individual `I` is subsumed by, or satisfies, or is an instance of a concept `C` if `I` is described by `C`, i.e., if `I` satisfies every description on `C`. Once an individual has been normalized, NeoClassic can easily determine whether or not it is subsumed by, or satisfies, a concept. Basically, in order for concept `C` to subsume a normalized individual `I`, each description on `C` must be satisfied by `I`, either because there is an equivalent or more specific description on `I`, or because the description on `C` can be derived from the descriptions, fillers, and closed roles on `I`.

For example, suppose there are the primitive concepts `Person` and `Alcoholic-Beverage` under the concept `ClassicThing`, the primitive concepts `Woman` and `Doctor` under the concept `Person`, and the primitive concept `Wine` under the concept `Alcoholic-Beverage` (see Figure 2).

Now define the concept `C1` as a `Person` with at least `1 child`, all of whose `children` are `Doctors`, and all of whose `friends` drink only `Alcoholic-Beverages`:

```
(and Person
     (atLeast 1 child)
     (all child Doctor)
     (all friend
          (all drink AlcoholicBeverage)))
```

Suppose that `Fran` is a `Woman` with exactly `2 children`, `Jack` and `Barbara`, both of whom have been previously asserted to be `Doctors`; all her `friends` drink only `wine`; and `Susy` is one of her `friends`:

```
(and Woman
      (fills child Jack Barbara)
      (atMost 2 child)
      (all friend (all drink Wine))
      (fills friend Susy)).
```

Then `C1` subsumes `Fran`, because each description on `C1` is satisfied by `Fran`:

- `Person` subsumes `Woman`;
- Fran has *at least* `1 child`—in fact, she has `2 children` (and both are known);
- *all* her `children` are `Doctors` (all of them are known, and they have all been previously asserted to be `Doctors`);
- *all* her `friends drink` only `AlcoholicBeverages`—in fact, they all `drink` only `Wine`, which is subsumed by `AlcoholicBeverage`.

The extra fact known about `Fran`—that `Susy` is her `friend`—does not affect this question of subsumption.

The Classification Process

Classification of an individual involves finding the parents of the individual, propagating any information from that individual to other individuals and reclassifying them, and firing any appropriate rules on the individual. The parents of an individual are found in the same way as the parents of a concept, using the subsumption check for individuals described above (an individual has no children, and of course, no instances).

If the individual has any new parents as a result of being classified, all rules with the new parents or their ancestors as antecedents are fired on the individual, and the individual is reclassified. For example, if `Mary` is classified as a `VegetarianPerson`, then any rules with `VegetarianPerson` or `Person` as antecedents are fired.

Information may be propagated from this individual to other individuals (see the example in Section 9.2 above, where `Jim`'s favorite food was inferred to be `Lasagne`). Any affected individuals are reclassified.

10 Incompleteness in NeoClassic

There are several types of incompleteness in NeoClassic that need to be mentioned. For concept processing, subsumption is incomplete in several ways. Some of these incompletenesses also apply to individual processing. In addition, there are some incompletenesses in rule processing and propagations for individuals. These types of incompleteness are all discussed below.

10.1 Concept Processing

There are a few ways in which concept subsumption is incomplete.

Test descriptions are treated as black boxes, in the same way as primitive concepts are. In fact there is really no way for NeoClassic to determine the meaning of test descriptions. For example, the built-in **HOST** concept `Integer` contains the test description `(testH integerp)`. If the concept `EvenInteger` is defined as `(testH evenp)`, then it will *not* be classified under `Integer`, because NeoClassic does not know that anything which satisfies the **evenp** test must also be an `Integer`. To truely define `EvenInteger` it must be defined using `Integer` (see Sections 2.1 and 2.2), as `EvenInteger` is:

```
(and Integer (testH evenp))
```

The only exception to treating test functions as black boxes is that `(testH integerp)` and `(testH floatp)` are subsumed by `(testH numberp)`.

Also, with respect to concept subsumption, **CLASSIC** individuals are treated as having no properties. The reason for this is that the concept hierarchy should not change when individuals change (concept definitions cannot change). This impacts the two operators **fills** and **oneOf**. If the individual `Mary` is known to be an `Athlete`, and concept `C` is the description of someone whose only child is `Mary`

```
(and (fills child Mary) (atMost 1 child))
```

then `C` is *not* subsumed by the concept of someone all of whose children are athletes

```
(all child Athlete)
```

This applies to **CLASSIC oneOf** descriptions as follows: if it is known that `Joe` and `Mary` are both `Mammals`, `(oneOf Joe Mary)` is not classified below `Mammal`. Thus, a **CLASSIC** concept containing only a **oneOf** description will be classified directly below either `ClassicThing` or another concept containing only a **oneOf** description, but never below a concept containing any other type of description.

Even if the properties of individuals are implied by the presence of these individuals in concept descriptions, NeoClassic doesn't take these properties into account. For example, suppose that concept `C1` is defined as

```
(and (fills child Sally) (all child Athlete)
      (fills friend Sally) (atMost 1 friend))
```

and concept `C2` is defined as `(all friend Athlete)`. NeoClassic does *not* infer that `C1` is subsumed by `C2`.

As a more complex example, suppose that `Susan` is known to have `Bob` as a `client` and `David` is known to have `Bill` as a `client`. Let `C1` be defined as

```
(and Company
      (atLeast 1 employee)
      (all employee (fills client Jack))
      (all employee (oneOf Susan David))
      (all contractor (atMost 1 client))
      (all contractor (oneOf Susan David)))
```

and let `C2` be defined as

```
(atMost 1 contractor)
```

NeoClassic does *not* infer that `C1` is subsumed by `C2`, because the establishment of the subsumption would require the use of contingent properties of `Susan` and `David`.

The above non-subsumptions are not really incompletenesses in NeoClassic, as the standard definition of subsumption ignores contingent properties of individuals. However, NeoClassic is incomplete with respect to this standard definition because it ignores properties of individuals that are implied by their presence in descriptions. For example, if `C3` was defined as

```
(and Company
     (atLeast 1 employee)
     (all employee (fills client Jack))
     (all employee (oneOf Susan David))
     (all contractor (atMost 1 client))
     (all contractor (oneOf Susan David))
     (fills r Susan) (all r (fills client Bob))
     (fills s David) (all s (fills client Bill)))
```

NeoClassic would not infer that it was subsumed by `C2`, even though this inference does follow from the standard definition of subsumption.

To detect this subsumption, NeoClassic would have to determine that either `Susan` or `David` must be an `Employee`; if `Susan` is an `Employee`, then she can't be a `Contractor` because she has to have at least 2 `clients`; if `David` is an `Employee`, then he can't be a `Contractor` because he has to have at least 2 `clients`. This reasoning by cases is computationally difficult, which is one reason it is not implemented in NeoClassic.

As another example, suppose that the concept `C4` is defined as

```
(and (fills child Sally) (all child Athlete)
     (fills friend Sally) (atMost 1 friend))
```

and concept `C5` is defined as `(all friend ATHLETE)`. NeoClassic does *not* infer that `C4` is subsumed by `C5`.

HOST oneOf descriptions do not have the same types of incompleteness as **CLASSIC oneOf** descriptions, because the properties of **HOST** individuals do not change. A **HOST** concept containing only a **oneOf** description may be classified under a concept containing a test description, if all the **HOST** individuals in the **oneOf** description satisfy the test description. In addition, a **HOST** concept containing only a **oneOf** description can be classified under a concept containing an interval description, if the **oneOf** description contains only numbers, and they are all within the specified interval.

10.2 Individual Processing

The incompletenesses in concept subsumption can appear when determining whether or not an individual satisfies a concept description. This is because **all** restrictions on individuals are handled as concepts, not as descriptions of individuals. Suppose that the individual `Sam` is known to be a `Vegetarian`, and that the individual `Mary` is defined as someone all of whose friends have `Sam` as a teacher, and no one else:

```
(all friend (and (fills teacher Sam)
                 (atMost 1 teacher)))
```

`Mary` will not be found to satisfy the concept described by

```
(all friend (all teacher Vegetarian))
```

because the properties of `Sam`, specifically that he is a `Vegetarian`, are not taken into account when doing the subsumption test.

Both rules and propagations are performed only on known instances. Thus, if NeoClassic knows that all `Mary`'s `sisters` are `Athletes`, and she has at least 1 `sister`, it does not create a skolem individual representing the `sister`, in order to reason about it.

Rules in NeoClassic are treated only as forward-chaining inferences, not as logical inferences. Thus, there is a rule stating that if someone is a `Vegetarian`, then he is known to be a `HealthyThing`, and NeoClassic knows that `Joe` is an `UnhealthyThing` (a concept disjoint from `HealthyThing`), it does *not* infer that `Joe` is not a `Vegetarian`.

11 Error Handling

There appear to be three kinds of errors that can occur in NeoClassic: syntax errors in NeoClassic input, such as mismatched parentheses; evaluator errors, such as providing the wrong kind of argument to a function; and knowledge errors, such as creating an inconsistent individual. However, the first two kinds of errors are really errors in the interface to NeoClassic and not really NeoClassic errors at all. These errors are discussed along with the interface.

Real NeoClassic errors result from syntactically valid calls to NeoClassic functions that attempt to perform illegal knowledge operations. There are three main kinds of illegal operations in NeoClassic: incorrect uses of names, attempts to create incoherent concepts, and attempts to make a knowledge base inconsistent.

Incorrect uses of names include attempts to look up objects by name, when no object of that type with that name exists; and attempts to create a second object of a given type with a given name. For example, if a concept with the name `Person` already exists in a knowledge base, a call of the form

```
(createConcept Person ...)
```

would be illegal. Functions calls that use names incorrectly do not perform the intended operation. Instead they return a "null" pointer of the type normally returned.

Attempts to create incoherent concepts do create a concept, but the concept is not actually added to the knowledge base. An incoherent concept is created and returned, but this concept is not part of any knowledge base. An example of this kind of knowledge error is

```
(createConcept Person (and (atLeast 3 r)
                           (atMost 2 r)))
```

Attempts to make a knowledge base inconsistent include attempts to create or modify an individual that would result in some individual inconsistent, and attempts to create a rule that would make an individual inconsistent. The creation functions do create an individual or rule, but this individual or rule is not added to the knowledge base. The individual or rule returned from attempts to make the knowledge base inconsistent encodes the (inconsistent) state of the knowledge base that would have resulted if the operation would have been performed, but the knowledge base ends up effectively unchanged. An example of this kind of knowledge error is

```
(cl-create-ind Mary
               (and (atMost 0 son)
                    (fills son Jack)))
```

(assuming the `son` role has already been defined).

Incoherent concepts and inconsistent individuals have associated with them the inference that caused the concept to become incoherent or the individual to become inconsistent. Inconsistent rules have associated with them an inconsistent individual that caused the rule to become inconsistent. The inference can be examined and printed in the same manner that any other inference resulting from explanation can be examined and printed.

12 Explanation

To be provided later.

A The CLASSIC Grammar

The following defines the syntax for typing in a CLASSIC concept or individual description:

Description ::= *ThingDescription* |
ClassicDescription |
HostDescription |
IncoherentDescription

ThingDescription ::= `Thing` |
(and)

ClassicDescription ::= `ClassicThing` |
ClassicConcept |
(**and***ClassicDescription*$^+$) |
(**oneOf** *ClassicIndividual*$^+$) |
(**atLeast** *PositiveInteger Role*) |
(**atMost** *NonNegativeInteger Role*) |
(**fills** *Role ClassicIndividual*$^+$)
(**fills** *Role HostIndividual*$^+$) |
(**all** *Role Description*) |
(**testC** *ClassicTestGenerate Parameter**)

HostDescription ::= `HostThing` |
`Number` | `Integer` | `Float` | `String` |
HostConcept |
(**and** *HostDescription*$^+$) |
(**oneOf** *HostIndividual*$^+$) |
(**minimum** *Number*) |
(**maximum** *Number*) |
(**testH** *HostTestGenerate Parameter**)

IncoherentDescription ::= `(one-of)`

Role	::=	*Symbol*
ClassicConcept	::=	*Symbol*
HostConcept	::=	*Symbol*
Rule	::=	*Symbol*
ClassicIndividual	::=	*Symbol*
HostIndividual	::=	*"string"* | *int* | *float*
ClassicTestDetail	::=	*Symbol*
HostTestDetail	::=	*Symbol*
Number	::=	*int* | *real*
Parameter	::=	*NeoObject*

B Extending NEOCLASSIC

NEOCLASSIC can be extended by means of C++ code. Such code is used in test descriptions and computed rules.

B.1 Writing Test Functions

A test function is required for classic test descriptions and host test descriptions. A test function is actually a C++ class with several member functions, only one of which need be written by the user.

The simplest kind of test function takes no parameters. For example, a test function that tests integers for primality would be written

```
HostTestFunction0(PrimeTest,primep);

TestVal
PrimeTest::run(const HostIndividual& ind)
                              const {
  int n = (int)ind;
  if (n <= 1) {
    return testFalse;
  }
  int bound = (int) sqrt((double)n);
  for (int i = 2; i <= bound; i++) {
    if (((n / i) * i) == n) {
      return(testFalse);
    }
  }
  return testTrue;
}
```

The first line of this example is a macro that sets up a C++ class for the test function, along with most of the data members and member functions. It also creates a generator for this test function, named `primep` and adds this name to the collection of host test function generators. The only part that need be written is the `run` function, which is a function that takes (in this case) a **HOST** individual and returns an element of `TestVal`.

This test function would be used in a host test description as follows:

(**testH** `primep`)

In general a test function is set up using a macro of the form

Kind`TestFunction`*Paramno*
(*Class*, *Name*, *Type1*, . . . , *TypeN*);

where *Kind* is either `Host` or `Classic`, *Paramno* is the number of parameters the test function has, *Class* is the name of the class for the test function, *Name* is the name of the test function in the character interface, and *TypeI* is the type of the i'th parameter for the test function. *Paramno* can range from 0 to 2. It can also be suffixed with an `L` to indicate that the last parameter is constructed from the trailing parameters used in the test description.

The `run` function for such a test class is written

```
TestVal
Class::run(const HostIndividual& ind) const {
  ...
}
```

or

```
TestVal
Class::run(const ClassicIndividual& ind,
           Set<ClassicIndividual> *pdeps,
           Set<ClassicIndividual> *ndeps)
      const {
  ...
}
```

this function can refer to data members `arg1` and so on to refer to the actual parameters used in the test description. The run member functions of **CLASSIC** test functions have two extra arguments, that are used to compute dependencies if necessary.

For example, the test function in Figure 3 mimics the **atLeast** description constructor.

The `run` function, when applied to an individual, must return one of the three possible elements of the `TestVal` class:

`testFalse`: the individual is inconsistent with this description;

`testMaybe`: the individual is currently consistent with this description, but if information is added to the individual, the individual may become either inconsistent with the description or definitely described by the description; or

`testTrue`: the individual definitely satisfies this description.

To guarantee correct behavior, test functions must follow these requirements, which are not enforced by NeoClassic:

- When information is added to an individual `I`, the result of applying a test function `f` to `I` may change from `testMaybe` to either `testTrue` or `testFalse`, but it may *never* change from `testTrue` to `testFalse`, or vice versa. (This requirement does not hold during the *retraction* of information from the individual—see Section 8.2.)
- When a **HOST** test run function is applied to a **HOST** individual, the function must return either `testTrue` or `testFalse`, because **HOST** individuals are unchanging, by definition.
- A **CLASSIC** test function must compute dependencies as necessary, provided that the `pdeps` and `ndeps` arguments are non-null. If adding information to some other **CLASSIC** individual could cause the answer returned from the test function to change, then that individual must be added to the `pdeps` argument, provided that that argument is non-null; if removing information from some other **CLASSIC** individual could cause an answer returned from the test function to change, then that individual must be added to the `ndeps` argument, provided that that argument is non-null.

Normally, test descriptions are not used alone, but are used as part of **and** descriptions (see Sections 2.1 and 2.2). For example, suppose the user defines the concept `EvenInteger` as a subconcept of `Integer` (a built-in concept—see Section 3.3), with a test function (**evenp**) that decides whether or not the integer is even:

```
(and Integer (testH evenp))
```

NeoClassic will first test to see if the individual is an integer, and then an even integer. Test functions can depend on this and can safely perform operations that are dangerous to objects that do not belong to the parents (i.e., the function **evenp** might blow up if called on a `String`, but this will never happen since before running the **evenp** function on an individual, NeoClassic will first run the **integerp** function, which the concept `EvenInteger` inherits by being a subconcept of `Integer`, and if this test fails, **evenp** will not be called). Thus, the test descriptions inherited from the parent concepts are always guaranteed to be run before the told test descriptions on a concept, although there is no way of determining the ordering of the test descriptions between the different parents, or the ordering between the different told descriptions for a concept.

Consider defining an `Employee` as a `Person` with at least `1 employer`, and who has an `age` between `18` and `65`. First use the above mechanisms to write a C++ class whose run function checks to see if an integer is within a range.

```
HostTestFunction2(RangeCheck,rangeCheck,int,int);

TestVal
RangeCheck::run(const HostIndividual& ind)
                           const {
  int i = ind;
  if (ind >= arg1 && ind <= arg2 ) {
    return testTrue;
  } else {
    return testFalse;
  }
}
```

Next define the concept `Employee` by the following concept description:

```
(and Person
     (atLeast 1 employer)
     (all age (and Integer
                   (testH rangeCheck 18 65))))
```

Note that `Integer` must be specified as part of the **all** restriction, because the conversion to `int` would not be

```
ClassicTestFunction2(AtLeastTest,atLeast,unsigned int,Role);

TestVal AtLeastTest::run(const ClassicIndividual& ind,
                         Set<ClassicIndividual>* pdeps,
                         Set<ClassicIndividual>* ndeps) const {
  ClassicIndividualDescription def = ind.getDefinition();
  unsigned int indmin = def.derivedAtLeast(arg2);
  unsigned int indmax = def.derivedAtMost(arg2);
  if (indmin >= arg1) {
    return testTrue;
  } else if (indmax < arg1) {
    return testFalse;
  } else {
    return testMaybe;
  }
}
```

Figure 3: atLeast test function

valid if the host individual was not an `int`. Note: this concept could have been defined using an interval (see Section 2.2), but a test function was used for illustrative purposes.

The following is a more complicated example, requiring a three-valued test function. Consider defining a `SuccessfulParty` as a `Party` where the number of male guests is the same as the number of female guests. First create a C++ class that takes two roles. It returns `testTrue` if both roles provably have exactly the same number of fillers, `testFalse` if both roles provably have a different number of fillers, and `testMaybe` otherwise. Note that "provably" here involves the roles being closed, or some provable generic relation between the **atLeast** and **atMost** descriptions on the roles.

Next define the concept `SuccessfulParty` by the following description:

```
(and Party (testC sameNumberFillers
                  maleGuests femaleGuests))
```

If the individual `party1` is a `Party` with exactly `8` `maleGuests` and exactly `8` `femaleGuests` (whether the guests are known, or just known to exist), then it will be classified as a `SuccessfulParty`. If the individual `party2` is asserted to be a `SuccessfulParty`, then as long as the **same-number-fillers** test does not return `testFalse` on `party2`, it is consistent for it to be a `SuccessfulParty`.

B.2 Allowable Operations in User Functions

User functions are not allowed to change the knowledge base in any way whatsoever, nor can they depend on the order in which inferences are performed, except as detailed here.

The simplest thing to do within a user function is to ask questions about the properties of the current individual (see the **sameNumberFillers** test function above). The user can assume that the individual has been completely normalized (see Section 9), and all local inferences have been done (i.e., those not involving any other individuals, and those that don't depend on the firing of rules).

It is trickier to access the properties of related individuals in the knowledge base, since not all inferences may have been done when the test function is run. In addition, if the properties of related individuals change, then unless a test function returns dependencies (see the discussion later), there is no way for NeoClassic to know that it must reclassify the affected individual (since a test function is like a black box, and NeoClassic cannot automatically calculate dependency relationships based on information in a test function).

For example, consider defining the `DoctorChild` concept as someone who has at least 1 child who is a doctor. First define a test function, **doctorChildFn**, which checks to see if any filler of the `child` role is an instance of the `Doctor` concept.

```
ClassicTestFunction0(DoctorChildFn,doctorChildFn)

TestVal
DoctorChildFn::run(const ClassicIndividual & ind
                   Set<ClassicIndividual> *,
                   Set<ClassicIndividual> *) const {
  Role role = "child";
  Concept doctor = "Doctor";
  IndividualSet fillers =
      ind.derivedFillers(role);
  ...
     doctor.subsumes(filler);
  ...
```

Then define the concept `DoctorChild` as (**testC doctorChildFn**). If `Mary` is created as a `Doctor`, and `John` is created as someone whose `child` is `Mary`, then `John` will be correctly classified under `DoctorChild`. However,

```
ClassicTestFunction2(SameNumberFillers,sameNumberFillers,Role,Role);

TestVal SameNumberFillersf::run(const ClassicIndividual& ind,
                                Set<ClassicIndividual> *,
                                Set<ClassicIndividual> *) const {
  int min1 ind.derivedAtLeast(arg1);
  int min2 ind.derivedAtLeast(arg2);
  int max1 ind.derivedAtMost(arg1);
  int max2 ind.derivedAtMost(arg2);
  if ( min1==max1 && min2==max2 && min1=min2 ) {
    return testTrue;
  } else if ( min1<max2 || min2<max1 ) {
    return testFalse;
  } else {
    return testMaybe;
  }
}
```

Figure 4: The sameNumberFillers function

if `Fred` is created as someone whose `child` is `Harry`, and it is later asserted that `Harry` is a `Doctor`, then NeoClassic will not know to reclassify `Fred`, and thus will not discover that `Fred` is now a `DoctorChild`. At some later point, if information is added to `Fred`, he will be reclassified under `DoctorChild`.

It is acceptable to produce side effects within test functions, *as long as the side effects are outside of* NeoClassic. However, test functions are run whenever NeoClassic determines that they need to be run, and this may depend on current implementation details. Thus, it is not meaningful to increment a counter every time a test function is run. *The user must not write test functions which produce side effects within* NeoClassic. It is not acceptable for a test function to call any NeoClassic functions that modify the knowledge base.

B.3 Dependencies for Test Functions

***Warning*: For Advanced Users Only**

During classification, NeoClassic calculates and keeps track of certain dependency relationships. For example, if all of `Mary`'s `children`'s `children` are known, and they are all `Athletes`, then `Mary` would be classified under the concept `GrandparentOfAthletes`. However, if one of her grandchildren, say `Fred`, stops being an `Athlete` at any point (the parent concept `Athlete` is removed from him), NeoClassic must know to reclassify `Mary` so that she is no longer under the concept `GrandparentOfAthletes`. Thus, when `Mary` is classified under `GrandparentOfAthletes`, all her `children` and grandchildren are stored as *negative dependencies* of `Mary`, which means that if any information is `removed` from them, `Mary` needs to be reclassified.

In addition, if `Mary` is not classified under `GrandparentOfDoctors`, because two of her grandchildren, `Lou` and `Sarah`, are not known to be `Doctors`, then `Lou` and `Sarah` are stored as *positive dependencies* of `Mary`, which means that if any information is `added` to them, `Mary` needs to be reclassified.

Since test descriptions are basically black boxes to NeoClassic, i.e., NeoClassic has no way of knowing what goes on inside them, NeoClassic cannot automatically calculate dependencies when an individual is being tested for subsumption against a concept containing a test description. Thus, if a test description uses information about other individuals when it is run on an individual, then if things change about those other individuals, NeoClassic somehow needs to know to reclassify that individual.

This can be done if a test function provides dependency information. Test functions can modify the positive and negative dependency sets passed in to them. As an example of a test function that returns dependencies, see the function **cl-test-all-closed?**, in **library.lisp**, which takes a list of roles and/or role-paths, and returns whether or not all the role-paths are closed on the individual. If all role-paths are closed, then any intermediate individual along any path is on the ndeps list (if information is removed from one of these individuals, specifically, that the role is closed, then this test function should no longer return `testTrue`). Otherwise, if any role-path is non-closed, then

- The individual which doesn't have the closed role is the only individual in the pdeps list.
- All intermediate individuals leading up to and including that individual are on the ndeps list (if any of these fillers is removed, then the pdeps need to be recalculated).

B.4 Writing Functions for Computed Rules

The functions for computed rules are similar to test functions, except that they are always run on **CLASSIC** individuals and return either a `Description` (for a computed description rule) or an `Individual Set` (for a computed fillers rule).

Computed Description Rules

A computed description function class is created

> `ComputedDescription`*Paramno*
> (*Class*, *Name*, *Type1*, ..., *TypeN*) ;

The `run` function for a computed description function takes as arguments a **CLASSIC** individual. When the rule is fired on a **CLASSIC** individual, the `run` function is called on the individual, and it produces a NeoClassic description, which is then added to the individual. The function must be written so that once it is run on an individual, it will always produce the same result, even if new information is added to the individual.

Computed Fillers Rules

A computed fillers function class is created

> `ComputedFillers`*Paramno*
> (*Class*, *Name*, *Type2*, ..., *TypeN*) ;

The `run` function for a computed fillers function takes as arguments a **CLASSIC** individual. When the rule is fired on a **CLASSIC** individual, the `run` function is called on the individual, and it produces an `IndividualSet`, whose elements are then added as fillers of the role given in the rule. This role is an implicit first parameter of the function class, and so it can be obtained as the value of `arg1`. The function must be written so that once it fires on an individual, it will always produce the same list of individuals, even if new information is added to the individual.

An illegal use of a filler rule would be a rule that fires, and the function generates an empty set as the result, but if more information were added to the individual, the function will generate a non-empty set of individuals.