

Intel® Cluster Checker Developer's Guide

Version 2017.0

July 13, 2016

Disclaimer and Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps. information.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at Intel.com, or from the OEM or retailer.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo, Xeon, Intel Xeon Phi, Cilk, and VTune are trademarks of Intel Corporation in the U.S. and/or other countries.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision 20110804

* Other names and brands may be claimed as the property of others.

© 2016 Intel Corporation.

Contents

1 Introduction **6**

1.1 End-to-end Extension Example 6

1.2 Extending Intel® Cluster Checker 6

2 Embedding Intel® Cluster Checker **8**

3 Data Providers **9**

3.1 Introduction 9

3.2 Running a data provider 10

3.3 Configuration 10

3.3.1 Preset Environment Variables 10

3.4 Definition 11

3.5 Example 14

3.5.1 Basic Data Provider 14

3.5.2 Configurable Command 14

4 Knowledge Base **16**

4.1 Background 16

4.1.1 Brief Overview of Expert Systems 16

4.1.2 CLIPS 16

4.2 Overview of the Intel® Cluster Checker Knowledge Base . . . 17

4.2.1 Key Concepts 17

4.2.1.1 Signs 17

4.2.1.2 Diagnoses 18

4.2.1.3 Remedies 18

4.2.2 Basic Implementation 19

4.2.2.1 Classes 19
4.2.2.2 Rules 19
4.2.2.3 Signs 19
4.2.3 Organization and Directory Structure 20
4.3 Automatically Created Objects 20
4.4 Configurability 21
4.5 Example 22
4.5.1 Class Definition 22
4.5.2 Rules 23
4.5.2.1 Rule 1: Missing good output 23
4.5.2.2 Rule 2: Error case 24
4.5.2.3 Rule 3: Uniformity 25
4.5.2.4 Rule 4: Diagnosis 26
4.6 Developing with CLIPS 27
4.6.1 Editor 27
4.6.2 Style 27
4.6.3 Debugging and Profiling 28

5 Connector **29**
5.1 Overview 29
5.2 Extensions 29
5.2.1 Transform Class Members 30
5.3 Database Interface 30
5.4 Knowledge Base and CLIPS Interface 31
5.4.1 Creating CLIPS Class Instances 31
5.4.2 Parsing database output 32
5.4.3 Handling Parse Errors 32
5.5 Building Extensions 32
5.6 Loading Extensions 32
5.7 Example 32

A Database Schema **33**

Chapter 1

Introduction

Intel® Cluster Checker verifies the configuration and performance of Linux-based clusters and checks compliance with the Intel® Cluster Ready architecture specification. If issues are found, Intel® Cluster Checker diagnoses the problems and may provide recommendations on how to repair the cluster.

This guide describes how developers can extend the tool or embed it into other software. For information on how to use Intel® Cluster Checker, please read the User's Guide.

Chapter 2 describes how to embed Intel® Cluster Checker functionality into other software.

1.1 End-to-end Extension Example

A single end-to-end example is used in Chapters 3, 4, and 5 to help illustrate how to extend Intel® Cluster Checker. In the example, the fictional Waterfowl Industries has developed the `duck` diagnostic tool. This program comprehensively evaluates nodes using its trade secret methodology and rates them on its patented "quack" scale. A node is rated between 1 and 5 "quacks", 5 being best, or if there is an error during the evaluation, prints "honk".

This example is used to illustrate the flow of data through Intel® Cluster Checker in Figure 1.1.

1.2 Extending Intel® Cluster Checker

The steps required extend Intel® Cluster Checker depend on the amount of data already available. Refer to the corresponding chapter(s) depending on the scenario, starting from the bottom of Figure 1.1 and working up:

- If the data is already available in the knowledge base and the goal is to write new or modify existing rules, see Chapter 4.

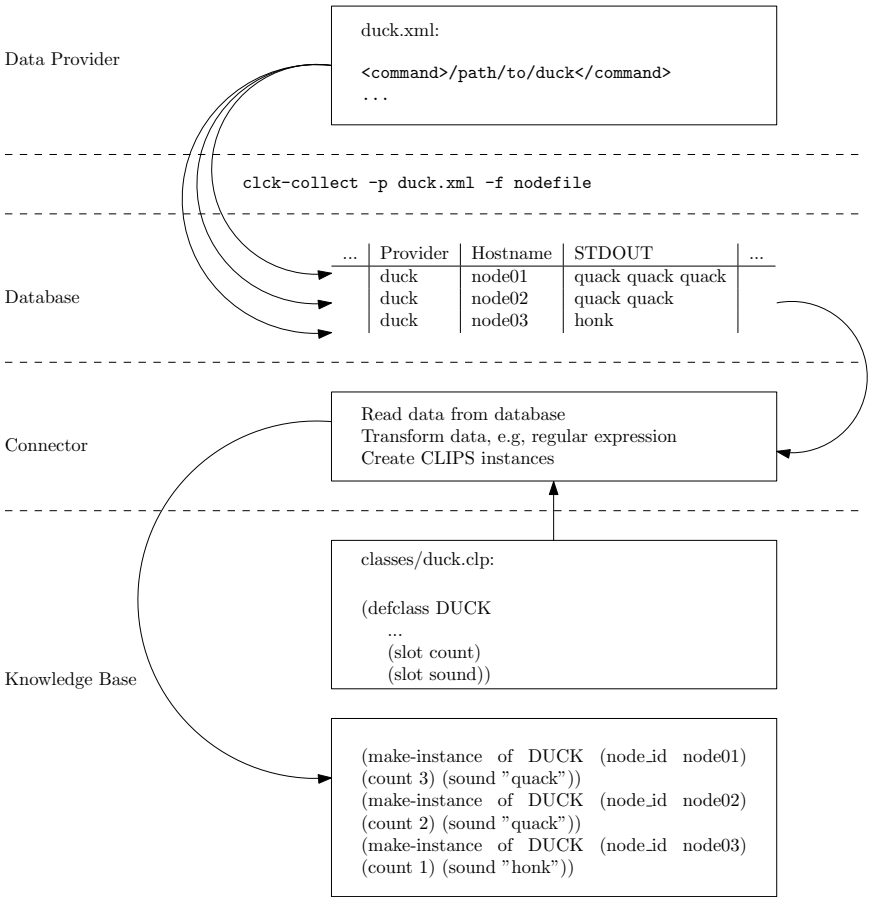


Figure 1.1 – Flow of data through Intel® Cluster Checker

- If the data is already available in the database but is not already available in the knowledge base, see Chapters 4 and 5.
- If the data is not available in the database, see Chapters 3, 4, and 5.

Chapter 2

Embedding Intel® Cluster Checker

A C++11 interface is provided to enable software to embed Intel® Cluster Checker capabilities. The application programming interface (API) documentation is a separate document located at `/opt/intel/clck/2017.0/doc/api.pdf`.

Tip: The API documentation is generated by Doxygen style comments embedded in the header.

The API is defined in `/opt/intel/clck/2017.0/include/clck.h`. Include this header in any source code that is embedding Intel® Cluster Checker and add `-Wl,-rpath,/opt/intel/clck/2017.0/lib/intel64 -lclck` when linking. If not using the environment setup scripts, `clckvars.sh` and `clckvars.csh`, then `-I/opt/intel/clck/2017.0/include` and `-L/opt/intel/clck/2017.0/lib/intel64` also need to be added to the compile and link commands, respectively.

At a minimum, a program embedding Intel® Cluster Checker would call `analyze()` followed by `get_faults()`. Additional functions and configuration options may also be set. The database, connector extensions, and knowledge base are external and their locations need to be specified.

An example is provided online at `https://software.intel.com/en-us/product-code-samples?topic=20903` and is called "API MPI Sample". This sample embeds Intel® Cluster Checker into Message Passing Interface (MPI) programs written in C, C++, Fortran 77, and Fortran 90. Intel® Cluster Checker is called by the first MPI rank immediately after MPI initialization to verify the cluster is performing correctly before starting the main task. If any serious performance issues are detected, the program is aborted before any sub-par "computation" occurs.

Chapter 3

Data Providers

3.1 Introduction

Data providers define the data to be gathered from nodes. A data provider is a metadata file, specifically an XML file, that specifies the command to run, how often it should be run, etc. Additional files may be associated with data providers, such as shell scripts to wrap functionality more complex than a command line, input data sets, etc.

Standard output and standard error of the data provider are saved in the database. Data providers should not perform unnecessary filtering of the output. The philosophy is to output raw, unfiltered data and let the connector (see chapter 5) filter the data into a form usable by the knowledge base. For example, a well-formed data provider would execute `cat /proc/cpuinfo` to obtain the complete set of information stored in this file rather than the more specific `cat /proc/cpuinfo | grep "model name" | uniq` to obtain the cpuid string. The cpuid string, in addition to lots of other useful data - including uses not currently envisioned, can be obtained from the complete output.

Most providers will be local and only collect data from the machine they ran on. These are termed *single node data providers*. However, some providers collect data from multiple nodes and are termed *cluster data providers*. Cluster providers will be supplied with a list of nodes (see section 3.3.1) and do not need worry about how the list is generated. Cluster data providers may be called multiple times with different sets or permutations of nodes.

3.2 Running a data provider

Data providers may be either invoked on-demand or run asynchronously. Please see the Data Collection chapter of the User's Guide for more information about these modes.

An easy way to test a data provider is to run it in on-demand mode. First, run the data provider with the `clck-collect` tool, e.g., `clck-collect -p /path/to/myprovider.xml -f nodefile`. Then, verify the output is correct and stored in the database with the `clckdb` tool, e.g., `clckdb --provider=myprovider`.

Note that before the provider file is parsed, the ownership and permissions of the XML file are verified. The XML file must be owned either by root or the user collecting the data. The XML file must also not be writable by a user other than the file owner. If either of these conditions is not satisfied, the data provider will not be run. These conditions are in place to avoid potential security issues.

3.3 Configuration

In most scenarios, data providers should written such that they do not require configuration. For example, a benchmark data provider should automatically set its input problem size based on querying the node rather than requiring the user to specify the size.

In cases where it may be necessary to override the default values, use environment variables. The recommendation is to prefix the environment variable with `CLCK_PROVIDER_PROVIDER-NAME` where `PROVIDER-NAME` should be replaced with the name of the provider. For example, if the provider is named `duck`, the `CLCK_PROVIDER_DUCK_CMD` could examined to override the default path to the `duck` command. Note that the data provider must handle the environment variables itself; there is no automatic handling provided by Intel® Cluster Checker.

3.3.1 Preset Environment Variables

The following environment variables are automatically set by Intel® Cluster Checker.

`CLCK_DP_NODEFILE`

: For cluster data providers, this variable contains the path to the supplied nodefile. The nodefile contains a list of node hostnames, one per line. For single node data providers, this environment variable is undefined.

`CLCK_TIMEOUT`

: This variable contains the number of seconds until the data provider is forcibly timed out. If a data provider needs to do some additional cleanup, it should set an internal time out less than this value.

3.4 Definition

Each data provider is defined by a XML configuration file. This section describes the XML tags.

The name of the data provider is derived from the filename of the XML file, without the .xml extension.

The base data provider format is:

```
<configuration>
  <architecture> </architecture>
  <command> </command>
  <disable/>
  <encoding> </encoding>
  <max_nodes> </max_nodes>
  <min_nodes> </min_nodes>
  <nodelist/>
  <period> </period>
  <priority> </priority>
  <role> </role>
  <timeout> </timeout>
  <version> </version>
</configuration>
```

The tags understood by the data provider configuration parser are as follows. All the tags are optional with the exception of `<command>`.

`<architecture>` *k1om | x86_64* `</architecture>`

If the value of this tag differs from the architecture of the node, the data provider will not be run.

The default value is empty, meaning to run on any architecture.

`<command>` *string* `</command>`

Command is the key provider metadata. The value specifies the shell command to be run. The value may be a simple command, or it might call a script that executes a series of commands. The contents of standard output and standard error are saved and stored in the database.

If the data provider depends on additional files, such as a script or input data set, the recommended location for these files is a sub-directory with the same name as the provider in `/opt/intel/clck/2017.0/provider/share`. The string token `%PROVIDER_AUXILIARY_PATH%` can be used in the command tag as an alias for this base directory.

The default value of `%PROVIDER_AUXILIARY_PATH%` is `/opt/intel/clck/2017.0/provider/share` and can be overridden by changing the corresponding setting in `/opt/intel/clck/2017.0/etc/clck.xml`.

Tip: A data provider should not assume it is running in any particular directory. If the command is not located in a directory contained in the PATH environment variable, then the full path should be specified in the command tag.

`<disable/>`

This parameter is a Boolean value denoting whether the provider should run or not.

The default value is false.

`<encoding>` *value* `</encoding>`

The encoding parameter is the encoding format to be applied to the standard output and standard error of the command before the output is stored in the database.

Tip: This tag should be used if the output will contain non-text data.

Values allowed are:
ENCODING_NONE
ENCODING_BASE64
ENCODING_RAW
The default value is ENCODING_NONE and output is not encoded.

`<max_nodes>` *integer* `</max_nodes>`

This tag specifies the maximum number of nodes to use when running a cluster provider.

The default value is 1.

When running a cluster provider, both the `<min_nodes>` and `<max_nodes>` tags are compulsory, and the `<max_nodes>` value must be greater than or equal to `<min_nodes>`.

`<min_nodes>` *integer* `</min_nodes>`

This tag specifies the minimum number of nodes to use when running a cluster provider.

This tag defines whether a provider is a single node provider or a cluster provider. Any value above 1 specifies that the provider is a cluster provider. A value of 1, or omitting this tag, specifies that the provider is a single node provider.

When a value higher than 1 is specified, the `<max_nodes>` tag is required.

The default value is 1.

`<nodelist method="`*value*`" number="`*integer*`"/>`

This tag specifies how the nodes in the node list provided to cluster data provider will be permuted. This tag only applies to cluster providers, and only takes effect when data is collected on-demand.

The “method“ attribute defines the algorithm to use to permute the node list.

Valid options for the “method“ attribute are “none“, “rotate_left“, “rotate_right“, “round_robin“ and “random“.

With the “none“ value, the “number“ attribute defaults to 1 and the nodelist will be used as is without permutations.

With the “rotate_left“ value, the first node in the node list is popped off and pushed onto the end of the list successively for each iteration.

With the “rotate_right“ value, the last node in the node list is popped off and pushed onto the beginning of the list successively for each iteration.

With the “round_robin“ value, the nodes are permuted based on a round robin tournament scheduling algorithm.

With the “random“ value, the nodes are shuffled randomly to generate new node lists for each permutation.

The “number“ attribute defines the number of permutations that will be generated for a given node list.

The value 0 specifies that all possible permutations for a given method should be generated. This value of 0 is however ignored for “none“ and “random“ methods, and a value of 1 will be used instead.

Additionally, if the specified number is greater than the maximum number of permutations needed to generate all possible combinations for the given method, then the lower value will be used.

The default method is “none“ and the default number of permutations is 1.

`<period>` *integer* `</period>`

This tag is only used for asynchronous data collection. It is ignored in on-demand mode.

This tag specifies the minimum number of seconds to wait between invocations of the data provider.

A value of ”0” is special and means that the provider should be run only once.

The default value is 60 seconds.

`<priority>` *integer* `</priority>`

This parameter is the operating system scheduling priority (aka the ”nice” value).

The default value is 0.

This tag is optional and is currently unimplemented.

This tag is only used for asynchronous data collection. It is ignored in on-demand mode.

`<role>` *value* `</role>`

The data provider will only run on nodes that have the corresponding node role.

Allowed values are specified in Table 3.1.

Role	Description
boot	Provides software imaging / provisioning capabilities
compute	Is a compute resource
enhanced	Provides enhanced compute resources, e.g., contains additional memory
external	Provides an external network interface
head	Alias for the union of boot, external, job_schedule, login, network_address, and storage
job_schedule	Provides resource manager / job scheduling capabilities
login	Is an interactive login system
network_-address	Provides network address to the cluster, e.g., DHCP
storage	Provides network storage to the cluster, e.g., NFS

Table 3.1 – Node Roles

If this tag is not specified, the default behavior is to run on all node roles.

This tag may be specified multiple times to run on multiple node roles.

`<timeout scale="`*value*`">` *integer* `</timeout>`

This tag specifies the amount of time (in seconds) to wait before forcibly terminating the provider to prevent hanging.

The scale attribute specifies the rate at which the timeout value should increase based on the number of nodes, in the case of a cluster provider.

Valid options for the scale attribute are “constant”, “linear”, “squared”, and “logarithmic”. “constant” does not scale with the number of nodes used, “linear” scales linearly with the number of nodes (e.g. $timeout * num_nodes$), “squared” scales with the number of nodes squared (e.g. $timeout * num_nodes^2$), and “logarithmic” scales logarithmically with the number of nodes (e.g. $timeout * \ln((e-1) + num_nodes)$).

The default and the minimum value is 60 seconds.

The default scale is “constant”.

`<version>` *integer* `</version>`

This tag specifies the version of the output format. The connector may use the value of this tag to parse the output differently.

The default value is 1.

Tip: The version tag should be incremented any time the output format of the data provider changes. If the data provider changes but the output format remains the same, then the value of this tag does not need to be updated.

3.5 Example

3.5.1 Basic Data Provider

The most basic “duck” data provider is shown below. This provider runs the configured command once per day in asynchronous mode, or whenever invoked in on-demand mode.

```
<?xml version="1.0"?>
<configuration>
  <!-- Assume that duck.sh is located in the default Intel(R)
       Cluster Checker install directory. -->
  <command>%PROVIDER_AUXILIARY_PATH%/duck/duck.sh</command>

  <!-- Run once per day in asynchronous mode. -->
  <period>86400</period>
</configuration>
```

A sample “duck” data provider is located at `src/provider` in the ”SDK Duck Sample” at `https://software.intel.com/en-us/product-code-samples?topic=20903`.

Tip: Data providers can be tested in on-demand mode. Assuming the data provider is located in `$HOME/duck.xml`, run the data provider with the command `clck-collect -p $HOME/duck.xml -f nodefile` and verify the output is saved in the database with the command `clckdb --provider=duck`.

3.5.2 Configurable Command

The data provider can be made configurable via an environment variable. For instance, if the data provider is a shell script:

```
#!/bin/sh

# default to use if $CLCK_PROVIDER_DUCK_CMD is not defined
CMD=/opt/waterfowl/3.14/bin/duck
```

```
if [ -n "${CLCK_PROVIDER_DUCK_CMD}" ] ; then
  CMD=${CLCK_PROVIDER_DUCK_CMD}
fi

exec $CMD
```

The value of `CLCK_PROVIDER_DUCK_CMD` may be set by adding the following to `/opt/intel/clck/2017.0/etc/clck.xml` or a custom configuration file:

```
<configuration>
  <collector>
    <provider>
      <duck>
        <cmd>/path/to/duck</cmd>
      </duck>
    </provider>
  </collector>
</configuration>
```

`CLCK_PROVIDER_DUCK_CMD` may also be set in the shell initialization file, e.g., `$HOME/.bashrc`.

See the Reference Manual for more information about configuring data providers.

Chapter 4

Knowledge Base

4.1 Background

4.1.1 Brief Overview of Expert Systems

Intel® Cluster Checker is an expert system. A classic definition of an expert system is "an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solutions." The problem that Intel® Cluster Checker solves is diagnosing system level issues with Beowulf style clusters.

Edward A. Feigenbaum, "Knowledge Engineering in the 1980s", Stanford University Computer Science Department, 1982

Two main elements are called out in this definition, knowledge and inference procedures. Knowledge comes from two sources, observations about an actual system, and if/then rules that encapsulate human expertise. Intel® Cluster Checker relies on data providers to make observations about the cluster and saves the result in a database (see Chapter 3).

Expert systems differ from typical procedural programs in there is not a fixed order of execution. The order is logically inferred, using one of several common schemes such as the Rete algorithm, by dynamically analyzing the interdependence of rules and facts.

http://en.wikipedia.org/wiki/Rete_algorithm

One limitation of expert systems is that they are only as good as the knowledge (rules) they contain. To remain relevant, a knowledge base needs to continually grow and change as new and variant cases are uncovered. This chapter addresses how to express human expertise as knowledge base rules to extend the diagnostic capabilities of Intel® Cluster Checker. Please consider contributing extensions to the Intel® Cluster Checker team so that other users can also benefit.

4.1.2 CLIPS

The C Language Integrated Production Systems (CLIPS) was originally created in the 1980s at NASA's Johnson Space Center. CLIPS is an expert system shell that combines an inference engine with a language for representing knowledge. Like many AI environments, the CLIPS language is very similar to LISP.

http://clipsrules.sourceforge.net/

Joseph Giarratano and Gary Riley, Expert Systems: Principles and Programming, Thomson Course Technology, 2005.

More recently CLIPS added object oriented capabilities. Intel® Cluster Checker is based on the CLIPS Object Oriented Language (COOL).

The CLIPS User's Guide is an excellent, duck filled, introduction to CLIPS (`http://clipsrules.sourceforge.net/documentation/v630/ug.pdf`).

4.2 Overview of the Intel® Cluster Checker Knowledge Base

4.2.1 Key Concepts

Intel® Cluster Checker is organized similarly to clinical decision support systems.

`https://en.wikipedia.org/wiki/Clinical_decision_support_system`

4.2.1.1 Signs

Signs are one of the core elements of the knowledge base. Conceptually, a sign roughly corresponds to a symptom, such as "I have a fever". Signs differ from symptoms in that signs are based on objective, measurable quantities, like the patients temperature. The patients temperature is collected, and then a rule is run to decide whether the temperature value qualifies as a fever or not. (A more complex rule might distinguish between low and high fevers, for example.)

All the varieties of signs have a `state` slot that represents a state diagram, where a sign is first initialized, then transitions to the observed state when a rule is run, and finally becomes diagnosed if the sign is used to make a diagnosis (see the next section).

The `confidence` and `severity` slots are values that range from 0 to 100. Higher values indicate higher confidence in the sign or the item represented by the sign is more serious, respectively. The rule that sets the sign (i.e., transitions it into the observed state) can set the confidence and severity values. Recalling the fever analogy, the corresponding sign may set the severity level higher depending on the temperature, so there could be just one sign representing "I have a fever" with a variable severity level describing whether it is a mild fever or a dangerously high fever.

The guidelines in Table 4.1 should be used to set the confidence level of an issue. The confidence level depends on the certainty of the data, the certainty of the analysis, as well as the occurrence rate of the issue. These three factors should be weighted equally, each accounting for one third of the total confidence level, and the overall confidence level should be the sum of the three.

Range	Data certainty	Analysis certainty	Occurrence rate
0 - 10	Doubtful	Doubtful	Rare
10 - 20	Suspect	Suspect	Unlikely
21 - 30	Probable	Probable	Common
31 - 33	Near certain	Near certain	Very common

Table 4.1 – Confidence level guidelines.

The guidelines in Table 4.2 should be used to set the severity level of an

issue.

Range	Description
0 - 20	The cluster is fully functional, but has minor performance issues and/or does not conform to best practices.
21 - 50	The cluster is essentially functional, but has moderate performance issues and/or a non-core capability has minor functionality problems.
51 - 80	The cluster is minimally functional, but is severely under-performing and/or a non-core capability is non-functional / missing.
81 - 95	A core cluster capability is non-functional / missing.
96 - 100	A cluster component may irreparably fail if not addressed immediately.

Table 4.2 – Severity level guidelines.

Every sign also has a `id` slot that corresponds to a message catalog key (`/opt/intel/clck/2017.0/kb/data/msg_en.xmc`). The message catalog contains a string that describes the sign; typically the string is a single sentence, but it may be longer. By convention, the id value should be the same as the name of the rule that created it. The id value is also used to look up the sign when making diagnoses.

Finally, the `args` slot contains variable values to be inserted into the message catalog string. Together, the id and args slots are roughly analogous to the C printf family of functions. The message catalog can be extended by simply adding new entries.

4.2.1.2 Diagnoses

Diagnoses are made based on the value of signs. A diagnosis is also defined by a rule. Whenever a diagnosis is made, the signs used to make the diagnosis should be transitioned to the 'diagnosed' state. This is important because signs that are not used to make a diagnosis (i.e., left in the observed state) will be printed out as 'undiagnosed signs'. Undiagnosed signs may indicate that the knowledge represented by the knowledge base is incomplete, i.e., an issue was found, but could not be root caused.

Similar to signs, diagnoses have severity, confidence, id, and args slots. The severity and confidence slots will typically be composites of the signs used to reach the diagnosis. For example, a diagnosis based on a sign reached with low confidence and another sign with a high confidence, should probably have a low to intermediate confidence value depending on the particular case.

4.2.1.3 Remedies

Remedies describe the step(s) to perform to resolve a diagnosis, such as change the permissions on a file or reboot a node. Remedies are specified using two optional sign slots, `remedy` and `remedy-args`. Similar to `id`, `remedy` corresponds to a message catalog key (`/opt/intel/clck/2017.0/kb/data/msg_en.xmc`) and `remedy-args` contains variable values to be inserted into the message catalog string. If the `remedy` slot is empty, then no remedy is displayed.

4.2.2 Basic Implementation

4.2.2.1 Classes

CLIPS classes are roughly analogous to C structures or C++ classes. 'Slots' are to member variables as classes are to C structures. A slot typically has some attributes, or 'facets', defined, such as the type, default value, etc. (see the CLIPS documentation for more information on facets). The slots are populated with information from the database (via the Intel® Cluster Checker connector component, see Chapter 5).

The class definition for the `DUCK` example follows and can also be found at `src/kb/classes/duck.clp` in the "SDK Duck Sample" at `https://software.intel.com/en-us/product-code-samples?topic=20903`:

```
(defclass DUCK
  "This class corresponds to the 'duck' node rating tool."
  (is-a BASE_NODE BASE_TIMESTAMP DATABASE MULTISET)
  (role concrete)
  (pattern-match reactive)

  (slot count (type INTEGER) (default 1))
  (slot sound (type SYMBOL) (allowed-values honk quack)
              (default honk)))
```

In addition to the explicitly defined slots, the `DUCK` class inherits slots from it's base classes. For instance, the `node_id` slot, which corresponds to an unique node identifier, is inherited from `BASE_NODE` class. If the class represents a property of multiple nodes, such as the network performance between a pair of nodes, it would instead inherit from the `NODE_PAIR` or `BASE_CLUSTER` base classes (`/opt/intel/clck/2017.0/kb/core/cluster.clp`).

4.2.2.2 Rules

For each class, there is typically a corresponding rule file. For instance, the `DUCK` class is defined in the file `src/kb/classes/duck.clp` and the corresponding rules are defined in the file `src/kb/rules/duck.clp` in the "SDK Duck Sample" located at `https://software.intel.com/en-us/product-code-samples?topic=20903`. Based on the data contained in the instances, plus potentially other information such as the hardware configuration of a node, a rule creates one or more signs or diagnoses.

A CLIPS rule has a "left-hand side" (LHS) and a "right-hand side" (RHS), separated by the `=>` token. The LHS is the set of if/then conditions that describe when the rule should "fire". The RHS contains the action that should be performed when the LHS conditions are met. Typically the action is to create a sign or diagnosis.

4.2.2.3 Signs

Several varieties of signs are provided (`/opt/intel/clck/2017.0/kb/core/sign.clp`).

- `BOOLEAN_SIGN` represents quantities that are either true or false.

For example, a process either is in the zombie state or it is not.

- `COUNTER_SIGN` represents quantities that correspond to a count of something. For example, the number of network retries.
- `PERFORMANCE_SIGN` represents a measure of performance that are either normal, substandard, or invalid. For example, the measured floating point performance meets expectations for the hardware configuration, does not meet expectations, or is an invalid value (e.g., negative, or not a number).
- Finally, `GENERIC_SIGN` is a general sign that can be used if one of the preceding specialized sign classes is not appropriate.

4.2.3 Organization and Directory Structure

The knowledge base is divided into several sub-components.

- The `/opt/intel/clck/2017.0/kb/core` sub-directory contains the core data structures and message handlers used by the rest of the knowledge base. These files should typically not be modified.
- The diagnostic knowledge is split between the `/opt/intel/clck/2017.0/kb/classes` and `/opt/intel/clck/2017.0/kb/rules` sub-directories; class definitions are part of `/opt/intel/clck/2017.0/kb/classes` while the logic defining correct and incorrect cluster behavior is contained in the `/opt/intel/clck/2017.0/kb/rules` sub-directory.
- The `/opt/intel/clck/2017.0/kb/data` sub-directory contains lists of hardware components and their properties, as well as the catalog of messages printed by the `clck` program.
- Functions that extend the base CLIPS functionality can be put in the `/opt/intel/clck/2017.0/kb/functions` sub-directory.

Tip: The lists of hardware components do not contain every SKU. If the observed hardware is not in the database, then some rules may not applicable and will not fire.

Each sub-directory has a file named `load.clp`. This file loads the rest of the files in the same sub-directory. If, for example, a new rules file is added, then it needs to be added to `/opt/intel/clck/2017.0/kb/rules/load.clp` to be enabled.

Finally, the file named `/opt/intel/clck/2017.0/kb/clck3.clp` in the top level knowledge base directory loads the `load.clp` file in each sub-directory. While this file should not typically be modified, commands can be added to this file to help debug the knowledge base.

4.3 Automatically Created Objects

A `NODE` object is automatically created for each node being checked. Each `NODE` object contains slots for the node architecture, roles, and subcluster membership. These slots may be used to restrict a rule to a particular type of node.

Tip: Data objects may be created for all the nodes in the database while only a subset of nodes may be checked. Therefore, it is recommended to always use a `NODE` condition on the left hand side of a rule to ensure that only the desired nodes are checked.

A single instance of the `CONFIG` class named `[config]` is automatically created and contains the input configuration parameters. The instance

name [config] is reserved for this purpose and no other instances should use this name. This instance may be used to make the behavior of a rule user configurable. See Section 4.4 for more information about how to write configurable rules.

4.4 Configurability

The CONFIG class contains all user configurable options and is defined in /opt/intel/clck/2017.0/kb/core/config.clp. A single instance of this class always exists with this reserved name. This class can be extended by adding new slots.

The slots of the CONFIG class form a global namespace, so slot names should be chosen with that consideration.

A simplified definition of the CONFIG class is as follows -

```
(defclass CONFIG
  (is-a USER)
  (role concrete)
  (pattern-match reactive)

  ; The list of checks to be performed.  One to one mapping
  ; with connector extensions of the same name.
  (multislot clck-checks (type SYMBOL)
             (default (create$ all_to_all cpu dgemm environment
                               ...))

  ; The checking mode.  Compliance checks compliance to the
  ; Intel(R) Cluster Ready architecture specification.  Health
  ; checks the functionality of the cluster.  Certification
  ; is essentially the union of compliance and health.
  (slot clck-mode (type SYMBOL)
        (allowed-values certification compliance health)
        (default health))

  ; The maximum allowable age of a data point, in seconds,
  ; before a data point is considered "too old".  The
  ; default is 1 week.
  (slot data-age-threshold (type NUMBER) (default 604800))

  ...
```

To use the CONFIG class, a corresponding rule would add a single condition to the left hand side:

```
(defrule duck-data-is-too-old
  "Identify instances where the most recent DUCK data should be
   considered too old."
  ; IF the mode is health and the 'duck' check is enabled
  (object (is-a CONFIG) (name [config]) (clck-mode health)
          (clck-checks $? duck $?)
          (data-age-threshold ?age-threshold))
  ...
```

The values of the CONFIG slots should always have defaults, and are

configurable via the XML file used to configure Intel® Cluster Checker (located in `/opt/intel/clck/2017.0/etc/clck.xml` by default).

The following construct can be used to set values for single slot variables.

```
<configuration>
  <analyzer>
    <config>
      <clck-mode>compliance</clck-mode>
    </config>
  </analyzer>
</configuration>
```

The following construct can be used to set values for multislot variables.

```
<configuration>
  <analyzer>
    <config>
      <clck-checks>
        <entry>PATTERN1</entry>
        <entry>PATTERN2</entry>
      </clck-checks>
    </config>
  </analyzer>
</configuration>
```

4.5 Example

This section steps through the complete `DUCK` knowledge base example. The source files are provided online in the "SDK Duck Sample" located at `https://software.intel.com/en-us/product-code-samples?topic=20903`, specifically in the folder `src/kb`.

4.5.1 Class Definition

Recall that the `duck` command rates nodes on a scale from 1 to 5 "quacks", or if there is an error during the evaluation, "honks" instead of "quacks". So the key data elements that need to be included in the knowledge base are a node identifier, the sound ("quack" or "honk"), and the number of times the sound is repeated. The following is an example CLIPS class definition that includes all of these elements. In an actual distribution, it would be added to the knowledge base as `/opt/intel/clck/2017.0/kb/classes/duck.clp`.

```
(defclass DUCK
  "This class corresponds to the 'duck' node rating tool."
  (is-a BASE_NODE BASE_TIMESTAMP DATABASE MULTISET)
  (role concrete)
  (pattern-match reactive)

  (slot count (type INTEGER) (default 1))
  (slot sound (type SYMBOL) (allowed-values honk quack)
              (default honk)))
```

The node_id slot is inherited from the BASE_NODE class, the row-id slot is inherited from the DATABASE class, and the timestamp slot is inherited from the BASE_TIMESTAMP class. The MULTISET inheritance will be described with the uniformity rule.

With the class defined, the connector can now create instances based on the content of the database (see Chapter 5). Rules can now be defined to check the output.

4.5.2 Rules

4.5.2.1 Rule 1: Missing good output

In this example, the first rule creates a sign whenever the number of quacks is less than 3. In an actual distribution, the rule would be added to knowledge base as /opt/intel/clck/2017.0/kb/rules/duck.clp.

```
(defrule duck-less-than-three-quacks
  "Create a sign whenever the number of 'quacks' is less than
   3."
  ; IF the mode is health and the 'duck' check is enabled
  (object (is-a CONFIG) (name [config]) (clck-mode health)
          (clck-checks $? duck $?))
  ; AND a node instance with the role 'compute' or 'enhanced'
  ; exists
  (object (is-a NODE) (node_id ?n)
          (role $?role&:(member$ compute ?role)
                        |:(member$ enhanced ?role)))
  ; AND an instance of the DUCK class exists for a node with
  ; the same node_id and with the sound 'quack'
  ?o <- (object (is-a DUCK) (count ?c) (node_id ?n)
                (sound quack))
  ; AND the number of quacks is less than 3
  (test (< ?c 3))
  =>
  ; THEN create a sign
  (make-instance of COUNTER_SIGN (node_id ?n)
                 (confidence 90) (severity 50)
                 (source ?o) (state observed) (value ?c)
                 (id "duck-less-than-three-quacks")
                 (args (create$ ?c))))
```

The LHS of this rule steps through a series of conditions.

1. An instance of the CONFIG class with the name [config] must exist with the clck-mode slot set to health and the clck-checks slot containing duck. In other words, only fire this rule if the “duck” check is enabled and the checking mode is “health”.

2. A NODE object must exist where the role slot contains either “compute” or “enhanced”. In other words, only fire this rule for compute / enhanced nodes. As a side effect, the ?n variable is populated with the id of the node.

3. A DUCK object must exist where the sound is “quack” and the node_id slot is same as the ?n value found in the prior condition.

In other words, only fire this rule for nodes with both a `NODE` object and a `DUCK` object. As a side effect, set the `?c` variable is populated with the number of "quacks".

4. The number of quacks, `?c`, must be less than 3.

Only if all four of these conditions are met will the rule fire and execute the action on the right hand side. The rule is automatically evaluated by the inference engine for all possible combinations of objects, i.e., each node is checked by this single rule.

The confidence and severity levels are arbitrary and a more sophisticated rule might scale them depending on the number of "quacks", e.g., 1 "quack" might have a severity level of 75 while 2 "quacks" has a severity level of 50. See Tables 4.1 and 4.2 for guidance on setting the confidence and severity levels.

A message catalog entry with the key "duck-less-than-three-quacks" would be added to `/opt/intel/clck/2017.0/kb/data/msg_en.xmc` in an actual distribution. An example message catalog entry is provided online in the "SDK Duck Sample" at `https://software.intel.com/en-us/product-code-samples?topic=20903`, located at `src/kb/data/msg_en.xmc`.

4.5.2.2 Rule 2: Error case

A second rule should be added for the case where the duck honks, indicating a serious error. The overall construction of the rule is similar to the previous rule.

```
(defrule duck-honking
  "If the duck honks like a goose, something serious has
    happened."
  ; IF the mode is health and the 'duck' check is enabled
  (object (is-a CONFIG) (name [config]) (clck-mode health)
          (clck-checks $? duck $?))
  ; AND a node instance with the role 'compute' or 'enhanced'
  ; exists
  (object (is-a NODE) (node_id ?n)
          (role $?role&:(member$ compute ?role)
                        |:(member$ enhanced ?role)))
  ; AND an instance of the DUCK class exists for a node with
  ; the same node_id and with the sound 'honk'
  ?o <- (object (is-a DUCK) (node_id ?n) (sound honk))
  =>
  ; THEN create a sign
  (make-instance of BOOLEAN_SIGN (node_id ?n)
                 (confidence 100) (severity 100)
                 (source ?o) (state observed) (value TRUE)
                 (id "duck-honking")))
```

As above, a message catalog entry with the key "duck-honking" should be added.

4.5.2.3 Rule 3: Uniformity

Finally, a rule might be added to verify that all nodes have the same "quack" rating.

Intel® Cluster Checker 2.x transition note: A "reference" node was arbitrary picked and every other node was compared to it. This approach suffers from the problem of selecting a suitable "reference" node. For example, if 99 out of 100 nodes are the same, and the one different node is used for the reference, then assuming the 99 are actually correct, many false positives are reported.

Usually the question of uniformity can be sufficiently answered by determining what fraction of nodes have the same / different value as a particular node. This approach avoids the combinatorial explosion of comparing every node to every other node and also avoids the problems associated with choosing a "reference" node. The `MULTISET` class is provided for determining uniformity. A multiset is similar to a set except it is a key / value pair where the value is the number of elements with the same key, e.g., the set {a, a, a, b} corresponds to the multiset {a:3, b:1}.

The `DUCK` class inherits from the `MULTISET` class. The `init` message-handler, roughly analogous to a C++ constructor, must be added to automatically insert the key / value pair into the multiset when each `DUCK` instance is created:

```
(defmessage-handler DUCK init after ()
  "Add MULTISET key / value pairs.  Skip non-quacks."
  (if (eq ?self:sound quack) then
    (send ?self add (send ?self multiset-key) ?self:count)))

(defmessage-handler DUCK multiset-key ()
  "Generate a distinct key for each node architecture, role,
   and subcluster combination."
  ; defaults
  (bind ?architecture x86_64)
  (bind ?role compute)
  (bind ?subcluster default)

  (bind ?ins (find-instance ((?n NODE)) (eq ?n:node_id
                                            ?self:node_id)))
  (if (= (length ?ins) 1) then
    (bind ?i (nth$ 1 ?ins))
    (bind ?architecture (send ?i get-architecture))
    (bind ?subcluster (send ?i get-subcluster))
    (if (member$ compute (send ?i get-role)) then
      (bind ?role compute)
      else (if (member$ enhanced (send ?i get-role)) then
             (bind ?role enhanced))))

  (bind ?key (sym-cat (class ?self) + ?subcluster + ?role
                      + ?architecture))
  (return ?key))
```

The `multiset-key` message handler creates distinct keys for each subcluster, node architecture, and node role. This is done to avoid comparing fundamentally different nodes, e.g., do not compare compute nodes to storage nodes.

The uniformity rule is:

```
(defrule duck-quack-count-is-not-consistent
  "Create a sign whenever the number of 'quacks' is not
   consistent."
  ; IF the mode is health and the 'duck' check is enabled
```

```
(object (is-a CONFIG) (name [config]) (clck-mode health)
        (clck-checks $? duck $?))
; AND a node instance with the role 'compute' or 'enhanced'
; exists
(object (is-a NODE) (node_id ?n)
        (role $?role&:(member$ compute ?role)
        |:(member$ enhanced ?role)))
; AND an instance of the DUCK class exists for a node with
; the same node_id and with the sound 'quack'
?o <- (object (is-a DUCK) (node_id ?n) (count ?c)
              (multiset_control TRUE) (sound quack))
; AND the fraction of nodes with the same quack count is
; less than 0.9
(test (< (send ?o fraction (send ?o multiset-key) ?c) 0.9))
=>
(bind ?key (send ?o multiset-key))
(bind ?fraction (- 1 (send ?o fraction ?key ?c)))
(make-instance of BOOLEAN_SIGN (node_id ?n)
               (confidence (* 100 ?fraction)) (severity 80)
               (state observed) (source ?o) (value TRUE)
               (id "duck-quack-count-is-not-consistent")
               (args (create$ (* 100
                                 (send ?o fraction ?key ?c))
                              ?c))))
```

The `(multiset_control TRUE)` condition appears in this rule to guarantee that all values have been added to the multiset before attempting to activate the rule. It should be used in all rules that rely on a multiset value.

The final LHS condition decides that if at least 90% of nodes have the same value, then it is actually correct. This is an arbitrary threshold to try to minimize the number of false positives that get reported.

The RHS creates a temporary variable `?fraction` that corresponds to the fraction of nodes that have a different number of "quacks". The confidence level scales with the fraction, i.e., the more nodes with a different value, the more likely that the outlier is actually an outlier.

4.5.2.4 Rule 4: Diagnosis

The `duck` diagnostic tool does not lend itself to diagnosis. The "quack" rating scale is unambiguous, but is a closely held trade secret by Waterfowl Industries and additional information such as why a node rated 2 "quacks" instead of 3 or the duck honked is not provided.

Diagnoses are typically made by combining one or more signs. For example, consider the combination of the proverbial "black swan" sign and the "duck-honking" sign to produce the diagnosis that the duck is honking because it is actually a black swan:

For a digression on black swans, see `https://en.wikipedia.org/wiki/Black_swan_theory`.

```
(defrule duck-duck-swan
  "Diagnose the root cause of the honking duck."
  ; IF the mode is health and the 'duck' check is enabled
  (object (is-a CONFIG) (name [config]) (clck-mode health)
          (clck-checks $? duck $?))
  ; AND a node instance with the role 'compute' or 'enhanced'
```

```
; exists
(object (is-a NODE) (node_id ?n)
        (role $?role&:(member$ compute ?role)
                      |:(member$ enhanced ?role)))
; AND a “”duck-honking sign exists for a node with the
; same node_id
?s1 <- (object (is-a SIGN) (node_id ?n) (id "duck-honking"))
; AND a "black-swan" sign exists for a node with the same
; node_id
?s2 <- (object (is-a SIGN) (node_id ?n) (id "black-swan"))
=>
; THEN create a DIAGNOSIS and mark the signs as diagnosed
(send ?s1 put-state diagnosed)
(send ?s2 put-state diagnosed)
(make-instance of DIAGNOSIS (node_id ?n)
               (confidence 20) (severity 100)
               (source (create$ (send ?s1 get-source)
                                (send ?s2 get-source)))
               (id "duck-duck-swan")
               (remedy "duck-duck-swan-remedy")))
```

Note that this rule is not part of the included sample files.

4.6 Developing with CLIPS

4.6.1 Editor

There is a not a specialized CLIPS editor. Any text editor may be used, although one with a LISP mode is recommended. Both vim and Emacs have built-in LISP modes.

- Emacs: `<M-x> lisp-mode`
- vim: `:set lisp`

Add the following to $HOME/.emacs to automatically open all files with the .clp extension in lisp-mode:

```
;;; CLIPS code
(defalias 'clips-mode 'lisp-mode)
(setq auto-mode-alist (cons '("\\.clp$" . clips-mode)
    auto-mode-alist))
```

4.6.2 Style

Coding style, as usual, is largely a matter of personal preference. The following style guidelines are recommended:

- Do not exceed 80 characters per line
- Generally use alphabetical order for any list of items
- Use all lower case, except for class names

- Uses dashes rather than underscores or CamelCase
- Document all classes, functions, message-handlers, rules, etc. using the CLIPS comment field rather than ';' style comments
- Use the same value for the rule name and sign / diagnosis id slot

4.6.3 Debugging and Profiling

CLIPS includes several techniques to help better understand what it is doing.

One of the most debugging useful techniques is the "watch" capability (see section 13.2.3 in the CLIPS Basic Programming Guide).

CLIPS also includes a good profiling capability (see section 13.16 in the CLIPS Basic Programming Guide).

Additional debug and/or profile statements may be included in `/opt/intel/clck/2017.0/kb/clck3.clp`, in which case, additional output will be displayed when running an analysis.

Chapter 5

Connector

5.1 Overview

The Intel® Cluster Checker connector is a C++ framework to bridge the database and the knowledge base. Conceptually, the connector's functionality is as follows:

1. Read data from the database.
2. Transform the data, e.g., extract the relevant information from the raw, unstructured database content via regular expressions.
3. Create CLIPS instances using the transformed data.

Extensions, in the form of shared libraries, plug into the connector framework, to perform these functions. Typically, there will be one connector extension per data provider / CLIPS class, but this may not always be the case.

The interfaces described in this chapter are located in `/opt/intel/clck/2017.0/include/connector`.

5.2 Extensions

Extensions are implemented through the `Extension` and `Transform` classes. Conceptually, the purpose of `Transform` class is to read in data from a source, process the data, and then send the data to an output. The `Extension` class is specialized to use the Intel® Cluster Checker SQLite or ODBC database sources as the input and create CLIPS instances as the output.

The connector framework performs the following actions on each extension:

1. Load the extension shared library using `dlopen()`.
2. Call the constructor of the extension.

3. Run the data input method `parse()`.
4. Call the destructor the extension.
5. Unload the shared library using `dlclose()`.

5.2.1 Transform Class Members

Methods:

`parse()`

Pull data from the input source, i.e., the database, transform it, and call route(). Parse executes data manipulation and extrapolation specific to every extension.

`set_header()`

Define the CLIPS slots to be populated. The order should match the order used in `route()`.

```
set_header({"node_id", "timestamp", "count", "sound"});
```

`set_name()`

```
set_name("duck");
```

Sets the internal name of the extension. This name should match the name of the shared library and is also used in `/opt/intel/clck/2017.0/etc/clck.xml` to configure the extension to be loaded.

`route()`

Send data to the output sink, i.e., create a CLIPS instance. The order should match the order used in `set_header()`.

```
route({rows[i].hostname, rows[i].timestamp, variable1,
               "quack"});
```

Variables:

`DbRead db_read`

Database reader instance

`void* clips_env`

Pointer to the CLIPS knowledge base environment

5.3 Database Interface

The database interface class derives two classes: SQLite and ODBC. The `db_read` base class is a general interface for reading data from the database. The following methods are provided for accessing the database, and are defined in `/opt/intel/clck/2017.0/include/connector/db_read.h`. The methods query the database view clck_1. If the database is provided by the user, the clck_1 view must be created manually (see Appendix A for the database view).

```
bool select(std::string provider_name, NsDb::Rows& rows,
            std::string where_clause="");
```

The database rows resulting from the query are appended to the vector of rows provided by the caller in the second argument. When this function is called, an SQL query of the following form is constructed and executed.

```
SELECT * FROM clck_1 WHERE provider=<provider_name>
   AND <where_clause>
```

A more general select method is also available.

```
bool select(const std::string query, NsDb::Rows& rows,
            const std::map<std::string, int>& columns);
```

As before, the database rows resulting from the query are appended to the vector of rows provided by the caller in the second argument. The difference is that the first argument may be any valid SQL SELECT query. Since not all database columns may be returned by the query, the third argument is a map of column names and their order in the SELECT query.

For example, the following would select the latest rows for each node corresponding to the `duck` provider. (see Appendix A for the database view).

```
db_read.select("select * from clck_1 a inner join (select
    Hostname, max(Timestamp) as Timestamp, Provider from clck_1
     where Provider= 'duck' group by Hostname, Provider) b on a
    .Provider=b.Provider and a.Timestamp=b.Timestamp and a.
    Hostname=b.Hostname", rows);
```

A nearly equivalent set of data can be obtained using the following function.

```
bool get_latest_rows_provider(std::string provider,
                              NsDb::Rows& rows);
```

Unlike the general `select()` function, `get_latest_rows_provider()` populates all of the database columns rather than just the specified subset.

5.4 Knowledge Base and CLIPS Interface

The connector makes use of the CLIPS C API for interacting with the knowledge base (`http://clipsrules.sourceforge.net/documentation/v630/apg.pdf`).

5.4.1 Creating CLIPS Class Instances

Connector extensions populate the knowledge base by creating CLIPS instances. The format of the data expected by the knowledge base, i.e., the CLIPS slots, is defined by the corresponding knowledge base class.

5.4.2 Parsing database output

Once the data is read from the database it is available for processing. Any method available to C++ can be used to filter and transform the data into the format expected by the knowledge base, e.g., regular expression.

5.4.3 Handling Parse Errors

Tip: An exception should be thrown only in the most exceptional circumstances when no recovery is possible.

Parse errors can occur when a connector extension reads unexpected or invalid data from the database. If the error is critical to the operation to the entire front-end system, then it is appropriate to log an error and throw an exception. In the case of non-critical errors, then the parser should log a warning message, ignore the offending row in the database, and continue processing the rest of the rows.

5.5 Building Extensions

Extensions are shared libraries and need to be built as such.

Sample extensions are located at `src/connector` and a sample Makefile at `Makefile` in the "SDK Duck Sample" at `https://software.intel.com/en-us/product-code-samples?topic=20903`.

GCC* 4.9 or later is required to build extensions. The Intel® C++ Compiler 15.0 or later may also be used, but GCC* 4.9 or later is still required.

Tip: Build extensions with the oldest Linux distribution / glibc version that is intended to be supported. Otherwise, the extension may include symbols that are not be available in older glibc versions and consequently will not run on older Linux distributions.

Intel® Cluster Checker uses features from C++11, therefore the command line option `-std=c++11` is required to build connector extensions.

5.6 Loading Extensions

The list of connector extensions to load are defined by the `clck-checks` section of `/opt/intel/clck/2017.0/etc/clck.xml`. New extensions should be added to list of extensions already present.

The basename of the extension should match the internal extension name assigned by `set_name()`. This name is the value that should be added to the list of checks.

5.7 Example

A complete, fully functional connector extension that transforms the output of the `duck` provider into instances of `DUCK` CLIPS class is located at `src/connector` in the "SDK Duck Sample" at `https://software.intel.com/en-us/product-code-samples?topic=20903`.

Appendix A

Database Schema

The database consists of a single SQL view named `clck_1`. The view contains columns described in Table A.1.

Name	SQLite* Type	Description
rowid	INTEGER	Unique row ID
Provider	TEXT	Data provider name
Hostname	TEXT	Hostname of the node where the data provider ran
num_nodes	INTEGER	Number of nodes used by the data provider
node_names	TEXT	Comma separated list of nodes used by the data provider. Empty if num_nodes = 1.
Exit_status	INTEGER	Exit status of the data provider
Timestamp	INTEGER	Timestamp when the data provider started (seconds since the UNIX epoch)
Duration	REAL	Data provider walltime (seconds)
Encoding	INTEGER	Encoding format of the STDOUT and STDERR columns. 0 = no encoding, 1 = base64 encoding.
STDOUT	TEXT	Data provider standard output
STDERR	TEXT	Data provider standard error
OptionID	TEXT	The ID of the option set with which the provider was run
Version	INTEGER	Output format version of the data provider
Username	TEXT	Username of the user who ran the data provider

Table A.1 – Database schema.

The Intel® Cluster Checker database is a standard SQLite* database and any SQLite* compatible tool may be used to browse the database contents. In addition, the `clckdb` utility is provided with Intel® Cluster Checker (see `clckdb -h` for more information).