UPGRADE OF LOWER LAYERS
IN A
HIGH AVAILABILITY ENVIRONMENT

EKANSH SINGH KATIHAR

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE (SOFTWARE ENGINEERING) AT

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

APRIL 2014

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Ekansh Singh Katihar

Entitled: "Upgrade of Lower Layers in a High Availability environment"

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. C. Constantinides

_____ Examiner, External
Dr. D. Goswami To the Program

_____ Examiner
Dr. J. Rilling

_____ Supervisor
Dr. F. Khendek

_____ Supervisor
Dr. M. Toeroe

Approved by: _____

Dr. W. E. Lynch, Chair

Department of Electrical and Computer Engineering

_____20_____ _____

Dr. C. W. Trueman

Interim Dean, Faculty of Engineering and Computer Science

Abstract

Various industries such as telecommunication, banking etc. require un-interrupted services throughout the year. The requirement takes into account the system maintenance and the upgrade operations. The Service Availability Forum (SA Forum) solution enables high availability of services even during the maintenance and the upgrade operation. This solution enables portability of application across various platforms.

The SA Forum defined a service - Software Management Framework (SMF) that orchestrates the upgrade of SA Forum managed system. To perform an upgrade SMF requires an upgrade campaign. The solutions proposed in SMF are applicable only for the application layer but not for the lower layers such as Operating Systems and the virtualization facilities which include Virtual Machines and Virtual Machine Managers. On the other hand, the work done previously within the MAGIC project for the automatic generation of an upgrade campaign is limited to application entities only.

The objective of this thesis is to propose solutions in the context of SMF for the upgrade of lower layers as well without impacting the availability of services. To accomplish this objective we proposed three new upgrade steps that properly handle the dependencies between the layers of a machine during the upgrade. We also devised an approach for the automatic generation of an upgrade campaign for lower layers. The extended SMF is capable of executing the generated upgrade campaign for upgrading the virtualization facilities which include VMs capable of live migration as well. The upgrade campaign generation approach has been implemented in a prototype tool as an eclipse plug-in and tested with a case study.

Acknowledgment

I would like to express my gratitude to

- My mother, my father and my sister who were always there for me. Without them and their immense support and encouragement I could not have made so far.

- My supervisor Dr. Ferhat Khendek for believing in me and giving me the opportunity to pursue my thesis under his supervision. This thesis would not have been possible without his wisdom, encouragement and the support he provided in every possible way.

- My co-supervisor Dr. Maria Toeroe (Ericsson Canada Inc.) whose profound knowledge in the domain of High Availability, her acumen and the positive nature are the biggest reasons which helped me in overcoming the toughest challenges in the thesis.

- My friends and all those who supported me to achieve this work.

- Concordia University and Ericsson Canada for offering their facilities and resources.

This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Software Research and Concordia University as part of the Industrial Research Chair in Model Based Software Management.

Table of Contents

List of Figures

List of Acronyms

MTBF	Mean Time Between Failure
MTTR	Mean Time To Recovery
HA	High Availability
OS	Operating System
SA Forum	Service Availability Forum
AIS	Application Interface Specification
AMF	Availability Management Framework
SMF	Software Management Framework
PLM	Platform Management Service
VM	Virtual Machine
VMM	Virtual Machine Manager
HPI	Hardware Interface Specification
SU	Service Unit
SI	Service Instance
CSI	Component Service Instance
SG	Service Group

EE	Execution Environment
HE	Hardware Element
CLM	Cluster Management
IMM	Information Model Management
Proc	Procedure

Chapter 1 - Introduction

In this chapter we explain briefly the context of our research project. We introduce high availability, service continuity and the SA Forum [1] [2] (see Section 1.1). We present the thesis motivation and its contributions. Finally, we present the organization of this thesis.

1.1 Service High Availability and SA Forum

All of the software and hardware providing services can fail due to a fault in them or in the software or the hardware on which they depend. Hence in order to maintain high availability of service, these systems are implemented by deploying redundant components (software and hardware) within the system. Redundancy helps in eliminating single point of failure.

The availability of a system is measured in terms of reliability of the system components and the required time to repair the system in case of failure [1]:

- MTBF: Mean Time Between Failure: the failure rate of the system
- MTTR: Mean Time To Repair: the time to restore service

$$\text{Availability} = \text{MTBF} / (\text{MTBF} + \text{MTTR})$$

If the availability of a system is more than 99.999% of the time (also known as five nines) then that system is considered as highly available. This implies that system can be down for at most 5.26 minutes for whole year including the downtime due to system upgrade.

In case of failure in a state full service, like a video streaming service, the service must be continued from the point of failure to ensure that user does not realize this failure. To maintain

service continuity, the redundant components of a system 1) must preserve the state of the application session for each user and 2) synchronize the state between themselves.

Just like any other normal software system undergo upgrades, systems providing HA services also undergo upgrades. These upgrades are necessary to meet the changing user requirements or to cope with the new trends in the technology. These upgrades can target the application software engaged in providing business services, middleware, OS or the hardware. However, in an HA environment, it is expected that the upgrades should be live i.e. services should be made available to the end users during the upgrade as well.

At the time of the upgrade multiple versions of software may coexist as some software have already been upgraded to the target version and some of the software still have to be upgraded. Software entities involved in the system can have various kinds of dependency between themselves. For example 1) dependency between the redundant components for the state synchronization 2) dependency between the layers of a machine; e.g. An OS which is up, running and compatible is required for the application software to provide services. Maintaining the dependencies and compatibilities between the software during the live upgrade presents a big challenge as the scope of affected entities gets bigger too. Also during the live upgrade if the upgrade of any entity is unsuccessful then it may be required to undo the effect of the upgrade on that entity and to retry or to roll back the upgraded entities to their previous version.

Most HA solutions are proprietary and this hinders the application portability. In order to enable application portability several telecommunication and computing companies joined forces to create Service Availability Forum (SA Forum) [1][2] that aims to define standard middleware service specifications. The service availability solutions offered by SA Forum enable high

availability along with the service continuity. The SA Forum specifications are developed in two main groups: The Application Interface Specification (AIS) [3] and the Hardware Platform Interface [4]. These groups are further divided into smaller groups that are used to manage the software and the hardware entities. The availability management of the application services are carried out by Availability Management Framework (AMF) [5] which is a service defined in AIS. AIS also defines the Software Management Framework (SMF) [6] that orchestrates the upgrade of AMF entities while maintaining the high availability of the application services. SMF requires a roadmap through which it can upgrade a system. This roadmap is called the Upgrade Campaign Specification and is provided to SMF as an XML file.

1.2 Thesis motivations and contributions

In HA system availability of services must be maintained throughout the year even during the maintenance and upgrade operation. These operations cover not only the application layer (application software) but also the lower layer comprising middleware, operating systems, virtual machines and virtual machine managers. Orchestrating the upgrade of these many layers brings the complexity of handling the dependencies and maintaining the compatibility between the layers during the upgrade.

The current specification of SMF addresses the upgrade of applications managed by AMF but does not addresses the upgrade of lower layers managed by the Platform Management (PLM) Service (another service of AIS) [7]. Also, existing SMF specification cannot upgrade multiple layers of a machine while handling the dependency between the layers. The work done previously on the automatic generation of upgrade campaign [8] generates an upgrade campaign for the AMF entities only.

In this thesis we investigate the complexities and propose the solution for the upgrade of lower layers of a software stack comprising OSs, virtual machines and virtual machine managers while maintaining the availability of services. Through the proposed solution we extend the specification of SMF so that the entities managed by the PLM (entities from lower layer) can be upgraded as well without impacting the HA of services. We propose new steps for SMF and define the principle through which these steps should be executed. We also developed an approach for the generation of upgrade campaign specification which when fed to the extended SMF will upgrade the entire system while maintaining the high availability of services.

1.3 Thesis Organization

The rest of this thesis is organized in seven chapters. In Chapter 2, we provide necessary background knowledge on SA Forum AIS services. In Chapter 3, we discuss the limitations of the existing upgrade steps and their limited applicability on the lower layers. In Chapter, 4 we propose new upgrade steps and the necessary changes to SMF for the execution of the new steps. Chapter 5 discusses the complexities in the upgrade of virtualization facilities and then proposes an approach for its upgrade with the service availability as a primary goal. This approach is then used in the Chapter 6 where we describe how to automatically generate an upgrade campaign for the entire system. The corresponding prototype tool is discussed in Chapter 7 with a case study. We wrap up this thesis in Chapter 8 and discuss the possible future extension for this research.

Chapter 2 - Background and related work

In this chapter, we introduce the types of lower layer in a software stack that we take into account and describe in details the layers of Virtualization (see Section 2.1). We then introduce the SA Forum (see Section2.2) specifications and the important services defined in the SA Forum and used in our thesis: Availability Management Framework (see Section2.3), Cluster Membership Service (see Section2.4), Platform Management Service (see Section2.5), Information Model Management Service (see Section2.6). In Section2.7 we introduce the runtime behaviour of the Availability Management Framework and Platform Management Framework. In Section 2.8we introduce the Software Management Framework. Finally, we review the related work (see Section2.9).

2.1 Lower layers

Our research is focused on the upgrade of lower layers of machines in a cluster with a minimal impact on the availability of services provided by the application layer. These layers are Operating System, Virtual Machine Manager and Virtual Machine. The last two layers are used to achieve virtualized environment.

2.1.1 Virtualization

Virtualization technology does the emulation of hardware elements by software [7]. Using this technology, software can emulate multiple virtual environments from a single hardware element. Although there can be different types of virtualizations our focus in this thesis is on the server virtualization. Using server virtualization, multiple operating system instances can run simultaneously on a single physical server as virtual machines, each with access to the

underlying server's computing resources [9]. The important participating entities in server virtualization are the virtual machine (VM) and the virtual machine manager (VMM).

Virtual Machine

A virtual machine (VM) [10] is a software emulation of a machine (i.e. a computer) that executes programs like a physical machine. The operating system installed on the virtual machine is called the guest operating system.

Hypervisor / Virtual machine manager (VMM)

A hypervisor is used to manage multiple instances of VMs [10]. A hypervisor could be installed directly on the hardware or on a host operating system.

2.1.2 Live migration of Virtual Machine (VM)

One of the key aspects of virtualization is the possibility of live migration. Live migration helps in achieving High Availability for virtualized workloads during the maintenance operation [11]. This process involves the transfer of a running instance of a VM from one VMM to another while continuously powered-up [10]. To migrate a running instance of a VM onto a different sponsoring VMM, VM's complete state has to be transferred to the target VMM [12]. The state of a VM can be a combination of many states which includes the state of the permanent storage (disk), the volatile storage (memory), the connected devices (such as network interface cards) etc. In this thesis we consider that the permanent storage is provided through a shared storage.

2.2 SA Forum specifications

SA Forum [1] [2] is a consortium of several telecommunication and computing companies that are involved in the development of open specifications for the standardization of high

availability platforms. OpenSAF middleware [13] is an implementation compliant to the SA Forum specifications for the Linux based OS. SA Forum is developed in two main groups:

- Application Interface Specification (AIS): Provides APIs that help in developing high availability system and enabling the open integration of commercial off-the-shelf (COTS) components into a highly available system [3] [14].
- Hardware Interface Specification (HPI): Provides APIs for accessing and managing hardware resources of the machine [4].

As illustrated in the Figure 2.1, AIS defines several services. The important services used in this thesis are described in the following sections.

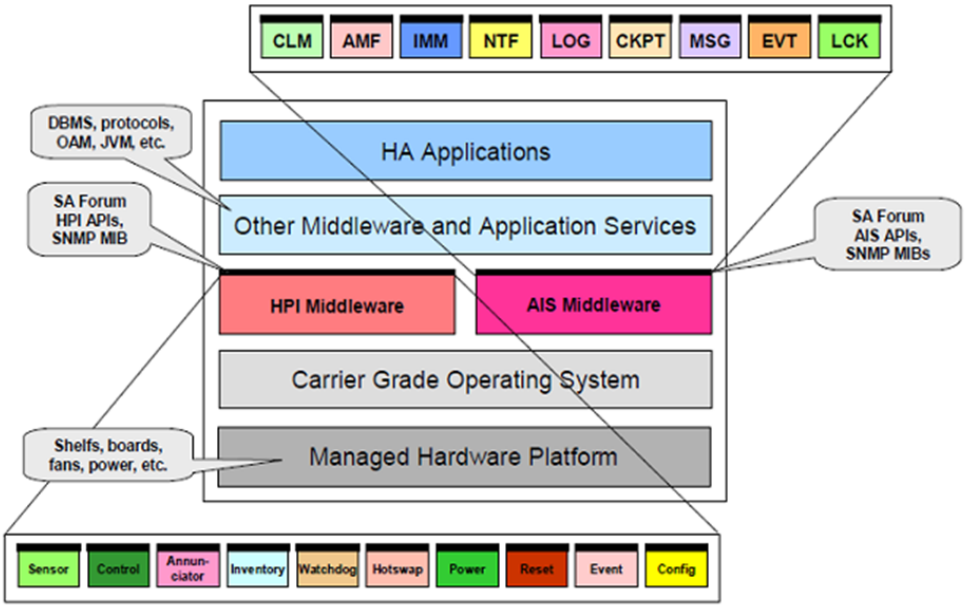


Figure 2.1 SA Forum middleware architecture (taken from [14])

2.3 Availability Management Framework (AMF)

AMF [5] is an AIS service that manages the availability of services provided by an application. It does so by managing the redundant entities of an application and dynamically shifting the workload of a faulty component to a healthy one. To fulfil this responsibility AMF requires a configuration of the application that consists of various logical entities and their relationships. Below we describe only the logical entities used in the context of this thesis.

Component

A component is the smallest entity that AMF manages. Software or hardware capable of providing services can be abstracted as a component.

Component type

A component type represents a particular version of software or hardware.

Service Unit (SU)

A service unit (SU) aggregates a set of components combining their individual functionalities to provide a higher level of service. Each component belongs to only one SU.

Service Unit Type

It defines a list of component types and for each component type, the number of components a SU of this type can accommodate.

Component Service Instance (CSI)

Component Service Instance (CSI) represents a workload or a job that AMF assigns to a component at runtime.

Component Service Type

The component service type is the generalization of the similar component service instances.

Service Instance (SI)

It is a workload that AMF assigns to an SU at the runtime. This workload is composed of the set of CSIs assigned to component of the same SU. Each CSI belongs to only one SI.

Service Type

Service type defines the list of component service type of which a SI can be composed of.

Service Group (SG)

A set of redundant SUs are organized into a service group (SG) to protect a particular set of SIs. Any SU of an SG must be able to take the assignment for any SI of this set. The redundant SUs collaborate with each other depending on the type of redundancy model of the SG.

Redundancy models

A SG is configured to have one of the five redundancy models: 2N, N+M, N-way, N-way active, No-redundancy. Depending on the redundancy model AMF assigns HA states of active or standby to an SU for handling an SI. In this thesis we discuss only 2N and N-way active redundancy model as we use only these in our example configuration.

2N redundancy model

In an SG with 2N redundancy model only one SU is assigned HA active state and only one SU is assigned HA standby state for all the SIs protected by that SG.

N-way active redundancy model

N-way active redundancy model does not support the standby service assignment but multiple SUs can be assigned active for one SI.

Service Group Type

A service group type specifies the list of service unit types that a SG of this type can support. All the SGs of a service group type follow the same redundancy model.

Application

An application is composed of set of SGs and the SIs protected by the SGs. Each SG can be part of one application only.

AMF Node

AMF Node is a logical entity on a cluster node. For example components and SUs are hosted on it.

2.4 Cluster Membership Service (CLM)

CLM [15] service provides users with the information about the nodes that are members of the CLM cluster. It keeps track of the nodes and also decides if a node can be part of the cluster. A CLM node has one to one optional relation with AMF node. A cluster consists of set of these nodes.

2.5 Platform Management Service (PLM)

The PLM Service [7] provides a logical view of the hardware and low-level software of the system. Low level software consists of lower layers within a machine such as Operating System (OS), Virtual Machine Manager (VMM) and Virtual Machine (VM).

The hardware and the software are represented as logical entities in the PLM configuration and PLM service provides APIs for the management of these logical entities. Some of the important logical entities in the PLM model are described in the subsequent subsection. Figure 2.2 shows how these logical entities are connected with each other.

Execution Environment (EE)

Low-level software of a machine is represented as an execution environment (EE). An EE must be capable of providing an environment required for the execution of software. An EE may or may not host a CLM node.

All EEs are modelled using objects of class *SaPlmEE*. Objects of this class realize the objects of class *SaPlmEEType,* as shown in Figure 2.2, which itself is inherited from the class *SaPlmEEBaseType.* This helps in the easy modelling of EEs belonging to the same type. VMs, VMMs and OSs are represented as objects of class SaPlmEE.

Hardware Element (HE)

A HE is a logical entity that represents any kind of hardware entity. It is represented as an object of class *SaPlmHE.*

SaPlmEntity

Class *SaPlmEntity* is a generalized class of *SaPlmEE* and *SaPlmHE*.

SaPlmDependency

A PLM entity can be dependent on other PLM entities. The dependent PLM entity cannot provide the service unless a certain number of minimum PLM entities on which it depends (sponsor PLM entities) are in service as well. This dependency between PLM entities is modelled using the class *SaPlmDependency.* For example, a VM capable of live migration has dependency to all of its sponsoring VMMs and it requires a minimum of one VMM in service for its operation. Therefore the dependency between a VM and all of its sponsoring VMMs is represented by the object of class *SaPlmDependency*. However, such a VM will be child of the object of class *SaPlmDomain* (described in the next section) rather being child of any specific

VMM. This way, when VM will migrate to any other VMM, the VM will still have the same identity.

SaPlmDomain

This is the root class that aggregates all the PLM entities of a domain.

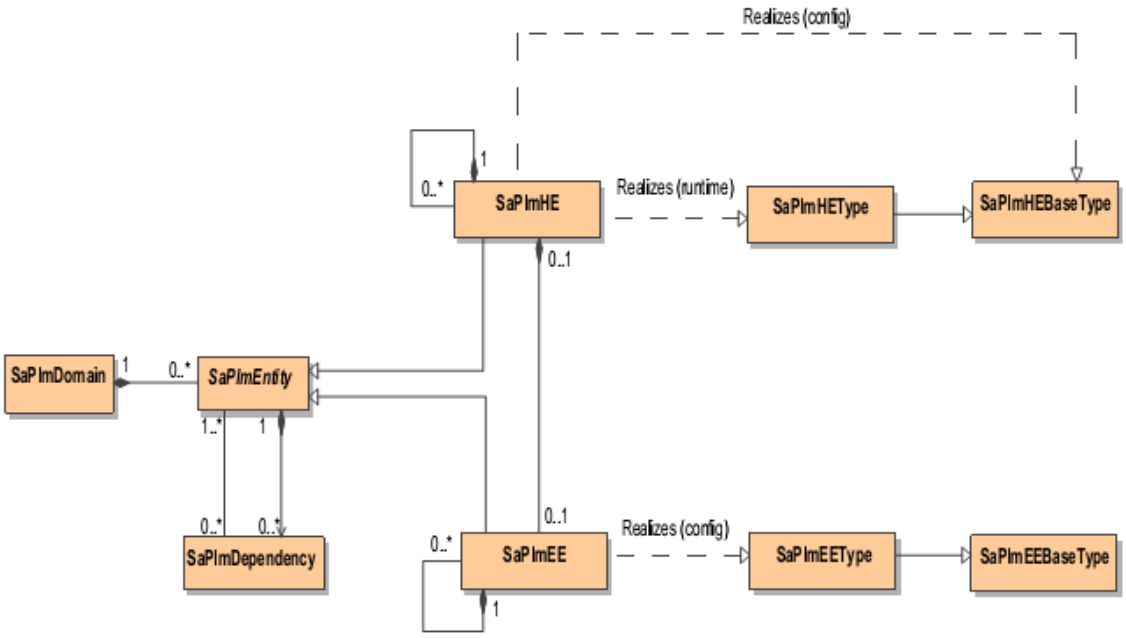


Figure2.2 Standard PLM model (taken from [7])

2.6 Information Model Management service

All the logical entities managed by various services (e.g. Components, SUs etc. provided by AMF, EE provided by PLM etc.) of an SA Forum cluster are represented as objects in the information model (IM). The information model is managed at runtime by and accessed through the Information Model Management (IMM) service [16].

Naming of objects presents in information model

All the objects in the information model are organized in a tree hierarchy. The hierarchy follows the structure of the LDAP [17] distinguished name of each object. Each object in the hierarchy distinguishes itself from its siblings using Relative Distinguished Name (RDN). A unique name of an object, also called distinguished name (DN) is constructed by concatenating RDN, a comma symbol: ',' and the DN of the parent object in the IMM tree.

2.7 Runtime behavior of AMF and PLM services

In this section we describe an example application installed on two machines (irrespectively it could be a virtual or physical machine). The application is managed by AMF and the EEs on the machines are managed by PLM. Below we discuss the behaviour of AMF and PLM services for the management of their respective entities.

Example AMF configuration

In Figure 2.3 we provide a sample AMF configuration of an application hosted on a two-node cluster which provides a media streaming service implemented using the Video LAN Client (VLC) media player 1.0.0. The media streaming service is protected using the 2N redundancy model. Each machine has an Ubuntu 10.04 OS with OpenSAF 4.2.0 middleware installed and running on it. As each AMF node maps into a CLM node and each CLM node maps into an EE; therefore for the simplicity we abstract away the CLM node from the figure and just show an AMF node mapped to an EE. The EE in our example represents the Ubuntu OS installed on the machine.

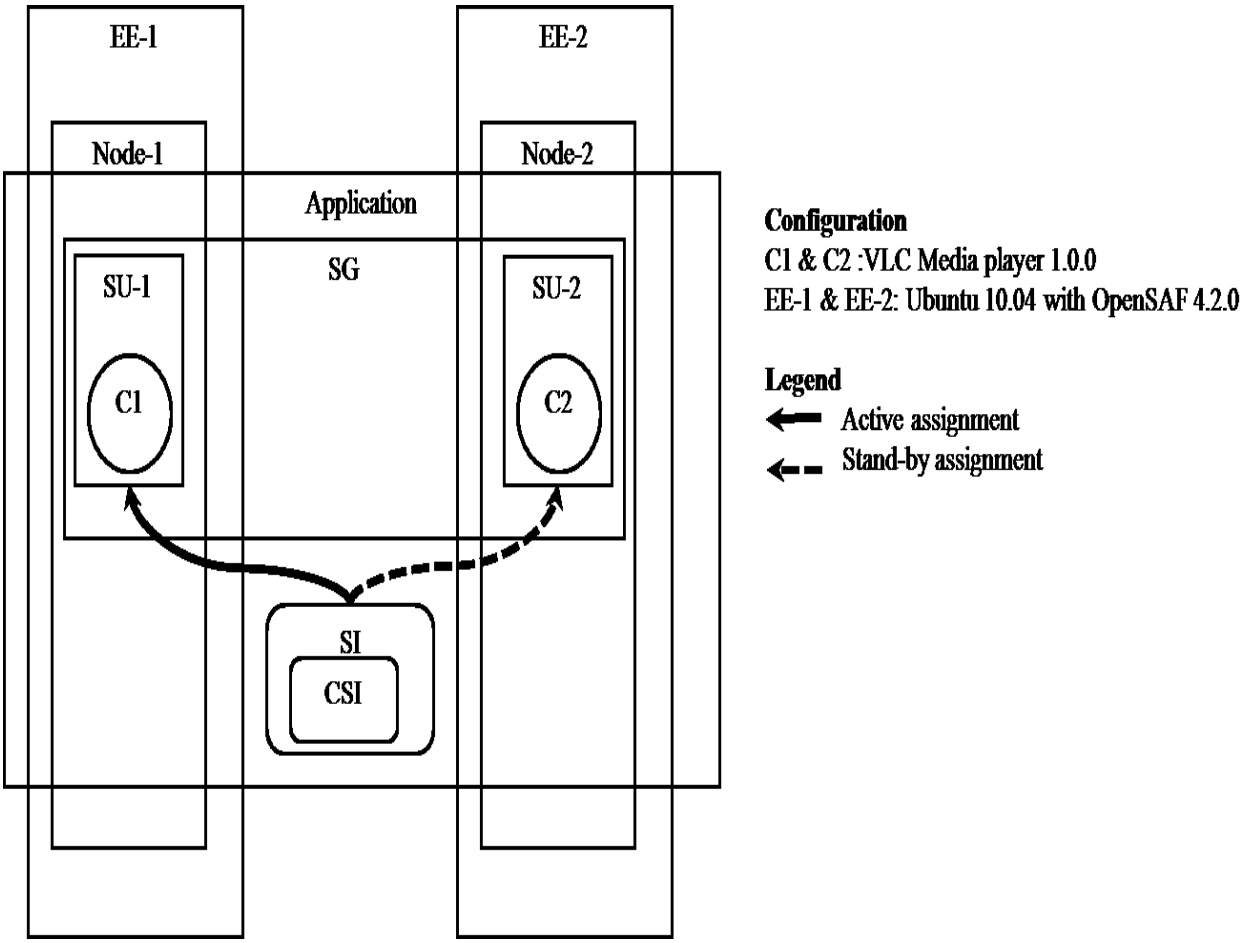


Figure 2.3 An example AMF configuration

At runtime AMF assigns HA active state for the SI-1 to the SU-1 and HA standby state to the SU-2 as shown in Figure 2.3. If a failure is detected in the SU-1, AMF fails-over the HA active state to the SU-2. Also, even without a failure in SU-1, the HA active state can be dynamically switched over to the SU-2 through the administrative operation – "Lock" provided by AMF, CLM and PLM services.

Lock operation when applied on EEs, SUs or AMF Nodes implies that these entities are not allowed to provide service. Lock operation on CLM node sets the CLM node out of the cluster membership (if it is member of the cluster). Locking an EE has the same effect on CLM node as locking a CLM node. Putting a lock on EE will terminate all the EEs and the SUs that were

hosted on the EE on which lock is applied. With the termination of SUs AMF cannot assign HA state for any SI to the terminated SUs.

As an example, putting a lock on EE-1 implies that EE-1 no longer provides service. As SU-1 has HA active state for SI-1 hosted on locked EE-1; therefore AMF will switch over the active assignment to the SU-2. Locking an EE affects the dependent EEs if there are not minimum number of sponsors in service that are required for the operation of the dependent EE. If an EE (dependent EE) is consuming service from a sponsoring EE then with the lock on that sponsoring EE, the dependent EE will start consuming services from its other unlocked sponsoring EEs. For example, when a VMM is locked all the VMs that depend on the locked VMM cannot be hosted on the locked VMM hence they migrate to the other sponsoring VMM.

These behaviours described as a result of administrative operation is particularly useful during the upgrade so that service can be switched over to the standby service provider when the active service provider is undergoing maintenance operation. In the next section we describe Software Management Framework which uses this feature as one of its action on entities so that upgrade can be performed on them.

2.8 Software Management Framework (SMF)

SMF [8] is a middleware service that orchestrates the software upgrade of a live system. It accomplishes this task in a tight collaboration with other SA Forum middleware services such as AMF, IMM etc. Before an upgrade is started by the SMF it takes some protective measures which include creating a backup of the system and the upgrade history. Backup of the system is required in case a full cold restart is required which restores the system to the state in which it was at the moment the backup was created. Upgrade history is required for undoing (discussed in

Section2.8.4) the executed actions of the current upgrade step or for rolling back the upgrade campaign (discussed in Section 2.8.5). SMF performs the upgrade as defined in an upgrade campaign specification. An upgrade campaign specification is an XML file which describes the flow of operations that takes the system from the source system configuration (current system configuration) to the target system configuration (desired system configuration).The SMF specification defines the XML schema for an Upgrade Campaign Specification [6]. An Upgrade Campaign Schema contains various SMF concepts and their attributes:

2.8.1 Upgrade step

An upgrade step is a sequence of actions that are applied on a set of entities. These actions logically belong together i.e. either all of the actions of a step are successfully executed on the set of entities or none of the actions. Hence SMF also defines undo of the upgrade step which reverses the effect of the actions performed by an upgrade step. SMF also defines rollback of an upgrade step which is applicable only at the step boundary. Rollback of an upgrade step reverts a successfully completed step of an upgrade campaign.

An upgrade step has either a deactivation unit and/or an activation unit or it can have a symmetric activation unit. These are described because the impact of the upgrade may not only be limited to the entities that are being upgraded but can affect the availability of entities that are not being upgraded. Hence all the affected entities have to be taken out of service while the availability of service is carried out from the redundant entities.

Deactivation Unit (DU)

Deactivation Unit describes the collection of software entities that need to be taken out-of-service so that they do not provide services while the upgrade step is applied to a set of entities.

Activation unit (AU)

At the end of the upgrade step software entities (that got added as a part of the upgrade step and some or all of the deactivated entities) need to be activated or reactivated. This collection of entities is called activation unit.

Symmetric Activation unit (SAU)

When the AU is equal to the DU it is described only once as a Symmetric Activation Unit (SAU), and it plays both the role of the DU and the role of the AU. In this unit entities are only being upgraded and no entities are added or removed from the system.

Upgrade step type

SMF defines two types of upgrade step: **normal upgrade step** and **reduced upgrade step.**

2.8.1.1 Normal upgrade step

In general a step for upgrading software consists of the following phases carried out in sequence as follow:

- Tear-down phase: In this phase the old software products are terminated and uninstalled.
- Reconfiguration phase: In this phase the logical entities that are not required any more are removed from the information model and the new logical entities that will be required are added in the information model. The entities that are being modified are also reconfigured in this phase.
- Build-up phase: In this phase the new software products are installed and instantiated.

Before uninstalling the old software, the DU is locked and terminated. This way AMF switches over the HA active state to the standby SUs for all the SUs that were locked by the “lock”

operation. Therefore the service availability during the execution of an upgrade step is maintained. Once the new software is installed during the build-up phase, the AU is instantiated and unlocked. Hence the new or the upgraded entities are once again available for active or standby assignment.

However, locking a DU for an entire step increases the time frame during which entities are out of service. Therefore to decrease the time frame we can use the concepts of online and offline (un)installation. An online (un)installation does not interfere with any entity in the system, especially not with one that is managed by the AMF, however offline operations do. Accordingly, an offline operation has a scope of impact which may impact other entities in the system too. To avoid any unplanned impact these entities must be put offline. On the other hand the online operations - (un)installation need not be executed after the lock is applied on the DU. To take the online operations outside the scope of lock, online un-installation of old software and online installation of new software are moved out of their respective phase and are put into build up phase and tear down phase, respectively. The resulting sequence of actions has been termed in SMF as the normal upgrade step. Figure 2.4 shows how a normal upgrade step is constructed by taking online operations out of phase. The right most sequence of actions in the figure shows sequence of actions of a normal upgrade step. Note that because of the out of phase nature of the normal upgrade step, we refer this step as “Out-Of-Phase” normal upgrade step in this thesis.

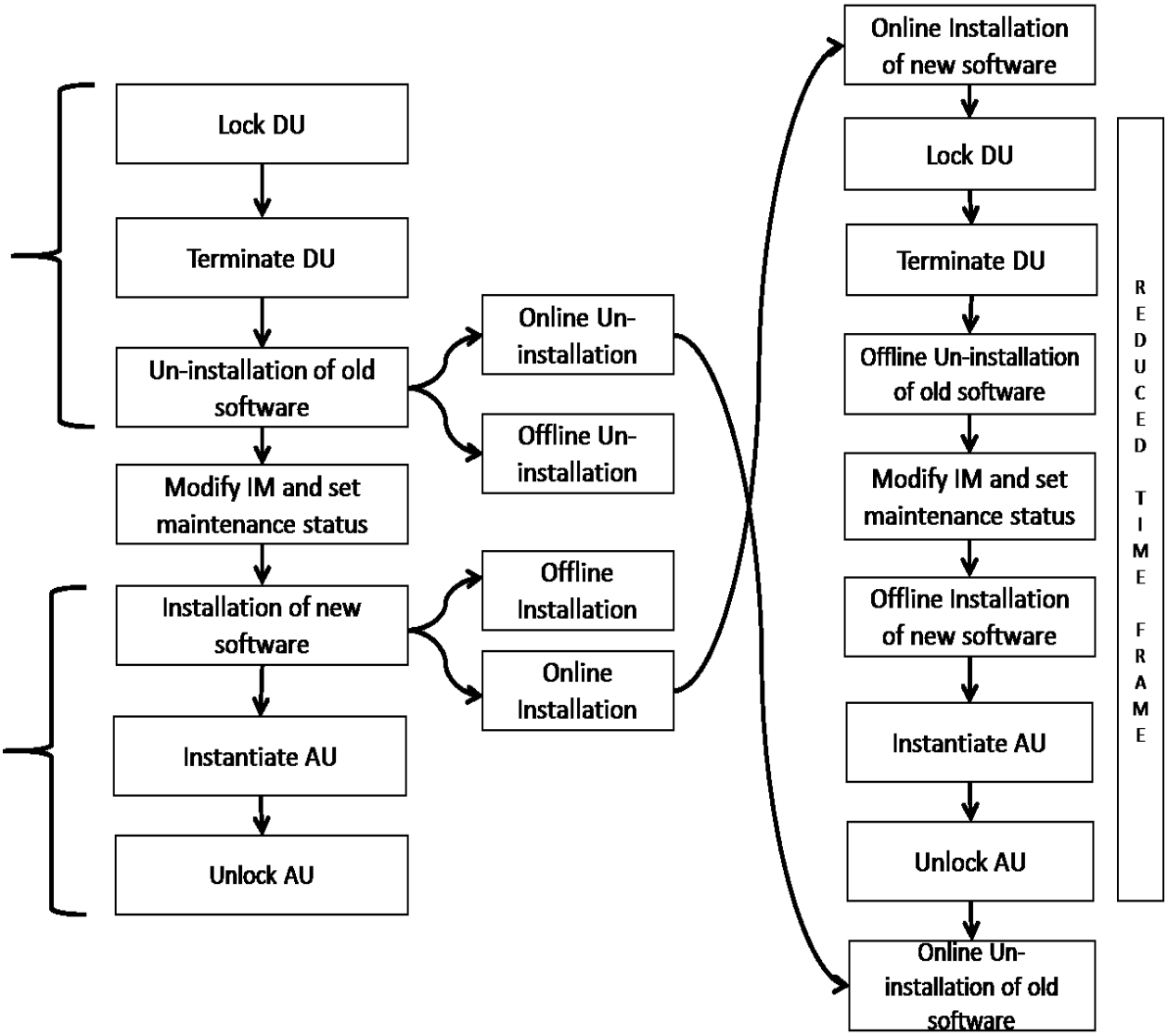


Figure 2.4 Deriving normal upgrade step

2.8.1.2 Reduced upgrade step

Reduced upgrade step is applicable for entities that have no offline operations (therefore no lock operation is required) and the time taken to restart entities in symmetric activation unit is less than the switch over time. This step is created from the "Out-Of-Phase" normal upgrade step by removing the offline operations from it and merging the actions: Terminate Deactivation Unit and Instantiate Activation Unit into Restart of Symmetric Activation Unit. Figure 2.5 shows how a reduced upgrade step is constructed from the normal upgrade step.

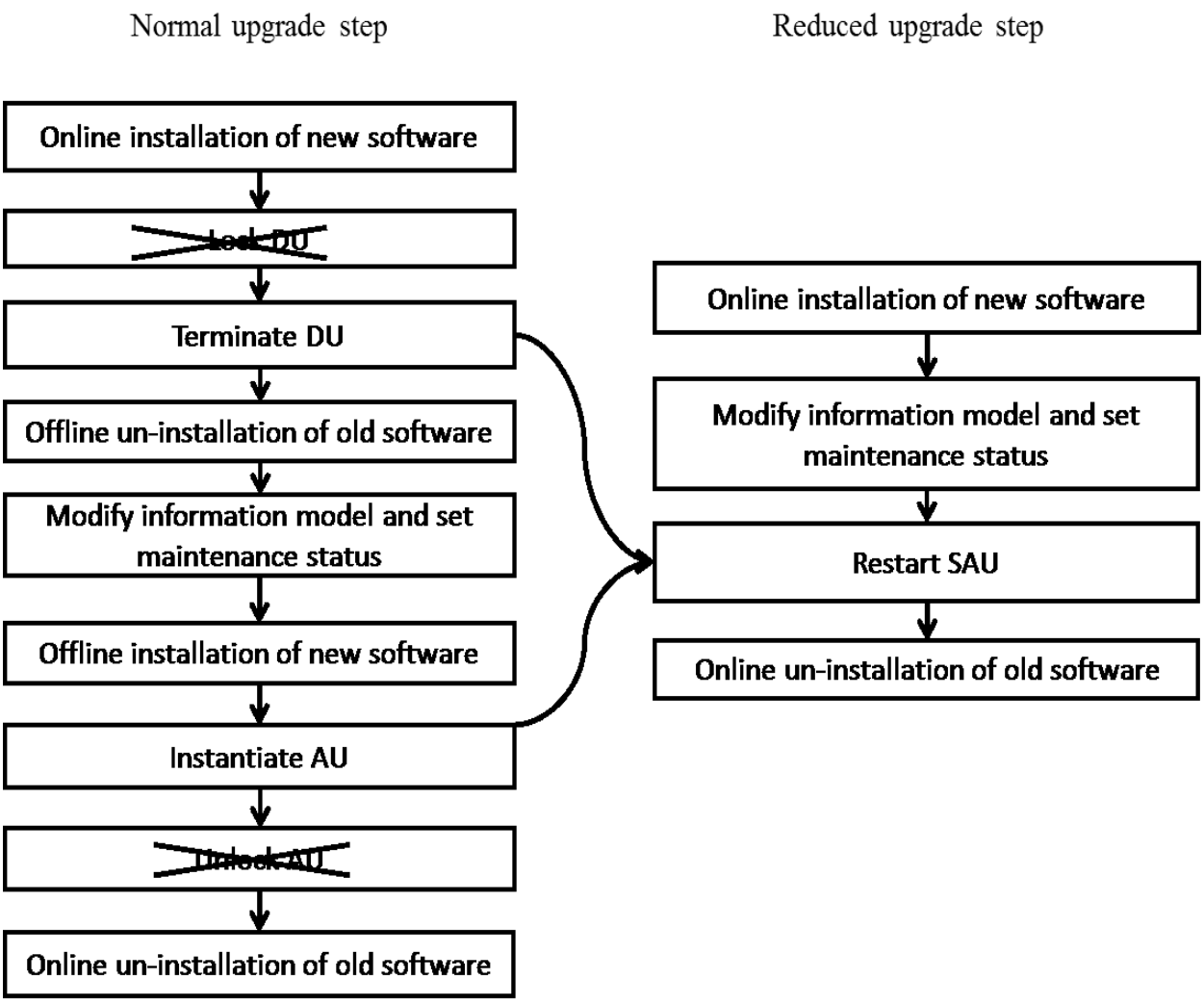


Figure 2.5 Reduced upgrade step

2.8.2 Upgrade procedure and upgrade scope

An upgrade procedure applies the same upgrade step on a set of similar entities under the constraints specified by the upgrade method (see Section 2.8.3) it is using. An upgrade campaign can have more than one upgrade procedure and their order of execution is determined by its associated property: execution level. Upgrade procedures are executed with the increasing order of their execution level. Upgrade procedures with same execution level are executed in parallel.

The set of software entities that will be affected by an upgrade procedure is called the upgrade scope of the upgrade procedure.

2.8.3 Upgrade method

An upgrade method describes the restrictions that must be observed while applying an upgrade step over a set of similar deactivation-activation unit pair. There are two types of upgrade methods defined in SMF: Single step upgrade and rolling upgrade.

- **Single step upgrade:** In a single step upgrade, upgrade scope is equivalent to one deactivation-activation unit pair. An upgrade step is applied at once on all the entities present in deactivation-activation unit pair. This step is intended for the use where service availability is not a concern e.g. addition of new entities or removal of old entities from the system. In the case of the addition of the new entities either the new entities will provide the service or are already provided by other existing entities. In the case of removal either the services from the service providing entities are not required or are provided by some other entities.
- **Rolling upgrade:** In this upgrade method, a set of redundant entities protecting a service is partitioned into smaller non-intersecting sub-sets such that taking one sub-set at a time and putting it out of service will not cause service outage. Such a sub-set is a deactivation-activation unit pair and the set is upgrade scope. On these sub-sets upgrade step is applied one by one until there are no more subsets left there by completing the upgrade of entire upgrade scope.

2.8.4 Undoing an upgrade step

If there is any failure in the execution of the actions within an upgrade step, that upgrade step must be undone by reversing the effect of all the successfully executed actions and bringing the state of the system back to the state where it was before the beginning of the upgrade step. Once in that state the upgrade step can be retried if permitted. To undo an upgrade step, reverse actions of all the successfully completed actions are executed in the reverse order of the execution. In the Table 2.1 we define the reverse action for each action.

Upgrade Step	Action Reversing Action
Online installation of new software	Online un-installation of new software
Lock deactivation unit	Unlock deactivation unit
Terminate deactivation unit	Instantiate deactivation unit
Offline un-installation of old software	Offline installation of old software
Modify information model and set maintenance status for activation unit	Reverse information model modifications and set maintenance status for deactivation unit
Offline installation of new software	Offline un-installation of new software
Instantiate activation unit	Terminate activation unit
Unlock activation unit	Lock activation unit
Restart activation unit	Restart activation unit
Online un-installation of old software	Online installation of old software

Table 2.1 Reverse actions depending on the Upgrade Step

2.8.5 Rollback

In case an administrator decides to roll back an upgrade campaign, all successfully executed upgrade steps have to be reversed so that configuration effective at the beginning of the upgrade

campaign can be recovered. In such a case rollback of each upgrade step has to be executed. Below we define the rollback of normal upgrade step and reduced upgrade step, respectively.

Rollback of normal upgrade step

1. Online installation of old software
2. Lock activation unit
3. Terminate activation unit
4. Offline un-installation of new software
5. Modify information model to old configuration and set maintenance status for deactivation unit
6. Offline installation of old software
7. Instantiate deactivation unit
8. Unlock deactivation unit
9. Online un-installation of new software

Reduced upgrade step

1. Online installation of old software
2. Modify information model to old configuration and set maintenance status
3. Restart symmetric activation unit
4. Online un-installation of new software

2.8.6 Finite State Machine (FSM) for modeling the execution of an upgrade step

The execution of an upgrade campaign by the SMF is modelled by three FSM, one for the upgrade campaign: campaign FSM, a second one for the upgrade procedure: procedure FSM and a third one for the upgrade step: step FSM. Also an upgrade campaign and its associated upgrade procedures and upgrade steps communicate with each other. For example upgrade procedure sends message to upgrade step to start executing which puts an upgrade step into the Initial state; and once the execution of an upgrade step is completed it pass the appropriate message to the upgrade procedure. We use the term FSM as it is used in the standard of SMF, but these machines are actually Extended Finite State Machine (EFSM). As we only modify the FSM of the upgrade step in the later chapter therefore we introduce only the step FSM in our thesis. More information on the other two FSMs can be found in [6]. The FSM in Figure 2.6 models the execution of an upgrade step by SMF [6]. The following conventions are used in the state diagrams throughout the thesis:

- A double circle denotes the initial state of the FSM.
- A thick bordered circle represents a final state.
- Transition between the states is represented by an arrow.
- Each transition is labelled with the input and the output interactions separated by a slash. Input signal starts the state transition and output signal, if applicable, is the one produced during the state transition.
- Input received from another FSM or output sent to another FSM is tagged with that FSM. Tag used for an upgrade procedure is: Proc. For example if a signal is received/sent from/to a FSM of an upgrade procedure then it is shown as Proc: signal.

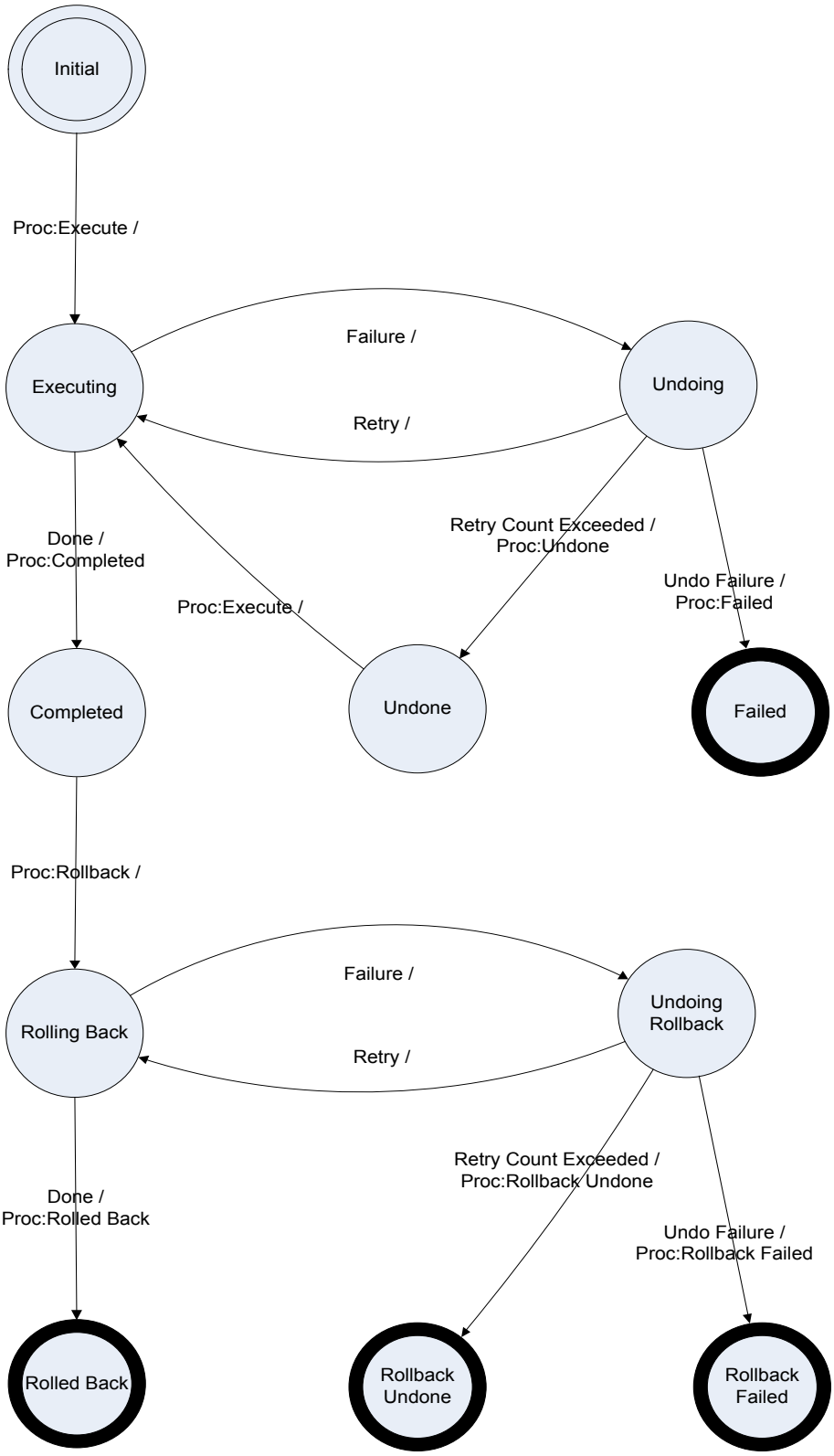


Figure 2.6 Finite state machine for an upgrade step (taken from [6])

Initial state

A step starts from an initial state and upon receiving Proc: Execute signal from the procedure to which this step belongs, the step starts executing and hence the state of the step is moved to Executing state.

Executing state

In this state all the actions of the upgrade steps are executed. During the execution if any failure (Failure) is detected the step moves to the state Undoing. If the execution of the upgrade step is successful i.e. no error is detected (Done) the step transitions to the state Completed and sends a signal Proc: Completed to the procedure to which it belongs.

Completed state

As mentioned already, this state indicates the successful execution of an upgrade step; however from this state, the step can make the transition to the state Rolling Back on receiving the signal Proc: Rollback from the procedure to which the step belongs. This transition is necessary as the administrator may decide to roll-back the campaign.

Undoing state

In this state all the reversing actions of the upgrade steps are applied. If any failure (Undo Failure) occurs during undoing the step, the step moves to the state Failed (step cannot be undone). In case undoing of an upgrade step is successful and retrying (Retry) is permitted, the step moves to the Executing state otherwise (Retry Count Exceeded) it transitions to the Undone state and sends the signal: Proc: Undone to the procedure to which it belongs.

Failed state

This is a final state for the step representing that a double failure has happened and the executed actions cannot be reversed.

Undone state

This state is similar to the Initial state. From this state the step makes a transition to the Executing state on receiving the signal Proc: Execute due to an administrative operation. Such a transition is required because execution of an upgrade step may have been failing due to a reason that can be solved by the administrator interference (such as disk space limitation).

Rolling Back state

In this state actions of the upgrade step are rolled back. If it is successful (Done) then the step makes the transition to Rolled Back state and sends the output signal Proc: Rolled Back to the procedure to which the step belongs. While rolling back if there is any failure (Failure), the step makes a transition to the Undoing Rollback state.

Undoing Rollback state

If undoing the rollback is successful and retrying (Retry) is permitted then the step moves to the Rolling Back state otherwise with Retry Count Exceed signal it moves to the Rollback Undone state and sends the signal Proc: Rollback Undone to the procedure to which it belongs. If there is any failure (Undo Failure) while undoing rollback, the step moves to the Rollback Failed state and sends the signal - Proc: Rollback Failed to the procedure to which it belongs.

Rolled Back state, Rollback Undone state, Rollback Failed state

Rolled Back state, Rollback Undone state, Rollback Failed state are the final state of the step.

2.9 Related work

The closest work to ours is by the OpenSAF [13]. The approach taken by the OpenSAF to facilitate the upgrade of lower layer in a machine modifies the "Out-Of-Phase" normal upgrade step by embedding the reduced upgrade step in it. This approach, first, is applicable only for the application layer (upgraded through “Out-Of-Phase” normal upgrade step) and the OS layer (upgraded through “Reduced upgrade step”). Second, the approach essentially tweaks both the upgrade steps to perform the upgrade. Reduced step embedded inside the “Out-Of-Phase” normal upgrade step restarts the node rather than restarting an entity represented in the IM. Third, the approach cannot be used if the upgrade of OS requires offline installation. Also in the context of the MAGIC project [18], a previous member worked on the automatic generation of the upgrade campaign [8] but that thesis covers only AMF entities.

Beside the SA Forum, the Object Management Group (OMG) [19] also standardizes middleware for the online upgrade [20]. OMG has developed standard specification for online upgrade of the Common Object Request Broker Architecture (CORBA). But, it does not address the availability, instead it recommends the combination of online upgrade and the fault tolerant CORBA. Furthermore, the objects used in CORBA have lower granularity than the SA Forum entities.

There are other solutions to upgrade lower layers of the machines in a cluster while maintaining High Availability but these solutions are proprietary and hence do not apply in the context of SA Forum.

Automation tool such as Puppet [21] is widely used in the industry for software upgrades, but by itself it does not consider service availability. Preparing a recipe to automate the upgrade process

in a SA Forum system using Puppet would require scripting service availability awareness into Puppet. SMF orchestrates the upgrade process with this awareness and is tightly coupled with AMF and the other SA Forum services. The same is true for package managers like Apt [22] and YUM [23]. Package managers in different nodes do not collaborate with each other and cannot orchestrate an upgrade to maintain service availability. However, package mangers and Puppet can be used along with SMF where SMF orchestrates the upgrade process and package managers or Puppet performs the required task of installation and un-installation on individual machines.

Chapter 3 - SMF and its limitations for lower layers upgrade

In this chapter, we investigate the aspects and issues in the live upgrade of lower layers (PLM entities) under the HA constraints. We consider three different aspects of the live upgrade: type of upgrade, the booting mechanism and the type of execution environment (see Section 3.1). After this we examine and demonstrate the limited applicability and the limitations of the existing upgrade steps for lower layers (see Section 3.2). Finally we conclude the chapter in Section 3.3.

3.1 Issues and complexities in lower layers upgrade

Upgrade of any layer from the software stack can cause incompatibility with the layer installed and relying on it. For example, an upgrade of lower layer such as OS can make the application layer consisting of software such as media player incompatible with the upgraded type of OS. Ultimately, this incompatibility will also affect the application layer software in that machine and hence will cause service outage. Therefore if a layer is being upgraded then all the impacted layers need to be upgraded as well. Also for the installation, the un-installation and the execution of a layer a compatible and running lower layer is required. Although, the information about the compatibility between the different layers of a machine in a cluster is provided as an input; yet for each lower layer we look into different aspects that need to be taken into account. We discuss them in this section and discuss the challenges associated with each aspect.

3.1.1 Type of upgrade

This aspect is related to the relation between the source and the target EE of the upgrade. We have identified four scenarios.

Base type change of the EE

It involves switching from the EE software provided by one vendor to the EE software provided by another vendor, e.g. switching from VirtualBox to VMWare or from Ubuntu OS to Windows OS. In such a case the EE to be upgraded may become incompatible with the other layers on that machine thus affecting the service availability from the application layer as well. Therefore all the old layers incompatible with the new EE has to be uninstalled and new layers compatible with the new EE need to be installed, and the upgrade of the whole system must be planned and executed carefully to avoid service outage at the application layer.

Version change of the EE

This involves upgrade of an EE software version, e.g. upgrade of VirtualBox from version 4.2.20 to version 4.3.6 or upgrade of Ubuntu from version 9.04 to 10.04. In this case there is a high probability that the layers installed on the old EE do not need to be upgraded as we usually have backward compatibility. Irrespectively, upgrade of that EE will require terminating and instantiating or restart of that EE again which implies all the layers above that EE will be terminated and instantiated or restarted too. These actions will impact the application software and hence affect the availability of services (e.g. upgrade of Ubuntu 9.04 to Ubuntu 10.04 can be performed online, however to bring the upgraded OS, a restart has to be performed which will restart the SU hosted on the OS that could be providing service). Hence in this case the challenge resides in planning the EE upgrade without service outage from the user perspective.

Clean installation of an EE

Clean installation of an EE and all the hosted EEs is equivalent to bringing a new machine into the cluster. This does not cause any specific challenge for HA as we are adding an EE.

Removal of an EE

An EE and all the hosted EEs are removed in this situation. Since an EE is being removed appropriate actions must be taken as a part of the upgrade so that the services remain available.

3.1.2 Booting mechanism

An EE can boot locally from a disk containing an image or remotely. When an EE boots from the network using a bootable image stored in a remote machine it is called remote booting. In this case the upgrade requires bringing the new software (EE software and the application software), to the machine and restarting them one by one with the updated path to the new software. In local booting, EE boots from the disk configured locally containing EE software installed on it. To automate the upgrade of EEs in a disk contained machine a minimal requirement of accessing the machine remotely should be maintained.

3.1.3 Type of EE

An EE can be virtualized or a real one (i.e. running directly on the hardware of the machine). When the environment is virtualized and the VMs are configured for live migration it brings dependencies to multiple VMMs in picture as well which needs to satisfied during the upgrade. With the virtualization all the advantages provided by virtualization can be exploited, for example for creating a new virtual machine and instantiating it from the available snapshot of the VM image.

3.2 SMF as a solution for the live upgrade of lower layers

In this section we review the existing upgrade steps of the SMF. Based on the review we look into the limited applicability of these steps for the upgrade of lower layers (PLM entities).

3.2.1 Applicability of existing upgrade step for lower layers upgrade

Although the existing upgrade steps suffer from some limitations (discussed in Section 3.2.2) yet they can be used for the upgrade of lower layers (PLM entities) under certain conditions. In the subsequent subsections we discuss the limited applicability of the existing upgrade steps. The limitations for each upgrade step are presented using an example which considers the aspects discussed in the previous section. Note that in all the example rolling upgrade method is used so that only one machine is taken out of service at a time and hence does not affect the service availability.

3.2.1.1 Applicability of "Out-Of-Phase" normal upgrade step for PLM entities

An "Out-Of-Phase" normal upgrade step is applicable under the following conditions:

1) "Online installation of new software" and "online un-installation of old software" must be compatible with the layers on which these actions are executed.

2) Different layers of software have to be bundled as one installation image. (e.g. Ubuntu OS disk image bundled with two layers comprising of OS itself and the application software such as VLC media player). Doing so will make all the software from different layers part of the lowest which has to be installed. Hence layered upgrade will not be a concern.

As an example let us consider the AMF configuration shown in Chapter 2 – Section 2.7deployed on nodes with the booting mechanism: local booting and the type of EE: OS running directly on the hardware (real environment). The type of upgrade the system undergoes: the version of the OS of each machine of the cluster needs to be upgraded to Ubuntu 12.04 with OpenSAF 4.2.2 middleware and VLC media player 2.0.0 while maintain the HA of services. Also, the software

for the upgrade comes as a single image bundling up entire software stack - Ubuntu 12.04 with OpenSAF 4.2.2 middleware and VLC media player 2.0.0; and the (un)installation of the software image requires some offline operations.

Considering that in the version change of the OS the online operations can be performed on the old version of the OS; therefore upgrade of such a cluster is performed by using "Out-Of-Phase" normal step.

3.2.1.2 Applicability of reduced upgrade step for lower layers

Reduced upgrade step is applicable on lower layers only if all the following conditions are met:

1) Upgrade of an EE from a source type to a target type in particular type of system organization can be carried out by restarting the EE. For example, a machine booting from remote image of an OS can be upgraded by setting the boot path to the new OS image and restarting that machine.

2) The restart time for all the entities in symmetric activation unit (EEs and other entity such as SUs hosted on the EEs) is less than the switch over time.

3) Upgrade should not contain any offline operations.

4) It must follow the applicability constraints of the "Out-Of-Phase" normal upgrade step as described in the previous section.

Furthermore, this step type is also useful for entities where "terminate" action followed by "instantiate" action is not feasible, e.g. terminating the OS may make machine inaccessible and hence "instantiate" action cannot be executed, but restart action will not let the machine become inaccessible.

As an example let us consider again the AMF configuration in Chapter 2 – Section 2.7 with the type of EE: OS running in a virtual environment. Type of upgrade the system undergoes: the version of OS of each machine of the cluster has to be upgraded to Ubuntu 10.10 with OpenSAF 4.2.2 middleware and VLC media player 1.1.0. Note that this example is independent of the booting mechanism.

Once again, considering that in the version change of the OS online operations can be performed on the old version of the OS; therefore upgrade of such a cluster is performed using reduced upgrade step provided the following conditions are met: (1) Installation of software image requires no offline operation and the entire software stack is bundled as one software image, (2) With the support of virtualization it is possible to achieve a configuration with the restart time of entities in the symmetric activation unit (OS, middleware and VLC media player) less than the switch over time to standby VLC media player, (3) The streaming state of VLC media player is stored outside of the VM so that it does not go away with the old VM.

3.2.2 Limitations of the existing upgrade step

Layered upgrade

Existing upgrade steps do not allow embedding to provision layered upgrade. With the existing upgrade steps if multiple layers need to be upgraded then the DU will contain the entities from all the layers and the termination of this DU will terminate all the layers. This implies the unavailability of the lower layers required for the un-installation or the installation of the upper layers.

Incompatible actions

Existing SMF steps do not consider the compatibility of the online actions with the layer on which it is executed. Consider the example discussed in the Chapter 2 - Section 2.7. As a part of

system upgrade, for each machine in the cluster the OS and the media streaming service provider has to be modified to Windows OS and Windows Media Player. The existing upgrade step would perform the online installation of the new software (Windows OS based) on the Ubuntu OS; thus resulting in a failure.

Limited applicability of the reduced step

The reduced upgrade step is applicable for the entities with the restart time lesser than the switch-over time. On the contrary restart time of EEs are generally very high. For example reduced upgrade step is not applicable for the upgrade of OS as its restart together with the middleware and the AMF entities takes more time than switching over their services to the appropriate standbys.

3.3 Conclusion

In this chapter we described the limitations of the existing upgrade step. We also defined the conditions under which the existing upgrade steps can be used for the upgrade of lower layers. Because of the limitations in the existing upgrade steps, in order to upgrade multiple layers of a machine the installation image has to include the complete software stack as one layer. In the next chapter we propose new upgrade steps which will overcome the limitations described in the existing upgrade steps.

Chapter 4 - New upgrade steps for SMF

In this chapter, we propose three new upgrade steps so that the lower layers of a machine can be upgraded without impacting the availability of services provided by the application software and hence extend the scope of the SMF to PLM entities as well (see Section 4.1). For these steps we also define their undo (see Section 4.2) and rollback (see Section 4.3) which are required for failure handling. Among the new upgrade steps, the upgrade step designed specifically to upgrade multiple layers of a machine in a system does not follow the same execution scheme as described in Chapter 2 – Section 2.8.6. Therefore, we make the changes in the step FSM so that it can handle the normal execution, the rollback and undoing of all the upgrade steps. We describe the necessity of these changes and the changes done in details in Section 4.4. We end the chapter with closing remarks (see Section 4.5).

4.1 New upgrade steps

To overcome the limitations, as seen in Chapter 3, of the SMF with respect to the upgrade of lower layers we propose three new step types. Below we describe these three step types. For the provided examples, as previously, we also upgrade the cluster using rolling upgrade method to prevent service outage.

4.1.1 Locked reduced upgrade step

This step type is an extension of the reduced step with the lock and the unlock actions. With this modification we can take advantage of the reduced step and use the restart action for the cases when the restart of the symmetric activation unit would cause longer service outage than its switch-over. The introduced lock forces a switch-over.

The locked reduced step type is defined with the following sequence of actions:

1. Online installation of new software.
2. Lock symmetric activation unit.
3. Modify information model and set maintenance status.
4. Restart symmetric activation unit.
5. Unlock symmetric activation unit.
6. Online un-installation of old software.

As an example, consider the AMF configuration of Chapter 2 – Section 2.7 deployed on machines with the booting mechanism: remote booting and the type of EE: OS running directly on the hardware (real environment). The type of upgrade the system undergoes: the version of the OS of each node of the cluster has to be upgraded to Ubuntu 10.10 with OpenSAF 4.2.2 middleware and VLC media player 1.1.0.

For such an upgrade, locked reduced step is appropriate provided that the software stack comes as a single image and its installation requires no offline operation. Using the locked reduced step it is assured that the services are switched over and remain available during the restart.

4.1.2 "In-Phase" normal upgrade step

To eliminate the limitation of "incompatible actions", online operations can be moved back to the phase to which they belonged originally. This was the starting point from which we constructed the "Out-Of-Phase" normal upgrade step (explained in chapter2, section 2.8.1.1). Although this will increase the time frame during which entities are locked but will ensure that the installation and un-installation of an upper layer do not happen on an incompatible lower layer. Below we define the sequence of actions for the "In-Phase" normal upgrade step.

1. Lock deactivation unit.
2. Terminate deactivation unit.
3. Offline un-installation of old software
4. Online un-installation of old software
5. Modify information model and set maintenance status
6. Online installation of new software
7. Offline installation of new software
8. Instantiate activation unit
9. Unlock activation unit

As an example, consider the AMF configuration of Chapter 2 – Section 2.7 deployed on the machines with the booting mechanism: local booting and the type of EE: OS running directly on the hardware (real environment). The type of upgrade the system undergoes: the base type of the OS has to be changed to Windows 8 with SA Forum compliant middleware compatible with OpenSAF 4.2.0 and VLC media player 2.0.0. Also the software for the upgrade comes as a single image bundling up the entire software stack.

As any (un)installation operation of Windows based software cannot be carried out on Ubuntu OS therefore for the upgrade of this cluster, “In-Phase” normal step would be the appropriate step type.

4.1.3 Embedded upgrade step

To address the limitation of "layered upgrade" we propose a new step – embedded upgrade step. An embedded step may nest any number of upgrade steps (also referred to as nested step from now on). Execution of an embedded step is not a sequential execution of the phases of its nested step. Its execution begins by the tear-down phase of each of its nested steps starting with the outermost. This is followed by applying the modifications of the information model from each of the nested steps. Finally the build-up phase is executed from each nested step starting with the innermost. The number of nested steps in an embedded upgrade step is defined as the *degree of embedding*.

In an embedded upgrade step each nested step upgrades a particular layer. The symmetric activation unit of individual nested steps should be defined such that while tearing down the software of a particular layer, e.g. layer '*i*', only the scope which represents that particular layer is locked and terminated hence the layer below, layer '*i+1*' is still providing service. Also, while building up the new layer '*i*', the layer below, layer '*i+1*', has been upgraded and instantiated so as to provide service to the layer above, layer '*i*', for its installation.

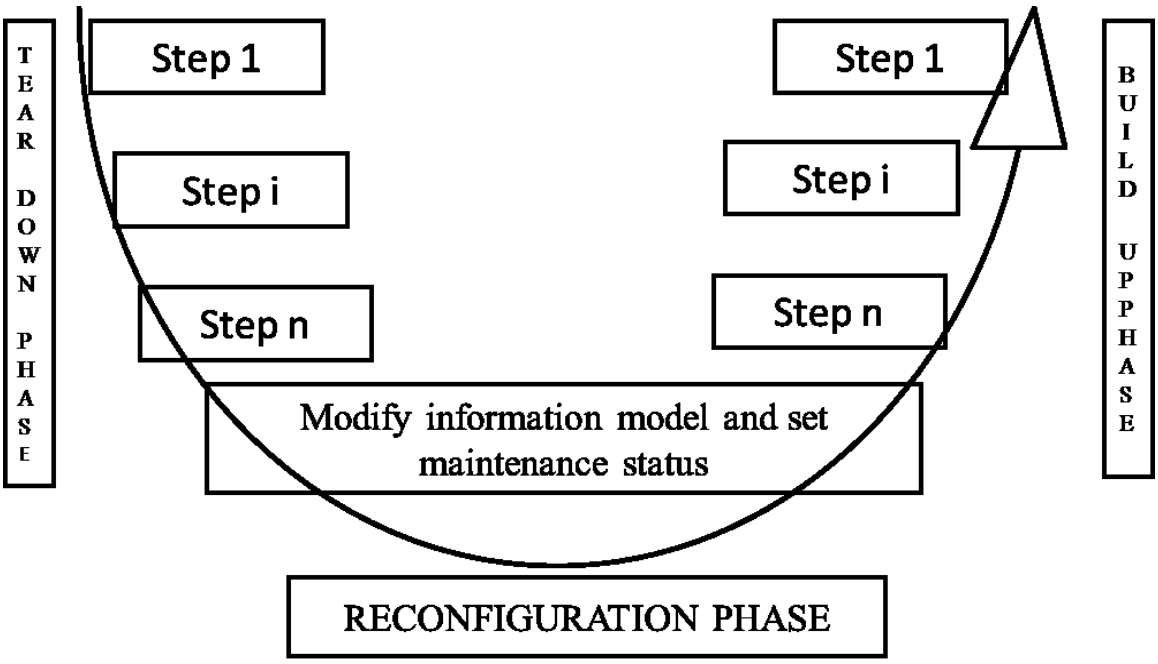


Figure 4.1 Normal execution of embedded upgrade step

The flow of execution of the embedded upgrade step is shown in the Figure 4.1 with degree of embedding = *n*. In section 4.1.3.1 "Rules of embedding" we define the rules that must be followed in nesting an upgrade step inside another nested step. Note that the out-of-phase actions of nested steps remain out of phase and are executed in the opposite phase of the given nested step. Also note that the outermost layer is the 'layer 1' and the innermost layer is the 'layer n'. This is in contrast to the general conventions where innermost layer is recognised as the layer 1.

4.1.3.1 Rules of embedding

There are certain rules that need to be followed when composing an embedded upgrade step using nested steps:

- Reduced or locked reduced upgrade steps cannot embed any further step, i.e. it can only be used as the innermost step.

- When using embedded step for addition or removal, the innermost step could be an "In-Phase" normal Upgrade step or an "Out-Of-Phase" normal upgrade but all the other nested steps must be "In-Phase" normal upgrade steps.
    - In an embedded upgrade step (degree of embedding = *n*) action from each nested steps are executed on the layer below it. In case of addition if an "Out-Of-Phase" normal upgrade step is used as one of the nested step (nested step *i*) except for the innermost step (*i!=n*) then the action "online installation of new software" of this nested step will be executed before the layer *i+1* was even added (because of the out of phase nature). Similarly in case of removal, online un-installation of the software will be executed after the layer below it has already been removed. Hence in the case of addition or removal only "In-Phase" normal upgrade step is used as a nested step except for the innermost nested step.

As an example let us again consider the AMF configuration of Chapter 2 – Section 2.7 deployed on machines with booting mechanism: local booting and the type of EE: OS running directly on hardware (real environment). The type of upgrade system undergoes: the base type of OS of each machine of the cluster has to be upgraded to Windows 8 with SAF compliant middleware compatible with OpenSAF 4.2.0 and VLC media player 2.0.0 for Windows OS. Unlike in other examples, let us assume that the VLC media player is not packaged with the Windows installation image.

As new VLC media player installation can be performed only when the Windows OS has been installed and instantiated on the respective machine therefore in this case embedded step has to be used. The degree of embedding for the embedded upgrade step is determined by the number

of the layers within a machine that have to be upgraded. As for each machine two layers (Layer 1: application software, Layer 2: OS) has to be upgraded therefore degree of embedding is 2.

As there is a base type change of Layer 2; online operations of an upgrade step that upgrade Layer 1 must be in its respective phase. Hence for Layer 1 nested step must be of type "In-Phase" normal upgrade step. Similarly for the Layer 2, the online installation of Windows OS cannot be performed on the Ubuntu OS; therefore upgrade step for Layer 2 is also "In-Phase" normal upgrade step.

The execution of the embedded step begins by executing the tear-down phase of the outer "In-Phase" step which locks, terminates and uninstalls the old VLC media player 1.0.0. Following, the inner "In-Phase" step tear-down phase is executed which locks, terminates and removes the old OS Ubuntu 10.04 and the middleware. During the reconfiguration phase the information model is modified to reflect the new software versions (Windows 8, VLC 2.0.0). After this the inner "In-Phase" nested step is executed which installs and instantiates the new OS – Windows 8 and unlocks it. Following this, the build-up phase of the outer "In-Phase" nested step is executed and installs VLC media player 2.0.0 and in the end instantiates and unlocks it so that the new AMF entities become available for service assignment. The embedded step is then rolled over to the next machine and the upgrade of the layers in that machine is carried out and hence completing the upgrade of all the machines in the cluster.

4.2 Undoing new upgrade steps

Undoing an "In-Phase" normal upgrade step or a locked reduced upgrade step is exactly the same as undoing an "Out-Of-Phase" normal upgrade step or reduced upgrade step described in

the Chapter 2 – Section 2.8.4. However, undoing an embedded upgrade step is not straightforward.

On an event of a failure during the build-up phase of a nested step (say step *i*) belonging to an embedded step it is not possible to undo just the step *i* because the old lower layers that provide services to the old software from layer *i* have been removed. Therefore, all the successfully executed nested steps (step *i+1*to step *n*) also have to be undone (i.e. install the old software and uninstall all the successfully installed new software from multiple layers of a machine). Therefore, if step *i* fails during the normal execution then only the phase in which error has occurred is undone. This way the layer *i* that was being upgraded using the step *i* reaches to the closest known stable state. This stable state of the layer *i* could be either the old software from the layer *i* are restored as part of undoing the tear-down phase or the new software from the layer *i* have been removed as a part of undoing the build-up phase. After undoing the required phase of step *i* it is retried if permitted. However, if the failure occurs in the innermost step then the entire innermost step is undone as there is no other nested step that needs to be undone to completely undo the innermost step. This also aligns the undoing of the innermost step to the undoing of non-embedded upgrade step ("In-Phase" normal upgrade step, "Out-Of-Phase" normal upgrade step, Reduced upgrade step, Locked reduced upgrade step).

If retry of the failed nested step is not permitted after a particular number of retries then the entire embedded upgrade step is undone. This is done by executing the reverse actions for all the actions that have been executed so far but in the order opposite of what is followed during the normal execution. Hence, the sequence of execution of undoing the entire upgrade step is opposite of what is shown in the Figure 4.1. In case there is a failure while undoing a nested step then the upgrade campaign has to fail as the state of the system has become unknown and

inconsistent. To maintain the availability of services this inconsistency must be removed as soon as possible. Therefore in such a scenario, to bring the system back into the consistent state a full cold restart of the system is ordered which restore the state of the system in which it was when the backup was created.

4.3 Rollback of new upgrade steps

The rollback of the new upgrade step follows the same concept as described in the Chapter 2 – Section 2.8.5. Old software products are installed in the build-up phase followed by reconfiguration and the un-installation of new software in the tear down phase. However as the new upgrade steps have different sequences of actions therefore in the subsequent subsection we define the rollback of the new upgrade steps.

Rollback of the "In-Phase" normal upgrade step

1. Lock activation unit
2. Terminate activation unit
3. Offline un-installation of new software
4. Online un-installation of new software
5. Modify information model and set maintenance status
6. Online installation of old software
7. Offline installation of old software
8. Instantiate deactivation unit
9. Unlock deactivation unit

Rollback of locked reduced upgrade step

1. Online installation of old software
2. Lock symmetric activation unit
3. Modify information model to old configuration and set maintenance status
4. Restart symmetric activation unit
5. Unlock symmetric activation unit
6. Online un-installation of new software

Rollback of embedded upgrade step

The rollback of an embedded upgrade step is similar to the normal execution of the embedded upgrade step, except that it deactivates the entities that were present in the original activation unit and un-install them and install the entities that were present in the original de-activation unit and activate them. While executing the rollback, the build-up phase of the original embedded upgrade step becomes the tear-down phase and the tear-down phase of the original embedded upgrade step becomes the build-up phase as shown in Figure 4.2. This figure shows the execution of the rollback of the embedded upgrade step with the arrow indicating the direction of execution.

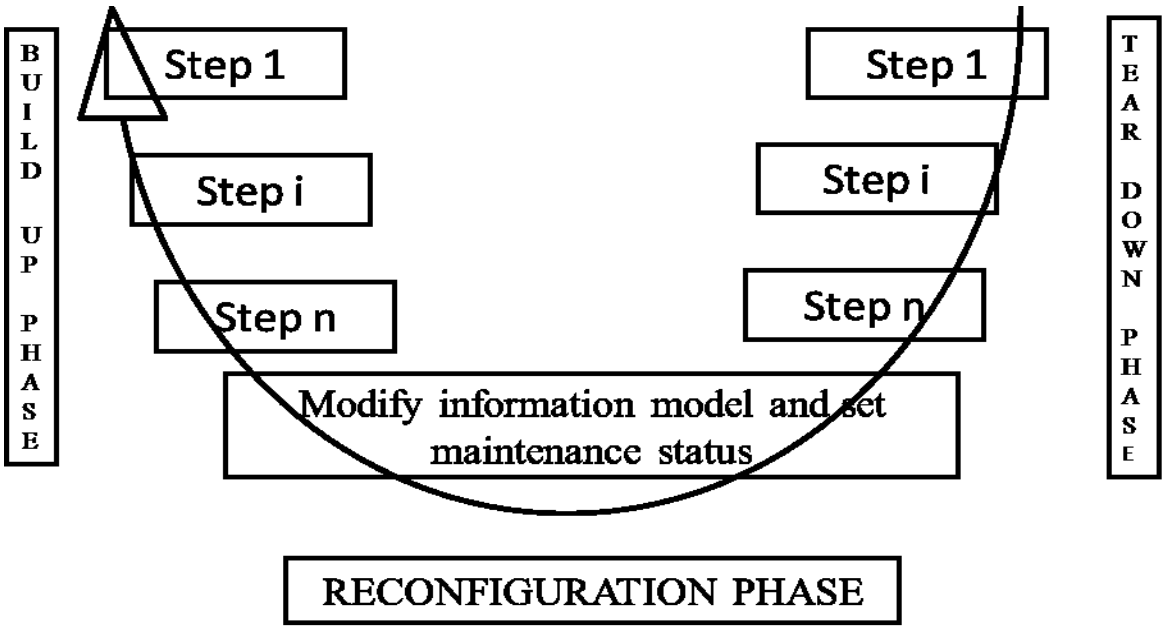


Figure 4.2 Execution of rollback of an embedded upgrade step

4.4 Modification of the step FSMs

A non-embedded upgrade step has a continuous sequential execution of all of its phases. In an embedded upgrade step, after the successful execution of one phase of a nested step, the nested step waits while its inner nested step completes its execution. This remains true during the normal execution, the rollbacks and the undoing of the entire embedded upgrade step. The existing step FSM models the execution of a non-embedded upgrade step only. Therefore, we modify the step FSM *i* so that it can model the execution of embedded step as well. The step FSM is modified so that while one of the nested steps of an embedded step is executing the other nested steps of the embedded step can go into states where they can wait. As there are *n* nested steps in an embedded upgrade step each step maintains its own state. Also each nested step must be aware of its immediate inner or outer nested step's state and the state transitions they make. Hence the communication between the nested steps is also required. Considering these requirements we made modification in the existing step FSM in such a way that it does not affect

the execution of the non-embedded upgrade step. Also the modifications in step FSM are done is such a way that the procedure FSM or the campaign FSM are not impacted. In the subsequent subsections we discuss these modifications.

4.4.1 Modified FSM of the upgrade step

As the execution of the innermost nested step of an embedded upgrade step is the same as of the non-embedded step therefore a non-embedded upgrade step is considered as an embedded upgrade step of a single nested step. A non-embedded upgrade step is an embedded upgrade step with a degree of embedding equal to one. Furthermore, in such an embedded upgrade step the outermost nested upgrade step is also the innermost nested upgrade step. Therefore, we do not change the way a non embedded upgrade step is executed. To present the modified FSM clearly we have split the step FSM into two figures: one shows the normal execution (Figure 4.3) and the other shows the rollback (Figure 4.4). In both of these figure there is a common state Completed for showing the connection between the two figures. The step FSM is modified to conform to the principle we described about the execution of the embedded upgrade step during the normal execution, the rollback and the undoing.

Each nested step follows the execution as modelled by the step FSM and each step is recognised by its position in the embedded upgrade step. Each step can communicate only with its immediate inner nested step and the immediate outer nested step. If there is no outer step (i.e. step is the outermost upgrade step) then the step *1*communicates with the procedure. Also the step *n* communicates only with the step *n-1*as there is no further inner step. With the modification in step FSM each step is now aware about the phase in which it is executing by using an attribute phase of type integer. Phase equal to '0' indicates that the nested step is in the tear-down phase and '1' indicates that the nested step is in the build-up phase.

In the subsequent subsection we explain the step FSM by taking an embedded step with degree of embedding *n*. We explain the meaning and the reason of the states and the transitions between the states by taking a step *i* ($1 \leq i \leq n$). As all the nested steps of an embedded step behave in the same way with a small difference in the innermost and the outermost step we explain the modified step FSM by describing the behaviour of the step *i* FSM. For the easy comprehension of the step *i* FSM we below mention some of the important points:

- The signal sent/received by the step *i* by the outer nested step or the inner nested step is shown as Step i-1: signal-message or Step i+1: signal-message, respectively.
- Proc or Step i-1: signal-message implies that if the step *i* is the outer most nested step it sends the signal message to the procedure to which it belongs else it sends the signal message to its immediate outer step.

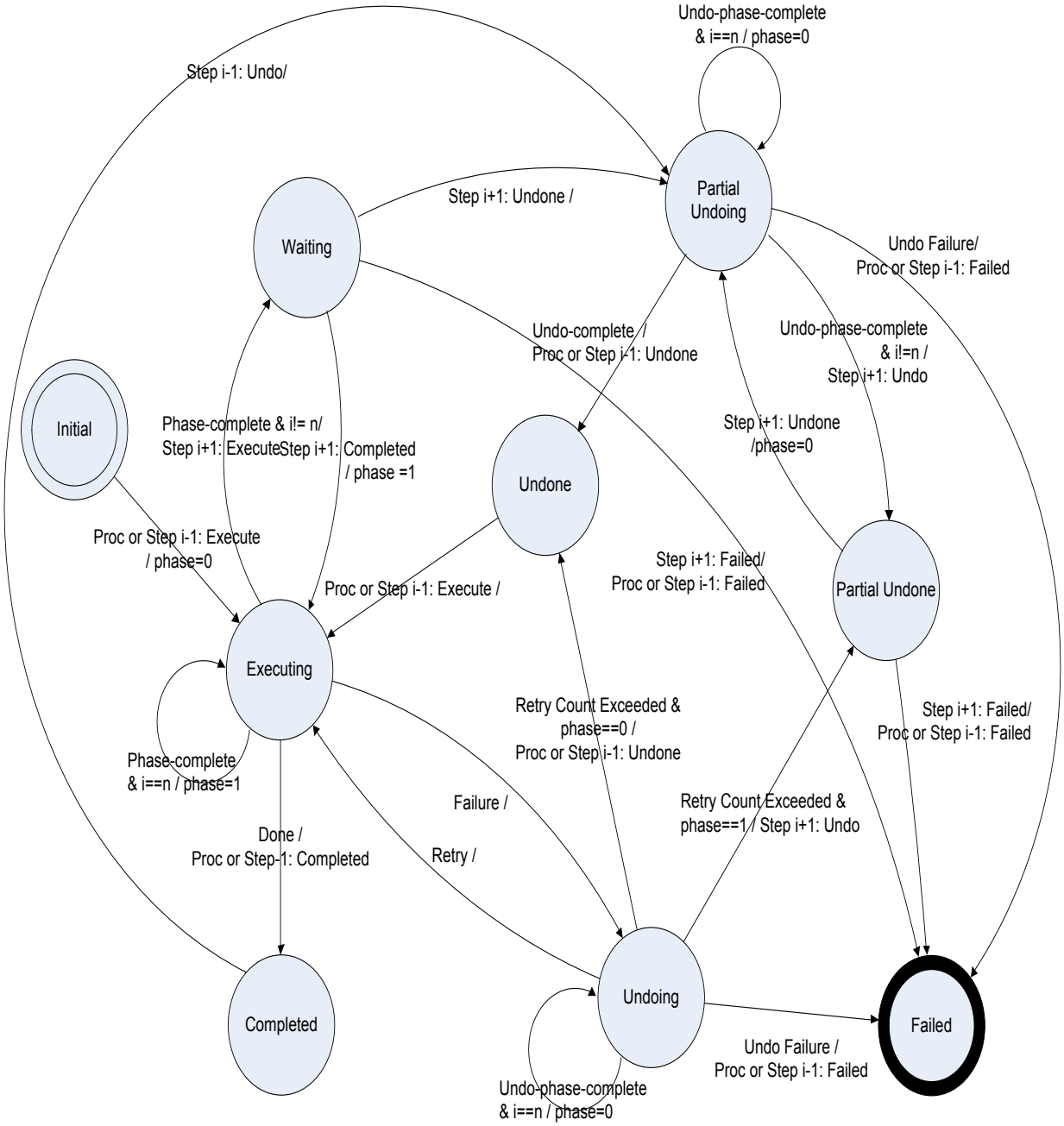


Figure 4.3 Modified upgrade step FSM

Initial state

The step starts at the Initial state. If the step *i* is the outermost nested step it will start executing when it receives the input signal Proc:Execute. If it is an inner nested step it will remain in the Initial state until the outer step (step *i-1*) has completed its tear-down phase. Therefore step *i* (i≠1) begins to execute when it receives the signal Step i-1: Execute. During the transition of step *i* from the Initial state to the Executing state, step *i*'s phase is set to 0 indicating that the step is in the tear-down phase.

Executing state

Depending on the phase in which a step *i* is, the corresponding actions from that phase are executed in this state. During the execution if any failure (Failure) is detected, the step *i* moves to the state Undoing. Once the phase in which a step is completed, the step can move either to the Waiting state, the Completed state or can remain in the Executing state. These transitions depend on the conditions explained below:

- If the step *i* has completed executing its tear-down phase (Phase-complete) and has no further nested step in it (*i*==*n*) then it continues executing the build-up phase but sets the phase to *1* indicating that the step *i* is in build-up phase.
- If the step *i* has completed its tear-down phase (Phase-complete) and has a nested step in it then the step *i* moves to the Waiting state so that the immediate inner nested step (step *i+1*) can start its execution. Therefore with this transition the step *i* sends the signal Step i+1: Execute.
- If the step *i* has completed its build-up phase (Done) the step *i* moves to the Completed state and send the signal Proc: Completed to its procedure FSM if *i=1* otherwise it sends

the signal Step i-1: Completed to its immediate outer step that it has completed the execution.

Undoing

In this state only the reversing actions of the phase in which the step *i* is currently are applied. This is in accordance with the undo concept described in the Section 4.2. If the step *i* does not have any further nested i.e. step (*i*==*n*) then the entire nested step *i* is reversed. Once the execution of actions at Undoing state is successful and retrying (Retry) is permitted then the step *i* moves to the Executing state.

Similar to the transition an upgrade step makes in undoing state when the retry count is exceeded, the embedded upgrade step must also move to the undone state. This is done by undoing all the phases that have been executed successfully so far starting from the most recently successfully executed phase and finishing by undoing the first successfully executed phase (the direction of execution of undoing of an embedded step is opposite of what is shown in the Figure 4.1).

If the actions of the step *i* are completely reversed then the step *i* moves to Undone state (as the phase = 0) and sends the signal - Step i-1:Undone to immediate outer step so that it can start undoing. If there is no outer step for step *i* then it sends the signal to the procedure - Proc: Undone indicating that the embedded step is undone.

In case where only the actions from the build-up phase (phase ==1) of the step *i* are reversed then the step *I* moves to the Partial undone state and sends the signal Step i+1: Undo to its immediate inner step implying that immediate inner step should start undoing itself.

While undoing the step *i* if a failure Undo Failure occurs then the step *i* moves to the Failed state. Since the step *i* has failed then entire embedded step has failed and this message must be corresponded to the procedure. Hence when the step *i* moves to the Failed state it sends the signal (Proc or Step i-1: Failed) to the outer step so that it can be escalated to the procedure.

Undone

This state remains the same as of the existing step in the standard SMF specification. Just like the Initial state, the outermost step can receive the signal Proc: Execute from the procedure due to an administrator attempt for a forced retry. With this signal from the procedure the embedded step will start executing. For all the other nested steps to start executing, their immediate outer step must send the Step i-1: Execute signal.

Waiting

When the step *i* finishes its tear-down phase it enters into this state so that its immediate inner step can start its execution. It remains in this state and waits until its immediate inner step has finished its execution completely and then step *i* can resume executing. For this transition the step *i* receives the signal Stepi+1:Completed from its immediate inner step. When the step *i* moves to Executing state it also sets its phase to 1 to indicate the execution of build-up phase.

From the waiting state it can also move either to the Partial Undoing state or to the Failed state.

If the entire embedded step has to be undone then the step *i* should also reverse the actions of its phase by entering into Partial Undoing state (refer Partial Undoing state for difference between Undoing and Partial Undoing). This happens only when its immediate inner step (step *i+1*) has completely undone (Undone) itself.

The step *i* can also move to the Failed state from the Waiting state if its immediate inner step has moved to Failed state (Signal received by step *i* Step i+1: Failed) and sends the message Proc or Step i-1: Failed. This transition is required because of one of the step it nests had a failure while undoing its actions.

Partial Undoing

If a step has completed undoing itself and cannot retry normal execution then all the nested steps of the embedded step should also undo themselves. Therefore following the order of undoing an embedded upgrade step (described in Section 4.2) the next nested step of the embedded step moves to the Partial Undoing state. This nested step could be either in Completed state or in the Waiting state. This state is similar to the Undoing state with the only exception that in the Partial Undoing state retrying is not permitted.

While undoing the step *i* in this state if there is a failure (Undo Failure) then the step *i* moves to the Failed state and sends the signal Proc or Step i-1: Failed so that the message Failed can be propagated to the procedure. This way all the outer steps also move to the Failed state.

Once the step *i* has completely undone itself (Undo-Complete - reversed the actions of the tear down phase) then the step *i* moves to the Undone state and sends the signal Step i-1 Undone to the outer step that it can start undoing itself. Once all the steps are undone the outermost step sends the signal Proc: Undone to the procedure indicating that the embedded upgrade step has been undone.

However, if only the build-up phase is undone (Undo-Phase-Complete) and the step *i* is not the innermost nested step ($i \neq n$) then the step *i* moves to the Partial Undone state indicating that only build-up phase has been undone and must wait for the other inner steps to completely undo before it can start undoing its tear-down phase.

Partial Undone

This is a waiting state which indicates that the step *i* has only undone its build-up phase and now must wait for the inner steps to undo themselves. Once its immediate inner step has completely undone itself (i.e. once *step: i+1* is in Undone state), step *i* receives the signal Step i+1: Undone and moves to Partial Undoing state (i.e. step *i* can start undoing itself in the tear-down phase), and sets the phase to 0.

The step *i* can also move to the Failed state from the Partial Undone state on receiving the signal Step i+1: Failed from its immediate inner step. This transition is required because one of the inner steps has failed and the message has to be propagated to the procedure.

Completed state

This state indicates that the step *i* has completed its execution on both the tear-down phase and build-up phase. If the entire embedded step has to be undone then the nested steps which are in Completed state also needs to be undone. In this case step *i* on receiving the signal Step i-1: Undo from the immediate outer step moves to the Partial Undoing state. If the administrator has decided to roll back the upgrade campaign then the step *i* moves to the Rolling Back state on receiving the signal Rollback. During this transition it resets the phase to 0.

Failed state

This is a final state for the step *i* when a double failure has happened and the executed actions cannot be reverted. This state is the same as in the step of the standard SMF specification.

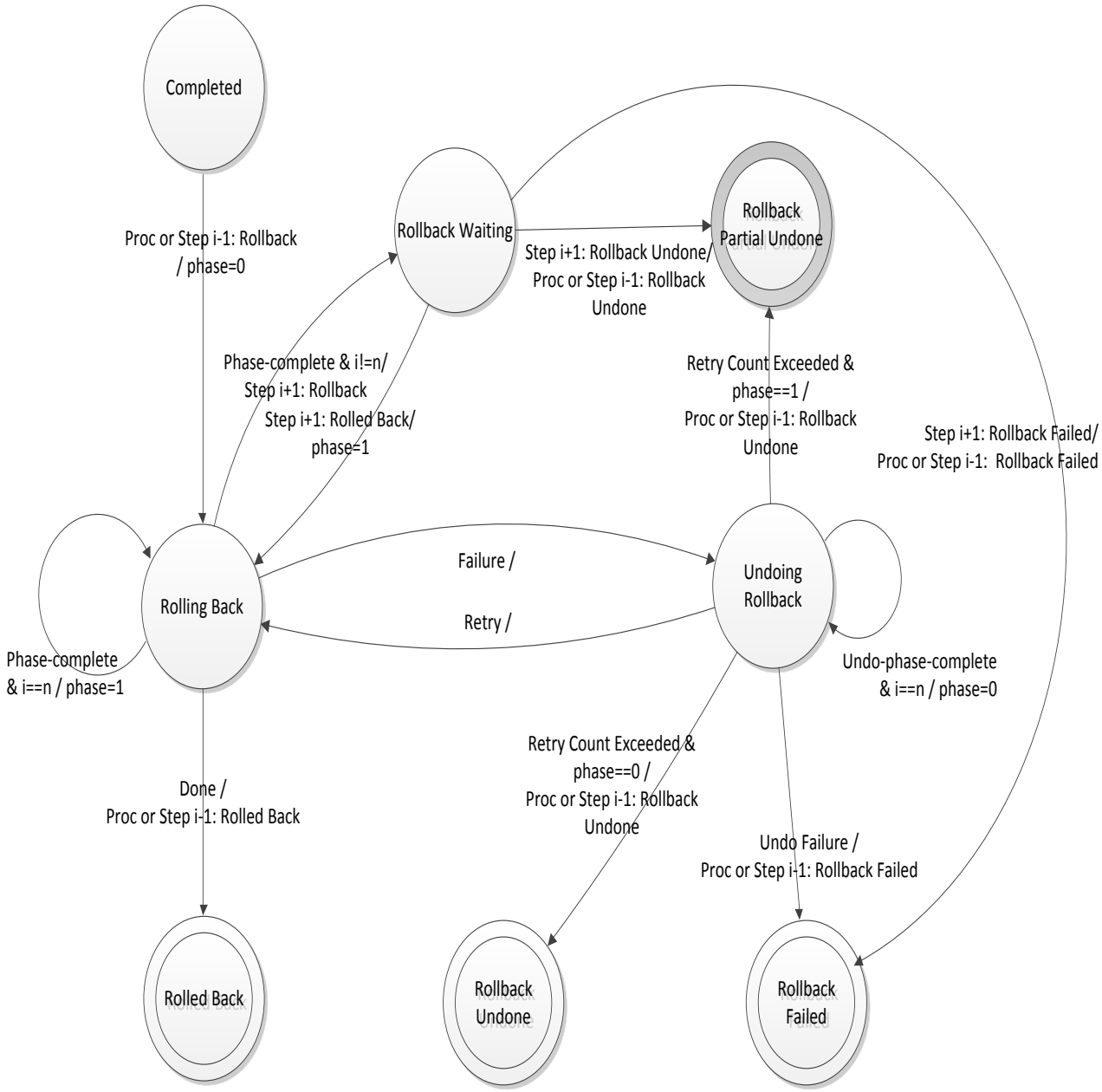


Figure 4.4 Modified step FSM for the rollback of upgrade step

Rolling Back

Rolling back of an embedded upgrade step starts if the administrator decides to rollback the upgrade campaign. Only the upgrade steps that are in Completed state can be rolled back as the

upgrade steps which are in Undone state have already reverted to the old configuration. The outermost upgrade step present in Completed state receives a signal from the procedure it belongs Proc: Rollback. This signal puts the outermost upgrade step in the Rolling back state. The rolling back of a nested step (step *i*) is similar to the normal execution of a nested step. A nested step once finishes its execution in tear-down phase suspend its execution (if it is not the innermost step) and moves to the Rollback Waiting state. During this transition the nested step sends the signal Step i+1: Rollback so that the inner step can start rolling back. If there is a failure (Failure) during the rolling back, the step *i* moves to the Undoing Rollback state where the actions corresponding to the phase in which error has occurred are reversed. Once the build-up phase of the rollback of the step *i* is complete (Done), the step *i* moves to the Rolled Back state and sends the signal to its immediate outer step (Step i-1: Rolled Back) to resume its execution for the rollback. If there is no immediate outer step it sends the signal (Proc: Rolled Back)to the procedure indicating that rollback of the embedded step is complete.

Rollback Waiting

When the step *i* suspends its execution during the rollback (Phase-Complete) it enters into the Rollback Waiting state. It is a waiting state in which the step just waits until its immediate inner step finishes its execution of rollback successfully. Once its immediate inner step moves to Rolled Back state, the step *i* moves from the Rollback Waiting to the Rolling Back state on receiving the signal Step i+1: Rolled Back where it continues rolling back its build-up phase (installation of old software). During this transition it sets its phase to 1 indicating that it is now going to roll back in the build-up phase.

The step *i* can also move to the Rollback Failed state from the Rollback Waiting state on receiving the signal Step i+1: Rollback Failed from its immediate inner step. This transition is the result of one of its

inner nested step failing while undoing its rollback. During this transition the step *i* attempts to send the Rollback Failed signal to the procedure by escalating it through the outer upgrade step (Proc: or Step i-1: Rollback Failed).

Rolled Back

This state signifies that the rollback of the embedded upgrade step has been completed successfully. It is a final state.

Undoing Rollback

Similar to the Undoing state of the normal execution, in this state as well the reversing actions of the phase in which the rollback of step *i* is currently are applied. If it is an innermost nested upgrade step then all the reversing actions of the rollback of step *i* are applied before retrying (Retry).

If retrying limit has exceeded then the step *i* moves to the Rollback Partial Undone or the Rollback Undone depending on if the step *i* has completely undone itself or undone only the build-up phase. During this transition it sends the signal rollback undone to its procedure if it is the outermost nested step (Proc: Rollback Undone) otherwise to its immediate outer step (Step i-1: Rollback Undone).

While executing the actions in the Undoing Rollback if there is any failure (Undo Failure) the step *i* moves to the Rollback Failed state and sends the signal rollback failed (Proc: or Step i-1: Rollback Failed) either directly to the procedure or by escalating it through outer steps.

Rollback Partial Undone, Rollback Undone

After successfully reversing the required actions of step *i* in the Undoing Rollback state if the retry count exceeded, the step *i* moves to the Rollback Undone State or the Rollback Partial Undone state depending on whether all the phases are undone or only the build-up phase is undone. Exceeding

the retry count is considered as a failure as the number of attempts required to fix the problem by retrying has exhausted. Therefore the message that rollback is undone must be propagated to the administrator as soon as possible. This message is escalated as the only remedy to bring the system out of the failure is fall-back. Therefore all the outer steps of the step *i* waiting in the Rollback Waiting state do not attempt to completely undo them self as it will lead to bringing the new software in the system but they rather move to the Rollback Partial Undone state by sending the signal Proc or Step i-1: Rollback Undone.

This way procedure to which the embedded step belongs comes to knows quickly that the rollback has been undone. The procedure treats the rollback undone as a failure and sends the failure message to campaign so that recovery action of restoring the system with the backup image can be taken.

Rollback Failed

If there is a failure while undoing during the rollback then we have lost the state in which the step *i* is, as it is not known which all actions were reversed. Therefore the step *i* moves to its final state Rollback Failed. If there are other nested steps (outer steps) that were waiting in the Rollback Waiting state then they also move to the Rollback Failed state. During this transition the signal sent by the step *i* is Proc or Step i-1: Rollback Failed.

4.5 Closing Remarks

In this chapter we presented the new upgrade steps which overcome the limitations of the existing upgrade steps and also solve the complexity faced in the layered upgrade steps. With the examples in this chapter we also explained how using these new upgrade steps along with the rolling upgrade step will not affect the service availability. Modifications have been brought to

SMF for the execution of the upgrade step. This execution is modelled by the modified FSM. We also made the changes in the Upgrade Campaign Schema to reflect the new upgrade steps and the degree of embedding. Before we describe the process to generate an upgrade campaign we discuss the complexities in the upgrade of virtualization facilities in the next chapter. In that chapter we also describe an approach to upgrade the same without impacting the service availability. This approach is used with the new and the old upgrade steps to generate an upgrade campaign which is discussed in the Chapter 6.

Chapter 5 - Upgrade in case of virtualization

Our next step is to devise a method to generate automatically the upgrade campaign for a system including the lower layers. However, before that we discuss the complexity associated with the live upgrade of migration enabled VMs. This complexity could not be addressed at the step level because the VMs are not tied to one particular VMM but could be present on any of their sponsoring VMM in the system. This sponsoring VMM cannot be identified at the time an upgrade campaign is designed. We discuss this complexity in Section 5.1. We introduce the assumptions we make to limit the scope of the problem in Section 5.2. In Section 5.3 we describe the strategy for the live upgrade of VMs capable of live migration. To make this approach more comprehensive we have limited in this chapter the upgrade of system to only VMs and VMMs. The next chapter integrates this approach with the upgrade of entire system taking host OS into account. We end the chapter with closing remarks.

5.1 Complexities in the live upgrade of VM

Live migration of VMs increases the complexities in lower layers upgrade. Below we discuss and explain this complexity.

Difficulty in maintain the compatibility between a VM and its sponsoring VMMs during the upgrade

When a VM (say VM1) has a set of sponsoring VMMs (set S1 consisting VMMs $\{VMM_1..VMM_i,..VMM_n\}$), the VM (VM1) can live migrate from its current host VMM to another sponsoring VMM. The knowledge of the host VMM for the VM (VM1) when the upgrade step is applied on the VM (VM1) by the SMF is not known to the upgrade campaign designer. Even if the sponsoring VMM for the VM (VM1) is known, when the lock is applied by

an upgrade step on the VMM the VM (VM1) will migrate to another sponsoring VMM. This brings the complexity in maintaining the compatibility between the VM and VMM when the old version of VM is not compatible with the new version of VMM and vice versa.. In this case we cannot directly use the existing upgrade steps or the new upgrade steps for the lower layer as discussed in the Chapter 3 and the Chapter 4. The problem faced using these steps directly are described below with an example:

The first option is to start with the upgrade of the VM (in this example VM1). Using any of the non-embedded upgrade steps on the VM will make the VM incompatible to all of its sponsoring VMM.

The second option is to start with the upgrade of the VMMs. Let us say a sub-set of S1 has already been upgraded and right now VMM_i (VMM_i element of S1) hosting the VM (VM1) is being upgraded. When VMM_i is locked or is restarting, the VM (VM1) hosted on it will migrate to the other sponsoring VMM which could have already been upgraded to an incompatible type. Hence this approach will bring the incompatibility in the system.

The third option is to use the embedded step consisting of two nested steps. Embedded step is designed to solve the layered upgrade complexity however neither the upgrade campaign designer nor the SMF is aware about the sponsoring VMM of a VM at the time upgrade procedure to upgrade the VMM and VM is executed. Therefore, if the embedded step is designed with the outermost step targeting the upgrade of VM (VM1, in this example) and the innermost step targeting the upgrade of VMM (say VMM_i and VMM_i element of S1), then it is not guaranteed that when the upgrade procedure using this embedded step is executed, the VM (VM1) is being sponsored by VMM_i. Hence, the outermost step will upgrade the VM (VM1) but

the innermost step may upgrade another VMM (VMM_k, i != k and VMM_k element of S1). Therefore, the incompatibility between the VM (VM1) and the VMM (VMM_k) is still not resolved.

5.2 Assumptions

In this section we introduce the assumptions we make and the cases we do not handle to limit the scope of the problem.

5.2.1 Assumptions

1. Addition or Removal of a VM does not affect any other VM present in the cluster. Hence the entities of an application hosted on the other VMs also remain unaffected.

2. The permanent storage for a migration enabled VM is provided through the shared storage and is accessible by all the existing sponsoring VMMs and the new sponsoring VMMs for the VM.

3. Service providers are hosted on EEs in such a way that the active and the standby service provider for a SI are never on the same HE. This way we are sure that when an EE hosted directly on a HE is locked and terminated, active and standby service providers for an SI do not become unavailable at the same time.

4. To maintain the service availability we must first perform addition of all the EEs (explained in details in Section 5.3.3).Therefore we assume that the addition of EEs can be performed on old hosting EEs.

5. A VM will not be modified to a type such that it becomes incompatible to all of its current sponsors (in the source configuration) and hence new compatible sponsors will be

provided in the target configuration. The reason for this assumption is: if a VM has to be modified in the above example then usually a new VM with a new DN (or new identity) is brought into the system with the new type and the old VM with old DN (or old identity) is removed from the system.

5.3 Strategy for upgrading VMs

The upgrade of a system can imply addition, modification, removal operations on various entities of the system. In this section we explain how a system consisting of VMs capable of migration and VMMs are upgraded without the loss of the services provided by an application. For this we take an example configuration (described in the Section 5.3.1) which we will be using throughout the sub-sections for making the approach more comprehensive.

Our approach consists (shown in Figure 5.1) of three phases carried out in sequence:

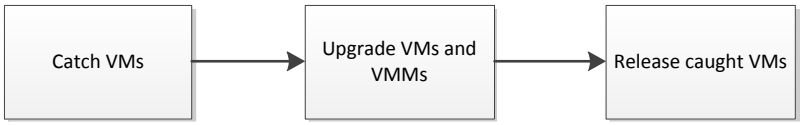


Figure 5.1 Overview of approach

1) In the first phase we catch (demobilize) the VMs (described in the Section 5.3.2) to counter the complexity discussed in the Section 5.1.

2) In the second phase upgrade is carried out by scheduling the execution of upgrade procedures in such a way that there is no loss of service.

3) In the last phase we wrap up the upgrade campaign by releasing the caught VMs (mobilize the VMs) that are present in the target configuration.

5.3.1 Example configuration for the upgrade

Consider the example shown in the Figure 5.2 with the initial system configuration consisting of five VMs which contain OSs and three VMMs. VM-R1, VM-R2, VM-M1 and VM-M2 are sponsored from VMM-R1 and VMM-M3. VM-M3 is sponsored from VMM-M4. Application services provided by the system are configured in such a way that for all the service providers in the system there is a service provider present on VM-M3.

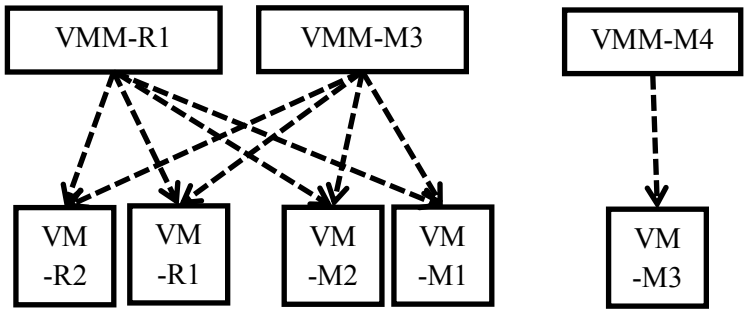


Figure 5.2 Initial system's configuration

The target system configuration (shown in Figure 5.3) removes VM-R1, VM-R2 and VMM-R1; adds VM-A1, VM-A2 and VMM-A2; and modifies the type of VM-M1, VM-M2, VM-M3 and VMM-M3 and VMM-M4. VM-A1, VM-A2, VM-M2 and VM-M1 can perform live migration on either VMM-A2 or VMM-M3. Application services follow the same characteristics as described in the source configuration.

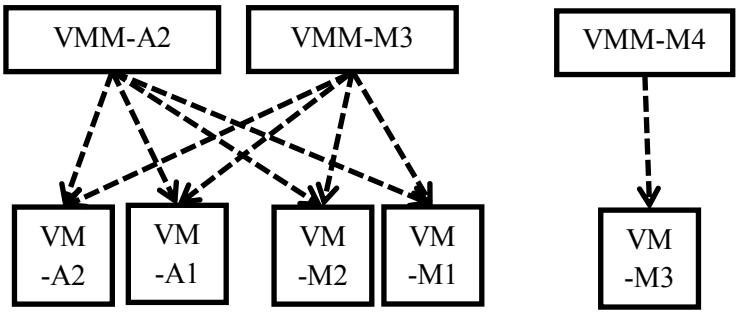


Figure 5.3 Target system's configuration

5.3.2 Phase of Catching the VMs

This is the first phase for the upgrade of the VMs and VMMs in which we modify the migration dependency of a VM to be upgraded to only one of its sponsoring VMM present in the source or the target configuration. This way we demobilize VMs so that when the upgrade campaign is designed the upgrade campaign designer knows the hosting VMM for each VM. Therefore, the embedded upgrade step is used to upgrade VMM and VM. This way we overcome the difficulty in upgrading VMs as described in Section 5.1.

This process is applied in the beginning of the upgrade campaign for the VMs that are either going to be removed or modified. As the VMs that are to be added are not yet present in the system they are not caught in the beginning. However, when these VMs are added by an upgrade procedure they are added on only one VMM, i.e. they are caught on to only one sponsoring VMM.

A VM that will be removed or added can be caught on any of its sponsoring VMM. However, a VM that will be modified cannot be caught on the VMM that will be removed or added because modification of VM requires presence of lower layer during the tear down and build up phase. A detailed flowchart of this process is presented in the section 6.4.4.

To upgrade the system configuration described in Section 5.3.1 we first catch the VMs capable of live migration that are being removed or modified. VMs that are being modified (VM-M1, VM-M2) can be caught only on a sponsoring VMM which is either being modified or undergoing no upgrade at all. In this example we catch these VMs on VMM-M3. VMs that will be removed (VM-R1, VM-R2) are caught on to the VMM that will be removed (VMM-R1).Catching the VMs has transformed the source system configuration to an intermediate system configuration shown in the Figure 5.4.

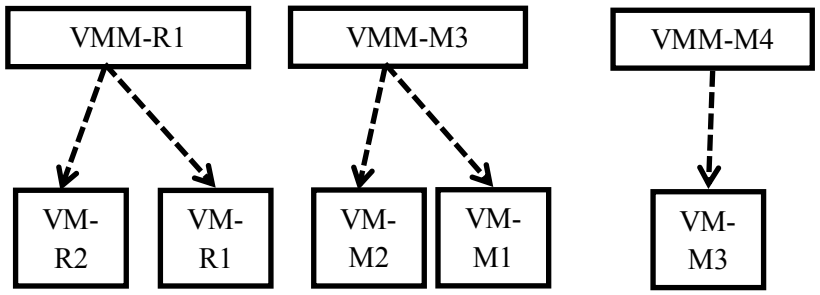


Figure 5.4 System's configuration after catching VMs

5.3.3 Upgrade Procedures Phase

In this phase the upgrade procedures are applied to upgrade the software entities to the target versions. As we mentioned in our assumption 3, the service providers are distributed in such a way that the active and the standby assignment for an SI will never land on the same HE at the same time, therefore we create upgrade procedures in such a way that at a time it puts at most only those service providers out of service which are hosted on the same HE. This way we will be sure that there is a redundant service provider available. Also, upgrade procedures are scheduled in way to avoid service outage. For example if all the EEs hosting SUs which protect a service (say S1) are being removed and the new set of EEs hosting SUs protecting the service

(S1) are being added then addition of service providers must happen before the old service providers are removed. Also, if there is one upgrade procedure which removes all the software running on one HE, its execution will not cause service outage because of the redundancy. However, execution of the next upgrade procedure in which even one service provider is being removed can cause service outage as it might be the only redundant service provider left. Therefore, the upgrade procedures that add the EEs are scheduled to be executed at the beginning so that the availability of services is maintained when the upgrade procedures that remove the EEs are executed. Similarly, the last set of upgrade procedure executing should be the one that only removes EEs.

The EEs that are undergoing modification may also add new service providers. Therefore after the addition upgrade procedure for EEs, the next set of upgrade procedures to be executing are the ones that modify at least one EE. As these upgrade procedures are modifying EEs, they must not be executed in sequence. Note that the upgrade procedure that modifies an EE may also be removing EEs but as an upgrade procedure can at most act on entities present on one HE; therefore, the service availability is guaranteed because of the redundant service provider present on the other HE.

Once the VMs are caught (shown in Figure 5.4), we first add VM-A1, VM-A2 and VMM-A2 (shown in Figure 5.5). The added VMs are not allowed to migrate on their other sponsoring VMMs (VMM-M3). To modify the EEs two upgrade procedures are created, one for the entities which include VMM-M3, VM-M2 and VM-M1; and another for the VMM-M4 and VM-M3. These upgrade procedures are executed in sequence. The last upgrade procedure removes VMM-R1, VM-R2 and VM-R1. The system configuration obtained after executing the modifying and the removal upgrade procedure is shown in Figure 5.6.

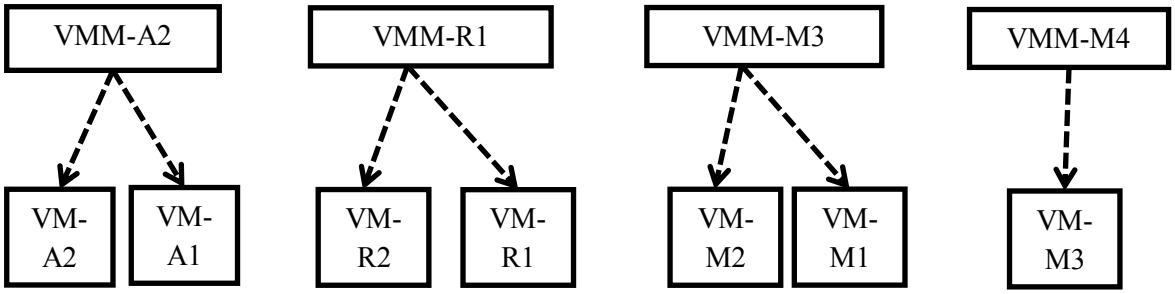


Figure 5.5 System configuration after execution of addition upgrade procedure

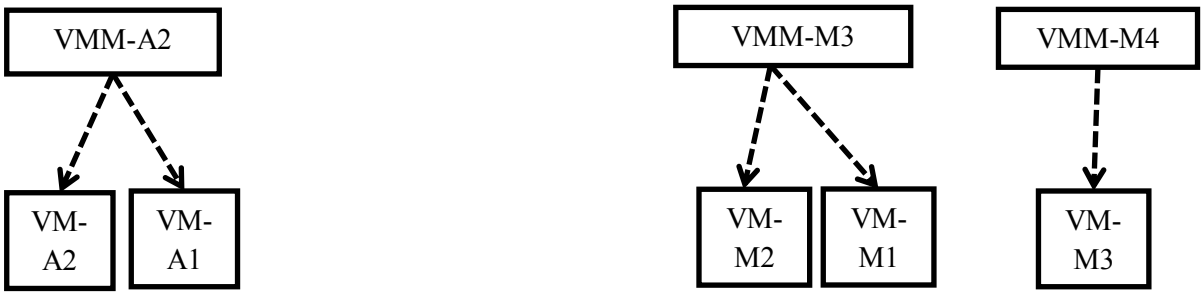


Figure 5.6 System configuration after execution of all upgrade procedures

5.3.4 Wrap-up of upgrade campaign phase

As a result of the catching process in the first phase and during the addition in the second phase, the configuration of the system obtained so far is not the target configuration. Therefore, in this phase we re-enable the migration of VMs according to the target configuration. In the example, we enable the dependency of VM-A1, VM-A2, VM-M1 and VM-M2, to VMM-M3 and VMM-A2 and hence achieve the target configuration shown in Figure 5.3 Target system's configuration.

5.4 Closing Remarks

In this chapter we discussed the strategy for upgrading VMs and VMMs where the HA of application services is paramount. The strategy covers the initial act of restricting the

complexities to few VMMs and then creating and scheduling the execution of upgrade procedures.

In the next chapter we describe how an upgrade campaign is generated containing a roadmap to upgrade the entire system consisting of AMF and PLM entities. The strategy described in this chapter to upgrade VMs and VMMs will be taken into account in the generated upgrade campaign.

Chapter 6 - Upgrade Campaign Generation

The work described in this chapter aims to expand the scope of upgrade campaign generation to support PLM entities as well. We describe a method for the generation of an upgrade campaign for an entire SA Forum managed system. This generated upgrade campaign handle the upgrade of the AMF and the PLM entities including the special case of virtualization as explained in the previous chapter (Chapter 5).

In this chapter we provide an overall view of the entire approach for the generation of the upgrade campaign (see Section 6.1), the Assumptions made for the upgrade of the entire system (see Section 6.2), input required (see Section 6.3), Pre-processing of the input (see Section 6.4) and finally the generation of upgrade campaign (see Section 6.5).We conclude the chapter in Section 6.6.

6.1 Overall view of the approach for upgrade campaign generation

A complete overview of the approach is shown in the Figure 6.1 and is explained in details in the subsequent section. In the beginning user provides the input which passes through two phases (discussed below) to produce an upgrade campaign xml file. These two phases are:

1) Pre-processing of the input

   - In this phase input is represented in the tree structure, where each node of the tree represents an entity from PLM model. Each node is also annotated with the information required to upgrade the entity represented by that node e.g. upgrade procedure, restartablity, sponsoring VMM for a VM after applying the process of Catching VMs, etc.

2) Generating the upgrade campaign

- In this phase, we generate the upgrade campaign by visiting the trees created in the previous phase which contain all the information required for the upgrade of entities.

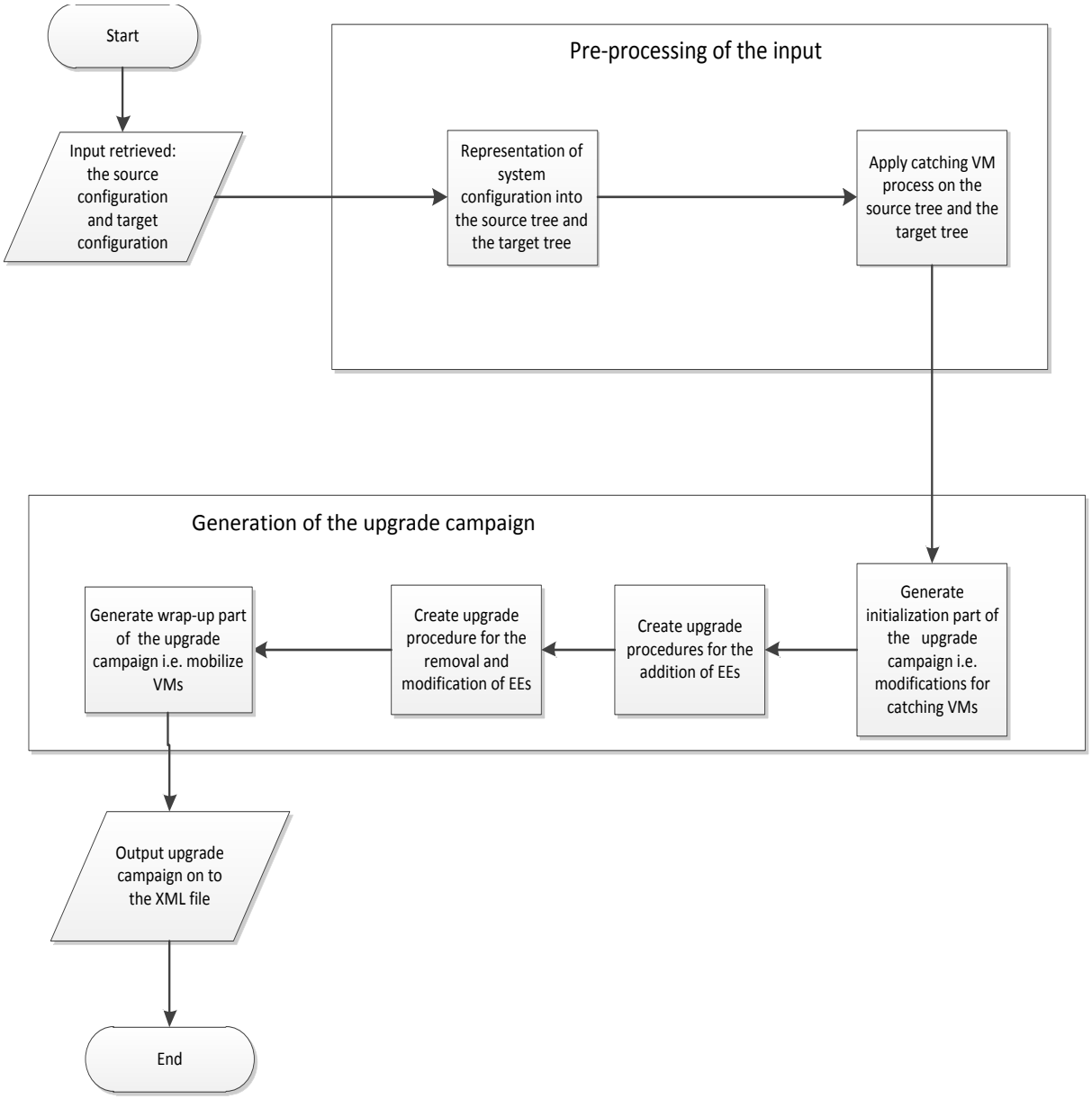


Figure 6.1Overall view of the approach to create upgrade campaign

6.2 Assumptions for the generation of an upgrade campaign

In this section we discuss some assumptions we make for the generation of an upgrade campaign. This list of assumptions adds to the assumptions introduced in Chapter 5.

1) A *SaPlmEE* object is considered to be a VM capable of live migration if it has a dependency to other *SaPlmEE* objects.

2) All of the *SaPlmEE* objects that are sponsoring VMs are considered to be VMM.

3) The dependency between a VM and VMMs represented by the object of class SaPlmDependency is always considered to be the migration dependency.

4) The object of class *SaPlmDomain* remains unchanged in the source and the target configuration.

5) In addition to the third assumption at section 5.2.1 of the chapter 5, we assume that the input provided by the user for the upgrade is a valid configuration.

6) Software bundle for the installation of EE comes with the application level software.

7) The SA Forum managed system undergoing upgrade can have at most three layers of EEs: 1) Host OSs 2) VMMs 3) VMs including guest OS which is represented as one EE .

6.3 Input required for the generation of an upgrade campaign

The integration of the AMF, the CLM and the PLM models into one model is referred as the system model. Based on the system model we take the following input:

Source system model

It provides the current system model configuration.

Target system model

It provides the target system. To evaluate the entities that are being added, removed, modified or remaining unchanged, the entities present in the source and the target configuration are compared

based on the DN of each object. Therefore, target system configuration must be created from the source system configuration by adding, removing and/or modifying entities in that model.

6.4 Pre-processing of the input

In this phase we retrieve the necessary upgrade related information for the PLM entities from the user input; and put it in a tree structure (described in the subsequent subsection). This phase is further divided in two sub-phases 1) Represent the PLM entities from the source system configuration and the target system configuration using two trees: source tree and the target tree 2) Applying catching VMs process on the trees.

6.4.1 Representation of system configuration as a tree

To upgrade an entity we must have the following parameters: 1) whether an entity is undergoing any addition, removal, modification or no change at all 2) In case of VMs capable of live migration and undergoing upgrade we need to know on which VMM they are caught. We retrieve all this information by evaluating the source and the target configuration and through the catching VMs process (discussed in Chapter 5 - Section 5.3.2) and put this information into a tree structure for manipulation. The nodes of the tree are the objects of class Node (see Table 6.1). The attributes of Node represent all of the above mentioned parameters for each PLM entity. Next we discuss how these attributes are set.

The entities of concern for constructing the trees from the source/target system configuration are EEs which can be direct or indirect children of the object of the class *SaPlmDomain*. In case there are HEs in a given configuration we ignore them as we are not using them and view the EEs hosted on the HEs as direct child of the object of class *SaPlmDomain*. Therefore, each instance of the Node (also referred to as node from now on) represents either an EE or the object

of the *SaPlmDomain*. If a node represents an EE then the attribute ***anEE*** and ***dependency***is set otherwise the attribute ***theDomain*** is set.

All the nodes from a given configuration are organized in a tree structure where children of each node are set in the attribute ***childrenNode.*** From the source configuration and the target configuration we get two trees: the source tree and the target tree. These trees are organized such that the parent-child relationship between the two nodes in a tree represent either a containment relationship between two EEs (e.g. when an EE such as VMM contains another EE such as VM then the former will be the parent node and the later will be the child node) or an aggregation relationship between an EE and the object of class *SaPlmDomain*(e.g. all the EEs such as HostOS or VMs capable of live migration are child object of the object of class *SaPlmDomain,* therefore the node which represents the former is the child node and the node which represents later is the parent node). These trees will be used to generate the upgrade campaign. Below we provide the source tree (Figure 6.2) and the target tree (Figure 6.3) constructed using the sample configuration discussed in Chapter 5.In the sample configuration we represent VM and the Guest OS installed on it as one EE. The dashed lines shown in the tree are used only for the illustration purpose to represent the dependency between the two EEs; the nodes of the tree are connected only by the solid lines which represent containment or the aggregation relation. As all the EEs are part of the object of class *SaPlmDomain* (safDomain in the example), therefore it becomes the root of the tree. In the subsequent subsection we describe how each of the attribute for a node is set.

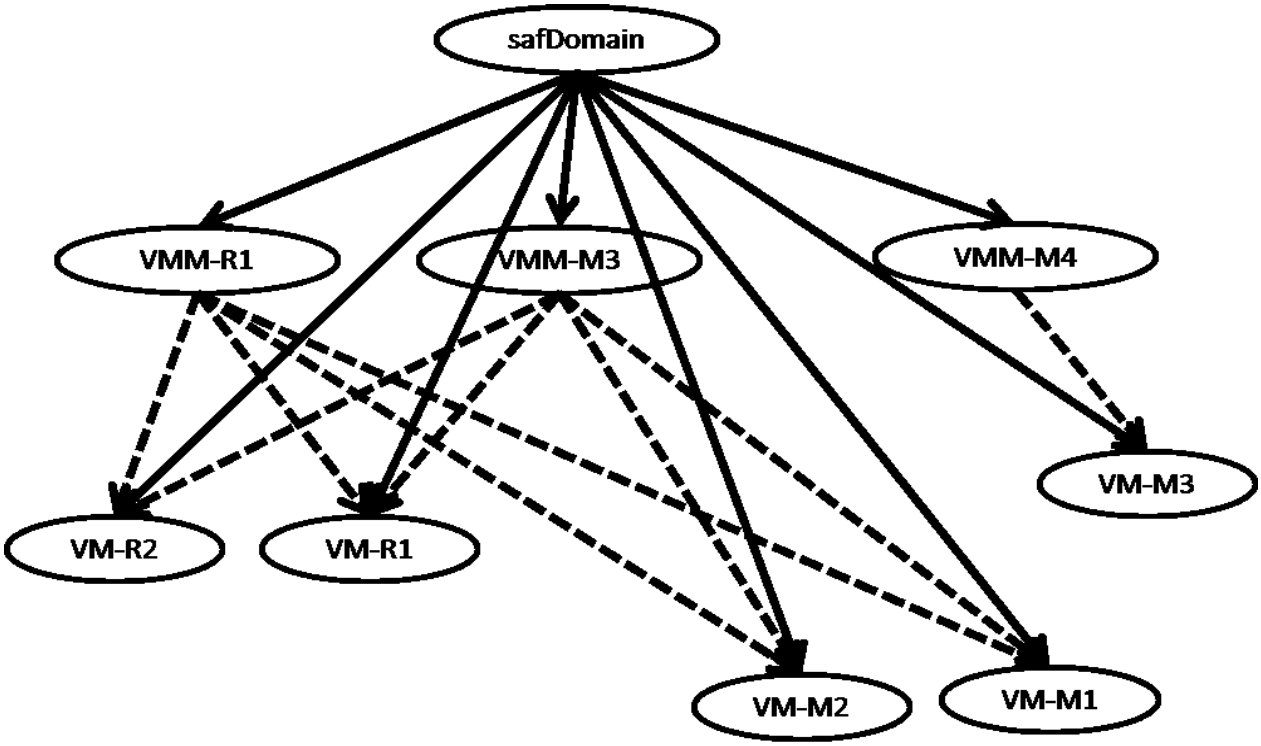


Figure 6.2 Source tree

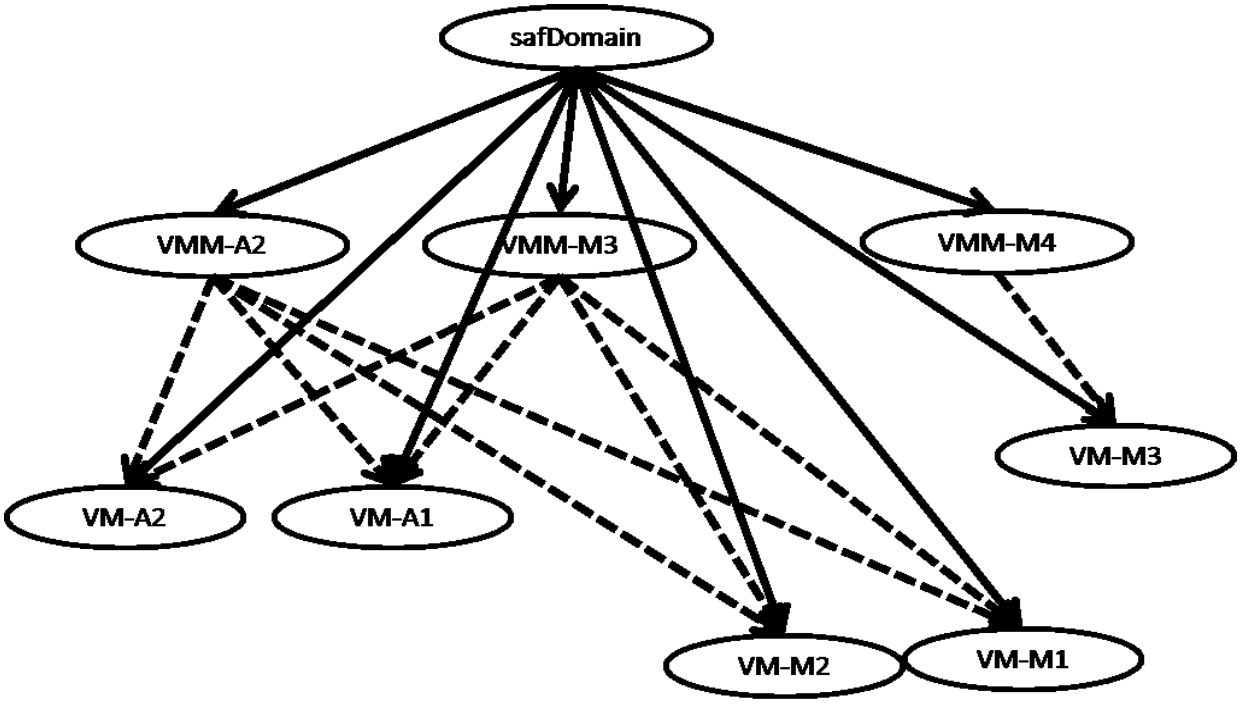


Figure 6.3 Target tree

Class Node	
{	
SaPlmEE	anEE; // attribute that represent the entity of type SaPlmEE from the source // and/or target configuration
SaPlmDomain	theDomain; // attribute that represent the entity of type SaPlmDomain from // the source and or target configuration
SaPlmDependency	dependency ; // the dependency object of the EE
String	operation; //suggest what kind of upgrade operation needs to be applied on //the entity represented by the node
SaSmfProcedure	procedure; //identifies the upgrade procedure that will upgrade the entity //represented by the node
Boolean	processed= FALSE; // used for checking if the node has been processed before
ArrayList<Node>	childrenNode; // represent the children of a node
}	

Table 6.1 Class Node

6.4.2 Determining operation type ADD/ REMOVE/ MODIFY/NO_OPERATION

In order to identify the type of the operation on a node, objects of the class *SaPlmEE* from the source and the target PLM configuration are compared with each other based on the DN and is set according to the following rule:

1) If an object is found in the source configuration and is not available in the target configuration then the attribute ***operation*** is set to REMOVAL.

2) If an object is found in the target configuration and is not available in the source configuration then the attribute ***operation*** of the corresponding node in the target tree is set to ADDITION.

3) If an object is found in the source configuration and in the target configuration then the value of all the attributes of the object in the source and the target configuration are compared

    a. If the value of attributes in the source and the target configuration are same then the ***operation*** of the corresponding nodes in the source tree and the target tree is set to NO_OPERATION.

b. If there is any modification then the attribute ***operation*** of the nodes that represent this object in the source tree and the target tree is set to MODIFY.

6.4.3 Setting remaining attributes of a node

The remaining attributes- ***procedure*** and ***processed*** are set when we are generating the upgrade campaign. The attribute ***procedure*** specifies the upgrade procedure used to upgrade the entity represented by that node. It serves as a unique identifier for all the entities that are meant to be upgraded by the same upgrade procedure. The attribute ***processed*** indicates if the node has already been processed by the upgrade campaign generator.

6.4.4 Catching VM nodes of the tree

In this sub-phase we put another piece of information in the trees that will answer on which VMM a VM is caught. For this we apply the catching VMs process on each node of the source and the target tree that represent VMs capable of live migration and undergoing upgrade. VMs that will be modified are caught on the same node in both the source and the target tree. As the catching process makes sure that a VM will be contained in one VMM only, therefore we represent this information in the tree. This is done by detaching the node that represents the VM from the root node and attaching it to a node that represents the sponsoring VMM determined by the catching VMs method.

Figure 6.4 shows a flowchart that describes how the trees (the source tree and the target tree) are modified. The flowchart follows the process described in Chapter 5. Figure 6.5 and Figure 6.6 show the result of execution of this flowchart on the source tree and the target tree, respectively. We used the following conventions in these flowcharts:

- All the words in bold and italics are variables and have the scope of the flowchart.

- ‘AND’ is a Boolean operator while ‘and’ is a simple English and.
- “OR’ is a Boolean operator.
- “Get all the nodes”:This instruction will get all the nodes from a tree with the constraints mentioned in the complete instruction. “Get the node” instruction is preceded by “Get all the nodes” instruction. This instruction will fetch one node that has not been fetched before from the set of nodes obtained by the instruction “Get all the nodes”.
- A node ‘x’ from source tree is said to correspond to node ‘y’ from target tree if the entity represented by them is the same i.e. ***anEE*** or ***theDomain*** attribute has the same value in ‘x’ and ‘y’.
- If a node of a tree has to be given an alias, it is done by getting the node and then using the keyword ‘as’ then alias name.
- Example: get the node as ***current.*** This way node that has been retrieved can be referred as ***current***.

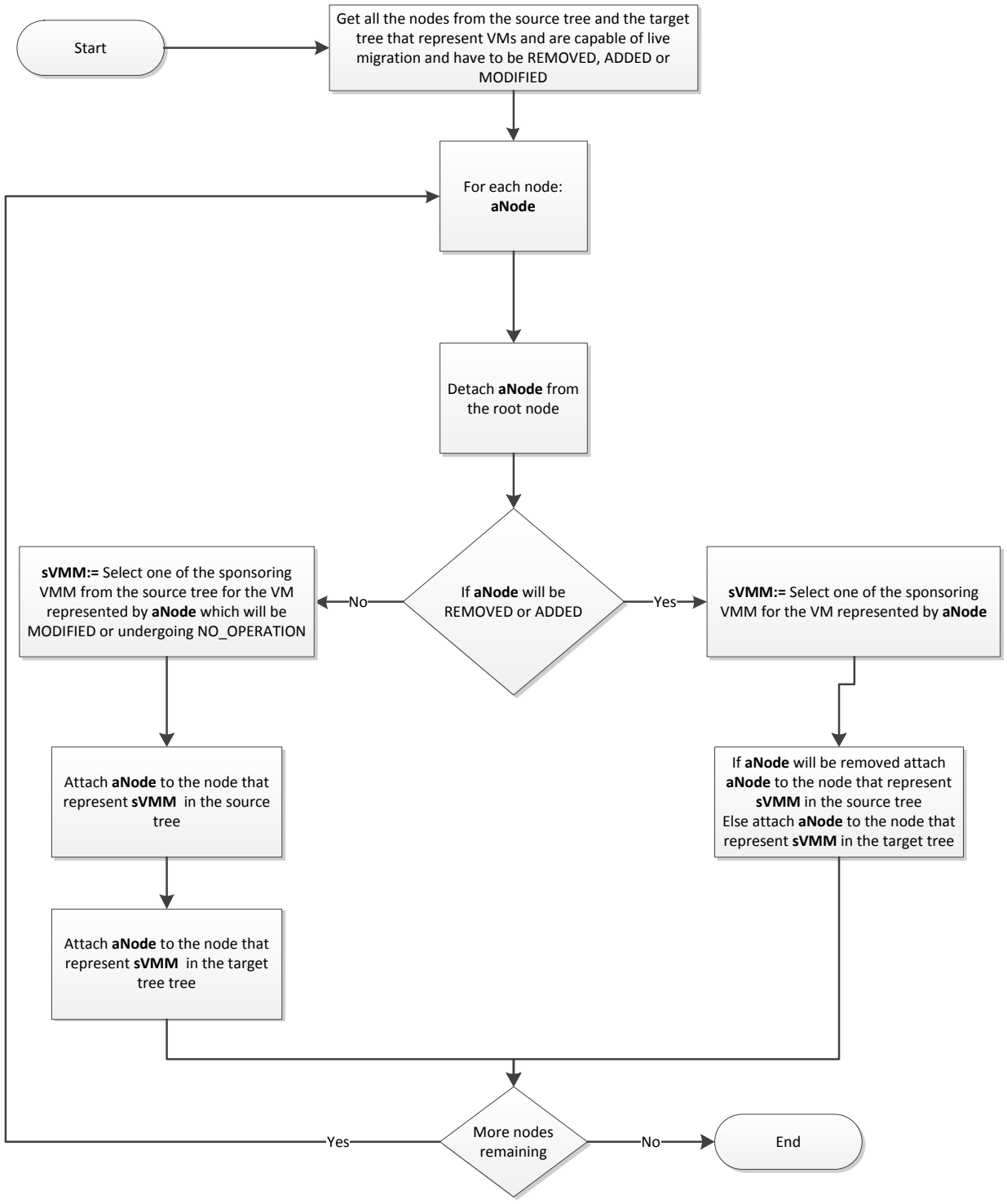


Figure 6.4 Flowchart for catching VMs

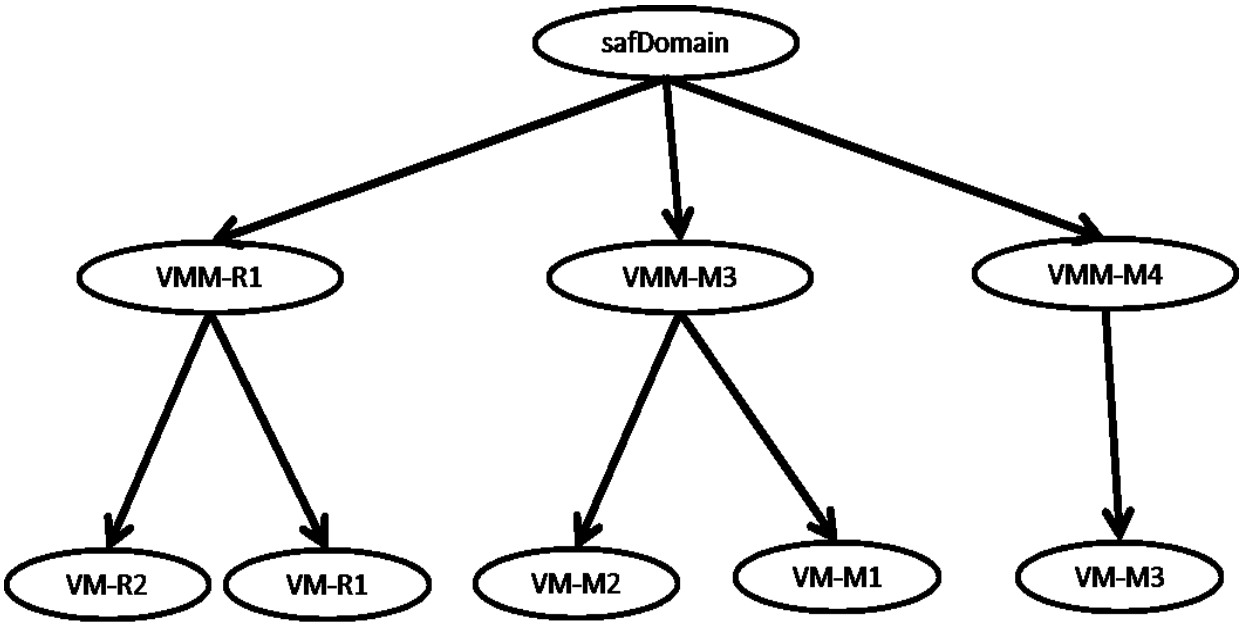


Figure 6.5 Source tree after catching VMs

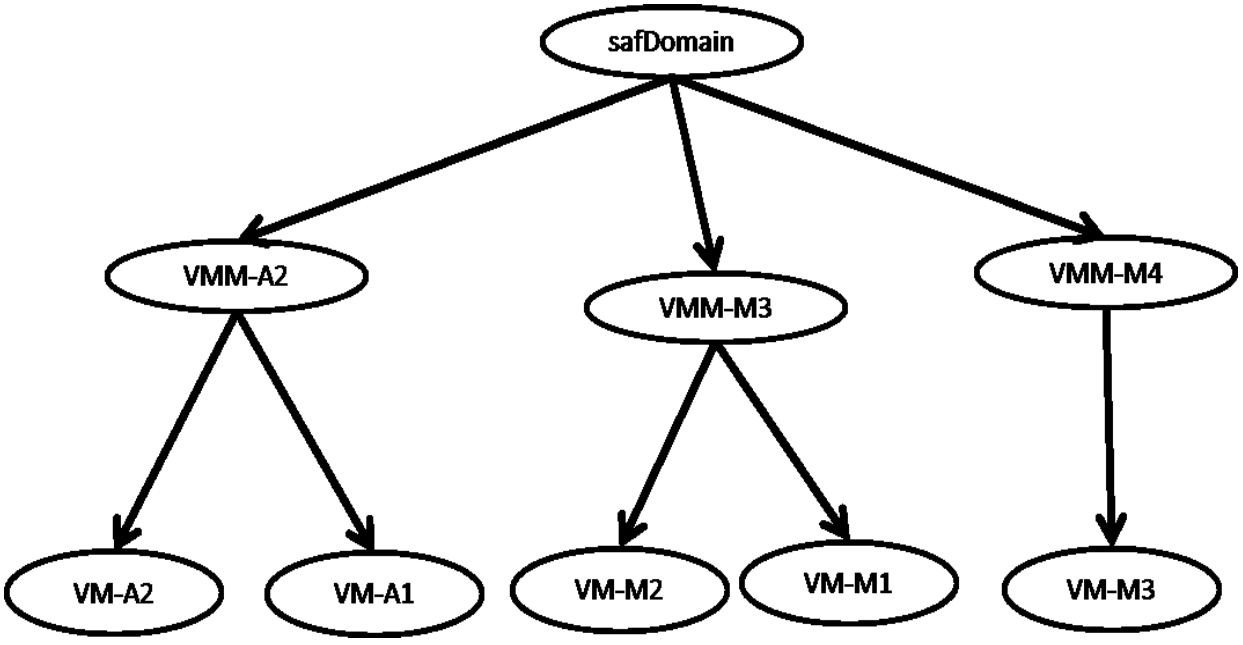


Figure 6.6Target tree after catching VMs

6.5 Generation of the upgrade campaign

In this phase we start generating the upgrade campaign by processing the information present in the source tree and the target tree. This phase itself is divided into four sub-phases 1) Generating the initialization section for the upgrade campaign 2) Creating upgrade procedures for addition of entities 3) Creating upgrade procedures for removal and modification of entities 4) Generating the wrap-up part of the upgrade campaign. In the subsequent subsection we discuss all these sub-phases and will describe how the availability of services is maintained during the execution of the upgrade campaign.

6.5.1 Generation of the initialization section of the upgrade Campaign

Since in the beginning of the upgrade campaign we must catch VMs that are to be removed or modified, therefore in the initialization part of the upgrade campaign we describe the modification in the dependency objects present in the IM for all those VMs such that each VM will have only one sponsoring VMM. The only sponsoring VMM for a VM is the one which is the parent node of the node that represents the VM in the source tree.

Also, in this phase the new PLM types, the AMF types and the software bundles that have to be added in the IM are also specified. Figure 6.7 gives a flowchart for generating the initializing section of the upgrade campaign.

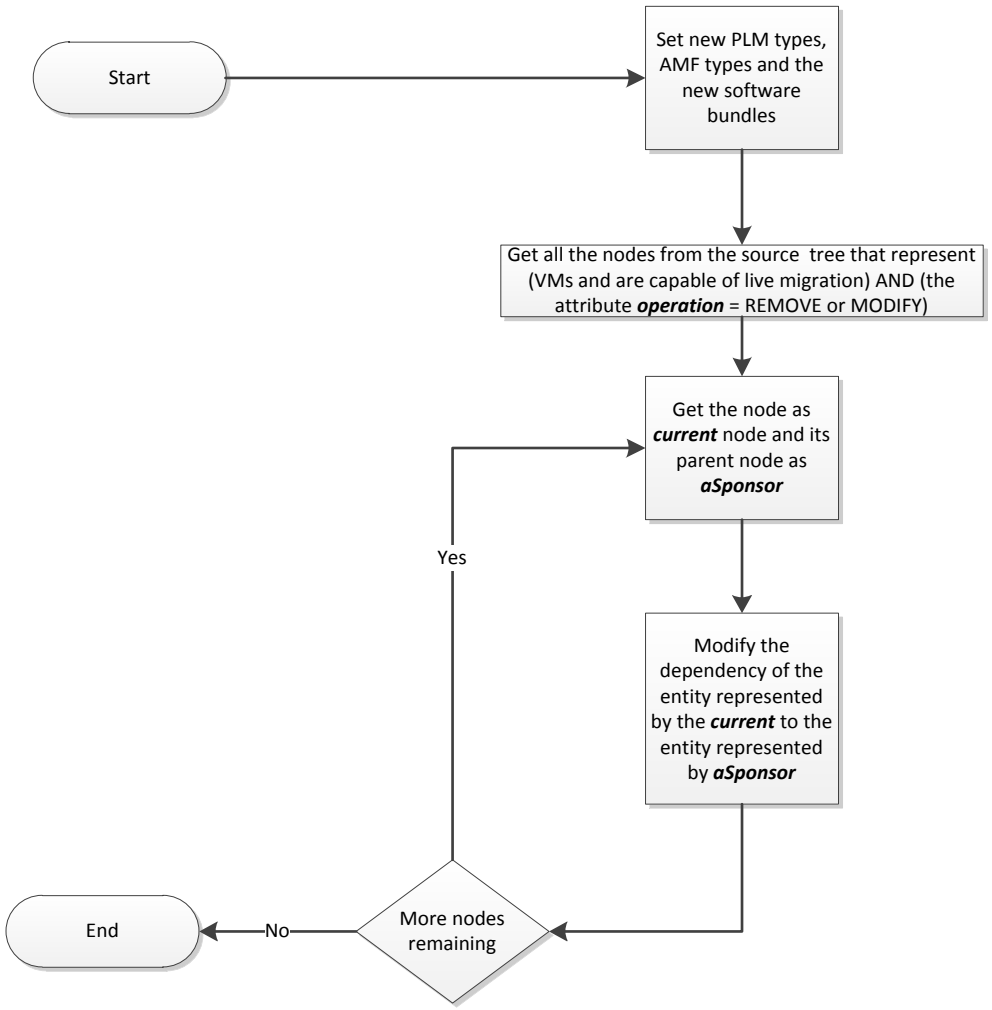


Figure 6.7Generation of the initializing section of an upgrade campaign

6.5.2 Generation of upgrade procedures

In this section we describe how upgrade procedures that add, remove and modify the PLM entities participating in the system are generated.

As the application software products are bundled along with the EE software we do not need upgrade procedures for AMF entities. Yet the IM must be modified to reflect the new, the modified and the removed AMF entities. Hence the addition, the modification and the removal of AMF entities in IM must be carried out along with the addition, the modification or the removal of PLM entities from the IM.

As discussed in the Chapter 5, to maintain service availability from AMF entities we must perform the addition of EEs first. Before upgrade procedures that only remove the entities, upgrade procedure that modifies at least one EE must be executed. Therefore, we divided the creation of upgrade procedures into three parts:

1. Create upgrade procedure to add EEs to the system by using the target tree.
2. Create upgrade procedure that modifies at least one EE using the source tree.
3. Create upgrade procedure for the removal of EEs using the source tree.

The basic rule for creating an upgrade procedure is: from either the source tree or the target tree we get sub-trees based on some constraint (discussed in the sections below). Each sub-tree represent multiple layers within a machine that has to be upgraded, hence for each sub-tree an upgrade procedure is created using a single step upgrade. If the height of the sub-tree is more than one, an embedded upgrade step is created following the rules described in Section 4.1.3.1.

6.5.2.1 Generation of addition upgrade procedures

To generate the upgrade procedures for the addition, we retrieve all the sub-trees from the target tree such that each sub-tree has only nodes which are undergoing addition and the parent node of the root of the sub-tree is not undergoing addition. For each sub-tree an upgrade procedure is created using single step upgrade. The ***execLevel*** (execution level) of the created upgrade procedure is set in way so that they are the first set of the procedures to be executed and in a sequential way. In Figure 6.8 we provide the flowchart to generate the upgrade procedure for the addition of EEs.

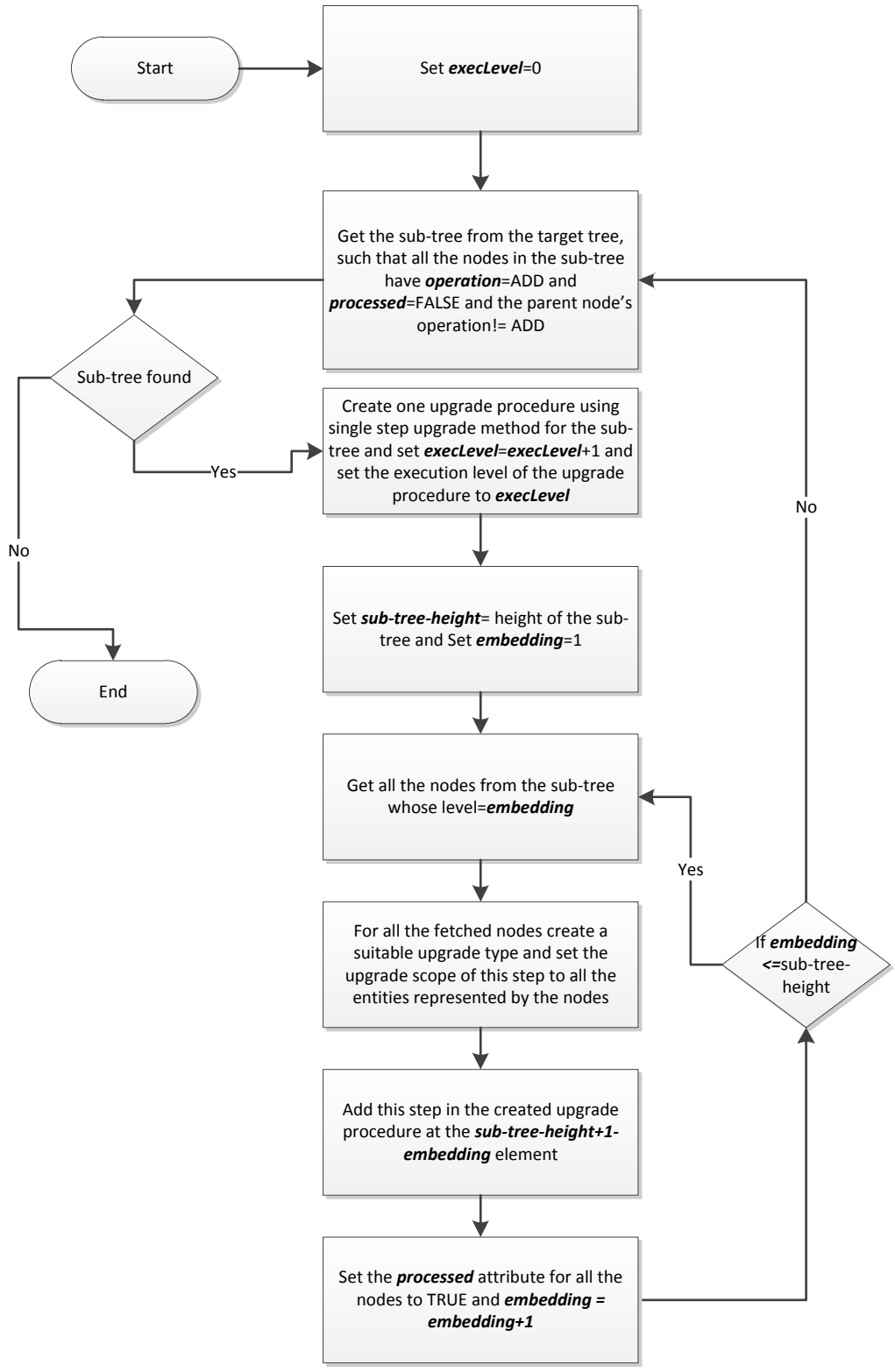


Figure 6.8 Generation of upgrade procedure for the addition of entities

6.5.2.2 Generation of upgrade procedure for the modification and removal of entities

In this part we generate the upgrade procedures that modify at least one EE. We retrieve all the sub-trees from the source tree such that each sub-tree's root is undergoing modification and all of its ancestors node's ***operation*** is set to NO_OPERATION.As these sub tress can contain nodes for removal and modification, these sub–trees are termed as hybrid sub-trees. For each of the hybrid sub-tree an upgrade procedure is created and is scheduled to be executed after the execution of the upgrade procedures created in Section 6.5.2.1. As these upgrade procedures modify the EEs they must not be executed in parallel. In Figure 6.9 we show the flowchart for the generation of these upgrade procedures.

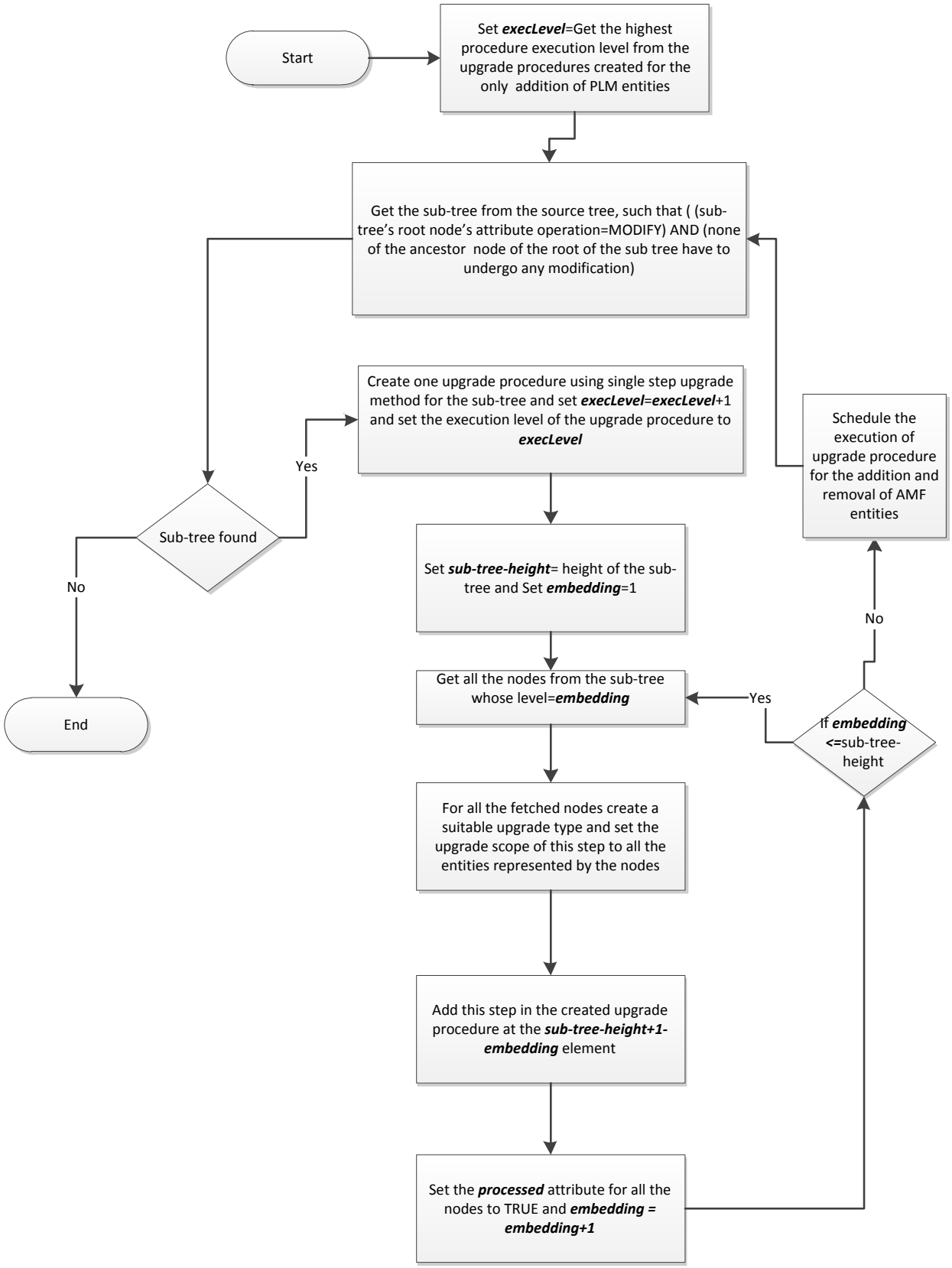


Figure 6.9 Generation of upgrade procedure for the modification and the removal of entities

6.5.2.3 Generation of upgrade procedures for the removal sub-trees

The last set of upgrade procedures to be executed will be the one that just remove entities from the system. Hence these upgrade procedures are scheduled to be executed sequentially in the end. In Figure 6.10 we provide a flowchart for the generation of such upgrade procedures.

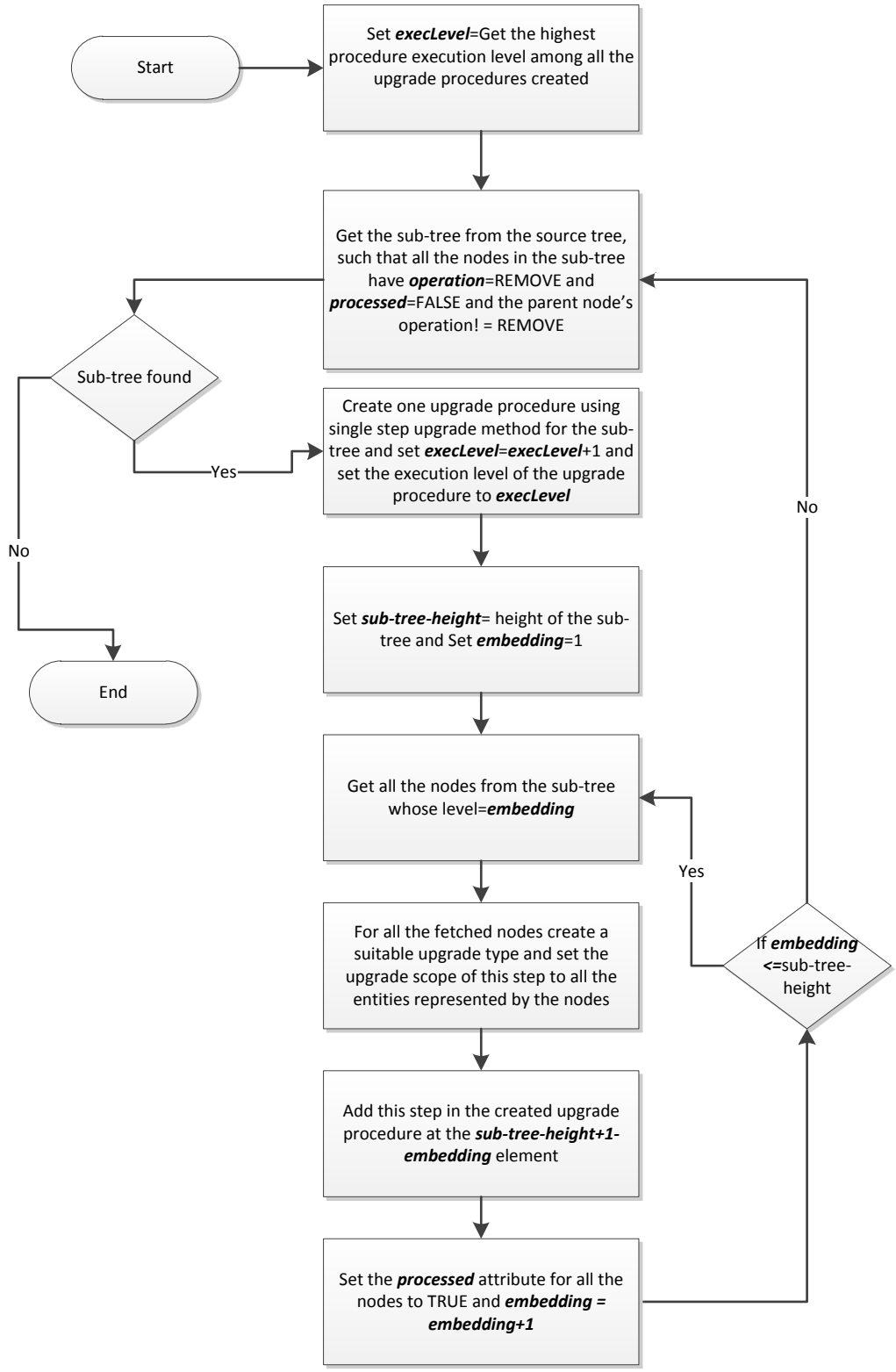


Figure 6.10Generation of upgrade procedures for removal of entities only

6.5.3 Generation of the wrap-up of upgrade campaign

In the beginning of the upgrade campaign all the VMs that were capable of live migration and were undergoing upgrade were caught on one of its sponsoring VMM; hence before the end of the upgrade, the caught VMs must be mobilized, i.e. VM should be enabled to migrate on all of its sponsors as per target configuration. Therefore in the wrap part, we again modify the dependency objects for the VMs that are present in the target configuration and whose dependency was modified in the previous sections.

Apart from above mentioned modification, the removal from the IM of the old PLM types, the AMF types and the software bundles is also generated in this phase.

6.6 Conclusion

In this chapter we discussed how an upgrade campaign is generated for the upgrade of entire system consisting of AMF and PLM entities. The approach described in this chapter uses the solution proposed for the upgrade of multiple layers (discussed in chapter 4) and integrates it with the solution proposed for the upgrade of virtualization facilities (discussed in chapter 5). It uses single step upgrade method for each upgrade procedure and schedules them in a way to have only minimal service outage. As the passed work on the upgrade campaign generation [8] of AMF entities generate rolling upgrade procedure and our approach can only generate single step upgrade procedure; therefore this approach makes an assumption that application software (AMF entities) are bundled along with the OS image. However, to make the new AMF entities visible they must be added/removed/modified to the IM. In this approach we pointed out when the AMF entities should be added/removed/modified in the IM. In the next chapter we describe the prototype we made based on this approach with a case study.

Chapter 7 - A prototype for upgrade campaign generation

Based on the approach for generating an upgrade campaign discussed in Chapter 6 we developed a prototype as a proof of concept. The prototype tool is developed as an Eclipse plug-in using Java. In this chapter we first describe the tool and then we discuss a case study.

7.1 Prototype tool description

Overall view of the data flow and the user interactions with the prototype is shown in Figure 7.1. The input (the source and the target configurations) provided by the user through the Graphical User Interface (GUI) must conform to the system model otherwise an appropriate error message is displayed to the user. If the input conforms to the system model the upgrade campaign generator module generates an upgrade campaign following the approach discussed in Chapter 6.

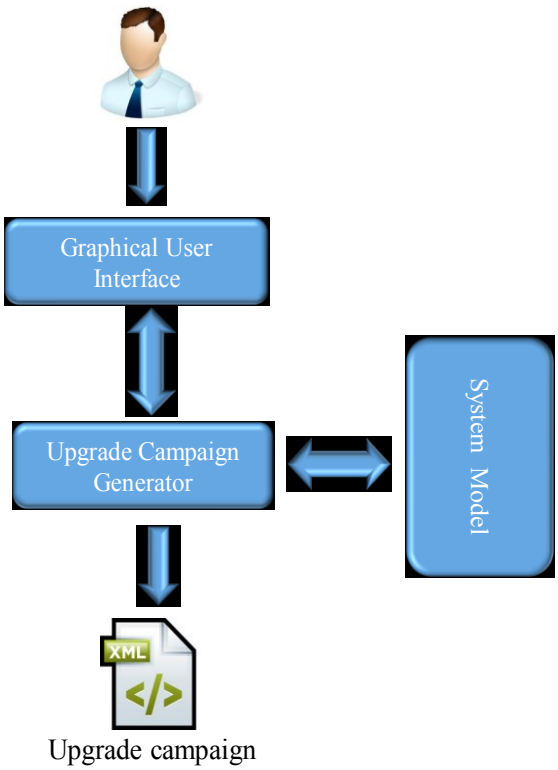


Figure 7.1 Dataflow diagram of upgrade campaign generator tool

7.1.1 Graphical User Interface (GUI) of the prototype tool

The GUI of the prototype tool consists of an input page as shown in Figure 7.2. In this page user provides the source and the target configurations. These configurations must conform to the system model. This input is passed to the upgrade campaign generator module that generates an upgrade campaign.

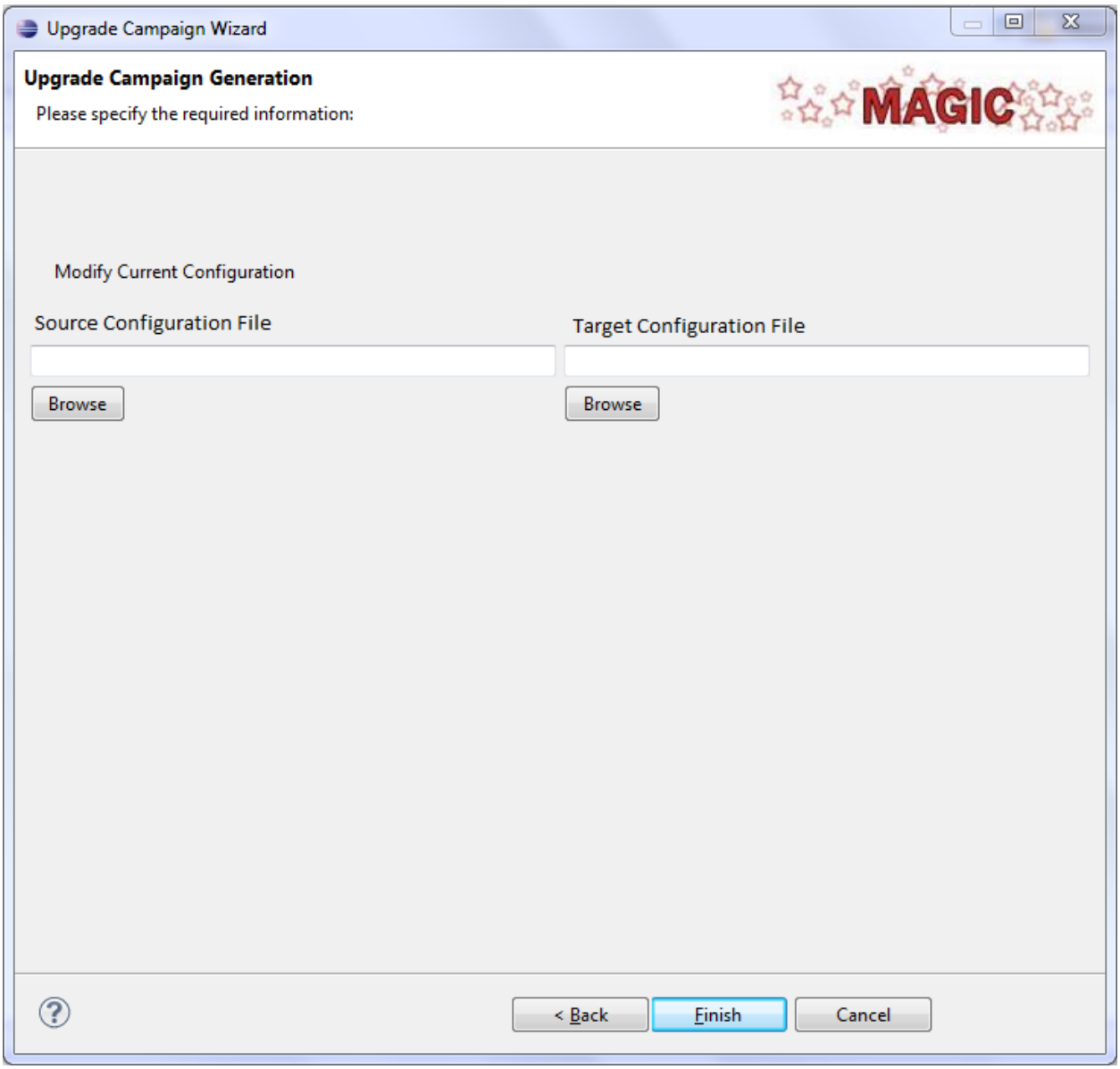


Figure 7.2 Source and target configurations input

7.1.2 Upgrade Campaign Generator

Upgrade campaign generator consists of two modules. For easy representation the Upgrade Campaign generator is split into two figures (Figure 7.3 and Figure 7.4). The main controller class *UpgradeCampaignWizard* is shown in both the figures to represent the connection between two figures.

Pre-processing of the input

Main classes participating in this module are shown in the Figure 7.3. Class *UpgradeCampaignWizard* receives the source and the target system configurations provided by the user. If the inputs conform to the system model, *UpgradeCampaignWizard* sends the input to the *PreProcessInput* class. *PreProcessInput* class uses *ConstructPlmTree* to construct the source tree and the target tree. Once the source tree and the target tree are constructed, *PreProcessInput* uses *PlmDiff* class to evaluate the difference between the source tree and the target tree. After this the catching VMs process is applied on the source tree and the target tree. To do so *PreProcessInput* invokes the *CatchVMs* class which contains the algorithm to catch the VMs. *CatchVMs* class reconfigures the nodes of the source tree and the target tree that represent VMs undergoing upgrade and capable of live migration. These nodes are made children nodes of the appropriate sponsoring node.

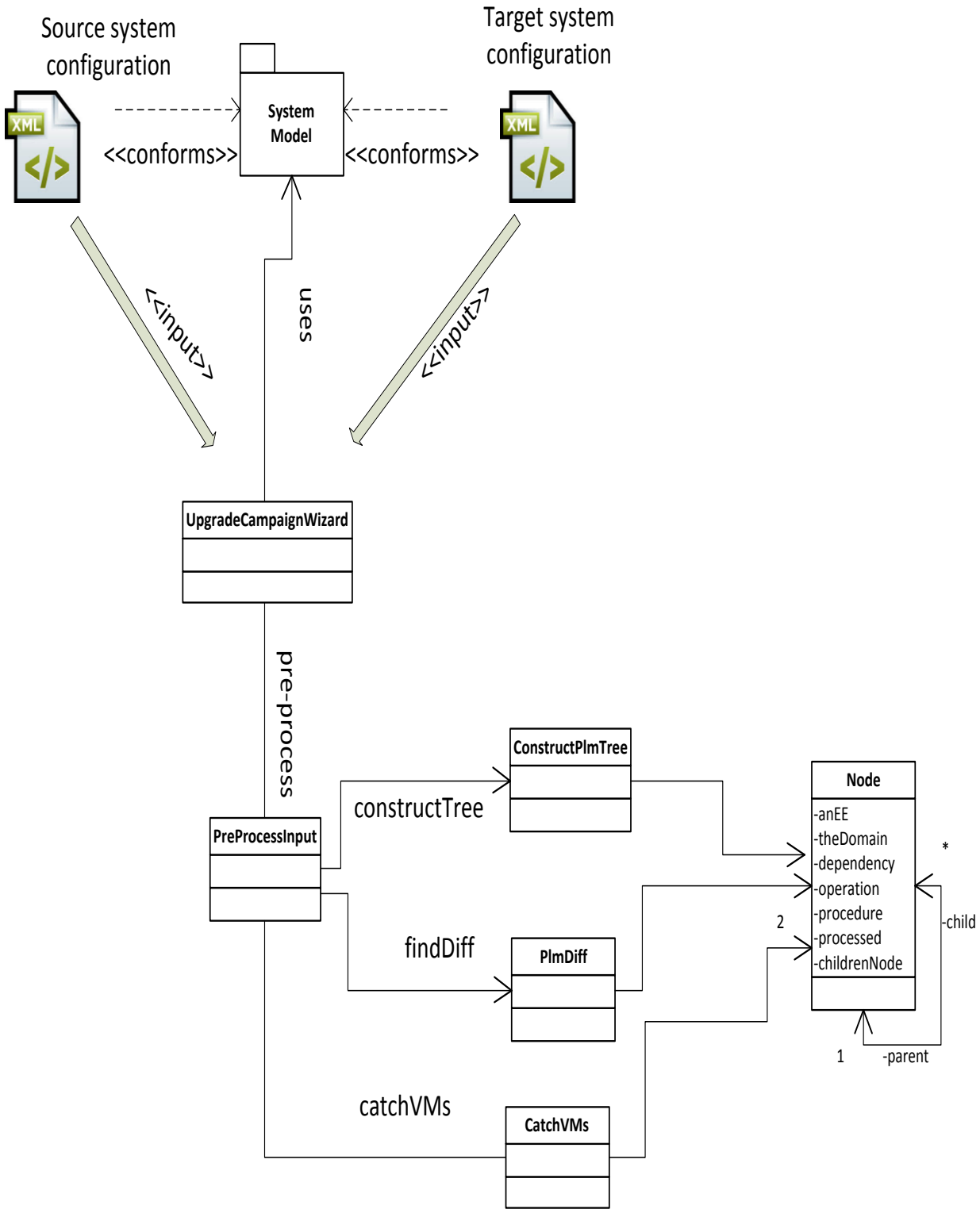


Figure 7.3 Pre-processing of the input

Upgrade campaign generation

In this module (shown in Figure 7.4) upgrade campaign is constructed using the source tree and the target tree. The generated upgrade campaign conforms to the modified Upgrade Campaign Schema which supports the changes we made to represent the new upgrade steps. The modified Upgrade Campaign Schema is represented as *SmfObjectModel* in the Figure 7.4.

UpgradeCampaignWizard passes the reference of the constructed source tree and the target tree to the *GenerateUpgradeCampaign* class. This class divides the construction of the upgrade campaign into five parts handled by the following classes:

- *InitalizeSectionUpgradeCampaign* class creates the initialization section of the upgrade campaign which describes how VMs should be caught by referring to the source tree.
- *CreatePlmEntityAdditionProcedure, CreatePlmEntityModifyProcedureandCreatePlmEntityRemovalProcedure*class creates the upgrade procedures for the addition, modification and removal of the entities. Here it also sets the execution level of the upgrade procedure by using the class ProcedureExecutionLevel.
- *WrapUpSectionUpgradeCampaign* class creates the wrap-up section of the upgrade campaign which describes how migration of VMs should be re-enabled. This is done by referring to the target tree.

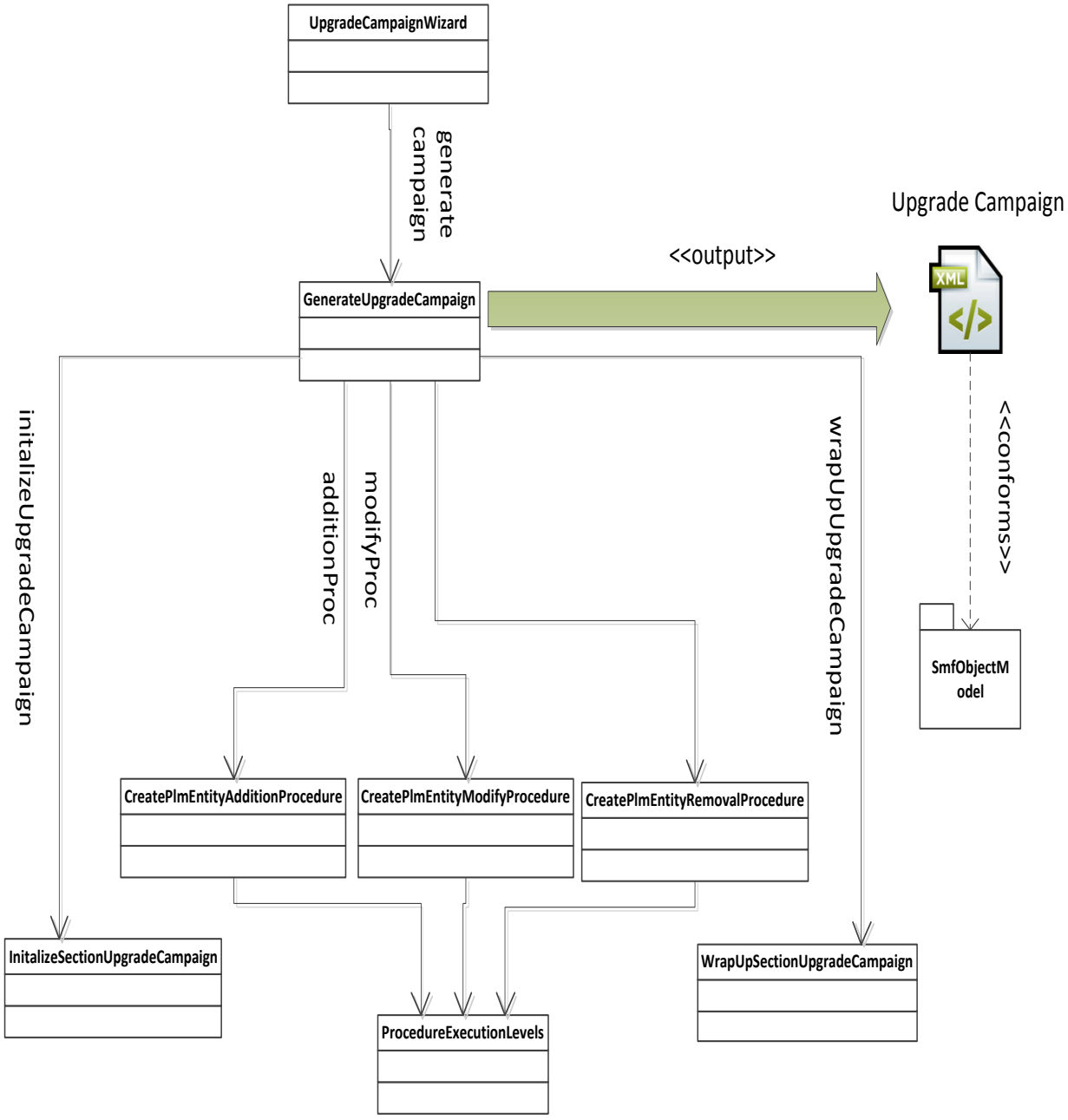


Figure 7.4 Upgrade campaign generation

7.2 A Case study

In this section we discuss a case study which describes a system consisting of an application, virtual machines, virtual machine managers and host OSs. The entire system is managed by OpenSAF. In the subsequent subsections we illustrate the usage of the tool by and the upgrade campaign generated by the tool.

7.2.1 Overview

The application is for allowing a user to select and watch a media displayed on the home page of the application. A user requests for the home page of the application using the HTTP protocol. The request is sent to the HTTP server which internally fetches the list of media files available for streaming from the database server. This list is sent back as a response to the user. The user then selects a media from the list and requests it to be streamed. This request is sent directly to the media server to stream the selected media.

7.2.2 Initial system configuration

The initial system configuration is shown in Figure 7.5. The application consists of three SIs: http service SI protected by the service group: SG-1 using the N-way active redundancy model as http server is state less; media streaming SI protected by the service group: SG-2 using 2N redundancy model as continuity of service is required; database service SI protected by the service group: SG-3 using N-way active redundancy model as the transactions are read only and both the databases have the same replica of data. The service units installed in the SG-1, SG-2 and SG-3 are composed of components of type – Tomcat HTTP Server, VLC Media Player, MySql server, respectively.

Each SU in the system configuration is configured to be hosted on a separate AMF node. Each AMF node corresponds to a cluster node which itself is hosted on an EE. Each of these EEs represents a virtual machine along with its guest OS. These virtual machines are hosted on virtual machine managers (represented as EEs in the system model) of the type VirtualBox. Each of the virtual machine managers is installed on its respective host OSs (represented as EEs in the system model).

All the virtual machines are configured for live migration. The live migration of virtual machines is configured in such a way that at any point of time two redundant SUs are never present on the same virtual machine manager (see Chapter 5, Section 5.2.1, Assumption 3). This way we are sure that when we are putting any virtual machine manager out of service for the upgrade, services provided by the application are still available.

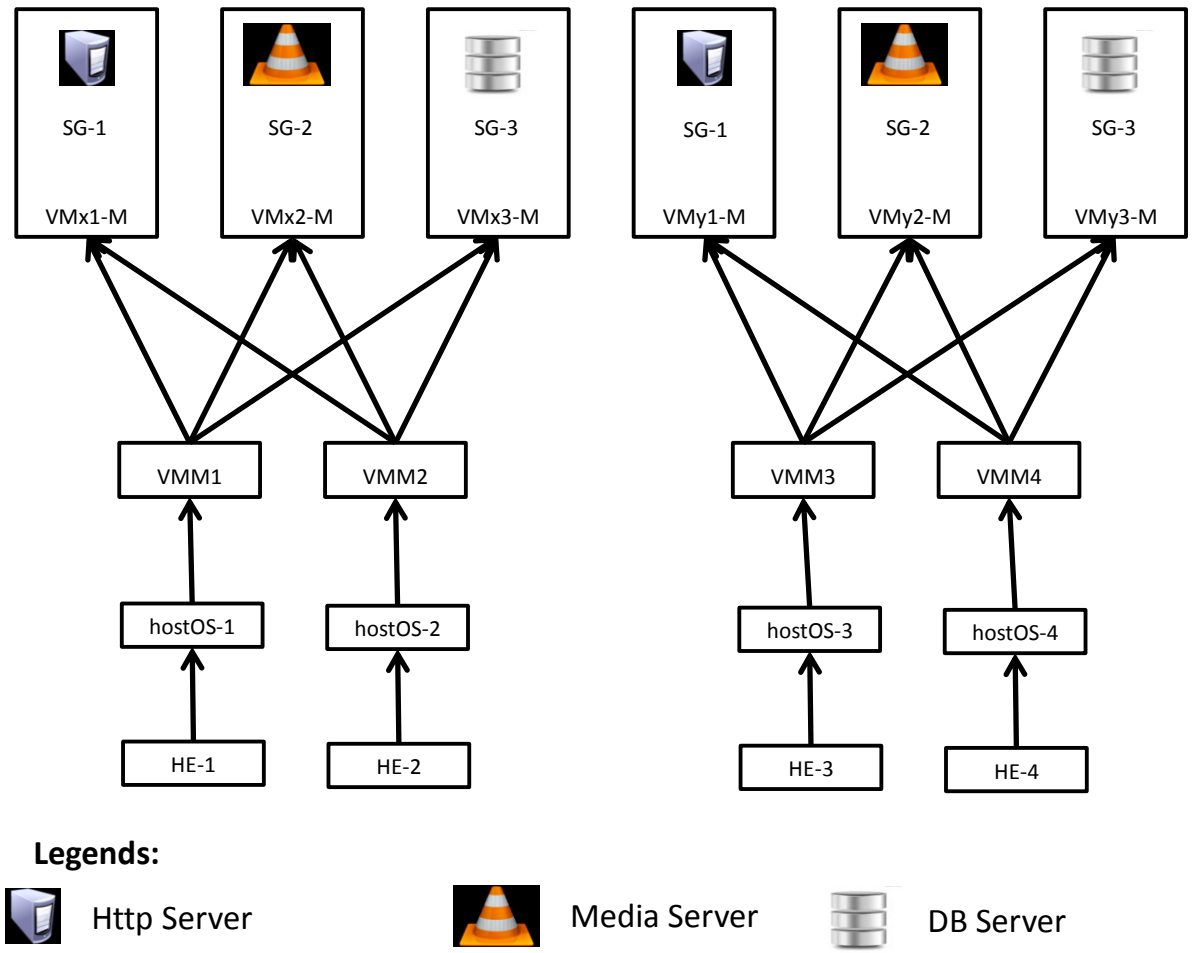


Figure 7.5 Source system configuration

7.2.3 Target system configuration

In the target system configuration we have new versions of the EEs but the same AMF entities as in the initial configuration. Here we assume that the old and new software bundle associated with the old and new version of VM contains the guest OS and the application software as well. Figure 7.6 shows the target configuration in which all the modified EEs are shown with a thick border around them.

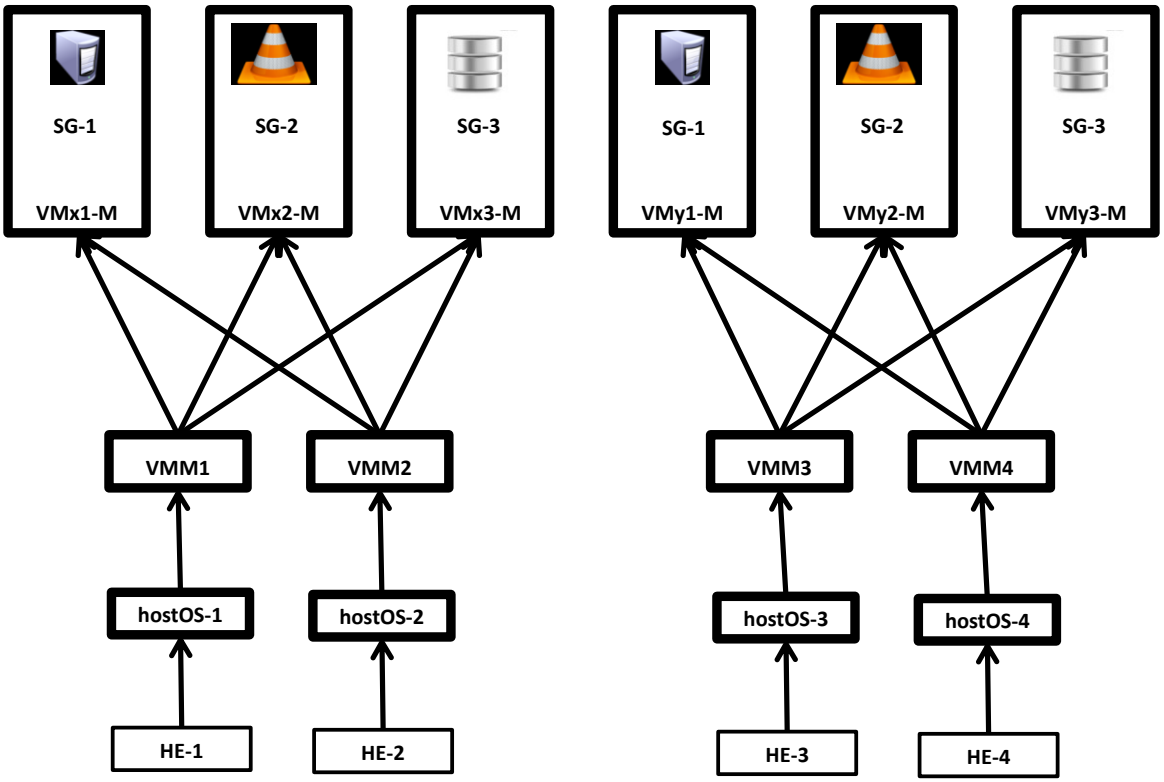


Figure 7.6 Target system configuration

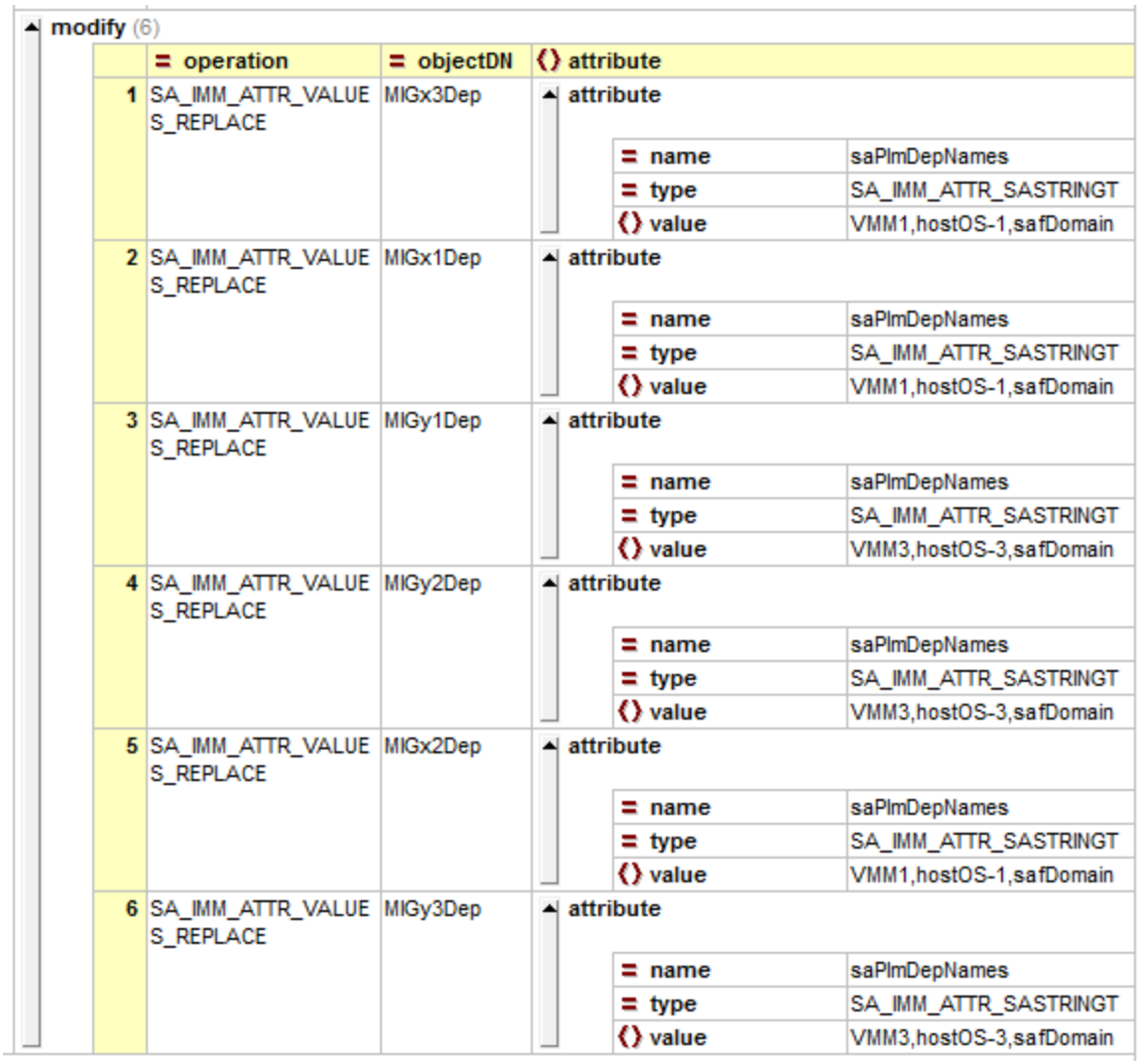
7.2.4 Generated upgrade campaign

The generated upgrade campaign consists of three sections: the initialization section, the upgrade campaign body and the wrap-up section.

7.2.4.1 Initialization section

The initialization section of the upgrade campaign catches the virtual machines (as discussed in Chapter 6, Section 6.5.1) on one of its sponsoring virtual machine manager. This section is shown in Figure 7.7. It shows that all the operations (shown in the column operation) are of type SA_IMM_ATTR_VALUES_REPLACE. This implies that the value of certain attributes of the

objects present in the objectDN has to be replaced with a new value. All the objects present in the objectDN column are the migration dependency (*SaPlmDependency*) objects. The virtual machine and its corresponding dependency object can be identified by the character sequence present in between the word “MIG” and “Dep”. For example the migration dependency of the VMx1-M is present in column 2 with the value of objectDN being MIGx1Dep. The attribute saPlmDepNames of the migration dependency objects indicating the dependency of virtual machine on virtual machine managers is modified to only one sponsoring virtual machine manager evaluated using the catching VMs method. For example the attribute saPlmDepNames of migration dependency object owned by VMx1-M is now set to VMM1,hostOS-1,safDomain which is a DN for the VMM-1 shown in the source and the target configuration.



modify (6)

	= operation	= objectDN	attribute		
1	SA_IMM_ATTR_VALUES_REPLACE	MIGx3Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				value	VMM1,hostOS-1,safDomain
2	SA_IMM_ATTR_VALUES_REPLACE	MIGx1Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				value	VMM1,hostOS-1,safDomain
3	SA_IMM_ATTR_VALUES_REPLACE	MIGy1Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				value	VMM3,hostOS-3,safDomain
4	SA_IMM_ATTR_VALUES_REPLACE	MIGy2Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				value	VMM3,hostOS-3,safDomain
5	SA_IMM_ATTR_VALUES_REPLACE	MIGx2Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				value	VMM1,hostOS-1,safDomain
6	SA_IMM_ATTR_VALUES_REPLACE	MIGy3Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				value	VMM3,hostOS-3,safDomain

Figure 7.7 Initialization section of the upgrade campaign

7.2.4.2 Upgrade campaign body

In this section the actual upgrade procedures are generated. These procedures are shown in the Figure 7.8 and the Figure 7.9. The Figure 7.8 shows four upgrade procedures generated with sequential execution level (show in column saSmfExecLevel column)in order to avoid any possible service outage. Each of these upgrade procedures upgrades a host OS, a virtual machine manager present on the host OS and the virtual machines that are caught on the virtual machine manager.

All these upgrade procedures use single step upgrade method as shown in the last column "upgradeMethod".

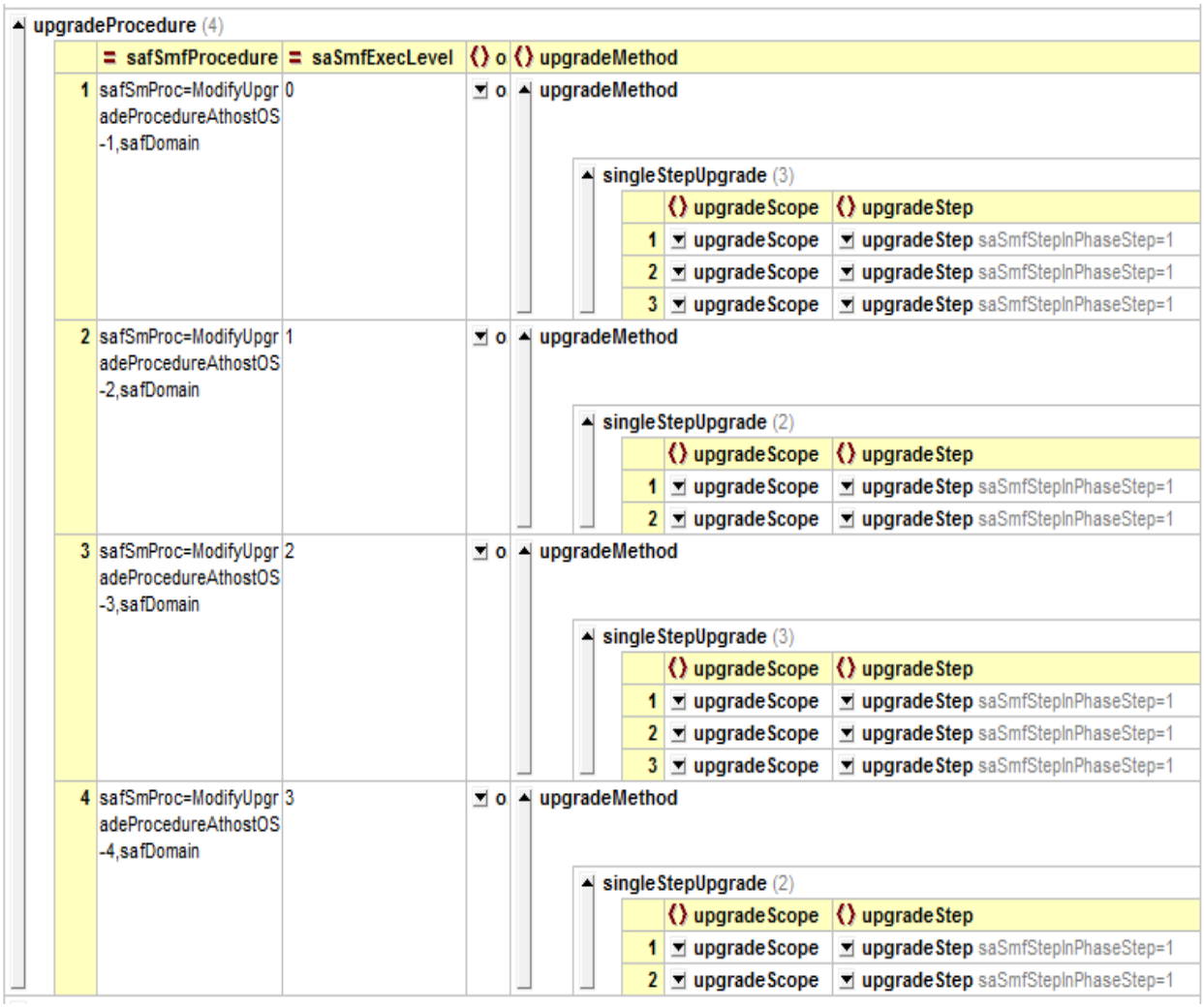


Figure 7.8 The generated upgrade procedures

Figure 7.9 describes the right most column – upgradeMethod of the previous figure in more details. It takes a closer look into the first upgrade procedure. The first upgrade procedure consists of three nested steps which define the degree of embedding in the embedded step as well. Selection of a nested upgrade step among the four non-embedded upgrade steps requires user information which has not been integrated in this prototype. As the "In-Phase" upgrade step handles the

compatibility with the layer on which it executes therefore the type of upgrade step chosen for each nested step of an embedded upgrade step is an “In-Phase” upgrade step. This selection is represented by setting the attribute saSmfStepInPhaseStep to 1 as shown in the leftmost column. The outermost nested upgrade step acts on three virtual machines: VMx1-M,VMx2-Mand VMx3-M indicated by the objectDN in the upgradeScope column. The second row shows another nested upgrade step which upgrades the virtual machine manager: VMM1 and the last row shows the innermost upgrade step which upgrades the HostOS: hostOS-1.

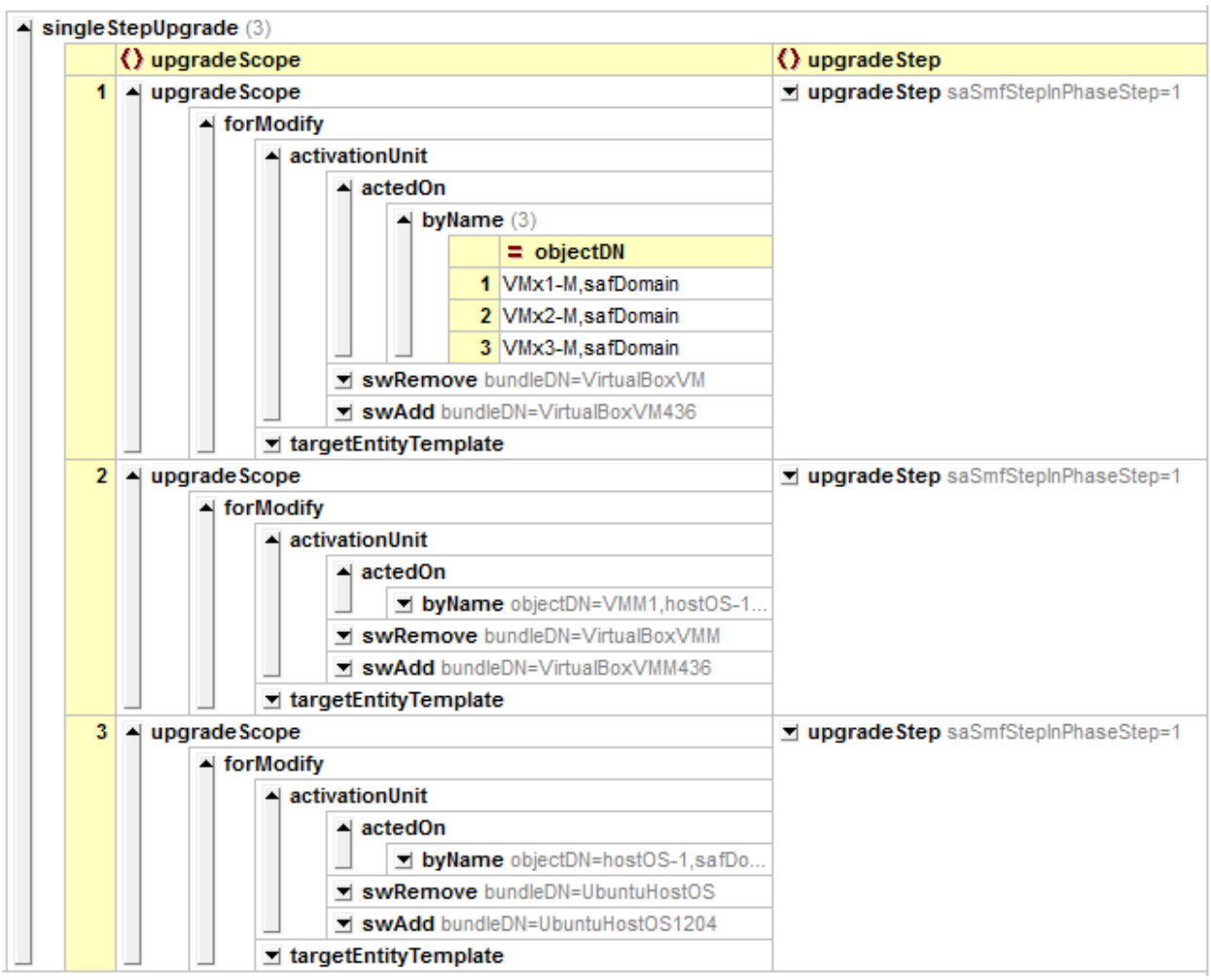


Figure 7.9 In-Depth look into an upgrade procedure of the generated upgrade campaign

7.2.4.3 Wrap-up section

In this section VMs that were caught are re-enabled for migration by restoring their dependencies. This is shown in Figure 7.10. The saPlmDepNames attribute of the migration dependency object for each virtual machine is set to the new values indicating the sponsoring virtual machine managers as per the target configuration.

modify (6)

	= operation	= objectDN	{} attribute		
1	SA_IMM_ATTR_VALUES_REPLACE	MIGx3Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				{} value	VMM1,hostOS-1,safDomain\ VMM2,hostOS-2,safDomain
2	SA_IMM_ATTR_VALUES_REPLACE	MIGx1Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				{} value	VMM1,hostOS-1,safDomain\ VMM2,hostOS-2,safDomain
3	SA_IMM_ATTR_VALUES_REPLACE	MIGy1Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				{} value	VMM3,hostOS-3,safDomain\ VMM4,hostOS-4,safDomain
4	SA_IMM_ATTR_VALUES_REPLACE	MIGy2Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				{} value	VMM3,hostOS-3,safDomain\ VMM4,hostOS-4,safDomain
5	SA_IMM_ATTR_VALUES_REPLACE	MIGx2Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				{} value	VMM1,hostOS-1,safDomain\ VMM2,hostOS-2,safDomain
6	SA_IMM_ATTR_VALUES_REPLACE	MIGy3Dep	attribute	= name	saPlmDepNames
				= type	SA_IMM_ATTR_SASTRINGT
				{} value	VMM3,hostOS-3,safDomain\ VMM4,hostOS-4,safDomain

Figure 7.10 Generated wrap-up section of the upgrade campaign

7.3 Conclusion

In this chapter we discussed the prototype tool implementing the approach developed in Chapter 6. We demonstrated its usage through a case study. The tool takes the source system and the target system configuration as input. If the input conforms to the system model then upgrade

campaign is generated. However, the generated upgrade campaign cannot be tested on OpenSAF as it has not been extended to support the new upgrade steps. Also the SMF implementation in OpenSAF does not support PLM entities. In the next chapter we conclude our work by restating the research contribution and giving some possible future research directions.

Chapter 8 - Conclusion

In this chapter we summarize our research contributions to the domain of live upgrade in the context of service high-availability. We also list some possible future research directions.

8.1 Research contributions

In this thesis we investigated the complexities in the upgrade of lower layers of a machine comprising OSs, virtual machines and virtual machine manager while maintaining the availability of services. We pinpointed the limitations of existing solutions. To overcome the limitations we proposed three new upgrade steps in the SMF in order to properly handle the dependencies between the different layers of a machine and avoid service outage. The step FSM which models the execution of an upgrade step is also modified to model the execution of the proposed embedded and non-embedded upgrade step. This has been done in a way that it does not affect the execution of the existing upgrade steps, the existing upgrade procedure FSM and the existing upgrade campaign FSM. It was also necessary to make changes in the Upgrade Campaign Schema to identify the new upgrade steps and the step embedding.

Furthermore, we extended the approach for the generation of upgrade campaigns to the lower layers as well. With this approach layered upgrade including the upgrade of application layer can be performed as well. This approach takes the source system configuration and the target system configuration and compares them to evaluate the entities that need to be added, removed and modified. More importantly, in this approach we handled the upgrade of virtualization facilities which includes virtual machines capable of live migration. To upgrade virtualization facilities we proposed a method for catching VMs and then set up the rules for scheduling the execution of upgrade procedure that must be followed to avoid outage of services.

Finally, we developed a prototype that implements the devised approach. The prototype is developed as an eclipse plug-in. Based on the input it automatically generates an upgrade campaign XML file which can then be used by SMF to take the live system from the source to the target configuration without impacting the availability of services.

8.2 Future research

In the embedded upgrade step proposal we described the rules that must be observed while nesting a step. These rules are based on two parameters 1) the restartability of the entities 2) the compatibility of the operations with the layer on which they are executed. The semantics of these parameters must be well defined. Restartability parameter is defined for AMF entities which imply that the restart time for the AMF component should be less than the switch over time. But, it should be figured out that how the restartability can be defined for the EEs. Also, more detailed analysis is required to define the compatibility so that it can be used for the step selection.

Automatic generation of the upgrade campaign for the AMF entities was developed previously [8]. This upgrade campaign generator generates upgrade procedures which can use rolling upgrade procedures as well. Contrary to this, our approach to generate upgrade campaign only creates single step upgrade procedures. Therefore we only partially integrated the upgrade campaign generation approach for the PLM entities with the AMF entities. This work can be extended by investigating how rolling upgrade procedures can be generated for the PLM entities and then devising an approach to fully integrate it with the upgrade campaign generation approach for the AMF entities.

References

[1] Service Availability: Principles and Practice, EdsToeroe, M., Tam, F., Wiley and Sons, Chichester, 2012

[2] SA Forum at http://www.SA Forum.org [Accessed on 10 Nov 2013]

[3] SA Forum, Application Interface Specification Overview http://www.myassociationvoice.com/HOA/assn16627/images/SAI-Overview-B.05.03.pdf.

[4] SA Forum, Hardware Platform Interface., http://www.myassociationvoice.com/HOA/assn16627/images/SAI-HPI-B.03.02.AL.pdf

[5] SA Forum, AIS., Availability Management Framework SAI-AIS-AMF-B.04.01 http://www.SA Forum.org/HOA/assn16627/images/SAI-AIS-AMF-B.04.01.pdf

[6] SA Forum, Application Interface Specification. Software Management Framework SAI-AIS-SMF-A.01.02 http://www.SAForum.org/hoa/assn16627/images/sai-ais-smf-a.01.02.pdf

[7] SA Forum, Application Interface Specification Information Platform Management Service SAI-AIS-PLM-A.01.02, http://www.SAForum.org/HOA/assn16627/images/SAI-AIS-PLM-A.01.02.AL.pdf

[8] Automatic generation of upgrade campaign specifications – Master Thesis by Setareh Kohzadi.

[9] VMware virtualization http://www.vmware.com/virtualization/ [Accessed on 30th October 2013]

[10] Virtualization: State of the Art, Version 1.0, April 3, 2008, Copyright © 2008 SCOPE Alliance

[11] White paper on “Windows Server® 2008 R2 Hyper- V™ Live Migration” http://www.microsoft.com/en-us/download/confirmation.aspx?id=12601

[12] Efficient Live Migration of Virtual Machines Using Shared Storage: Changyeon Jo, Erik Gustafsson, Jeongseok Son, and Bernhard Egger School of Computer Science and Engineering, Seoul National University fchangyeon, erik, jeongseok, bernhardg@csap.snu.ac.kr

[13] OpenSAF, http://opensaf.org [Accessed on 10 Feb 2014]

[14] Service Availability Forum, Overview Tutorial SAI-Overview at: http://www.SA Forum.org/link/linkshow.asp?link_id=227891

[15] SA Forum, Application Interface Specification Information Cluster Membership Service SAI-AIS-CLM-B.04.01, http://www.SA Forum.org/HOA/assn16627/images/SAI-AIS-CLM-B.04.01.AL.pdf

[16] SA Forum, Application Interface Specification Information Model Management Service SAI-AIS-IMM-A.03.01 http://www.SA Forum.org/HOA/assn16627/images/SAI-AIS-IMM-A.03.01.pdf

[17] Service Availability Forum, Service Availability Interface, C Programming Model, SAI-AIS-CPROG-B.05.01

[18] MAGIC Project. http://users.encs.concordia.ca/~magic/ [Accessed on 15 March 2014]

[19] Object Management Group at: http://www.omg.org. [Accessed on 10 Feb 2014]

[20] L. E. Moser, P. M. Melliar-Smith, L. A. Tewksbury, "Online Upgrades Become Standard", COMPSAC, pp.982-988, 26th Annual International Computer Software and Applications Conference, 2002.

[21] Puppet labs: What is Puppet https://puppetlabs.com/puppet/what-is-puppet/ [Accessed on 20 Feb 2014]

[22] Debian wiki: Apt package manager, http://wiki.debian.org/Apt [Accessed on 5 March 2014]

[23] Fedora: YUM package manger, http://fedoraproject.org/wiki/Yum [Accessed on 5 March 2014]