CBECC Rule Processor Reference Manual

Version 1.0

September, 2016

Contributors Include:

Scott Criswell, Wrightsoft Corporation

Martha Brook, California Energy Commission

David Reddy, 360 Analytics

Jason Glazer, GARD Analytics

# Introduction

## Purpose

This document is intended to provide energy code compliance rule authors with information about the structure and contents of the CBECC data model and ruleset source to assist in their efforts to create and maintain CBECC rulesets.

## Definitions, Acronyms, and Abbreviations

### Definitions

***ACM Manual*** - A manual published by the CEC that documents the requirements for approval of Alternative Calculation Methods (also referred to as compliance software programs) used to demonstrate compliance with the California Energy Efficiency Standards.

***Application Programming Interface*** (API) - A mechanism that serves as an interface between different software programs in a manner that facilitates their interaction, similar to the way a software tool’s user interface facilitates interaction between the (human) user and the software running on the user’s computer.

***Building Transformation***- describes a building model either specified by the user or generated during the course of testing the models compliance to the Energy Code. A typical non-residential Title-24 compliance analysis run includes 5 individual building transformations, (1) the beginning user-defined building model, (2) and proposed HVAC sizing model, (3) and final/annual proposed building model, (4) the baseline HVAC sizing model and (5) and final/annual baseline building model.

***Compliance rules*** - procedures that restrict or modify attributes in the computer description of the budget building design and/or proposed design. Compliance rules restrict or overwrite building description information input by the user and are used in the process of generating the energy cost budget and design energy cost.

***Data model*** -defines the structure and organization of data associated with a specific tool or purpose. In the case of this requirements document, the term data model refers to the structure and organization of data used to describe building designs that the software analyzes and manipulates in order to determine whether the building complies with a specific energy code.

***Data-model approach ruleset*** - refers to a compliance ruleset whose rules are defined and organized based on the list of component types and attributes defined in the building data model (SDD for this project). The order in which rules are evaluated in a data-model approach ruleset is defined by the Rules Engine, as opposed to procedural approach rulesets where the rules are evaluated in the same order in which they are defined/listed in the ruleset source.

***Energy Code*** - refers to the energy code which is represented in the compliance rules (i.e. California’s 2013/16/19 Title-24).

***Extensible Markup Language*** (XML) - A set of rules for encoding documents in machine-readable form emphasizing simplicity, generality, and usability over the Internet.

***Input Data Format (IDF)*** – Input file format for the building energy simulation program, EnergyPlus.

***OpenStudio*** – open source, cross-platform middleware software development kit (SDK) to make it easier to use EnergyPlus, and to provide higher-level functionalities related to software interoperability, multi-domain simulation workflows (EnergyPlus + Radiance), standards, and analysis.

***OpenStudio Model –*** OpenStudio’s building energy model functional block and corresponding API.

***OpenStudio Model file format (OSM)*** – serialized file format of a single building model instance. At this time OSM is very similar to IDF.

***Portable Document Format (PDF)*** – an open standard for document exchange, first created by Adobe Systems in 1993, used for representing documents in a manner independent of application software, hardware, and operating systems. Each PDF file encapsulates a complete description of a fixed-layout flat document, including the text, fonts, graphics, and other information needed to display it.

***Procedural approach ruleset*** - refers to a compliance ruleset whose rules are defined in a series of lists (rulelists). The rules in each rulelist are evaluated on building models in the order in which they are defined within the rulelist. This is in contrast to data-model approach rulesets where rule evaluation order is determined by the Rules Engine based on each rule’s dependencies.

***Rule Circularity*** - refers to a condition where rules that describe two or more building attributes depend on one another such that there is no specific evaluation order that ensures each attribute be set to its final value prior to being referenced by a rule setting another attribute. Rule circularities are acceptable for procedural approach rulesets, where the rule author specifies the order in which each rule is evaluated. Rule circularities are however not allowed in data-model approach rulesets where the Rule Engine is responsible for determining the order in which each rule is to be evaluated.

***Rulelist*** - refers to a list of rules contained in the (typically procedural) ruleset.

***Simulation engine*** - a building energy simulation computer program, such as EnergyPlus.

***Software Development Kit*** (SDK) - Software that facilitates the implementation of tools that take advantage of capabilities of a particular system or are designed to operate on a particular type of hardware.

***Standards Data Dictionary*** (SDD) - the collection of building components and descriptors that are manipulated by the compliance rules during the process of testing a building model for compliance to an energy code.

### Acronyms, and Abbreviations

***ACM*** - Alternative Calculation Method

***AEC*** - Architectural Energy Corporation

***ASCII*** - American Standard Code for Information Interchange

***ASHRAE*** - American Society of Heating, Refrigerating and Air-Conditioning Engineers, Inc.

***API***  - application programming interface

***BEM*** - building energy model

***CEC*** - California Energy Commission

***CEM*** - compliance energy model

***COMNET*** - The Commercial Energy Services Network

***DOE*** - United States Department of Energy

***DLL*** - dynamic link library

***IDF*** - input data format

***NREL*** - National Renewable Energy Laboratory

***OSC*** - OpenStudio Component

***OSM*** - OpenStudio Model

***PDF*** - Portable Document Format

***SDD*** - standards data dictionary

***SDK*** - software development kit

***XML*** - extensible markup language

## References

California Energy Commission, NORESCO & Wrightsoft. 2014. *Functional Requirements for ACM Standards Compliance Software*. Dec-2014.

CEC. 2008. *Nonresidential Alternative Calculation Method (ACM) Approval Manual for the 2008 Building Energy Efficiency Standards*. Dec-2008. CEC-400-2008-003-CMF

CEC & AEC. 2010. *Nonresidential Alternative Calculation Method (ACM) based on COMNET*. Draft - Oct-2010.

Regional Economic Research, Inc. *COMcheck-Plus Compliance Ruleset Documentation*. Dec-2003.

# Compliance Energy Model and Ruleset Source

Files that describe the compliance energy model (CEM) and ruleset are defined outside of source code compiled into executable (.exe or .dll) files. These software tools are designed to provide flexible frameworks and evaluation engines for models and rules that can be defined and applied to building descriptions without having to recompile executable files. This design allows for the easy interchange and update of rulesets without requiring re-distribution of compiled software and also helps to ensure that non-software developers have the power to create and modify building models and rules without possessing the tools, knowledge or experience that goes into creating modern software tools and graphical interfaces.

This section describes some fundamental attributes of the building energy model that is acted upon by a ruleset as well as the individual source files that make up the CEM and rulesets.

## CEM Data Model Component Relationships

Compliance energy models are made up of a fairly broad and complex range of information, including architectural, thermodynamic, energy loads/gains and HVAC equipment performance and controls. Inputs covering these various areas are divided into separate components by topic or use that enable the consolidation of energy simulation input data so that the model needn’t include an excessive amount of duplicate data. In order to facilitate this data consolidation while also allowing for a great amount of flexibility and specificity in the building models, the underlying data storage mechanism must be capable of managing component relationships that fall into a couple of different categories that we refer to as (a) parent/child and (b) object assignment.

### Component Parent/Child Relationships

Compliance energy models are comprised primarily of hierarchical components. An example of topmost components in a CEM might be those that describe aspects of the site and overall facility. The next level of components describe individual buildings and floors within each building, then spaces/zones followed by walls and finally windows and doors. The relationship between these components is referred to as parent/child. A space within a building belongs to one and only one parent floor, making the space a child of the floor. A wall is a child of a space, that space being the parent of the wall, etc.

### Component Assignment

Many components of the building model are designed to consolidate data that describes the performance characteristics of another component or other energy simulation data that might otherwise be repeated many times throughout a building model. Examples of components of this nature include wall construction, window performance, internal gains hourly profiles, and a variety of other data. A single component describing these data in a CEM can typically be assigned to many other components, sometimes of varying type themselves. For instance, window performance data can be stored in a Glass Type component that can be referenced through many individual windows and/or skylights throughout a building model.

## CEM Data Model Source Files

There are two separate files that define the CEM data model. The first and primary file lists and provides information pertaining to each component of the data model and all attributes/properties that can be used to describe each component. The secondary CEM data model source file defines the character strings and associated numeric values for each CEM model attribute that can be specified by the user as a selection from a pre-defined list of options (referred to as enumerations).

### CEM Base.txt

The CEM Base text file lists the individual component types and attributes that describe each type of CEM component. The principal data that describes each component type includes:

* Short and long names used to identify the component type (SName & LName),
* An integer value specifying the maximum number of components of that type that can be present in a single building model (MD),
* Up to 15 individual parent component type names (short names of up to fifteen other component types that can serve as a parent to this component type) (P1-P15), and
* Topic ID to facilitate help documentation linkages (HI).

Each individual CEM component can be described by many individual attributes. The principal data that describes each component attribute or property includes:

* Short and long names used to identify the attribute (SName & LName),
* The type of data that describes the attribute, such as float, integer, character string, symbol (enumeration) or object assignment (VType),
* The number of values that define this attribute (1 for a single value vs. > 1 for arrays of values) (NV),
* A character string to label the units that describe the attribute (UL),
* The name of the component type that can be assigned to this attribute (if the attribute is an object assignment) (OC), and
* Item ID to facilitate help documentation linkages (HI).

### CEMEnumerations.txt

The CEMEnumerations file is where the lists of options are defined for each CEM component attribute that is defined as type Symbol, meaning that it is specified by the user as a selection from a list of pre-defined options. Each symbol (single option within a list) is identified by a unique numeric value and the lists of options available to the user can depend on properties of the local component or any global components of the building model.

Enumerations can also be defined within RULE blocks of data model rulesets. If enumerations for a certain property are defined in both the CEMEnumerations file and RULE blocks, then the RULE block definition will take precendence.

## Compliance Ruleset Source Files

The compliance rules input files are ASCII text files that include all the data and logic necessary to test compliance of a building model to the Energy Code. The formats of these files are designed to make creation and maintenance of compliance rulesets as simple as possible for rule authority organizations.

The source files describing a compliance ruleset can be defined in a couple of different ways. Past rulesets of this nature have been organized to include a series of procedural-oriented lists of rules (rulelists) accompanied by a series of supporting files that provide additional data referenced by the rules or used during rule evaluation. Each list of rules is designed to manipulate a building model in a certain way, and these lists may be evaluated multiple times or perhaps not at all during the course of a single compliance analysis based on the individual building model being analyzed by the ruleset. Source files consistent with this organization can be found in the Procedural Compliance Ruleset Source Files section below.

A newer organization of ruleset source files is referred to as the data model approach. This approach consolidates data and rules related to each individual building model component attribute into a single block of text (as opposed to the procedural rulesets where this data might be spread across multiple files). Source files consistent with this organization can be found in the Data Model Approach Ruleset Source Files section below.

### Procedural Compliance Ruleset Source Files

Procedurally organized rulesets are made up of a variety of individual text/CSV files that contain tables of information or rules. Each of the unique types of file that can be included in a procedurally organized ruleset is described in the following sections. As is the case throughout the text source files, file headers should be referenced for the latest information describing the format and contents of each file.

#### Primary Compliance Ruleset Text File

The primary ruleset text file contains a series of identifiers and references to other files that are to be compiled into the ruleset. These identifiers and subordinate file references include:

* Ruleset structure version – a numeric version ID denoting the version of ruleset structure or organization that the ruleset is compatible with to facilitate ruleset source backward compatibility.
* Ruleset ID & version – identifiers available at runtime to identify the ruleset and its version.
* Table list – a list of references to text/CSV look-up tables containing data that can be referenced by rules throughout the ruleset.
* Data types – a separate text file containing ruleset-specific data pertaining to each individual building model component attribute.
* Range – a separate text file containing range limits (message, warnings and errors) for a variety of building model component attributes.
* Enumerations – a file similar to the CEMEnumerations text file that enable ruleset-specific option lists that override those contained in the CEMEnumerations file.
* Resets – a file used to identify a series of component attribute relationships that cause certain attributes to be re-defaulted (user input removed) in the event that other attributes are modified by the user or compliance ruleset.
* Component Libraries – one or more files that include individual component definitions that can be imported into building models and referenced in building models generated by the ruleset.
* Rulelist Files – individual files that contain lists of rules designed to manipulate a building model.

#### Rulelist Text File

A text file that contains one or more rulelists (individual lists of rules). There are a series of pre-defined (reserved) rulelist names that are applied to a building model during certain phases of the compliance analysis and rule authors can create any number of additional rulelists that can be reference by rules contained in either the pre-defined or other supplemental rulelists.

Refer to the Rule Syntax and Functionality Reference section of this document for more information on the format of rules and rulelists.

#### Look-up Table Text/CSV File

Each compliance ruleset can reference one or more tables of data. The use of tables is often the most convenient way to integrate conditional data. These files can contain any number of both independent and dependent data arranged in columns. This format is consistent with tables of building description inputs provided in many compliance rule documents.

Each look-up table reference includes three pieces of data:

* a table name which can be referenced throughout the compliance rule expressions as if it were a predefined ruleset function,
* the number of table independents (the number of values passed into a look-up table in order to return a specific dependent value), and
* the number of dependents defined in the table.

Throughout the compliance rule expressions, data contained in the look-up tables can be referenced by indicating the table name followed by parenthesis’ containing look-up values for each independent variable in the table and a 1-based integer index identifying the dependent variable to return.

#### Data Types Text File

The data types file includes a series of values that are used by the compliance ruleset and user interfaces alike. These data are specified for each individual component attribute defined in the building data model and include:

* Attribute classification – one of several options such as Compulsory, Required, Prescribed, Default Optional and NotInput, used primarily by user interfaces in determining how to handle the interface to the attribute.
* An integer flag value (0/1) indicating whether or not it is a “primary” attribute (an independent variable upon which other attributes describing the same building component depend). Attributes with this flag value set should be collected immediately after a component of the type that this attribute describes is created in a user interface.
* An integer flag value (0/1) indicating whether or not the attribute can be edited by the user from within a user interface. This flag should be set to TRUE for all attributes which must be input by the user and should be set to FALSE for all attributes whose values are prescribed by the compliance rules.
* Additional flag values that can be referenced by user interfaces (refer to a DataTypes.txt file header for more details).

#### Range Limits Text File

In order to help guide the user and prevent the input of errant data, the compliance ruleset can define a series of range checks to be performed on any data input or modified by the user. The data describing each individual range check include:

* The name of the attribute for which the range check is defined.
* A flag indicating whether the range check value represents a minimum or a maximum value.
* The floating point range check value.
* A logical condition (such as “Building:Area> 0”)which must evaluate to TRUE in order for the range check to be enforced (or “none” if it is to be enforced at all times).
* A range check classification, including Message, Warning and Error.
* A message that provides an explanation of the range check violation.

#### Enumerations Text File

A text file whose contents and format is identical to that described above in the section entitled CEMEnumerations.txt.

#### Attribute Reset Definitions Text File

This file contains a series of building component attribute groups. Each group is led by one or more records beginning with a value of 0 followed by a component attribute and each group is concluded by one or more records beginning with a value of 1 (followed by a component attribute). Whenever the user modifies a building attribute that is referenced in a ‘0’ (leading) reset data record, then all attributes that are referenced in subsequent ‘1’ (trailing) records will be re-defaulted, meaning that any user-specified data is removed and the attribute is then re-defaulted.

#### Component Library Text File

Each compliance ruleset can reference one or more libraries, each of which contains any number of building components which may be referenced by the compliance rule expressions. This file is formatted very much like the ASCII files which store the user input for building description data. The main difference is that the library is made up of a collection of building components which do not describe a single building or project, but instead describe a wide variety of schedules, equipment performance curves, and any other components which are specifically defined by the compliance ruleset.

### Data Model Approach Ruleset Source Files

The syntax for the data model ruleset approach is significantly different than the procedural compliance ruleset syntax and is described in the following sections. Conceptually, the data model ruleset approach is based on a few simple concepts that together provide an expanded method of expressing the rules for a performance based building energy standard such as the Non-residential Alternative Calculation Method (NACM) of California’s Building Energy Efficiency Standards (Title 24).

At a high level, there are three main components of the Compliance Engine to understand:

* The hierarchical building data representation (the SDD XML)
* The ruleset that transforms the hierarchical building data representation provided by the user into similar hierarchical building data representations for the proposed and baseline building.
* Each hierarchical building data representation gets translated into input suitable for a building energy simulation program and then simulated. The simulation results for the baseline and proposed would be compared and if the proposed energy costs (based on time-dependant valuation for Title 24) are less than the baseline the building would pass.

This chapter focuses on the second part, the ruleset, but it is important to understand how the ruleset is connected to the hierarchical data representation and the simulation program. The following paragraphs provide a conceptual overview and the sections that follow dive deeper into the details.

* The user’s building description is expressed in a XML document based on a XML Schema file. This XML document would typically be generated by a software program used during the building design process. Such a program would usually have a graphical user interface that allows users to edit the geometric, envelope, lighting, and HVAC attributes for a building. The XML document is an interchange format that directly corresponds to the user’s hierarchical building data representation.
* The software would take the XML document that corresponds to the user’s hierarchical building data representation and the rules expressed in a specific syntax and create one or more hierarchical building data representations. Imagine if the standard consisted of only two rules that required a minimum amount of insulation and a maximum amount of lighting power. The resulting baseline hierarchical building data representation would look almost identical to the user’s hierarchical building data representations except the data items related to insulation and lighting power would likely be altered.
* The ruleset is expressed in a group of text files in the same directory which operate on the user’s hierarchical building data representation (as provided in the XML document) and create a series of similar but modified hierarchical building data representations for the proposed model and the baseline model. The files contain blocks of text for describing the ruleset including: RULE, TABLE, REPLACE, and TRANSFORMATIONS which will be described further below. All files in the ruleset would have file extensions of TXT or CSV.
* Each RULE in the group of text files is based on a specific data element from hierarchical building data representation and uses spreadsheet-like expressions to show how that data element should be modified for the proposed version and the baseline version (and any other version) of each created hierarchical building data representation. The ordering of the RULEs is not important since the rule processor program determines how each is dependent on one another and computes the values in the proper order. See the section titled RULE and RULE NEW Blocks below for more details.
* For expressing complex rules, intermediate values are computed using RULE NEW blocks which have the same syntax as the RULE block. New temporary data elements are created in each hierarchical building data representation that can be used for other computations. See the section titled RULE and RULE NEW Blocks below for more details.
* RULEs can reference TABLEs of data that are either present in the same text file or in other text files. Each TABLE effectively creates a new function that can be used within a specific expression in a RULE with independent and dependent variables. The each TABLE would be complete with every possible combination of independent variables. Each TABLE consists of one or more columns of independent values and one or more columns of dependant values. The RULEs reference a specific TABLE and provide values that are looked up to find a result. Much of the standard is represented as tables so this feature is used to represent those tables. See the section titled TABLE Block below for more details.
* When portions of one or more of the hierarchical building data representations need to be removed and replaced, the text files of the rules contain a REPLACE block which describes which portions of the hierarchical data building representation are removed and how they are replaced. This is usually limited to only certain hierarchical building data representations (baseline or proposed, etc.). The rules related to the inserted data elements would contain the details on how to set their values in this case. See the section titled REPLACE Block below for more details.
* One at a time, the hierarchical building data representations created by the rules for the proposed and baseline (and perhaps others) would be translated into a building energy simulation input file such as an IDF file for EnergyPlus or an INP file for DOE-2. Each would be simulated and the summary results from the simulation would be added back into the data model. Having the results available for other rules is especially important for sizing of HVAC equipment.
* The number of different hierarchical building data models transformed by the software can itself be defined using the TRANSFORMATIONS block. The TRANSFORMATIONS block describes each transformed hierarchical building data model based on the original user provided hierarchical building data model. Commonly, the TRANSFORMATIONS include a proposed model used for HVAC component sizing, a proposed model, a baseline model used for HVAC component sizing, and a baseline model.

The ruleset is completely described using text files. The files should be organized by topic but this is for the convenience of the rule developers, since ultimately the contents of all files are compiled together. No file is the “main” file. Dependencies of the rules in the files are determined on an expression-by-expression basis so no ordering of the files is required. Circular referencing would be flagged by the compiler and identified so the rule author can correct. Most files will contain both RULE blocks and TABLE blocks and one file would contain a single TRANSFORMATIONS block. All files in a directory with TXT or CSV file extensions would be part of the ruleset.

The following sections provide additional details on each block that can appear in the group of text files that express the rules.

#### RULE and RULE NEW Blocks

Most files will contain RULE blocks which begin with a line labeled RULE and end with a line ENDRULE as shown below:

RULE<data model element>

<content of a rule>

ENDRULE

Each RULE block operates on a single specific data element from the data model. It would contain everything related to that data element including defaults, maximum, minimum, transformations for the proposed and baseline building, references, error messages, help messages, units, etc. The RULE block would be divided into areas for defaulting, proposed, and baseline as well as general restrictions. The rules for each transformation are the last parts of the RULE block using the names from the TRANSFORMATIONS block. The RULE blocks do not need to be shown in the order of execution and instead are evaluated in order of dependency, like cells in a spreadsheet. Strict dependency checking is expected that is not tied to any conditional expressions. Dynamic dependency checking that does rely on how conditional expressions are evaluated does not need to be supported.

The <data model element> is a reference to the data element that the results of the rule expressions get assigned. For example, building.envelope.window.fenestrationConstruction.solarHeatGainCoefficient would reference the Solar Heat Gain Coefficient for all windows in the building. The portions of the RULE block that sets the values would be evaluated and the result set to the value of that data element. The RULE is executed for each specific instance of the data element in the hierarchical building data representation. If the rule pertains to windows, it would be executed for each window in the data model. However, since expressions can contain logical conditions, the values can be changed or not changed depending on the specific logic.

Each RULE block starts with the RULE line and ends with the ENDRULE line and can contain the sections shown below:

RULE <data model element>

DESCRIPTION

HELP

REFERENCE

INPUTCLASS

MINIMUM

COMMONMINIMUM

COMMONMAXIMUM

MAXIMUM

UNITS

OPTION

DEFAULT

SIZING\_PROPOSED

ANNUAL\_PROPOSED

SIZING\_BASELINE

ANNUAL\_BASELINE

ENDRULE

After each section, an expression, a list of options, values, or text may appear.

A fictitious example of a complete rule is shown below:

RULE building:construction:rValue

DESCRIPTION

"Insulation R Value”

HELP

"Multiple lines of help about the rules pertaining to this particular

rule. The start and end double quotes can be on different lines"

REFERENCE

NonResACM2.1

Section6.3Table23

INPUTCLASS

Optional

MINIMUM

0.001

COMMONMINIMUM

8

COMMONMAXIMUM

38

MAXIMUM

100.00

UNITS

h-ft2-F/Btu

DEFAULT

11

SIZING\_PROPOSED

u:building:construction:rValue

ANNUAL\_PROPOSED

u:building:construction:rValue

SIZING\_BASELINE

IF building:climateZone > 3 THEN

12

ELSE

14

ENDIF

ANNUAL\_BASELINE

zb:building:construction:rValue

ENDRULE

The DESCRIPTION, HELP and REFERENCE sections provide information about the rule. The primary purpose of these sections is to provide rule reviewers with additional context on the purpose of the rule. When an error occurs this information would be provided to the user so they know where to start looking to find a reason for the error.

The sections called MINIMUM, COMMONMINIMUM, COMMONMAXIMUM, and MAXIMUM show the allowed and preferred range of values for the data model element. If the value is below the MINIMUM or above the MAXIMUM values, a run time error is produced and building model would be deemed non-compliant. The COMMONMIMIMUM and COMMONMAXIMUM are values which will prompt the user to provide additional supporting documentation if values are outside that range. The values, such as the 38 in the COMMMONMAXIMUM example above, are usually just numeric however, they could also be computed values represented by a expression including functions. While the MINIMUM, COMMONMINIMUM, COMMONMAXIMUM, and MAXIMUM sections are primarily needed to filter invalid values from the USER data model, their use with the transformed data models provide an additional check on computed values. It is expected that they would be evaluated at the end of the process for error reporting only.

INPUTCLASS options include:

* Compulsory – User MUST provide input, collected first when object created.
* Required – User MUST provide input if they apply to the proposed design.
* Optional – User MAY provide input if they apply to the proposed design.
* Default – Default value provided in rules (which user may override).
* Prescribed – Can be defined in the user's model (but not a documented user input).
* NotInput – No user input allowed.

The UNITS block shows the specific units that apply to the numeric values shown in other parts of the rule. The UNITS shown should be consistent with the units used in the data model XML schema. If the values in the UNITS block are not consistent an error would be generated.

The DEFAULT block shows the value used if not provided by the user data model.

The remaining sections each correspond to a specific transformation defined in the TRANSFORMATION block described below. For the NACM ruleset development effort, it is anticipated that four transformations will be used titled SIZING\_PROPOSED, ANNUAL\_PROPOSED, SIZING\_BASELINE, ANNUAL\_BASELINE but theoretically fewer or more transformations could be defined within each RULE block. For each transformation, the value shown or computed from the expression is put into the data model for the data model element referenced at the top of the RULE. These sections are the foundation of the use of rules to modify data elements in a data model. They define the value of data model elements for each TRANSFORMATION, and therefore, are the actionable statements evaluated in each case. The expressions can be as simple as a numeric value or as complicated as nested IF statements.

The previous example did not include the OPTION section. The OPTION section is only for data model elements that are not numeric, and instead can be one of a specific group of identifiers. This is often called an enumerated list. The list shown in the OPTION block will also be represented in the XML Schema file. The two sources of OPTION values should be checked for consistency. All values shown in OPTION blocks are available, none are available conditionally.

The RULE NEW block is very similar to the normal RULE block but instead targeting an existing data element in the data model, the rule defines a new temporary data element of the data model. The syntax is:

RULE NEW <new data model element>

DATATYPE

Integer / Float / String / Enumeration / ObjectType

LONGFORM

LongVersionOfPropertyName

<other sections defined in RULE blocks>

ENDRULE

In every other respect the functionality is the same as RULE. The new data element that is created by RULE NEW in the data model exists only during the evaluation of each transformation in the ruleset and does not exist after the transformed data models are finished being created. One way to think of RULE NEW is as an intermediate variable or calculation. When a single RULE becomes too complex, it might make sense to try to break it down to a RULE and one or more RULE NEW blocks each which has simpler logic and together compute complex values. When defining a data model element with a single RULE that is exceedingly complex, it can beneficial to both the rule author and reviewer to break down rule expressions into separate, intermediate RULE NEW blocks, each with simpler logical steps.

Another common scenario where RULE NEW is appropriate is when the result of a single operation or function is required in several rules, such as the window/wall ratio for each wall. Instead of each RULE of the wall including the logic required to calculate this value for each rule, defining that logic once via a RULE NEW and then referencing the new data element in each RULE that relies on that data greatly reduces duplication of (potentially) complex expressions.

The following table shows when each section of the RULE and RULE NEW is required, optional, or not applicable.

Table X. RULE and RULE NEW Sections

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **RULE** | | | **RULE NEW** | |
| **Section** | *Numeric* | *List* | *Numeric* | | *List* |
| **DESCRIPTION** | Required | Required | Required | | Required |
| **HELP** | Required | Required | Optional | | Optional |
| **REFERENCE** | Optional | Optional | Optional | | Optional |
| **DATATYPE** | Not Applicable | Not Applicable | Required | | Required |
| **SHORTFORM** | Not Applicable | Not Applicable | Optional | | Optional |
| **LONGFORM** | Not Applicable | Not Applicable | Optional | | Optional |
| **INPUTCLASS** | Optional | Optional | Optional | | Optional |
| **MINIMUM** | Optional | Not Applicable | Optional | | Not Applicable |
| **COMMONMINIMUM** | Optional | Not Applicable | Optional | | Not Applicable |
| **COMMONMAXIMUM** | Optional | Not Applicable | Optional | | Not Applicable |
| **MAXIMUM** | Optional | Not Applicable | Optional | | Not Applicable |
| **UNITS** | Required | Not Applicable | Required | | Not Applicable |
| **OPTION** | Not Applicable | Required | Not Applicable | | Required |
| **DEFAULT** | Optional | Optional | Optional | | Optional |

Note: Each transformation is required.

It is expected to be common practice to provide a block of comments for each rule including the approach, name of rule author, revisions, and dates of revisions.

#### TABLE Block

Tables are very commonly used in setting requirements for building energy standards so it is not surprising that they are used extensively when implementing a ruleset on the same topic. The syntax for using the TABLE block is shown below:

TABLE <table-name>

<col-1> <col-2> <col-3>…

<ind/dep-1> <ind/dep-2> <ind/dep-3>

<val-1-1> <val-1-2> <val-1-3>

<val-2-1> <val-2-2> <val-2-3>

<val-3-1> <val-3-2> <val-3-3>

ENDTABLE

Each TABLE can be in the same file as the RULE expression that references it or it can be located in a separate file. When located in the same file as the RULE, the columns are fixed width and are usually used for smaller tables. If it is located in a separate file, it can be a CSV file with the columns separated by commas instead of spaces. Use of a separate CSV file is common for larger tables. When the TABLE is located in a CSV file, the file must be located in the same directory as the files that contain the RULE blocks. The syntax for the TABLE when located in the CSV is the same as when it is located in the file with the RULEs except commas are used to separate the columns.

The first row contains the names of each column. The second row is a designation if the column is INDEPENDENT or DEPENDENT. Columns that are independent are used to lookup values and are provided as arguments for the expression. Columns that are dependent are the resulting values and are returned as the value when used in an expression. The remaining rows are the values for the columns. The following provides a more specific example, although it is fictional.

TABLE RValueTable

ClimateZone Building RValue

INDEPENDENT INDEPENDENT DEPENDENT

1 Residential 5

1 NonResidential 7

2 Residential 6

2 NonResidential 7

3 Residential 9

3 NonResidential 9

4 Residential 3

4 NonResidential 4

ENDTABLE RValueTable

Each table must be complete with every combination of independent values as are shown. An asterisk can be used in any independent column of the table to indicate that any value for that independent value can be used. It is a shortcut that is equivalent to that row being repeated with every possible value for that column. The following table is equivalent to the one shown above since the RValue for ClimateZone 3 is the same for both Residential and NonResidential.

TABLE RValueTable

ClimateZone Building RValue

INDEPENDENT INDEPENDENT DEPENDENT

1 Residential 5

1 NonResidential 7

2 Residential 6

2 NonResidential 7

3 \* 9

4 Residential 3

4 NonResidential 4

ENDTABLE RValueTable

Expressions will be described more in a following section but it is important to understand the syntax of how TABLEs are referenced. The following shows how a table would be referenced within a RULE expression:

RvalueTable:RValue(ClimateZone, u:location, Building, u:classification)

The syntax is similar to how functions are called. The name of the table is followed by a period followed by the dependent variable represent the “function” being called. For tables with multiple dependent variables the function syntax would be different for each dependent variable. The arguments ClimateZone and Building are explicitly assigned to within the parenthesis and can appear in any order. In this example the value of data element u.location is assigned to ClimateZone. All arguments must be assigned or an error condition occurs. If the values passed as arguments do not correspond with items shown in the table, a run time error occurs.

Each TABLE shall be checked by the software during rule compilation to be consistent with each possible set of INDEPENDENT values shown. The checking would be limited to looking at each column of INDEPENDENT values and finding unique values and ensuring that every combination of these unique values exists in the table. No checking outside the individual table would be needed.

#### TRANSFORMATIONS Block

A typical TRANSFORMATIONS block is shown below:

TRANSFORMATIONS

USER,u

SIZING\_PROPOSED,zp

SIZING\_BASELINE,zb

ANNUAL\_PROPOSED,ap

ANNUAL\_BASELINE,ab

ENDTRANSFORMATIONS

This block only needs to occur in a single file and provides identifiers for sections in the RULE and RULE NEW blocks as well as abbreviations used in expressions in RULE and RULE NEW blocks. Each row shows the name of a transformation and abbreviation of a transformation separated by a comma. There should be no limit on the number of transformations possible but for the T24 NACM ruleset, the five transformations shown are expected. At minimum two rows should appear. The first row should always indicate the USER transformation and its abbreviation. The USER transformation is the data model coming into the software from the user. Given the use of the transformation names and abbreviations in RULES, it is likely that the software parsing the ruleset files should first locate the TRANSFORMATION block so that the RULE section names that are transformations can be validated.

#### Expressions

Expressions are used in RULE, RULE NEW, and REPLACE blocks. In RULE and RULE NEW they are used for each transformation as well as MINIMUM, COMMONMINIMUM, COMMONMAXIMUM, MAXIMUM, and DEFAULT sections. They consist of a notation similar to how expressions are used in spreadsheets. The simplest expression would be a simple number such as:

2

But more common expressions usually involve mathematical functions such as the following:

8 + 7

3 \* 24

(2 + 3) \* 4

Expressions can be more than just numbers and can reference other parts of the hierarchical building data representation and that is where they become most useful:

u:building:envelope:window:height \* bz:building:envelope:Multiplier

Unlike a spreadsheet, no equals sign is needed prior to the expression. Expressions can also reference functions such as:

SQRT(bz:building:envelope:wallArea)

More on details on functions appear in the Section X.X Functions.

Expressions can include more complicated logical statements such as:

IF <conditional-expression> THEN

<true-expression>

ELSE

<false-expression>

ENDIF

Both the true-expression and false-expression must be present since each expression needs to return a value. Similarly, an expression can also include a SWITCH statement with similar syntax and functionality as found in other programming languages.

SWITCH <choice-expression>

CASE <choice1>

<expression-value1>

CASE <choice2>

<expression-value2>

CASE <choice3,choice4>

<expression-value3and4>

DEFAULT

<expression-default>

ENDSWITCH

Please note that SWITCH and IF statements can be nested. Also it may be worth considering to replace SWITCH statements with the use of TABLES, since they are often more readable.

String constants can appear over multiple lines with a double quote indicating the beginning of the string on one line and the double quote indicating the end of a string on another line. This is to encourage long and meaningful explanations of data elements in the ruleset. The line breaks are removed when using the string.

#### Referencing Data Model Elements

Expressions will frequently reference data elements from the data model. The method of referencing those data elements is to provide a full path to the data element as shown in previous examples. Each level starting with the root to the leaf element are shown delimited by periods. No relative referencing is available using the Data Model Rule Syntax so that readers will always understand how a reference fits within the entire data model.

u:building:envelope:window:height

Abbreviations of the reference may also be used, if the abbreviations are used throughout the entire reference except for the leaf element. Using abbreviations the previous reference could also be expressed as:

u:bldg:env:win:height

This syntax allows expressions to be shorter.

#### Functions

A number of functions are available for use within expressions. The most common mathematical functions are supported, such as shown in section 5.2.4.2. There are also a large number of functions needed to reference and manipulate the building model, and more still that will be required to perform complex building changes, such as defining daylit areas and positioning daylighting controls that cannot be represented using conventional rules. The scope and descriptions of all building model functions shall be produced by the ruleset development team.

#### Capitalization

The Data Model Rule Syntax is case insensitive so whether something is indicated in capital letters or lowercase, they have the same meaning.

#### Comments

Comments can be indicated as any text on a single line following a double slash “//” symbol.

# Rule Syntax and Functionality Reference

Each compliance rule in a Procedural ruleset is made up of a title and an expression. Each rule expression identifies a building component type and one or more attribute names, followed by the equality operator and a statement which evaluates to the value to be posted to the attribute identified prior to (to the left of) the equality operator. The following provides an example of the most basic procedural compliance rule expression.

Proj:RunYear = { 1991 }

This expression sets the project’s RunYear attribute to the value ‘1991’. Note that the start and end of each rule expression is denoted by opening and closing curly brackets.

Each compliance rule is evaluated in the order defined for each list of rules (rulelist) contained in the ruleset. When each rule is evaluated, it is evaluated for each and every component contained in the building database of the type identified by the first part of the attribute name portion of the rule expression. Therefore, the above rule expression would be evaluated for every component defined in the database of type ‘Proj’.

The following sections will describe each component of a rule expression, starting with the attribute name and continuing on to cover the entire set of function calls and statements compatible with the compliance rules processor engine.

Rule examples throughout the following sections are nicely formatted with indented rows aligning executable statements to best illustrate the use of these rule features. However, there are no requirements with regard to lining up portions of each expression in particular columns, so alignment of this nature has no effect on the evaluation of the rule.

## Identifying the Rule Target

The sample rule expression provided above starts with the character string ‘Proj:RunYear’. This character string is referred to as an attribute name and includes two or more parts, separated by a colon. The first part of every attribute name identifies a building component type, in this case ‘Proj’ which is an abbreviation for the Project building component type. The second part of every attribute name identifies an attribute which describes the type of building component identified in the first part (‘RunYear’).

Most compliance rule expressions begin with two-part attribute names as described above, however attribute names can contain additional parts which reference other building components. For example, the following procedural compliance rule sets the ‘Absorp’ attribute of each wall construction component assigned to each exterior wall (‘ExtWall’) via the ‘Construction’ attribute.

ExtWall:Construction:Absorp = { 0.7 }

When evaluating this expression the rule processor loops over each exterior wall (‘ExtWall’) defined in the building database. If the exterior wall’s ‘Construction’ attribute is defined and references a valid construction component, then that construction component’s ‘Absorp’ attribute is set to the value 0.7. This example extends the normal two-part attribute name into a three-part attribute name. The rule processor supports as many attribute name parts as can be referenced based on the definition of the compliance software’s building data model.

There is one other characteristic which is sometimes included in attribute name definitions, and that is an array index. In the event an attribute is defined by a collection of values (as opposed to just a single value), each attribute name which references that attribute should define which array value the attribute name is actually referencing. The following procedural rule expression provides an example of such a case which sets an occupancy density value (units are in people per 1,000 square feet).

Space:OccDensity[1] = { 20 }

The space attribute ‘OccDensity’ is defined as an array of 4 values. Each of these 4 values can be specifically referenced by including an array index (1-4) within square brackets immediately following the attribute name. You can include array indices on the end of any (not just the last) attribute defined as an array of values, but not following a component type string (the first part of each attribute name).

## Compliance Rule Expression Statements

For Procedural rulesets, everything contained within the starting and ending curly brackets following the equality operator of each rule expression is referred to as the compliance rule expression statement. These statements can contain a wide variety of parts, including constant numeric values or character strings, references to other database attributes, function calls and if..then and/or case/switch statements. The preceding examples all contain statements which are in the form of constant numeric values. The following example includes a statement consisting of a constant character string.

ExtWall:ConstructionType = { "WoodFramingAndOtherWall" }

In Data Model rulesets, the expression itself is unchanged, but there is no leading “= {“ and no trailing “}”. The rule expression includes everything following a model transformation identifier and prior to the next model transform ID or “ENDRULE”, such as:

RULE ExtWall:ConstructionType  
 SIZING\_PROPOSED  
 "WoodFramingAndOtherWall"  
ENDRULE

Where SIZING\_PROPOSED is a model transformation ID and ENDRULE marks the end of the rule.

When using constant character string statements to set database attributes of type string, the only thing you need to remember is to include the entire string on a single line. The expression syntax does not support character strings entry which spans multiple lines in the rules text file

You can also use constant character string statements to set database attributes of type symbol, which is the case with the example above. In addition to the rule about including the entire string on a single line of the ruleset file, you must also remember to define the string to be the exact contents (case sensitive) of one of the enumerations defined in the building database to describe that attribute. In the above example, we could have set ‘ExtWall:ConstructionType’ to one of the strings “MassHeavyWall”, “MassLightWall”, “MassMediumWall”, “MetalBuildingWall”, “MetalFramingWall”, or “WoodFramingAndOtherWall”. Setting ‘ExtWall:ConstructionType’ to any other character string would result in a rule evaluation error.

### Referencing Other Database Attributes

Compliance rule statements can reference attributes which describe the same component which is identified by the first portion of the attribute name portion of the rule expression (to the left of the equality operator). These types of references are called local. Rule statements can also reference a variety of other component’s attributes which are somehow related to the component identified by the first portion of the attribute name. Any rule statement may also reference attributes which define components which are classified as “Global”.

The first and most important thing to understand when referencing other component’s attributes is what component is the local component. For an expression which starts with the attribute name ‘Proj:RunYear’, the local component is the project. The local component is always identified as the leftmost (first) portion of the attribute name. For an expression which starts with the attribute name ‘ExtWall:Construction:Absorp’, the local component is the exterior wall and NOT the construction which is assigned to the exterior wall.

Rule statements reference component attributes via a set of functions. Which function you use to reference a component’s attribute depends on the relationship between the local component and the component whose attribute you are referencing. The following bullets describe each type of component attribute referencing available in the compliance rules processor.

#### Global References

An expression may reference attributes which describe any building component which is classified as global. Global component types include Project, Site and Building. The user is only able to define one of each of the global building components for each compliance project. Referencing global component attributes is performed via the Global() function. The Global() function requires one argument, a two-part attribute name (Component:Attribute). The Global() attribute name argument must include both and only these two parts, with no subordinate component references. The following example rule sets each window’s height to the product of its current height and the building’s ‘WinAreaMult’ attribute:

Window:Height = { Local( Height ) \* Global( Bldg:WinAreaMult )

#### References To All Components of a Specified Type

An expression may generically reference the sum or maximum value of a particular attribute for all components of a specified type currently defined in the building database. This is performed via the SumAll() and MaxAll() functions, each of which require a single two-part attribute name argument. The first part of the attribute name argument identifies the type of building component to sum across, and the second part identifies the attribute to sum. The following example sets the building’s ‘WallArea’ attribute to the sum of all ‘TotExtWallArea’ attributes for every space defined in the building.

Bldg:WallArea = { SumAll( Space:TotExtWallArea ) }

#### Local References

An expression may reference attributes which describe the same component as the one being described by the expression. Referencing local component attributes is performed via the Local() function. The Local() function requires one argument, a one-part attribute name (Attribute only). The Local() attribute name argument need not (and must not) include a component type in the attribute name, since this is defined by the component type provided in the attribute name preceding the equality operator in the rule expression. For example, an expression defining each window’s area can reference that window’s Height and Width attributes as shown in the following sample expression:

Window:Area = { Local( Height ) \* Local( Width ) }

#### Assigned Component References

An expression may reference attributes which describe a component that is assigned to the local component. For example, if a Construction (Cons) component can be attached to an Exterior Wall (ExtWall) component via the exterior wall’s Construction attribute, then expressions which describe an ExtWall can reference attributes which describe the Cons which is assigned to that ExtWall. However, the reverse condition does not apply (i.e. expressions which describe a Cons can not reference attributes which describe an ExtWall which may have that Cons assigned to it). This local referencing functionality is provided via the LocalRef() function, which accepts a single attribute name argument. The LocalRef() function argument must contain multiple parts, but each part must be an attribute name, starting with an attribute of the local component, followed by attribute names of each referenced component. An example of a local reference is an exterior wall’s U-value being set to the U-value of the construction which is assigned to the exterior wall, as shown here:

ExtWall:UValue = { LocalRef( Construction:UValue ) }

#### Parent References

An expression may reference attributes which describe the component which serves as the parent of the local component which is being described by the expression. For example, since every Exterior Wall has a parent Space, then expressions which define attributes for an Exterior Wall may reference attributes which describe that Exterior Wall’s parent Space. Such references are performed via the Parent() function which requires a single attribute name argument. Like the Local() function, the Parent() function’s attribute name argument must only contain a single (attribute name) part, since the component type is already known to be the component type of the currently assigned parent component. Additional functions are available for referencing attributes which describe a component’s grandparent (Parent2()) and great-grandparent (Parent3()). For example, an exterior wall’s height can be set to the height of its parent space by the rule:

ExtWall:Height = { Parent( Height ) }

#### Parental Assigned Component References

An expression may reference attributes which describe a component that is assigned to the parent component of the component being described by the expression. This is performed via the ParentRef() function, whose single argument has the same restrictions as defined for the LocalRef() function. Additional functions are available for referencing attributes which describe a component assigned to the grandparent (Parent2Ref()) and great-grandparent (Parent3Ref()) of a component. In the following example, each thermal zone’s ‘PeakAirFlow’ attribute is set equal to the product of its current ‘PeakAirFlow’ and the ‘FanFactor’ attribute associated with the HVAC sizing rules component which is assigned to that zone’s parent HVAC system (via the ‘DesignRules’ attribute):

Zone:PeakAirFlow = { Local( PeakAirFlow ) \*  
 ParentRef( DesignRules:FanFactor ) }

#### Child Component References

An expression may reference attributes of a single attached child component. This is performed via the ChildRef() function, which requires two arguments. The first argument is an attribute name containing two or more parts, specifying both the child component type and attribute. The second argument is a one-based integer index into the local component’s list of children for the specified component type. For example, a Thermal Zone’s ScheduleType attribute can be set to the same value as it’s first child Space’s ScheduleType using this expression:

Zone:ScheduleType = { ChildRef( Space:ScheduleType, 1 ) }

#### Child Component Attribute Sums & Maximums

An expression may reference the sum or maximum value of a particular attribute for all components which serve as children of the local component. This is performed via the SumChildren() and MaxChild() functions, each of which require a single attribute name argument. The attribute name argument must be two-part, starting with the component type of the children whose attributes are to be referenced, followed by the name of the attribute to be summed/peaked for all children of the type specified by the first part. For example, since every exterior wall can have one or more child windows. The following example uses SumChildren() to set each exterior wall’s ‘TotWinArea’ attribute to the sum of all attached window components’ ‘TotArea’ attributes:

ExtWall:TotWinArea = { SumChildren( Win:TotArea ) }

#### Referencing the Number of Attached Child Components

An expression may reference the number of components (of a particular type) which are attached as children to the local component. This is performed via the ChildCount() function which requires a single component type argument. This argument is equivalent to the first part of most attribute names and identifies the type of child component which will be counted. There are no occurrences of this function in existing compliance rulesets, so the following example was created for your reference. In this example, we set the ‘AvgWinArea’ attribute for each exterior wall to zero if no child windows exist, or to the sum of the wall’s child window areas divided by the number of child windows attached to the wall. Note that the attribute being set in this example ‘AvgWinArea’ is provided for the sole purpose of creating this example and in fact does not exist in the current building database.

ExtWall:AvgWinArea = { if (ChildCount( Win ) > 0) then  
 SumChildren( Win:TotArea ) /  
 ChildCount( Win )  
else 0 endif }

#### Reverse Reference Attribute Sums & Maximums

An expression may reference the sum or maximum value of a particular attribute for all components which reference or assign the local component. This is performed via the SumRevRef() and MaxRevRef() functions, each of which require two attribute name arguments. The first attribute name argument must be two-part, starting with the component type of the component which references or assigns the local component and ending with the attribute used to reference/assign the local component. The second attribute name argument must also be a two-part “component:attribute” name which identifies the attribute of the referencing component to sum or find the peak value of. For example, the SumRevRef() function is required in order to determine the total square footage which each HVAC system serves. In this case, the local component is the HVAC system and the components which we will sum across are all the spaces which are served by the HVAC system. The first SumRevRef() function argument identifies the component:attribute pair which reference the local component (HVAC system), in this case it is Space:HVACSystem. The second SumRevRef() function argument identifies the Space attribute which we will sum, in this case Space:TotArea. Here is how such a rule would be written:

HVACSys:FloorArea = { SumRevRef( Space:HVACSystem, Space:TotArea ) }

#### Assigning Compliance Library Components

An expression may be used to assign a building component defined in the compliance ruleset’s building component library to a database attribute of type object. This is performed via the RuleLibrary() function, which requires two arguments. The first argument identifies the type of component to retrieve from the ruleset’s library. The second argument contains a double-quoted character string consisting of the name of the library component to be retrieved. For example, a heat pump’s ‘CoolEIRCurve’ attribute can be assigned to the performance curve stored in the ruleset’s library by the name “Pckgd Cool EIR” with the expression:

HtPump:CoolEIRCurve = { RuleLibrary( Curve, "Pckgd Cool EIR" ) }

### Referencing Table Look-Up Values

All look-up tables which are defined in the TABLELIST section of the rules file can be referenced by compliance rule statements as if they were pre-defined rule processor functions (similar to those defined in the previous subsection). In order to retrieve a value from a look-up table, the user must provide the look-up table name followed (in parentheses) by values for each independent attribute and a one-based index into the dependent attributes identifying which to return. In the following example, the table OccupAssumpTable was defined as containing one independent attribute and 12 dependent attributes. Therefore, references to this table look-up must contain two arguments, the first defining the independent value to be looked up and the second providing an index (1-12) to identify which dependent table value to return. This example sets the ‘ScheduleType’ attribute for each space to the 1st dependent value contained in the OccupAssumpTable based on the space’s ‘OccType’ attribute.

Space:ScheduleType = { OccupAssumpTable( Local( OccType ), 1 ) }

For another example, lets look at a table look-up designed to set each HVAC system’s ‘’ attribute. The table SystemTypeTable is defined to contain 5 independent attributes and 1 dependent attribute. Therefore, all table look-ups for the SystemTypeTable will contain six arguments, the first five passing in the values for the five independent attributes and the last one to define which dependent attribute to return (in this case, the last argument must be 1). It just so happens that the five independent attributes designed for this table are (1) schedule type (Nonres, Hotel, or Res), (2) a low-rise flag (0 => high rise, 1 => low rise), (3) zone type, (4) heating source and (5) cooling source.

HVACSys:Type = { SystemTypeTable( Local( ScheduleType ),  
 Global( Bldg:LowRise ),  
 Local( ZoneType ),   
 Local( HeatSource ),  
 Local( CoolSource ), 1 ) }

### Executable Expression Statements

The compliance rules processor is compatible with two types of executable expression statements, including if...then...else and case (switch) statements. These statements are used to reference other database values in order to create multiple branches for a single expression. The exact syntax for using these statements must be followed in order to ensure accurate evaluation.

#### if...then...else Statement

The syntax associated with the if...then...else statement is quite simple. Each ‘if’ is directly followed by an expression (which evaluates to TRUE or FALSE) which is in turn followed by a ‘then’ and a statement which is evaluated only if the expression following the ‘if’ evaluates to TRUE. If the user wishes, they may follow this statement by an ‘else’ and another statement which is evaluates only if the expression following the ‘if’ evaluates to FALSE. Finally, each ‘if’ statement must be concluded with an ‘endif’. A simple example of an if...then...else statement is the following rule which sets each window’s ‘SkyArea’ to the product of the window’s ‘Height’ and ‘Width’ attributes if the ‘Tilt’ of it’s parent wall is <= 45.

Win:SkyArea = { if (Parent( Tilt ) <= 45) then   
 Local( Height ) \* Local( Width )  
 else  
 0  
 endif }

The expression following the ‘if’ can result in ANDing and/or ORing multiple expressions together using standard syntax for either FORTRAN or C programming languages (i.e. Use ‘.AND.’ or ‘&&’ to denote AND, and use ‘.OR.’ or ‘||’ to denote OR). The following example sets the ‘SlabPerimArea’ attribute of all underground walls to the parent space’s ‘ExtPerim’ attribute, but only if the underground wall’s ‘Type’ attribute is equal to either 1 or 2.

UndWall:SlabPerimArea = { if ( Local( Type ) == 1 .OR.  
Local( Type ) == 2 ) then  
 Parent( ExtPerim )  
 else  
 0.0  
 endif }

If more than two branches are required, you can nest multiple if...then...else statements as illustrated in the following example.

Furnace:HIR = { if (Local( Configuration ) == 0) then  
 1/(0.005163 \* Local( AFUE ) + 0.4033)  
 else  
 if (Local( AFUE ) <= 83.5) then  
 1/(0.002907 \* Local( AFUE ) + 0.5787)  
 else  
 1/(0.011116 \* Local( AFUE ) - 0.098185)  
 endif  
 endif }

#### case (switch) Statement

The case statement is very useful when you wish to create an expression with several branches based on a single attribute being equal to several individual values. Each case statement begins with the word ‘switch’ followed by an expression which is typically a reference to another building database variable. Following this expression is one or more case statements, each having the form ‘case’ followed by a constant value, followed by a colon, followed by a statement which is evaluated in the event the original expression is equal to the constant value of this case. The last case of the group is the default case and for this case the ‘case’ and following constant value is replaced by a single ‘default’. At the end of the case statement is a ‘endswitch’. Like if...then...else statements, case statements can be nested within each other. A simple example of a case statement is the following rule which sets each space’s ‘MassWeight’ to a specified value (either 30 or 80) based on the value of the space’s ‘Conditioning’ attribute.

Space:MassWeight = { switch ( Local( Conditioning ) )  
 case 0 : 80  
 case 1 : 80  
 case 2 : 80  
 case 3 : 30  
 case 4 : 30  
 default : 80  
 endswitch }

#### Nesting if…then…else and case Statements

Both if...then...else and case statements can be nested together into a single rule expression. This example sets an HVAC system’s ‘FanPowerIdx’ based on its ‘Type’ and ‘FanPowerIdx’ attributes:

HVACSys:FanPowerIdx = {   
 switch Local( Type )  
 case 1: if ( Local( FanPowerIdx ) <= 0.40 ) then  
 0.40  
 else if ( Local( FanPowerIdx ) <= 0.80 ) then  
 Local( FanPowerIdx )  
 else  
 0.80  
 endif endif  
 case 2: if ( Local( FanPowerIdx ) <= 0.40 ) then  
 0.40  
 else if ( Local( FanPowerIdx ) <= 0.80 ) then  
 Local( FanPowerIdx )  
 else  
 0.80  
 endif endif  
 default: 1.25  
 endswitch }

### Expression Syntax and Functionality Tables

The following tables provide a summary of the expression syntax and functionality defined in the preceding sections.

#### Arithmetic and Logical Expression Operators

***Arithmetic:*** \* Multiplication

/ Division

+ Addition

- Subtraction (or Unary Minus)

% Remainder (mod)

\*\* Exponential

***Logical:*** || or .OR. Or

&& or .AND. And

! or .NOT. Not

== or .EQ. Equal

!= or .NE. Not equal

>or .GT. Greater than

<or .LT. Less than

>= or .GE. Greater than or equal to

<= or .LE. Less than or equal to

#### Standard Expression Functions

abs( x ) Absolute value

max( x1, x2 ) Maximum

min( x1, x2 ) Minimum

int( x ) Rounds x to nearest integer

ftoa( x ) Converts a floating point number to a character string

strlen( x ) Number of characters contained in string x

strlower( x ) Returns an all lower case version of string x

strupper( x ) Returns an all upper case version of string x

log( x ) Natural logarithm

log10( x ) Base-10 logarithm

exp( x ) Exponential (e raised to the power x)

pow( x1, x2 ) Power (x1 raised to the power x2)

mod( x1, x2 ) Modulus (remainder) of x1 / x2

sqrt( x ) Square root

sin( x ) Sine (angle expressed in radians)

asin( x ) Arcsine (result in radians)

cos( x ) Cosine (angle expressed in radians)

acos( x ) Arccosine (result in radians)

tan( x ) Tangent (angle expressed in radians)

atan( x ) Arctangent (result in radians)

*Note:* All function names are case insensitive.

#### Expression Comment Syntax

**//** Single line comment, ignores all characters until the end of line encountered

**/\*** Single or multiline comment, ending with the end comment symbol **\*/**

; Single line comment, ignores all characters until the end of line encountered

#### Executable Expression Statements

**if** (*expression*) **then**

*statement*

**else if** (*expression*) **then**

*statement*

**else**

*statement*

**endif**

**endif**

*Note:* Each if statement must contain an else.

**switch** (*expression*)

**case** (*const*) : *statement*

**case** (*const*) : *statement*

**default** : *statement*

**endswitch**

*Note:* Each switch statement must contain a default.

*Note:* The individual case values listed as *const* must be either numeric constants or   
 enumerations which can be directly translated into numeric constants at parse-time  
 in order to prevent excessive evaluation-time error checking and minimize the  
 complexity of the statement.

*Notes*: All reserved words in bold (if, else, endif, ...) are case insensitive.

If and Switch statements can be nested as long as the expression evaluates to a single return value.

### Format Specifications

Several of the expression functions allow for the formatting of character string and numeric data into messages and strings output to files or database properties. These functions include Format(), PostError() and PostWarning().

Format specifications always begin with a percent sign (%). When the string/message argument of the above listed functions includes a single format specification, it converts the value of the first argument following the string/message and outputs it accordingly. A second format specification causes the second argument to be converted and output, and so on. A maximum of 7 format specifications and subsequent arguments can be included in a single function call and if there are more arguments than there are format specifications, the extra arguments are ignored.

Format specifications are written as “%[flags][width][.precision]type”, where the portions enclosed in square brackets are optional. The following table describes supported options for each component of the format specification fields.

#### Format Specification Fields

***flags:*** Optional character(s) that control justification of output and printing of signs, blanks and decimal points. More than one flag can appear in a single format specification.

- Left align the result within the given field width (default is right aligned).

+ Prefix the output value with a sign (+ or –) (default is only show sign if value is negative).

0 If width is prefixed with 0, zeros are added until the minimum width is reached. If 0 and – appear,  
 the 0 is ignored.

‘ ’ (blank space) Prefix the output value with a blank if the output value is positive; the blank is ignored  
 if both the ‘ ’ and + flags appear

***width:*** Optional number that specifies the minimum number of characters output. This value can only serve to increase the width of an output string and will not truncate the value.

***precision:*** Optional number that specifies the maximum number of characters printed for all or part of the output field.

for type = e, E or f Specifies the number of digits after the decimal point.

for type = g or G Specifies the maximum number of significant digits to output.

for type = s or S Specifies the maximum number of characters to be output.

***type:*** Required character that determines whether the associated argument is interpreted as a string or a number.

e Signed value having the form [ – ]d.dddd e [sign]ddd where d is a single digit, dddd is one or  
 more digits, ddd is exactly three digits, and sign is + or –.

E Identical to the e format (above) except that E (as opposed to e) precedes the exponent.

f Signed value having the form [ – ]dddd.dddd, where dddd is one or more digits. The number of  
 digits before the decimal point depends on the magnitude of the number, and the number of  
 digits after the decimal point depends on the requested precision.

g Signed value printed in f or e format, whichever is more compact for the given value and  
 precision. The e format is used only when the exponent of the value is less than –4 or greater  
 than or equal to the precision argument. Trailing zeros are truncated, and the decimal point  
 appears only if one or more digits follow it.

G Identical to the g format (above) except that E (as opposed to e) precedes the exponent (when  
 appropriate).

s Character string of length not to exceed precision.

### Ruleset Function Reference

**Global( comp:param*[i]* )** or **#G( comp:param*[i]* )**Returns the *i*th value of the parameter (param) describing global component (comp)

For example, in order to scale up and/or down the overall window area of a building, a building-wide window area multiplier is calculated (Bldg:WinAreaMult) and then applied to each window in the building via the rule:

Window:Height = { Local( Height ) \* Global( Bldg:WinAreaMult ) }

**GlobalStatus( comp:param*[i]* )** or **#GST( comp:param*[i]* )**Returns the status of the property corresponding to the function argument (comp:param*[i]*), as described in the Global() function description above.  
The following are valid return values for this (and all other) property status retrieval functions:  
 0 Undefined  
 1 Program Default (defaulted within program source code)  
 2 Ruleset Default (defaulted via symbol definitions within the ruleset)  
 3 Ruleset Library (value was included in ruleset library component definition)  
 4 Ruleset Defined (defaulted via rules in the ruleset)  
 5 User Default (defaulted based on data in user-defined default file)  
 6 User Library (value was included in user library component definition)  
 7 User Defined (value was specified by the user)  
 8 Simulation Result (value imported from the most recent simulation output)

**GlobalSymbolString( comp:param*[i]* )** or **#GSS( comp:param*[i]* )**Returns the symbolic string corresponding to the global component property identified by the function argument (comp:param*[i]*), as described in the Global() function description above. Each symbolic property within the database has associated with it a list of symbols. Each item in a symbol list includes a character string (typically selected within a drop-down list control, or combobox) and a numeric value. When retrieving data for symbolic properties, the Global() function returns only the numeric value associated with the currently selected symbol. The GlobalSymbolString() function was therefore implemented in order to provide access to the character string associated with the currently selected symbol.

**Local( param*[i]* )** or **#L( param*[i]* )**Returns the *i*th value of the local parameter (param)

For example, this rule which calculates the area of each exterior wall does so by multiplying the each wall’s width and height, like so:

ExtWall:Area = { Local( Height ) \* Local( Width ) }

**LocalSymbolString( param*[i]* )** or **#LSS( param*[i]* )**Returns the symbolic string corresponding to the local parameter identified by the function argument (param*[i]*), as described in the Local() function description above. Each symbolic property within the database has associated with it a list of symbols. Each item in a symbol list includes a character string (typically selected within a drop-down list control, or combobox) and a numeric value. When retrieving data for symbolic properties, the Local() function returns only the numeric value associated with the currently selected symbol. The LocalSymbolString() function was therefore implemented in order to provide access to the character string associated with the currently selected symbol.

**LocalStatus( param*[i]* )** or **#LST( param*[i]* )**Returns the status of the property corresponding to the local parameter identified by the function argument (param*[i]*), as described in the Local() function description above.  
Refer to the description of the GlobalStatus() function (above) for a listing of valid return values for this function.

3.2.6 Ruleset Function Reference (continued)

**LocalCompAssigned( param*[i]* )** or **#LCA( param*[i]* )**Returns the value 1 if the *i*th value of the local parameter (param) has a component assigned to it. Otherwise, this function returns the value 0.

For example, this rule calculates the BrakeHP of a HVAC system’s return fan after first confirming that a return fan is assigned to the system:

HVACSys:ReturnFan:BrakeHP = { if (LocalCompAssigned( ReturnFan ) == 1) then  
 LocalRef( ReturnFan:AirFlow ) \*  
 LocalRef( ReturnFan:FanKW ) \*  
 LocalRef( ReturnFan:MotorEff ) / 0.746  
 else 0.0 endif }

**LocalIsDefault( param*[i]* )** or **#LID( param*[i]* )**Returns the value 1 if the *i*th value of the local parameter (param) has been set to the reserved keyword “DEFAULT”. Otherwise, this function returns the value 0.

For example, this rule calculates the EERb of each heat pump if and only if the heat pump’s CAPa is not set to “DEFAULT”:

HtPump:EERb = { if (LocalIsDefault( CAPa )) then  
 DEFAULT  
 else Local( SEER ) / (1 - 0.5 \* Local( Cd ))  
 endif }

**LocalRef( param1*[i]*:param2*[i]:param3[i]...* )** or **#LR( param1*[i]*:param2*[i]:param3[i]...* )**Returns the *i*th value of the parameter (param*N*) for the component referenced by the last param argument listed. Each parameter listed must be a component assignment except the last one. Each subsequent parameter is an element describing the component assigned by the previously encountered parameter. There will always be at least two parameters listed, and more may be included to indicate additional levels of component referencing.

For example, the following rule calculates the total wattage across all fixtures assigned to each lighting system:

LtSys:TotalWatts = { ( LocalRef( Fixture[1]:TotalWatts ) \*  
 Local( NumFixtures[1] ) ) +  
 ( LocalRef( Fixture[2]:TotalWatts ) \*  
 Local( NumFixtures[2] ) ) +  
 ( LocalRef( Fixture[3]:TotalWatts ) \*  
 Local( NumFixtures[3] ) ) +  
 ( LocalRef( Fixture[4]:TotalWatts ) \*  
 Local( NumFixtures[4] ) ) }

**Parent( param*[i]* )** or **#P( param*[i]* )**Returns the *i*th value of the parent’s parameter (param)

For example, this rule which calculates the fixture density for a lighting system by dividing the area of its parent Space by the number of fixtures, like so:

LtSys:SqFtPerFixture = { Parent( Area ) / Local( NumFixtures ) }

**Parent2( param*[i]* )** or **#P2( param*[i]* )**Returns the *i*th value of the grandparent’s parameter (param)

This function acts just like the Parent() function above, except that it references parameters which describe the grandparent, rather than parent, of the local component.

3.2.6 Ruleset Function Reference (continued)

**Parent3( param*[i]* )** or **#P3( param*[i]* )**Returns the *i*th value of the great-grandparent’s parameter (param)

This function acts just like the Parent() function above, except that it references parameters which describe the great-grandparent, rather than parent, of the local component.

**ParentCompAssigned( param*[i]* )** or **#PCA( param*[i]* )**Returns the value 1 if the *i*th value of the parent’s parameter (param) has a component assigned to it. Otherwise, this function returns the value 0.

**ParentStatus( param*[i]* )** or **#PST( param*[i]* )**Returns the status of the property corresponding to the parent component’s parameter identified by the function argument (param*[i]*), as described in the Parent() function above.  
Refer to the description of the GlobalStatus() function (above) for a listing of valid return values for this function.

**ParentIsDefault( param*[i]* )** or **#PID( param*[i]* )**Returns the value 1 if the *i*th value of the parent’s parameter (param) has been set to the reserved keyword “DEFAULT”. Otherwise, this function returns the value 0.

**ParentRef( param1*[i]*:param2*[i]:param3[i]...* )** or **#PR( param1*[i]*:param2*[i]:param3[i]...* )**Exactly the same as LocalRef() described above with the exception that param1 is a parameter which describes the current component’s parent component.

**Parent2Ref( param1*[i]*:param2*[i]:param3[i]...* )** or **#PR2( param1*[i]*:param2*[i]:param3[i]...* )**Exactly the same as ParentRef() described above with the exception that param1 is a parameter which describes the current component’s grandparent component.

**Parent3Ref( param1*[i]*:param2*[i]:param3[i]...* )** or **#PR3( param1*[i]*:param2*[i]:param3[i]...* )**Exactly the same as ParentRef() described above with the exception that param1 is a parameter which describes the current component’s great-grandparent component.

**ChildRef( comp:param1*[i]*:param2*[i]:param3[i]...*, n )** or **#CR( comp:param1*[i]*:param2*[i]:param3[i]...*, n )**Exactly the same as LocalRef() described above with the exception that param1 is a parameter which describes the (n)th child component of type (comp).

**SumChildren( comp:param*[i]* )** or **#SC( comp:param*[i]* )**Returns the sum across all children of type (comp) of the *i*th value of the child’s parameter (param). All children, and children of those children are searched for all components of type “comp”.

For example, the following rule sums the areas of all child windows for each exterior wall:

ExtWall:TotWinArea = { SumChildren( Win:TotArea ) }

**SumAll( comp:param*[i]* )** or **#SA( comp:param*[i]* )**Returns the sum across all building components of type (comp) of the *i*th value of the component’s parameter (param) regardless of any parent/child or reference attachments.

For example, the following rule creates a building-wide sum of all exterior wall area by summing the TotExtWallArea parameter of each Space in the project:

Bldg:ExtWallArea = { SumAll( Space:TotExtWallArea ) }

3.2.6 Ruleset Function Reference (continued)

**SumRevRef( comp:param1*[i]*, comp:param2*[i]* )** or  
**#SR( comp:param1*[i]*, comp:param2*[i]* )**Returns the sum of param2[i] values for all components which reference the local component via the comp:param1[i] parameter. This function was defined primarily to allow HVAC systems to reference parameters that describe the Spaces which assign each system, since no parent/child relationship exists between the HVAC systems and the Spaces they serve.

For example, the following rule calculates the total floor area that each HVAC system serves by summing the area of all Spaces which reference each system:

HVACSys:FloorArea = { SumRevRef( Space:HVACSystem, Space:TotArea ) }

**MaxChild( comp:param*[i]* )** or **#MC( comp:param*[i]* )**

**MinChild( comp:param*[i]* ) or #MIC( comp:param*[i]* )**Returns the maximum or minimum value across all children of type (comp) of the *i*th value of the child’s parameter (param). All children, and children of those children are searched for all components of type “comp”.

For example, the following rule calculates the largest schedule type ID across all child Activity Areas for each Space:

Space:SchedTypeNum = { MaxChild( ActArea:SchedTypeNum ) }

**MaxAll( comp:param*[i]* )** or **#MA( comp:param*[i]* )**

**MinAll( comp:param*[i]* )** or **#MIA( comp:param*[i]* )**Returns the maximum or minimum value across all building components of type (comp) of the *i*th value of the component’s parameter (param) regardless of any parent/child or reference attachments.

**MaxRevRef( comp:param1*[i]*, comp:param2*[i]* )** or  
**#MR( comp:param1*[i]*, comp:param2*[i]* )**

**MinRevRef( comp:param1*[i]*, comp:param2*[i]* )** or  
**#MIR( comp:param1*[i]*, comp:param2*[i]* )**Returns the maximum or minimum value of param2[i] values for all components which reference the local component via the comp:param1[i] parameter.

For example, the following rule calculates the largest schedule type ID across all Spaces which assign each HVAC system:

HVACSys:SchedTypeNum = { MaxRevRef( Space:HVACSystem, Space:SchedTypeNum ) }

*Note:* Each of the above listed functions will automatically return a value of the same type as the final parameter which it references. In the event that the expression is to evaluate to a symbol, the return value shall be an integer value.

*Note:* For all functions mentioned above, the argument *i* is a one-based index and must be included when calling these functions only when the number of values which describe the parameter (param) is greater than 1 (i.e. an array or list of data).

*Note:* All *i* arguments must be in the form of a numeric constant (not an expression).

3.2.6 Ruleset Function Reference (continued)

**MaxAllComp( comp:param*[i]* )** or **#MAC( comp:param*[i]* )**This function is identical to the MaxAll() function described above, with the exception that the value (actually a character string) returned consists of the name of the component for which the maximum comp:param*[i]* value was found.

**MaxRevRefComp( comp:param1*[i]*, comp:param2*[i]* )** or **#MRC( comp:param1*[i]*, comp:param2*[i]* )**This function is identical to the MaxRevRef() function described above, with the exception that the value (actually a character string) returned consists of the name of the component for which the maximum value was found.

**CountRefs( comp:param1*[i]*, *MaxArrayIndex* )** or **#CRS( comp:param1*[i]*, *MaxArrayIndex* )**Returns the number of components which reference the local component via the comp:param1[i] parameter.  
If comp:param1 is defined as an array of component assignments and the array index for this argument is either not specified or [1], then the number of array elements checked for references will be the 1-based array index specified by the MaxArrayIndex argument, or the entire array (if the MaxArrayIndex argument is not specified).

**CountNoRefs( comp:param1*[i]*, *comp:param2[i]*, *comp:param3[i]* )** or **#CNRS( comp:param1*[i]*, *comp:param2[i]*, *comp:param3[i]* )**Very similar to the CountRefs() function above, although the primary function here is to count the lack of references to any component. The first (comp:param1[i]) argument is interpreted and used exactly as with CountRefs() function, in that it describes the property which we will scan for component assignments. If comp:param2[i] is specified, then it represents the largest argument 1 array index to scan for missing assignments.  
The return value from this function is 0 if all component assignments (based on arguments 1 & 2) exist and are valid. If one or more missing or invalid assignments are found, then the return value will be the value associated with comp:param3[i] (if specified), or 1 (if comp:param3[i] is not specified).

**CountUniqueParentRefs( comp:param1*[i]*, *comp:param2[i]* )** or **#CUPRS( comp:param1*[i]*, *comp:param2[i]* )**The meaning and usage of arguments in this function are identical to that of the CountRef() function above, with one important difference. For this function, not all component references are counted, but rather only those that have a unique parent component.

**ChildCount( comp )** or **#CC( comp )**Returns the number of children of type (comp) attached to the building component.

**ChildIndex( *bOnlyComponentsOfSameType* )** or **#CHI( *bOnlyComponentsOfSameType* )**Returns the index (1-N) associated with the object on which the rule is evaluated in relation to all other child components of its parent component. If specified, the boolean (0/1, 0 being the default) *bOnlyComponentsOfSameType* argument determines whether or not the index returned will be in relation to only those children of the component type for which the rule is being evaluated (value of 1) versus all child components of the parent (value of 0).

3.2.6 Ruleset Function Reference (continued)

**CreateChildren( comp, num children, “Name Prefix”*, “Rulelist Name”, EvalOption* )**or **#CCH( comp, num children, “Name Prefix”*, “Rulelist Name”, EvalOption* )**Causes the creation of one or more (num children) components of the type “comp” where each new component’s parent is the primary component for which the rule is being evaluated (the leftmost component in the left side of the rule expression). The “Name Prefix” argument forms the beginning of each new child component’s name, followed by an index which ensures that the component’s name is unique. The optional “Rulelist Name” argument names a rulelist that can be evaluated immediately following the successful creation of the child component(s). The final (and also optional) argument, EvalOption, is a numeric value of either 0 or 1. An EvalOption value of 1 is the default and causes the rulelist to be evaluated only on the newly created child components. Alternatively, an EvalOption value of 0 causes the rulelist to be evaluated for all components in the building description.

For example, the following rule causes a single child window to be created for each exterior wall that is a child of a conditioned or indirectly conditioned space:

ExtWall:Action = { if (Parent( Conditioning ) < 4) then  
 CreateChildren( Win, 1, "BudgetWin" )  
 else 0 Endif }

**DeleteChildren( comp, *“Rulelist Name”, EvalOption* )**or **#DCH( comp, *“Rulelist Name”, EvalOption* )**Causes the deletion of all child components of type “comp” for each primary (parent) component for which the rule is being evaluated (the leftmost component in the left side of the rule expression). The optional “Rulelist Name” argument names a rulelist that can be evaluated immediately following the successful deletion of the child component(s). The final (and also optional) argument, EvalOption, is a numeric value of either 0 or 1. An EvalOption value of 1 is the default and causes the rulelist to be evaluated only on the parent component. Alternatively, an EvalOption value of 0 causes the rulelist to be evaluated for all components in the building description.

For example, the following rule causes all child windows to be deleted for each exterior wall that is a child of a conditioned or indirectly conditioned space:

ExtWall:Action = { if (Parent( Conditioning ) < 4) then  
 DeleteChildren( Win )  
 else 0 Endif }

**CreateComp( comp, “Name Prefix”*, “Rulelist Name”, EvalOption* )**or **#CCO( comp, “Name Prefix”*, “Rulelist Name”, EvalOption* )**Causes the creation of a building component of the type “comp” and the return value is a reference to the new component which can be used to assign that component to an existing building component. The “Name Prefix” argument forms the beginning of the new component’s name, followed by an index that ensures the new component name’s uniqueness. The optional “Rulelist Name” argument names a rulelist that can be evaluated immediately following the successful creation of the new component. The final (and also optional) argument, EvalOption, is a numeric value of either 0 or 1. An EvalOption value of 1 is the default and causes the rulelist to be evaluated only on the newly created component. Alternatively, an EvalOption value of 0 causes the rulelist to be evaluated for all components in the building description.

3.2.6 Ruleset Function Reference (continued)

**CompExists( comp, “Comp Name” )** or **#CE( comp, “Comp Name” )**This function checks for the existence of a building component of the type “comp” and name “CompName”. This function returns 1 if the component exists, otherwise 0.

**CompName( comp, iCompIndex )** or **#CN( comp, iCompIndex )**Returns the name of the (iCompIndex)th building component of the type “comp”. The iCompIndex argument must be an integer value greater than 0.

**ComponentIndex()** or **#COI()**Returns the index (1-N) of the building component for which the rule is being evaluated in relation to all other building components of the same type.

**ComponentCount( comp )** or **#CMPC( comp )**Returns the total number of building components of the type “comp” currently defined in the database.

**UniqueComponentName( “Name Prefix” )** or **#UCN( “Name Prefix” )**This function is used to find a unique component name within the database. If no component in the database exists by the name “Name Prefix”, then this same argument character string is returned by the function. If there is a building component by this name already defined, then a trailing “-#” is appended to the name prefix and returned, where # is the lowest integer value greater than 0 for which no building component of that name exists.

**DeleteComp( *“Rulelist Name”, EvalOption* )**or **#DCO( *“Rulelist Name”, EvalOption* )**Causes the deletion of a component currently assigned to the property identified by the left side of the rule expression. The optional “Rulelist Name” argument names a rulelist that can be evaluated immediately following the successful deletion of the assigned component. The second (and also optional) argument, EvalOption, is a numeric value of either 0 or 1. An EvalOption value of 1 is the default and causes the rulelist to be evaluated only on the primary component for which the rule is being evaluated (the leftmost component in the left side of the rule expression). Alternatively, an EvalOption value of 0 causes the rulelist to be evaluated for all components in the building description.

For example, the following rule causes any air economizer of type “None” (0) assigned to an HVAC system to be deleted:

HVACSys:AirEcon = { if (LocalRef( AirEcon:Type ) == 0) then  
 DeleteComp()  
 else Local( AirEcon )  
 Endif }

**DeleteAllComps( comp, *“Rulelist Name”, EvalOption* )**or **#DAC( comp, *“Rulelist Name”, EvalOption* )**Causes the deletion of all components of type “comp” that currently exist in the building description. The optional “Rulelist Name” argument names a rulelist that can be evaluated immediately following the successful deletion of the components. The second (and also optional) argument, EvalOption, is a numeric value of either 0 or 1. An EvalOption value of 0 is the default and causes the rulelist to be evaluated for all components in the building description. Alternatively, an EvalOption value of 1 causes the rulelist to be evaluated only on the primary component for which the rule is being evaluated (the leftmost component in the left side of the rule expression).

3.2.6 Ruleset Function Reference (continued)

**EnsureSymbolExists()** or **#ESE()**Returns the numeric value corresponding to the current symbol selection if that selection is valid, otherwise it returns the default symbol value. This function helps to ensure that certain symbolic parameters are always assigned to valid symbols.

**RuleLibrary( comp, “Comp Name” )** or **#RL( comp, “Comp Name” )**Returns a reference to the component of type (comp) which is named “Comp Name”.

For example, the following rule defaults each window’s glass type to the minimally-compliant glass type contained in the ruleset library:

Win:GlassType = { RuleLibrary( Glass, “Minimally-Compliant Window Glass” ) }

**EvalRulelist( “Rulelist Name”, *EvalOption* )** or **#ER(“Rulelist Name”, *EvalOption* )**Causes the compliance rulelist by the name “Rulelist Name” to be evaluated. The second EvalOption argument is optional. An EvalOption value of 0 (the default) causes the rulelist to be evaluated for all components in the building description. An EvalOption value of 1 causes the rulelist to be evaluated only on the primary component for which the rule is being evaluated.

For example, the following rule causes the rulelist named “EnvelopeConstruction” to be evaluated:

Proj:Action = { EvalRulelist( "EnvelopeConstruction" ) }

In the following example, the rulelist named “” is evaluated only in the event an economizer is already assigned to the HVAC system. By specifying an EvalOption argument of 1, only rules where the left side begins “HVACSys:…” will be evaluated, and only for the HVACSys component from which the rulelist was being called:

HVACSys:AirEcon = {  
 if (LocalCompAssigned( AirEcon ) == 0) then  
 CreateComp( Econ, "BudgetAirEcon", "BudgetEconRules" )  
 else EvalRuleList( "BudgetEconRules", 1 )  
 endif }

**WizardZoning( iZoningType, bForceZoning )** or **#WZ( iZoningType, bForceZoning )**This function causes building footprint and thermal zone defaulting source code in the BDBase32.dll library to be executed and returns the value 1 if the code evaluated successfully, or 0 if errors occurred (in which case error messages are written to the ruleset evaluation log file). The second, bForceZoning, boolean (0/1) argument indicates whether or not all footprint and zoning is to be re-evaluated (if this argument is a non-zero value) versus only performing zoning associated w/ newly modified data (if bForceZoning equals 0).  
Values associated with the first (iZoningType) argument are as follows:  
 1 Refresh footprint and thermal zoning based on data stored in database  
 2 Position skylights based on data stored in database  
 3 Calculate daylighting fractions and position daylighting controls  
 4 Calculate the most general set of 3-5 primary facing building orientations  
 5 Refresh building footprint properties only

**CurrentTime()** or **#CT()**Returns an integer value corresponding to the time when the rule is evaluated, according to the computer’s system clock. The actual value represents the number of seconds that has elapsed since midnight, January 1, 1970.

For example, the following rule, when placed in the budget conversion rulelist, sets the exact time in which the proposed building description is being converted into the budget building:

Proj:BudgConvTime = { CurrentTime() }

3.2.6 Ruleset Function Reference (continued)

**CurrentYear()** or **#CY()**Returns an integer value corresponding to the year when the rule is evaluated, according to the computer’s system clock.

**Date( year, month, day )** or **#D( year, month, day )**Returns a integer value corresponding to the number of seconds that has elapsed between midnight, January 1, 1970 and midnight the day identified by the function arguments. The year argument is a four-digit year (i.e. 1999), the month argument is an integer value between 1 and 31 and the day argument is an integer value between 1 and the number of days in the month identified by the month argument.

For example, the following rule sets a schedule’s start date to January 1 of the analysis year:

Sched:StartDate = { Date( Global( Proj:AnalysisYear ), 1, 1 ) }

**NumDaysInMoDaRange( iStartMonthDay, iEndMonthDay )** or **#MDND( iStartMonthDay, iEndMonthDay )**Returns the number of days spanned by the two month/day arguments, each of which are equivalent to ((Month Index (1-12) \* 100) + Day Index (1-31)). The start and end month/day arguments must be in the range 101 thru 1231, for example 101 represents January 1st, 306 represents March 6th and 1130 represents November 30th.  
The end month/day argument can be a value less than the start month/day argument, in which case the number of days spanning the two is assumed to cross a year boundary.  
February is modeled as having 28 days, regardless of whether or not the simulation year is a leap year (due to limitations of the DOE-2.1E simulation engine).

**MonthDayShift( iStartMonthDay, iNumDaysToShift, iYear )** or **#MDS( iStartMonthDay, iNumDaysToShift, iYear )**Returns the month/day value associated with the first (iStartMonthDay) argument shifted by value of the second (iNumDaysToShift) number of days argument for the year specified as the third and final argument. Both the first argument (iStartMonthDay) and return value (only if iYear is equal to 0) are numbers that describe a date equivalent to ((Month Index (1-12) \* 100) + Day Index (1-31)). All month/day values must be in the range 101 thru 1231, for example 101 represents January 1st, 306 represents March 6th and 1130 represents November 30th.  
The second argument value, iNumDaysToShift, can be either positive or negative, but cannot cause the return value to wrap from the selected year into the subsequent or previous years.  
If the third argument (iYear) is equal to 0, then the return value will be a month/day value (as described above). If the iYear argument is greater than 0, then it is assumed to be a four-digit year in which case the return value corresponds to the number of seconds that has elapsed between midnight, January 1, 1970 and midnight on the day resulting from the start month/day shifted by iNumDaysToShift.

3.2.6 Ruleset Function Reference (continued)

**Format( “string w/ format specs”*, var1, var2, … varN* )** or **#FMT( “string w/ format specs”*, var1, var2, … varN* )**Returns a character string based on the specified function arguments. The “string w/ format specs” function argument consists of the character string to be returned, including format specification fields that enable the reporting of string or numeric data referenced by optional arguments var1-N. Refer to the preceding [Table 3-5](#_Format_Specifications) describing compatible format specification fields for more information.

**FltToStr( Value, DecimalPrecision, InclCommas, ZeroFillLeftToLength )** or **#F2S( Value, DecimalPrecision, InclCommas, ZeroFillLeftToLength )**This function provides customized formatting of floating point values, including commas (based on location settings stored in the Windows operating system) and a specified decimal precision. The return value is a character string containing the formatted number.  
The Value argument consists of the number to be represented in the returned character string. The DecimalPrecision argument is an integer value specifying the decimal precision for the formatted number. The InclCommas argument is a boolean (0/1) value indicating whether or not to include commas (or other digit grouping symbol specified in the Windows operating system’s regional/language settings) in the resulting character string. The ZeroFillToLength argument is an integer value that defines a minimum length for the resulting character string, causing leading zeroes to be added to the string until the minimum length is achieved.

**EnsureStringUniqueness( comp:param*[i], MaxStringLength* )** or  
**#ESU( comp:param*[i], MaxStringLength* )**The purpose of this function is to scan through all occurrences of building components of type “comp”, to ensure that the character strings stored in each param*[i]* property are unique. If/when non-unique strings are encountered, they are trimmed in length (as required based on the optional second MaxStringLength argument) and then appended w/ numeric values beginning w/ “ 2” until a unique string is found.  
The return value is the number of comp:param*[i]* character strings that were modified to ensure uniqueness.

**StoreBDBase( “File Ext”, InputOnlyOption )** or **#SB( “File Ext”, InputOnlyOption )**This function is provided to assist in ruleset debugging efforts and causes the building database to be written to a file. The first argument consists of the file extension you would like the output database file to use. The file directory and name will be consistent with the currently active project file. The second option is a numeric flag value 0 or 1, where 1 indicates you would like only user inputs to be written to the database file and 0 causes all values to be written.  
For example, lets say you have a rule that you suspect is not modifying the database as you designed it to. Simply include two separate calls to this function. One immediately prior to and one following the rule in question, making sure to provide different “File Ext” arguments for each. By comparing the two output database files you can determine exactly how the rule in question is modifying the database. Here is an example of such a scenario:

Proj:Action = { StoreBDBase( "DBG1", 0 ) }  
 < -- rule in question goes here -- >  
 Proj:Action = { StoreBDBase( "DBG2", 0 ) }

3.2.6 Ruleset Function Reference (continued)

**PostError( “string w/ format specs”*, var1, var2, … varN* )** or **#PE( “string w/ format specs”*, var1, var2, … varN* )**This function is identical to the Format() function described above in terms of both argument descriptions and return value (character string). The only difference lies in the fact that the character string returned by this function is also posted to the log file as an error message.

**PostWarning( “string w/ format specs”*, var1, var2, … varN* )** or **#PW( “string w/ format specs”*, var1, var2, … varN* )**This function is identical to the Format() function described above in terms of both argument descriptions and return value (character string). The only difference lies in the fact that the character string returned by this function is also posted to the log file as a warning message.

*Note:* All function names are case insensitive.