CIMPLE: An Embeddable CIM Provider Engine

Michael E. Brasher
mbrasher@inovadevelopment.com
Karl Schopmeyer
kschopmeyer@inovadevelopment.com

March 20, 2006

Abstract

CIMPLE is a CIM provider engine. It offers a complete environment for developing providers and an interface for invoking them from other applications. CIMPLE is used to: (1) build providers that work with a variety CIM servers, and (2) provide a foundation for implementing CIM-based standards, such as WBEM, SMASH, WSDM, and WS Management. CIMPLE has three major advantages over conventional provider technologies. First, it substantially reduces the provider development lifecycle due to several factors discussed below. Second, providers can be written once for a wide range of CIM servers. Third, CIMPLE has a very small footprint. This paper explains what problems CIMPLE solves and how it solves them.

1 Introduction

CIMPLE was designed to address four formidable problems encountered by CIM adopters. First, provider development is too difficult, leading to long development lifecycles and low-quality deliverables. Second, developing a distinct set of providers for each targeted CIM server is often unavoidable. Third, conventional provider technology produces providers that are too heavy, especially for embedded systems. Finally, there is a shortage of reusable CIM infrastructure for implementing emerging CIM-based standards like SMASH, WSDM, and WS Management. The following sections explain how CIMPLE addresses each of these problems.

2 Reducing Provider Complexity

CIMPLE substantially reduces the complexity of developing providers by (1) transforming MOF class definitions into concrete C++ classes, sometimes referred to as *first-class objects*, (2) automatically generating the provider skeleton source code, (3) reducing the requisite number of provider operations, (4) providing tools to automatically load, validate, and register providers, (5) equipping the developer with provider development patterns. Each of these techniques is discussed below.

The first technique is *class generation*. CIMPLE provides a tool called `genclass` that generates a concrete C++ class corresponding to the MOF class. In this way, providers work with real C++ classes rather than the dynamic abstractions used in conventional provider interfaces. This technique offers several benefits. First, it is much easier to work with concrete classes rather than with dynamic abstractions. Second, concrete classes enable the compiler to detect many errors at compile time that would otherwise slip into run time. Third, concrete classes are more efficient. For example, accessing a property has a time complexity of O(1) rather than O(n), associated with dynamic abstractions. Finally, concrete classes produce much less object code. To illustrate class generation, consider the following C++ class generated from the corresponding MOF class definition.

```
class Fan
{
  Property<String> DeviceID;
  Property<uint64> Speed;
  Property<uint64> DesiredSpeed;
};
```

The second technique is *provider skeleton generation*. The `genprov` tool generates a provider skeleton from a MOF class definition. The tool generates stubs corresponding to each of the provider operations (e.g., enum-class, enum-instances). Default implementations are generated for each operation and for many providers it is only necessary to implement one or two stubs. Additionally, `genprov` produces a stub for each method found in the MOF class definition. Each parameter in the MOF method definition becomes a real parameter in the generated stub, which relieves the developer from having to process parameters manually.

The third technique reduces the number of operations the developer must implement. Besides generating default implementation automatically, CIMPLE eliminates the need to implement some operations altogether. For example, enum-instance-names is obsolete since it is a special cases of enum-instances with additional qualification. Some operations are optional. For example, by

opting not to implement get-instance, it is satisfied in terms of enum-instance (suitable for providers with only a handful of instances). The table below indicates which operations are optional, required, and obsolete (note that operations marked required followed by an asterisk are only required under some circumstances).

Operation	Comment
get-instance	optional
get-instance-name	obsolete
enum-instances	required
enum-instance-names	obsolete
get-property	obsolete
set-property	obsolete
create-instance	required*
modify-instance	required*
delete-instance	required*
associators	obsolete
associator-names	optional
references	optional
reference-names	obsolete
enabled-indications	required*
disable-indications	required*

The fourth way CIMPLE reduces provider complexity is by providing tools for loading, validating, and registering providers automatically. CIMPLE provides two tools: `testprov` and `regmod`.

`Testprov` automatically loads a provider and tests some of its operations, insofar as that is possible. For example, `testprov` checks to see if each instance obtained with enum-instances can also be obtained with get-instance. It also attempts to delete and recreate an instance, if supported.

`Regmod` (for OpenPegasus only in this release) automatically registers all providers contained in a module (library) with the CIM server. This entails (1) validating the CIM server environment, (2) checking that each provider can be loaded and unloaded, (3) creating registration instances for each provider and (4) adding each provider's generated class to the CIM repository.

The fifth way CIMPLE reduces complexity is by equipping the developer with provider-development patterns. CIMPLE currently support two main patterns: the preload provider and the CGI provider.

The *preload provider pattern* helps develop read-only providers that preload their instances when they start up. When using this pattern it is only necessary to implement one method: the `preload()` method, which loads instances into memory once.

The *CGI provider pattern* is similar to CGI programs used with HTTP servers. The CGI provider is any program that effectively implements the enum-instances operation by printing instances to standard output. When a request is received, the program is spawned and the instances are read from standard output. Note that get-instance is implemented by sifting through the returned instances for a match. The following bash script is a complete provider that provides the `User` class by scanning the `/etc/passwd` file.

```
#!/bin/sh
SCRIPT=`cat<<END
{
  print "name="\\$1;
  print "uid="\\$3;
  print "gid="\\$4;
  print "fullName="\\$5;
  print "homeDir="\\$6;
  print "shellProgram="\\$7;
  print "";
}
END
`
awk -F ":" "$SCRIPT" /etc/passwd
```

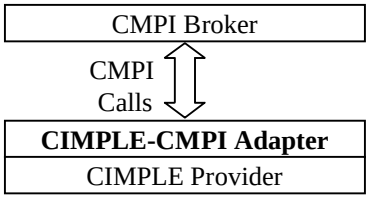
Each property is formatted as a simple name-value pair separated by an equal sign. Properties appear one per line and instances are comprised of one or more properties. Instances are separated by a double linefeed sequence.

3 Writing the Provider Once

Providers developed with CIMPLE work with a variety of CIM server implementations, including OpenPegasus as well as CMPI-compliant servers. This makes it possible to write just one provider that interoperates with a wide range of CIM servers.

Of course the same range of interoperability can be achieved with CMPI, but recall that CIMPLE offers many additional benefits (e.g., concrete classes, automatic provider generation, operation reduction, automated validation and registration, and provider patterns).

It would be a mistake to think that CIMPLE is in competition to CMPI. On the contrary, CIMPLE complements CMPI by providing a safer and more convenient interface for developing CMPI-compliant providers. To a CMPI broker, CIMPLE providers and CMPI providers are indistinguishable. Any CIMPLE provider can masquerade as a CMPI provider using a thin *adapter* depicted below.



In addition to the CMPI interface, CIMPLE has a custom *provider manager* that enables CIMPLE providers to work with the OpenPegasus server in installations lacking CMPI support.

4 Shrinking the Provider

Before CIMPLE, it was generally believed that lightweight providers could only be written in C. Conventional C++ provider interfaces boosted productivity and made provider development safe and convenient. But at the same time, they bloated providers. In OpenPegasus, for example, the average object size for a provider is 75 kilobytes (on Linux x86). Although this is acceptable for servers and workstations, it is too large for many embedded systems.

Surprisingly, CIMPLE providers are written entirely in C++ but average only 5 kilobytes a piece (on Linux x86). This result is due to two main factors. (1) CIMPLE employs special C++ techniques that grew out of research and years of experience using C++ on embedded systems. (2) The use concrete classes (rather than dynamic abstractions) vastly reduces source code complexity and hence object-size.

To achieve the first factor, one might imagine that CIMPLE makes very restricted use of C++. To a small degree this is true; for example, CIMPLE avoids exceptions and RTTI (run-time-type-identification). But CIMPLE also provides a template-based array class as well as a string class. Where convenience and safety are concerned, CIMPLE exploits C++. Elaborating on these C++ space-saving techniques is the subject for another (long overdue) white paper.

The second factor, that concrete classes reduce object size, may not be obvious without an example. To illustrate this, we show how to obtain the value of a property using CIMPLE, OpenPegasus, and CMPI. We start with CIMPLE.

```
Instance* inst;
…
uint64 speed = fan->Speed.value;
```

Next, we perform the same task with OpenPegasus.

```
CIMInstance inst;
…
Uint64 speed;
try
{
  CIMProperty prop = inst.getProperty(
      inst.findProperty("Speed"));
  prop.getValue().get(speed);
}
catch(…)
{
  // Handle error.
}
```

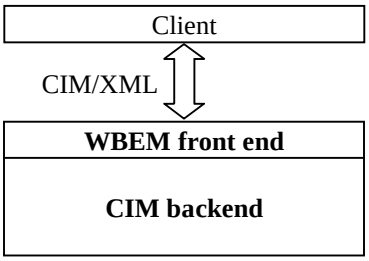
And finally, we do the same thing using CMPI.

```
CMPIInstance* inst;
CMPIStatus status;
CMPIData data;
Uint32 speed;
…
data = CMGetProperty(
  inst, "Speed", &status);

if (status.rc != CMPI_STATUS_OK)
{
  /* Handle error */
}

if (data.type != CMPI_uint64)
{
  /* Handle error */
}

speed = data.value.uint64;
```

It is obvious that the CIMPLE version, by using a concrete class, produces the least object code by far. And concrete classes offer other benefits as well, including (1) reduced development effort, (2) compile-time type checking, and (3) reduction of run-time errors.

5 Implementing Emerging Standards

The CIMOM (CIM Object Manager) implements two standards: CIM and WBEM. CIM defines the underlying CIM infrastructure whereas WBEM defines the protocol for accessing that infrastructure. Viewed in this way, WBEM is the front end of the CIMOM implementation and CIM is the backend, as depicted below.



Since the advent of the CIMOM, other CIM-based standards have emerged, including SMASH, WSDM, and WS Management. Each of these emerging architectures needs a CIM backend but none needs a WBEM front end, since each defines its own alternative protocol.

Implementers of SMASH, WSDM, and WS Manage-

ment architectures must either reuse or reinvent most of the CIM backend. Unfortunately, it is infeasible to reuse the CIM backend of any existing CIM server, since there is no formal interface for doing so and there is no way to use it apart from the WBEM front end.

Fortunately, CIMPLE was designed to be embedded directly in other applications. It is ideal for building SMASH, WSDM, and WS Management implementations, as well as CIMOMs. It furnishes the developer with a *provider engine* for loading and managing providers and an interface for invoking them. The key functions supported by the interface are identified in the following table.

Function
`get_class()`
`get_instance()`
`enum_instance()`
`create_instance()`
`modify_instance()`
`delete_instance()`
`enum_associator_names()`
`enum_references()`
`enable_indications()`
`disable_indications()`
`invoke_method()`
`load_providers()`
`unload_providers()`
`get_instance()`

The CIMPLE provider engine has a tiny footprint of 50 kilobytes (on Linux x86) and as mentioned before, providers are typically around 5 kilobytes. This makes CIMPLE ideal for resource-constrained environments.

There are two main advantages to using CIMPLE to implement CIM-based standards. (1) It is unnecessary to build a very complex subsystem from scratch and (2) CIMPLE providers can be shared across implementations. For example, the same provider can be used within a CIMOM as well as a WSDM server.

6 Conclusion

This paper examined how CIMPLE addresses some of key problems faced by CIM adopters. These include (1) the difficulty of developing providers, (2) the lack of provider interoperability, (3) the excessive weight of providers, and (4) the lack of infrastructure for implementing emerging CIM-standards.

CIMPLE is part of an ongoing effort to simplify the adoption of CIM. We are now looking ahead at new ways to achieve this. We are working to answer several key questions. What are the key patterns for developing providers and how can they be automated? Can entire provider frameworks be automatically generated from profiles? Can the concept of concrete classes be applied to the CIM client?