INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Untersuchung heterogener Management-attribute von virtuellen Maschinen und Hypervisoren an den Beispielen Xen und VMware ESXi

Michael Kasch

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Untersuchung heterogener Management-attribute von virtuellen Maschinen und Hypervisoren an den Beispielen Xen und VMware ESXi

Michael Kasch

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Vitalian Danciu Dr. Nils gentschen Felde Dipl.-Inf. Martin Metzker
Abgabetermin:	15. März 2011

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 9. März 2011

. . . . . .
(Unterschrift des Kandidaten)

Zusammenfassung

Im Rahmen der zunehmenden Virtualisierung in Rechenzentren nehmen die Managementanforderungen immer weiter zu. Die herstellerspezifischen Werkzeuge können die Herausforderungen aber nur unzureichend behandeln und die Integration dieser in umfassende Managementprozesse ist schwierig. Als Vorraussetzung für die meisten Anforderungen wird umfangreiches Wissen über den Zustand der Managed Objects – in diesem Kontext der virtuellen Maschinen und Hypervisoren – und deren Attribute benötigt.

Das Ziel dieser Arbeit ist es, einen herstellerunabhängigen Attributsstamm zu schaffen, der als Grundlage für ein Informationsmodell dienen kann. Hierzu werden zwei Hypervisoren ausgewählt, deren Attribute – im Bereich virtuelle Maschinen und Hostsysteme – semantisch untersucht und katalogisiert werden.

Im weiteren Verlauf wird ein Schema zur Klassifikation von Attributen mehrerer Datenquellen anhand ihrer Abbildbarkeit aufeinander eingeführt. Dieses Klassifikationsschema wird auf die zwei vorher bestimmten Attributsmengen angewandt, wodurch eine abbildbare Schnittmenge mit normalisierter Semantik entsteht. Dieser generische Attributsstamm kann als Basis für ein Modell zum Management unterschiedlicher Virtualisierungslösungen dienen.

Im Rahmen einer prototypischen Implementierung wird die Funktionalität der Bibliothek *libvirt* um den Bezug dieser abgebildeten normalisierten Attribute der Schnittmenge für die beiden Hypervisoren Xen und VMware ESXi erweitert.

Diese Arbeit beschreibt einen Weg zur Schaffung eines gemeinsamen Attributsstamms und zeigt die Machbarkeit von der Klärung der Semantik von Attributen bis hin zur Implementierung dieses Attributsstamms in einer Managementbibliothek.

Inhaltsverzeichnis

1. Einleitung

Virtualisierung in Rechenzentren verspricht neben Kostenersparnissen auch Vereinfachungen beim Management. So können virtuelle Maschinen (VMs) zu Wartungszwecken oder aufgrund geänderter Ressourcenanforderungen im laufenden Betrieb von einem physischen Host auf einen anderen migriert werden. Allerdings werden durch Virtualisierung nicht nur Vorteile, sondern auch neue Herausforderungen für das Management geschaffen. Durch Paravirtualisierung oder Vollvirtualisierung eines Hostsystems entsteht mindestens eine zusätzliche Softwareschicht: der Hypervisor, der durch das Management zu berücksichtigen ist. Dieser emuliert an vielen Stellen die Hardware, die Gastbetriebssysteme zu nutzen glauben, bildet diese auf andere physische Hardware ab und sorgt für das Multiplexing zwischen virtuellen Maschinen, auf einem physischen Host. Durch Virtualisierung von Speicherkomponenten und Virtualisierung der Netze ergeben sich auch auf diesen Gebieten weitere neue Problemstellungen für das Management. Die folgende beispielhafte Aufstellung gibt einen Überblick über Herausforderungen aus den fünf funktionalen Managementbereichen (FCAPS):

- Im *Fault Management* entstehen durch die virtualisierte Hardwareschicht und durch virtuelle Netze, die meist nicht mit der physischen Netztopologie übereinstimmen, neue Fehlerquellen.
- Im *Configuration Management* muss die dynamische Zuordnung zwischen virtuellen Maschinen und Hostsystemen abgebildet werden, oder allgemeiner: zwischen virtueller und physischer Infrastruktur.
- Im *Accounting Management* werden neue Metriken benötigt. Bei Leistungszusicherungen und -messungen können nicht die Leistungsdaten eines physischen Hosts genutzt werden, da es sich bei Hostsystemen um gemeinsam genutzte Ressourcen handelt, die von virtuellen Maschinen unterschiedlicher Kunden genutzt werden können.
- Im *Performance Management* werden analog zum Accounting Management neue Metriken benötigt. Durch Virtualisierung werden neue Möglichkeiten geschaffen, beispielsweise zur Migration von virtuellen Maschinen, die beachtet werden müssen. Bei Überlastung eines Hosts besteht die Möglichkeit einen Teil oder alle darauf laufenden VMs zu migrieren, das aber andererseits einen vorübergehenden Anstieg der Netzauslastung zur Folge hat.
- Im *Security Management* muss sichergestellt werden, dass die virtuellen Maschinen voneinander abgeschottet sind, auch wenn sie auf demselben physischen Host laufen, um Interessenkonflikte zwischen Kunden, zu vermeiden. Auch müssen Angriffe auf den Hypervisor verhindert werden, da die Kontrolle über den Hypervisor Sicherheitsmechanismen der Gastbetriebssysteme aushebeln kann.

Eine darüber hinausgehende Aufstellung findet sich in [Dan09].

Momentaner Stand ist, dass über proprietäre Tools der einzelnen Hersteller von Virtualisierungslösungen nur einige dieser Aspekte behandelt werden können. Da durch generische

Zugriffsmechanismen auf die Managementfunktionalitäten Zeit und Geld eingespart werden können, ist es, langfristig gesehen, wünschenswert, alle diese Aspekte nicht herstellerabhängig, sondern durch umfassende Lösungen in Angriff zu nehmen. Ein Schritt in Richtung dieses Ziels ist ein gemeinsames Informationsmodell. Der Wunsch nach Herstellerunabhängigkeit entsteht aus unterschiedlichen Beweggründen.

Zum einen können auf Grund der Tatsache, dass unterschiedliche Hypervisoren unterschiedliche Stärken und Schwächen haben, in einem Rechenzentrum mehrere Virtualisierungslösungen gleichzeitig im Einsatz sein. Hierfür wäre ein herstellerunabhängiges Management eine deutliche Arbeitserleichterung, da der Arbeits- und Schulungsaufwand sinkt, wenn alle Managementoperationen unabhängig vom Hypervisor über eine gemeinsame Schnittstelle erfolgen können.

Zum anderen ist es notwendig das Management von virtuellen Maschinen in übergeordnete Managementprozesse und -lösungen zu integrieren. Eine Anpassung der hierzu nötigen Agenten an die einzelnen Hypervisoren erzeugt einen höheren Aufwand als die Nutzung einer generischen Schnittstelle. Aus diesem Grund wird die Entwicklung von geeigneten Managementlösungen gehemmt. Zu diesem Schluss gelangt auch [BSB+10]: ,,The development of appropriate management solutions is hindered by the various management interfaces of different hypervisors."

Um ein gemeinsames Informationsmodell zu schaffen, gibt es von der Distributed Management Task Force (DMTF) Ansätze im Rahmen des VMAN Standards das Common Information Model (CIM) durch virtualisierungsspezifische Attribute und Klassen zu erweitern, um Web Based Enterprise Management (WBEM) für die Virtualisieungslösungen zu ermöglichen. Auf der anderen Seite bieten die APIs der Hypervisoren unterschiedliche Funktionen und Informationen an, wobei selbst bei gleichen Attributsnamen nicht von gleicher Semantik der Attribute ausgegangen werden kann. Diese Attribute und Funktionen müssen auf das Modell der DMTF abgebildet werden. Aktuell unterstüzt kein Hypervisorhersteller die DMTF Standards korrekt. ,,There are DMTF schemas concerning the management of virtual machines, but no vendor actually implements them correctly." [BSB+10].

Auch bei VMware, die behaupten zu CIM kompatibel zu sein, ist die Unterstüzung nur in Teilen gegeben, da nur Eigenschaften von Hardwarekomponenten der angeschlossener Storage Devices exportiert werden, aber keine Funktionen zum Management der virtuellen Maschinen. ,,VMware ESX server provides information about hardware components or attached storage devices using WBEM, but no virtual machine management functions." [BSB+10]

Aufgrund der Komplexität und der nötigen Initiative der Hypervisorhersteller ist nicht mit einer zeitnahen und korrekten Unterstützung dieses Top-Down-Ansatzes durch die Virtualisierer auszugehen, da das Hauptproblem die Abbildung der Hypervisorattribute auf das CIM ist.

Zielsetzung

Diese Arbeit geht das Problem von unten nach oben an, indem für das Management nützliche Attribute von Hypervisoren und virtuellen Maschinen identifiziert werden. Die identifizierten Attribute werden auf ihre Abbildbarkeit zwischen den beiden Hypervisoren VMware ESXi und XEN überprüft, anhand der Abbildungen klassifiziert und anschließend normalisiert.

Dies hat das Ziel einen Attributsstamm aufzubauen, für den Abbildungsmöglichkeiten aus einer großen Zahl an Virtualisierungslösungen existieren. Aus diesem könnte als langfristiges Ziel ein Informationsmodell inklusive der nötigen Abbildungen zwischen den Hypervisoren

und dem Modell entwickelt werden.

Des weiteren soll eine Implementierung geschaffen werden, die hypervisorübergreifendes Management ermöglicht. Da das Hauptproblem in der unterschiedlichen Struktur der Managementinterfaces der einzelnen Hypervisoren liegt, kann diese wie in [BSB+10] vorgeschlagen, durch eine Schicht realisiert werden, die den Zugriff von den einzelnen Hypervisoren abstrahiert.

Struktur

In Kapitel 2 wird ein Überblick über die Bibliothek *libvirt* sowie den VMAN Standard gegeben, und eine Positionierung dieser Arbeit in dieses Umfeld vorgenommen. Um das Ziel dieser Arbeit umzusetzen, ist es erforderlich, die Schnittstellen der Hypervisoren zu betrachten, da die Attributswertbeschaffung bei den Hypervisoren jeweils verschiedene Ausprägungen besitzt (vgl. Abschnitt 3.1). Anschließend folgt eine Sammlung, und semantische Untersuchung der über diese Schnittstellen zur Verfügung gestellten Hypervisorattribute (vgl. Abschnitt 3.2). In Kapitel 4 werden die Attribute nach ihrer Abbildbarkeit klassifiziert, und somit eine Schnittmenge an generischen Attributen geschaffen. Abschließend wird ein Prototyp entwickelt, der diese Attribute zur Verfügung stellen kann (vgl. Kapitel 5) und ein Ausblick auf mögliche Folgeentwicklungen gegeben (vgl. Kapitel 6).

2. Verwandte Arbeiten

Auf dem Gebiet des hypervisorübergreifenden Managements sind vor allem zwei Projekte hervorzuheben. Zum einen libvirt aufgrund des großen Verbreitungsgrads dieser Bibliothek; zum anderen der VMAN Standard der Distributed Management Task Force (DMTF), da diese mit dem Common Information Model (CIM) einen wichtigen Standard auf dem Gebiet des Managements verteilter Systeme pflegen. In Abschnitt 2.3 wird eine Positionierung dieser Arbeit im Hinblick auf CIM und libvirt vorgenommen.

2.1. libvirt

Bei libvirt handelt es sich um eine Bibliothek zum Management virtueller Maschinen. Das Projekt wird hauptsächlich von RedHat gepflegt, ist unter der GNU Lesser General Public License verfügbar und bietet eine breite Unterstützung für unterschiedliche Virtualisierungslösungen. So werden unter anderem Xen, VMware ESXi und GSX, KVM, HyperV, und VirtualBox unterstützt.

Da der Fokus der Bibliothek auf Funktionen zum Management der virtuellen Maschinen liegt und weniger auf Attributen zur Beschreibung der physischen und virtuellen Infrastrukturen, gibt es diverse Funktionen zum Starten, Stoppen, Herunterfahren und auch Migrieren von virtuellen Maschinen. Die Zahl der abfragbaren Attribute ist gering. Auch ein dokumentiertes Datenmodell existiert nicht. Libvirt bietet eine Möglichkeit XML-Abbilder zur Beschreibung virtueller Maschinen zu erzeugen. Diese haben zwar eine gemeinsame Struktur, werden aber vom Hypervisortreiber erzeugt und sind nur durch denselben Treiber wieder interpretierbar. Hierdurch wird der Implementierungsaufwand niedrig gehalten, aber keine Interoperabilität zwischen den Hypervisoren hergestellt, sodass auch diese nicht als gemeinsame Basis für ein Informationsmodell dienen können.

Die Bibliothek lädt zum Aufbau einer Verbindung zu den Hypervisoren einen hypervisorabhängigen zentralen Treiber, welcher die Kommunikation mit den Managementschnittstellen der Hypervisoren abwickelt. Neben diesen zentralen Treibern existieren noch unterschiedliche Zusatztreiber für spezialisierte Aufgaben. Diese können zusätzlich geladen werden und setzen zum Teil auf den Systemen laufende Dienste voraus.

Die Unterstützung der einzelnen Funktionen der Bibliothek ist bei den verschiedenen Virtualisierungslösungen unterschiedlich ausgeprägt. Die aktuelle Driver Support Matrix kann unter `http://libvirt.org/hvsupport.html` eingesehen werden. Die umfassendste Unterstützung ist für Xen gegeben, allerdings erfolgt hier der Zugriff noch nicht mittels der 2006 eingeführten XenApi, sondern über die vorherigen Managementschnittstellen mittels des XenD UNIX Socket Interface. Die Unterstützung der XenApi ist in Arbeit, allerdings hemmen einige Einschränkungen den Einsatz dieser Schnittstelle (vgl. Abschnitt 5.2). Im Rahmen dieser Arbeit werden ein paar dieser Hindernisse beseitigt.

Trotzdem ist libvirt momentan die verbreitetste Bibliothek, wenn es darum geht hypervisorübergreifende Managementfunktionen bereit zu stellen. Sie bietet viele für darauf aufset-

zende Managementapplikationen nützliche Funktionalitäten, die mit unterschiedlichen Hypervisoren funktionieren. Durch den Einsatz von libvirt wird der Zugriff auf Hypervisoren transparent. Dies setzt zwar auch die Unterstützung durch die internen Hypervisortreiber voraus, die, wie vorher schon angemerkt, die Funktionalität der libvirt unterschiedlich gut darstellen. Im Allgemeinen ist die Unterstützung aber ausreichend. Der Schwerpunkt der API liegt dabei auf dem konkreten Management des Betriebs und nicht darin, die funktionalen OSI-Managementbereiche abzudecken.

Libvirt ermöglicht durch *virsh* – einer auf der Bibliothek basierenden Kommandozeile – einen schnellen und universellen Zugriff auf virtuelle Maschinen. Der Funktionsumfang von *virsh* deckt die gesamte Funktionalität der zentralen Hypervisortreiber in libvirt ab.

Das Projekt ist lebendig und wird konstant weiterentwickelt. Der im Rahmen dieser Arbeit entwickelte Prototyp basiert auf libvirt und erweitert die Funktionalität um diverse Attribute. Der Versuch einer Übernahme in den Hauptentwicklungszweig ist geplant.

2.2. DMTF CIM und VMAN

Beim Common Information Model (CIM) handelt es sich um einen offenen Standard, der beschreibt, wie Managed Objects – also Bausteine der IT-Infrastruktur – als Menge von Objekten repräsentiert werden können. Darüber hinaus werden auch die Zusammenhänge zwischen diesen Objekten spezifiziert. Gepflegt wird der Standard von einem Industriekonsortium der DMTF, in dem IT-Größen wie AMD, Cisco, Dell, HP, IBM, Intel, Microsoft, Oracle, VMware und viele andere Mitglied sind. Darüber hinaus bestehen viele Partnerschaften mit Universitäten auf der ganzen Welt. Mit CIM ist auch das Web Based Enterprise Management (WBEM) zu assoziieren, was ein Oberbegriff für die Meisten von der DMTF herausgegebenen Standards ist.

Der Virtualization Mangagement (VMAN) Standard ist als Erweiterung dieser Konzepte zur Adressierung des Managements von virtualisierten Umgebungen zu sehen. Hierbei soll der ganze Lebenszyklus (siehe Abbildung 2.1) einer virtuellen Appliance thematisiert werden.

Quelle: Open Virtual Machine Format Whitepaper[DMT09b]

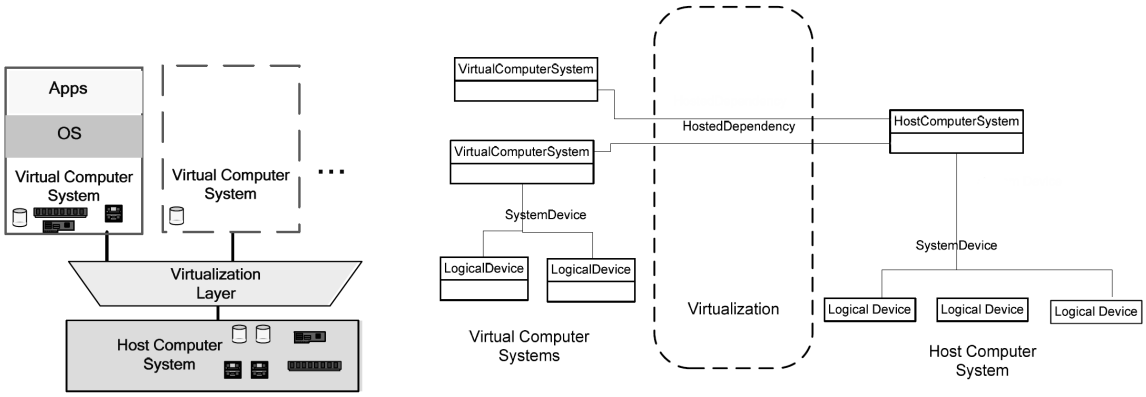
Abbildung 2.1.: Lebenszyklus einer virtuellen Appliance

Ein großes, relevantes Produkt ist die Spezifikation des Open Virtualisation Formats (OVF). Hierbei handelt es sich um einen weit verbreiteten und unterstützten Standard zur Beschreibung virtueller Infrastrukturen und Maschinen. Der Fokus dieses Standards zielt auf die Lebenszyklusphasen Package, Distribute und Deploy. Im Rahmen dieser Arbeit ist allerdings die Manage-Phase von zentralem Interesse. Auf diese Phase zielt das CIM System Virtualization Model ab. Hierzu gehören zusätzlich die Modellerweiterungen des CIM Schemas um virtuelle Maschinen und Hostsysteme.

Die Anforderungen, die von vornherein an dieses Modell gestellt wurden, beinhalteten auch den Aspekt, dass sich die Fähigkeiten der unterschiedlichen Virtualisierungslösungen enorm unterscheiden können. Daher muss das Modell flexibel genug sein, die unterschiedlichen Ty-

pen der Virtualisierung – von Hardware-Virtualisierung bis hin zu Containern innerhalb der Betriebssyteme – zu unterstützen. Des weiteren spezifiziert der Standard Methoden, virtuelle Maschinen durch die Spezifikation ihrer Ressourcen zu erzeugen, zu verändern oder auch zu löschen. Die Abbildung von virtuellen Ressourcen auf die darunterliegenden Hostressourcen ist, wenn es technisch möglich ist, über so viele Virtualisierungsschichten wie nötig darstellbar. [DMT07]

Ein einführendes Beispiel wie ein virtuelles System abgebildet wird, zeigt Abbildung 2.2; auf der linken Seite wird die Architektur dargestellt, rechts daneben die Abbildung auf CIM-Klassen.



Quelle: CIM System Virtualization Model White Paper[DMT07]

Abbildung 2.2.: Beispiel zur Abbildung von virtuellen Maschinen bei CIM

Des Weiteren gibt es Eigenschaften mittels derer ein Client Objekte korrellieren kann. Somit kann ein Client beispielsweise feststellen, dass das *ComputerSystem*, das ein auf einer VM laufender CIM Object Manager(CIMOM) bereitstellt, und das *VirtualComputerSystem*, das von einem CIMOM der auf einem Virtualization Control Point läuft, ein und dasselbe *ComputerSystem* beschreiben. Auch die Reservierung von Ressourcen sowie Zustände von Virtuellen Maschinen werden von dem Standard behandelt. [DMT07]

Veröffentlicht wird der Standard als Dokumentsammlung, die als Ergänzung des CIM Standards zu sehen ist. Die Aufgabe der Implementierung muss von den Hypervisorherstellern geleistet werden. Die Implementierung der CIM Schnittstelle von VMware ist unvollständig und beschränkt sich auf Hardwarekomponenten und angeschlossene Storage Devices [BSB+10]. Wie schon in der Einleitung erwähnt ist, aufgrund des Top-Down Ansatzes, der zuerst ein Modell spezifiziert und später Implementierungen dieses Modells auf Hypervisorseite fordert, und auf Grund der Komplexität des Modells eine breite Unterstützung in der Praxis in nächster Zukunft nicht zu erwarten.

Zwischen CIM und den Hypervisoren gibt es Unterschiede in den Attributssemantiken. Auch definiert CIM Attribute, die die Hypervisoren über ihre Managementschnittstellen nicht anbieten.

Ein Beispiel wäre der Speicher virtueller Maschinen. CIM stellt den verfügbaren Speicher eines aktiven Virtual Systems mittels einer Blockgröße (*BlockSize*) und der Anzahl der nutzbaren Blöcke (*ConsumableBlocks*) dar (siehe Abbildung 2.3, Klasse VS_MEM), der verfügbare Speicher eines aktiven Virtual Systems soll als Produkt dieser beiden Werte berechnet werden: ,,For each of the instances [of the CIM_Memory class] returned from step

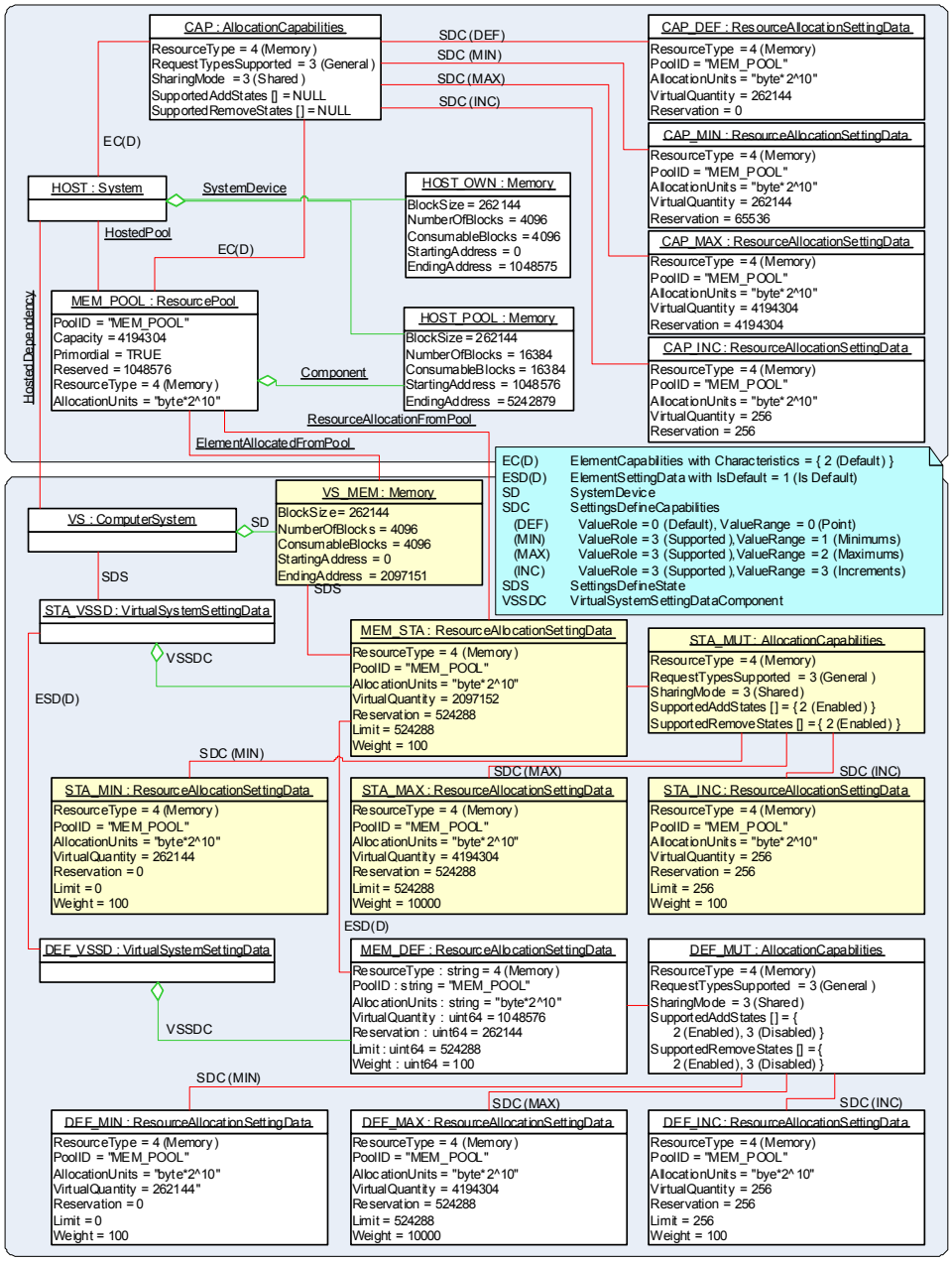
1), the client inspects the values of the BlockSize and NumberOfBlocks properties, multiplies these values, and adds the results. The resulting sum is the amount of virtual memory available to the virtual system." [DMT09a, Zeile 1032 ff.]. Diese beiden Größen sind mittels xenapi oder vSphere API (siehe Abschnitt 3.1) so nicht abfragbar. Bei diesen liegt der verfügbare Speicher direkt vor (vgl. Abschnitt 3.2 und Abschnitt 4.2.3), eine Abbildung auf die Blockgröße und -anzahl ist aber nicht möglich.

2.3. Positionierung der Arbeit in diesem Umfeld

Diese Arbeit teilt sich mit CIM das Ziel der Schaffung eines gemeinsamen Informationsmodells, grenzt sich aber durch die Vorgehensweise stark davon ab. Der CIM Standard spezifiziert ein solches Informationsmodell, betrachtet allerdings die darunter nötige Abbildung der hypervisorspezifischen Attribute auf dieses Modell nicht. Diese Arbeit beschreitet den umgekehrten Weg und beginnt bei den Attributen, die von den Hypervisor-APIs bereitgestellt werden und bildet diese auf einen generischen Attributsstamm ab. Aus diesem kann ein Modell erstellt werden, das die tatsächlich verfügbaren Attribute abbildet. Ein konkretes Modell wird im Rahmen der Arbeit noch nicht angegeben.

Die praxisbezogene Ausrichtung der libvirt auf die hypervisorunabhängige Abbildung von generischen Funktionen deckt sich mit dieser Arbeit. Im Gegensatz zu libvirt zielt diese Arbeit auf die Attribute der Hypervisoren und virtuellen Maschinen ab. Für die Implementierung des Attributsstamms wird auf libvirt als Plattform zurückgegriffen.

Als ersten Schritt zur Schaffung dieses Attributsstamms wird im folgenden Kapitel auf den Bezug und die Katalogisierung der Attribute der unterschiedlichen Hypervisoren eingegangen.



Quelle: Memory Resource Virtualization Profile [DMT09a]

Abbildung 2.3.: Beispielhafte CIM Darstellung der Memory Ressource Virtualization

3. Bezug und Sammlung der Attribute

Nach der Klärung des Attributsbegriffs zu Beginn dieses Kapitels folgt in Abschnitt 3.1 eine Betrachtung der Mangementschnittstellen der Hypervisoren Xen und VMware ESXi. Da die beiden vorgestellten Management APIs im Rahmen dieser Arbeit eine wichtige Rolle spielen, wird in Abschnitt 3.1.3 noch einmal auf Gemeinsamkeiten, Unterschiede, Vor- und Nachteile eingegangen. Zum Abschluss dieses Kapitels wird in Abschnitt 3.2 die Katalogisierung und semantische Untersuchung der Hypervisorattribute vorgenommen.

Der Federal Standard 1037C definiert ein Attribut wie folgt: „In network management, a property of a managed object that has a value.“[ITS96]

Kommen Attribute aus mehreren Datenquellen zum Einsatz, so genügt es nicht Beziehungen zwischen den Bezeichnern der Attribute herzustellen, da auch bei gleich lautenden oder auf den ersten Blick übereinstimmenden Attributen Unterschiede in der Syntax und Semantik der Attribute bestehen können. Auf der anderen Seite, können eventuell unterschiedlich lautende Attribute dasselbe bezeichnen oder mehrere Attribute auf ein anderes abgebildet werden.

Definition 1. Zwei Attribute sind gleich, wenn sie in folgenden Punkten übereinstimmen:

- ihrem *Bezeichner*, einem eindeutigen Namen;
- ihrer *Werteskala* (engl. domain). Die Werteskala beschreibt sowohl den Datentyp eines Attributs, als auch einen Wertebereich, der für dieses Attribut gültig ist.
- ihrer *Semantik*, der Bedeutung der konkreten Werte für die Anwendung.

Ein Beispiel für ein Attribut ist „PC Hauptspeicher:Integer“. Hier ist der Bezeichner „PC Hauptspeicher“, der Datentyp „Integer“. Der Wertebereich kann implizit als > 0 angenommen werden. Die Semantik muss beschreiben, ob dieser Wert nun als Byte, Megabyte oder Gigabyte interpretiert werden muss. Allerdings ist die Semantik nicht auf die physikalische Einheit reduzierbar. Sie legt darüber hinaus die Skalierung der Werteachse, sowie auch die Aussage eines konkreten Wertes in der Praxis fest. In diesem Beispiel legt sie fest, dass unter ,,PC Hauptspeicher” die Größe des im System verfügbaren Arbeitsspeichers ist.

Auch müssen die Werte nicht numerisch sein. Ein Beispiel dafür wären Zustandsattribute, die mit einem Bezeichner für unterschiedliche Zustände belegt werden. Die Semantik beschreibt in diesem Fall darüber hinaus, was die einzelnen Zustandsbezeichner aussagen.

Da das Ziel der Arbeit eine Untersuchung von heterogenen Managementattributen ist, wird in diesem Kapitel zuerst darauf eingegangen, wie bei den beiden in dieser Arbeit betrachteten Virtualisierungslösungen Managementattribute abgerufen werden können. Anschließend wird ein kurzer Vergleich dieser Schnittstellen vorgenommen.

Für den Verlauf dieser Arbeit wird die Abbildung von Bezeichnern unterschiedlicher Datenquellen aufeinander oder auf generische Bezeichner implizit durch Gegenüberstellen der Attributsbezeichner aller Datenquellen angegeben.

Mit Hilfe der folgenden Äquivalenzrelation erfolgt eine Beschränkung der Gleichheit auf die syntaktische und semantische Vergleichbarkeit:

Definition 2. Seien a und b Attribute. Es gelte $a \equiv_s b$ genau dann, wenn a und b syntaktisch und semantisch vergleichbar sind, d.h. die Werteskalen der Attribute übereinstimmen und ihre Semantik gleich ist.

3.1. Managementschnittstellen von Virtualisierungslösungen

Im Folgenden werden die Schnittstellen der Hypervisoren Xen und VMware ESXi dargestellt und ein Vergleich dieser durchgeführt. Dies ist erforderlich, da sich die unterschiedlichen Schnittstellen auf die Attributswertbeschaffung auswirken. Die API steckt darüber hinaus den Rahmen ab, inwiefern und welche Attribute beschafft werden können.

3.1.1. XenApi

Die XenApi, manchmal auch Xen Management Api, ist eine mit Xen in Version 3.0.5 eingeführte Bibliothek zum Management von Xen Hostsystemen über ein Netz. Die API selbst liegt momentan in Version 1.0.10 vor. Zu beachten ist, dass die XenApi in zwei leicht verschiedenen Ausprägungen existiert. Einmal für die Open-Source Variante von Xen und einmal für den Citrix XENServer. Die Unterschiede sind marginal, allerdings reichen sie aus, um Kompatibilitätsprobleme zwischen den zugehörigen Bibliotheken (libxen, auf Open Source Seite, bzw. libxenserver bei Citrix) zu schaffen.

Spezifiziert ist die XenApi als eine Sammlung von Remote Procedure Calls (RPCs), also als von der Wahl der Programmiersprache unabhängiges Protokoll, dessen Übertragungsformat auf XML-RPC basiert. Die RPCs werden als XML dargestellt und via HTTP/S übertragen. Da es sich bei XML und HTTP/S um weit verbreitete Standards handelt, sind Implementierungen von XML-RPC für viele Programmiersprachen verfügbar. Somit ist ein Zugriff auf die XenApi aus vielen Programmiersprachen heraus möglich. Unter C kann auf die schon erwähnte libxen zurückgegriffen werden. Für Python existiert eine Bibliothek (siehe Listing 3.1). In Listing 3.2 werden beispielhaft die XML-RPCs dargestellt, die durch das Aufrufen von `login_with_password()` in Listing 3.1 entstehen.

```
1 import xen.xm.XenAPI
2
3 session = xen.xm.XenAPI.Session("http://localhost:9363")
4 session.login_with_password("user", "pass")
5
6 vms = session.xenapi.VM.get_all()
7 for i in vms:
8   print session.xenapi.VM.get_name_label(i)
```

Listing 3.1: Beispiel zur Verwendung der XenApi in Python

```
1 <?xml version='1.0'?>
2 <methodCall>
3   <methodName>session.login_with_password</methodName>
4   <params>
5     <param>
6       <value><string>user</string></value>
7     </param>
8     <param>
9       <value><string>pass</string></value>
10    </param>
```

```
11   </params>
12 </methodCall>
13
14 <?xml version='1.0'?>
15 <methodResponse>
16   <params>
17     <param>
18       <value><struct>
19         <member>
20           <name>Status</name>
21           <value><string>Success</string></value>
22         </member>
23         <member>
24           <name>Value</name>
25           <value><string>5e9b720e-8b09-3fbd-a4b5-d57859a8b045</string></value>
26         </member>
27       </struct></value>
28     </param>
29   </params>
30 </methodResponse>
```

Listing 3.2: Beispiel der XML-RPC

XenApi kennt neben den Standard-Datentypen *int*, *float*, *bool*, *DateTime* und *String* noch

- *t Ref* für Referenzen auf Objekte des Typs *t*,
- *t Set* für Mengen von Werten des Typs *t*,
- *(t_1 t_2) Map* für Schlüssel-Werte Paare. Schlüssel sind vom Typ t_1; Werte vom Typ t_2.

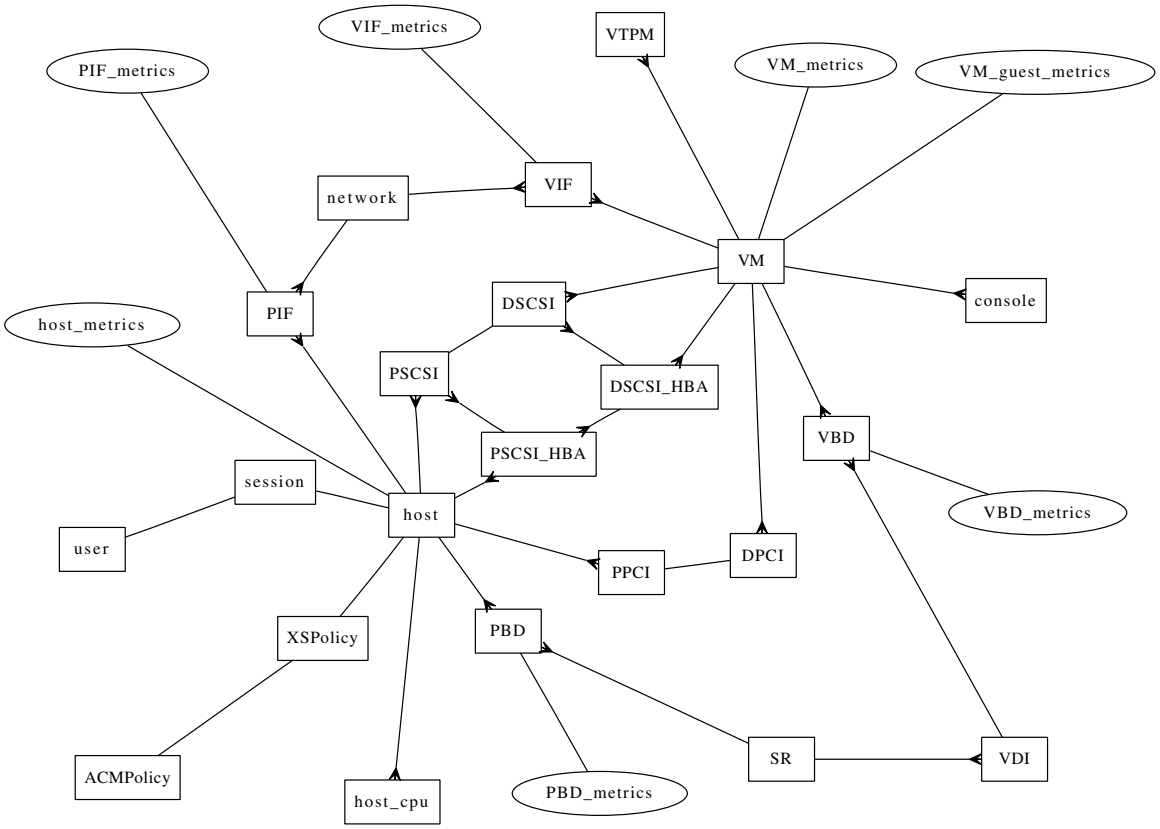
Für die genaue Abbildung der XenApi Datentypen auf die Datentypen von XML-RPC wird auf die Dokumentation der XenApi verwiesen: [MSS+10]. Alles, was nicht direkt mit in XML-RPC spezifizierten Datentypen ausdrückbar ist, wird als Zeichenfolge vom Typ String ausgedrückt, darunter fällt auch der bei XenApi standardmäßig 64-Bit breite *int* Datentyp.

XenApi ist vom Grundgedanken her nicht objektorientiert, kann aber clientseitig objektorientiert implementiert werden. Trotzdem bedient sich die XenApi der objektorientierten Terminologie mit *Klassen* und *Objekten*. In der Terminologie der XenApi versteht man unter einer *Klasse* einen Namensraum für semantisch zusammengehörige Attribute, ein *Objekt* ist eine Instanziierung einer *Klasse* mit spezifischen Werten.

Das der XenApi zugrunde liegende Datenmodell ist eine Sammlung von Klassen, die untereinander mittels Referenzen verknüpft werden. Eine Referenz gleicht von ihrer Form einer UUID, darf aber keinesfalls mit einer solchen verwechselt werden. Ein Server darf diese absolut frei wählen, sie muss keinen Anforderungen des Standards für UUIDs genügen. Die Referenzen sollten nicht interpretiert werden, da es sich dabei im Gegensatz zu UUIDs keinesfalls um globale dauerhafte Namen für ein Objekt handelt.

Abbildung 3.1 bietet einen Überblick über alle Klassen der XenApi und deren Zusammenhänge. Gut zu erkennen sind die zentrale Stellung der Klassen *Host* und *VM*. Da es keine vollständige Hierarchie auf dem Modell gibt, besitzen die wichtigsten Klassen einen `_get_all()` RPC, der eine Auflistung aller Referenzen der Objekte dieser Klasse zurückliefert. Referenzen sind nur serverspezifisch gültig.

Zum Abruf der Attribute gibt es für jedes Attribut einen einzelnen RPC (Getter), durch den dieses Attribut zur Verfügung gestellt wird. Ist Schreibzugriff auf ein Attribut möglich,



Quelle: Xen Management API [MSS+10]

Die ovalen `_metrics` Klassen, beschreiben keine Managed Objects, sondern stellen Laufzeitinformation anderer Managed Objects dar.

Abbildung 3.1.: Überblick über die Klassen der XenApi

so wird analog ein „Setter" in der API definiert (vgl. Listing 3.3). Darüber hinaus existiert zur jeder Klasse ein `_get_record()` RPC, der alle Attribute dieser Klasse zurückliefert.

```
1  string get_name_label (session_id s, VM ref self)
2  void set_name_label (session_id s, VM ref self, string value)
```

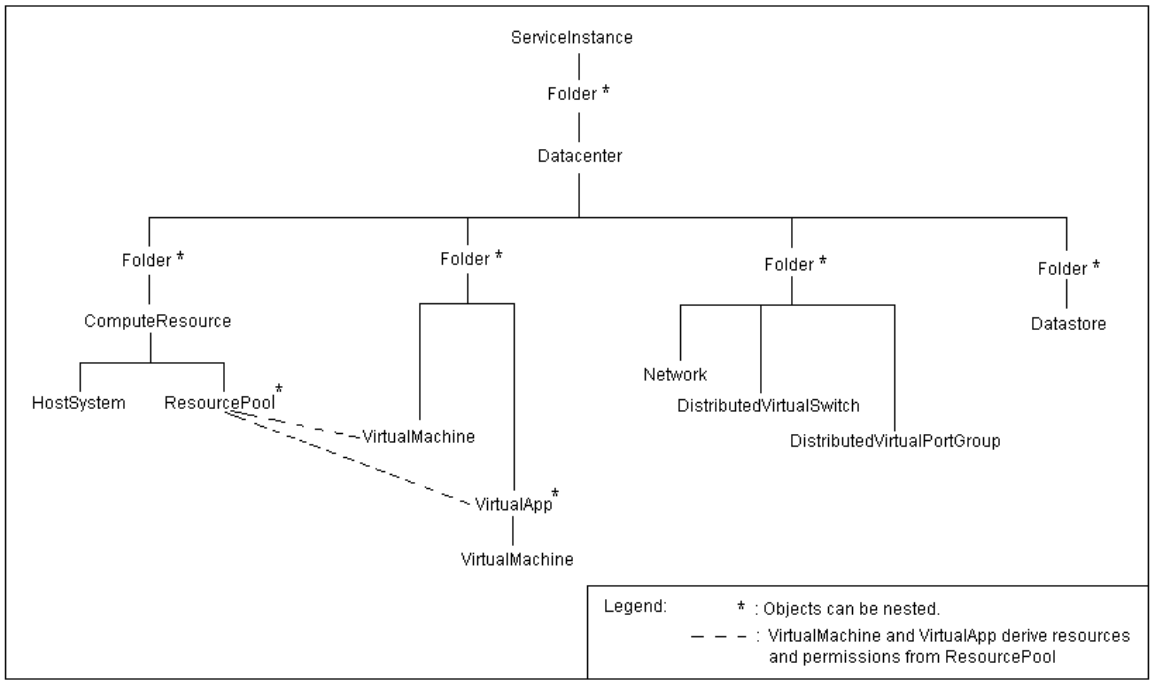
Listing 3.3: Signatur der Getter und Setter für das Attribut `name/label` einer VM

Im Verlauf dieser Arbeit werden hauptsächlich Attribute aus den Klassen *host*, *host_cpu*, *host_metrics*, *PIF*, *VM*, *VM_metrics* und *VIF* betrachtet.

3.1.2. VMware vSphere API

Als Managementschnittstelle für die aktuellen VMware Produkte VMware vSphere und VMware vSphere Hypervisor (ESXi) dient das VMware vSphere API (auch Virtual Infrastructure SDK oder VI-SDK). Dabei handelt es sich um einen auf dem Hypervisor bereitgestellten WebService. Der WebService erfüllt die Vorgaben des Web Services Interoperability Organization (WS-I) Basic Profile 1.0. Demnach werden SOAP 1.1, WSDL 1.1 sowie XML Schema 1.0 unterstützt. Transportiert werden die in XML dargestellten SOAP Envelopes mittels HTTP/S. Das vSphere WebService SDK stellt eine Reihe von WSDL Dokumenten

zur Beschreibung der Schnittstellen des Hypervisors bereit. Aus diesen können WebService Entwicklungstools – wie zum Beispiel aus dem Apache AXIS Toolkit – den benötigten Client Proxy Stubcode erzeugen.



Quelle: vSphere Web Services SDK Programming Guide [VMw10]

Abbildung 3.2.: Organisation der Managed Objekt Klassen

.

Das von VMware genutzte Datenmodell ist ein Baum (vgl. Abbildung 3.2), dessen Wurzelelement eine ServiceInstance bildet. Alle abfragbaren Objekte und Attribute sind Kinder der ServiceInstance. Auch die Datentypen in diesem Baum sind hierarchisch aufgebaut. Die oberste Hierarchiestufe bilden *Managed Object Types*, darunter liegen die *Data Object Types*, gefolgt von Enummerationstypen und elementaren Datentypen. Sowohl bei Managed Object Types, als auch bei Data Object Types handelt es sich, wie der Name schon nahe legt, um objektorientierte Datentypen, die Vererbung unterstützen.

Zu Managed Objects existieren *Managed Object References* (MOR), mit Hilfe derer Clientapplikationen Objekte referenzieren können. Der Zugriff auf Objekte und deren Attribute geschieht bei VMware immer ausgehend von einer Managed Object Reference. Alle Methoden beziehen sich auf Managed Objects.

Im Gegensatz dazu handelt es sich bei den Data Object Types um methodenfreie Klassen, die innerhalb von Managed Objects genutzt werden, um Attribute hierarchisch zu strukturieren. Dabei wird ebenfalls Vererbung eingesetzt, das rechtfertigt die Bezeichnung als Klasse. Data Objects können nur über die Managed Objects, denen sie zugehörig sind, referenziert werden. Data Objects können daher auch als Attribute der Managed Objects gesehen werden.

Die Möglichkeit der Vererbung auf den Data Objects führt auf der einen Seite dazu, dass viele Objekte sehr spezialisiert und detailliert beschrieben werden. Auf der anderen Seite

erschließen sich viele Zusammenhänge und Bezugsmöglichkeiten nur nach Durchsuchung aller abgeleiteten Klassen. Bei Anfragen nach vererbten Attributen muss sichergestellt sein, welche abgeleitete Klasse die angefragte Basisklasse instanziiert. An folgendem Beispiel für die Vererbung von Data Objects werden Nachteile dieser Vererbung sichtbar. *VirtualEthernetCard* wird von einem *VirtualDevice* abgeleitet und kann selbst wiederum beispielsweise durch ein *VirtualE1000* spezialisiert werden. Um auf eine *VirtualEthernetCard* zuzugreifen, müssen alle Elemente von `VirtualMachine.config.hardware.devices[]`, das eine Menge an Data Objects vom Typ *VirtualDevice* enthält, daraufhin überprüft werden, ob sie vom Typ *VirtualEthernetCard* sind.

Ein weiterer Nachteil des Baumes ist, dass er nicht frei von Redundanzen ist. Das bedeutet, dass einige Attribute eines Mangaged Objects in mehreren unterschiedlichen darunterliegenden Data Objects redundant abrufbar sind. Dabei wird nicht sichergestellt, dass die Attribute auf bestimmte Grundeinheiten normiert werden. Dasselbe Attribut kann also in unterschiedlichen Einheiten vorliegen. Beispielsweise `HostSystem.hardware.cpuInfo.hz` und `HostSystem.summary.hardware.cpuMhz`, die bis auf unterschiedliche Einheiten dieselbe Semantik besitzen (vgl. Definition 1). Gerade zwischen Attributen innerhalb des *summary*-Astes mancher Managed Objects und anderen Ästen treten diese Redundanzen auf.

Durch kreuzweises Referenzieren mittels Managed Object References wird die Baumstruktur durchbrochen und Beziehungen zwischen unterschiedlichen Managed Objects möglich.

Benutzerverwaltung ist möglich und die Rechte eines Nutzers können für Teilbäume eingeschränkt werden.

Konkret abfragbar sind die Attribute mittels der Methode `RetrieveProperties()` eines `PropertyCollector` Objekts. Diese erwartet eine Managed Object Reference auf das Objekt, auf die sich die Anfrage bezieht, und ein Data Object vom Typ `PropertyFilterSpec`, das im Endeffekt eine Liste aller abzufragenden Äste ist. Ein Beispiel für einen solchen RPC wird in Listing 3.4 dargestellt.

Im Rahmen dieser Arbeit werden Attribute der Managed Objects *HostSystem* und *VirtualMachine* betrachtet.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:xsi="http://
       www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
       XMLSchema">
3    <soapenv:Body>
4      <RetrieveProperties xmlns="urn:vim25">
5        <_this xmlns="urn:vim25" xsi:type="ManagedObjectReference" type="
             PropertyCollector">
6          ha-property-collector
7        </_this>
8        <specSet xmlns="urn:vim25" xsi:type="PropertyFilterSpec">
9          <propSet xmlns="urn:vim25" xsi:type="PropertySpec">
10           <type xmlns="urn:vim25" xsi:type="xsd:string">
11             HostSystem
12           </type>
13           <pathSet xmlns="urn:vim25" xsi:type="xsd:string">
14             summary.hardware.uuid
15           </pathSet>
16         </propSet>
17         <objectSet xmlns="urn:vim25" xsi:type="ObjectSpec">
```

```
18            <obj xmlns="urn:vim25" xsi:type="ManagedObjectReference" type="
                  HostSystem">
19              ha-host
20            </obj>
21            <skip xmlns="urn:vim25" xsi:type="xsd:boolean">
22              false
23            </skip>
24          </objectSet>
25        </specSet>
26      </RetrieveProperties>
27    </soapenv:Body>
28 </soapenv:Envelope>
29
30 <?xml version="1.0" encoding="UTF-8"?>
31 <soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/".
       xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://
       www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
       instance">
32    <soapenv:Body>
33      <RetrievePropertiesResponse xmlns="urn:vim25">
34        <returnval>
35          <obj type="HostSystem">
36            ha-host
37          </obj>
38            <propSet>
39              <name>
40                summary.hardware.uuid
41              </name>
42              <val xsi:type="xsd:string">
43                e0f9408c-6e99-db11-b586-001d6088a529
44              </val>
45            </propSet>
46          </returnval>
47        </RetrievePropertiesResponse>
48      </soapenv:Body>
49 </soapenv:Envelope>
```

Listing 3.4: Beispiel eines `RetrieveProperties` RPC anhand der SOAP Envelopes

3.1.3. Vergleich der Schnittstellen

Beide APIs sind von Plattformen und Programmiersprachen unabhängig, da sie zwar unterschiedliche, aber standardisierte Schnittstellen zum Zugriff über ein Netz bereitstellen. Beide basieren auf dem Gedanken eines Remote Procedure Calls, einmal mittels XML-RPC und einmal mittels SOAP, die wiederum beide als Darstellungsform XML nutzen und via HTTP/S übertragen werden.

Die zugrunde liegenden Datenmodelle sind komplett verschieden: Während bei VMware das Baummodell nur durch die nicht vermeidbare Referenzierung der Objekte untereinander durchbrochen wird, bietet das Modell der XenApi keinerlei Hierarchie. Es wird bei Xen immer eine *Session* als Zugriffspunkt benötigt. Aber, diese vorausgesetzt, sind dann alle Objekte flach adressierbar. Dies erschwert eine Benutzerverwaltung, wie sie bei VMware möglich ist, und ist daher von der API nicht in dieser Form vorgesehen. Durch den großen Verzweigungsgrad bei VMware und die Vererbungsmöglichkeiten der Klassen können Attribute sehr fein modelliert werden. XenApi verfolgt in dieser Hinsicht einen anderen Ansatz

und beschränkt sich auf aus Sicht der Entwickler wesentliche Attribute. Das zieht zwar eine geringere Detailliertheit nach sich, vereinfacht im Gegenzug aber Zugriffe und erhöht die Übersichtlichkeit.

Im Falle von Schreibzugriffen ist bei XenApi aus der Definition klar ersichtlich, welche Attribute direkt manipuliert werden können. Die entsprechenden RPCs tragen statt eines *get* ein *set* im Namen. Bei VMware können Attribute nur durch Methoden der Managed Objects verändert werden, dafür sind als Data Object spezifizierte Änderungsanweisungen nötig. Die Deklaration eines solchen RPCs erschließt sich nicht offensichtlich und die API Referenz zeigt nicht deutlich, auf welche Attribute überhaupt schreibend zugegriffen werden kann. Ein weiterer Vorteil der XenApi ist das deutlich schlankere Leitungsformat des XML-RPC im Vergleich mit SOAP.

3.2. Katalogisierung der Attribute

Ein Ziel dieser Arbeit ist es, für das Management relevante Attribute der Hypervisoren zu sammeln, um diese später abbilden zu können. Zu diesem Zweck wurden Host- und VM-Attribute der beiden Hypervisoren gesammelt und deren Semantik geklärt. Für den Verlauf der Arbeit repräsentative Attribute werden im Rahmen dieses Abschnittes dargestellt, für die gesamten Ergebnisse dieser Untersuchung wird auf Anhang A verwiesen.

Der Bereich Speichermedien wird aufgrund seiner Komplexität bewusst ausgenommen. Die Komplexität ergibt sich durch die Vielzahl der unterstützten Speicherlösungen wie beispielsweise Festplatten in den Hosts oder unterschiedlicher NAS Systeme. Die Sammlung erfolgte durch Durchforsten der API Referenzen beider Hypervisoren ([MSS+10] bzw. [VMw10]). Zur Klärung der Semantik wurden sowohl die API Referenzen untersucht, als auch Versuche an Testsystemen durchgeführt. Bei Xen stellten auch das Xen Wiki, die xen-devel, xen-users und xen-api Mailinglisten, sowie Teile des Xen Quellcodes Quellen dar. Da die Größe einer möglichen Schnittmenge der Attribute durch die geringere Anzahl der Attribute bei Xen definiert wird, wurden bei VMware Attribute, die – unter Berücksichtigung potenzieller Abbildbarkeit – keine Entsprechung bei Xen haben, nicht weitergehend untersucht. Desweiteren werden die untersuchten Attribute auf virtuelle Maschinen und Hostsysteme eingeschränkt.

Bei den Testsytemen handelt es sich um zwei – bis auf die Zahl der Ethernet Schnittstellen – identische Systeme, auf denen Xen in Version 4.0.1 unter der Distribution Debian Squeeze (testing) beziehungsweise VMWare ESXi 4.1 eingerichtet wurden.

Der in Anhang A vollständig angegebene Katalog besteht genaugenommen aus zwei Katalogen. Ein Katalog für die XenApi, einer für das vSphere API. Die weitere Untergliederung folgt den Klassen der XenApi beziehungsweise den Managed Object Types bei VMware. Data Object Types sind ihrer Hierarchie folgend – ausgehend von den Managed Object Types – eingegliedert.

3.2.1. Auszüge des Attributskatalogs

Im folgenden wird ein Auszug der Attributskataloge dargestellt, der für den weiteren Verlauf der Arbeit von zentraler Bedeutung ist. Der gesamte Katalog ist in Anhang A einsehbar.

Attribute von Xen

host.uuid	string	Global eindeutiger Objektbezeichner des Hosts
host.name/label	string	Name des Hosts
host.name/description	string	Informelle Beschreibung des Hosts
host.enabled	bool	*1
host.cpu_configuration	(string →string) Map	*2
host_cpu.speed	int	Geschwindigkeit der PCPU in Mhz
host_cpu.modelname	string	Die Modellbezeichnung der PCPU
host_cpu.utilisation	float	Momentane Auslastung der PCPU im Interval [0,1]
host_metrics. memory/total	int	Gesamter Speicher des Hosts in Bytes
host_metrics.memory/free	int	Freier Speicher des Hosts in Bytes
PIF.MAC	string	Ethernet MAC Adresse einer physikalischen Schnittstelle
VIF.MAC	string	Ethernet MAC Adresse einer virtuellen Schnittstelle
network.name/label	string	Name des Netzes
VM.uuid	string	Global eindeutiger Objektbezeichner
VM.power_state	vm_power_state	*3
VM.name/label	string	Name
VM.name/description	string	Informelle Beschreibung der virtuellen Maschine
VM.is_a_template	bool	true, falls diese VM nur als Template fungiert. Template VMs können nicht gestartet, sondern nur als Vorlage zur Erzeugung neuer VMs genutzt werden
VM.is_control_domain	bool	true falls dies eine Control Domain ist (dom0)
VM_metrics.uuid	string	Global eindeutiger Objektbezeichner
VM_metrics.memory/actual	int	Aktuell dem Gast zugeteilter Speicher in Bytes
VM_metrics.VCPUs/number	int	Momentane Anzahl virtueller CPUs
VM_metrics. VCPUs/utilisation	(int →float) Map	Aktuelle Auslastung der einzelnen VCPUs der VM im Interval [0,1]

VM_metrics.VCPUs/CPU	(int →int) Map	Mapping von VCPUs auf PCPUs
VM_metrics.start_time	datetime	Zeitpunkt, zu dem die VM zuletzt gebootet wurde (ISO 8601)

*1 true, falls der Host eingeschaltet ist, und sich in einem Zustand befindet, in dem VMs gestartet werden können.
*2 Die CPU beschreibende Attribute wie nr_nodes,nr_cpus,sockets_per_node,cores_per_socket oder threads_per_core, die Semantik folgt aus den Namen Schlüssel.
*3 Der power_state kann folgende Werte annehmen: Halted, Paused, Running, Suspended, Crashed, oder Unknown. Unknown steht für einen anderen unbekannten Status. Crashed bezeichnet ein abgestürztes Gastbetriebssystem. Der Unterschied zwischen Paused und Suspended ist, dass im Zustand Suspended das Image der VM auf Hintergrundspeicher geschrieben wurde, bei Paused liegt dieses im Arbeitsspeicher. Den Zusammenhang dieser Attribute verdeutlicht Abbildung 3.3.

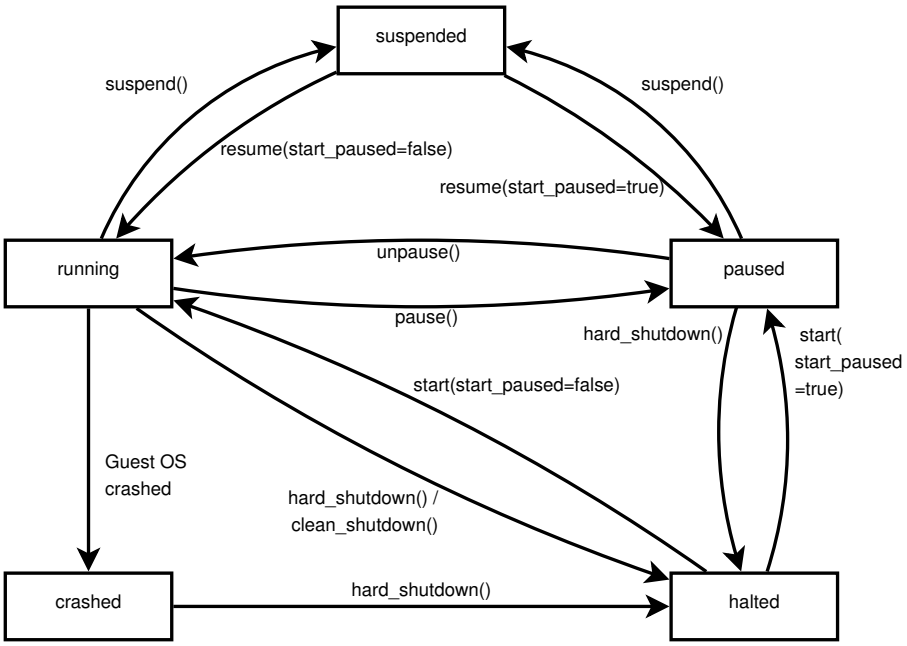


Abbildung 3.3.: Zustände virtueller Maschinen bei Xen

Attribute von VMware ESXi

HostHardwareInfo. memorySize	xsd:long	Gesamtmenge des physischen Speichers in Bytes
HostHardwareSummary. cpuMhz	xsd:int	Bei mehreren CPUs Durchschnitt in Mhz

HostHardwareSummary.cpuModel	xsd:string	Das CPU Modell
HostHardwareSummary.numCpuCores	xsd:short	Anzahl der physischen Cores auf einem Host
HostHardwareSummary.numCpuThreads	xsd:short	Anzahl der physischen CPU Threads auf dem Host
HostHardwareSummary.memorySize	xsd:long	Gesamtmenge des physischen Speichers in Bytes
HostHardwareSummary.numNics	xsd:int	Anzahl der physischen Netzschnittstellen
HostHardwareSummary.uuid	xsd:string	Global eindeutiger Bezeichner des Hosts
HostListSummaryQuickStats.overallCpuUsage	xsd:int	Aggregierte CPU Auslastung über alle Cores in Mhz
HostListSummaryQuickStats.overallMemoryUsage	xsd:int	Speichernutzung auf dem Host in MB
HostSystem.name	xsd:string	Name eines Hosts (geerbt von ManagedEntity)
HostSystemRuntimeInfo.powerState	HostSystemPowerState	*1
Network.name	xsd:string	Name des Netzes (geerbt von ManagedEntity)
VirtualEthernetCard.macAddress	xsd:string	Ethernet MAC Adresse
VirtualHardware.memoryMB	xsd:int	Größe des Speichers in MB
VirtualHardware.numCPU	xsd:int	Anzahl der in dieser VM verfügbaren VCPUs
VirtualMachine.name	xsd:string	Name der VM (geerbt von ManagedEntity)
VirtualMachineConfigInfo.annotation	xsd:string	Informelle Beschreibung der VM
VirtualMachineConfigInfo.template	xsd:boolean	Beschreibt ob eine VM ein Template ist
VirtualMachineConfigInfo.uuid	xsd:string	128-bit SMBIOS UUID einer VM
VirtualMachineConfigSummary.numEthernetCards	xsd:int	Anzahl der Virtuellen Netzschnittstellen
VirtualMachineRuntimeInfo.bootTime	xsd:dateTime	Zeitpunkt zu dem die VM zuletzt gebootet wurde
VirtualMachineRuntimeInfo.powerState	VirtualMachinePowerState	*2

*1 `poweredOff`; `poweredOn`; `standBy`; `unknown`, sollte nur auftreten falls der Host `disconnected` ist.
*2 `poweredOff`, die VM ist ausgeschaltet;
`poweredOn`, die VM ist eingeschaltet;
`suspended`, die VM ist momentan suspendiert.

Vergleich mit Attributen von CIM

Beispielhaft wird an dieser Stelle nochmals ein Vergleich mit CIM gezogen. Wie schon in Abschnitt 2.2 angesprochen sind weder bei VMware noch bei Xen eine Blockgröße sowie Blockanzahl für Speicherbereiche beziehbar. Der einer aktiven VM zur Verfügung stehende Speicher ist bei Xen über das Attribut `VM_metrics.memory/actual` und bei VMware über das Attribut `VirtualHardware.memoryMB` beziehbar. Diese beiden unterscheiden sich in der genutzten Einheit: Bytes bei Xen, Megabytes bei VMware. Die DMTF schlägt allerdings für Reservierungen von Speicher die Einheit Kilobyte vor, für Blockgrößen beispielsweise nutzen sie ebenfalls Bytes. Dies zeigt, dass man ohne Abbildungen zwischen Hypervisoren und dem standardisierten Modell von CIM immer Abbildungen benötigen wird.

Ein weiteres Beispiel ist der *Virtual system state*. CIM kennt die Zustände:

- `defined` Das System ist definiert, aber benötigt keine Ressourcen, ausser den Persistenten Ressource wie beispielsweise Platz für ein Image auf dem Hintergrundspeicher. In diesem Zustand kann das System keine Aufgaben ausführen.
- `active` In diesem Zustand kann das System Aufgaben ausführen.
- `paused` In diesem Zustand kann das System keine Aufgaben ausführen, allerdings bleiben Reservierungen bestehen.
- `suspended` In diesem Zustand sind alle Ressourcen des Systems auf nicht flüchtigem Speicher gesichert. Das System kann keine Aufgaben ausführen.
- `unknown` Der Zustand des Systems nicht festgestellt werden kann.
- herstellerspezifische Zustände; Diese erlauben die Überwachung weiterer herstellerspezifischer Zustände. Besitzen aber keine definierte Semantik.

CIM ist zwar bei den einzelnen Zuständen semantisch zu Xen und VMware kompatibel, allerdings müssen die Bezeichner der einzelnen Zustände aufeinander abgebildet werden. Die Kompatibilität zu beiden durchaus unterschiedlichen Modellen bei Xen und VMware, die sich schon in der Anzahl der möglichen Zustände unterscheiden, wird dadurch erreicht, dass die Implementierung einiger Zustände bei CIM optional ist. Durch die Unterstützung für herstellerspezifische Zustände kann der Zustand *crashed* von Xen unterstützt werden, wenn auch ohne die Semantik wiederzugeben. Ob diese Zustände sinnvoll sind, ist fraglich, da diese einer herstellerunabhängikeit eventuell eher im Weg stehen, da sie nicht herstellerunabhängig interpretierbar sind.

Im folgenden Kapitel wird eine Generalisierung der in diesem Kapitel extrahierten und in Anhang A tabellierten Attribute vorgenommen. Des weiteren werden Abbildungen untersucht, die Attribute eines Hypervisors auf die generalisierten Attribute abbilden.

4. Klassifikation und Normalisierung der Attribute

Aus der semantischen Untersuchung der Attribute im letzten Kapitel wird in diesem Kapitel in Abschnitt 4.1 ein Klassifikationsschema abgeleitet, welches zur Bewertung von Attributen im Hinblick auf eine Eignung als generisches Attribut dient und auf den Abbildungen zwischen den Attributen unterschiedlicher Quellen basiert. In Abschnitt 4.2 wird dieses Klassifikationsschema auf Attribute aus dem in Abschnitt 3.2 gewonnenen Attributskatalog angewandt.

Vorüberlegungen haben ergeben, dass die Abbildbarkeit und die Umkehrbarkeit der Abbildungen zwischen den Virtualisierern Rückschlüsse auf die Möglichkeiten und Flexibilität des Managements bieten. Nur wenn Abbildungen umkehrbar sind, kann auch schreibend auf generalisierte Attribute zugegriffen werden. Des weiteren haben Attribute, auf die nicht sinnvoll abgebildet werden kann, keinen Nutzen für ein gemeinsames Informationsmodell, da diese der Herstellerunabhängigkeit im Weg stehen. Um als generalisierte Attribute in Frage zu kommen, muss ein Attribut in möglichst vielen zu unterstützenden Datenquellen vorliegen, oder aus anderen Attributen erzeugbar sein. Es ist nicht sinnvoll, Attribute in ein Modell aufzunehmen, die nur von wenigen Virtualisierern unterstützt werden. Auf das Modell zugreifende Applikationen können dann nicht mehr davon ausgehen, dass dieses Attribut verfügbar ist und müssen somit wieder Spezifika der Hypervisoren beachten. Doch genau das Gegenteil davon ist das mit dem Modell verfolgte Ziel.

4.1. Klassifikationsmengen

Im Folgenden wird ein Klassifikationsschema für Attribute anhand von Mengen eingeführt.

Sei A die Menge aller Attribute, sowie A_H die Menge der in einer Datenquelle H verfügbaren Attribute.

Definition 3. Seien $H_1, \ldots H_n (n > 1)$ Datenquellen.

$$ABB(H_1, \ldots, H_n) := \{a \in A | \forall i \in [1, n] : \quad \exists b_{(i,1)}, \ldots, b_{(i,m)} \in A_{H_i} : \exists f_i : A_{H_i}^m \to A \\ \text{mit } f_i(b_{(i,1)}, \ldots, b_{(i,m)}) \equiv_s a \qquad \}$$

ist die Menge der *abbildbaren* Attribute.

Definition 4. Seien $H_1, \ldots H_n (n > 1)$ Datenquellen. Dann sei

$$NABB(H_1, \ldots, H_n)$$

die Menge der *nicht abbildbaren* Attribute. Es gilt:

$$NABB(H_1, \ldots, H_n) = A \backslash ABB(H_1, \ldots, H_n)$$

Die Bezeichner f_i und b_i genügen im Folgenden implizit der Definition von $ABB(H_1, \ldots, H_n)$.

Definition 5. Seien H_1, H_2 Datenquellen.

$$A_{H_1 \to H_2} := \left\{ a \in A_{H_2} | \exists b_1, \ldots, b_n \in A_{H_1} : \exists f_i : A_{H_1}^n \to A_{H_2} \text{ mit } f_i(b_1, \ldots, b_n) \equiv_s a \right\}$$

ist die Menge der Attribute aus A_{H_2}, die mit Attributen A_{H_1} darstellbar sind. Sie werden als *einseitig abbildbar* bezeichnet.

Zur weiteren Klassifikation von $ABB(H_1, \ldots, H_n)$ wird im Folgenden die Struktur der Funktionen f_i herangezogen.

Definition 6. Die Teilmenge $BABB(H_1, \ldots, H_n) \subset ABB(H_1, \ldots H_n)$ ist definiert als die Menge der *bijektiv abbildbaren* Attribute:

$$BABB(H_1, \ldots, H_n) := \{ a \in ABB(H_1, \ldots, H_n) | \quad \forall f_i \quad (i \in [1, n]) \text{ gilt} \quad f(b_i) : A_{H_i} \to A \text{ ist bijektiv} \quad \}$$

Eine spezielle Unterform der bijektiven Funktionen stellt die Identitätsfunktion Id dar.

Definition 7. Seien $H_1, \ldots H_n (n > 1)$ Datenquellen.

$$TABB(H_1, \ldots, H_n) := \bigcap_{i=1}^{n}{}_s A_{H_i} := \quad \{ a \in A | \forall i \in [1, n] : \exists b \in A_{H_i} \text{ mit } a \equiv_s b \}$$

die Menge der *trivial abbildbaren* Attribute. Es sei definiert

$$A_{H_1} \cap_s A_{H_2} := \bigcap_{i=1}^{2}{}_s A_{H_i}$$

Satz. *Seien H_1, H_2 Datenquellen.*

$$(A_{H_1} \cap_s A_{H_2}) = TABB(H_1, H_2) \subseteq BABB(H_1, H2) \subseteq A_{H_1 \to H_2} \subseteq ABB(H_1, H_2)$$

Der Beweis folgt direkt aus der Definition dieser Mengen.

Während die Abbildungsfunktionen f_i in $ABB(H_1, \ldots, H_n)$ im allgemeinen nicht bijektiv und somit umkehrbar sind, gilt dies insbesondere für $BABB(H_1, \ldots, H_n)$ und somit auch für $TABB(H_1, \ldots, H_n)$ schon. Die Umkehrbarkeit hat zur Folge, dass nicht nur auf a abgebildet werden kann, sondern für alle $i, j \in [1, n]$ gilt auch, dass a auf alle b_i sowie jedes b_i auf ein beliebiges b_j abgebildet werden kann.

Die nicht vorhandene Bijektivität in $ABB(H_1, \ldots, H_n)$ bedeutet allerdings nicht, dass eine Umkehrung grundsätzlich unmöglich ist. Eine Abbildung $a \mapsto b_i$ ist zwar im Allgemeinen nicht möglich, aber durch Zusatzinformationen - wie zum Beispiel weitere Attribute - kann die Umkehrung trotzdem möglich sein. Beispielsweise:

$$a_1 \equiv_s b_1 \cdot b_2 \qquad b_1 \equiv_s a_1 \cdot a_2 \qquad a_2 \equiv_s b_2 \cdot c \quad (c \neq 0)$$

Die Umkehrbarkeit der Attribute ist keine zwingend notwendige Voraussetzung für ein gemeinsames Informationsmodell. Sie erweist sich als günstig, wenn diese Attribute schreibend

genutzt werden. Aber auch bei lesendem Zugriff bedeutet eine Nichtumkehrbarkeit, dass entweder Informationen verloren gehen, oder dass in dem Modell mehr dargestellt werden kann, als von der Schnittmenge der Datenquellen geliefert wird.

Als Grundlage für die Klassifikation müssen Attributsmengen mehrerer Datenquellen vorliegen. Diese Attributsmengen werden verglichen. Attribute, die in allen Mengen vorkommen, werden als trivial abbildbar klassifiziert. Daran anschließend werden Attribute betrachtet, die für ein Modell interessant scheinen und es wird versucht, auf diese aus den anderen Mengen abzubilden. Die Klassifikation wird aus der Abbildung abgeleitet. Nach der Abbildung wird eine Normalisierung dieser Attribute im Hinblick auf ein einheitliches Modell vorgenommen.

Die Abbildbarkeit gibt daher Aufschluss über die Universalität der Attribute. Mindestens bijektiv abbildbare Attribute sind ideal um als generalisierte Attribute zu dienen. Bei Attributen, die ,,nur" abbildbar sind, muss bei einer Modellbildung auf ein Gleichgewicht zwischen Informationsverlust und Überrepräsentation geachtet werden.

Die Klassifikation liefert somit eine Menge grundlegender Attribute für ein gemeinsames Informationsmodell, sowie eine Bewertung der Attribute für eine Eignung in einem generalisierten Modell. Um ein vollständiges Modell zu schaffen muss, zusätzlich noch eine Struktur der Entitäten (z.B. Hosts, virtuelle Maschinen) geschaffen werden, deren Beziehungen ausgearbeitet und Methoden zur Manipulation spezifiziert werden.

Zur Veranschaulichung werden in Tabelle 4.2 die Klassifikationsmengen am Beispiel zweier Hypervisoren – Xen 4.0 mit XenApi als Schnittstelle (A_X), VMware ESXi 4.1 mit VI-SDK als Schnittstelle (A_V) – dargestellt. Eine detaillierte Aufstellung anhand dieser beiden Hypervisoren wird im folgenden Abschnitt gegeben.

4.2. Klassifikation am Beispiel XEN und VMware

In diesem Abschnitt, wird die Klassifikation von Attributen aus den in Abschnitt 3.2 generierten Attributskatalogen für XenApi und VMware durchgeführt. Bei der folgenden Klassifikation der Attribute wird, bei XenApi immer der Klassenname und der Attributsname und bei VMware immer der Bezugsweg ab dem betreffenden Managed Object angegeben. Ein Pfeil ($\rightarrow$) bedeutet, dass an dieser Stelle einer Referenz auf ein anderes Objekt gefolgt wird.

Die im Folgenden genutzten Datentypen müssen wie folgt interpretiert werden:

- `dateTime` Zeichenfolge nach ISO 8601
- `int` ein ganzzahliger Datentyp, die Größe der Speicherzelle wird implizit als genügend groß angenommen
- `float` ein Gleitkomma Datentyp, die Genauigkeit genügt allen Anforderungen
- `string` ein Datentyp für Zeichenfolgen
- `string[]` eine Liste von Strings. Es gebe eine Funktion $append : string[] \times string \rightarrow string[]$, die einer Liste von Zeichenfolgen eine weitere Zeichenfolge anfügt. `[]` steht für eine leere Liste.
- `enum` ein Aufzählungstyp

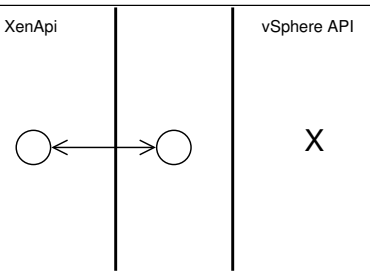
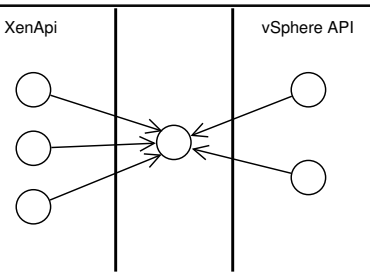
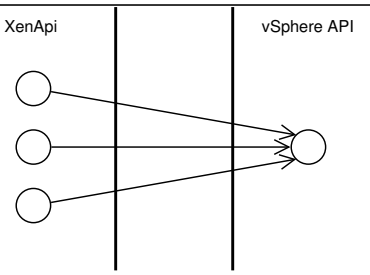
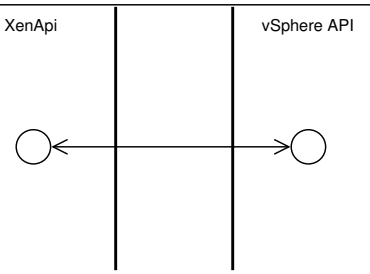
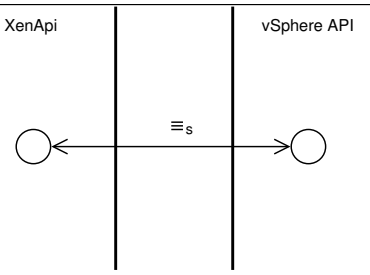
	Menge	Schematisches Beispiel	Formales Beispiel
K_0	$NABB(A_X, A_V)$		Das Attribut `Host.is_control_domain` von Xen kann nicht mittels Attributen der vSphere API abgebildet werden.
K_1	$ABB(A_X, A_V)$ bzw. $ABB(A_V, A_X)$		`Host_memoryUsed`
K_2	$A_{H_X \to H_V}$ bzw. $A_{H_V \to H_X}$		`Host_pCpuUtilisationMhz`
K_3	$BABB(A_X, A_V)$		`VM_memoryBytes`
K_4	$A_X \cap_s A_V$		`VM_name`

Tabelle 4.2.: Visualisierung der Klassifikationsmengen

4.2.1. Nicht abbildbare Attribute

Unter die nicht abbildbaren Attribute fallen insbesondere hypervisorspezifische, sowie bei keinem Hypervisor abrufbare Attribute. Da diese im Verlauf dieser Arbeit nicht weiter betrachtet werden sollen, wird hier nur eine kleine Auswahl dieser zahlreichen Attribute beispielhaft angegeben.

- Die aktuelle CPU Taktung ist bei keinem der Hypervisoren abfragbar.
- Das aktuelle VCPU→PCPU Mapping ist nur bei XEN abrufbar.
- Das Attribut `VM.is_control_domain` von Xen ist unter VMware nicht sinnvoll, da die Existenz einer Control Domain (Dom0) eine Besonderheit des Xen-Hypervisors ist. Ein alternatives Vorgehen wäre hier natürlich bei VMware immer *false* zurückzugeben. Dann wäre das Attribut abbildbar.

4.2.2. Trivial abbildbare Attribute

Wie ein Blick auf die folgenden Attribute zeigt, handelt es sich bei den trivial abbildbaren Attributen (nach Definition 7) hauptsächlich um Namensattribute, die als String gespeichert werden.

Host

`Host_MemoryTotal:int`
: Semantik: Größe des Speichers in Bytes
Xen: `host_metric.memory/total`
VMware: `HostSystem.hardware.memorySize`

`Host_Name:string`
: Semantik: Name des Hosts
Xen: `host.name/label`
VMware: `HostSystem.name`

`Host_PCpuModel:string`
: Semantik: Bezeichner für das CPU Modell des Hosts
Xen: `host_cpu.modelname`
VMware: `HostSystem.summary.hardware.cpuModel`

`Host_UUID:string`
: Semantik: UUID des Hosts
Xen: `host.uuid`
VMware: `HostSystem.summary.hardware.uuid`

VM

`VM_Description:string`
: Semantik: Beschreibung der VM
Xen: `VM.name/description`
VMware: `VirtualMachine.config.annotation`

`VM_IsTemplate:bool`
: Semantik: `true` falls es sich um ein Template für eine VM handelt
Xen: `VM.is_a_template`
VMware: `VirtualMachine.config.template`

`VM_Name:string`
: Semantik: Name der VM
Xen: `VM.name/label`
VMware: `VirtualMachine.name`

`VM_NumVCpu:int`
: Semantik: Anzahl der aktuell in dieser VM vorhandenen CPUs
Xen: `VM_metrics.VCPU/number`
VMware: `VirtualMachine.config.hardware.numCPU`

`VM_ResidentOnHostName:string`
: Semantik: Name des Hosts auf dem die VM läuft
Xen: `VM.resident_on`→`name`
VMware: `VirtualMachine.runtime.host`→`name`

`VM_ResidentOnHostUUID:string`
: Semantik: UUID des Hosts auf dem die VM läuft
Xen: `VM.resident_on`→`uuid`
VMware: `VirtualMachine.runtime.host`→`summary.hardware.uuid`

`VM_UUID:string`
: Semantik: UUID der VM
Xen: `vm.uuid`
VMware: `VirtualMachine.config.uuid`

4.2.3. Bijektiv abbildbare Attribute

Diese Klasse ist am schwächsten besetzt, da die meisten Attribute trivial abbildbar, oder aber nicht mehr vollständig nach Definition 6 bijektiv abbildbar sind.

VM

`VM_BootTime:dateTime`
: Semantik: Zeit zu der diese VM gestartet wurde
Xen: `VM_metrics.start_time`
VMware: `VirtualMachine.runtime.bootTime`
Dieses Attribut ist nicht trivial abbildbar, da der Typ `xsd:dateTime` zwar immer als ISO 8601 interpretiert werden kann, aber `xsd:dateTime` nicht alle Möglichkeiten von ISO 8601 zulässt. Die Abbildung kann aber einfach vorgenommen werden.

`VM_memoryBytes:int`
: Semantik: Der VM zugewiesener Speicher
Xen: `VM_metrics.memory/actual`
VMware:

$$f(\texttt{VirtualMachine.config.hardware.memoryMB}) = \texttt{memoryMB} * 1024^2$$

4.2.4. Einseitig abbildbare Attribute

Es wird auf Xen Mengen Typen (Set) folgende Hilfsfunktion definiert:

$$count : Set \rightarrow int \qquad a \mapsto \sum_{i \in a} 1$$

Des Weiteren wird auf den Schlüssel-Wert-Paaren (Map) folgende Hilfsfunktion benötigt:

$$\pi : (t_1, t_2)Map \times t_1 \rightarrow t_2 \qquad \pi_{x,i}(M) \mapsto \begin{cases} m_2, & \text{falls} \quad ((x, m_2) \in M) \\ i, & \text{falls} \quad (\forall y (x, y) \notin M) \end{cases}$$

Host

`Host_Enabled:bool`
: Semantik: Beschreibt ob eine Hostsystem läuft und virtuelle Maschinen auf diesem Host gestartet werden können
Xen: `host.enabled`
VMware: $f(\texttt{HostSystem.runtime.powerState}) =$

$$= \begin{cases} \texttt{true}, & \text{falls} \quad \texttt{powerState} = \texttt{poweredOn} \\ \texttt{false}, & \text{sonst} \end{cases}$$

`Host_MemoryFree:int`
: Semantik: Freier Arbeitsspeicher in Bytes
Xen: `Host.memory/free`
VMware:
$a :=$ `HostSystem.hardware.memorySize`
$b :=$ `HostSystem.summary.quickStats.overallMemoryUsage`

$$f(a, b) = a - b * 1024^2$$

`Host_noPhysicalNics:int`
Semantik: Anzahl der physischen Netzschnittstellen
Xen:

$$f(\texttt{host.host\_PIFs}) = count(host\_PIFs)$$

VMware: `HostSystem.summary.hardware.numNics`

`Host_PCpuThreads:int`
Semantik: Anzahl der CPU-Threads des Hosts
Xen:

$$\begin{aligned}(\texttt{host.PCpu\_configuration}) \quad = \quad & \pi_{\texttt{threads\_per\_core},1}(\texttt{cpu\_configuration}) \\ & \cdot\pi_{\texttt{cores\_per\_socket},1}(\texttt{cpu\_configuration}) \\ & \cdot\pi_{\texttt{sockets\_per\_node},1}(\texttt{cpu\_configuration})\end{aligned}$$

VMware: `HostSystem.summary.hardware.numCpuThreads`

`Host_PCpuCores:int`
Semantik: Anzahl der CPU-Cores des Hosts
Xen:

$$\begin{aligned}(\texttt{host.cpu\_configuration}) \quad = \quad & \pi_{\texttt{cores\_per\_socket},1}(\texttt{cpu\_configuration}) \\ & \cdot\pi_{\texttt{sockets\_per\_node},1}(\texttt{cpu\_configuration})\end{aligned}$$

VMware: `HostSystem.summary.hardware.numCpuCores`

`Host_PCpuMhz:int`
Semantik: Durchschnittliche Taktrate der Host-CPUs in Mhz
Xen:

$$f(\texttt{host.host\_CPUs}) = \left\lceil \frac{\sum_{i \in \texttt{host\_CPUs}} (i.\texttt{speed})}{count(\texttt{host\_CPUs})} - 0,5 \right\rfloor$$

VMware: `HostSystem.summary.hardware.cpuMhz`

`Host_PCpuUtilisationMhz:int`
Semantik: Auslastung der CPU in Mhz
Xen:

$$f(\texttt{host.host\_CPUs}) = \left\lceil \frac{\sum_{i \in \texttt{host\_CPUs}} (i.\texttt{utilisation} \cdot i.\texttt{speed})}{count(\texttt{host\_CPUs})} - 0,5 \right\rfloor$$

VMware: `HostSystem.summary.quickStats.overallCpuUsage`

VM

`VM_noVirtualNics:int`
Semantik: Anzahl der virtuellen Netzschnittstellen
Xen:

$$f(\texttt{host.VIFs}) = count(VIFs)$$

VMware: `VirtualMachine.summary.config.numEthernetCards`

4.2.5. Abbildbare Attribute

Host

`Host_AttachedNetworkNames:string[]`

Semantik: Liste der Namen aller Netze, zu denen dieser Host verbunden ist

Xen:

```
Host_AttachedNetworkNames = []
for i in Host.PIFs do
    append(Host_AttachedNetworkNames, i→network. name)
```

VMware:

```
Host_AttachedNetworkNames = []
for i in HostSystem.network do
    append(Host_AttachedNetworkNames, i→name)
```

`Host_CpuUtilisationPercentage:float`

Semantik: Auslastung aller CPUs des Hosts, repräsentiert als Gleitkommazahl zwischen 0 und 1

Xen:

$$f(\texttt{host.host\_CPUs}) = \frac{\sum_{i \in \texttt{host\_CPUs}} i.\texttt{utilisation}}{count(\texttt{host\_CPUs})}$$

VMware:

$a := \texttt{HostSystem.summary.quickStats.overallCpuUsage}$

$b := \texttt{HostSystem.summary.hardware.cpuMhz}$

$c := \texttt{HostSystem.hardware.cpuInfo.numCpuCores}$

$$f(a, b, c) = \frac{a}{b \cdot c}$$

`Host_MemoryUsed:int`

Semantik: Genutzter Arbeitsspeicher in Bytes

Xen:

$a := \texttt{host\_metrics.memory/total}$

$b := \texttt{host\_metrics.memory/free}$

$$f(a, b) = a - b$$

VMware:

$$f(\texttt{HostSystem.summary.quickStats.overallMemoryUsage}) =$$

$$\texttt{overallMemoryUsage} * 1024^2$$

Würde hier der genutzte Speicher in MiB gefordert werden, würde dieses Attribut zu den einseitig abbildbaren gehören. Dies ist aber insofern nicht sinnvoll, da für ein späteres Modell aus den normierten Attributen auf sinnvolle Grundeinheiten geachtet werden sollte; hierbei empfiehlt sich die Nutzung von Bytes, um Verwechslungen bei den 1000er bzw 1024er Potenzen auszuschließen.

`Host_PhysicalInterfaceMACs:string[]`

Semantik: Liste der MAC-Adressen der physischen Schnittstellen des Hosts

Xen:

```
Host_PhysicalInterfaceMACs = []
for i in Host.PIFs do
    append(Host_PhysicalInterfaceMACs, i→mac)
```

VMware:

```
Host_PhysicalInterfaceMACs = []
for i in HostSystem.config.network.pnic do
    append(Host_PhysicalInterfaceMACs, i.mac)
```

`Host_ResidentVMUUIDs:string[]`

Semantik: Liste der UUIDs der VMs auf diesem Host

Xen:

```
Host_ResidentVMUUIDs = []
for i in Host.resident_VMs do
    append(Host_ResidentVMUUIDs, i→uuid)
```

VMware:

```
Host_ResidentVMUUIDs = []
for i in HostSystem.vm
    append(Host_ResidentVMUUIDs, i→config.uuid)
```

`Host_ResidentVMNames:string[]`

Semantik: Liste der Namen der VMs auf diesem Host

Xen:

```
Host_ResidentVMNames = []
for i in Host.resident_VMs do
    append(Host_ResidentVMNames, i→name)
```

VMware:

```
Host_ResidentVMNames = []
for i in HostSystem.vm
    append(Host_ResidentVMNames, i→name)
```

VM

`VM_AttachedNetworkNames:string[]`

Semantik: Liste der Namen aller Netze zu denen dieser Host verbunden ist

Xen:

```
VM_AttachedNetworkNames = []
for i in VM.VIFs do
    append(VM_AttachedNetworkNames, i→network→name)
```

VMware:

```
VM_AttachedNetworkNames = []
for i in VirtualMachine.network do
    append(VM_AttachedNetworkNames, i→name)
```

`VM_PowerState`:enum{halted, paused, running, suspended, crashed, unknown}

Semantik: Der Ausführungszustand der virtuellen Maschine. Die einzelnen Zustände sind wie folgt analog zu Xen definiert:

0 *halted* - Die virtuelle Maschine ist ausgeschaltet bzw. heruntergefahren

1 *paused* - Die Ausführung der virtuellen Maschine wurde unterbrochen, ihr Zustand wird im Arbeitsspeicher des Hostsystems gehalten.

2 *running* - Die virtuelle Maschine wird ausgeführt.

3 *suspended* - Die Ausführung der virtuellen Maschine wurde unterbrochen, ihr Zustand ist auf nichtflüchtige Speichermedien gesichert.

4 *crashed* - Ein Absturz des Gastbetriebssystems wurde erkannt, die virtuelle Maschine wird nicht ausgeführt.

5 *unknown* - Der Zustand der virtuellen Maschine ist unbekannt

Xen: g(`VM.power_state`); wobei g eine 1:1 Abbildung der Stringbezeichner von Xen auf Bezeichner des Aufzählungstyps des generischen Attributs vornimmt. Diese Abbildung ist bijektiv.

VMware:

$a :=$ `VirtualMachine.runtime.powerState`

$$f(a) = \begin{cases} \texttt{halted}, & \text{falls} \quad a = \texttt{poweredOff} \\ \texttt{running}, & \text{falls} \quad a = \texttt{poweredOn} \\ \texttt{suspended}, & \text{falls} \quad a = \texttt{suspended} \end{cases}$$

Diese Abbildung ist zwar injektiv, aber nicht surjektiv.

`VM_Running:bool`

Semantik: Beschreibt, ob eine virtuelle Maschine Laufzeitressourcen, also Ressourcen außer Hintergrundspeicher, benötigt.
Xen: $f(\texttt{VM.power\_state}) =$

$$= \begin{cases} \texttt{true}, & \text{falls} \quad \texttt{power\_state} \in \{\texttt{running}, \texttt{paused}\} \\ \texttt{false}, & \text{sonst} \end{cases}$$

VMware: $f(\texttt{VirtualMachine.runtime.powerState}) =$

$$= \begin{cases} \texttt{true}, & \text{falls} \quad \texttt{powerState} = \texttt{poweredOn} \\ \texttt{false}, & \text{sonst} \end{cases}$$

`VM_SafeHostPowerOff:bool`

Semantik: Beschreibt, ob diese Maschine in einem Zustand ist, in dem der Host ohne Gefahr eines Datenverlusts ausgeschaltet werden darf.
Xen: $f(\texttt{VM.power\_state}) =$

$$= \begin{cases} \texttt{true}, & \text{falls} \quad \texttt{power\_state} \in \{\texttt{halted}, \texttt{suspended}\} \\ \texttt{false}, & \text{sonst} \end{cases}$$

VMware: $f(\texttt{VirtualMachine.runtime.powerState}) =$

$$= \begin{cases} \texttt{true}, & \text{falls} \quad \texttt{powerState} \in \{\texttt{poweredOff}, \texttt{suspended}\} \\ \texttt{false}, & \text{sonst} \end{cases}$$

`VM_VirtualInterfaceMACs:string[]`

Semantik: Liste der MAC-Adressen der virtuellen Schnittstellen der virtuellen Maschine.
Xen:

```
VM_VirtualInterfaceMACs = []
for i in Host.VIFs do
    append(VM_VirtualInterfaceMACs, i→mac)
```

VMware:

```
VM_VirtualInterfaceMACs = []
for i in VM.config.hardware.device do
    if (type_of(i) == 'VirtualEthernetCard') then
        append(VM_VirtualInterfaceMACs, i.macAddress)
```

4.2.6. Weitere Ableitungen aus den klassifizierten Attributen

Aus der Klassifikation folgt, dass Attribute, die nicht aus der Klasse der bijektiv abbildbaren Attribute stammen, unter bestimmten Umständen – meist durch Zuhilfenahme anderer Attribute – trotzdem in beiden Richtungen abgebildet werden.

Ein Beispiel hierfür wären die Attribute `Host_MemoryUsed` und `Host_MemoryFree`. Diese sind als abbildbar beziehungsweise einseitig abbildbar zu klassifizieren. Allerdings hängen beide über das bijektiv abbildbare Attribut `Host_MemoryTotal` zusammen. Wird das bijektiv abbildbare Attribut `Host_MemoryTotal` als gegeben vorausgesetzt, so kann

auch der genutzter Speicher bijektiv auf den freien Speicher abgebildet werden.

Das 2-Tupel (`Host_MemoryTotal`, `Host_MemoryUsed`) kann bijektiv auf das 2-Tupel (`Host_MemoryTotal`, `Host_MemoryFree`) abgebildet werden.

4.2.7. Verfügbarkeit der Attribute

Auch die Verfügbarkeit der Attribute ist eine wichtige Nebenbedingung. So können manche Attribute nur gültig sein, wenn ein Managed Objekt in Betrieb ist, also ein Host oder eine VM eingeschaltet sind. So sind fast alle Hostattribute mit Ausnahme des Namens, der UUID, und des Zustands, ob dieser Host läuft, nur sinnvoll, falls der Host aktiv ist. Bei virtuellen Maschinen kann wiederum die BootTime undefiniert sein, falls eine virtuelle Maschine noch nicht gebootet wurde. Auch die Attribute, die beschreiben, auf welchem Host eine virtuelle Maschine ausgeführt wird, sind nur sinnvoll, wenn die virtuelle Maschine nicht *halted* oder *suspended* ist. Alle anderen Attribute einer virtuellen Maschine sind immer gültig.

Auch mögliche Benutzerrechteverwaltungen können Attribute nicht zugänglich machen. Hierbei ist es wichtig, dass die Funktionen in einer Implementierung adäquate Rückmeldungen liefern, wenn ein solcher Fall vorliegt. Während bei nicht vorhandenen Rechten durchaus auch ein Funktionsabbruch mit Fehler in Frage kommt, wäre dies bei gerade ungültigen Attributen nicht sinnvoll.

Des Weiteren muss untersucht werden, inwiefern die Attribute schreibbar sind, oder ob auf diese nur lesend zugegriffen werden kann. Auch bijektiv abbildbare Attribute müssen nicht schreibbar sein. Tabelle 4.3 gibt für jedes der untersuchten Attribute an, ob ein Schreibzugriff generell denkbar und bei den Hypervisoren möglich ist. Hier tritt sehr oft der Fall ein, dass ein Attribut nicht direkt manipulierbar ist, sondern nur mittels eines RPCs verändert werden kann, der sich nicht auf das Setzen des Attributs beschränkt. Beispielsweise verändert ein Shutdown RPC den Powerstate einer VM, aber dies ist kein reines Setzen des Attributs, da ein anderer RPC benötigt wird, um die VM wieder zu booten. Ein weiteres Beispiel wäre die Anzahl der virtuellen Schnittstellen. Diese ist durch hinzufügen oder löschen einer virtuellen Schnittstelle veränderbar, allerdings sind hierzu weitere Informationen nötig, die nicht durch die hier vorgestellten Attribute abgedeckt werden können.

In diesem Kapitel wurde ein Klassifikationsschema vorgestellt und die praktische Durchführbarkeit der Klassifikation an den Beispielhypervisoren gezeigt. Die in Abschnitt 4.2 gesammelten und normierten Attribute bilden die Grundlage für die prototypische Implementierung eines Attributsstamms inklusive der benötigten Abbildungen auf Basis von libvirt.

Attributsname	Zugriff generell	Zugriff XenApi	Zugriff VMware
Host_PCpuCores	*ro*	*ro*	*ro*
Host_PCpuMhz	*ro*	*ro*	*ro*
Host_PCpuModel	*ro*	*ro*	*ro*
Host_PCpuThreads	*ro*	*ro*	*ro*
Host_PCpuUtilisationMhz	*ro*	*ro*	*ro*
Host_PCpuUtilisationPercentage	*ro*	*ro*	*ro*
Host_Enabled	*rw*	$*^1$	$*^1$
Host_MemoryFree	*ro*	*ro*	*ro*
Host_MemoryTotal	*ro*	*ro*	*ro*
Host_MemoryUsed	*ro*	*ro*	*ro*
Host_Name	*rw*	*rw*	*rw*
Host_NumOfPhysicalNics	*ro*	*ro*	*ro*
Host_PhysicalInterfaceMACs	*rw*	*rw*	*nb*
Host_ResidentVMNames	*rw*	*rw*	*rw*
Host_ResidentVMUUIDs	*ro*	*ro*	*ro*
Host_UUID	*ro*	*ro*	*ro*
VM_BootTime	*ro*	*ro*	*ro*
VM_Description	*rw*	*rw*	*rw*
VM_IsTemplate	*rw*	*rw*	*rw*
VM_MemoryBytes	*rw*	$*^2$	*rw*
VM_Name	*rw*	*rw*	*rw*
VM_NumVCpu	*rw*	$*^2$	*rw*
VM_NumVirtualNics	*rw*	$*^1$	$*^1$
VM_PowerState	*rw*	$*^1$	$*^1$
VM_ResidentOnHostName	*rw*	*rw*	*rw*
VM_ResidentOnHostUUID	*ro*	*ro*	*ro*
VM_Running	*ro*	*ro*	*ro*
VM_SafeHostPowerOff	*ro*	*ro*	*ro*
VM_UUID	*ro*	*ro*	*ro*
VM_VirtualInterfaceMACs	*rw*	*rw*	*rw*

Legende:
ro nur Lesezugriff möglich.
rw Lese- und Schreibzugriff möglich.
$*^1$ Direktes Setzen nicht möglich, aber über RPCs veränderbar.
$*^2$ Direktes Setzen nicht möglich, aber durch andere Attribute beeinflussbar.
nb nicht bekannt

Tabelle 4.3.: Schreibverfügbarkeit der Attribute

5. Entwurf und Implementierung eines Prototypen

Als Teil dieser Arbeit soll eine prototypische Implementierung geschaffen werden, die Attribute von VMware und Xen Hypervisoren – über die APIs VI-SDK beziehungsweise XenApi – abfragen, abbilden und normalisieren kann. Diese Implementierung soll auf libvirt in Version 0.8.4 aufsetzen.

5.1. Entwurf des Agenten

Bevor die Implementierung praktisch umgesetzt wird, erfolgt eine Untersuchung der Bibliothek um die Möglichkeiten einer Implementierung auszuloten und eine geeignete Schnittstelle zur Implementierung zu finden.

5.1.1. Schnittstellen der libvirt

Als Erstes sollten mögliche Schnittstellen von libvirt untersucht werden, um eine Möglichkeit zu finden, Normalisierungserweiterungen zu implementieren. Als grundlegende Orte, an denen angesetzt werden kann, bieten sich folgende drei Möglichkeiten an:

1. Implementierung als Schicht unterhalb von libvirt, d.h. als Schnittstelle zwischen Hypervisor und libvirt,
2. Implementierung innerhalb der libvirt,
3. Implementierung als Schicht oberhalb von libvirt, als Bibliothek die ihrerseits libvirt nutzt.

Bei den Alternativen unter beziehungsweise oberhalb von libvirt entsteht der Vorteil, dass der Quellcode von libvirt nicht oder nur minimal angepasst werden müsste. Allerdings ermöglichen diese Varianten bei genauerer Betrachtung nur eine sehr kleine Menge an Attributen, nämlich die, die schon mittels libvirt abfragbar sind. Diese bilden eine kleine Teilmenge der möglichen durch die Hypervisoren abfragbaren Attribute. Daher ist eine Integration in den Quellcode der libvirt unumgänglich.

Eine Codeanlyse der libvirt zeigt eine dreischichtige Softwarestruktur, die in Abbildung 5.1 dargestellt wird. Die Bibliotheksaufrufe bilden die oberste Schicht; darunter befindet sich eine Abstraktionsschicht, die mittels einer `struct` von Funktionszeigern realisiert wird; die unterste Schicht bilden die Instanziierungen der Abstraktionsschicht. „Applications can use libvirts public API, which internally maps to appropriate driver functions through an internal driver API." [BSB+10]

Die Nutzung von libvirt setzt immer einen zentralen Treiber voraus, es können aber nebenher noch weitere zusätzliche spezialisierte Treiber geladen werden. Im Fall des zentralen Treiber bildet die `_virDriver` Struktur – definiert in `driver.h` – die Abstraktionsschicht (vgl.

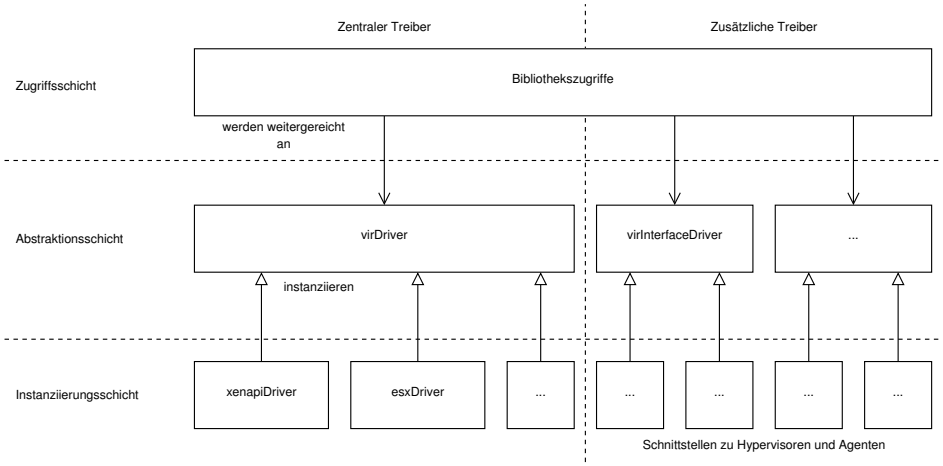


Abbildung 5.1.: Struktur der libvirt im Urzustand

Listing 5.1). Diese wird durch die unterschiedlichen hypervisorspezifischen Treiber instanziiert, die die Schnittstelle zu den Hypervisoren bilden. Jede Instanziierung durch einen Hypervisortreiber setzt die Funktionszeiger entsprechend auf die hypervisorspezifischen Funktionen. Analog zum zentralen Treiber gibt es mehrere Zusatztreiber, die nach demselben Abstraktionsschema aufgebaut sind (siehe Abbildung 5.1). Diese Zusatztreiber sind allerdings nicht immer an Hypervisoren gebunden, sondern können teilweise auch mittels entfernter eigenständiger Agenten implementiert sein. Für die Normalisierungserweiterungen bietet es sich an, ebenfalls einen solchen Zusatztreiber zu implementieren, der, analog zum zentralen Treiber, vom Hypervisor abhängt (vgl. Abbildung 5.2).

```
1  typedef virDrvOpenStatus
2          (*virDrvOpen)     (virConnectPtr conn,
3                             virConnectAuthPtr auth,
4                             int flags);
5  typedef int
6          (*virDrvClose)      (virConnectPtr conn);
7  typedef int
8      (*virDrvDrvSupportsFeature) (virConnectPtr conn, int feature);
9  typedef const char *
10         (*virDrvGetType)    (virConnectPtr conn);
11 typedef int
12         (*virDrvGetVersion)   (virConnectPtr conn,
13                                    unsigned long *hvVer);
14 /*[...]*/
15
16 /**
17  * _virDriver:
18  *
19  * Structure associated to a virtualization driver, defining the various
20  * entry points for it.
21  *
22  * All drivers must support the following fields/methods:
23  *  - no
24  *  - name
25  *  - open
26  *  - close
```

```
27  */
28 struct _virDriver {
29     int         no;  /* the number virDrvNo */
30     const char * name;  /* the name of the driver */
31     virDrvOpen      open;
32     virDrvClose     close;
33     virDrvDrvSupportsFeature   supports_feature;
34     virDrvGetType     type;
35     virDrvGetVersion    version;
36     /*[...]*/
37 };
```

Listing 5.1: Struktur eines `_virDriver` aus der Quelldatei `src/driver.h` (gekürzt)

5.1.2. Implementierungsidee der Normalisierungserweiterung

Wie bereits festgestellt, sind nur wenige Attribute via libvirt abfragbar. Ziel ist es, die Anzahl der abfragbaren Attribute zu erhöhen, und dabei eine Normalisierung über mehrere Hypervisoren hinweg vorzunehmen. Des Weiteren sollten die Eingriffe in die libvirt minimal-invasiv ausfallen und strukturell an bisherige Teile der libvirt angelehnt sein, um eine eventuelle spätere Aufnahme in den Hauptentwicklungszweig zu ermöglichen. Um die abfragbaren Attribute möglichst flexibel zu halten und durch die Menge der möglichen Hypervisoren nicht von vornherein stark einzuschränken, sollte die Möglichkeit gegeben sein, je nach Hypervisor nur eine Teilmenge der normalisierten Attribute zu implementieren und schließlich als gültig zu kennzeichnen. Aus diesem Grund wird in dieser Arbeit folgender Implementierungsansatz vorgeschlagen (vgl. Abbildung 5.2):

- Erweiterung der Bibliotheksaufrufe um weitere Aufrufe, um normalisierte Attribute nach außen zur Verfügung zu stellen;
- Einführung einer `struct _virNormalizationDriver`, die analog zu den schon bestehenden libvirt Treibern implementiert wird, da sich dieser Ansatz bewährt: ,,This driver model has proven to be very flexible and easy to extend, so drivers for storage and network interfaces using the same model have been added.”[BSB+10]
- Hypervisorspezifische Instantiierung des Treibers.

Alle weiteren bestehenden Bibliotheksaufrufe bleiben von diesen Änderungen unberührt.

5.2. Notwendige vorausgehende Anpassungen der libvirt

Der Code zur Unterstützung von XenApi und VMware ist in libvirt Version 0.8.4 noch experimentell. Während der VMware spezifische Code stabil zu sein scheint und keine ungewöhnlichen Abhängigkeiten besitzt, wurde der XenApi-Code hauptsächlich von Citrix beigesteuert. Die APIs des Citrix XenServers und der Open-Source-Implementierung von Xen sind leicht verschieden, sodass es hier bei der Verwendung des Open-Source Xen zu Problemen kommen könnte. Außerdem hängt der Code von der Bibliothek libxenserver ab, deren Code zwar von Citirx unter der GNU LGPL Lizenz verteilt wird, wobei allerdings (zumindest in der getesteten Version Version 5.6.0-1) der mitgelieferte Makefile nicht voll funktionsfähig ist. Citrix

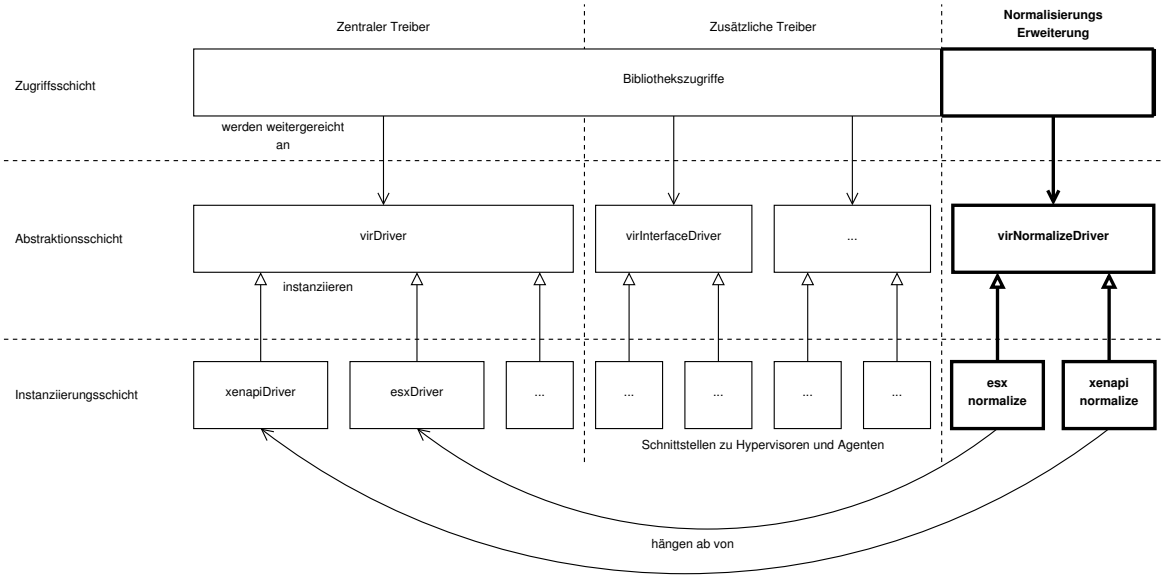


Abbildung 5.2.: Struktur der libvirt mit Normalization Layer

selbst empfiehlt die Verwendung eines Live Images eines XenServers zur Entwicklung mit dieser Bibliothek.

Daher wurde eine Portierung des XenApi Treibers in libvirt von libxenserver auf libxen vorgenommen. Hierzu mussten hauptsächlich die Makefiles der libvirt angepasst werden. Zusätzlich mussten auf Grund der kleinen Unterschiede in der API einige wenige Funktionsaufrufe innerhalb der libvirt angepasst werden. Hierbei wurde im Rahmen der Arbeit nur darauf geachtet, dass die grundlegende Funktionalität gewahrt bleibt und pragmatische Abstriche gemacht. Vor einer Übernahme in den Hauptentwicklungszweig muss die Funktionalität in manchen Teilen noch getestet bzw. ausgebessert werden (siehe Abschnitt 5.4).

Auf der anderen Seite wird die Kompatibilität mit dem Open-Source Xen Hypervisor deutlich erhöht. Vergleiche von Schlüsseln in den *Map* Typen werden nun – im Gegensatz zur ursprünglichen Version– ohne Beachtung der Groß-/Kleinschreibung durchgeführt. Die Schreibweisen, die bei Citrix gültig waren, waren in der Open-Source Variante nicht zwingend gültig. Außerdem wird nun der TCP-Port in der an libvirt übergebenen URI ausgewertet und die Verbindung zu diesem, statt zu einem fest vorgegebenen Standardport aufgebaut.

Des Weiteren wurde die schon in libvirt vorhandene Normalisierungsfunktion `virParseVersionString` angepasst, die formatierte Versionstrings in numerische Werte umwandelt, da diese in ihrer ursprünglichen Form nur mit Versionsbezeichnungen im Format `major.minor.micro[.suffix]` umgehen konnte (vgl. Listing 5.2). Versionsbezeichnungen, die nur aus `major.minor` bestanden, führten zu einem Funktionsabbruch. Solche trivialen Beispiele zeigen, wie wichtig ein umfassender Blick auf Attribute und deren Abbildbarkeit ist.

```
1 @@ -2177,15 +2178,22 @@ virParseVersionString(const char *str, unsigned long *
      version)
2
3      if (virStrToLong_ui(str, &tmp, 10, &major) < 0 || *tmp != '.')
4          return -1;
5
```

```
 6       if (virStrToLong_ui(tmp + 1, &tmp, 10, &minor) < 0 || *tmp != '.')
 7 -         return -1;
 8 +     {
 9 +         if (*tmp != '\0')
10 +                 return -1;
11 +         else
12 +                 minor = 0;
13 +                 goto calc;
14 +     }
15
16       if (virStrToLong_ui(tmp + 1, &tmp, 10, &micro) < 0)
17           return -1;
18
19 +calc:
20       *version = 1000000 * major + 1000 * minor + micro;
21
22       return 0;
23   }
```

Listing 5.2: Patch der Funktion `virParseVersionString`

5.3. Umsetzung der Implementierung

Zur Umsetzung der eigentlichen Implementierung wurden im Wesentlichen folgende Schritte durchgeführt:

- Anpassung der Makefiles, um die neuen Quelldateien einzupflegen;
- Erweiterung der Bibliotheksaufrufe in `libvirt.c`;
- Erweiterung der zugehörigen Deklarationen in `libvirt.h.in`;
- Hinzufügen der neuen Symbole in `libvirt_public.syms`;
- Definition der Struktur `_virNormalizeDriver` und aller zugehörigen Funktionszeiger in `driver.h`;
- Registrierung der Normalisierungstreiber bei Initialisation der Bibliothek
- Hypervisorspezifische Teile der Implementierung in `xenapi_normalize` beziehungsweise in `esx_normalize`;
- Implementierung des virTester Programms, das als Demoanwendung fungiert.

Darüber hinaus wurden noch zusätzliche Datentypen definiert, sowie im VMware-spezifischen Teil kleinere funktionale Ergänzungen implementiert, um weitere Attribute des Hypervisors abfragen zu können. Die gesamte Kommunikation mit dem Hypervisor ist in diesem Teil der libvirt implementiert, da bei VMware im Gegensatz zu Xen mit libxen zurückgegriffen werden kann. Diese Implementierung kann außerdem für andere Projekte auch losgelöst von libvirt zur Realisierung der Kommunikation mit einem VMware Hypervisor eingesetzt werden.

Analog zu anderen Zusatztreibern, werden alle implementierten Normalisierungstreiber beim Aufruf von `virInitialize()` registriert. Beim Öffnen einer Verbindung zu einem

Hypervisor greift libvirt intern immer auf die Funktion `do_open()` zurück, in der sowohl die Verbindung mittels des zentralen Treibers zum Hypervisor, als auch die Verbindungen aller Hilfstreiber aufgebaut werden. Die Entscheidung, ob ein Treiber für eine Verbindung geeignet ist, trifft der Treiber selbst. Die zentralen Treiber entscheiden das anhand der übergebenen URI. Der Normalisierungstreiber muss an dieser Stelle überprüfen, ob der dazugehörige Hypervisortreiber genutzt wird. Dies wird bewerkstelligt, indem er überprüft, ob sein Strukturelement `no` mit dem Strukturelement `no` des zentralen Treibers übereinstimmt.

Alle Bibliotheksaufrufe haben die in Listing 5.3 angegebene Form.

```
1 virNormReturnCode virNorm{Domain|Node}Get<Attributsname>({virDomainPtr dom|
    virConnectPtr conn}, <type>* ret)
```

Listing 5.3: Allgemeine Deklaration einer Bibliotheksfunktion des Normalisierungstreibers

Bei `virNormReturnCode` handelt es sich um einen Aufzählungstyp, der Zustände für Fehler- und Erfolgsfall, sowie eine Möglichkeit für einen Zustand Unset kennt. Dieser Zustand kann später dazu genutzt werden, um Zugriffsverweigerungen aufgrund beschränkter Rechte oder nicht gesetzter Werte auszudrücken; aktuell ist dieser meist ungenutzt. Für ein gültiges Ergebnis muss der Rückgabewert mit `VIR_NORM_DRV_SUCCESS` übereinstimmen.

`Domain` drückt aus, dass sich dieses Attribut auf eine virtuelle Maschine bezieht. In diesem Fall muss zum Aufruf ein `virDomainPtr` übergeben werden. `Node` drückt aus, dass es sich hierbei um ein Attribut des Hosts, zu dem aktuell eine Verbindung besteht, handelt. In diesem Fall wird die Funktion mit einem `virConnectPtr` als erstem Argument aufgerufen. Die Möglichkeiten, Rechnerpools zu verwalten, sind in libvirt noch nicht weit genug ausgebaut, um sinnvoll auf mehrere Hosts zuzugreifen, daher werden Hostattribute momentan auf den aktuell verbundenen Host bezogen.

Das zweite Argument ist ein Zeiger auf ein Objekt, das als Rückgabe dient. Hierbei werden folgende Datentypen für `<type>` genutzt:

- für ganzzahlige Werte `int64_t`
- für Gleitkommazahlen `double`
- für Zeichenfolgen `char*`
- für Listen von Zeichenfolgen `virCharList*`
- für Wahrheitswerte `bool`
- für Datums-/Zeitangaben `time_t`
- für powerState von virtuellen Maschinen `virNormDomainPowerState`

Hierbei gilt, sollte `<type>` schon ein Zeigertyp sein – also bei `char*` und `virCharList*` –, so reserviert die Bibliothek selbstständig den Speicher. Ansonsten hat die Anwendung für die Allokation des Speichers zu sorgen. Die Freigabe von durch die Bibliothek reserviertem Speicher obliegt der Anwendung und geschieht mittels `free()` bei normalen Zeichenfolgen, bei `virCharList*` ist die Bibliotheksfunktion `virCharListFree` zu nutzen.

Die Einzelheiten der Implementierungen sprengen den Umfang der schriftlichen Ausarbeitung und können den Quelldateien entnommen werden. Die Implementierung deckt den gesamten Umfang, der in dieser Arbeit klassifizierten Attribute ab. Ein Ergebnis, das nicht

ganz unerwähnt bleiben sollte, ist, dass sich der Code der einzelnen Normalisierungsfunktionen bei VMware trotz der aufwändigeren API etwas besser generalisieren lässt.

Unter Debian Squeeze testing und Ubuntu 10.10 ist die Implementierung mit den in Listing 5.4 aufgeführten Befehlen kompilierbar.

```
1  ./autogen.sh --with-xenapi --with-esx --with-normalize --without-vbox --without
       -driver-modules --without-qemu --without-lxc --without-xen --without-openvz
        --without-uml --without-test --without-remote --without-network --without-
       python
2  make
3  make install
```

Listing 5.4: Kompilieranleitung

5.4. Bekannte Fehler

An dieser Stelle werden die bekannten Probleme der Implementierung aufgelistet:

- Durch Portierung von libxenserver auf libxen sind einige Funktionen besonders betroffen. Hierbei handelt es sich um folgende Funktionen:
  - `xenapiDomainDumpXML` (`xenapi_driver.c`)
  - `createVifNetwork` (`xenapi_utils.c`)
  - `createVMRecordFromXml` (`xenapi_utils.c`)

  Ihre Funktionalität bleibt im Rahmen dieser Arbeit ungetestet. Für die hier implementierte Erweiterung sind diese Funktionen unerheblich. Sollte ein Programm auf libvirt aufsetzen und von diesen Funktionen abhängen, so kann dies zu unvorhergesehenem Verhalten führen. In Verbindung mit einem Citirx XenServer wäre die Erweiterung auch ohne diese Portierung lauffähig, allerdings wurde als Testsystem ein Open Source Xen System auf Debian Basis gewählt.

- Teilweise kann es bei Verwendung des XenApi-Treibers, sowie der XenApi-Normalisierungserweiterung zur Ausgabe einer Fehlermeldung nach diesem Muster kommen: "Struct did not contain expected field %s.\n". Diese werden von libxen erzeugt, da in der zum Test genutzten Xen Version 4.0.1 (Debian Sqeeze testing) ein Datensatz nicht vollständig zurückgeliefert wird. Durch folgenden Patch der Datei `tools/libxen/src/xen_common.c` wird diese Ausgabe nach einer Neukompilierung von libxen unterdrückt:

- die Funktion `virNormDomainGetDescription` liefert bei Verwendung von XenApi, wiederum in Verbindung mit Xen Version 4.0.1 (Debian Sqeeze testing) einen Fehler, da hier eine Lücke in der Implementierung der spezifizierten API auf Hypervisorseite vorliegt.

- Manchmal liefert der Disconnect des Hypervisortreibers einen Fehler. Die Ursache hierfür ist unbekannt.

```
1 @@ -989,9 +989,9 @@
2                  if (j == seen_count)
3                  {
4  #if PERMISSIVE
5 -                    fprintf(stderr,
6 +                    /*fprintf(stderr,
7                              "Struct did not contain expected field %s.\n",
8 -                            mem->key);
9 +                            mem->key);*/
10 #else
11                   server_error_2(s,
12                                  "Struct did not contain expected field",
```

Listing 5.5: Workaround zur Unterdrückung von Fehlermeldungen durch libxen

5.5. Erweiterungsmöglichkeiten der Implementierung

Die Implementierung ist in mehreren Dimensionen erweiterbar. Die wichtigsten hierbei wären:

- Ergänzung weiterer Hypervisoren
- Ergänzung weiterer Attribute
- Ergänzung um schreibenden Zugriff auf Attribute

Eine Ergänzung um weitere Hypervisoren erfordert die hypervisorspezifische Implementierung des `_virNormalizeDriver`, die Registrierung dieses Treibers in `virInitialize()` sowie das Einpflegen in die Makefiles.

Zur Erweiterung um weitere Attribute oder zur Erweiterung um schreibenden Zugriff müssen die in Abschnitt 5.3 beschriebenen Schritte durchgeführt werden, wobei eine Änderung der Makefiles nur bei Hinzufügen weiterer Quelldateien nötig wird. Diese Erweiterung kann durch ein Skript, das sich in den Quellen unter `libvirt/helpers/` befindet, unterstützt werden. Genauere Erläuterungen finden sich in der Readme Datei im selben Ordner.

6. Ergebnisse

Rückblickend wurden in der Arbeit nach gewisser Vorarbeit, wie in Kapitel 3 beschrieben, Attribute der Hypervisoren Xen und VMware ESXi gesammelt und deren Semantik experimentell und durch Recherche in den API Referenzen bestimmt. Der hieraus in Abschnitt 3.2 gewonnene Attributskatalog (siehe Anhang A) dient als Referenz für die Semantik dieser Attribute. Auch für sich allein gesehen kann er diese Aufgabe erfüllen. Dieser Katalog bestätigt die Voraussetzung über die gegebene Heterogenität der Attribute bei unterschiedlichen Virtualisierungslösungen, die den Anlass für diese Arbeit gegeben hat. Auf nicht im Rahmen dieser Arbeit untersuchten Gebieten, wie Speichersysteme oder Ressource Pools, gibt es weitere Attribute, die für das Management interessant sein können. Die semantische Untersuchung der Attribute muss von Hand durchgeführt werden und ist mühsam, da weder einheitliche Beschreibungssprachen vorliegen, noch die Beschreibungen der Attribute in den einschlägigen Referenzen aussagekräftig genug sind, um eine genaue Semantik direkt daraus abzuleiten. Hier wäre Platz für Verbesserungen, wenn bei Erstellung solcher Dokumente darauf geachtet wird, dass die Sematik der Attribute und alle Abhängigkeiten und Zusammenhänge eindeutig dokumentiert wird.

In Kapitel 4 wird ein Klassifikationsschema anhand der Abbildbarkeit der Attribute vorgestellt. Dieses kann zur Bewertung der möglichen darstellbaren Attribute dienen. Nicht abbildbare Attribute sind ungeeignet, um als Basis eines gemeinsamen Informationsmodells zu dienen. Allgemein sollte dieses aus abbildbaren Attributen bestehen, wobei es für die meisten Anwendungsfälle günstig ist, dass diese Abbildungen umkehrbar sind, da dann alle darauf abgebildeten Attribute rückgewonnen werden und somit beispielsweise auch schreibenden Zugriff ermöglichen. Dieses Klassifizierungsschema ist nicht auf den Bereich Virtualisierung beschränkt, sondern ist generell auf Attribute aus unterschiedlichen Datenquellen anwendbar.

In Abschnitt 4.2 wird diese Klassifikation auf die in Abschnitt 3.2 gewonnenen Attribute angewendet. Hierbei ergeben sich Abbildungen zwischen den Virtualisierungslösungen, die somit einen Attributsstamm bilden, der für ein gemeinsames Management herangezogen werden kann. Die Abbildungen stellen das Hauptproblem der Klassifikation dar. Die Suche nach den Abbildungen setzt ein semantisches Verständnis der Attribute und deren Zusammenhänge voraus. Auf Basis der abgebildeten Attribute kann anschließend ein Modell aufgebaut werden, das als Grundlage für ein hypervisorübergreifendes Management dient.

Darauf aufbauend wird in Kapitel 5 ein Prototyp auf Basis von libvirt erstellt, der die Umsetzbarkeit der in dieser Arbeit vorgeschlagenen Arbeitsweise aufzeigt, als Nebeneffekt die Funktionalität der libvirt erweitert und den Anwendungsbereich der libvirt ausdehnt.

Aufgrund der vielfältigen Einsatzmöglichkeiten des Prototypen in darauf aufsetzenden Projekten wird eine Integration desselben in den Hauptentwicklungszweig der libvirt angestrebt. Mit Ausnahme der Portierung des XenApi Codes auf libxen dürfte die Aufnahme der Normalisierungserweiterung möglich sein, die selbst nicht direkt von der Portierung abhängig ist.

Die Arbeit zeigt beginnend mit der semantischen Beschreibung, der Klassifikation der At-

tribute und der damit verknüpften Suche nach Abbildungsmöglichkeiten zwischen den Attributen der unterschiedlichen Hypervisoren, bis hin zur Implementierung des abgebildeten Attributsstamms in einer Bibliothek, die Machbarkeit der Idee, die dieser Arbeit zugrunde liegt. Es ist möglich das Problem eines gemeinsamen Modells für das hypervisorübergreifende Management, von unten nach oben anzugehen. Hierbei werden alle nötigen Attribute beschrieben und abgebildet. Ein deutlicher Vorteil dieses Ansatzes im Vergleich mit dem Top-Down Ansatz bei CIM ist die sofortige Umsetzbarkeit in einer Implementierung. Alle dafür nötigen Informationen werden direkt durch das konstruktive Vorgehen gewonnen.

6.1. Vorschläge für weiterführende Arbeiten

Die drei zentralen Punkte dieser Arbeit, der Attributskatalog, der Attributsstamm klassifizierter Attribute, sowie die Implementierung bilden einen Ausgangspunkt für weitere Arbeiten. Es können weitere Virtualisierungslösungen betrachtet werden und – mittels der dort abfragbaren Attribute – die Klassifikation angepasst werden, so dass sie weitere Hypervisoren berücksichtigt. Auch eine Erweiterung um weitere Managed Objects und andere Attributsbereiche ist denkbar.

Nach vorhergegangener Untersuchung weiterer Virtualisierer kann ein Schritt in Richtung Informations-Modell oder eine Abbildung auf CIM erfolgen.

Auch Erweiterungen der Implementierung um schreibenden Attributszugriff, weitere Hypervisoren sowie Attribute ist denkbar. Hierzu enthält diese Arbeit in Abschnitt 5.5 Hilfestellungen.

Des Weiteren ist die Implementierung höherer Managementanwendungen, die die nun normalisiert abfragbaren Attribute praktisch nutzen, um Managementprozesse zu unterstützen, ein Anwendungsgebiet, das durch diese Arbeit eröffnet wurde.

A. Attributskatalog

Der folgende Attributskatalog basiert auf den Api-Referenzen der beiden Hypervisoren. [MSS+10][VMw10]

A.1. XenApi

Die folgenden Tabellen wurden um Referenzen auf nicht behandelte Klassen und bei VMware auch um einige sehr spezifische Attribute gekürzt. (nwb) steht für Attribute, die im Rahmen dieser Arbeit nicht weitergehend betrachtet wurden.

A.1.1. Klasse host

uuid	string	Global eindeutiger Objektbezeichner
name/label	string	Name
name/description	string	Beschreibung des Hosts
API_version/major	int	Major Versions Nummer (nwb)
API_version/minor	int	Minor Versions Nummer (nwb)
API_version/vendor	string	Herstelleridentifikation (nwb)
API_version/↵ vendor_implementation	(string →string) Map	Details über die Hersteller Implementation (nwb)
enabled	bool	*1
software_version	(string →string) Map	*2
other_config	(string →string) Map	Zusätzliche Konfigurationsmöglichkeiten (nwb)
capabilities	string Set	*3
cpu_configuration	(string →string) Map	*4
sched_policy	string	Aktuelle Scheduling Policy (nwb)
supported_↵ bootloaders	string Set	Liste aktuell auf dem Host unterstützter Bootloader (nwb)
resident_VMs	(VM ref) Set	Referenzen auf alle aktuell auf dem Host befindlichen VMs
logging	(string →string) Map	logging Einstellungen (nwb)
PIFs	(PIF ref) Set	Referenzen auf alle aktiven physischen Schnittstellen
host_CPUs	(host cpu ref) Set	Referenzen auf alle physischen CPUs des Hosts

metrics	host_metrics ref	Referenz auf die Metriken des Hosts

*1 true, falls der Host eingeschaltet ist, und sich in einem Zustand befindet, in dem VMs gestartet werden können.
*2 Die nachfolgenden Ausgaben des Testsystems ergeben ein gutes Bild über verfügbare Schlüssel und die grundlegende Semantik dieser Attribute:

```
'xen_commandline': 'placeholder'
'cc_compile_domain': 'debian.org'
'cc_compile_date': 'Fri Sep  3 15:38:12 UTC 2010'
'xen_minor': '0'
'xen_extra': '.1'
'cc_compiler': 'gcc version 4.4.5 20100824 (prerelease)
                (Debian 4.4.4-11) '
'Xen': '4.0'
'system': 'Linux'
'machine': 'x86_64'
'xen_changeset': 'unavailable'
'host': 'kasch-xen4-host'
'version': '#1 SMP Fri Dec 10 17:41:50 UTC 2010'
'release': '2.6.32-5-xen-amd64'
'cc_compile_by': 'waldi'
'xen_major': '4'
'xend_config_format': '4'
```

*3 Hierbei handelt es sich um eine Ausgabe möglicher Fähigkeiten dieser Xen Instanz. Im Folgenden werden einige Beispiele angegeben:

```
'xen-3.0-x86_64', 'xen-3.0-x86_32p', 'hvm-3.0-x86_32',
'hvm-3.0-x86_32p', 'hvm-3.0-x86_64'
```

*4 Die CPU beschreibende Attribute wie nr_nodes,nr_cpus,sockets_per_node,cores_per_socket oder threads_per_core.

A.1.2. Klasse host_cpu

uuid	string	Global eindeutiger Objektbezeichner
host	host ref	Host, zu dem die PCPU gehört
number	int	Anzahl der PCPUs im Host
vendor	string	Hersteller der PCPU
speed	int	Geschwindigkeit der PCPU in Mhz
modelname	string	Die Modellbezeichnung der PCPU
stepping	string	Das Stepping PCPU
flags	string	*1
features	string	*1
utilisation	float	Momentane Auslastung der PCPU im Interval [0,1]

*[1] Feature Felder der CPU. Einmal als Bitmap (`features`), einmal decodiert String (`flags`). Die Semantik des `features` Feld bleibt unklar, so handelt es sich beim ersten Block um die Rückgabe von CPUID in EDX bei Aufruf mit EAX=1; das zugehörige ECX Register ist der 5. Block. Block 2 enthält das EDX Register bei Aufruf mit EAX=80000001h; Block 7 ist das ECX Register bei diesem Aufruf. Auf dem Testsystem sehen diese beispielhaft wie folgt aus:

```
('features', '178bf3ff:ebd3fbff:00000000:00000010:
              00002001:00000000:0000011f:00000000')
('flags', 'fpu de tsc msr pae mce cx8 apic mtrr mca
           cmov pat clflush mmx fxsr sse sse2 ht syscall
           nx mmxext fxsr_opt lm 3dnowext 3dnow rep_good
           extd_apicid pni cx16 hypervisor lahf_lm
           cmp_legacy extapic cr8_legacy 3dnowprefetch')
```

A.1.3. Klasse host_metrics

`uuid`	string	Global eindeutiger Objektbezeichner
`memory/total`	int	Gesamter Speicher des Hosts in Bytes
`memory/free`	int	Freier Speicher des Hosts in Bytes
`last_updated`	datetime	Zeitpunkt der letzten Aktualisierung dieser Daten (ISO 8601)

A.1.4. Klasse PIF

`uuid`	string	Global eindeutiger Objektbezeichner
`device`	string	Name der Schnittstelle z.B. eth0
`network`	network ref	Netz, zu dem dieses PIF verbunden ist
`host`	host ref	Referenz auf den Host zu dem diese PIF gehört
`MAC`	string	Ethernet MAC Adresse der phy. Schnittstelle
`MTU`	int	MTU
`VLAN`	int	VLAN tag für den gesammten Traffic über dieses Interface
`metrics`	PIF_metrics ref	Referenz auf Metriken des PIF

A.1.5. Klasse network

`uuid`	string	Global eindeutiger Objektbezeichner
`name/label`	string	Name des Netzes
`name/description`	string	Beschreibung
`VIFs`	(VIF ref) Set	Referenzen auf VIFs in diesem Netz
`PIFs`	(PIF ref) Set	Referenzen auf PIFs in diesem Netz
`default_gateway`	string	Standardgateway dieses Netzes
`default_netmask`	string	Standard Netzmaske dieses Netzes
`other_config`	(string →string) Map	(nwb)

A.1.6. Klasse PIF_metrics & Klasse VIF_metrics

uuid	string	Global eindeutiger Objektbezeichner
io/read_kbs	float	Lesebandbreite in KiB/s
io/write_kbs	float	Schreibbandbreite in KiB/s
last_updated	datetime	Zeitpunkt der letzten Aktualisierung dieser Daten (ISO 8601)

A.1.7. Klasse VIF

uuid	string	Global eindeutiger Objektbezeichner
device	string	Name der Schnittstelle z.B. eth0
network	network ref	Netz, zu dem dieses VIF verbunden ist
VM	VM ref	Referenz zu der virtuellen Maschine zu der dieses VIF gehört
MAC	string	Ethernet MAC Adresse
MTU	int	MTU
currently_attached	bool	true, falls die Schnittstelle angeschlossen ist
status_code	int	error/success Code der letzten attach-Operation
status_detail	string	error/success Informationen der letzten attach-Operation
runtime_properties	(string →string) Map	Laufzeiteigenschaften (nwb)
qos/algorithm_type	string	(nwb)
qos/algorithm_params	(string →string) Map	(nwb)
qos/supported_algorithms	string Set	(nwb)
metrics	VIF_metrics ref	Referenz auf Metriken des VIF

A.1.8. Klasse VM

uuid	string	Global eindeutiger Objektbezeichner
power_state	vm_power_state	*[1]
name/label	string	Name
name/description	string	Beschreibung der virtuellen Maschine
user_version	int	Durch Nutzer festgelegte Versionsnummer

is_a_template	bool	true, falls diese VM nur als Template fungiert. Template VMs können nicht gestartet, sondern nur als Vorlage zur Erzeugung neuer VMs genutzt werden
auto_power_on	bool	true, falls diese VM automatisch mit dem Host gestartet werden soll
resident_on	host ref	Referenz auf den Host, auf dem sich die VM befindet
memory/static_max	int	*2
memory/dynamic_max	int	*2
memory/dynamic_min	int	*2
memory/static_min	int	*2
VCPUs/params	(string →string) Map	*3
VCPUs/max	int	Maximale Anzahl an VCPUs
VCPUs/at_startup	int	Anzahl der VCPUs mit der die VM gestartet wird
actions/after_shutdown	on_normal_exit	*4
actions/after_reboot	on_normal_exit	*4
actions/after_crash	on_crash_behaviour	*4
VIFs	(VIF ref) Set	Virtuelle Netzschnittstellen
PV/bootloader	string	Name oder Pfad des Bootloaders (nwb)
PV/kernel	string	URI des Kernels (nwb)
PV/ramdisk	string	URI der initrd (nwb)
PV/args	string	Kommandozeilenargumente des Kernels (nwb)
PV/bootloader_args	string	Argumente des Bootloaders (nwb)
HVM/boot_policy	string	*5
HVM/boot_params	(string →string) Map	*5
platform	(string →string) Map	*6
PCI_bus	string	PCI Bus Pfad für pass-through devices (nwb)
other_config	(string →string) Map	*7
domid	int	domain ID (falls verfügbar, -1 sonst)
is_control_domain	bool	true falls dies eine Control Domain ist (dom0)
metrics	VM_metrics ref	Referenz auf Metriken zu der VM
guest_metrics	VM_guest_metrics ref	*8

`security/label`	string	Wird für Security Policies in Verbindung mit sHype genutzt (nwb)

*[1] Der `power_state` kann folgende Werte annehmen: `Halted`, `Paused`, `Running`, `Suspended`, `Crashed`, oder `Unknown`. `Unknown` steht für einen anderen unbekannten Status. `Crashed` bezeichnet ein abgestürztes Gastbetriebssystem. Der Unterschied zwischen `Paused` und `Suspended` ist, dass im Zustand `Suspended` das Image der VM auf Hintergrundspeicher geschrieben wurde, bei `Paused` liegt dieses im Arbeitsspeicher. Den Zusammenhang dieser Attribute verdeutlicht Abbildung A.1.

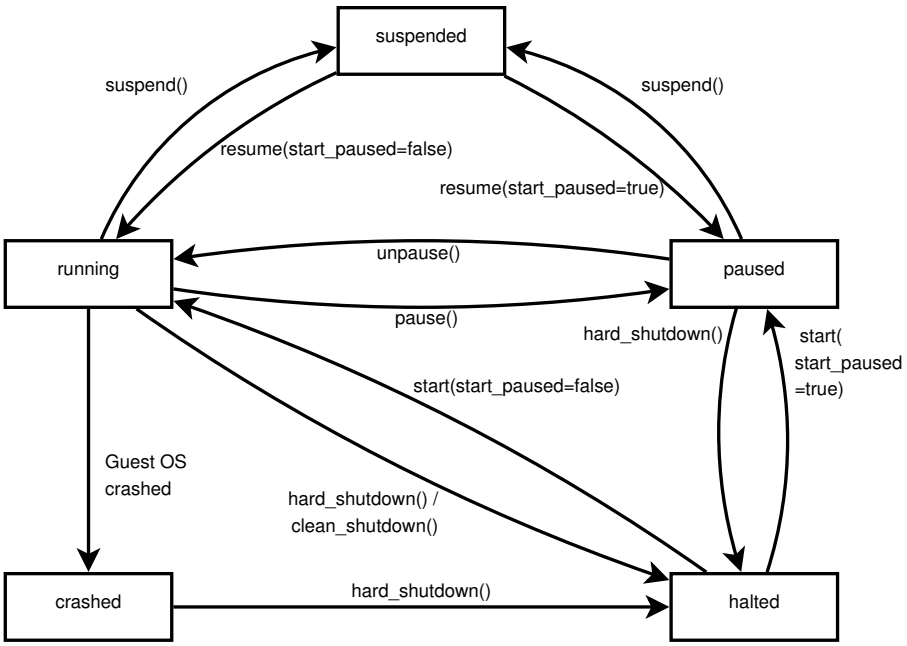


Abbildung A.1.: Zustände virtueller Maschinen bei Xen

*[2] Diese Werte legen die Grenzen fest, die einer virtuellen Maschine als Arbeitsspeicher in Bytes zur Verfügung gestellt werden. Die statischen Werte dienen hierbei als harte Grenzen und können während der Ausführung eines Gastes nicht geändert werden. Die dynamischen Grenzen können während der Ausführung des Gastes geändert werden, sie müssen innerhalb der statischen liegen. Es muss gelten:

$$\text{static-min} \leqslant \text{dynamic-min} \leqslant \text{dynamic-max} \leqslant \text{static-max}$$

Falls genug Speicher auf dem Host verfügbar ist, wird jedem Gast das dynamamische Maximum zugeteilt; sollte nicht genügend Speicher verfügbar sein, so wird der Host versuchen, bei allen Gästen den Speicher – proportional zu der Distanz zwischen dynamischem Maximum und Minimum – um denselben Anteil zu verringern.
*[3] Für dieses Attribut sind zwei Schlüssel möglich:
Zum einen `weight`: hiermit ist eine Gewichtung der Zuteilung von CPU-Zeit möglich. Eine VM mit einem `weight` von 512 erhält doppelt so viel CPU-Zeit wie eine VM mit einem `weight` von 256. Der Standardwert ist 256, der Wertebereich geht von 1 bis 65535.

Zum anderen gibt es den Schlüssel `cap`: dieser legt einen maximalen Prozentsatz fest, den eine VM, auch bei nicht ausgelasteter Host-CPU, erhält. 100 steht hier für eine CPU, 50 eine halbe CPU, 200 für 2 CPUs, der Standartwert 0 sagt aus, dass es keine obere Grenze gibt.
*4 Aktionen, die nach einem Shutdown, Reboot, oder einem Absturz ergriffen werden sollen. Mögliche Aktionen sind `restart` oder `destroy`, im Falle eines Absturzes ist es zusätzlich noch möglich, mittels `coredump_and_destroy` bzw. `coredump_and_restart` vorher einen Coredump anzulegen, die VM mittels `preserve` unangetastet zu lassen, oder mittels `rename_restart` sie unangetastet zu lassen, aber als Kopie neu zu starten.
*5 Diese zwei Parameter legen bei Xen Eigenschaften für die Hardwarevirtualisierung fest, und werden im Rahmen dieser Arbeit wie auch die `PV`-Attribute zur Paravirtualisation nicht weiter behandelt. Sollte die Boot Order auf "BIOS" gesetzt sein, so kann über die Parameter beispielsweise mittels "dc" angeben werden, dass zuerst von CD und dann von Festplatte gebootet werden soll.
*6 Diese Map kennt Schlüssel wie `acpi`, `videoram`, `nomigrate`. Sollte beispielsweise `nomigrate` ungleich 0 sein, so kann die virtuelle Maschine nicht migriert oder wiederhergestellt werden. Da es hier in Xen (Version 4.0.1) achtundvierzig mögliche Schlüssel gibt, wird hier für eine Aufzählung auf die Datei `XendConfig.py` aus dem Quellcode von Xen verwiesen, die Klärung der Semantik aller Felder bleibt offen. Dieses Attribut wird ebenfalls im Verlauf nicht weitergehend betrachtet.
*7 Kann beispielsweise den Schlüssel `install-repository` beinhalten, der auf ein Debian Repository zeigt, aus dem der Gast installiert werden soll. Auch dieses Attribut wird nicht weiter betrachtet.
*8 Die Metriken für das Gastbetriebssystem werden nicht näher analysiert, da dieses Feld auf dem Testsystem nicht verfügbar ist.

A.1.9. Klasse VM_metrics

`uuid`	string	Global eindeutiger Objektbezeichner
`memory/actual`	int	Aktuell dem Gast zugeteilter Speicher in Bytes
`VCPUs/number`	int	Momentane Anzahl virtueller CPUs
`VCPUs/utilisation`	(int →float) Map	Aktuelle Auslastung der einzelnen VCPUs der VM im Interval [0,1]
`VCPUs/CPU`	(int →int) Map	Mapping von VCPUs auf PCPUs
`VCPUs/params`	(string →string) Map	siehe Klasse VM
`VCPUs/flags`	(int →string Set) Map	CPU Flags der einzelnen VCPUs (blocked, online, running)
`state`	string Set	Zustand der gesamten VM (blocked, dying, running)

start_time	datetime	Zeitpunkt, zu dem die VM zuletzt gebootet wurde (ISO 8601)
last_updated	datetime	Zeitpunkt der letzten Aktualisierung dieser Daten (ISO 8601)

A.2. VMware

MO steht im Folgenden für Managed Objects, DO für Data Objects.

A.2.1. MO HostSystem

config	HostConfigInfo	siehe Abschnitt A.2.1.1
hardware	HostHardwareInfo	siehe Abschnitt A.2.1.6
name	xsd:string	Name (geerbt von ManagedEntity)
network	ManagedObjectReference[]	Referenz auf die Netze, die auf diesem Host verfügbar sind
runtime	HostRuntimeInfo	siehe Abschnitt A.2.1.11
summary	HostListSummary	siehe Abschnitt A.2.1.12
vm	ManagedObjectReference[]	Mit diesem Host assoziierte VMs

A.2.1.1. DO HostConfigInfo

host	ManagedObjectReference	Referenz auf das Host System
network	HostNetworkInfo	siehe Abschnitt A.2.1.2
product	AboutInfo	Diverse Versionsattribute des Hosts
virtualMachine↵ Reservation	VirtualMachineMemoryReservationInfo	siehe Abschnitt A.2.1.5

A.2.1.2. DO HostNetworkInfo

atBootIpV6Enabled	xsd:boolean	falls true dann wird IPv4/IPv6 im Dual-Stackbetrieb aktiviert
ipV6Enabled	xsd:boolean	true, falls ipv6 Support gegeben ist
pnic	PhysicalNic[]	siehe Abschnitt A.2.1.3

A.2.1.3. DO PhysicalNic

autoNegotiateSupported	xsd:boolean	Gesetzt, falls der Adapter Autonegotiation unterstützt
device	xsd:string	Name
driver	xsd:string	Treibername
linkSpeed	PhysicalNicLinkInfo	siehe Abschnitt A.2.1.4
mac	xsd:string	MAC Adresse des PNIC
pci	xsd:string	PCI-Device Hash
validLinkSpecifikation	PhysicalNicLinkInfo[]	Gültige Kombinationen für linkSpeed
wakeOnLanSupported	xsd:boolean	wakeOnLanSupport

A.2.1.4. DO PhysicalNicLinkInfo

duplex	xsd:boolean	full-duplex=true, half-duplex=false
speedMb	xsd:int	Die Übertragungsrate auf dem Link in Mbit/s

A.2.1.5. DO VirtualMachineMemoryReservationInfo

allocationPolicy	xsd:	(nwb)
virtualMachineMax	xsd:long	*1
virtualMachineMin	xsd:	*1
virtualMachineReserved	xsd:long	*1

*1 Maximale, minimale und aktuelle Größe des für virtuelle Maschinen reservierten Hauptspeichers.

A.2.1.6. DO HostHardwareInfo

biosInfo	HostBIOSInfo	(nwb)
cpuFeature	HostCpuIdInfo[]	siehe Abschnitt A.2.1.8
cpuInfo	HostCpuInfo	siehe Abschnitt A.2.1.7
cpuPkg	HostCpuPackage[]	siehe Abschnitt A.2.1.9
memorySize	xsd:long	Gesamtmenge des physischen Speichers in Bytes
systemInfo	HostSystemInfo	siehe Abschnitt A.2.1.10

A.2.1.7. DO HostCpuInfo

hz	xsd:long	Durchschnittliche Taktrate in hz
numCpuCores	xsd:short	Anzahl der physischen Cores auf einem Host
numCpuPkgs	xsd:short	Anzahl der physischen Prozessoren auf einem Host. Ein Dual-Core ist ein Prozessor

A.2.1.8. DO HostCpuIdInfo

eax	xsd:string	Stringrepräsentation des EAX Registers
ebx	xsd:string	Stringrepräsentation des EBX Registers
ecx	xsd:string	Stringrepräsentation des ECX Registers
edx	xsd:string	Stringrepräsentation des EDX Registers
level	xsd:int	EAX Eingabe für CPUID

Die Repräsentation erfolgt in einer binären Darstellung, wobei nach jeweils vier Ziffern mittels ":" separiert wird.

A.2.1.9. DO HostCpuPackage

busHz	xsd:long	Taktrate des Bus in hz.
cpuFeature	HostCpuIdInfo[]	siehe Abschnitt A.2.1.8
description	xsd:string	zur Darstellung geeignete Beschreibung
hz	xsd:long	Taktrate der CPU in hz.
index	xsd:short	Package index, beginnt bei 0
threadId	xsd:short[]	Bezeichnung der logischen CPU-Threads
vendor	xsd:string	Hersteller, momentan Intel, AMD oder unknown

A.2.1.10. DO HostSystemInfo

model	xsd:string	Modellname des Systems
otherIdentifyingInfo	HostSystemIdentificationInfo[]	Weitere Identifikationsmerkmale (nwb)
uuid	xsd:string	Global eindeutiger Bezeichner des Hosts
vendor	xsd:string	Herstellerbezeichnung

A.2.1.11. DO HostRuntimeInfo

bootTime	xsd:dateTime	Boottime des Hosts
connectionState	HostSystemConnectionState	*1
inMaintenanceMode	xsd:boolean	true, falls der Host im Maintenance Mode ist
powerState	HostSystemPowerState	*2
standbyMode	xsd:string	(nwb) nur bei vCenter gültig

*1 connected, der Host ist zum Server verbunden (bei ESX ist dieser Wert fest, die anderen existieren nur bei vCenter);
disconnected, der Host wurde bewusst durch den Nutzer deaktiviert, keine Heartbeats werden empfangen;

notResponding, vCenter empfängt keine Heartbeats des Hosts.
*2 poweredOff; poweredOn; standBy; unknown, sollte nur auftreten falls der Host disconnected ist.

A.2.1.12. DO HostListSummary

config	HostConfigSummary	(nwb)
hardware	HostHardwareSummary	siehe Abschnitt A.2.1.13
host	ManagedObjectReference	Referenz auf den Host
overallStatus	ManagedEntityStatus	*1
quickStats	HostListSummaryQuickStats	siehe Abschnitt A.2.1.14
rebootRequired	xsd:boolean	zeigt an ob der Host einen Neustart benötigt
runtime	HostRuntimeInfo	siehe Abschnitt A.2.1.11

*1 Für den Status gibt es vier Zustände:
gray, der Status ist unbekannt;
green, die Entität ist ok;
red, die Entität hat sicher ein Problem;
yellow, die Entität könnte ein Problem haben.

A.2.1.13. DO HostHardwareSummary

cpuMhz	xsd:int	Bei mehreren CPUs Durchschnitt in Mhz
cpuModel	xsd:string	Das CPU Modell
memorySize	xsd:long	Die Größe des physischen Speichers in Bytes.
model	xsd:string	Modellname des Systems
numCpuCores	xsd:short	Anzahl der physischen Cores auf einem Host
numCpuPkgs	xsd:short	Anzahl der physischen Prozessoren auf einem Host. Ein Dual-Core ist ein Prozessor
numCpuThreads	xsd:short	Anzahl der physischen CPU Threads auf dem Host
numHBAs	xsd:short	Anzahl der Host Bus Adapter (nwb)
numNics	xsd:int	Anzahl der physischen Netzschnittstellen
uuid	xsd:string	Global eindeutiger Bezeichner des Hosts
vendor	xsd:string	Hardwarehersteller

A.2.1.14. DO HostListSummaryQuickStats

distributedCpuFairness	xsd:int	*1
distributedMemoryFairness	xsd:int	*1
overallCpuUsage	xsd:int	Aggregierte CPU Auslastung über alle Cores in Mhz
overallMemoryUsage	xsd:int	Speichernutzung auf dem Host in MB
uptime	xsd:int	Uptime des Hosts in Sekunden

*1 Absolute Fairness wird durch den Wert 1000 ausgedrückt, sie wird relativ zu den anderen Hosts berechnet. Je mehr der Wert von 100 divergiert, desto unfairer ist die Verteilung

A.2.2. MO Network

host	ManagedObjectReference[]	Hosts in diesem Netz
name	xsd:string	Name des Netzes
summary	NetworkSummary	siehe Abschnitt A.2.2.1
vm	ManagedObjectReference[]	VMs in diesem Netz

A.2.2.1. DO NetworkSummary

accessible	xsd:boolean	Mindestens ein Host in diesem Netz ist erreichbar
ipPoolName	xsd:string	Name des assoziierten IP Pools (nwb)
name	xsd:string	Name
network	ManagedObjectReference	Referenz auf das Netz

A.2.3. MO VirtualMachine

config	VirtualMachineConfigInfo	siehe Abschnitt A.2.3.1
guestHeartbeatStatus	ManagedEntityStatus	*1
name	xsd:string	Name (geerbt von ManagedEntity)
network	ManagedObjectReference[]	siehe Abschnitt A.2.2
runtime	VirtualMachineRuntimeInfo	siehe Abschnitt A.2.3.5
summary	VirtualMachineSummary	siehe Abschnitt A.2.3.6

*1 gray, VMware Tools nicht installiert oder nicht laufend;
red, kein Heartbeat, Gastsystem könnte angehalten sein;
yellow, Unterbrochener Heartbeat, Gast könnte ausgelastet sein;
green, Gast antwortet normal.

A.2.3.1. DO VirtualMachineConfigInfo

alternateGuestName	xsd:string	Name des Gastes wenn die guestId other oder other-64 ist
annotation	xsd:string	Beschreibung der VM
changeVersion	xsd:string	Eindeutige Identifikation der Version der Configuration, als transparenten String betrachten
cpuAllocation	ResourceAllocationInfo	siehe Abschnitt A.2.3.2
cpuHotAddEnabled	xsd:boolean	falls CPUs im Betrieb hinzugefügt werden können
cpuHotRemoveEnabled	xsd:boolean	falls CPUs im Betrieb entfernt werden können
guestFullName	xsd:string	Vollständiger Name des Gastbetriebssystems
guestId	xsd:string	Kurzname des Gastbetriebssystems
hardware	VirtualHardware	siehe Abschnitt A.2.3.3
hotPlugMemoryIncrement Size	xsd:long	Erhöhungen der Speichergröße müssen Vielfache dieses Wertes in MB sein
hotPlugMemoryLimit	xsd:long	Der maximale Speicher in MB, der im laufenden Betrieb zu einer VM hinzugefügt werden kann
instanceUuid	xsd:string	vCenter spezifische UUID zur Unterscheidung von VMs mit selber SMBIOS UUID
memoryAllocation	ResourceAllocationInfo	siehe Abschnitt A.2.3.2
modified	xsd:dateTime	Zeitpunkt der letzten Modifikation
name	xsd:string	Name der VM
template	xsd:boolean	Beschreibt ob eine VM ein Template ist
uuid	xsd:string	128-bit SMBIOS UUID einer VM

A.2.3.2. DO ResourceAllocationInfo

`expandableReservation`	xsd:boolean	true, falls die `reservation` bei freien Hostressourcen ansteigen kann
`limit`	xsd:long	Maximale `reservation` bei Speicher in MB, bei CPU in Mhz
`reservation`	xsd:long	Aktuelle `reservation` bei Speicher in MB, bei CPU in Mhz

A.2.3.3. DO VirtualHardware

`device`	VirtualDevice[]	unsortierte Liste aller virtuellen Devices darunter auch VirtualEthernetCard siehe Abschnitt A.2.3.4
`memoryMB`	xsd:int	Größe des Speichers in MB
`numCPU`	xsd:int	Anzahl der in dieser VM verfügbaren VCPUs

A.2.3.4. DO VirtualEthernetCard

`addressType`	xsd:string	*1
`macAddress`	xsd:string	Ethernet MAC Adresse
`wakeOnLanEnabled`	xsd:boolean	true, falls wake on lan aktiv

*1 `Manual` Statisch manuell zugeteilte MAC Adresse;
`Generated` Automatisch generierte statische MAC Adresse;
`Assigned` durch vCenter zugeteilte MAC Adresse

A.2.3.5. DO VirtualMachineRuntimeInfo

`bootTime`	xsd:dateTime	Zeitpunkt zu dem die VM zuletzt gebootet wurde.
`cleanPowerOff`	xsd:boolean	Bei ausgeschalteter VM ob der letzte Shutdown ordnungsgemäß verlief
`host`	ManagedObjectReference	Hostsystem, auf dem die VM sich befindet
`maxCpuUsage`	xsd:int	Obere Grenze für die CPU Auslastung in Mhz
`maxMemoryUsage`	xsd:int	Obere Grenze für die Speicherauslastung
`powerState`	VirtualMachinePowerState	*1

suspendInterval	xsd:long	Zeit, die die VM suspendiert war, die aktuelle Periode ausgenommen, in Sekunden
suspendTime	xsd:dateTime	Zeitpunkt der letzten Suspendierung der VM

A.2.3.6. DO VirtualMachineSummary

config	VirtualMachineConfigSummary	siehe Abschnitt A.2.3.7
guest	VirtualMachineGuestSummary	(nwb)
overallStatus	ManagedEntityStatus	vgl. bei HostListSummary (Abschnitt A.2.1.12)
quickStats	VirtualMachineQuickStats	siehe Abschnitt A.2.3.8
runtime	VirtualMachineRuntimeInfo	siehe Abschnitt A.2.3.5
vm	ManagedObjectReference	Referenz auf die VM

A.2.3.7. DO VirtualMachineConfigSummary

annotation	xsd:string	Beschreibung der VM
cpuReservation	xsd:int	CPU Reservation in Mhz
guestFullName	xsd:string	Vollständiger Name des Gastbetriebssystems
guestId	xsd:string	Kurzname des Gastbetriebssystems
instanceUuid	xsd:string	vCenter spezifische UUID zur Unterscheidung von VMs mit selber SMBIOS UUID
memoryReservation	xsd:int	Konfigurierte Memory Reservierung in MB
memorySizeMB	xsd:int	Hauptspeichergröße der VM
name	xsd:string	Name
numCpu	xsd:int	Anzahl der Prozessoren in der VM
numEthernetCards	xsd:int	Anzahl der virtuellen Netzschnittstellen
numVirtualDisks	xsd:int	Anzahl der virtuellen Festplatten
product	VAppProductInfo	Diverse Produktinformationen (nwb)
template	xsd:boolean	wahr, falls diese VM Template ist
uuid	xsd:string	128-bit SMBIOS UUID einer VM

A.2.3.8. DO VirtualMachineQuickStats

`balloonedMemory`	xsd:int	*1
`compressedMemory`	xsd:long	aktueller Verbrauch an komprimiertem Speicher in kB
`guestMemoryUsage`	xsd:int	*2
`hostMemoryUsage`	xsd:int	*2
`overallCpuUsage`	xsd:int	Aktuelle CPU Auslastung in Mhz
`privateMemory`	xsd:int	*3
`sharedMemory`	xsd:int	*3
`staticCpuEntitlement`	xsd:int	*4
`staticMemoryEntitlement`	xsd:int	*4
`swappedMemory`	xsd:int	Größe des Speichers in MB, der vom Host geswapped wird
`uptimeSeconds`	xsd:int	Uptime der VM in Sekunden

*1 Größe des ballooned Memory in MB. Sollte ein Host wenig Speicher zur Verfügung haben, so versucht er mittels des ballooning Treibers vom Betriebssystem des Gastes Speicher zu alloziieren und somit reservierten Speicher zurückzuerhalten.
*2 Aktuelle Nutzung des Gast- bzw. Hostspeichers durch die VM in MB, dieser Wert kann zwischen 0 und dem konfigurierten Maximum liegen.
*3 Größe der VM garantiertem nicht gemeinsam genutztem Hostspeichers, beziehungsweise des gemeinsam genutzen Speichers in MB
*4 Diese statische Speicherzuteilung in MB bzw. CPU-Zuteilung in Mhz spiegelt ein Worst-Case Szenario wieder. Sie errechnet sich aus den limits und reservations und lässt die aktuelle Nutzung außen vor.

Abbildungsverzeichnis

Literaturverzeichnis

[BSB+10] Bolte, Matthias, Michael Sievers, Georg Birkenheuer, Oliver Niehorster und André Brinkmann: *Non-intrusive virtualization management using libvirt*. In: *DATE*, Seiten 574–579, 2010.

[Dan09] Danciu, V.: *Host virtualization: a taxonomy of management challenges*. In: *GI–Edition Lecture Notes in Informatics (LNI)*, Band 2009 der Reihe *P–154*, Bonn, Germany, Oktober 2009. Gesellschaft für Informatik (GI).

[DMT07] DMTF: *CIM System Virtualization Model White Paper*, November 2007.

[DMT09a] DMTF: *Memory Ressource Virtualization Profile*, July 2009.

[DMT09b] DMTF: *Open Virtualization Format White Paper*, June 2009.

[DMT10a] DMTF: *System Virtualization Profile*, April 2010.

[DMT10b] DMTF: *Virtual System Profile*, April 2010.

[DMT11] DMTF: *CIM Schema: Version 2.28.0*, February 2011.

[ITS96] Institute for Telecommunication Sciences - National Telecommunication & Information Administration, National Comunications Systems: *Federal Standard 1037C, Telecommunications: Glossary of Telecommunication Terms*, 1996.

[MSS+10] Mellor, Ewan, Richard Sharp, David Scott, et al.: *Xen Management API - API Revision 1.0.10*, January 2010.

[VMw10] VMware, Inc, Palo Alto: *vSphere Web Services SDK Programming Guide, vSphere Web Services SDK 4.1*, July 2010.