Self-Adaptive Multi-Cloud Applications

Brice Morin
SINTEF

Cloud and Big Data ecosystems are dynamic and complex

- Many cloud providers, with offers changing over the time
  - Location of data
  - Pricing models
- Many services "doing the same things"
  - With different QoS
  - At different prices
- Many types of data sources
  - Social networks (typically "fuzzy")
  - Sensors (typically "binary")
- Many dependencies
  - Some services might require some other services
  - Some services might require to run on a given cloud provider
- VMs or services might fail

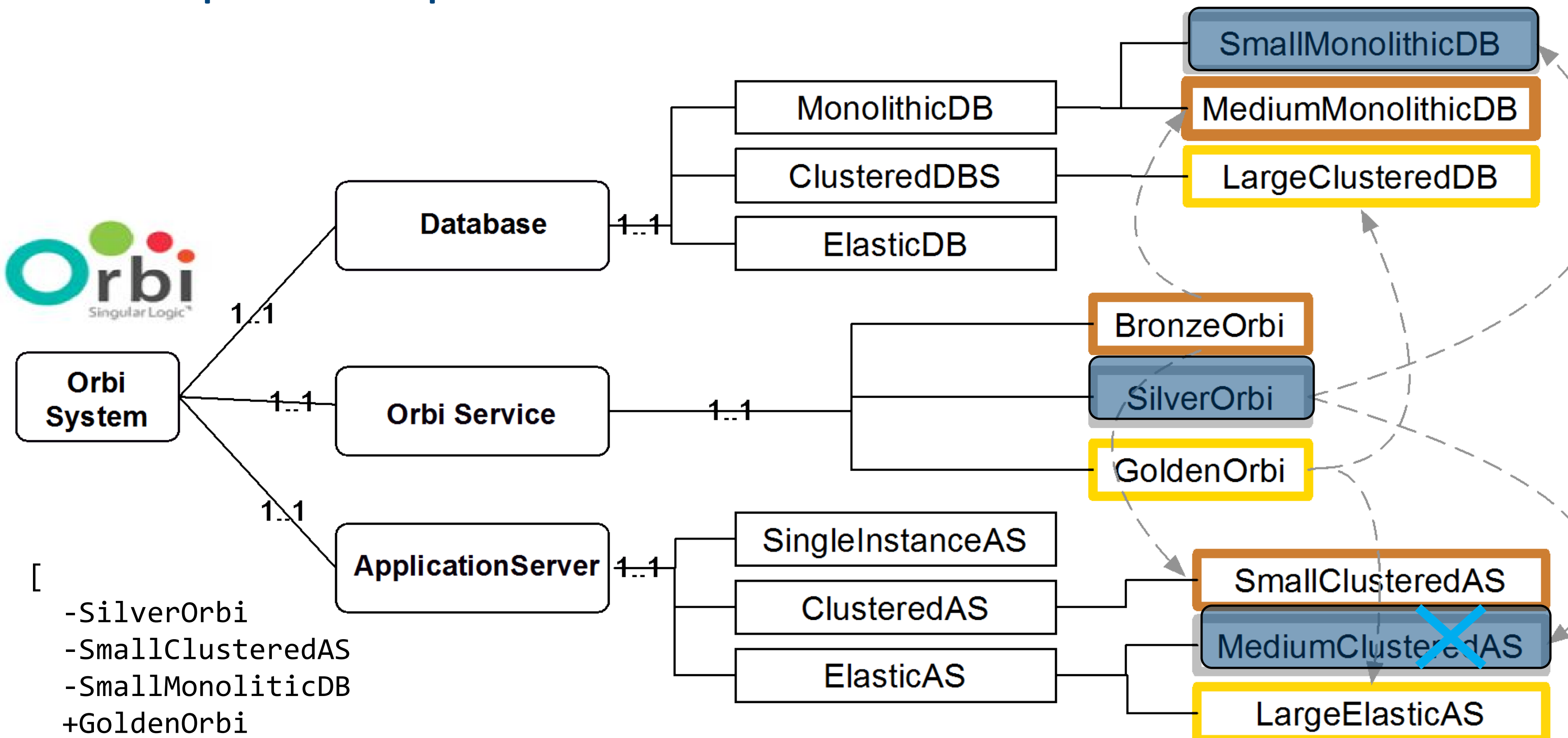
Challenges

- Not tractable to try to accommodate this complexity at design-time
  - Combinatorial explosion of the number of possible configurations (assemblies of services) for real multi-cloud applications

- Not satisfactory to just crash at runtime when facing a situation not validated at design-time

- Not sustainable to pre-book a large number of VMs and pre-deploy a large number of alternative services

A simple example




```
[
  -SilverOrbi
  -SmallClusteredAS
  -SmallMonoliticDB
  +GoldenOrbi
  +LargeElasticAS
  +LargeClusteredDB
]
```

One component failing ➔ 6 operations to fix the system

- What if the system operator just changed the failing component
  - From MediumClusteredAP to LargeElasticAP
  - ➔ yields an invalid configuration, the system is likely not to work properly

- We need tools to
  - Reason about failures (occurred or **impending**)
  - Automatically enact decisions

Abstractions to support reasoning (1/2)

	Name	ID	Values
Enum	RAM load	RAM	{LOW, MED, HI}
Literal	Low	LOW	-
Literal	Medium	MED	-
Literal	High	HI	-
Enum	CPU load	CPU	{LOW, MED, HI}
Literal	Low	LOW	-
Literal	Medium	MED	-
Literal	High	HI	-
Enum	Google Map status	GMAP	{OK, NOK, FAIL}
Literal	Normal	OK	-
Literal	Failure predicted	NOK	-
Literal	Failed	FAIL	-
Enum	Bing Map status	BMAP	{OK, NOK, FAIL}
Literal	Normal	OK	-
Literal	Failure predicted	NOK	-
Literal	Failed	FAIL	-

High-level context information:
- not everything you can possibly monitor
- but rather what makes to support reasoning

Can be obtained by monitoring infrastructure and services
+ Complex Event Processing (CEP)

Context information can be used
to describe basic/reflex rules

	Name	ID	Lower	Upper	dependency	available	required
Dimension	Map Service	MAP_S	1	1	-	-	-
Variant	Google Map	GMAP_S	-	-			BMAP = FAIL
Variant	Bing Map	BMAP_S	-	-	L or M		GMAP = FAIL
Dimension	Virtual Machine	VM	1	1	-	-	-
Variant	Large	L	-	-			CPU = LOW or RAM = LOW
Variant	Medium	M	-	-		(CPU = MED or CPU = LOW) and (RAM = MED or RAM = LOW)	
Variant	Small	S	-	-		CPU = LOW and RAM = LOW	

Abstractions to support reasoning (2/2)

	Name	Direction	Value
Property	Cost	0	-
PropertyLiteral	HIGH	-	8
PropertyLiteral	MEDIUM	-	4
PropertyLiteral	LOW	-	2
Property	Usability	1	-
PropertyLiteral	HIGH	-	8
PropertyLiteral	MEDIUM	-	4
PropertyLiteral	LOW	-	2

QoS properties

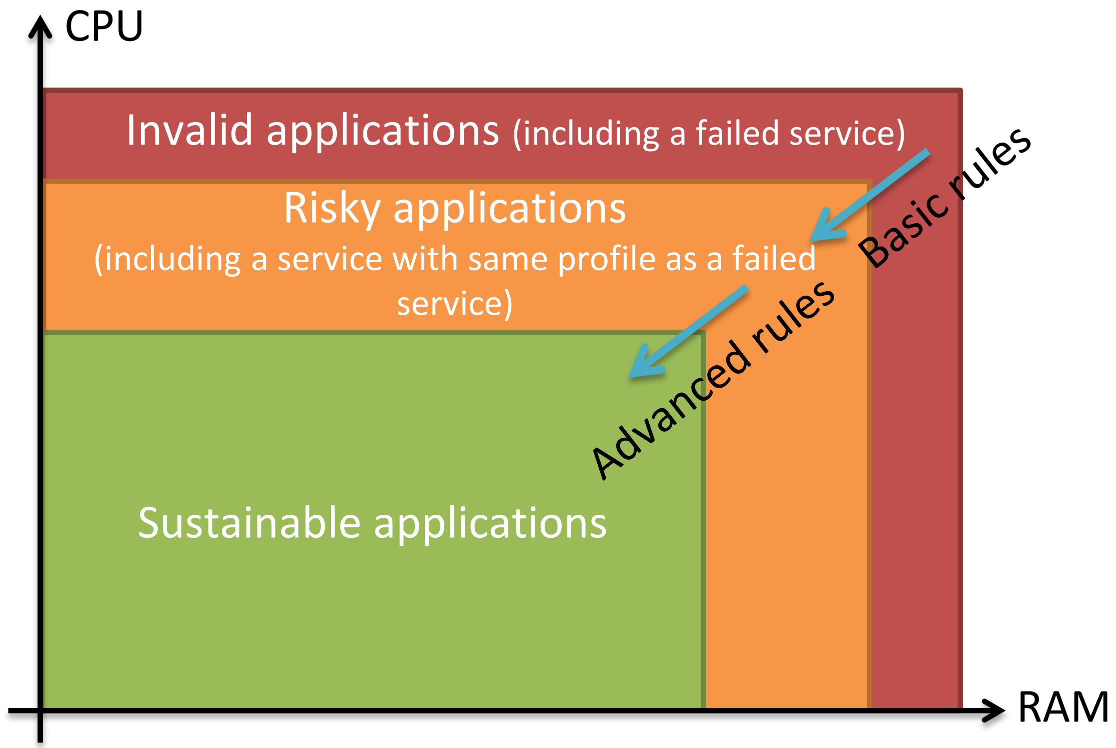
Impact of services on QoS

	Cost	Usability
Map Service (MAP_S)	false	true
Google Map (GMAP_S)	-	MEDIUM
Bing Map (BMAP_S)	-	LOW
Virtual Machine (VM)	true	true
Large (L)	HIGH	HIGH
Medium (M)	MEDIUM	MEDIUM
Small (S)	LOW	MEDIUM

When to optimize what

	Name	ID	context	Cost	Usability
Rule	Low resources	LOW	CPU = HI or RAM = HI	Very Low	Medium
Rule	High resources	HIGH	CPU = LOW and RAM = LOW	High	Very High
Rule	Med resources	MED	CPU = MED or RAM = MED	Low	High
Rule	Unstable	UNSTABLE	GMAP = NOK or BMAP = NOK or GMAP = FAIL or BMAP = FAIL	Very Low	High

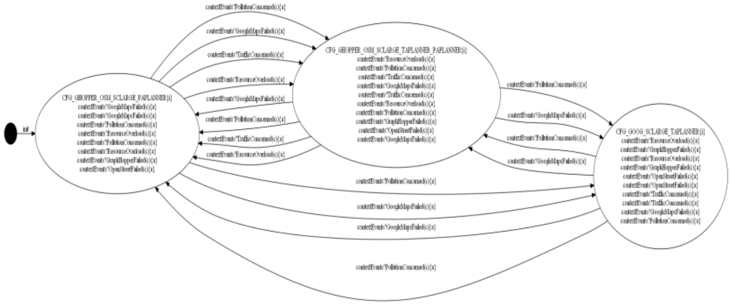
Optimization problem



In a real system, the problem is not limited to a 2-axis optimization problem

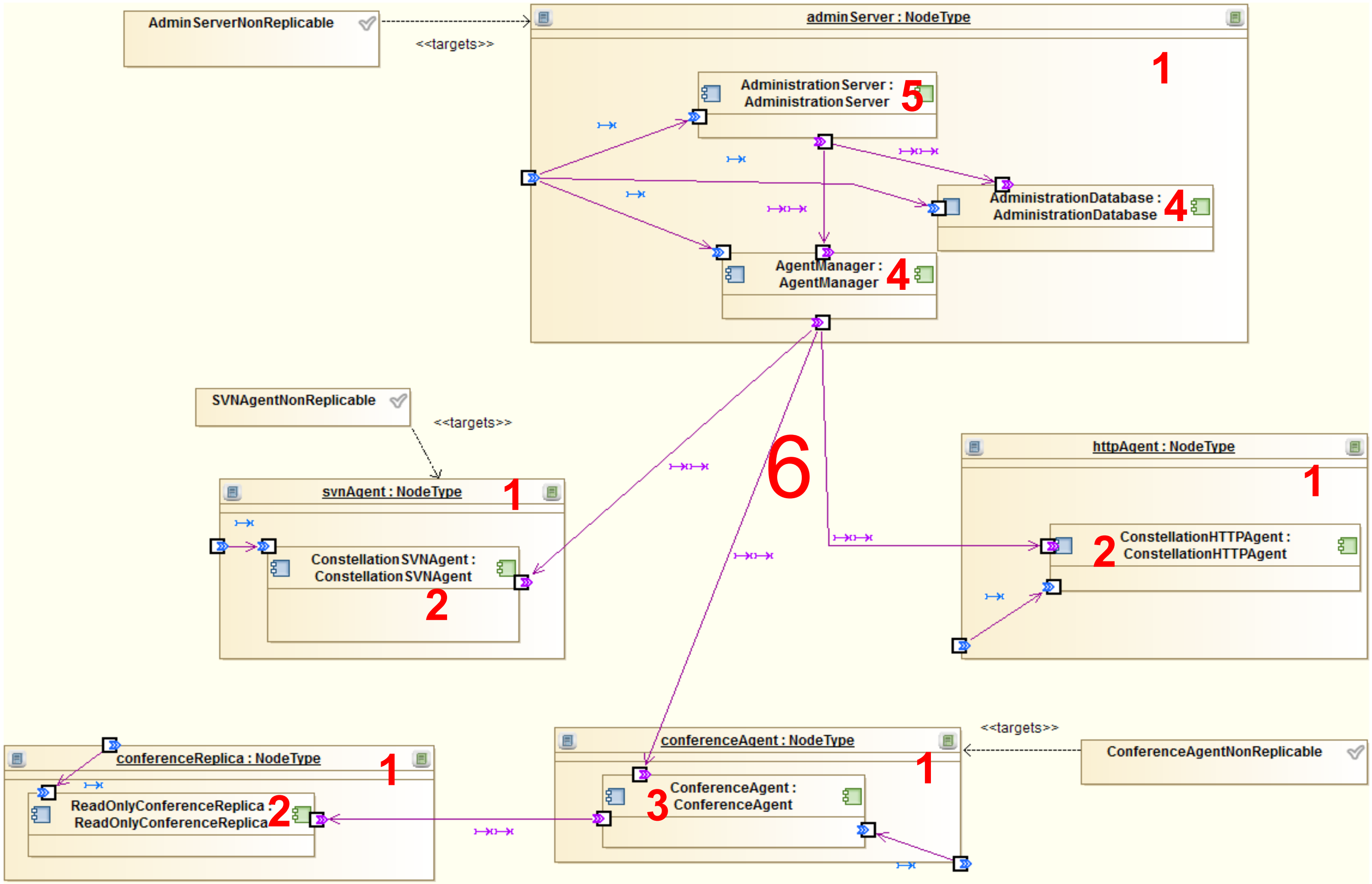
Notes

- Can be interpreted
  - Gives more flexiblity e.g. new rules can be added at runtime, transparently for the reasoner
  - Can be slow in some use cases
- Can be compiled
  - First into a state machine model, then into Java, JS or C code (based on ThingML.org)
  - Faster
  - Less flexible as we need to regenerate/re-compile code (need to embed code generators and compilers at runtime)
  - Less scalable as we need to (automatically) enumerate all possible configurations

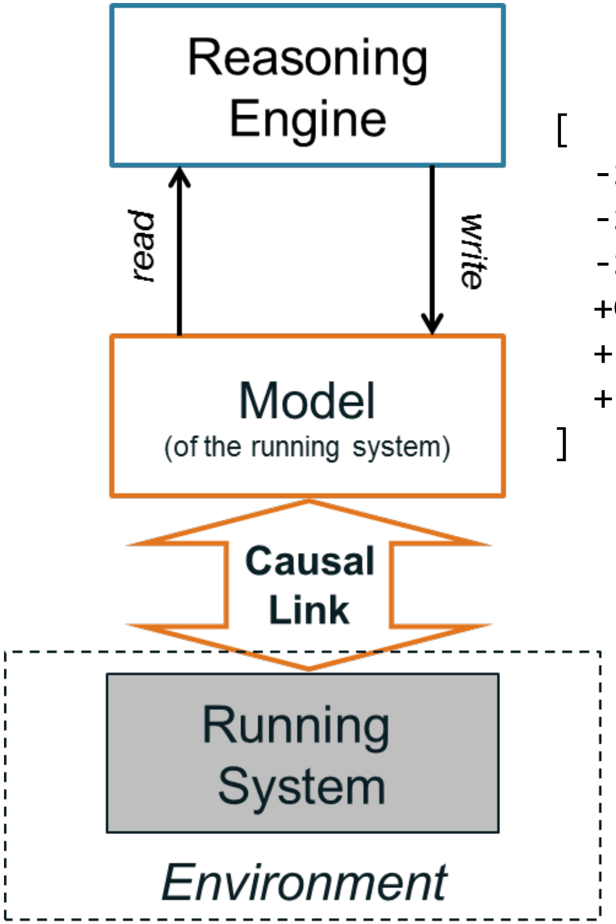


Abstractions to support enactment: CloudML

- Two main components:
  1. A **modelling environment** with a tool-supported domain-specific modelling language (DSML) to model the provisioning and deployment of multi-cloud systems
  2. A **models@run-time environment** for enacting the provisioning, deployment and adaptation of these systems


AdminServerNonReplicable
<<targets>>
adminServer : NodeType
1
AdministrationServer :
AdministrationServer
5
AdministrationDatabase :
AdministrationDatabase
4
AgentManager :
AgentManager
4
SVNAgentNonReplicable
<<targets>>
svnAgent : NodeType
1
ConstellationSVNAgent :
ConstellationSVNAgent
2
6
httpAgent : NodeType
1
ConstellationHTTPAgent :
ConstellationHTTPAgent
2
<<targets>>
conferenceReplica : NodeType
1
ReadOnlyConferenceReplica :
ReadOnlyConferenceReplica
2
conferenceAgent : NodeType
1
ConferenceAgent :
ConferenceAgent
3
ConferenceAgentNonReplicable

How both languages fit together?




```
[
   -SilverOrbi
   -SmallClusteredAS
   -SmallMonoliticDB
   +GoldenOrbi
   +LargeElasticAS
   +LargeClusteredDB
]
```

Atomic actions for the first (high level) language, e.g –SmallClusteredAS, can translate into complex scripts for the second language, e.g. including the provisioning of new nodes, the deployment of components, etc.

➔ Combining several of these complex script is not trivial

Conclusion

- Cloud applications are not just big, monolithic binaries that execute in the cloud
- Cloud applications should be able to adapt to their context, both
  - Short term, e.g. how to react if memory get saturated
  - Long term, e.g. how to accommodate changes in Amazon's terms and conditions, or pricing
- Abstraction and tool support is the key
  - Should be human readable, to include operators in the loop when needed
  - Should be machine processable

Credits

MODAClouds

PAASAGE

REMICS

Broker@Cloud