Cache NodeBrain Module

Release 0.9.03

Cache NodeBrain Module
December 2014
NodeBrain Open Source Project

Release 0.9.03

Author: Ed Trettevik

MIT License

NodeBrain License

History

2005-10-12 Title: *NodeBrain Tutorial*
Author: Ed Trettevik <eat@nodebrain.org>
Publisher: NodeBrain Open Source Project

2014-12-14 Release 0.9.03

- Included node kids, rows, and hits node functions.

Preface

This tutorial is intended for readers seeking an introduction to NodeBrain through a series of simple examples. Other documents are available for readers looking for a more complete reference to the rule language, modules, or API (application programmatic interface).

The intent of the examples in this tutorial is to illustrate individual concepts, not to provide complete working applications or show all related options. We avoid formal syntax descriptions, thinking you are here because you want to figure it out from examples.

Files referenced in this tutorial are included in the tutorial directory of the NodeBrain distribution.

See www.nodebrain.org for more information and the latest update to this document.

Documents

NodeBrain Guide - Information on using `nb`
NodeBrain Tutorial - A gentle introduction to `nb` and the rule language
NodeBrain Language - Rule language syntax and semantics
NodeBrain Library - C API

Document Conventions

Sample code and input/output examples are displayed in a monospace font, indented in HTML and Info, and enclosed in a box in PDF or printed copies. Bold text is used to bring the reader's attention to specific portions of an example. In the following example, the first and last line are associated with the host shell and the lines in between are input or output unique to NodeBrain. The `define` command is highlighted, indicating it is the focus of the example. Lines ending with a backslash \ indicate when a command is continued on the next displayed line. This is supported by the language within source files, but not for other methods of command input. If you copy an example of a command displayed over multiple lines, you must enter it as a single line when used outside the context of a source file.

```
$ nb
> define myFirstRule on(a=1 and b=2) mood="happy";
> assert mood="sad";
> show mood
mood = "sad"
> assert a=1,b=2,c=3,d="This is an example of a long single line that",\
    e="we depict on multiple lines to fit on the documnet page";
2008/06/05 12:09:08 NB000I Rule myFirstRule fired(mood="happy")
> show mood
mood = "happy"
> quit
$
```

Table of Contents

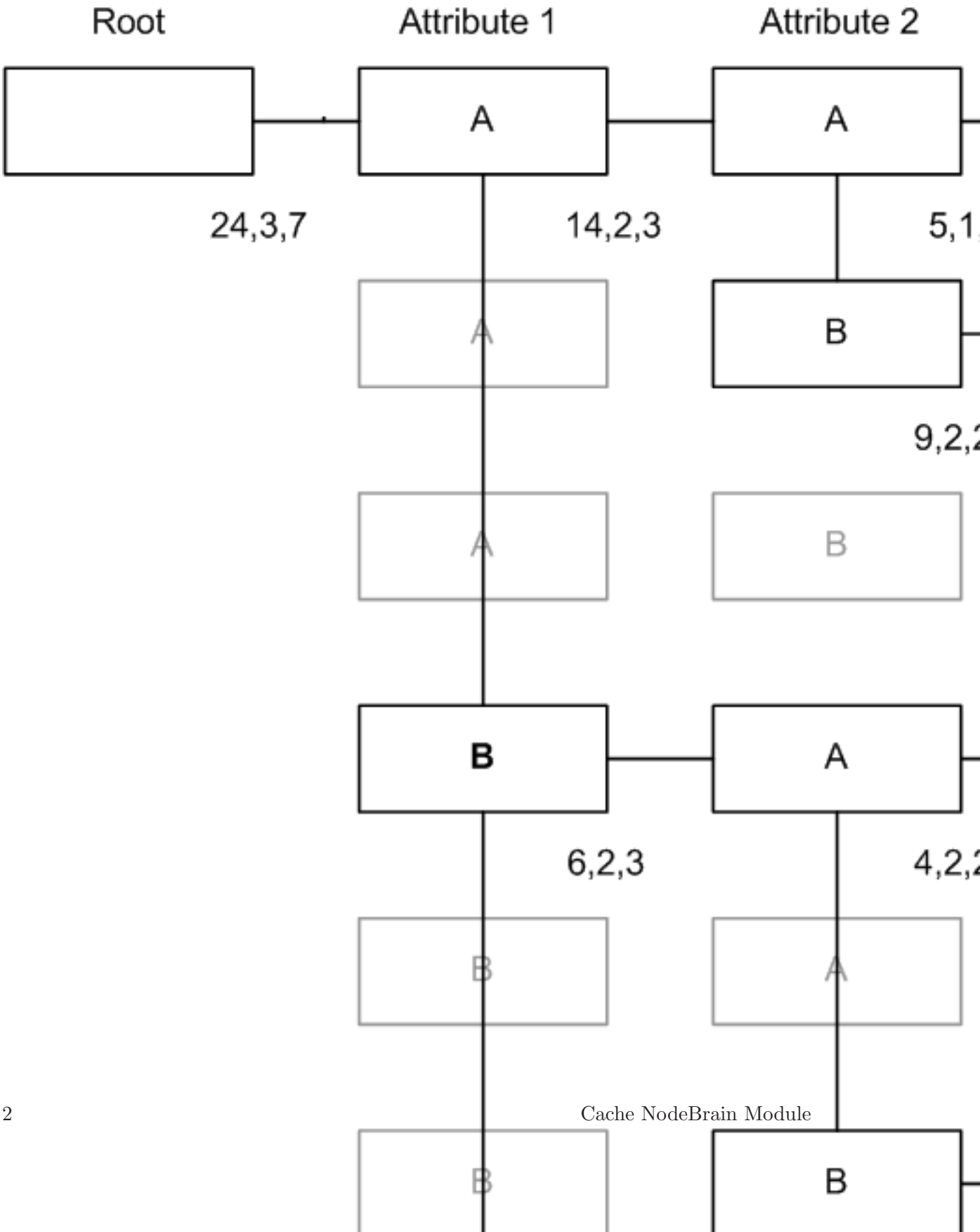
1 Concepts

The Cache module implements a single skill by the same name. A cache node is a table whose rows can be set to expire if not refreshed within some interval of time.

To perform real-time event correlation on a stream of events, you compare the parameters of each new event with parameters of prior events. For this purpose, NodeBrain provides a simple structure called an *event cache*, a memory resident table for relatively short-term storage of events.

An event cache is logically a table where each row represents an event, defined by the values in each column. NodeBrain implemented an event cache as a tree structure with embedded

counters for measuring repetition and variation. There may be any number of attributes (columns), but three attributes are used here to illustrate the concepts.



Each node in a cache structure contains an event attribute value and three counters: hits, kids, and rows.

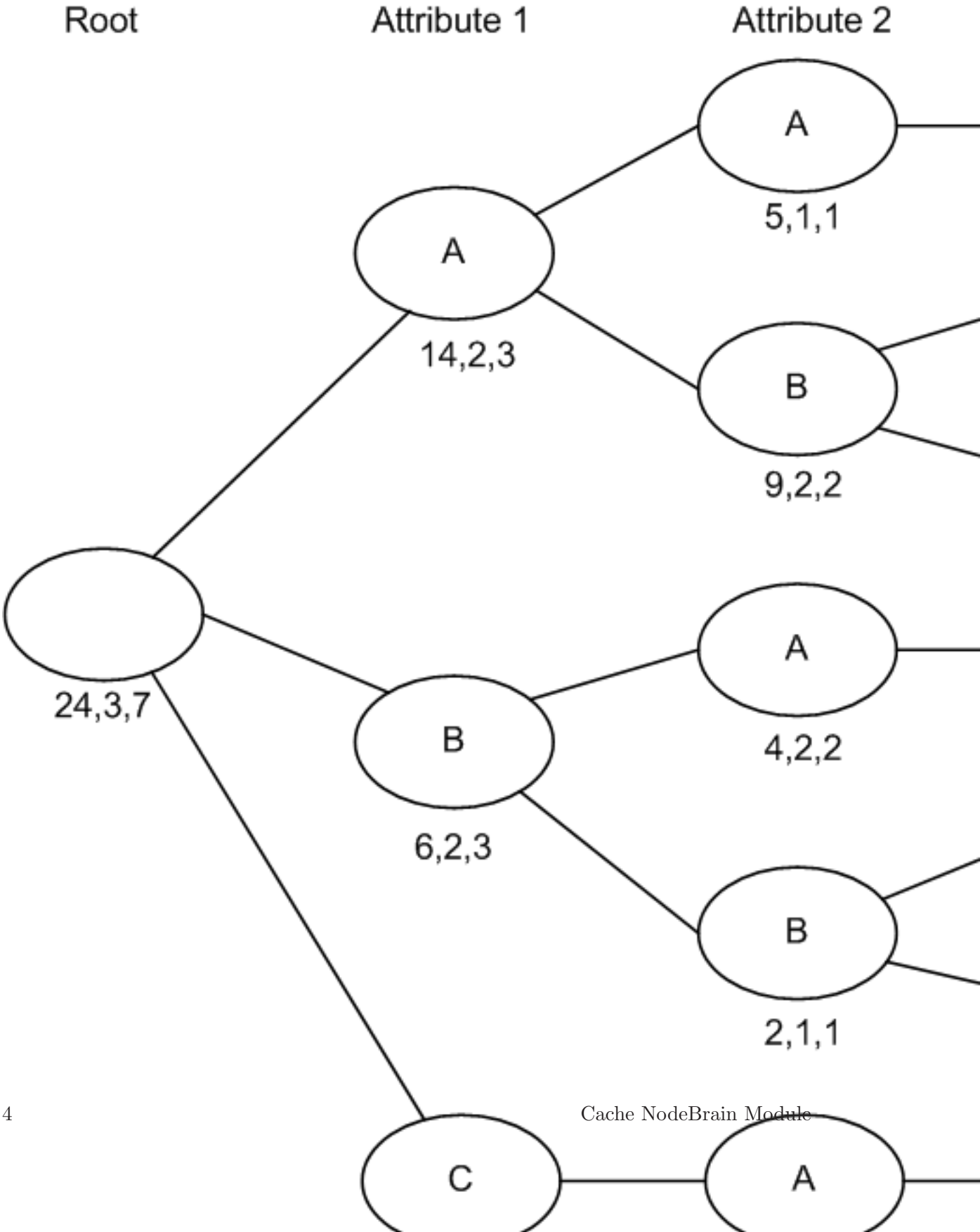
- Hits–number of times the value is represented by a node
- Kids–number of subordinate nodes in the next column (branches)
- Rows–number of rows in the subordinate sub-cache (leaves)

The hit count is used to measure repetition. When you "add" a row (set of values) to an event cache, if a node already exists for a given value, it increments the hit counter instead of inserting a new node. In the illustration above, the first number below a box (node) represents the hit count. You will notice that hit counts sum from right to left, so the hit count in a given node is the sum of the hit counts of the child nodes in the next column to the right.

The kid and row counts are used to measure variation. These counts appear above as the second and third number below a box. The root node in the example has 3 kids and 7 rows. The first node in the Attribute 1 column has 2 kids and 3 rows.

You can also represent an event cache as a sideways tree (see illustration below). What is called the kid count is just the number of branches on the right side of a node. What is called the row count is just the number of final nodes to the right of a node. A final node on the right represents a complete row and only has a hit count. The event (A,B,B) has occurred 7 times (hit count is 7). The event (A,B) has a hit count of 9, a kid count of 2,

and a row count of 2. The event (A) has a hit count of 14, a kid count of 2, and a row count of 3. Notice that both hits and rows sum from right to left.

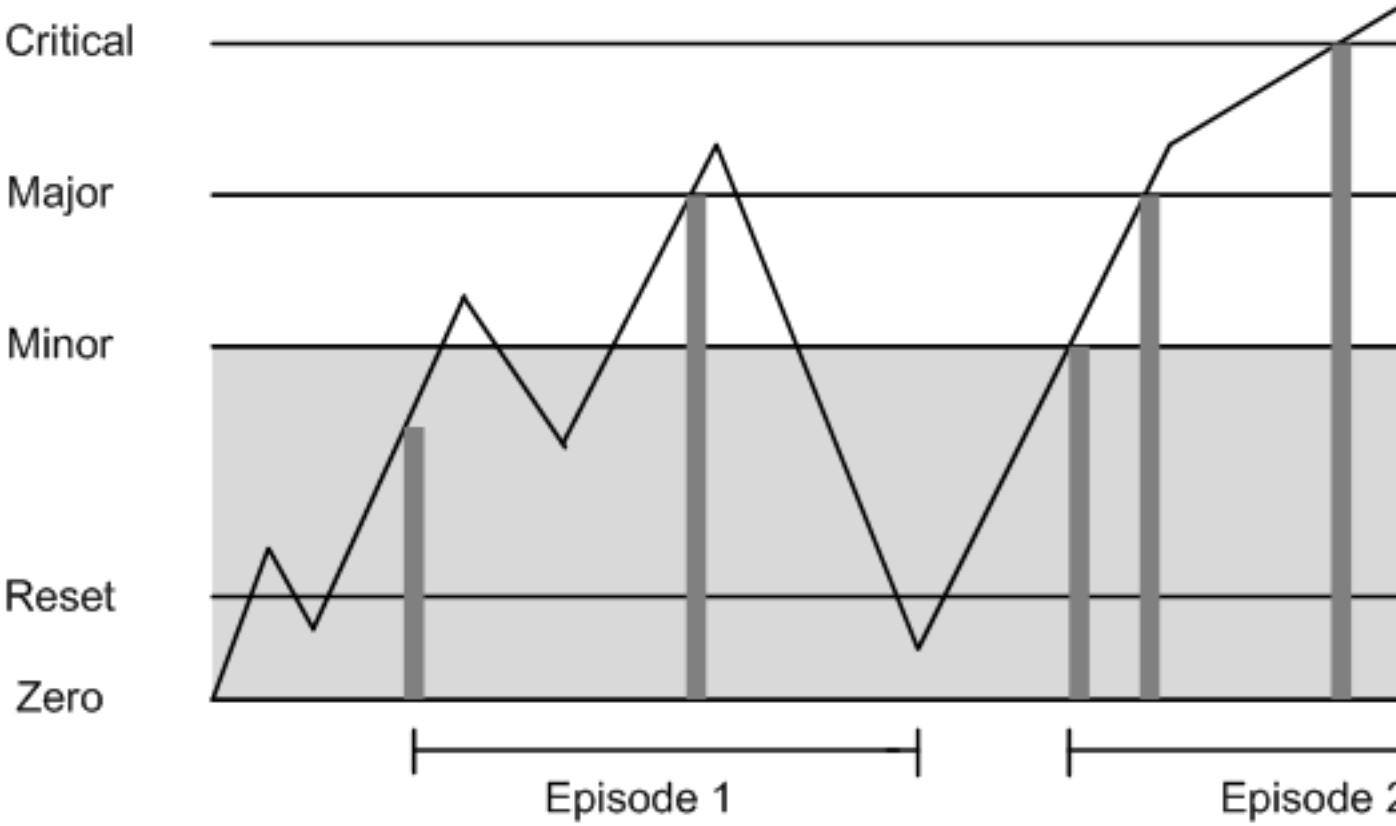


Events are retained in an event cache for a specified interval. Let's say the cache interval is 4 hours and last asserted (B,B,A) at 02:00. At 06:00, the hit counter on the final node A of (B,B,A) is decremented. NodeBrain moves from right to left decrementing counts down to the root node. If a hit counter goes to zero, the node is removed. This causes the kid count on the node to the left to be decremented. So at any time the counters represent what has happened over the past cache interval. This is a sliding interval, not a fixed interval. At 06:23, the cache represents the activity from 02:23 to 06:23.

This enables easy implementation of rules of the forms listed below. Here a cache (x,y) with an interval of T is used. In both cases, asserting (x,y) gets (A,x,y). In the first case, when you get (C,x,y), you test for (x,y) and it responds on true. In the second case when you get (x,y), it asserts not (x,y). In the second case, it responds when any (x,y) expires.

1. if (A,x,y) occurs followed by (C,x,y) within T seconds, then ...
2. if (A,x,y) occurs without (C,x,y) within T seconds, then ...

Up to three thresholds may be defined for each type of counter (hits, kids, rows) for each attribute (column). A reset threshold may also be specified.



When a counter crosses a threshold for the first time, it triggers a response. A trigger point is represented with a vertical bar in the figure above. The actual response is determined by rules that are described later. For now, let's just say the cache triggers the response rules. Any given threshold (minor, major, or critical) will not trigger a second time for a

given counter until the count has dropped to the reset threshold. The reset threshold is used as evidence that an abnormal episode ended and was returned to a normal state. If a threshold is crossed again, it is treated as a new episode. If a reset threshold is not specified, it defaults to zero.

1.1 Cache Definition

The cache module supports a relatively complex syntax for specifying various cache parameters. This complexity is illustrated by the following example.

```
define connie node cache(~(d)):(~(3h):x(100,200,300)[3,6,9]{10,20,30},y);
```

The individual parameters are covered in the following sections on related topics.

1.2 Cache Attributes

Cache attributes are specified as a list of names within parentheses. To define a cache named abc with attributes a, b, and c, the following define statement is used.

```
define abc node cache:(a,b,c);
```

The terms a, b, and c are automatically defined within the context and may be referenced as if they had been explicitly defined as follows.

```
abc. define a cell;
abc. define b cell;
abc. define c cell;
```

1.3 Cache Assertions

Rows are added to a cache and counters incremented, when you assert values.

```
abc. assert ("NodeBrain","patch","Solaris");
abc. assert ("NodeBrain","patch","Linux");
abc. assert ("NodeBrain","defect","HP-UX");
abc. assert ("NodeBrain","patch","Linux");
abc. assert ("cron","patch","Linux");
```

After the assertions above, the abc cache root node would have 5 hits, 4 rows, and 2 kids. Each unique partial row would have counts as shown.

```
() 5 hits, 4 rows, 2 kids
("NodeBrain") 4 hits, 3 rows, 2 kids
("NodeBrain","patch") 3 hits, 2 rows, 2 kids
("NodeBrain","patch","Solaris") 1 hits
("NodeBrain","patch","Linux") 2 hits
("NodeBrain","defect") 1 hits, 1 rows, 1 kids
("NodeBrain","defect","Linux") 1 hits
("cron") 1 hits, 1 rows, 1 kids
("cron","patch") 1 hits, 1 rows, 1 kids
("cron","patch","Linux") 1 hits
```

The assertion syntax illustrated below is normally used when you want to include assertions for terms within the node's context that are not managed by the node module. This provides additional information for use by the node's rules.

```
abc. assert ("NodeBrain","patch","Solaris"),component="nb",status="beta";
```

If that is not a requirement, you may prefer the following syntax.

```
assert abc("NodeBrain","patch","Linux");
```

When a cache assertion is a rule action, avoiding the context prefix enables the elimination of the verb ASSERT and simplifies the reference to terms within the context where the rule is defined.

```
define failedSystem node cache:(~(20m):system);
define event node;
event. assert ?.type,?.system,?.text;

event. define r1 on(type="Failed") failedSystem(system);
... instead of ...
event. define r1 on(type="Failed"):failedSystem. assert (event.system);
...
event. alert type="Failed",system="happynode.com";
```

1.4 Cache Intervals

A cache interval is optional. If specified, it determines the life of cache assertions. The following example specifies a cache interval of 4 hours.

```
define abc node cache:(~(4h):a,b,c);
```

Four hours after a row is asserted to this cache, the hit counter for that row is decremented. When a hit counter goes to zero, a row is removed. This may have an impact on cache conditions and rules.

```
abc("fred","joe","sam") # cache condition

abc. assert ("fred","joe","sam"); # cache assertion
```

If the assertion above is not repeated within four hours, the cache condition will transition from true to false when the row expires. As long as the assertion is repeated in less than 4 hours, the cache condition remains true.

Specifying an exclamation point (!) in front of the time interval will cause the context to be alerted each time a row expires.

```
define abc node cache:(!~(4h):a,b,c);
```

1.5 Cache Thresholds

Cache thresholds are specified as a list of numbers following an attribute. The type of threshold is specified by the choice of enclosing symbols.

‘()’ hits

‘[]’ kids

‘{}’ rows

The command below defines a cache with a hit threshold of 20, a kid threshold of 3, and a row threshold of 10 for the attribute a.

```
define abc node cache:(a(20)[3]{10},b,c);
```

When a set of values are asserted to a cache and a threshold is reached, the cache node is automatically alerted. Rules are written to respond to these alerts. We’ll cover that in a bit.

Four thresholds may be set for each attribute: reset, minor, major, and critical. If a reset threshold is specified, it must be the first value in the list and prefixed with a toggle (^). The following example specifies thresholds of 20, 100, and 250 for the hit count on any row in the cache. A reset threshold of 5 is specified.

```
define abc node cache:(a,b,c(^5,20,100,250));
```

When a threshold is reached on a given counter the node’s context is alerted. For a given counter to trigger on the same threshold again, it must first drop down to the reset threshold. By default, the reset threshold is zero.

The following example specifies thresholds of 3, 9, and 27 for (a,b) kids, or the number of unique values of c for a given value of a and b.

```
define abc node cache:(a,b[3,9,27],c);
```

All of these examples may be combined in a single statement because each type of threshold may be specified for any attribute. Actually the last attribute only has a hit counter since there are no kids or subordinate rows.

```
define abc context cache:(a(20)[3]{10},b[3,9,27],c(^5,20,100,250));
```

To specify thresholds at the cache root level, use a colon before the first attribute. The following definition establishes a hit threshold of 1000 and a kid threshold of 3 at the cache root level. The abc node will be alerted when 3 unique values of a are asserted or 1000 assertions are made.

```
define abc node cache:((1000)[3]:a,b[3,9,27],c);
```

1.6 Cache Rules

Cache rules operate like any other rule within a context, but you need to base them on terms that are implicitly defined and alert commands that come from "within" a cache when thresholds are reached. Consider the following definitions.

```
define abc node cache:(~(4h):a,b[10,20],c(40,60));

abc. define r1 if(b__kidState):$ action including $${a} $${b} $${b__kids}
abc. define r2 if(c__hitState):$ action including $${a} $${b} $${c} $${c__hits}
```

When a cache assertion causes a counter to hit a threshold, the context is alerted with values assigned to special terms that describe the threshold condition. For example, suppose the following assertion produces a tenth value of c for (a,b). In other words, "sam" is the tenth value of c for ("fred","joe") within the previous 4 hours.

```
abc. assert ("fred","joe","sam");
```

The internal alert is equivalent to the following.

```
abc. alert a="fred", b="joe", c="sam", b__kidState=1, b__kids=10;
```

1.7 Cache Terms

The referenced terms are automatically defined when a cache is defined and asserted by a cache when it alerts the context. Here is a list of them.

‘*attribute*__hits’
: number of times the node has been asserted

‘*attribute*__kids’
: number of nodes in the next column

‘*attribute*__rows’
: number of rows a node participates in

‘*attribute*__hit’
: State hit threshold reached

‘*attribute*__kid’
State kid threshold reached

‘*attribute*__row’
State row threshold reached

‘_interval’
cache interval in text appropriate for a message

‘_action’ "expire" (only when "!" precedes the interval)

1.8 Cache Conditions

A cache condition returns a value of True if the parameter list matches a row in the cache, a value of Unknown if any of the arguments are Unknown, and otherwise returns False.

Suppose you want to take some action when an event of Type T2 occurs within 5 minutes after an event of Type T1 if both events have the same value for attributes A and B. This could be accomplished with the following rule set.

```
define event node;     # define a context to be alerted
event. define t1ab node cache:(~(5m):a,b); # define cache
event. define r1 if(Type="T1") t1ab(A,B);   # populate cache
event. define r2 if(Type="T2" and t1ab(A,B)):action # lookup
```

The highlighted cache condition is True when the t1ab cache contains an entry for the current values of A and B. If either A or B is Unknown, the cache condition is Unknown. Otherwise, the cache condition is False.

The event stream for this context is generated through a series of commands of the following form.

```
event. alert Type="type",A="a",B="b";
```

When an event of type T1 occurs, rule r1 asserts (A,B) to the cache. This inserts an entry for the current value A and B. This entry will expire within 5 minutes. When an event of type T2 occurs, rule r2 will fire if the cache contains an entry for the values of A and B. If the following events occur within a 5-minute period, the final event will cause rule r2 to fire.

```
event. alert Type="T1",A="man",B="happy";
event. alert Type="T2",A="pilot",B=52;
event. alert Type="T1",A="sister",B="good";
event. alert Type="T0",A="buddy",B="cool";
event. alert Type="T2",A="man",B="happy";
```

If you defined the cache without scheduled expiration of entries, you must explicitly delete entries when appropriate.

```
event. define t1ab node cache:(a,b); # define cache
event.t1ab. assert ("abc","xyz"); # insert entry if new
event.t1ab. assert !("abc","xyz"); # delete entry
event.t1ab. assert !("abc"); # delete group of entries
event.t1ab. assert !(); # delete all entries
```

With or without an expiration period, you may want to delete entries based on some condition. This is simply a way of forcing the cache condition to be False, just as asserting an entry forces it to be True. So, you can think of a cache condition as a dynamic set of named Boolean switches. You address a specific switch via the argument list.

Because a cache condition cache(x) is False when cache(x) has not been asserted or the assertion has expired, we say that a cache uses the closed world assumption. This is just a way of saying what isn't known to be true is assumed to be false. Because of this assumption, a cache condition will never return a value of Unknown when the arguments are known—it is always True or False. This means the condition ?*cache*(x) is never true.

Although it may seem logically inconsistent, a cache will accept an assertion to an Unknown state and arrive at a False state.

```
assert ?cache(x);  # make the condition cache(x) false
```

This is consistent with the closed world assumption. Once you no longer know something to be True, it is immediately assumed to be False.

1.9 Cache Facet Conditions

Facets are different aspects of a node the you can operate on in some way using node functions. The Cache module provides three functions that can be used in the context of an evaluation (formula), but not assertions or commands.

- @kids(*argList*) - returns the number of unique values directly subordinate to the element identifed by the argument list.
- @rows(*argList*) - returns the number of unique rows subordinate to the element identified by the arument list.
- @hits(*argList*) - returns the number of assertions for the element identified by the arument list.

In the example below, we set a useless threshold of (0) on `c` just to force the cache to maintain hit counts. This is because the cache wants to be lazy. (This requirement may be removed in a future release.) See if you agree with the comments indicating the number assigned to `x` in each of the assertions.

```
define payday node cache:(a,b,c(0));
payday. assert (1,2,3);
payday. assert (1,3,3);
payday. assert (1,4,3);
payday. assert (1,4,4);
payday. assert (1,4,4);
assert y=1,z=4;;
assert x=payday@kids(y);     # 3  (2,3,4)
assert x=payday@rows(y);     # 4  ((2,3),(3,3),(4,3),(4,4))
assert x=payday@hits(y);     # 5  Number of assertions starting with 1
assert x=payday@kids(y,z);   # 2  (3,4)
assert x=payday@rows(y,z);   # 2  (3,4)
assert x=payday@hits(y,z);   # 3  Number of assertions starting with (1,4)
assert x=payday@hits(y,z,4); # 2  Number of assertions for (1,4,4)
```

1.10 Cache Release Condition

A cache release condition provides a convenient way to empty a cache on a predefined condition. Normally the condition is a time condition used to periodically empty the cache. The following cache is emptied on the hour.

```
define abcCache node cache(~(h)):(~(10m):a,b,c);
```

The following cache and rule implement the same functionality.

```
define abcCache node cache:(~(10m):a,b,c);
abcCache. define release on(~(h)) ?abcCache();
```

Sometimes a cache release condition should be used instead of an interval, and other times it should be used in addition to an interval. Consider the following two cache nodes.

```
define A node cache:(~(2h):a,b,c(20));
define B node cache(~(2h)):(a,b,c(20));
```

Node A uses an interval, while node B uses a release condition. There is a significant difference. Assertions to node A expire 2 hours after they are last made, and the cache keeps track of how many times a given assertion was made in the last 2 hours. Assertions to node B expire at the end of the current 2-hour release period—anywhere from 1 second to 2 hours. For node B, it is not necessary for the cache to set timers for every assertion because all assertions to the cache expire at the same time. When the threshold is reached for a given assertion to node A, it is not reset as long the assertion is repeated within 2 hours. With node B, all counters are reset every 2 hours, so the threshold may be reached once within every 2-hour release period for a continuously repeating assertion. So node A is handy when you want to respond only once for each episode, and node B is handy when you want a reminder every 2 hours that an episode is continuing.

Now let's consider the following node C that has both a release condition and an interval. It will detect 20 assertions within a sliding 5-minute interval and not trigger again for continuing episodes until the end of an 8-hour release period.

```
define C node cache(~(8h)):(~(5m):a,b,c(20));
```

The release condition is a cell expression—not necessarily a time condition. Here's an example using a release condition based on x and y.

```
define D node cache(x>27 and y):( ~(5m):a,b,c(20));
```

2 Tutorial

> *There is a measure in everything. There are fixed limits beyond which and short of which right cannot find a resting place.* —Horace (65 BC–8 BC)

A cache node, like a tree node, provides a place to remember events as associations, or relations. But a cache adds to this the ability to measure repetition and variation within intervals of time and respond when predefined limits are reached. It enables you to detect patterns of events that can be described as follows:

- X happened N times within time period P or interval I.
- X was associated with Y in N events within time period P or interval I.
- X was associated with N different values of Y within time period P or interval I.
- X happened within interval I after Y happened.

Here X and Y represent the values of a set of one or more event attributes (A, B, C, ...). For example, X might represent a (type, city) tuple while Y represents a (customer, item, quantity) tuple. If you think of a cache as a table, which is sometimes a good way to visualize it, a *tuple* is just a table row. If you think of a cache as a table-of-tables, a better way to visualize it, then Y is a row in a table associated with row X.

2.1 Thresholds

Here's a 24-hour cache node with four thresholds and four rules to respond when they are reached.

```
#!/usr/local/bin/nb
# File: tutorial/cache/cache.nb
define horace node cache:(~(24h):type(20),city{10}[5],customer,item,quantity(3));
horace. define r1 if(type__hitState): $ # There have been ${type__hits} ${type} events
horace. define r2 if(city__rowState): \
  $ # (${type},${city}) had ${city__rows} different events
horace. define r3 if(city__kidState): \
  $ # (${type},${city}) had ${city__kids} different customers
horace. define r4 if(quantity__hitState): \
  $ # (${type},${city},${customer},${item},${quantity}) happened ${quantity__hits} times
```

Here's a set of events represented as assertions to the 24-hour cache node named "horace."

```
# File: tutorial/cache/events.nb
horace. assert ("purchase","Paris","Bruno","iPod",5);
horace. assert ("purchase","Paris","Bruno","iPod",5);
horace. assert ("purchase","Paris","Bruno","iPod",5);
horace. assert ("purchase","Paris","Bruno","shirt",2);
horace. assert ("purchase","Paris","Bruno","shoes",1);
horace. assert ("purchase","Paris","Madeleine","shoes",5);
horace. assert ("purchase","Paris","Madeleine","skirt",1);
horace. assert ("purchase","Paris","Madeleine","bread",2);
horace. assert ("purchase","Paris","Jeannine","bread",1);
horace. assert ("purchase","Paris","Laure","bread",1);
horace. assert ("purchase","Paris","Henri","iPod",1);
horace. assert ("return","Paris","Henri","iPod",1);
horace. assert ("return","Paris","Madeleine","iPod",1);
horace. assert ("purchase","London","Abigail","milk",1);
horace. assert ("purchase","London","Addie","bread",1);
horace. assert ("purchase","London","Alston","bread",1);
horace. assert ("purchase","London","Alston","candy",1);
horace. assert ("purchase","London","Alston","candy",2);
horace. assert ("purchase","London","Alston","candy",3);
horace. assert ("purchase","London","Alston","candy",4);
horace. assert ("purchase","London","Alston","candy",5);
horace. assert ("purchase","London","Alston","candy",6);
horace. assert ("purchase","London","Alston","candy",7);
horace. assert ("purchase","London","Alston","candy",8);
horace. assert ("purchase","London","Alston","candy",9);
```

Here's a sample execution.

```
$ ./cache.nb events.nb
2009/01/31 09:10:04 NB000I Argument [1] ./cache.nb
> #!/usr/local/bin/nb
> # File: tutorial/cache/cache.nb
> define horace node cache:(~(24h):type(20),city{10}[5],customer,item,quantity(3));
> horace. define r1 if(type__hitState): \
  $ # There have been ${type__hits} ${type} events
> horace. define r2 if(city__rowState): \
  $ # (${type},${city}) had ${city__rows} different events
> horace. define r3 if(city__kidState): \
  $ # (${type},${city}) had ${city__kids} different customers
> horace. define r4 if(quantity__hitState): \
  $ # (${type},${city},${customer},${item},${quantity}) happened ${quantity__hits} times
2014-07-20 22:04:46 NB000I Source file "./cache.nb" included. size=537
2014-07-20 22:04:46 NB000I Argument [2] events.nb
> # Filte: tutorial/cache/events.nb
> horace. assert ("purchase","Paris","Bruno","iPod",5);
> horace. assert ("purchase","Paris","Bruno","iPod",5);
> horace. assert ("purchase","Paris","Bruno","iPod",5);
2014-07-20 22:04:46 NB000I Rule horace.r4 fired
: horace. # (purchase,Paris,Bruno,iPod,5) happened 3 times
> horace. assert ("purchase","Paris","Bruno","shirt",2);
> horace. assert ("purchase","Paris","Bruno","shoes",1);
> horace. assert ("purchase","Paris","Madeleine","shoes",5);
> horace. assert ("purchase","Paris","Madeleine","skirt",1);
> horace. assert ("purchase","Paris","Madeleine","bread",2);
> horace. assert ("purchase","Paris","Jeannine","bread",1);
> horace. assert ("purchase","Paris","Laure","bread",1);
> horace. assert ("purchase","Paris","Henri","iPod",1);
2014-07-20 22:04:46 NB000I Rule horace.r3 fired
: horace. # (purchase,Paris) had 5 different customers
> horace. assert ("return","Paris","Henri","iPod",1);
> horace. assert ("return","Paris","Madeleine","iPod",1);
> horace. assert ("purchase","London","Abigail","milk",1);
> horace. assert ("purchase","London","Addie","bread",1);
> horace. assert ("purchase","London","Alston","bread",1);
> horace. assert ("purchase","London","Alston","candy",1);
> horace. assert ("purchase","London","Alston","candy",2);
> horace. assert ("purchase","London","Alston","candy",3);
> horace. assert ("purchase","London","Alston","candy",4);
> horace. assert ("purchase","London","Alston","candy",5);
> horace. assert ("purchase","London","Alston","candy",6);
2014-07-20 22:04:46 NB000I Rule horace.r1 fired
: horace. # There have been 20 purchase events
> horace. assert ("purchase","London","Alston","candy",7);
2014-07-20 22:04:46 NB000I Rule horace.r2 fired
: horace. # (purchase,London) had 10 different events
> horace. assert ("purchase","London","Alston","candy",8);
> horace. assert ("purchase","London","Alston","candy",9);
2014-07-20 22:04:46 NB000I Source file "events.nb" included. size=1441
2014-07-20 22:04:46 NB000I NodeBrain nb[4608] terminating - exit code=0
$
```

Okay, now why did rules `r1`, `r2`, `r3`, and `r4` fire when they did? Each time a tuple is asserted to the cache, it updates a tree-of-trees structure as needed to retain the full set of asserted tuples. It also updates three counters at each node within the tree-of-trees: hits, rows, and kids. The cache module defines *hits* as the number of times an assertion arrives

at a given node within the tree, *rows* as the number of subordinate table rows represented by the subordinate tree-of-trees, and *kids* as the number of directly subordinate nodes—or the number of unique values in the first column of the subordinate table. In the definition of the "horace" cache node, you specify thresholds for hits using (), rows using `{}`, and kids using []. When a threshold is reached, the cache node alerts itself. The rules defined for the node handle the alerts. In the example above, when `r4` fired, it was responding to an invisible alert that would look something like the following if it were not invisible.

```
horace. alert quantity__hitState,quantity__hits=3,type="purchase",city="Paris", \
  customer="Bruno",item="iPod",quantity=5;
```

This alert was triggered because the hit counter for the (`purchase,Paris,Bruno,iPod,5`) node in the cache reached the specified limit of 3 within a 24-hour interval. See if you can figure out why rules `r1`, `r2`, and `r3` fired when they did.

In this example, the rules just issue comments. In a real application, the rules would have taken action of some kind: an alert, assertion, alarm, shell command, and so on.

2.2 Tuple Expiration

A tuple in a cache can be retained for a defined time interval (e.g., 24 hours in the example above), until the end of a period (e.g., end of current minute, hour, day), until some state is detected, or indefinitely.

```
# interval
define horace node cache:(~(24h):type(20),city{10}[5],customer,item,quantity(3));
# end of period
define horace node cache(~(d)):(type(20),city{10}[5],customer,item,quantity(3));
# state a=1
define horace node cache(a=1):(type(20),city{10}[5],customer,item,quantity(3));
# indefinitely
define horace node cache:(type(20),city{10}[5],customer,item,quantity(3));
```

You can also remove a tuple or set of tuples from a cache at any time. The following command removes all tuples starting with (`"purchase","London"`).

```
horace. assert ?("purchase","London");
```

In addition to alerting when a threshold is reached, you can direct a cache to alert when a tuple expires. This enables us to assert a tuple to the cache and take action if you haven't asserted it again within the expiration period. In other words, you can use a cache to know when something hasn't happened for some interval.

```
#!/usr/local/bin/nb
# File: tutorial/cache/tardy.nb
define tardy node cache:(!~(6s):Source);
tardy. define r1 if(_action="expire"): $ # ${Source} has been quiet for ${_interval}
tardy. assert ("Fred");
```

When executed below, you are notified when "Fred" has not been asserted to the cache for 6 seconds.

```
./tardy.nb -
2009/01/31 10:38:06 NB000I Argument [1] ./tardy.nb
> #!/usr/local/bin/nb
> # File: tutorial/cache/tardy.nb
> define tardy node cache:(!~(6s):Source);
> tardy. define r1 if(_action="expire"): $ # ${Source} has been quiet for ${_interval}
> tardy. assert ("Fred");
> # Press the ENTER key once repeatedly until the rule fires
> # Should happen in 6 seconds
2014-07-20 22:10:30 NB000I Source file "./tardy.nb" included. size=284
2014-07-20 22:10:30 NB000I Argument [2] -
2014-07-20 22:10:30 NB000I Reading from standard input.
---------- --------
>
>
>
>
>
>
2014-07-20 22:10:36 NB000I Rule tardy.r1 fired
: tardy. # Fred has been quiet for 6 seconds
>
```

2.3 Event Sequence

A cache node can be used like a tree node for detecting a sequence of events. However, the cache node, having support for scheduled tuple expiration, can also support a timing condition. The `OnJust` cache below is used to remember for 5 seconds that a switch has been turned on. The `TurnedOn` rule asserts the name of a switch to the cache each time a switch is turned on. The `TurnedOff` rule responds to a switch being turned off if the cache still remembers the switch being turned on.

```
#!/usr/local/bin/nb
# File: tutorial/cache/sequence.nb
define OnJust node cache:(~(5s):switch);
define TurnedOn if(on) OnJust(switch);
define TurnedOff if(!on and OnJust(switch)):...
... $ # The ${switch} turned off within ${OnJust._interval} of turning on

# Sample events
alert on,switch="kitchen light";
alert on,switch="porch light";
alert !on,switch="kitchen light";
-sleep 6
alert !on,switch="porch light";
alert on,switch="porch light";
alert !on,switch="porch light";
```

In the execution below, notice that the `TurnedOff` rule did not respond when the porch light stayed on for 6 seconds.

```
$ ./sequence.nb
2014-07-20 22:14:53 NB000I Argument [1] ./sequence.nb
> #!/usr/bin/nb
> # File: tutorial/cache/sequence.nb
> define OnJust node cache:(~(5s):switch);
> define TurnedOn if(on) OnJust(switch);
> define TurnedOff if(!on and OnJust(switch)): \
  $ # The ${switch} turned off within ${OnJust._interval} of turning on
> # Sample events
> alert on,switch="kitchen light";
2014-07-20 22:14:53 NB000I Rule TurnedOn fired (OnJust(switch)=!!)
> alert on,switch="porch light";
2014-07-20 22:14:53 NB000I Rule TurnedOn fired (OnJust(switch)=!!)
> alert !on,switch="kitchen light";
2014-07-20 22:14:53 NB000I Rule TurnedOff fired
: # The kitchen light turned off within 5 seconds of turning on
> -sleep 6
[4619] Started: -sleep 6
[4619] Exit(0)
> alert !on,switch="porch light";
> alert on,switch="porch light";
2014-07-20 22:14:59 NB000I Rule TurnedOn fired (OnJust(switch)=!!)
> alert !on,switch="porch light";
2014-07-20 22:14:59 NB000I Rule TurnedOff fired
: # The porch light turned off within 5 seconds of turning on
2014-07-20 22:14:59 NB000I Source file "./sequence.nb" included. size=463
2014-07-20 22:14:59 NB000I NodeBrain nb[4618] terminating - exit code=0
009/01/31 11:23:20 NB000I NodeBrain nb[6409] terminating - exit code=0
$
```

2.4 Event Correlation

A cache node is useful for event correlation where the goal is to detect repetition, variation, or sequence. Rules can be used to recognize input events and assert multiple attribute combinations (tuples) to different cache nodes to detect different patterns. A single cache can detect multiple patterns, but it often necessary to specify attributes in a different order in different cache nodes to detect all the required patterns. For example, a cache specified as (`child,action[5],toy`) could detect a given child taking a given action on five different toys, while a cache specified as (`toy[10],child[3],action(7)`) could detect when 10 different children performed a given action on a given toy, a given child performed three different actions on a given toy, and a given child performed a given action on a given toy seven times. The last condition could be detected by the first cache if you included another threshold (`child,action[5],toy(7)`). But the first two conditions detected by the second cache could not be detected by the first cache. It is necessary to use two different cache nodes with the attributes in a different order to detect all four conditions.

3 Commands

This section describes the commands used with a cache node.

3.1 Define

Syntax

cacheDefineCmd	::= define <9a> term <9a> node cache[(cacheRelease)] : [<9a> cacheSpec] <95>
cacheRelease	::= cellExpression
cacheSpec	::= ([cacheRootSpec :] cacheAttrList)
cacheRootSpec	::= [!] [cacheInterval] [cacheThresholds]
cacheInterval	::= ~ "(" integer (s | m | h | d) ")"
cacheThresholds	::= [cacheHitSpec] [cacheKidSpec] [cacheRowSpec]
cacheHitSpec	::= "(" cacheThreshold ")"
cacheKidSpec	::= "[" cacheThreshold "]"
cacheRowSpec	::= "{" cacheThreshold "}"
cacheThreshold	::= [^ integer ,] integer [, integer [, integer]]
cacheAttrList	::= cacheAttrSpec { , cacheAttrSpect }
cacheAttrSpec	::= term [cacheThresholds]

4 Triggers

Index

C

D

T