THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Présentée et soutenue par :
Ludi AKUE

le mercredi 12 février 2014

Titre :

UN CADRICIEL POUR LA VÉRIFICATION EN LIGNE, GENERIQUE, FLEXIBLE ET EVOLUTIVE DE CONFIGURATIONS DE SYSTÈMES COMMUNICANTS COMPLEXES

École doctorale et discipline ou spécialité :

ED MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture

Unité de recherche :

IRIT - Institut de Recherche en Informatique de Toulouse (UMR 5505)

Directeur(s) de Thèse :

Michelle SIBILLA

Jury :

M. Olivier FESTOR, Professeur, Télécom Nancy (Président)
Mme Marie-Pierre GERVAIS, Professeur, Université Paris-Ouest (Rapporteur)
Mme Francine KRIEF, Professeur, ENSEIRB-IPB (Rapporteur)
M. Frédéric BONIOL, Professeur, Université de Toulouse - ONERA (Examinateur)
M. Emmanuel LAVINAL, Maître de Conférence, UT3 Paul Sabatier (CoDirecteur de thèse)
Mme Michelle SIBILLA, Professeur, UT3 Paul Sabatier (Directeur de thèse)

Remerciements

Mes sincères remerciements s’adressent tout d'abord à Mesdames et Messieurs les membres du jury,

- Mesdames Marie-Pierre Gervais et Francine Krief, rapporteurs de ma thèse, pour l'honneur qu'elles ont bien voulu m'accorder en acceptant d'expertiser mon travail.
- Messieurs Frédéric Boniol et Olivier Festor pour l'intérêt qu'ils ont porté à mes travaux en acceptant de faire partie de mon jury de thèse.

Je tiens à remercier chaleureusement ma directrice de thèse, Madame Michelle Sibilla et mon encadrant Emmanuel Lavinal pour la confiance qu'ils m'ont témoignée pendant ces quatre années. Merci Michelle pour ton *mentoring* et ton soutien tant du point de vue professionnel que personnel. Merci à toi Emmanuel, pour ta disponibilité permanente, tes conseils aussi bien sur des questions de recherche que d'enseignement.

Je tiens également à remercier,

- Monsieur Thierry Desprats. Je me souviens encore de tes cours sur CORBA, l'approche MDA et JXTA qui ont confirmé une vocation :). Merci pour tes précieux retours et pour nos échanges.
- Monsieur Daniel Marquié, pour sa disponibilité, son écoute et nos échanges.
- Messieurs Philippe Vidal et Julien Broisin, mes collègues pendant trois années de monitorat pour leur sympathie et le plaisir que j'ai eu à les côtoyer et à travailler avec eux.

Un grand merci à Messieurs Abdelmalek Benzekri, François Barrère, Romain Laborde, les permanents de l'équipe ainsi que celles et ceux qui animent ou ont animé de leur présence l'équipe SIERA, parce que je ne veux oublier personne : recevez mes sincères remerciements pour votre accueil, et le quotidien que nous avons partagé.

Je tiens à remercier la «*Dream Team*» du bureau 119 (acronyme M.I.B.) pour leur bonne humeur, leur complicité et leur amitié. Je n’oublie pas les acolytes du bureau 122 pour la musique classique :), la créativité et la bonne ambiance, et les geeks (l'ancien et le nouveau) du bureau 127.

Une thèse c'est également un écosystème universitaire fait de belles rencontres et amitiés professionnelles. Un grand merci à monsieur Bruno Roussel pour les discussions, les joutes oratoires, merci pour votre intégrité. Merci à madame Chantal Morand pour sa sympathie et sa convivialité.

Enfin toute ma reconnaissance va à ma famille et à mes ami-e-s qui m'ont soutenue pendant toutes ces années. Merci pour votre affection et votre soutien. Merci d'avoir été là.

À toutes et à tous, merci.

A ma mère Elisabeth AJIKE, partie trop tôt...
A Mag
A Léna

Résumé

Les systèmes communicants complexes constituent une base fondamentale de la vie d'aujourd'hui. Ils supportent de plus en plus de services et d'usages critiques, essentiels tant aux entreprises et administrations qu'à la société en général. L'exemple-type est celui d'Internet avec l'ensemble de ses services et usages variés, architectures et média allant de petits équipements mobiles comme les *smartphones* aux systèmes critiques à grande échelle comme les *clusters* de serveurs et le *cloud*. Il devient dès lors indispensable d'en garantir le fonctionnement effectif et continu.

Pour ce faire, une vision consiste en la mise en œuvre de systèmes de gestion autonomes et adaptifs, capables de reconfigurer dynamiquement et en permanence ces systèmes afin de maintenir un état de fonctionnement désiré face à des conditions opérationnelles, instables et de moins en moins prévisibles. Un frein à l'exploitation effective des solutions de reconfiguration dynamique réside dans le manque de méthodes et de moyens garantissant l'effectivité et la sûreté de ces changements dynamiques de configurations. La contribution générale des travaux de cette thèse fournit des concepts, des méthodes et des outils qui favorisent la mise en œuvre d'une vérification en ligne de configurations.

Notre démarche pour construire ce cadriciel a consisté dans un premier temps à définir un langage de haut niveau dédié à la spécification et la vérification de configurations. Nous avons architecturé dans un deuxième temps, un service de vérification générique, flexible et évolutive *at runtime* capable de manipuler les concepts définis dans ce langage. Enfin, nous avons défini une architecture de composants intermédiaires d'intégration de l'existant. Ce cadriciel permet de supporter un processus de vérification opérationnelle de configurations qui commence, en phase de conception par une spécification rigoureuse de modèles de configurations, puis se poursuit en phase d'exécution du système de gestion à travers une vérification automatique de configurations basée sur ces modèles.

Le cadriciel a fait l'objet d'un prototype que nous avons expérimenté sur une série de cas issus de deux contextes applicatifs différents : la vérification de configurations d'un middleware orienté messages dans un environnement JMX et la vérification de configurations de machines virtuelles dans un environnement CIM/WBEM (standards du DMTF). Les résultats ont montré la faisabilité de l'approche ainsi que la capacité du cadriciel à soutenir une vérification en ligne, flexible et évolutive de configurations favorisant l'intégration de l'existant.

Abstract

Complex networked systems are a fundamental basis of today's life. They increasingly support critical services and usages, essential both to businesses and the society at large. The evident example is the Internet with all its services and usages in a variety of forms, architectures and media ranging from small mobile devices such as smartphones to large-scale critical systems such as clusters of servers and cloud infrastructures. It is therefore crucial to ensure their effective and continuous operation.

A vision to do this consists in the development of autonomous and adaptive management solutions, capable of dynamically and continuously reconfiguring these systems in order to maintain a desired state of operation in the face of unstable and unpredictable operational conditions. A main obstacle to the effective deployment of dynamic reconfiguration solutions is the lack of methods and means to ensure the effectiveness and safety of these dynamic configuration changes. The overall contribution of this thesis is to provide concepts, methods and tools to enable an online configuration verification.

Our approach to build this framework was first to define a high-level language dedicated to the specification and verification of configurations. Second, we designed a generic flexible and adaptable runtime verification service, able to manipulate the concepts defined in this language. Finally, we defined an architecture of adapters for integrating existing systems and platforms. This allows our framework to support a runtime configuration verification process that starts at design time with a rigorous specification of configuration models and continues at runtime through automatic checking of configurations based on these models.

The framework has been implemented in a prototype that has been experienced on a series of experiments from two different application domains: the verification of a messaging middleware's configurations in a JMX environment and the verification of virtual machines' configurations in a CIM/WBEM (DMTF standards) environment. The results showed the feasibility of our approach and the framework's ability to support a flexible and adaptable online verification of configurations that can be integrated with existing management solutions.

Table des matières

Liste des figures

Liste des tableaux

Introduction générale

Nous introduisons dans cette partie, le cadre scientifique de nos travaux, c'est-à-dire le domaine de la gestion de réseaux et services et ses défis actuels. Nous précisons ensuite la problématique ainsi que les motivations de nos travaux. Nous présentons les contributions que nous apportons et concluons avec le plan du manuscrit.

Contexte des travaux et problématique

La gestion des réseaux et services (GRS) désigne le processus global de configuration, de supervision et d'optimisation de ressources informatiques dans le but d'offrir et de garantir aux usagers une certaine qualité de service. Récemment, les limites des approches traditionnelles de gestion face à l'accroissement de la complexité des ressources gérées (taille, dynamicité, interdépendance et hétérogénéité technologique) ont favorisé l'apparition d'un paradigme de la gestion dit gestion autonome et adaptative. Dans ce contexte, les systèmes de gestion sont dotés de capacités pour d'une part, mettre en œuvre et maintenir par leurs propres moyens les objectifs de gestion des administrateurs humains, et d'autre part, s'adapter et adapter dynamiquement les ressources gérées aux changements perçus de l'environnement opérationnel.

La reconfiguration dynamique, c'est-à-dire la capacité du système à modifier sa configuration en cours d'exécution sans interrompre son fonctionnement, est fonctionnellement et opérationnellement fondamentale à la mise en œuvre de cette nouvelle vision. En effet, si la supervision permet d'observer l'état opérationnel du système géré, la capacité de reconfiguration dynamique permet de contrôler le comportement du système et de supporter la maintenance et l'optimisation de son fonctionnement en cours d'exécution.

Afin de permettre l'exploitation effective de la reconfiguration dynamique, il devient critique de disposer de méthodes garantissant la correction et la sûreté des changements de configurations. La réponse à ce genre de problématique est l'objectif du domaine de l'assurance qualité logicielle via les processus de vérification et validation.
La vérification et la validation sont des processus qui visent à garantir qu'un système ou un modèle répond aux spécifications de sa conception et aux exigences de son utilisation. Nos travaux s'intéressent à la vérification de configurations dans un contexte de reconfiguration dynamique.

Traditionnellement, la vérification de configurations est plutôt effectuée en phase de conception ou de façon générale avant la mise en exploitation du système. Elle se concentre sur l'évaluation de contraintes d'intégrité structurelle sur les données de configuration (e.g. respect des types de données, des dépendances, des valeurs autorisées) et n'est pas concernée de fait par les contraintes opérationnelles. Cependant, dans le cadre de la reconfiguration dynamique, cette vérification doit se faire en cours d'exécution (*«at runtime»)*, ce qui introduit de nouvelles exigences à considérer :

- La vérification doit tenir compte des conditions opérationnelles : l'information de configuration ne peut plus être considérée comme une notion purement statique, sa modification et son application sont dépendantes des évolutions des conditions opérationnelles. La vérification opérationnelle d'une configuration doit s'étendre à l'évaluation de son «applicabilité opérationnelle», c'est-à-dire à l'évaluation de sa capacité à être appliquée vu les conditions opérationnelles courantes.

- La vérification doit considérer la dynamicité des environnements gérés et des objectifs de gestion : les éléments gérés supportent des usages dynamiques, ils sont ajoutés, retirés, déplacés à la volée selon des objectifs de gestion eux aussi évolutifs. La vérification opé-

rationnelle de configurations doit donc elle aussi, être adaptable et évolutive en fonction des évolutions des propriétés à vérifier et des configurations sur lesquelles les vérifier.

- La vérification doit prendre en compte l'hétérogénéité de l'existant. Cet existant est marqué par la variété des environnements de gestion et des domaines d'application, la diversité des aires fonctionnelles et la présence de plusieurs standards de gestion. La vérification opérationnelle de configurations doit être à la fois indépendante des plates-formes et protocoles de gestion et favoriser l'intégration de l'existant.

Contributions

Nos travaux visent à créer un cadriciel supportant la vérification opérationnelle de configurations candidates lors de reconfigurations dynamiques dans le contexte de la GRS.

D'un point de vue conceptuel, nous avons pensé l'intégration d'une fonction de vérification au sein de la boucle de gestion MAPE (*Monitor-Analyze-Plan-Execute*). Cette fonction de vérification vient enrichir la décision (*Plan*) en permettant la vérification des configurations candidates avant leur application. Pour répondre au besoin de vérification des configurations en fonction du contexte d'exécution, la fonction de vérification s'appuie sur le bloc fonctionnel de la supervision (*Monitor*). Ce dernier est sollicité pour la récupération de valeurs courantes de paramètres d'état nécessaires à la vérification.

Ce positionnement de la vérification au sein de la boucle de gestion permet d'externaliser une vérification de configurations candidates (issues de la décision) qui tient compte des conditions opérationnelles courantes (remontées par la supervision). L'externalisation de la fonction de vérification présente plusieurs avantages qui permettent de supporter les nouveaux besoins de vérification de configurations :

- La vérification n'est pas liée à un système de décision ou de gestion en particulier. Elle pourrait donc être greffée à tout système de gestion afin de supporter la vérification opérationnelle de ses configurations.

- La vérification devient un mécanisme à part entière. Elle pourrait donc être modulée, adaptée et gérée en fonction des besoins d'utilisation.

Selon nous, pour supporter pleinement ces avantages, le cadre de vérification doit permettre une spécification formelle de configurations indépendante des plates-formes et des protocoles de gestion et un support opérationnel de la vérification adaptable et évolutive.

Notre démarche pour construire ce cadriciel a consisté dans un premier temps à définir un langage de haut niveau, dédié à la spécification et la vérification de configurations. Nous avons architecturé dans un deuxième temps un service générique et adaptable de vérification en ligne capable de manipuler les concepts définis dans ce langage. Enfin, nous avons défini une architecture de composants adaptateurs facilitant l'intégration de l'existant. L'ensemble forme un cadriciel qui supporte un processus de vérification de configurations qui commence, en phase de conception, par la définition de modèles de configuration puis se poursuit en phase d'exécution de l'environnement de gestion à travers une vérification automatique de configurations basées sur ces modèles.

- La phase de conception s'appuie sur le métamodèle MeCSV (*Metamodel for Configuration Specification and Validation*), un métamodèle que nous avons créé et dédié à la spécification et à la vérification de configurations. Ce métamodèle fournit des concepts constructeurs pour modéliser la configuration d'un domaine géré, exprimer les contraintes de conformité structurelle et d'applicabilité opérationnelle, et renseigner les paramètres d'état utiles à l'évaluation de ces dernières. MeCSV a été mis en œuvre sous la forme

d'un profil UML utilisable dans la plupart des modeleurs UML. Il permet ainsi la création outillée de modèles dits « de référence » utiles à la vérification d'instances de configuration.

- Pour la phase d'exécution, un service de vérification basé sur ce métamodèle a été conçu. Ce service présente des interfaces d'utilisation génériques et bien définies qui permettent d'invoquer et d'adapter de façon automatique la vérification selon les scénarii d'usage. Le service de vérification s'appuie sur un moteur d'évaluation de contraintes OCL. Nous avons identifié un besoin de modulation de la vérification afin, par exemple, de pouvoir vérifier une configuration partielle ou de ne vérifier qu'un certain jeu de contraintes. Nous avons ainsi défini une interface d'invocation flexible offrant différents niveaux de granularités de vérification. Un autre besoin d'adaptation concerne l'évolution de la vérification pour supporter la dynamicité des environnements gérés, par exemple l'ajout ou la suppression de contraintes, leur renforcement ou leur relâchement. Pour cela, nous avons également la possibilité d'éditer ce modèle en phase d'exécution du système.

- Une architecture de composants adaptateurs est également fournie afin de favoriser l'intégration avec l'existant. La définition du métamodèle MeCSV permet d'atteindre une généricité dans la spécification et dans la vérification de configurations. Afin de prendre en compte l'existant, il faut supporter une intégration opérationnelle avec les systèmes de gestion existants. Le cadriciel inclut à cet effet l'architecture d'un composant adaptateur qui permet de faire la correspondance entre les modèles de données spécifiques au système de gestion existant et les modèles de données conformes à MeCSV.

- Le cadriciel a fait l'objet d'un prototype qui a été validé expérimentalement dans deux contextes applicatifs différents : la gestion d'un intergiciel orienté messages dans un environnement JMX et la gestion de machines virtuelles dans un environnement CIM/WBEM (standards du DMTF). Les résultats ont montré l'applicabilité du métamodèle sur les deux cas d'études ; l'évaluation effective de configurations candidates par le service de vérification vis-à-vis des valeurs opérationnelles courantes remontées par la supervision ; l'intérêt de la flexibilité et de l'évolutivité de la vérification proposée et la capacité d'intégration du service de vérification à un système de gestion existant.

Organisation du manuscrit

Le manuscrit est organisé en deux parties, chacune composée d'un ensemble de chapitres.

- La première partie présente l'évolution de la gestion de réseaux et de services et introduit les concepts et terminologies relatifs à la vérification de configurations dans le contexte de la gestion autonome et adaptative. L'analyse de cette problématique au regard de l'état de l'art, permet d'identifier les propriétés clés d'une solution idéale de vérification de configurations dans ce contexte.

- La seconde partie développe les contributions de nos travaux, notamment la mise en œuvre d'un cadriciel répondant aux propriétés clés précédemment identifiées. Les briques logicielles de ce cadriciel et leur méthodologie d'utilisation sont détaillées dans un premier temps. Elles font ensuite l'objet d'un prototypage qui a été validé expérimentalement dans la vérification de configurations de systèmes gérés observables et reconfigurables issus de deux contextes applicatifs différents.

1
Contexte et Problématique

Chapitre 1

Evolution de la reconfiguration dans la gestion de réseaux et de systèmes complexes

L'objectif de ce chapitre est de présenter le contexte général de nos travaux à savoir la gestion des réseaux et services et plus précisément la reconfiguration dynamique au sein de cette gestion. Face à la complexité croissante des systèmes actuels, la gestion de réseaux et services évolue vers une vision plus autonome qui s'appuie fonctionnellement et opérationnellement sur les capacités des systèmes gérés à être reconfigurés dynamiquement. Afin de permettre l'exploitation effective de cette capacité, il devient critique de disposer de méthodes garantissant la correction et la sûreté des changements autonomes de configurations.

I.1. Reconfiguration dans les systèmes complexes

Cette section introduit la reconfiguration dans les systèmes complexes actuels, en particulier l'évolution de la reconfiguration pour s'adapter aux caractéristiques des systèmes informatiques actuels.

I.1.1. Caractéristiques des systèmes complexes actuels

I.1.1.1. Définition

Un système peut être vu comme un ensemble cohérent d'entités ou composants en interaction dans le but d'assurer une fonction donnée. Un système interagit généralement avec d'autres systèmes (naturels, humains, matériels, logiciels), ces autres systèmes constituent son environnement extérieur.

Définition 1 ***Système complexe*** *: sont qualifiés de systèmes complexes, les systèmes composés d'un grand nombre d'entités souvent de différents types, structurés en différents niveaux d'organisation. La complexité dans le cadre de ces systèmes provient des interactions diverses, simultanées et non linéaires entre d'une part, les entités du système et d'autre part entre le système dans sa globalité et son environnement extérieur.*

On parle notamment dans ces systèmes de la notion de comportement émergent, c'est-à-dire que des interactions entre entités peut émerger un comportement global non prévisible par la connaissance du comportement individuel des entités.

Cette définition recouvre autant les systèmes complexes naturels (par exemple, le système nerveux, les colonies de fourmis) que les systèmes complexes artificiels dont les systèmes informatiques ou à dominante logicielle. Ce sont ces derniers qui nous intéressent.

I.1.1.2. **Classification des systèmes informatiques**

Il devient de plus en plus difficile de classer les systèmes informatiques en catégories nettes de par l'évolution et la complexification croissante des usages qu'ils supportent. On peut néanmoins distinguer les systèmes d'information, les systèmes d'exploitation, les systèmes embarqués, les systèmes temps réel, les systèmes communicants, les systèmes expert, les systèmes scientifiques etc. [Pressman, 2001]. La démocratisation de l'informatique et son usage dans différents domaines, rendent ces systèmes de plus en plus complexes, interdépendants et interconnectés.

I.1.1.3. **Caractéristiques des systèmes complexes informatiques actuels**

Les systèmes complexes informatiques actuels sont caractérisés par une complexité croissante dont les manifestations rendent difficile le maintien des méthodes de développement et d'exploitation traditionnelles.

– Une complexité croissante

Les systèmes informatiques actuels se distinguent notamment par une complexité croissante en :

- Taille : ils sont composés d'entités de plus en plus nombreuses, interconnectées et distribuées géographiquement.
- Hétérogénéité technologique : ils sont hétérogènes tant du point de vue matériel et logiciel que des domaines d'application ciblés.
- Complexité technologique : ils se diversifient et sont en constante évolution. Ils fournissent davantage de fonctionnalités et ces fonctionnalités sont de plus en plus évoluées.
- Qualité de service : les systèmes informatiques sont devenus des ressources vitales de presque tous les aspects de notre société, supportant des usages dynamiques de plus en plus critiques et exigeants en qualité de service et sûreté de fonctionnement.

L'exemple-type est celui d'Internet constitué de plusieurs systèmes communicants, support d'activités économiques comme les activités bancaires, le commerce en ligne ou les administrations.

– De nouveaux paradigmes de développement et d'exploitation de systèmes

Du fait de cette complexité, la maîtrise de la conception, de l'exploitation et de la maintenance de ces systèmes pose problème et fait l'objet de travaux de recherche actifs. Une tendance générale est de les doter de capacités d'auto-adaptation, c'est-à-dire pouvoir dès la phase de conception y intégrer des propriétés d'adaptation, pour qu'ils puissent lors de la phase d'exploitation, détecter automatiquement des changements opérationnels et modifier leur comportement en fonction.
Une approche de solution repose sur la capacité de ces systèmes à se reconfigurer ou à être reconfigurés dynamiquement en cours d'exécution. Nous nous focaliserons sur ce type de solution dans la suite de nos travaux.

I.1.2. **Configuration et reconfiguration de systèmes complexes**

Les notions de configuration et de reconfiguration interviennent dans la phase d'exploitation des systèmes. Le développement du système en l'occurrence la programmation de code permet d'implanter les fonctionnalités requises. Dans certains systèmes dits reconfigurables, leur

mise en exploitation requiert une phase de configuration initiale où le système est paramétré en vue de sa mise en service. Lorsque le système le permet, il peut être également reconfiguré au cours de sa vie pour maintenir un fonctionnement désiré, c'est le cas par exemple des équipements réseaux comme les routeurs ou des serveurs d'applications [Desertot, 2007 ; Buchmann, 2008].

I.1.2.1. **Notion de configuration**

Pour qu'un système informatique reconfigurable fournisse un service attendu, il est nécessaire de le configurer. La mise en oeuvre de la configuration diffère selon les différents domaines d'application, néanmoins, elle sert cet objectif commun.

– Définitions

Le Larousse propose deux définitions de la configuration d'un système informatique, dans la première, la configuration est définie comme *«l'ensemble des caractérisques matérielles ou logicielles d'un système informatique»* et dans la deuxième, la configuration fait référence à la *«modification ou réglage de paramètres informatiques en vue de l'optimisation du fonctionnement du système»*. Afin de lever toute ambiguïté, nous retiendrons la première définition pour la configuration et nous parlerons de processus de configuration initiale ou de processus de reconfiguration lorsqu'il s'agira de parler de la configuration en tant que processus.

Définition 2 ***Configuration*** *: une configuration définit la structure et le comportement fonctionnel d'un système. Une configuration représente un ensemble de paramètres prédéfinis fonctionnels ou non-fonctionnels à appliquer à un système afin qu'il délivre les fonctionnalités attendues. Ces paramètres prédéfinis sont appelés des paramètres de configuration [CIM-WP, 2000].*

Définition 3 ***Paramètre de configuration*** *: un paramètre de configuration représente l'unité d'information de configuration, il est généralement accessible en lecture et écriture [Burgess & Couch, 2006].*

– Description des informations de configurations

La vue unifiée des paramètres de configuration d'un système constitue ses informations ou ses données de configuration. Ces informations sont habituellement sauvegardées dans des fichiers de configuration, mais elles peuvent également l'être dans des bases de données dédiées (par exemple le registre Windows). La nature et l'organisation des informations de configuration diffèrent selon les domaines applicatifs (matériel, logiciel, architecture, réseaux et systèmes).

Nature des informations de configuration

Une configuration peut faire référence à un ensemble de valeurs d'attributs, des fichiers de configurations, un ensemble de commandes et fonctions, ou un assemblage de composants et de connecteurs. Des exemples de paramètres de configuration sont la résolution d'un écran, le fichier du pilote d'un périphérique, un composant d'un serveur d'application ou encore l'adresse IP d'une interface réseau.

Organisation des informations de configuration

L'organisation des paramètres de configuration, c'est-à-dire leur regroupement en catégories peut s'effectuer en fonction des objectifs poursuivis ou selon les rôles joués par les entités. Ces catégories peuvent être à leur tour composées entre elles. Les paramètres de configuration

présentent souvent des dépendances les uns par rapport aux autres qui traduisent des dépendances de fonctionnement entre les systèmes qu'ils caractérisent.

Spécification des informations de configuration

De façon générale, les configurations sont spécifiées à travers la description de paramètres de configuration sous la forme de paire «attribut - valeur» ou à l'aide de langages plus ou moins évolués, tels que les langages de programmation, langages de modélisation, langages architecturaux et autres grammaires (Java, UML, ADL, XML, BNF). Cette spécification peut se faire de façon manuelle ou automatique.

Si la mise en service du système requiert une configuration initiale du système, dans les systèmes reconfigurables, un processus continu d'adaptation de la configuration peut suivre cette phase initiale. Ce processus a pour nom la reconfiguration.

I.1.2.2. **Notion de reconfiguration**

Lorsque la configuration courante n'est plus adaptée au contexte d'exécution, la reconfiguration est le processus qui va permettre de décider et d'appliquer une nouvelle configuration plus adaptée [Desertot, 2007].

Définition 4 ***Reconfiguration*** *: La reconfiguration peut être définie comme la modification d'une configuration existante et déjà déployée. Reconfigurer un système permet de l'adapter c'est à dire de modifier sa structure et/ou son comportement fonctionnel pour répondre aux variations de son contexte opérationnel, à l'évolution des besoins ou bien pour prévenir ou corriger un comportement incorrect [Siddiqi, 2006].*

Les modalités de mise en œuvre de la reconfiguration sont différentes en fonction du contexte et de la nature des systèmes concernés. La reconfiguration peut se faire soit de façon statique, soit de façon dynamique :

Définition 5 ***Reconfiguration statique*** *: on parle de reconfiguration statique, lorsque la reconfiguration est effectuée lorsque le système est à l'arrêt, par exemple le système est arrêté, sa configuration est changée puis le système est redémarré.*

Définition 6 ***Reconfiguration dynamique*** *: on parle de reconfiguration dynamique lorsque les changements de configurations sont réalisés à chaud pendant l'exécution du système sans interrompre son fonctionnement.*

Si la reconfiguration statique représente la façon classique de procéder à une reconfiguration, la tendance générale dans plusieurs domaines informatiques est à la reconfiguration dynamique. Cette capacité est devenue nécessaire dans les systèmes complexes actuels, en particulier dans les systèmes dont l'environnement change constamment, tels les réseaux mobiles, les systèmes de haute disponibilité dont l'arrêt est coûteux ou difficile tels les services télécoms ou les services bancaires [Ketfi, 2004].

I.1.2.3. **Reconfiguration dynamique**

La reconfiguration dynamique reposent sur plusieurs capacités : la capacité à prendre en compte des objectifs de comportement désiré, la capacité à détecter et à qualifier des change-

ments «indésirables» internes comme externes, la disponibilité de mécanismes de décision et d'exécution de modifications (de configurations) appropriées [Besseron, 2010].

– Dimensions de la reconfiguration dynamique

On peut catégoriser la reconfiguration dynamique suivant plusieurs dimensions qui sont fonction de la nature des objectifs visés, de la nature du système considéré (système fermé, semi-ouvert ou ouvert) et du périmètre concerné par la reconfiguration [Samaan & Karmouch, 2009 ; Cheng et al., 2009].

Nature des objectifs visés

Une reconfiguration poursuit en général trois catégories d'objectifs [Siddiqi, 2006] :

- La survie du système : pérenniser le fonctionnement du système. La configuration est modifiée pour prévenir ou corriger un comportement incorrect ou indésirable.
- L'évolution du système : adapter le fonctionnement du système à de nouveaux besoins. La configuration du système est modifiée pour traduire les évolutions de l'environnement d'exécution et des besoins métier.
- L'augmentation de capacités : augmenter les fonctions du systèmes. La configuration du système est modifiée pour altérer le comportement fonctionnel du système, par exemple rajouter de nouvelles fonctionnalités, modifier les fonctionnalités délivrées.

La nature des objectifs visés influe sur le type de reconfiguration envisagée : la reconfiguration peut être de type fonctionnel, non-fonctionnel ou technologique [Cheng et al., 2009].

- Reconfiguration fonctionnelle : on parle de reconfiguration fonctionnelle lorsque la décision de reconfiguration répond à l'évolution d'un objectif de fonctionnement, par exemple un nouveau service doit être supporté.
- Reconfiguration non-fonctionnelle : on parle de reconfiguration non-fonctionnelle lorsque la décision de reconfiguration répond à l'amélioration de propriétés non-fonctionnelles, par exemple l'amélioration de la performance.
- Reconfiguration technologique : on parle de reconfiguration technologique lorsque la décision de reconfiguration concerne des changements liés à la technologie, par exemple la version d'une entité doit être mise à jour.

Nature du système considéré

La nature du système considéré (ouvert, fermé ou semi-ouvert) influe sur la nature de la stratégie de reconfiguration adoptée : la reconfiguration peut être soit planifiée, soit spontanée [Samaan & Karmouch, 2009].

- Reconfiguration planifiée : dans une stratégie de reconfiguration planifiée, les configurations potentielles du système sont prédéfinies à l'avance, avant la mise en service du système.
- Reconfiguration spontanée : dans une stratégie de reconfiguration spontanée, les configurations potentielles sont décidées à chaud.
- Reconfiguration fermée : on parle de reconfiguration fermée dans le cas de systèmes reconfigurables fermés. Ces derniers embarquent et maintiennent des stratégies de reconfiguration prédéfinies constantes sur le cycle de vie du système géré.

- Reconfiguration ouverte : on parle de reconfiguration ouverte lorsque les stratégies de reconfiguration évoluent pour s'adapter aux changements opérationnels au cours du cycle de vie du système.

Périmètre affecté

La reconfiguration peut affecter un composant du système, plusieurs composants du système ou le système dans sa globalité. Elle peut être paramétrique ou structurelle [Cheng et al., 2009].

- Reconfiguration paramétrique : la reconfiguration paramétrique consiste à modifier les paramètres des composants du système (ces paramètres peuvent être des attributs ou des fonctionnalités).

- Reconfiguration structurelle ou «compositionnelle» : dans la reconfiguration structurelle ou compositionnelle, la structure même du système est modifiée, des composants sont ajoutés, enlevés, déplacés.

Modalités de mise en oeuvre

La reconfiguration est mise en oeuvre à travers différentes méthodes et algorithmes de raisonnement selon les domaines applicatifs concernés. On peut citer par exemple les approches basées sur les politiques et les approches logicielles.

- Approches basées politiques : ces approches sont basées sur des politiques de configuration telles que les politiques ECA (Evènement-Condition-Action). On retrouve ce type d'approches dans la reconfiguration dynamique des réseaux et systèmes [Lymberopoulous et al., 2003 ; Agrawal et al., 2005].
  «Si le temps de réponse de 95% des serveurs web dépasse 2 secondes et que des ressources sont disponibles Alors augmenter le nombre de serveurs web» [Walsh et al., 2004] est un exemple de politique qui déclenche une reconfiguration structurelle dans une optique d'optimisation du fonctionnement des ressources.
  Les politiques de configuration sont souvent utilisées conjointement avec d'autres approches comme les fonctions d'utilité, les approches probabilistes et les techniques d'apprentissage automatique [Derbel et al., 2009; Bahati & Bauer, 2008].

- Approches logicielles : la mise en œuvre de la reconfiguration dynamique dans les domaines des architectures logicielles s'appuie sur des mécanismes tels que l'utilisation des modèles architecturaux, la réflexivité et la modélisation orientée aspect [Desertot, 2007 ; Morin et al., 2009].
  Les modèles architecturaux représentent le système informatique sous forme d'un ensemble de composants et connecteurs. Ces composants et connecteurs sont annotés de contraintes et de propriétés dont la violation déclenche des reconfigurations.
  La réflexivité permet de rendre les systèmes capables de raisonner et d'agir sur eux-mêmes. La modélisation orientée aspect est une technique de modélisation permettant de rendre les configurations plus modulaires et plus faciles à évoluer. La modélisation d'un système est séparée en deux modèles, le modèle de base (fonctionnalités communes) et les modèles *variants* qui capturent les variations possibles du système. La reconfiguration fait évoluer les modèles *variants*.

I.2. Reconfiguration dynamique dans la Gestion de Réseaux et Services

La Gestion des Réseaux et Services (GRS) évolue vers une gestion plus autonome et adaptative où les systèmes de gestion deviennent capables de s'auto-configurer, réparer, optimiser et protéger par rapport à des objectifs métier de haut niveau. La reconfiguration dynamique est une capacité fondamentale pour la réalisation de cette évolution.

I.2.1. Evolution de la Gestion de Réseaux et Services

I.2.1.1. Définition de la Gestion de Réseaux et de Services

Définition 7 ***La gestion de réseaux et de services (GRS)*** *est l'ensemble des activités englobant la mise en œuvre de méthodes, de procédures et d'outils ayant trait à la supervision et au contrôle d'entités informatiques dans le but d'offrir et de garantir aux usagers une certaine qualité de service.*

Les entités informatiques gérées peuvent être matérielles (e.g. équipements d'interconnexion, stations de travail), logicielles (e.g. systèmes d'exploitation, applications) ou encore des ressources plus abstraites (domaines administrations, flux de données, droits d'utilisateurs). On les retrouve au niveau des réseaux, des systèmes, des applications et des services.

– Modèles conceptuels

Quatre modèles conceptuels ont été définis pour encadrer la mise en œuvre de ces activités de gestion [ISO10040, 1998]: le modèle informationnel pour identifier et représenter ce que l'on veut gérer, le modèle organisationnel pour déterminer les entités qui assurent la gestion, le modèle de communication pour définir les protocoles d'interactions entre ces entités, et le modèle fonctionnel pour définir les objectifs poursuivis.

Le modèle informationnel

Ce modèle permet de caractériser et représenter une vue logique des entités gérées sous la forme de modèles de données. Cette vue abstraite reflète via des attributs ou paramètres, différents types d'informations de gestion. On distingue notamment les informations représentant les ressources gérées et les informations générées par leur fonctionnement. Ces informations peuvent être classées en quatre catégories [Clemm, 2006] :

- les informations d'état opérationnel: ce type d'information reflète l'état courant des ressources physiques et logiques gérées. Ce sont des métriques de fonctionnement des ressources, elles sont surtout utilisées pour la supervision. Elles sont de nature dynamique (e.g. le nombre de paquets IP émis par une interface, le statut opérationnel d'un serveur etc.).
- les informations d'historique: ces informations proviennent de la compilation d'informations d'état opérationnel passées ainsi que des enregistrements d'évènements passés (e.g. évolution du nombre de paquets émis sur une période d'un semestre etc.).
- les informations de configuration physique: les informations de configuration physique sont soit statiques ou changent très peu. Elles ne peuvent pas être modifiées par les applications de gestion (e.g. l'adresse mac d'une station, le nombre de processeur d'un serveur etc.).

- les informations de configuration logique : ces informations font référence aux paramètres de configuration qui peuvent être modifiés par les administrateurs ou les applications de gestion. Elles permettent de contrôler l'entité gérée (e.g. l'adresse IP d'une machine, le nombre maximum de connexions simultanées à un service etc.). Leur modification est l'objet du processus de reconfiguration.

Le modèle organisationnel

Ce modèle décrit la répartition de rôles entre différentes entités de gestion. Le modèle le plus répandu est celui de l'*Agent*/*Manager*. L'agent et le manager sont des entités logicielles qui échangent des informations de gestion. Le *manager*, positionné sur les stations d'administration demande la réalisation d'opérations de gestion auprès des agents (e.g., consultation de l'état opérationnel, modification d'un paramètre de configuration). L'*agent*, installé sur les entités gérées répond aux sollicitations du manager et le notifie d'évènements particuliers.

Le modèle de communication

Ce modèle définit les protocoles de communication entre les entités de gestion. Ces protocoles permettent aux applications de gestion de recueillir ou de modifier des informations de gestion au niveau des entités gérées et aux entités gérées de notifier les applications de gestion d'évènements particuliers issus de leur fonctionnement.

Le modèle fonctionnel

Les activités de gestion sont regroupés en cinq domaines ou aires fonctionnels représentés sous l'acronyme FCAPS (Fault, Configuration, Accounting, Performance and Security) pour la gestion des fautes, de la configuration, de la comptabilité, de la performance et de la sécurité :

- La gestion des fautes englobe l'ensemble des activités qui permettent la détection, l'isolation et la correction d'anomalies de fonctionnement dans le réseau ou dans les services gérés dans un objectif de rétablissement ou d'assurance de service.

- La gestion de la comptabilité a pour objectif d'offrir les fonctionnalités qui permettent d'établir les charges et de déterminer les coûts relatifs à l'utilisation des ressources notamment pour la facturation ou la gestion des limites utilisateurs.

- La gestion de la performance regroupe les activités de collecte de données et d'analyse statistique, de gestion de trafic et de mesure de la qualité de service afin d'évaluer le comportement des ressources gérées, l'efficacité de leur opération et de maintenir leur performance globale.

- La gestion de la sécurité couvre un ensemble de fonctionnalités ayant trait à la protection des réseaux et services gérés contre les accès non-autorisés et les attaques.

- La gestion de la configuration regroupe les activités nécessaires à l'exécution d'opérations de spécification et de modification de configurations des entités gérées, afin qu'elles délivrent les services attendus. Elle inclut l'audit et l'inventaire des entités gérées, leur configuration initiale et démarrage ainsi que leur reconfiguration continue nécessaire au fonctionnement du système.

Nos travaux se concentrent sur cette aire fonctionnelle, nous nous intéressons en particulier au processus de configuration initiale et de reconfiguration.

– Boucle de gestion

Les activités de gestion sont traditionnellement représentées sous la forme d'une boucle de contrôle illustrée dans la Figure 1. Cette boucle de contrôle ou boucle de gestion se décompose en quatre phases, la Supervision, l'Analyse, la Décision et l'Exécution : le système géré est surveillé, cette supervision se manifeste par la collecte de diverses métriques caractérisant son fonctionnement. Ces métriques servent à analyser l'évolution du comportement opérationnel du système afin de détecter ou de prévenir d'éventuelles anomalies. Puis suit la phase de décision d'actions correctives ou proactives qui sont exécutées sur le système dans le but d'en améliorer le fonctionnement.

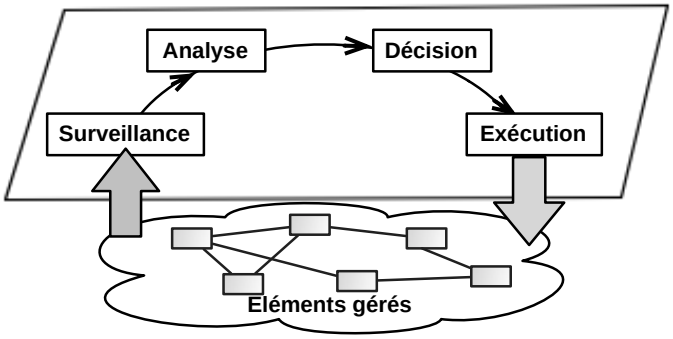

Figure 1 - Boucle de contrôle de la GRS

Boucle de gestion et reconfiguration

Dans la mise en œuvre de la reconfiguration, la phase de Décision est le lieu de l'identification des modifications à apporter (la nouvelle configuration). La phase d'exécution consiste en l'application effective de ces modifications sur le système géré.

I.2.1.2. La gestion autonome et adaptative de réseaux et de services

Définition 8 ***La gestion autonome et adaptative*** *a pour finalité la réduction de l'intervention humaine dans la gestion. Dans ce contexte, les systèmes de gestion sont dotés de capacités pour d'une part, mettre en oeuvre et maintenir par leurs propres moyens les objectifs de gestion des administrateurs humains, et d'autre part s'adapter et adapter les ressources gérées dynamiquement aux changements perçus de l'environnement opérationnel.*

– Vision de l'informatique autonomique d'IBM

Le paradigme de la gestion autonome et adaptative est inspiré du concept de l'informatique autonomique (*Autonomic Computing*) introduite par IBM, pour décrire une nouvelle génération d'entités informatiques intelligentes capable de s'auto-gérer [Horn, 2001] en fonction d'objectifs de haut niveau et de stimuli internes ou externes [Kephart & Chess, 2003]. Cette vision a pour finalité la maximisation de l'utilisation des entités informatiques tout en minimisant les coûts inhérents à leur complexité grâce à l'automatisation de leur gestion.

Selon ces auteurs, les entités informatiques autonomiques doivent exhiber quatre propriétés clés qui démontrent leur capacité d'auto-adaptation : leur capacité à s'auto-configurer, à s'auto-réparer, à s'auto-optimiser et à s'auto-protéger. Ces caractérisques sont collectivement appelées les propriétés self-CHOP.

– **Application au fonctionnement des systèmes de gestion autonomes**

Les systèmes de gestion autonomes doivent exhiber les propriétés self-CHOP (Figure 2), c'est-à-dire qu'ils doivent être capables d'adapter dynamiquement la configuration des ressources gérées (auto-configuration), de détecter, diagnostiquer et réparer leurs mauvais fonctionnements (auto-réparation), d'exécuter les opérations de gestion de manière efficiente sans pénaliser la performance des ressources gérées (auto-optimisation) et de prendre les mesures nécessaires pour protéger leur propre fonctionnement et celui des ressources gérées de menaces extérieures (auto-protection). Ces nouvelles propriétés sont équivalentes aux aires fonctionnelles FCAPS.

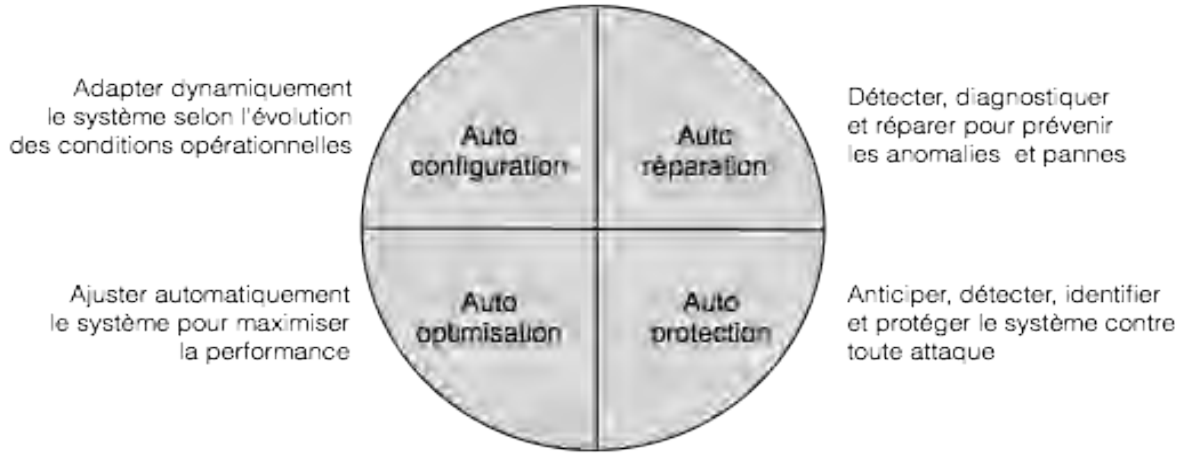

Figure 2 - Propriétés self-CHOP

I.2.2. La reconfiguration support des fonctions de gestion

I.2.2.1. La reconfiguration support des aires fonctionnelles de gestion

Ni les capacités self-CHOP ni les fonctions FCAPS n'existent de manière cloisonnée. Elles sont interdépendantes et concourent ensemble à l'efficacité et la robustesse opérationnelle du système géré. En l'occurrence, les capacités d'auto-réparation, d'auto-optimisation et d'auto-protection s'appuient sur la capacité d'auto-configuration, c'est-à-dire la capacité du système à se reconfigurer dynamiquement [Miller, 2005].

En effet, si la supervision permet d'observer l'état opérationnel du système géré, la capacité de définition de configurations et de reconfiguration permet de contrôler le comportement du système et est à la base de son dimensionnement, de la maintenance et de l'optimisation de son fonctionnement [RFC3512].

Dans la plupart des cas, la mise en œuvre des aires fonctionnelles s'appuie sur cette capacité :

- la gestion des fautes : la mise en œuvre de mesures préventives ou correctrices en cas de détection d'anomalies s'appuie sur la capacité de reconfiguration du système géré, par exemple, la modification de la taille maximum de la pile d'exécution de la mémoire d'un processus donné afin de corriger une mauvaise gestion.

- la gestion de la performance : l'optimisation du fonctionnement du système géré s'effectue via sa reconfiguration, par exemple, il est courant que la réallocation de ressources pour accroître la performance opérationnelle se fasse à travers l'ajout ou le retrait de clusters de serveurs.

- la gestion de la comptabilité : par exemple, la capacité de reconfiguration permet de tenir compte des ajouts d'équipements ou d'utilisateurs et de paramétrer les services fournis et leur facturation en fonction.
- la gestion de la sécurité : la reconfiguration est le support de la mise en œuvre des droits d'accès, la configuration des firewalls etc.

Dans le contexte de la gestion autonome et adaptative, la reconfiguration dynamique est donc un besoin transversal aux aires fonctionnelles. Que l'on se réfère au cadre des FCAPS ou à celui des self-CHOP, la reconfiguration dynamique est fonctionnellement et opérationnellement fondamentale pour la gestion des ressources complexes actuelles.

I.2.2.2. Reconfiguration dans les standards de gestion

I.2.2.2.1. Le modèle de gestion des environnements IP : SNMP

Le protocole de gestion SNMP (Simple Network Management Protocol) est le protocole le plus utilisé pour la gestion de réseaux dans les environnements TCP/IP. Il a été conçu par l'IETF pour faciliter un déploiement rapide de solutions de gestion dans ces environnements.

SNMP se fonde sur le modèle agent/manager et propose un ensemble de standards permettant de gérer à distance des réseaux contenant des équipements et ressources qui disposent d'agents SNMP. Il regroupe un ensemble de standards de gestion proposant à la fois un modèle informationnel et un modèle de communication :

- le modèle informationnel définit une structure de l'information de gestion (SMI) en vue de la représentation des objets gérés, et fournit un ensemble de bases d'information de gestion (MIB) pour le regroupement des objets gérés et de leurs attributs,
- le modèle de communication spécifie un protocole d'interaction entre les applications de gestion et les ressources gérées.

Le cadre architectural de SNMP est illustré dans la Figure 3.

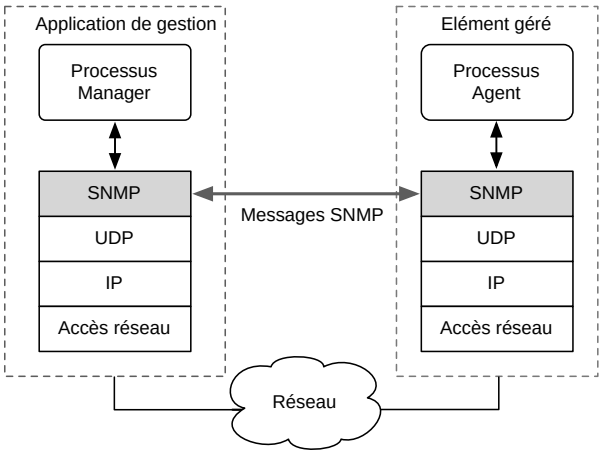

Figure 3 - Architecture SNMP

SNMP possède trois versions SNMP v1, SNMPv2c, SNMP v3. Il est surtout utilisé dans la supervision et l'analyse en support de la corrélation d'évènements et détection de fautes.

– Configuration et reconfiguration avec SNMP

SNMP définit trois types de primitives pour la gestion des ressources dont deux pour leur supervision (primitives «Get» et «Trap» respectivement pour le polling et la réception de notifi-

cations) et une primitive «Set» pour la modification de leur état de fonctionnement (écriture d'informations dans la MIB). La primitive «Set» est celle qui permet reconfigurer les ressources.

Son utilisation se traduit par l'envoi de messages SNMP «SetRequest» précisant la liste des instances d'objets (attributs) à modifier et leur nouvelle valeur. Cette modification est atomique c'est-à-dire qu'elle réussit pour toute liste de variables spécifiées ou échoue pour toutes. Un message de réponse «GetReponse» est renvoyé au manager avec la liste des instances modifiées et leur nouvelle valeur en cas de succès ou un code d'erreur le cas échéant.

SNMP n'a pourtant pas réussi à s'imposer comme standard pour la reconfiguration. En effet, l'utilisation de SNMP pour la reconfiguration présente plusieurs limites [RFC3535; Stallings, 1999]. Parmi celles-ci :

- les MIBs standards proposent très peu d'objets accessibles en écriture donc utiles pour la configuration et la reconfiguration. Les objets modifiables se retrouvent donc dans des modules de MIB propriétaires avec des taux de couverture différents.
- le côté procédural : le modèle de données et les interfaces de programmation sont bas-niveau et rendent difficile leur utilisation pratique pour une reconfiguration continue de haut niveau.
- la taille limitée des informations modifiables : la taille des groupes de paramètres de configurations modifiables est limitée à 64 Kilo-octets.
- il est impossible de revenir à une ancienne configuration : le schéma de nommage rend difficile la sauvegarde et le retour à une ancienne configuration. En outre, la représentation des objets gérés mélange les informations de configurations et celles d'état opérationnel, ce qui rend difficile leur identification.
- SNMP ne permet pas de découvrir les fonctionnalités supportées par un équipement.

Pour toutes ces raisons, peu de MIBs modifiables ont été développées et il est par exemple difficile, voire impossible, de reconfigurer entièrement un commutateur ou un routeur via SNMP.

I.2.2.2.2. L'architecture WBEM

WBEM (Web-Based Enterprise Management) est une initiative industrielle consistant en un ensemble de standards et de technologies de gestion développé dans le but d'unifier la gestion d'environnements informatiques distribués et multi-vendeurs.

L'initiative est standardisée par le DMTF (Distributed Management Task Force), un consortium regroupant un ensemble de partenaires industriels tels que IBM, Microsoft, Citrix, Redhat etc., créé pour piloter le développement et l'adoption de l'interopérabilité dans les standards de gestion.

L'objectif principal n'est donc pas de remplacer les standards et technologies existants mais plutôt de les intégrer dans le but de présenter aux applications des données et interfaces de gestion uniformes. Pour ce faire, l'approche WBEM propose de nouveaux modèles conceptuels standards dont un modèle informationnel, un modèle de communication et un modèle organisationnel et architectural.

Le modèle informationnel définit CIM (Common Information Model), un langage de modélisation orienté objet pour la représentation des informations de gestion. CIM se compose d'un méta-schéma qui spécifie la structure globale du langage et d'un ensemble de schémas standardisés pour différents types d'environnements et d'objectifs de gestion [CIM, 2012]. CIM

est disponible dans un format de représentation textuelle appelé MOF (Managed Object Format) et supporte également une représentation graphique à base de classes UML (Unified Modeling Language).

En ce qui concerne le modèle de communication, le langage XML est utilisé pour l'encodage de l'information et le protocole HTTP pour le transport des messages. Le cadre organisationnel et architectural s'éloigne du schéma classique Agent/Manager. Il se base sur le serveur WBEM et ses interfaces qui agissent comme un courtier entre les applications de gestion et les systèmes gérés.

Comme le montre la Figure 4, le serveur WBEM introduit de nouvelles entités : au cœur du serveur WBEM, se trouve le gestionnaire d'objets CIMOM (CIM Object Manager) qui offre une interface d'accès uniforme aux applications de gestion et une vue unifiée des ressources gérées.

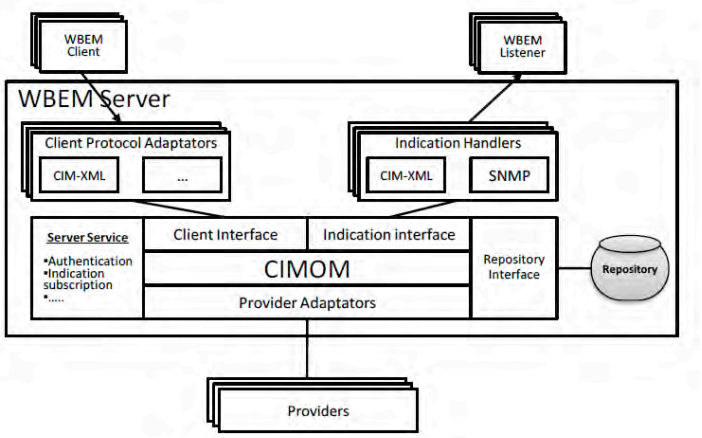

Figure 4 - Architecture WBEM

Le CIMOM fournit les interfaces d'accès :

- au «CIM Object Repository» équivalent d'une base d'information de gestion où sont stockées les informations de gestion des ressources gérées, modélisées avec CIM.
- aux «CIM Providers», des composants logiciels qui assure la liaison entre le CIMOM et les objets gérés. Typiquement les providers savent communiquer avec les agents installés sur les ressources. Il existe par exemple des providers SNMP, CMIP.
- aux «WBEM Clients» et «WBEM Listeners» correspondant aux applications de gestion elles-mêmes. Ceci leur permet de communiquer avec le CIMOM afin de manipuler les informations de gestion des ressources (*WBEM Clients*) et de recevoir des notifications (*WBEM Listeners*).

– Configuration et reconfiguration dans WBEM

De façon générale, les informations de configuration et d'état opérationnel sont mélangées au sein des classes et objets CIM. CIM fournit toutefois un patron de configuration basé sur la classe abstraite `CIM_SettingData` spécialement dédiée à la représentation de configurations. CIM définit également diverses associations telles que `CIM_ElementSettingData` et `CIM_SettingDefineState` qui relient la classe `CIM_SettingData` à `CIM_ManagedElement` permettant ainsi de séparer la configuration d'un élément géré de sa vue opérationnelle. La reconfiguration via CIM/WBEM passe par l'implémentation d'application-clientes qui manipulent les objets via des opérations WBEM sur des instances `CIM_SettingData`.

I.2.2.2.3. La gestion à base de politiques

La gestion à base de politiques est un paradigme de gestion qui sépare les règles gouvernant le comportement du système géré des fonctionnalités que le système offre. C'est un paradigme qui reste d'actualité, il est présent au cœur des nouvelles approches de gestion.

Une politique est un ensemble de règles utilisé pour administrer, contrôler, gérer le comportement d'un ou de plusieurs éléments gérés. [Sloman, 1994].

On distingue différents types de politiques selon les objectifs de gestion [Boutaba et Aib, 2007] :

- les politiques de sécurité et de contrôle d'accès pour vérifier les droits et auditer les ressources,
- les politiques de configuration qui gèrent l'initialisation et le contrôle de la configuration des ressources en fonction de leur utilisation,
- les politiques de service pour décrire les capacités des ressources par exemple les politiques de routage.

L'IETF a proposé un cadre de gestion à base de politiques [RFC2753] qui identifie un modèle organisationnel et un modèle de communication. Le modèle organisationnel est constitué de deux composants de base, le PDP (Policy Decision Point) et le PEP (Policy Enforcement Point). Le modèle de communication fournit un protocole de communication entre ces deux composants.

Les politiques sont centralisées dans une base de données (Policy Repository) qui offre une interface-utilisateur de spécification, d'édition et d'administration des politiques de gestion (console de gestion de politiques). Ce cadre organisationnel est présenté dans la Figure 5.

- Le PDP est le serveur de politiques. C'est une entité qui prend des décisions en se basant sur les politiques stockées dans la base de politiques. Il gère le comportement des ressources gérées en décidant des politiques à appliquer sur quels équipements réseaux.
- Le PEP est une entité logicielle implantée sur les équipements et dont le rôle est d'appliquer les décisions de politiques de gestion que le PDP lui adresse. Il met en oeuvre ces politiques suivant un schéma de règles «si condition alors action» ou bien « à tel moment faire action» traduisant ainsi les politiques en paramètres de configuration spécifiques à l'équipement.

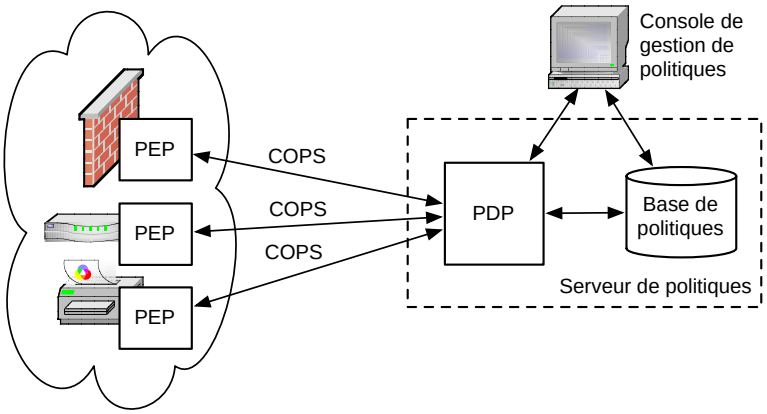

Figure 5 - Architecture de gestion à base de politiques

– **Configuration et reconfiguration avec les politiques**

Un protocole standard d'échanges de politiques entre un serveur de politiques (PDP) et ses clients (PEP) nommé COPS-PR a été publié en 2000 par l'IETF [RFC2748]. Il est dédié à la reconfiguration dans les environnements TCP/IP. Dans ce protocole, chaque client (PEP) maintient une base locale de politiques appelée PIB (Policy Information Base) où sont stockées les politiques de configurations locales.

Ce protocole résout la plupart des limites de SNMP en matière de configuration (les états entre gestionnaires et équipements sont synchronisés, les capacités d'un équipement peuvent être découvertes, la reconfiguration peut se faire de manière transactionnelle). Cependant la récupération d'informations de configuration et le retour à une ancienne configuration demeurent difficiles. Le système de nommage du PIB et le choix de l'encodage binaire des données de gestion rendent difficile l'automatisation de leur manipulation. COPS-PR a été peu déployé et son utilisation n'est plus recommandée depuis 2004 [RFC6632].

I.2.2.2.4. L'architecture JMX

JMX (Java Management Extensions) est une initiative industrielle qui s'inscrit dans le cadre du *Java Community Process* (JCP) permettant à des tiers de participer à l'extension de la technologie Java. JMX a été défini pour répondre aux besoins de gestion d'applications développées en Java. Il supporte également la gestion d'équipements, réseaux et services qui n'intègrent pas forcément cette technologie. Nous présentons ici l'architecture générale de JMX. Une étude plus complète peut être trouvée dans le chapitre 6 de [Festor & Andrey, 2003].

L'architecture JMX est composée de trois niveaux définis selon le modèle organisationnel *Agent/Manager* (Figure 6) :

- le niveau d'instrumentation : ce niveau est le plus bas, c'est-à-dire le plus proche de l'objet géré. Il repose sur le concept de MBean (Managed Bean), équivalent au concept d'objet géré. Un MBean est une classe Java respectant un certain patron de programmation défini dans la spécification JMX. Ce niveau permet d'instrumenter les ressources gérées et fournit un modèle pour l'envoi et la réception de notifications (JMX notifications) semblables au mécanisme des *traps* SNMP.

- le niveau agent : le niveau du milieu encore appelé le niveau agent fournit un conteneur de Mbeans ou serveur de Mbeans (Mbean server) qui permet d'accéder aux ressources gérées. Il fournit également des services (*agent services*) qui permettent la manipulation des Mbeans (création, destruction, modification d'attributs etc.). La combinaison d'une instance de serveur de MBeans, des Mbeans qu'il contient et des services-agent utilisés au sein d'une machine virtuelle Java (JVM) unique constitue un agent JMX.

- le niveau des services distribués : c'est le troisième niveau de l'architecture JMX. Ce niveau contient un intergiciel qui relie les agents JMX à des applications de gestion. L'intergiciel est divisé en deux catégories : les adaptateurs de protocole et les connecteurs. Les adaptateurs de protocole assurent la communication entre des protocoles spécifiques (par exemple HTTP) et les agents JMX, c'est le cas par exemple d'une application web de gestion qui se connecte à un ou plusieurs agents JMX. Les connecteurs respectent une architecture client-serveur. Ils permettent à un client (application de gestion) de faire des appels de méthodes à distances sur un serveur Mbean (agent JMX), par exemple avec Java RMI (Remote Method Invocation).

De manière transversale à cette architecture, JMX propose un ensemble d'interfaces de programmation pour les développeurs d'agents JMX et de ressources gérées MBeans pour l'intégration d'approches existantes (SNMP, CMIP, WBEM).

– Configuration et reconfiguration dans JMX

Bien que JMX ne fournisse pas de modèle informationnel standard, les MBeans peuvent être considérés comme les modèles d'information de gestion dans les environnements JMX. Un MBean déclare des attributs, des méthodes *getter* et *setter* pour leur lecture et écriture, et des méthodes correspondant aux actions pouvant être effectuées sur la ressource gérée réelle (par exemple la démarrer).

Un paramètre de configuration connu a priori est un attribut de classe suivant une convention de nommage particulière et disposant de méthodes *getter/setter* correspondantes. Pour la reconfiguration de dynamique, JMX propose une interface de gestion, découverte à l'exécution via deux types de méthodes d'accès aux paramètres de configuration : `void setAttribute(Attribute attribute)`qui permet de modifier un seul attribut à la fois et `void setAttributes(AttributeList attributes)` qui permet de modifier un ensemble d'attributs simultanément.

A l'instar des classes Java, les Mbeans peuvent être composés et reliés entre eux par des dépendances.

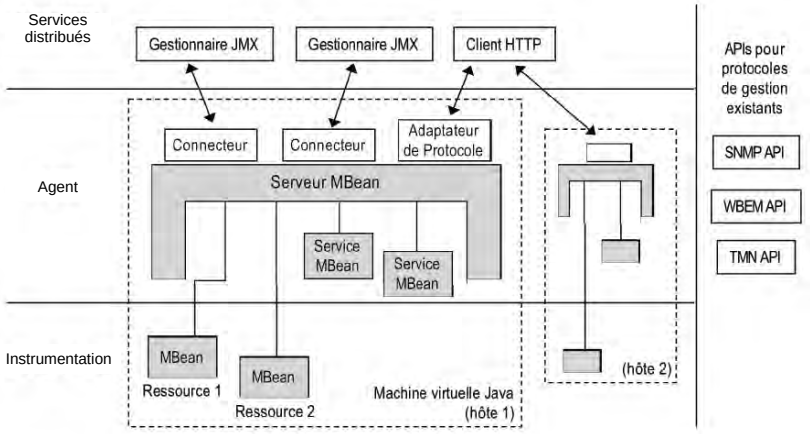

Figure 6 - Architecture JMX

I.2.2.2.5. Les standards NETCONF/YANG

– Le protocole NETCONF

Des protocoles de gestion présentés jusqu'alors, le protocole NETCONF est celui le plus récent. Standardisé par l'IETF en 2006 puis revu en 2011 [RFC6241], il vise la gestion de la configuration dans les équipements réseaux. Il fournit des mécanismes pour installer, manipuler et supprimer des configurations d'équipements réseaux.

NETCONF sépare les informations de gestion en deux classes distinctes : les informations de configuration et les informations d'état. Les informations de configuration sont les informations modifiables nécessaires à la configuration initiale et à la reconfiguration du système. Les informations d'état sont les métriques collectées du fonctionnement du système. Ce sont des informations en lecture seule.

Le modèle organisationnel sous-jacent est constitué d'un serveur NETCONF (équivalent traditionnel de l'agent et d'un client NETCONF (équivalent traditionnel du manager) qui communiquent par appel de procédures à distance. Le serveur NETCONF une entité logicielle installée sur les équipements compatibles. Le client NETCONF est une entité logicielle qui fait partie de l'application de gestion.

NETCONF est défini suivant un modèle de quatre couches illustrées dans la Figure 7 :

- la couche Transport (sécurisé) spécifie le protocole de transport, d'échange de l'information de configuration entre client et serveur. Il fournit les mécanismes nécessaires à la sécurisation des échanges.
- la couche Messages définit le format de communication qui se fait via des mécanismes d'appels de procédures à distances et de notifications.
- la couche Opérations décrit les primitives protocolaires de manipulation des informations de configuration. On peut récupérer une configuration, la modifier, l'appliquer etc. Ces primitives sont invoquées sous forme de méthodes d'appel de procédures à distances (RPC).
- la couche de Données donne accès aux données de configurations.

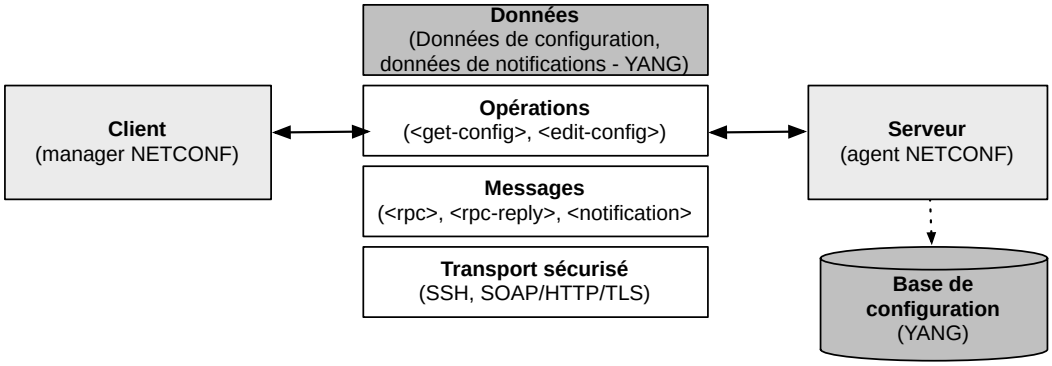

Figure 7 - Architecture du protocole NETCONF

Le protocole NETCONF utilise le format XML pour la représentation des données de configuration ainsi que pour les messages échangés. Il fournit un mécanisme d'extension avec la notion de capacités.

Une capacité NETCONF est un ensemble d'opérations additionnelles dont la définition permet d'augmenter la spécification de base de NETCONF. Chaque capacité est identifiée par une adresse URI (Uniform Resource Identifier) pour permettre sa découverte par le client.

NETCONF introduit un modèle informationnel via les bases de configurations NETCONF (NETCONF datastores). Une base de configuration représente l'ensemble minimal d'information de configuration requis pour amener un équipement d'un état initial à un état opérationnel désiré. NETCONF distingue trois types de bases de configuration :

- la base de la configuration courante qui représente la configuration actuellement déployée sur l'équipement. Sa présence est obligatoire.
- la base de configuration de démarrage qui contient la configuration à utiliser au prochain redémarrage de l'équipement. L'implémentation de cette base est optionnelle.
- la base de configuration candidate qui contient une configuration potentielle applicable lors d'une reconfiguration. L'implémentation de cette base est optionnelle.

Ce modèle informationnel repose sur un langage de description de données appelé YANG standardisé par l'IETF en 2010 [RFC6020].

– YANG, le langage de modélisation de données de NETCONF

YANG est le langage de modélisation des données du protocole NETCONF. Il permet d'avoir une représentation complète de toutes les données échangées entre un client NETCONF (application de gestion) et un serveur NETCONF (équipement réseau). YANG peut être utilisé pour modéliser les données de configuration et d'état opérationnel du réseau, les signatures des appels de procédure à distance ainsi que les notifications NETCONF.

C'est un langage modulaire. Les spécifications YANG sont organisées de façon hiérarchique en modules et sous-modules dans un format textuel. Un module YANG est structuré sous forme d'arbre (ensemble de nœuds) : chaque nœud a un nom et soit une valeur, soit un sous-ensemble de nœuds-fils. Les modules et sous-modules contiennent les définitions de types de données et d'opérations.

Les modèles YANG peuvent définir des contraintes à vérifier sur les données échangées pour garantir leur validité. Cette vérification est menée soit au niveau du serveur (cas recommandé), mais elle peut se faire au niveau du client. Elle concerne prioritairement la vérification de la syntaxe des configurations, mais également le respect des types et la présence de nœuds obligatoires.

– Configuration et Reconfiguration avec YANG et NETCONF

Pour reconfigurer un équipement réseau, NETCONF permet à une application de gestion (client NETCONF) d'appeler un ensemble d'opérations de modification sur sa configuration courante ou de lui envoyer une nouvelle configuration décrite en YANG à appliquer. Ces décisions de reconfiguration doivent être incluses dans une requête `<edit-config>` envoyée à l'équipement réseau. L'équipement réseau exécute ces modifications et renvoie un message de confirmation en cas de succès ou un message d'erreur en cas d'échec.

I.2.2.3. Autre approche de reconfiguration : Interface en ligne de commandes (CLI)

Bien que SNMP soit l'approche la plus populaire dans la gestion des environnements TCP/IP, la gestion par interface de ligne de commandes (CLI) est très utilisée dans l'administration des équipements réseaux. Par exemple, des interfaces de gestion par ligne de commandes sont systématiquement fournies sur la majorité des routeurs et commutateurs (plus de 70% du marché des équipements réseaux [Lindblad, 2013]).

Cela est dû à la faiblesse de SNMP en matière de gestion de configuration, notamment au niveau de la gestion des transactions et de la sécurité des modifications [Pavlou, 2007], et parce que SNMP ne couvre pas forcément toutes les fonctionnalités offertes par un équipement.

Il n'y a pas à proprement dit de CLI standard mais plutôt des implémentations qui diffèrent selon les vendeurs ou les systèmes d'exploitation. Néanmoins, les CLIs partagent les mêmes principes de base, ils sont pensés pour être des interfaces utilisateurs permettant des interactions directes et simples entre l'administrateur humain et un équipement donné.

A la différence de SNMP où la ressource réelle est représentée comme une collection d'objets gérés, la ressource gérée est considérée comme une boîte noire à laquelle un administrateur peut envoyer une séquence de commandes visant à la configurer ou à récupérer des informations d'état courant dans une dynamique de requête et de réponse.

La gestion par ligne de commandes est essentiellement une gestion manuelle. Elle est difficilement automatisable car elle nécessite une granularité fine spécifique à un équipement donné, ainsi qu'une détermination de la bonne séquence de commandes à exécuter. De plus, les interfaces CLI étant propriétaires, elles ne peuvent pas être généralisées pour automatiser les processus de reconfiguration dans des environnements hétérogènes.

La Figure 8 montre un exemple de configuration d'un routeur en ligne de commandes.

```
Router# configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
Router(config)# interface fastethernet 5/4
Router(config-if)# ip address 172.20.52.106 255.255.255.248
Router(config-if)# no shutdown
Router(config-if)# end
Router#
```

Figure 8 - Configuration d'une interface FastEthernet en ligne de commandes (Cisco)

Des défis majeurs encadrent la reconfiguration dynamique, notamment la spécification d'objectifs de fonctionnement, la détection de changements non désirés et la mise en place de mécanismes de reconfiguration pouvant passer à l'échelle. En l'occurrence, une problématique importante pour la généralisation de la reconfiguration dynamique réside dans l'assurance qu'elle s'opère de façon effective et préserve la viabilité du système.

I.3. Impact des erreurs de configuration sur le fonctionnement des systèmes

Cette section souligne la nécessité d'encadrer les changements (autonomes) de configurations. En effet la reconfiguration présente des risques potentiels d'erreurs de configuration qui impactent la disponibilité et l'utilisation des réseaux et systèmes complexes.

I.3.1. Erreurs de configuration

Les erreurs de configuration sont la première cause de pannes et de dysfonctionnements dans la gestion des réseaux et systèmes configurables avec des incidences en matière de sûreté de fonctionnement, de qualité de service et de continuité de service [Patterson et al., 2002 ; JuniperNetworks, 2008].
En nous appuyant sur la taxonomie des fautes, des erreurs et des pannes de [Avizienis et al., 2004] et de l'étude des erreurs de configuration de [Patterson, 2002], nous définissons une erreur de configuration comme suit :

Définition 9 ***Erreur de configuration*** *: une erreur de configuration est une violation constatée de valeurs de paramètres de configuration qui aboutit à deux grands types de dysfonctionnements au niveau du système géré : la rupture de service ou le fonctionnement en mode dégradé.*

De façon générale, les erreurs de configuration sont dues soit à des spécifications erronées de valeurs de paramètres de configuration d'un composant, soit à des mauvaises coordinations de dépendances de valeurs de paramètres de configuration entre composants. Ces erreurs sont provoquées lors de reconfigurations paramétriques et structurelles. Elles représentent la majorité des dysfonctionnements observés dans la gestion de réseaux et systèmes complexes Oppenheimer et al., 2003].

Une étude récente de systèmes informatiques (architectures 3-tier open source, systèmes de stockage de données commerciaux) a montré que ce type d'erreurs de configurations représente 70% à 85.5% des pannes observées [Yin et al., 2011]. Dans cette même étude, entre 14% et 30% des erreurs proviennent d'incompatibilités logicielles (incompatibilités des versions) et d'erreurs de configuration de composants, par exemple une configuration référence des composants manquants ou bien une configuration référence un composant dont les ressources courantes sont insuffisantes.

En empruntant la terminologie adoptée dans cette étude, nous catégorisons les erreurs de configuration en deux grands groupes : les erreurs illégales et les erreurs légales.

Définition 10 ***Erreur de configuration illégale*** *: une erreur de configuration illégale viole des contraintes explicites ou implicites obligatoires de format ou syntaxique, de consistance interne de valeurs de configurations ou de consistance vis-à-vis d'autres valeurs de configurations. Ce sont des erreurs qui portent sur la structure de l'information de configuration. Par exemple une valeur de configuration ne respecte pas une certaine limite.*

Définition 11 ***Erreur de configuration légale*** *: à la base, une erreur de configuration légale n'est pas due à une violation de contraintes obligatoires mais provient d'inconsistance créées par les conditions environnementales courantes. Par exemple, une valeur de configuration pénalise la performance. Ces erreurs sont difficilement décelables à moins que des objectifs de service soient spécifiés et puissent être évalués sur les configurations. Elles sont généralement dues aux conditions opérationnelles telles que le passage à l'échelle, non prévisible en phase de conception.*

Les solutions générales pour éviter ces erreurs consistent en la mise en place de cadre de vérification de configurations, par exemple le fait d'exprimer et vérifier des contraintes sur les configurations résoudrait la majorité des erreurs de configuration observées [Oppenheimer et al., 2003 ; Stawowski, 2011].

I.3.2. Impact des erreurs de configuration sur les aires fonctionnelles

Les erreurs de configuration impactent les aires fonctionnelles, engendrant des coûts très élevés. Cet impact est souligné dans [Patterson, 2002] où après avoir étudié les pannes dans des réseaux de télécommunications et des services Internet, les auteurs proposent un cadre de calcul des coûts qu'elles engendrent. [JuniperNetworks, 2008] estime que le coût engendré par les erreurs de configurations d'équipements réseau équivaut à 3,6% des revenus annuels des entreprises voire jusqu'à 80% des budgets IT.

Les erreurs de configuration aboutissent soit à des ruptures de service, soit à des fonctionnements dégradés, par exemple le service est lent ou limité, le service présente des vulnérabilités en matière de sécurité.

La sévérité de ces erreurs peut être caractérisée par leur implication en termes de disponibilité (durée de la panne), sûreté (vie humaine en jeu), confidentialité (révélation d'informations sensibles), intégrité (corruption de données de services) [Oppenheimer et al., 2003 ; Avizienis et al., 2004]. L'impact principal est celui de la non-disponibilité de certaines fonctionnalités et de la dégradation des performances. Les erreurs de configuration rendent également le système vulnérable en matière de sécurité. Plusieurs études donnent des exemples d'impacts d'erreurs de configuration :

- [Yin et al., 2011] dans une étude de serveurs et de systèmes de stockage de données (serveur CentOS, MySQL, Apache, OpenLDAP) en service, montrent que des erreurs de configuration sont à l'origine de 16,1% à 47.3% des indisponibilités et des dégradations sévères de services. 38% à 53% de ces erreurs sont des erreurs illégales dans la mesure où elles violent explicitement des contraintes de format, de valeur et de structure de la configuration. 46% à 61.9% sont des erreurs légales.

- Dans une étude de serveurs DNS, [Pappas et al., 2004], montrent que 15% des zones DNS présentent des erreurs dans la spécification des dépendances de configurations entre serveurs parent et fils. Ces erreurs ont un impact non seulement sur la disponibilité des zones mais aussi sur la performance des temps de résolution de noms qui passent d'une moyenne de 100 millisecondes à 3 secondes voire 30 secondes dans les pires cas.

- [Mahajan et al., 2002], dans une étude sur la fréquence des erreurs de configuration dans les routeurs BGP (Border Gateway Protocol), montrent que les erreurs de configuration de routeurs BGP sont quotidiennes et correspondent entre 0.2% et 1% de la taille totale de leur table de routage. Ces erreurs de configuration de routeurs BGP entraînent pour la plupart des suppressions d'annonces de routage et des surcharges de routeurs. Ces erreurs provoquent également des ruptures de connectivité et des inconsistances dans les contrôles d'accès.

Les erreurs de configuration vont à l'encontre de l'objectif de la reconfiguration dynamique, qui est de maximiser l'utilisabilité du système. Vu les usages critiques que supportent les systèmes complexes actuels, la contingence de telles erreurs de configuration ne peut plus être tolérée. De plus, il est crucial de disposer de méthodes permettant de prévenir les erreurs de configuration et de garantir que les changements dynamiques de configurations préservent la survie du système.

I.4. Conclusion

L'objectif de ce premier chapitre était de présenter les concepts généraux autour de la reconfiguration de systèmes complexes. Nous avons vu l'importance de cette capacité dans la mise en œuvre des aires fonctionnelles de la gestion.

Face à la complexité croissante des systèmes et de leurs usages, la synergie de toute la communauté de gestion se tourne vers l'autonomie et l'adaptation. Dans cette vision, les systèmes gérés sont dotés de capacités d'observation, de décision afin qu'ils puissent se coordonner et se réguler notamment par la capacité de reconfiguration dynamique. Un besoin fondamental pour l'exploitation effective de cette capacité est de disposer de moyens garantissant que les mécanismes de reconfiguration dynamique opèrent de manière effective et sûre. Ceci est particulièrement primordial dans les systèmes à fortes contraintes et dont les usages sont critiques.

Le prochain chapitre se concentre sur les besoins de vérification de reconfigurations dynamiques dans la GRS. Les mécanismes de reconfiguration que nous envisageons sont ceux qui procèdent par décision de nouvelles configurations ou configurations candidates avant leur application.

Chapitre 2

Vérification de reconfiguration dynamique dans la GRS : exigences et état de l'art

L'objectif de ce chapitre est de définir la problématique exacte de nos travaux autour de la vérification de configurations de systèmes communicants complexes reconfigurables. Traditionnellement, la vérification de configurations se fait hors ligne et permet de contrôler la correction et la consistance interne des configurations. Les caractéristiques des mécanismes de reconfiguration dynamique dans ces systèmes bouleversent les besoins fondamentaux de vérification de configurations. La vérification classique de configurations doit évoluer vers une vérification en ligne prenant en compte le contexte d'exécution du système géré. Nous présentons les nouvelles exigences que doit supporter cette vérification en ligne et étudions les réponses de l'état de l'art à ce propos.

II.1. **Vérification de configurations de réseaux et systèmes complexes**

II.1.1. **Vérification et Validation: généralités**

La vérification et la validation (V&V) sont deux processus distincts et indépendants faisant partie de l'assurance qualité relative à un produit (matériel, logiciel, services) telle qu'elle est définie dans la norme ISO 9000 [ISO9000]. L'objectif fondamental de la V&V est de garantir que le produit répond aux spécifications de conception et aux exigences de son utilisation.

– Définitions

La vérification et la validation sont deux processus importants en génie logiciel, elles interviennent habituellement dans le cycle de vie de développement d'un système informatique. Pour les besoins de nos travaux, nous retenons les définitions suivantes de la vérification et de la validation :

Définition 12 ***Vérification*** *: la vérification est «le processus d'examen du résultat d'une activité en vue de déterminer sa conformité aux exigences fixées pour ladite activité» [ISO9000]. La vérification répond à la question «construisons-nous correctement le système ?» [Boehm, 1984]. l'objectif recherché étant de prouver que la conception du système est conforme aux descriptions et exigences conceptuelles fonctionnelles comme non fonctionnelles en entrée.*

Définition 13 ***Validation :*** *la validation est «le processus d'examen d'un produit en vue de déterminer sa conformité aux besoins utilisateurs» [ISO9000]. La validation répond donc à la question «construisons-nous le bon système» [Boehm, 1984], l'objectif étant d'évaluer le degré auquel le système répond aux exigences de l'utilisation qui doit en être faite.*

– **Techniques de vérification et de validation**

On peut regrouper les techniques de V&V en deux grandes catégories : les techniques statiques et les techniques dynamiques.

- Les techniques statiques se déroulent sans exécution du système. Elles consistent en des inspections et en des analyses statiques du système (ou de ses divers spécifications, modèles et programmes).
- Les techniques dynamiques nécessitent l'exécution du système. Elles consistent généralement en des tests (tests fonctionnels structurels et d'acceptation).

Lorsque la vérification et la validation sont mises en oeuvre via les méthodes formelles, on parle de vérification et de validation formelles. Les méthodes formelles sont des techniques mathématiques, souvent outillées, pour le développement de systèmes informatiques [Woodcock et al., 2009]. Elles consistent en l'usage de modèles mathématiques pour l'analyse et la vérification des différentes étapes du cycle de développement logiciel.

La vérification et la validation formelles s'appuient sur des spécifications de propriétés décrites à l'aide de langages mathématiques. Des mécanismes d'inférence ou de preuve associés permettent de raisonner sur leur satisfaction. On peut citer des exemples de méthodes formelles tels que la notation Z [Spivey, 1989], la méthode VDM (Vienna Development Method) [Jones, 1990], la méthode B [Abrial, 1996] ou encore Alloy [Jackson, 2002].

– **Vérification et Validation dans le cycle de vie du système**

La vérification et la validation interviennent avant le déploiement du système durant les phases de conception de développement et de tests. La vérification intervient à chaque étape du cycle de développement du système. Elle culmine à la phase de tests où le système est exercé avec des jeux de données de tests avant sa mise en opération. Quant à la validation, elle se situe généralement en fin de cycle. Elle est laissée aux utilisateurs finaux qui vont évaluer l'adéquation du logiciel avec son utilisation et valider sa mise en service.

Pouvoir vérifier des configurations candidates avant leur application (par rapport à des exigences fixées qui contrôleraient leur correction et cohérence) permettrait d'encadrer leurs changements dynamiques en prévenant d'éventuelles erreurs de configuration. La problématique exacte de nos travaux consiste donc à comprendre et à caractériser la mise en œuvre d'une vérification de configurations dans le cadre de la reconfiguration dynamique de systèmes communicants complexes.

II.1.2. **Vérification de configurations dans les systèmes reconfigurables**

Traditionnellement, la vérification de configurations dans les systèmes gérés reconfigurables se fait hors ligne suivant deux cas de figure : le premier, par rapport au cycle de vie de leur développement et le second, par rapport au cycle de vie de leur gestion.

- Dans le premier cas, la vérification est effectuée en phase de conception ou en général, avant la mise en opération du système, par exemple la vérification de la configuration initiale ou la vérification de diverses configurations en phase de tests.
- Dans le second cas, la vérification est effectuée pendant la phase d'exploitation en support de reconfigurations statiques : lorsqu'un besoin de reconfiguration est détecté, l'administrateur décide d'une nouvelle configuration, la vérifie, puis l'applique, par

exemple, vérifier le fichier de configuration d'un serveur web Apache. Cette vérification repose habituellement sur des contraintes codées en dur [Agrawal et al., 2004].

A la différence de la vérification du système qui s'opère en phase de développement (Figure 9 - ①), la vérification de configurations intervient plutôt dans la phase d'utilisation du système, en support à son déploiement et à son exploitation (Figure 9 - ②).

Cette vérification contrôle la conformité structurelle des configurations. Elle est semblable à la vérification de contraintes d'intégrité dans les bases de données.

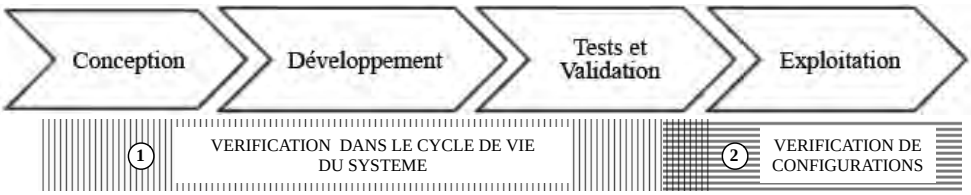

Figure 9 – Vérification de configurations dans le cycle de vie d'un système reconfigurable

Définition 14 ***Vérification de la conformité structurelle d'une configuration*** *: la vérification de la conformité structurelle d'une configuration évalue la correction des valeurs de configuration vis-à-vis du respect des types de données, des dépendances entre entités gérées et des propriétés fonctionnelles et non-fonctionnelles définies par l'utilisateur. Cette vérification vise à déterminer la correction de la structure et de la composition des paramètres de configuration en termes de valeurs autorisées.*

Cette vérification consiste pour l'essentiel en l'analyse de la correction syntaxique, du respect de types de données et de la consistance des informations de configuration vis-à-vis d'invariants structurels. Par exemple, vérifier que le numéro de port d'un service est compris dans un intervalle autorisé ou vérifier la présence d'une valeur IP dans une table de routage. Elle permet d'éliminer la plupart des erreurs illégales (Section I.3.1).

Par définition, la vérification de configurations dans le cadre d'une reconfiguration dynamique doit se faire en cours d'exécution, en effet les changements de configuration sont décidées et appliquées *at runtime*. Cette vérification introduit de nouvelles exigences à considérer, en l'occurrence la vérification de configurations doit prendre en compte le contexte d'exécution du système.

II.2. Exigences de vérification de configurations pour la reconfiguration dynamique

La nature des mécanismes de reconfiguration dynamique introduit de nouvelles exigences de vérification de configurations à considérer : la vérification de configurations doit se faire en ligne et doit prendre en compte le contexte d'exécution des éléments gérés.

II.2.1. Prise en compte du contexte d'exécution du système géré

Le contexte d'exécution correspond à l'ensemble des éléments qui influencent le système lors de son exécution. Ce contexte prend en compte à la fois l'environnement technologique (matériel et logiciel) mais aussi les attentes des utilisateurs. Le contexte d'exécution peut être très dynamique notamment dans les systèmes ouverts [Besseron, 2010]. C'est un paramètre essentiel à prendre en compte dans la vérification de configurations en ligne.

L'état courant et la dynamicité des environnements gérés ont des influences sur la mise en œuvre de cette vérification.

II.2.1.1. Prise en compte de l'état opérationnel courant

La modification dynamique de configurations est dépendante des évolutions des conditions opérationnelles du système géré. Par exemple, dans le cadre d'une reconfiguration dynamique pour la migration d'un serveur sur une nouvelle machine-hôte, il ne suffit plus de vérifier que l'adresse de l'hôte est conforme au format attendu, il faut également vérifier que la machine hôte fonctionne (c'est-à-dire qu'elle est dans un état «actif»).

La vérification doit donc prendre en compte l'état opérationnel courant dans l'appréciation de la conformité d'une configuration donnée. Cette vérification détermine si l'application d'une configuration ne viole pas l'état courant du système. Nous la nommons «vérification de l'applicabilité opérationnelle».

Définition 15 ***Vérification de l'applicabilité opérationnelle d'une configuration*** *: la vérification de l'applicabilité opérationnelle d'une configuration évalue la cohérence des valeurs de configuration vis-à-vis de conditions opérationnelles courantes.*

La vérification de l'applicabilité opérationnelle de configurations répond notamment à la problématique des erreurs légales (Section I.3.1). Elle complète celle de la conformité structurelle et permet ainsi une vérification en ligne prenant en compte à la fois l'intégrité des configurations vis-à-vis de leur structure et de leur composition, et l'état courant du système géré. Ces deux types de vérification (Définition 14 et Définition 15) nous permettent d'introduire le concept de vérification opérationnelle définie comme suit :

Définition 16 ***Vérification opérationnelle d'une configuration*** *: la vérification opérationnelle d'une configuration est la vérification en ligne de la conformité structurelle et de l'applicabilité opérationnelle de cette configuration.*

Au regard de l'état de l'art sur les erreurs potentielles de configuration, selon nous, une capacité de vérification opérationnelle contribuerait à garantir que les changements dynamiques de configurations sont corrects et qu'ils préservent la viabilité du système.

II.2.1.2. Prise en compte de la dynamicité

Un critère inhérent au fonctionnement des systèmes complexes actuels est la notion de dynamicité tant à l'évolution imprévisible de l'environnement qu'aux changements à apporter pour répondre à ces évolutions. En effet, les éléments gérés supportent des usages dynamiques, ils sont ajoutés, retirés, déplacés à la volée selon des objectifs de gestion eux aussi évolutifs.

– Une dynamicité des modifications de configurations

Cette dynamicité se traduit par une modification plus fréquente de configurations pour adapter le comportement des éléments gérés [RFC3512]. Par exemple, les types de reconfiguration dynamiques ont une incidence sur le périmètre de vérification : une reconfiguration paramétrique nécessite de vérifier les paramètres de configuration d'un composant, une reconfiguration structurelle demande le contrôle de la composition et des dépendances entre configurations.

– **Une dynamicité au niveau des objectifs de gestion**

Cette dynamicité se traduit également par des objectifs de gestion qui peuvent changer en phase d'exécution. Ces objectifs peuvent couvrir tout le cycle de vie d'exploitation du système ou être ponctuels par rapport à des scénarii spécifiques d'utilisation. Ils peuvent préexister à la mise en service du système ou être introduits pendant son opération [Cheng et al., 2009]. Ils induisent des propriétés à vérifier au niveau des configurations.

La vérification de configurations doit s'adapter à ces nouvelles caractéristiques en étant elle-même flexible et évolutive. En effet, elle doit pouvoir se moduler au gré des observations de l'environnement et des besoins d'adaptation. Elle doit pouvoir être adaptable pour ne concerner que les propriétés et les données de configurations à vérifier. En outre, les propriétés à vérifier doivent pouvoir évoluer si besoin, c'est-à-dire créées, modifiées, supprimées en cours d'exécution pour refléter les évolutions observées du système géré et maintenir la vérification au plus près des scénarii d'usage.

II.2.2. **Prise en compte de l'hétérogénéité des systèmes et de leurs usages**

La complexité croissante des éléments gérés traduit celle de ses configurations marquées par une extrême hétérogénéité à différents points de vue qu'il faut prendre en compte [Anderson & Smith, 2005]. Cette hétérogénéité est marquée par :

- la nature des paramètres de configuration : les paramètres de configuration ne sont pas de même nature, de même importance et n'évoluent pas au même rythme. Un paramètre de configuration peut être fonctionnel, il est nécessaire pour rendre le système fonctionnel et utilisable, ou non-fonctionnel, il n'en conditionne que l'utilisation efficiente comme la performance ou la sécurité. Certains paramètres sont plutôt statiques et d'autres plus dynamiques.

- la présence de plusieurs standards de gestion à prendre en compte : ces standards influent spécialement sur la représentation et la structure des configurations, l'accès à leurs valeurs. Les paramètres de configuration sont disponibles dans des syntaxes et formats différents, un même paramètre de configuration est représenté dans un format différent, peut posséder un type de données différent suivant l'élément géré.

- la diversité des propriétés à vérifier : l'hétérogénéité des domaines applicatifs induit celle de la nature des propriétés à vérifier. Ces propriétés peuvent soient couvrir le cycle de vie d'utilisation du système, soit présenter une évolution plus dynamique (situation de mobilité). Elles peuvent être des propriétés fonctionnelles ou «contraintes dures» ou des propriétés non-fonctionnelles ou «contraintes molles» selon la dénomination dans [Burgess & Couch, 2006]. Les contraintes dures sont des contraintes dont la vérification est obligatoire, leur violation impacte le fonctionnement du système. Les contraintes molles sont plutôt des préférences, elles dépendent des exigences des utilisateurs.

Ces caractéristiques appellent à penser une vérification de configurations capable d'intégrer l'hétérogénéité de l'existant.

II.2.3. **Propriétés d'une vérification opérationnelle de configurations**

A la lumière de ces nouvelles exigences, nous avons identifié les propriétés d'une solution de vérification en ligne de configurations adaptée au contexte de reconfiguration dynamique. Elles sont présentées dans le Tableau 1.

Cycle de vie	Vérification en ligne	La solution doit pouvoir être utilisée pour une vérification de configurations *at runtime*.
Type	Vérification de la conformité structurelle	La solution doit pouvoir supporter une vérification de la conformité structurelle, i.e. vérifier la structure et la composition des éléments de configurations, le respect des dépendances.
	Vérification de l'applicabilité opérationnelle	La solution doit pouvoir supporter une vérification par rapport à l'état opérationnel du système, par exemple ne pas référencer un serveur non-opérationnel.
Expressivité	Vérification expressive	La solution doit pouvoir permettre une vérification de propriétés fonctionnelles et non-fonctionnelles standards, ainsi que de propriétés *user-defined*.
Pilotage	Vérification flexible	La solution doit pouvoir permettre une vérification modulable, par exemple effectuer une vérification partielle, renforcer ou relâcher des contraintes.
	Vérification évolutive	La solution doit pouvoir permettre une modification des informations de vérification tout au long du cycle de vie du système.
Ouverture (GRS)	Vérification générique	La solution ne doit pas être spécifique à un domaine applicatif, un protocole ou un système de gestion particulier.
	Vérification intégrée	La solution doit prendre en compte l'existant marqué par des systèmes de gestions hétérogènes.

Tableau 1 - Propriétés attendues pour une vérification opérationnelle de configurations

Ces diverses propriétés caractérisent une solution de vérification automatique permettant d'assurer en phase d'exploitation, une vérification opérationnelle flexible et évolutive de configurations candidates au sein d'un système de gestion autonome. Cette solution permettrait d'enrichir un système de gestion autonome de capacité d'auto-vérification de ses reconfigurations dynamiques.

A la lumière de ces nouvelles exigences, nous procédons, dans la section suivante, à l'étude de l'état de l'art de la vérification de configurations dans les réseaux et systèmes complexes.

II.3. **Etat de l'art**

Cette section porte sur l'étude de l'état l'art de la vérification de configurations dans la gestion de réseaux et de systèmes complexes. Nous étudions dans un premier temps les approches de vérification de configurations dans les standards de gestion. Dans un deuxième temps, nous nous focalisons sur les travaux de la littérature qui se sont penchés sur la vérification de configurations dans un contexte de reconfiguration dynamique ou dans un contexte proche d'automatisation de la reconfiguration. Ces travaux sont ensuite critiqués par rapport aux propriétés présentées dans le Tableau 1.

II.3.1. **Approche de vérification de configurations**

L'étude de l'état de l'art de la vérification de configurations de réseaux et systèmes complexes nous a permis d'identifier deux grandes catégories d'approches : les approches de vérification basée sur une spécification préalable de contraintes et les approches de vérification par apprentissage et reconnaissance de motifs.

II.3.1.1. **Approche par spécification de contraintes**

L'approche par spécification préalable de contraintes est l'approche classique de vérification de configurations. Dans cette approche, la vérification de configurations s'appuie sur une phase d'expression de contraintes sur des modèles de configuration plus ou moins évolués. Ces contraintes sont ensuite évaluées sur les instances de configuration afin de s'assurer de leur validité, par un système de vérification.

– Expression de contraintes

L'expression des contraintes se fait en phase de conception des modèles de configuration. Les administrateurs modélisent la configuration du système et expriment des contraintes que les instances de configuration devront respecter. Les contraintes exprimées portent sur la structure et la composition des paramètres de configuration et sur la restriction de leurs valeurs possibles par rapport à des exigences diverses comme les exigences techniques, les exigences des opérateurs et les exigences de service [Wuttke, 2010].

Ces contraintes consistent en des invariants que toutes les instances de configuration devront vérifier. En adaptant la classification dans [Agrawal et al., 2004], les contraintes peuvent être regroupées en trois catégories :

- les contraintes intra-élément qui permettent une vérification des paramètres de configuration intrinsèques d'un élément : la vérification de l'existence ou de l'unicité d'une valeur ou la vérification qu'une valeur respecte un intervalle de valeurs donné etc., par exemple «le numéro de port d'un serveur web est égal à 80» [Delaet & Joosen, 2007].
- les contraintes inter-élément pour la vérification de paramètres de configuration intrinsèques d'un élément par rapport aux paramètres de configuration d'un autre élément : la vérification de restrictions sur les connexions et les dépendances entre éléments de configuration, par exemple, «l'adresse et le port de chaque hôte virtuel doit correspondre à l'adresse et au port que le serveur Web écoute» [Sinz et al., 2003].
- les contraintes de groupe pour la vérification des paramètres de configurations sur un ensemble d'éléments : la vérification de propriétés communes à un ensemble d'éléments de configuration, par exemple «il n'existe pas plusieurs services écoutant sur le même numéro de port, sur la même machine » [Goldsack et al., 2009].

Un langage de spécification de contraintes permet leur expression. Le langage de spécification de contraintes peut être une partie du langage de modélisation de la configuration ou être séparée. L'expressivité de ce langage, c'est-à-dire sa capacité à supporter l'expression de contraintes de différentes natures, telles que dures, molles, sur la structure du système ou les exigences de service, conditionne l'expressivité et le type de vérification supporté.

– Modalités de mise en œuvre de la vérification

La vérification peut être mise en oeuvre de deux manières distinctes :

- la première, un système de vérification évalue les contraintes sur les instances de configurations et renvoie les résultats de l'évaluation. Leur évaluation renvoie vrai ou faux, éventuellement avec la liste des assertions non respectées [Yin et al., 2011].
- la seconde, un solveur de contraintes utilisent ces contraintes pour générer automatiquement les configurations valides ou à défaut prouver qu'aucune solution n'existe. La vérification de contraintes sur les configurations devient un problème de satisfaction de contraintes. Ces vérifications sont faites hors ligne pour la décision de configurations initiales [Stumptner et al., 1998].

Vérification par évaluation de contraintes

Dans cette approche, les contraintes sont transformées en assertions exécutables qu'un moteur d'exécution évalue sur des instances de configuration. L'évaluation retourne un résultat de vérification positif ou négatif avec si possible les raisons de l'échec. Cette vérification de contraintes est surtout effectuée hors ligne.

Problème de satisfaction de contraintes

Cette approche transforme la vérification de la configuration en un «problème de satisfaction de contraintes». Les contraintes sont analysées par un solveur soit pour déterminer dans un jeu de configurations donné, celles qui répondent aux contraintes, soit pour générer automatiquement la configuration solution respectant les contraintes exprimées. L'objectif n'est pas d'évaluer la correction d'une configuration donnée, mais de générer, dans un univers de configurations possibles, des configurations candidates adéquates si elles existent.

La notion de «Problème de Satisfaction de Contraintes» ou CSP (Constraint Satisfaction Problem) désigne l'ensemble des problèmes définis sous la forme d'un ensemble de contraintes posées sur des variables ou inconnues, et consistant à chercher une solution les respectant. Formellement, un CSP est un triplet (X, D, C) tel que :

- $X = \{X1, X2, ..., Xn\}$ est l'ensemble des variables ou des inconnues du problème,
- D est la fonction qui associe à chaque variable X_i, son domaine $D(X_i)$ c'est-à-dire l'ensemble des valeurs qu'elle peut prendre.
- $C = \{C1, C2, ..., Cm\}$ est l'ensemble des contraintes restreignant les valeurs que peuvent prendre simultanément les éléments de X.

La résolution d'un CSP se fait via un solveur CSP, comme les solveurs SAT (SATisfiability) ou BDD (Binary Decision Diagram), qui procède par affectation de valeurs aux variables de manière à satisfaire toutes les contraintes. Un solveur de CSP permet de trouver une solution au problème CSP ou de prouver qu'aucune solution n'existe.

II.3.1.2. Approches par apprentissage et reconnaissance de motifs

Ces approches veulent remplacer celles par spécification préalable de contraintes jugées sujettes à erreur. Les approches par spécification de contraintes par les administrateurs peuvent être sujettes à des erreurs de formulation et d'interprétation pouvant fausser la consistance des évaluations. De plus, elles supposent une connaissance étendue des propriétés devant être évaluées. Telles sont les critiques formulées par les travaux de vérification par apprentissage et reconnaissance de motifs [Kiciman & Wang, 2004].

Pour répondre à ces limites, ces approches s'appuient sur les méthodes de l'Intelligence Artificielle, telles que l'apprentissage automatique, l'analyse statistique et la reconnaissance de

motifs [Ramachandran et al., 2009 ; Uchiumi et al., 2012] pour proposer des techniques de vérification automatique sans spécification de contraintes par les administrateurs. Ces approches sont spécialisées dans les erreurs syntaxiques et les erreurs de valeurs intra et inter-composants.

– Modalités de mise en oeuvre de la vérification

La vérification de configurations se fait en ligne, pendant l'exécution du système. Dans ces approches, un moteur de vérification passe par une phase d'apprentissage de «bonnes» configurations en phase de conception. Une fois en phase d'exécution, il peut détecter toute configuration qui dévie de la forme apprise.

Dans la suite du chapitre, notre étude de l'état de l'art se concentre sur les approches par spécification de contraintes. Nous écartons les approches par apprentissage et reconnaissance de motifs. Bien que ces approches permettent une vérification en ligne, elles demeurent très limitées dans le type d'erreurs détectables avec des problèmes de précision et de détection de faux positifs. De plus, elles ne peuvent s'adapter à l'évolutivité du système, elles ne détectent pas les changements structurels et dépendent des motifs appris en phase d'apprentissage [Kiciman & Wang, 2004 ; Ramachandran et al., 2009].

II.3.2. Réponses des standards de gestion

Les standards peuvent être classés en deux grands groupes : ceux qui ne fournissent pas de vérification de configuration native et ceux qui fournissent des briques plus ou moins évoluées pour la vérification de configurations, c'est le cas des standards WBEM/CIM et NETCONF/YANG. En effet, en dehors de ces deux standards, les mécanismes de vérification proposés sont des mécanismes de bas niveau qui consistent en des retours de code d'erreurs ou d'exceptions suite à des erreurs syntaxiques ou de type.

II.3.2.1. Vérification non supportée nativement

– SNMP

SNMP ne permet pas de tester une configuration avant son déploiement. La version 2 de SNMP consolide et étend des mécanismes de gestion d'erreurs par les notions de codes d'erreur et d'exception [RFC3410 ; RFC3416]. Par exemple, à la réception d'une requête «SetRequest», l'agent analyse et vérifie les informations qu'elle contient pour s'assurer de leur consistance. Cependant, c'est à l'application de gestion (manager) recevant les erreurs ou les exceptions qu'il revient de les traiter, en analysant le code d'erreur retourné ou en détectant l'exception renvoyée. Par exemple, l'agent peut vérifier notamment le nom des instances à modifier (code erreur `inconsistentName`), contrôler si la valeur de l'instance est modifiable (code erreur `notWritable`) ou encore vérifier si la valeur spécifiée respecte le type et la taille recommandés (codes erreur `badValue`, `wrongType` et `wrongLength`).

– JMX

La spécification JMX propose un certain nombre d'exceptions qui encadrent la manipulation d'objets `MBeans` (les exceptions sont un mécanisme de gestion d'erreurs dans le langage Java). Ces exceptions peuvent être utilisées pour encadrer une vérification basique de configurations dans JMX. On retrouve par exemple les exceptions qui encadrent la non-existence d'un attribut (`AttributeNotFoundException`), la non-validité de sa valeur (`InvalidAttri-`

buteValueException). Ces exceptions doivent être généralement prévues à l'avance et codées en dur dans l'application de gestion. Difficiles à évoluer et peu flexibles, ces mécanismes d'exception sont bas-niveau pour être exploités dans la vérification opérationnelle de configurations que nous visons.

II.3.2.2. Vérification de configurations dans WBEM/CIM et NETCONF/YANG

– CIM/WBEM

Le standard CIM fixe un ensemble de *qualifiers* pour l'expression de contraintes à vérifier sur des éléments de modèles CIM. Un *qualifier* est un concept de CIM utilisé pour fournir des informations supplémentaires sur des éléments de modèle. C'est une métadonnée permettant de rajouter de la sémantique à un élément de modèle CIM.

CIM propose trois types de *qualifiers* pour la spécification de contraintes : le *qualifier* `ClassConstraint` pour la définition de contraintes sur les classes, les associations et les indications, le *qualifier* `PropertyConstraint` pour la définition de contraintes sur les attributs des classes de modèles, le qualifier `MethodConstraint` pour la définition de contraintes sur les méthodes des classes de modèles.

Pour la définition de ces contraintes, CIM propose l'utilisation du langage OCL (Object Constraint Language). OCL est un langage formel de description de contraintes sur des modèles, standardisé par l'OMG (Open Management Group) [OCL, 2006]. Un *qualifier* de contrainte comprend une description textuelle de la contrainte, exprimée en langage OCL.

La définition récente de la version 3.0 de CIM [CIM-V3, 2013], qui effectue par ailleurs une refonte globale du langage, supprime ces différents types de *qualifiers* et simplifie la spécification de contraintes OCL via l'introduction d'un qualifier unique, l'`ocl qualifier`.

La Figure 10 montre un exemple de définition de contraintes de type `ClassConstraint` : pour vérifier que les propriétés x et y ne sont pas nuls dans toutes les instances d'une classe donnée, il faut définir le *qualifier* suivant dans cette classe.

```
ClassConstraint {
   "inv: not (self.x.oclIsUndefined() and self.y.oclIsUndefined())"
}
```

Figure 10 - Exemple d'expression de contrainte avec CIM 2.3

CIM fournit des constructeurs pour exprimer des informations de configuration, d'état opérationnel et de contraintes. Il peut potentiellement supporter une description de haut niveau de configurations et de contraintes pouvant prendre en compte la conformité structurelle et l'applicabilité opérationnelle des configurations. CIM fournit donc les bases pour une vérification expressive, générique et intégrée.

En ce qui concerne l'exécution et le pilotage de la vérification, les standards CIM/WBEM ne fournissent aucune recommandation concernant l'exécution et le pilotage de la vérification de ces contraintes sur les éléments de configuration. Les choix d'implémentation de cette vérification sont laissés aux utilisateurs et aux outilleurs.

– YANG/NETCONF

YANG fournit des constructeurs pour l'expression de contraintes que les données de configuration doivent vérifier. Les contraintes dans YANG sont des contraintes de conformité structurelle. Elles sont limitées à des restrictions de types et à la vérification d'expressions XPath

(XML Path Language). La définition de contraintes sur les données configuration vis-à-vis des données d'état opérationnel n'est pas supportée [Lindblad, 2013].

YANG fournit deux mécanismes pour la définition de contraintes.

- Un mécanisme implicite de bas-niveau de restriction de types des éléments de configuration. Par exemple la précision de propriétés telles que `range`, `length` ou encore `pattern` lors de la définition du type d'un attribut pour restreindre l'intervalle de valeurs possibles, la longueur et l'expression régulière à respecter pour un type chaîne de caractères.

- Un mécanisme explicite de haut niveau d'ajout d'invariants à la définition des données de configuration via l'utilisation de contraintes prédéfinies et d'expressions XPATH 1.0. Des exemples de contraintes prédéfinis sont l'ajout de contraintes `mandatory` à la définition d'un nœud pour rendre sa présence obligatoire, `min-elements`/`max-elements` et `unique` dans une liste de nœuds, pour préciser la taille de la liste et l'unicité d'une valeur. Des exemples de contraintes XPATH sont la définition de contraintes `must` ou `path` qui contiennent des expressions XPATH dont l'évaluation permettra de vérifier la validité d'un nœud donné.

```
leaf interface-group-name {
  type string {
  length "1..31";
  pattern "[a-zA-Z][a-zA-Z0-9]*";
  }
  must "not(/sys:sys/interface[name = current()])" {
    error-message ″Must be different from all interface
names";
  }
}
```

Figure 11 - Définition de contraintes avec YANG

La Figure 11 montre un exemple de définition de contraintes avec YANG. D'une part, les contraintes implicites `length`, `pattern` sur le type permettent de préciser le type du paramètre de configuration à savoir une chaîne de caractères alphanumériques d'une longueur entre 1 et 31 et commençant par une lettre alphabétique. D'autre part, une contrainte XPATH `must` permet de vérifier que la valeur `interface-groupe-name` est unique.

YANG est générique dans la mesure où ses constructeurs ne sont pas spécifiques à un domaine d'application. Cependant, il ne permet pas de supporter la vérification de l'applicabilité opérationnelle des configurations. De plus, les types de contraintes exprimables sont fixés à l'avance, l'expressivité d'une vérification basée sur des contraintes exprimées en YANG est donc limitée.

Le protocole NETCONF émet des recommandations en ce qui concerne la vérification de contraintes :

- les contraintes implicites sont interprétées automatiquement lors des communications RPC entre clients et serveurs via le *parsing* des spécifications YANG.

- quant aux contraintes explicites, le protocole NETCONF définit une capacité de vérification que les outils compatibles pourront implémenter. Cette capacité repose sur l'opération `validate(source)` qui permet de vérifier des contraintes sur une configuration `source`. Cependant, le support de cette capacité est optionnel et laissé à l'appréciation des utilisateurs et outilleurs.

Bien que le langage YANG soit générique (vis-à-vis d'un domaine applicatif donné), une vérification de contraintes explicites est liée à l'opération «`validate`» du protocole NETCONF, cela implique donc une dépendance à un protocole de configuration spécifique.

De plus, NETCONF préconise une vérification de configurations au niveau de l'équipement, c'est-à-dire que c'est au serveur installé sur l'équipement de vérifier localement sa configuration candidate avant son application. Ce choix rend difficile la vérification de contraintes inter-composants et de contraintes de groupes sur le système.

Enfin, les standards NETCONF/YANG ne proposent pas de recommandations en matière de pilotage de la vérification.

II.3.2.3. **Discussions**

Peu de standards existants fournissent des spécifications exploitables pour la vérification de haut niveau que nécessite la prise en compte de la reconfiguration dynamique. Les mécanismes proposés, code erreurs et exceptions des standards SNMP et JMX sont bas-niveau. Quant aux standards CIM/WBEM et YANG/NETCONF, bien qu'ils proposent des constructions de base pour la spécification des contraintes, ces constructions restent insuffisantes face aux besoins actuels de vérification telles que la prise en compte de l'état opérationnel du système géré et la gestion du cycle de vie de la vérification. De plus, l'implémentation des fonctionnalités de vérification est laissée à l'outilleur ou l'utilisateur.

Il existe, en dehors du paysage des standards, des travaux académiques qui se sont intéressés à la vérification de configurations de réseaux et systèmes. Leur présentation fait l'objet de la section suivante.

II.3.3. **Présentation des principaux travaux**

Quelques travaux de vérification de configurations de réseaux et systèmes complexes peuvent être trouvés dans la littérature. Nous ne détaillons ici que des travaux qui se sont penchés sur une vérification de configurations soit dans le contexte de la reconfiguration dynamique, soit dans un contexte proche d'automatisation de la reconfiguration. Les travaux retenus se divisent en deux groupes, ceux qui procèdent par vérification de contraintes et ceux qui procèdent par génération de configurations valides.

II.3.3.1. **Vérification par évaluation de contraintes**

– Les travaux de Konstantinou et al.

Les travaux de [Konstantinou et al., 2002] ont pour objectif la conception d'une architecture pour l'auto-configuration de réseaux appelée `NESTOR`. Bien que la vérification de contraintes ne soit pas l'objectif final de `NESTOR`, il inclut une capacité de vérification de contraintes dans l'architecture afin de détecter les erreurs de configuration et prévenir l'application de configurations invalides.

`NESTOR` fournit à cet effet RDL (Resource Definition Language) un langage orienté-objet similaire au langage IDL (Interface Definition Language) de l'OMG [IDL, 2012] pour la définition de modèles de configuration et CDL (Constraint Definition Language), un langage de définition de contraintes basé sur OCL. Les déclarations de la structure de données et des contraintes sont séparées pour permettre l'évolution aisée de l'une et de l'autre.

Un exemple de définition de contraintes CDL est illustré dans la Figure 12 : cette contrainte vérifie que la valeur de paramètre `hostname` de tout composant nommé `NetworkedSystem` est non nulle et unique.

```
nestor::system::NetworkedSystem::->allInstances
  ->select(h | h.hostname <> null)
  ->forAll(h1, h2 | h1 <> h2 implies
           h1.hostname <> h2.hostname);
```

Figure 12 - Définition de contraintes dans NESTOR

Bien que NESTOR supporte une reconfiguration dynamique des équipements réseaux, l'expression de contraintes dans NESTOR est limitée aux données de configuration seules. Ces contraintes ne supportent que la vérification de la conformité structurelle.

Les langages de spécification de NESTOR possèdent un haut niveau d'abstraction qui permet une expression générique de contraintes. Cependant, la vérification de l'applicabilité opérationnelle n'est pas prise en compte et l'architecture support est dédiée spécifiquement aux réseaux.

– **Les travaux de Goldsack et al.**

Les travaux de [Goldsack et al., 2003] visent à concevoir un cadriciel appelé `SmartFrog` pour la gestion automatique de la configuration d'applications distribuées développées en Java. Une évolution de ces travaux en 2009 permet de supporter une reconfiguration dynamique d'applications supportant cette capacité [Goldsack et al., 2009].

`SmartFrog` propose un langage de configuration qui inclut un mécanisme de définition de contraintes sous forme de prédicats logiques. Le langage permet de décrire des gabarits de configurations de haut niveau qui peuvent être spécialisés si besoin pour correspondre aux formats attendus par les entités gérées.

Des exemples de contraintes sont présentées dans la Figure 13 : la contrainte `portOk` est définie sur un modèle de configuration de serveur web (`webserver`). Elle garantit que le numéro de port (`port`) est inférieur ou égal à `0xFFFF` (65535 en hexadécimal).

Les contraintes sont des invariants de conformité structurelle. Elles sont incluses au sein des descriptions de configuration, toutes les instances de configuration doivent les vérifier.

`SmartFrog` inclut également un gestionnaire (*lifecycle manager*) qui procède à l'évaluation des contraintes sur les configurations candidates avant leur application sur l'élément géré.

```
webServer extends {
    port TBD;
    logLevel "WARN";
    valid extends Assertions, {
        portOK (port <= 0xFFFF);
        logOK (logLevel == "WARN" OR
               logLevel == "ERROR");
    }
}
system extends {
   server1 extends webServer, {
       port 80;
   }
   server2 extends webServer, {
       port (server1:port + 1);
   }
   valid extends Assertions, {
       portsDiff (server1:port != server2:port);
   }
}
```

Figure 13 - Définition de contraintes dans SmartFrog

Bien que supportant la reconfiguration dynamique, `SmartFrog` ne prend pas en compte la vérification de l'applicabilité opérationnelle. De plus, la vérification proposée n'est pas modulable et le cadriciel ne fournit pas explicitement de mécanismes pour la gestion de son évolution.

– Les travaux de Agrawal et al. et de Gençay et al.

Les travaux de [Agrawal et al., 2004] et de [Gençay et al., 2008] se sont intéressés à la vérification automatique de configurations de réseaux de stockage ou SANs (Storage Area Networks).

Agrawal et al., propose un cadre de vérification de configuration de réseaux de stockage basé sur l'utilisation de politiques `Condition-Action-priorité` dans un contexte général où la spécification procédurale (*hard-coding*) de contraintes au sein des logiciels de gestion devenait impossible face à la complexité croissante de ces réseaux. La condition de la politique est la contrainte à évaluer, l'action spécifie ce qu'il faut faire des résultats des évaluations et la priorité permet d'éviter les conflits entre des politiques concurrentes.

Ces travaux déterminent également une série de politiques de conformité structurelle intra-composant, inter-composant et de groupe qui doivent être vérifiées par toute configuration SAN.

En se basant sur ce cadre, Gençay a implémenté `SANChk`, une solution de vérification de configurations de SANs qui définit un schéma de base de données pour le stockage des informations de configuration et utilise le standard SQL (Standard Query Language) pour l'expression des contraintes. Une contrainte devient une requête SQL de sélection, de projection, d'agrégation et de jointure sur des tables contenant des valeurs de configuration de SAN.

Ces travaux étendent la proposition d'Agrawal et al. en y ajoutant des attributs supplémentaires comme le statut d'une contrainte (activé ou désactivé) et le niveau de sévérité rendant plus souples les vérifications de contraintes et posant les bases d'une gestion de leur cycle de vie :

- leur périmètre : c'est-à-dire l'élément de configuration qu'elles vérifient : cette information permet de savoir sur quelle table de la base de données effectuer la requête,
- leur niveau de priorité : le niveau de priorité permet de prévenir les conflits de politiques,
- leur niveau de sévérité : par exemple la violation de la contrainte peut être bénigne (Warning) ou critique (Error). Cette information a été introduite pour permettre une vérification flexible, par exemple n'évaluer que des requêtes d'un certain niveau de sévérité,
- et leur statut d'activité : le statut d'activité permet de gérer le cycle de vie d'une contrainte, elles peuvent être activées ou désactivées suivant le cas de test envisagé.

Les requêtes SQL sont encapsulées dans un modèle XML comme illustré dans la Figure 14 : cette contrainte vérifie que la zone contient des systèmes de sauvegardes de type disque ou bandes magnétiques. Cette contrainte a une sévérité de niveau erreur et une priorité de 60. Elle est active, donc elle sera considérée dans la vérification d'une configuration potentielle.

`SanChk` évalue les formats XML et génèrent des rapports de vérification au format HTML que les administrateurs peuvent consulter afin de modifier les configurations en faute. Il s'inscrit dans une démarche de vérification de configuration basée sur les politiques, le résultat d'exécution de requêtes est utilisé comme condition de déclenchement d'actions telles que «envoyer un email à un administrateur».

```
<Test
  Name="zonesWithTapeLibrariesAndDiskSubsystems"
  Description="Every zone should have either
               disk subsystems or tape libraries."
  Severity="Error"
  Priority="60"
  IsActive="1">
  <TestQuery QueryType="StandardSQL">
    0 = (SELECT COUNT(*) FROM
          (SELECT DISTINCT prefix_id, zone_id
           FROM tpc.t_view_zone2member_ent_con_to_switch)
           AS res1,
          (SELECT DISTINCT prefix_id, zone_id
           FROM tpc.t_view_zone2member_ent_con_to_switch)
           AS res2
         WHERE res1.zone_id = res2.zone_id
         AND res1.prefix_id = 'tapelibrary:'
         AND res2.prefix_id = 'subsystem:')
  </TestQuery>
</Test>
```

Figure 14 - Définition de contraintes dans SanChK

Un prototype de `SanChk` a été développé sur l'architecture WBEM afin de permettre une vérification intégrée de réseaux de stockage. Dans ce prototype, SanChk remplit sa base de données de configuration à partir des instances de configuration CIM récupérées auprès d'un serveur WBEM.

La solution de vérification `SanChk` répond à l'essentiel des propriétés attendues. Elle permet une vérification flexible et évolutive des configurations via les attributs additionnels de sévérité et d'activité sur les contraintes. Cependant elle ne prend pas en compte le contexte d'exécution des systèmes gérés. De plus, la vérification proposée n'est pas complètement générique puisqu'elle repose sur un schéma relationnel spécifique à la configuration des SANs.

II.3.3.2. Vérification par génération de configurations valides

– Les travaux de Ramshaw et al.

Le contexte général de ces travaux est l'automatisation de la gestion de configuration face à la complexité des systèmes actuels.

[Ramshaw et al., 2006] définit `Cauldron` une solution de génération de configurations par satisfaction de contraintes basée sur les politiques ECA (Event-Condition-Action). Cette architecture inclut une version simplifiée du format `MOF` du DMTF et un ensemble de solveurs et de systèmes experts.

`Cauldron` étend la classe `CIM_Policy` avec des attributs supplémentaires qui permettent de spécifier des assertions en logique de premier ordre. Un modèle de configuration est organisé en classes et associations. Les contraintes sont des assertions incluses dans chaque classe via le mot-clé `satisfy`. La génération d'instances de configurations se fait via un solveur SAT et des systèmes experts qui soit retournent une solution potentielle, soit prouvent l'insatisfiabilité.

La Figure 15 montre un exemple de définition d'un problème de satisfaction de contraintes dans `Cauldron`. Elle présente un modèle de configuration constitué de deux classes : `Server` et `Job`. Les classes `Server` et `Job` sont liées par une association un-à-plusieurs (mot-clé `ref`). Les contraintes `satisfy` de cet exemple visent à assurer qu'un serveur peut exécuter plusieurs tâches mais qu'une tâche n'est assignée qu'à un et un seul serveur.

`Cauldron` repose sur la définition de politiques structurelles et ne prend pas en compte l'état opérationnel.

```
Job {
    myServer: ref Server;
    satisfy gor(i, myServer.myJobs,
        myServer.myJobs[i] == this);
}
Server {
    myJobs: (ref Job)[..50];
    satisfy gand(i, myJobs,gand(j, myJobs,
        myJobs[i] == myJobs[j] ==> i ==
        j));
    satisfy gand(i, myJobs,
        myJobs[i].server == this);
}
```

Figure 15 - Définition de contraintes avec CAULDRON

– Les travaux de Delaet et Josen

Les travaux de [Delaet & Joosen, 2007] visent la conception de PODIM, un langage de haut niveau pour la génération automatique d'instance de configuration valides dans le cadre de l'administration de systèmes. Ces travaux ont pour objectif la génération d'instances de configuration à partir d'une description de modèles de configuration et de règles. Ces travaux interviennent dans le contexte général de l'automatisation de la gestion de la configuration.

PODIM est constitué d'un langage de modélisation d'informations de configurations et d'un langage de définition de politiques «Condition-Action» spécifiques consistant en des règles de création et de modification de configurations et des règles de définition de droits d'accès. Il permet une modélisation indépendante des plateformes d'informations de configuration. Il est basé sur le langage de programmation EIFFEL [Meyer, 1992].

L'utilisation du langage consiste dans un premier temps, à définir dans un format textuel, le modèle de configuration du système géré (ensemble de classes et associations). On peut spécifier des invariants structurels au sein des classes pour encadrer l'intervalle des valeurs de certains attributs.
Dans un deuxième temps, on définit des règles qui peuvent être de création d'objet (`creation constraint`), par exemple «le système possède deux serveurs web», de modification de configuration existante (`assignment constraint`), «le numéro de port d'un serveur doit être compris entre 80 et 1024 » ou de limitation de droits (`filter`), «la modification de l'attribut `php_enable` n'est pas autorisée». La définition des règles s'appuie sur un formalisme similaire au langage SQL.

Un exemple de ces règles est montré sur la Figure 16 : cette règle permet de configurer des interfaces réseaux de telle sorte qu'elles aient toutes une adresse IPv4 statique comprise entre 134.58.39.0 et 134.58.47.255 sauf l'interface de l'équipement «jasje».

```
creation
-- Each interface has 1 IPv4 address ,
-- except "jasje"
[1-1] NETWORK_IPV4_STATIC_ADDRESS
select NETWORK_INTERFACE
where NETWORK_INTERFACE.device.name
/= "jasje" and
NETWORK_INTERFACE.labels.has("KULEUVENNET")

assignment constraint
-- Public address space
address [!!IPV4_ADDRESS.make("134.58.39.0"):
 !!IPV4_ADDRESS.make("134.58.47.255")]
select NETWORK_IPV4_STATIC_ADDRESS
where NETWORK_IPV4_STATIC_ADDRESS.interface.
 labels.has("PUBLIC_SUBNET")
```

Figure 16 - Définition de contraintes dans PODIM

PoDIM fournit un compilateur qui peut être intégré à un outil de gestion de configuration existant [Delaet et al., 2008]. A partir du modèle de configuration et des divers politiques, il génère des fichiers de configurations prêts à être déployés. La configuration du système est complètement regénérée à chaque fois, sans prise en compte des anciennes configurations.

Bien que PODIM soit générique dans ces constructions, les politiques PODIM sont des politiques structurelles qui ne prennent pas en compte l'état opérationnel.

– **Les travaux de Narain et al.**

Les travaux de Narain et al. [2008; 2010] se situent dans le contexte du développement de IPAssure, une solution (commercialisée par la société Telcordia) de vérification de configurations et de génération automatique d'instances de configuration valides d'équipements réseaux. Ces travaux s'appuient sur les méthodes formelles pour définir des contraintes logiques sur un modèle de configuration. Ils exploitent des langages de programmation logiques tels que Prolog et Datalog et des solveurs SAT.

Le modèle de configuration est déduit de l'analyse des fichiers de configuration du réseau ou de commandes CLI et consiste en une base de données constituée de tuples Prolog. Des contraintes logiques de conformité structurelle, couvrant des exigences d'intégrité, de connectivité, de sécurité et de performance des équipements réseaux (routeurs) sont exprimées sous forme de clauses Prolog et évaluées sur ces tuples par un solveur SAT.

Les auteurs dans [Narain et al., 2008] mettent en particulier l'accent sur la résolution de l'insatisfiabilité. Les contraintes Prolog sont d'abord simplifiées par un moteur d'élimination de quantificateurs. Ces contraintes simplifiées sont ensuite évaluées sur les tuples de configuration Prolog par le solveur dédié. Lorsque le solveur ne trouve pas de solutions, il retourne le jeu de contraintes non solvables. Ce jeu fait l'objet d'un algorithme de suppressions successives qui se termine dès qu'un sous-ensemble solvable est trouvé.

La Figure 17 illustre un exemple de définition de problèmes de satisfaction de contraintes sur une base de configurations. La base de configuration (Figure 17 - ①) est constituée de quatre tuples, les trois premiers décrivent la configuration du routeur `ra` : elle comprend une route statique, une interface `tunnel_0` qui fonctionnera suivant le protocole de mise en tunnel GRE (Generic Routing Encapsulation), et une interface physique `eth_0`. Le dernier tuple configure l'interface `eth_0` sur le routeur `rb`. Une instance de configuration valide (`good`) doit garantir une connectivité GRE entre les routeurs `ra` et `rb`, c'est-à-dire que `ra` et `rb` doivent être reliés par un tunnel GRE et il doit exister une route entre `ra` et `rb` (Figure 17 -②).

Ces travaux ne traite pas de l'applicabilité opérationnelle d'une configuration.

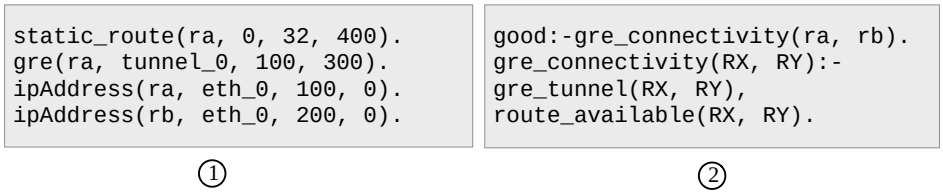

Figure 17 - Définition de contraintes dans Narain et al.

– **Les travaux de Hewson et al.**

Les travaux de [Hewson et al., 2011] visent la conception d'une solution de génération automatique de configurations initiales de machines virtuelles selon les exigences des opérateurs.

Ces travaux fournissent ConfSolve, un langage déclaratif orienté-objet de haut niveau qui permet de définir des contraintes sous forme de prédicats logiques sur des classes de configurations. Cette définition est utilisée par un solveur CSP appelé Gecode pour la génération de configurations valides.

Les contraintes ainsi définies sont des invariants structurels sur des données de configuration. La satisfaction de contraintes produit une instance de configuration solution appelée instance ConfSolve. L'instance ConfSolve peut être alors traduite dans le format attendu par le système géré. Les auteurs ne précisent pas la gestion des cas d'insatisfiabilité.

```
class Machine {
        var cpu as int
        var memory as int // MB
        var disk as int // GB
        var network as Network;

        where cpu == 16
        where memory == 16384
        where disk == 2048
        where network = Network.Public
    }

    class SmallRole {
        var host as ref Machine
        var disk as int
        var cpu as int
        var memory as int
        var network as Network;

        where cpu == 1
        where memory == 768
        where disk <= 20
    }
    var machines as Machine[2]

    var web_server as SmallRole
        where web_server.disk == 15
        where web_server.network == Network.Public
```

Figure 18 - Définition de contraintes dans ConfSolve

La Figure 18 montre un exemple de définition d'un problème de satisfaction de contraintes avec ConfSolve : cette spécification décrit le modèle de configuration d'une machine virtuelle (web_server) et de la machine hôte qui l'héberge. Les contraintes considérées sont des contraintes structurelles. Dans cet exemple, la génération de configurations valides supporte donc la vérification de la conformité structurelle. Elle répond également à l'optimisation de l'allocation de ressources à des machines virtuelles.

Des travaux récents et prometteurs sur l'évolution de ConfSolve pour supporter la reconfiguration dynamique font écho à nos travaux.

Evolution récente de ConfSolve pour la reconfiguration dynamique

Des travaux récents ont apporté des évolutions majeures à ConfSolve pour supporter la reconfiguration dynamique [Hewson et al., 2013].

Les auteurs ont introduit la possibilité d'exprimer des contraintes logiques par rapport au contexte d'exécution du système géré. Pour ce faire, deux constructeurs ont été ajoutés au langage : le constructeur «Parameter» pour la définition de paramètres d'état opérationnel et le constructeur «~» (tilde) pour référencer la configuration courante.

Les contraintes ne sont plus incluses dans la description des classes de configuration. Elles sont séparées en deux blocs : les contraintes init qui ne sont prises en compte que pour la génération de configurations initiales (elles n'incluent pas de paramètres opérationnels dans leur spécification) et les contraintes change qui permettent d'exprimer des contraintes par

rapport à l'état courant du système pour une génération de configurations candidates valides applicables opérationnellement.

Un exemple de cette évolution est illustrée dans la Figure 19 : une machine virtuelle possède un paramètre de configuration faisant référence à sa machine hôte (`host`). Une machine hôte possède un paramètre opérationnel `failed` qui indique si elle est en panne. Pour préserver la disponibilité, lorsqu'une machine-hôte est en état de panne, ses machines virtuelles sont migrées vers d'autres hôtes en état actif.

L'expression de la contrainte `change` permet de garantir que seules les machines virtuelles dont l'hôte est en panne seront migrées.

L'expérimentation de cette évolution sur un système réel n'a pas encore été présentée, mais les premières simulations offrent des résultats encourageants. Cette évolution rejoint les besoins que nous avons identifiés de vérification opérationnelle.

```
class Machine {
  param failed as bool;
}

abstract class VM {
  var host as ref Machine;
}

var rack1 as Machine[48]; // standard rack size
var vms as VM[192];       // 4*VM per Machine

change {
  forall vm in vms where !vm.host.failed {
    vm.host = ~vm.host;
  };
}
```

Figure 19 – Evolution de ConfSolve prenant en compte l'état courant du système.

II.3.4. Tableau récapitulatif

Le Tableau 2 suivant dresse un récapitulatif des travaux et des solutions de l'état de l'art étudiés par rapport aux propriétés attendues (Section II.2.3) :

- L'émoticône ☹ signifie soit que la solution ne répond pas à la propriété attendue soit que cette information est non disponible.
- L'émoticône 😐 signifie que la solution répond partiellement à la propriété, par exemple, la propriété est uniquement supportée au niveau du langage de spécification de contraintes.
- L'émoticône ☺ signifie que la solution répond à la propriété attendue.

	Cycle de vie	Types de vérification		Expressivité	Pilotage de la vérification		Ouverture (GRS)	
	En ligne	Conformité structurelle	Applicabilité Opérationnelle	Expressive	Flexible	Evolutive	Générique	Intégrée
CIM/WBEM	☹	☺	😐	☺	☹	☹	☺	☺
YANG/NetCONF	☹	☺	☹	😐	☹	☹	😐	☹
Konstantinou et al.	☺	☺	☹	☺	☹	☹	☹	☹
Gençay et al.	☹	☺	☹	☺	☺	😐	☹	😐
Goldsack et al.	☺	☺	☹	😐	☹	☹	😐	😐
Ramshaw et al.	☹	☺	☹	😐	☹	☹	😐	😐
Delaet et Joosen	☹	☺	☹	☺	☹	☹	😐	☹
Narain et al.	☹	☺	☹	☺	☹	☹	☹	☹
Hewson et al.	😐	☺	☺	☹	☹	☹	😐	☹

Tableau 2 – Comparaison des approches et solutions de vérification de configurations

Que ce soit au niveau des modalités de spécifications de contraintes ou d'exécution de la vérification, les propriétés identifiées pour une solution appropriée ne sont jamais complètement satisfaites.

- La majorité des travaux ne supporte pas la vérification en ligne et se limite à la vérification de la conformité structurelle sans prendre en compte le contexte d'exécution des éléments gérés.
- Aucun de ces travaux, à l'exception de Gençay et al., ne supportent une vérification flexible et évolutive (par exemple ne vérifier qu'un sous-ensemble de contraintes ou rajouter des contraintes tout au long du cycle d'exploitation du système).
- Quant à la prise en compte de l'existant, la plupart sont dédiés à des domaines spécifiques (réseaux, SAN, applications distribuées Java).

Deux solutions se rapprochent toutefois de notre solution idéale :

- la solution `SanChk` de Gençay et al. : `SanChk` répond à la problématique de la flexibilité de la vérification et jette les bases d'une vérification évolutive à travers l'ajout de métadonnées de sévérité et d'activité aux contraintes utilisateurs. Cependant cette solution repose sur un schéma relationnel spécifique aux réseaux de stockage de SAN et ne peut donc être utilisée en l'état pour une vérification générique.
- la solution `ConfSolve` de Hewson et al. : `ConfSolve` est une solution prometteuse au vu des récentes évolutions qui vont dans le sens de nos travaux. L'objectif de ces évolutions est de supporter une vérification par satisfaction de contraintes prenant en compte le contexte d'exécution des éléments gérés. Aucune information relative à des expérimentations de ces évolutions sur des systèmes réels n'est disponible à ce jour, pour en apprécier la viabilité. De plus, la solution ne supporte pas de vérification flexible et évolutive.

Nos travaux ont donc consisté à définir un cadriciel de vérification de configurations qui répond aux propriétés attendues :

- le cadriciel doit supporter une vérification en ligne de configurations : il doit permettre autant la vérification de la conformité structurelle des configurations que leur vérification par rapport aux conditions opérationnelles.
- le cadriciel doit répondre à la problématique de la dynamicité du système géré et des objectifs de gestion : il doit favoriser une vérification flexible qui peut s'adapter aux scé-

narii opérationnels et une vérification évolutive, les contraintes doivent pouvoir être modifiées en phase d'exécution et tout au long du cycle de vérification

- le cadriciel doit prendre en compte l'hétérogénéité de la GRS : il doit supporter une vérification générique et prend en compte l'existant.

II.4. Conclusion

L'objectif de ce chapitre était d'étudier la vérification de configurations comme une des réponses à la problématique d'assurance des comportements autonomes de reconfigurations. Cette vérification se fait traditionnellement hors ligne et se préoccupe de la correction et de la consistance structurelle des configurations. La vérification de configurations dans un contexte de reconfiguration dynamique doit se faire en ligne. Ce nouveau paramètre introduit une série d'exigences à couvrir afin de penser une solution de vérification adaptée.

Au-delà de la vérification classique de la conformité structurelle des configurations, nous avons identifié un deuxième niveau de vérification, la vérification de l'applicabilité opérationnelle c'est-à-dire la vérification des configurations vis-à-vis de conditions opérationnelles courantes. Cette nouvelle forme de vérification doit également prendre en compte la dynamicité et l'évolution du contexte d'exécution en favorisant une vérification flexible et évolutive. Elle doit également intégrer l'hétérogénéité de l'existant.

A la lumière de ces nouvelles exigences, nous avons examiné l'état de l'art des approches et solutions existantes de vérifications de configurations. La plupart des travaux existant se concentrent sur l'automatisation d'une vérification classique qui ne prend pas en compte ou ne le fait que partiellement, les exigences de vérification vis-à-vis de l'état courant, de flexibilité et d'évolutivité de la vérification.

Pour répondre à ces limites, nos travaux se sont concentrés sur la définition d'un cadriciel dédié à la vérification en ligne de configurations. Le cadriciel supporte une vérification opérationnelle générique, flexible et évolutive de configurations en situation de reconfiguration dynamique de réseaux et systèmes complexes. Ce cadriciel est présenté dans la deuxième partie de ce manuscrit.

2
Contributions

Chapitre 3

Définition d'un cadriciel de vérification de configurations

L'objectif de ce chapitre est de présenter le cadriciel de vérification de configurations que nous avons défini pour répondre aux exigences de vérification opérationnelle de configurations. Le cadriciel que nous proposons, fournit un métamodèle dédié à la spécification et la vérification de configurations. L'architecture d'un service de vérification opérationnelle s'appuie sur cette spécification pour évaluer automatiquement les configurations candidates. Un des défis majeurs est la prise en compte de l'existant et des standards actuels de gestion, le cadre inclut à cet effet une architecture de composants adaptateurs pour l'intégration de l'existant.

III.1. **Positionnement au sein de la boucle de gestion**

Nos travaux visent à créer un cadriciel supportant la vérification de configurations candidates lors d'une reconfiguration dynamique. D'un point de vue conceptuel, nous avons pensé une intégration d'une fonction de vérification au sein de la boucle de gestion MAPE. Cette fonction de vérification vient enrichir la décision en permettant de vérifier les configurations candidates avant leur application. Pour répondre au besoin de vérification des configurations en fonction du contexte d'exécution, la fonction de vérification s'appuie sur le bloc fonctionnel de la supervision. Ce dernier est sollicité pour la récupération de valeurs courantes de paramètres d'état nécessaires à la vérification.

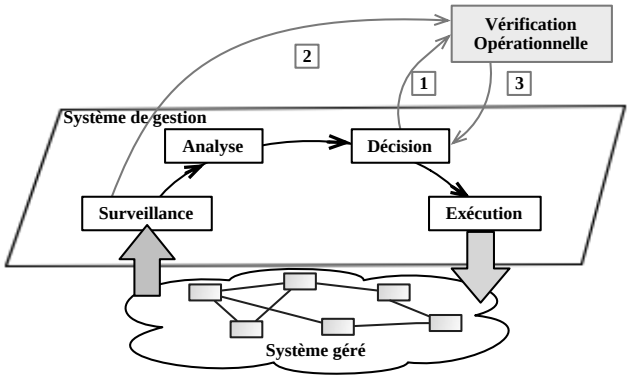

Figure 20 - Positionnement de la vérification au sein de la boucle de gestion

L'intégration de cette fonction de vérification dans la boucle de gestion repose sur trois étapes principales :

- Demande d'évaluation de configurations, (Figure 20 – 1) : un besoin de reconfiguration a été détecté et le module de *décision* a choisi une nouvelle configuration à appliquer, il la soumet au module de *vérification opérationnelle* pour évaluation.

- Collecte des données d'état courant, (Figure 20 – 2) : pour les besoins de la vérification, le module de vérification requiert des informations liées à l'état courant du système géré, le module de *supervision* les lui fournit en réponse.

- Exécution de la vérification, (Figure 20 – 3) : le module de vérification évalue la configuration candidate à la fois structurellement et en fonction des informations opérationnelles récupérées à l'étape 2. Il retourne ensuite les résultats de cette vérification au module décisionnel.

L'identification de ces deux niveaux d'intégration (avec le module de décision et avec le module de supervision) de la vérification au sein de la boucle de gestion permet d'assurer une vérification des configurations candidates (issues de la décision) qui tient compte des conditions opérationnelles courantes (remontées par la supervision).

Cette externalisation de la fonction de vérification présente plusieurs avantages qui permettent de supporter les nouveaux besoins de vérification de configurations :

- La vérification n'est pas liée à un système de décision ou de gestion en particulier. Elle pourrait donc être greffée à tout système de gestion afin de supporter la vérification opérationnelle de ses configurations.

- La vérification devient un mécanisme à part entière. Elle pourrait donc être modulée, adaptée et gérée en fonction des besoins d'utilisation.

Pour supporter pleinement ces avantages, et en tenant compte des nouvelles exigences identifiées (section II.2.3), le cadriciel de vérification doit permettre une spécification formelle de configurations indépendante des plates-formes et des protocoles de gestion et un support opérationnel de la vérification générique et flexible.

III.2. **Un cadriciel de vérification de configurations**

III.2.1. **Présentation générale du cadriciel**

Notre démarche pour construire ce cadriciel de vérification opérationnelle a consisté dans un premier temps à définir un langage de haut niveau, dédié à la spécification et la vérification de configurations. Nous avons architecturé dans un deuxième temps un service adaptable de vérification opérationnelle capable de manipuler les concepts définis dans le langage. Enfin, nous avons défini une interface facilitant l'intégration de l'existant. L'ensemble forme un cadriciel (Figure 21) qui supporte un processus de vérification opérationnelle de configurations qui commence par une phase de conception de modèles formels de configuration (Figure 21a) et se poursuit en phase d'exécution à travers une vérification basée sur ces modèles (Figure 21b).

- Phase de conception : un concepteur utilise notre langage pour spécifier formellement des modèles de configuration pour un système géré.
  Idéalement, tout système configurable est livré avec une spécification générale de sa configuration. Dans ce cas, le rôle du concepteur est tenu par un expert du côté du vendeur ou du constructeur du système. Cette spécification de base peut être étendue ou spécialisée si nécessaire, pour prendre en compte des besoins métier spécifiques côté utilisateur. Dans ce deuxième cas, le rôle du concepteur est joué par un administrateur ou tout autre expert qui a la maîtrise de la configuration du système.

- Phase d'exécution : le service de vérification s'appuie sur cette spécification de base pour vérifier les configurations qui lui sont soumises. Ce service supporte également une édition des spécifications en cours d'exécution. Cette édition opérationnelle est nécessaire afin de supporter l'évolution des besoins métier. Elle est donc effectuée du côté utilisateur, par exemple par un administrateur.

Les briques supportant ces deux phases sont détaillées ci-dessous.

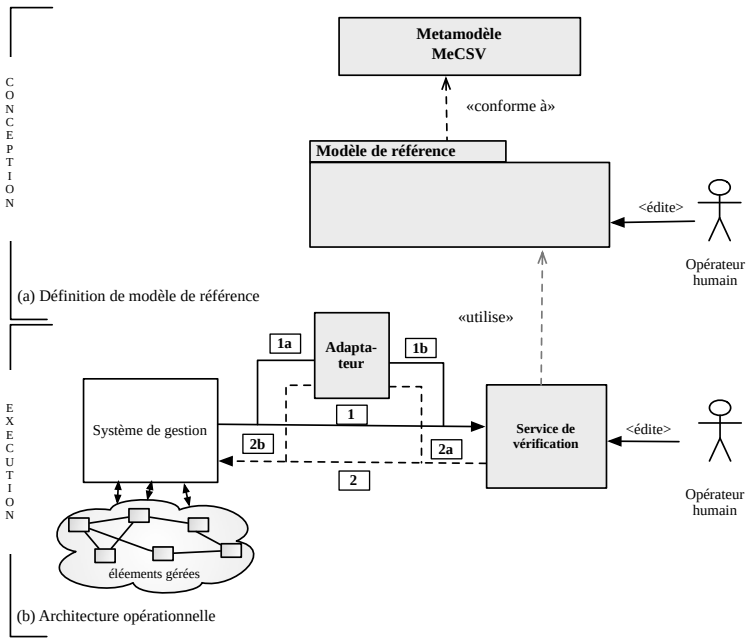

Figure 21 - Cadre de vérification de configuration

– Eléments supports de la phase de conception (Figure 21a)

Le cadriciel inclut MeCSV (Metamodel for Configuration Specification and Validation), un métamodèle que nous avons crée et dédié à la spécification et à la vérification de configurations. Ce métamodèle fournit des constructeurs pour la représentation d'informations de configuration, d'informations d'état opérationnel et de contraintes à vérifier. MeCSV peut ainsi être utilisé pour concevoir des modèles de référence propres à des environnements gérés.

Définition 17 ***Modèle de référence MeCSV*** *: L'utilisation du métamodèle MeCSV pour modéliser une spécification formelle de la configuration d'un système géré produit le modèle de référence pour un domaine applicatif donné. Le modèle de référence comprend le modèle de configuration du système géré, les contraintes qui garantissent la conformité structurelle et l'applicabilité opérationnelle des configurations vis-à-vis de paramètres d'état opérationnel.*

La description contenue dans un modèle de référence MeCSV permet une spécification des informations utiles à la vérification de configurations. Le cadriciel supporte ainsi une spécification rigoureuse de modèles de configuration en phase de conception.

– Eléments supports de la phase d'exécution (Figure 21b)

Le cadriciel comprend une architecture logicielle de vérification basée sur le métamodèle MeCSV, et donc sur les modèles de référence qui en sont issus.

- Cette architecture inclut un service de vérification présentant des interfaces génériques d'invocation opérationnelle et de gestion.
- Cette architecture permet également le développement de composants adaptateurs en vue de favoriser l'intégration avec l'existant. Comme les entrées du service de vérification sont conformes à MeCSV, l'adaptateur permet de faire la passerelle entre les modèles de données de l'existant et les modèles conformes à MeCSV.

Le modèle de référence, construit en phase de conception, est exploité par le service de vérification pour vérifier les configurations candidates, Figure 21b. Deux types de processus de vérification peuvent être identifiés : lorsque le système de gestion interagit directement avec le service ou le système de gestion s'intègre au service de vérification via un adaptateur.

Dans le premier cas, nous avons (Figure 21b – étapes 1 et 2):

- Demande d'évaluation de configurations : le système de gestion (module de décision) choisit une configuration à appliquer, il envoie une requête de vérification de configurations au service de vérification.
- Collecte des données d'état courant : Cette requête contient l'instance de configuration à vérifier ainsi que les valeurs courantes d'attributs d'état opérationnel récupérées auprès du module de supervision du système de gestion.
- Exécution de la vérification : le service exécute la vérification et renvoie les résultats.

Ce processus est légèrement modifié (Etapes 1a-1b et 2a-2b de la Figure 21b) dans le cas où une intégration d'un système de gestion spécifique existant est réalisée via un composant adaptateur. Les requêtes de vérification sont effectuées au niveau de l'adaptateur qui fait le lien entre ce système de gestion particulier et le service de vérification.

III.2.2. **Un métamodèle de spécification et de vérification de configurations**

Le métamodèle MeCSV permet de modéliser la configuration d'un domaine géré, d'exprimer les contraintes garantissant la conformité structurelle et l'applicabilité opérationnelle, et de renseigner les paramètres d'état utiles à l'évaluation de ces dernières. Une présentation détaillée des concepts constructeurs du métamodèle fait l'objet du Chapitre 4.

– Représentation de l'information de configuration pour la vérification

Pour la représentation de l'information de configuration, nous avons étudié la représentation habituelle des modèles de configuration dans les langages d'information de gestion tels que CIM et YANG. MeCSV permet de représenter une configuration et ses paramètres de configurations, d'exprimer des dépendances et de composer des configurations entre elles. L'utilisation de métadonnées de configurations permet en général d'apporter des sémantiques utiles au processus de reconfiguration, par exemple marquer une configuration de «configuration par défaut» ou «configuration courante». MeCSV fournit également la possibilité de définir ces métadonnées.

– Expression de contraintes

La vérification que nous proposons se base sur l'expression et l'évaluation de contraintes.

Définition 18 ***Contrainte*** *: une contrainte est une condition à satisfaire. C'est une règle sous forme d'une expression booléenne qui restreint les modifications possibles d'un élément ; son évaluation retourne vrai ou faux [UML, 2007 ; Sun, 2006].*

Pour répondre aux besoins de vérification opérationnelle, nous avons identifié et défini deux types de contraintes : les contraintes *offline* qui conditionnent la structure et la composition des informations de configuration et les contraintes *online* qui garantissent son applicabilité au regard du contexte opérationnel courant. L'expression de ces deux types de contraintes permet de couvrir les deux niveaux de vérification que nous avons identifiés, la vérification des contraintes *offline* permet de garantir la conformité structurelle des configurations et celle des contraintes *online* permet d'assurer leur applicabilité opérationnelle.

– Spécification de paramètres d'état opérationnel

L'évaluation des contraintes *online* requiert la connaissance de valeurs de données d'état. MeCSV fournit des constructeurs pour spécifier ces paramètres d'état qui vont être utilisés dans les contraintes. Cela permet de lier la fonction de vérification à la supervision afin de prendre en compte l'état opérationnel courant.

La définition du métamodèle MeCSV permet de disposer d'un langage de haut niveau de spécification de configurations, de contraintes et de paramètres d'état à surveiller. Il permet la construction d'un modèle de référence du domaine géré qui va servir de support à la vérification qui en découle.

III.2.3. Construction du modèle de référence MeCSV

La construction du modèle de référence est une phase clé du processus de mise en œuvre de la vérification opérationnelle. Si une vision idéale est qu'il puisse être fourni par le constructeur ou le vendeur du système configurable, dans le cadre d'un système géré existant, sa définition peut être obtenue de deux manières :

- Soit il n'existe pas de modèle d'information de gestion du système géré, dans ce cas, le modèle de référence est construit en totalité, sans référence à un existant (*ex nihilo*).
- Soit il existe un modèle d'information de gestion, alors des chaînes de transformation peuvent être définies entre le modèle d'information existant et MeCSV afin de générer automatiquement le modèle de référence.

Ces deux types de construction de modèles de référence ont été expérimentés sur deux cas d'études qui seront présentés dans le Chapitre 5.

Le modèle de référence est un élément central du processus de vérification, il est dédié à un domaine applicatif donné (réseau, intergiciels, applications web etc.). Ainsi toute instance de configuration qui lui est conforme peut dès lors être vérifiée en cours d'exécution, par une plate-forme conforme à l'architecture présentée ci-après.

III.2.4. Une architecture de vérification en ligne

Nous avons également défini une architecture supportant la vérification opérationnelle de configurations. Cette architecture fournit un moteur de vérification et des interfaces d'utilisation bien définies qui permettent d'invoquer et d'adapter la vérification selon les scénarii d'usage. L'architecture de vérification opérationnelle est approfondie dans le Chapitre 6.

– Un moteur de vérification prenant en compte les conditions opérationnelles

Le cadriciel comprend un moteur de vérification qui sait interpréter tout modèle de référence et évaluer toute instance de configuration qui lui est conforme. Ce processus est automatique

et offre un véritable service support à des interactions de forme requête-réponse. Le moteur de vérification est au cœur du service de vérification, les systèmes de gestion sont ses clients. Il propose des interfaces d'interrogation et de gestion pour supporter l'adaptation de la vérification.

– **Une interface flexible de vérification**

Face aux exigences d'adaptation et d'évolution de la vérification opérationnelle, nous avons discerné un besoin de modulation de la vérification, par exemple pouvoir vérifier une configuration partielle ou ne vérifier qu'un certain jeu de contraintes. Nous avons défini une interface offrant l'expression de différents niveaux de granularités de vérification.

– **Une interface d'édition du modèle de référence MeCSV**

Un autre besoin d'adaptation concerne l'évolution de la vérification pour supporter la dynamicité opérationnelle, par exemple l'ajout ou la suppression de contraintes, leur renforcement ou leur relâchement. Pour ce faire, nous avons spécifié une interface d'édition opérationnelle d'un modèle de référence.

Le moteur de vérification propose un service de vérification en ligne de configurations d'une part grâce à l'exploitation du modèle de référence et d'autre part grâce à son articulation avec le système de supervision existant. La définition d'une interface d'invocation flexible et de gestion opérationnelle du modèle de référence permet de poser les bases importantes d'une vérification adaptable et évolutive.

III.2.5. **La prise en compte de l'existant**

La définition du métamodèle MeCSV permet d'atteindre une indépendance vis-à-vis des plates-formes de gestion dans l'expression du modèle de référence et dans la vérification des instances de configuration. Afin de prendre en compte l'existant, il faut supporter une intégration avec les systèmes de gestion existants. Le cadriciel inclut à cet effet une architecture pour le développement de composants adaptateurs. Pour utiliser le service de vérification avec un système de gestion donné, il faut implémenter un adaptateur qui convertit l'interface générique du service en interface spécifique au système de gestion. L'intégration avec l'existant fait l'objet du Chapitre 6.

III.3. **Conclusion**

L'objectif de nos travaux est de spécifier un cadriciel pour la vérification de reconfigurations dynamiques dans le domaine de la GRS. Selon nous, une vérification dans ce contexte doit tenir compte des conditions opérationnelles courantes, doit être flexible afin de s'adapter aux scénarii de fonctionnement et doit être évolutive pour prendre en compte la dynamicité des éléments gérés. De plus, cette vérification doit pouvoir prendre en compte l'existant. Un cadriciel a été défini pour répondre à ces exigences :

- En phase de conception : la définition du métamodèle MeCSV permet une création outillée de modèles de références décrivant les informations de configurations, d'état opérationnel et de contraintes nécessaires à la vérification. La présentation détaillée des constructeurs de MeCSV et son utilisation effective sur deux cas d'études font respectivement l'objet des chapitres 4 et 5.

- En phase d'exécution : la conception d'une architecture de vérification en ligne rend possible une vérification en fonction du contexte d'exécution. Cette architecture est capable de supporter une vérification flexible : la définition de plusieurs niveaux de granularités de vérification permet une vérification plus souple et personnalisée ; la spécification d'une interface d'édition opérationnelle des modèles de référence ouvre la voie à une vérification évolutive. Cette partie est détaillée dans le chapitre 6.

- Enfin, l'intégration opérationnelle de l'existant est supportée via la disponibilité de composants adaptateurs, passerelles entre les modèles d'informations existants et les modèles conformes à MeCSV. L'architecture d'un adaptateur est également présentée au chapitre 6.

Les différentes utilisations du cadriciel, positionnées dans le cycle d'un système informatique, sont résumées dans la Figure 22.

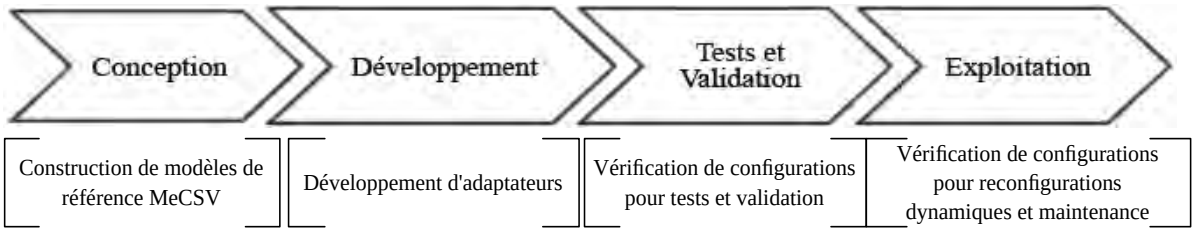

Figure 22 – Positionnement du cadriciel dans le cycle de vie des systèmes gérés

Chapitre 4

MeCSV : un métamodèle dédié à la spécification et à la vérification de configurations

Pour permettre l'utilisation du cadriciel en phase de conception, nous avons défini MeCSV, un métamodèle dédié à la spécification de configurations et de contraintes. MeCSV fournit des concepts constructeurs de haut niveau pour la définition de modèles d'information de configuration, d'information d'état opérationnel et de contraintes. Il permet ainsi la définition en phase de conception de modèles de référence qui vont servir à évaluer les configurations candidates en phase d'exécution. Ce chapitre présente les différents concepts constructeurs du métamodèle. MeCSV a été implémenté sous forme de profil UML et testé dans des modeleurs UML existants avec succès.

IV.1. **Présentation générale du métamodèle MeCSV**

La définition du métamodèle MeCSV répond à la nécessité de disposer d'un langage de spécification de configuration à des fins de vérification. Pour la mise en œuvre du métamodèle, nous nous sommes appuyés sur les méthodes et outils de l'Ingénierie Dirigée par les Modèles (IDM). Les définitions suivantes présentent les concepts de l'IDM qui nous ont été utiles pour la mise en œuvre de MeCSV. Elles sont tirées de [Combemale, 2008].

Définition 19 ***Ingénierie Dirigée par les Modèles*** *: l'IDM, est «une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles». Le principe fondamental de l'IDM est le découplage de la logique métier de la mise en œuvre technologique via la définition systématique de modèles métier. Des techniques de création et de transformation de modèles permettent de rendre les modèles productifs à des fins de génération de code, d'exécution, de vérification etc. La définition de langages d'expression de ces modèles s'appelle la métamodélisation.*

Définition 20 ***Modèle*** *: «un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière.»*

Définition 21 ***Métamodèle*** *: «un métamodèle est un modèle qui définit le langage d'expression d'un modèle. Le métamodèle représente le langage de modélisation». Un métamodèle est composé de concepts et de règles métier.*

Définition 22 ***Langage de modélisation dédié :*** *La définition de langages de modélisation dédiés ou DSML (Domain Specific Modeling Language) est une technique de métamodélisation prônée par l'IDM. Elle permet «d'offrir aux utilisateurs des concepts propres à leur métier et dont ils ont la maîtrise» sous forme de constructions de langage. Par exemple, l'utilisation de concepts constructeurs métier tels que* `<configuration>`*,* `<paramètre de configuration>` *au lieu de concepts généraux comme* `<classe>` *ou* `<attribut>`*.*

L'adoption d'une approche IDM nous a permis de nous affranchir des spécificités techniques et protocolaires des plates-formes de gestion et de nous concentrer sur les concepts propres à la vérification de configurations dans un contexte de reconfiguration dynamique. MeCSV a été conçu comme un DSML dédié à la vérification opérationnelle de configurations. Il fournit des concepts constructeurs organisés en trois catégories, chacune dédiée à un objectif spécifique présenté dans la Figure 23 :

- la première catégorie, (partie a), permet la définition de modèles de configuration,
- la deuxième catégorie, (partie b), permet la représentation de paramètres d'état opérationnel en vue de la connexion au module de supervision,
- et la dernière catégorie, (partie c), fournit des constructeurs pour exprimer les contraintes *offline* et *online*.

En phase de conception, le concepteur utilise MeCSV pour définir le modèle de configuration du système géré, représenter le modèle d'informations d'état opérationnel à surveiller et exprimer les contraintes qui doivent être respectées. Le résultat de cette modélisation donne le modèle de référence. Ce dernier est utilisé tel quel en phase d'exécution pour vérifier les configurations candidates. Il permet d'assurer ainsi une vérification opérationnelle indépendante des plates-formes.

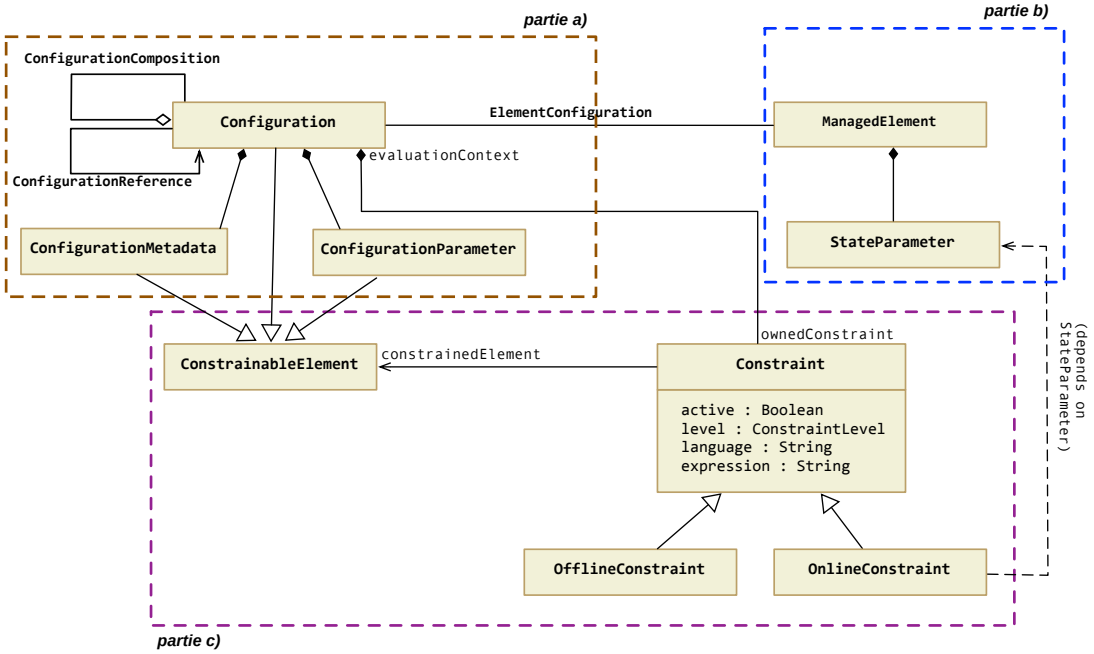

Figure 23 - Concepts constructeurs fondamentaux du métamodèle MeCSV

Nous allons définir ces concepts constructeurs dans la section suivante en s'appuyant sur un exemple décrit ci-après.

– Exemple d'application

Afin d'illustrer les divers concepts de MeCSV, nous introduisons un exemple d'application à travers la construction du modèle de référence d'un middleware orientée messages (MOM).

Un MOM est globalement un logiciel qui permet à différentes applications, éventuellement réparties, de communiquer et d'interopérer l'échange de messages. La plupart des MOM proposent deux modes de communications :

- Le mode point-à-point qui repose sur le concept de file de messages ou *queue*. Un message émis est stocké dans une file de messages ou *queue* jusqu'à sa consommation.
- Le mode publication/souscription qui est basé sur le concept de sujet de discussion ou *topic*. Les applications s'abonnent à un *topic* et reçoivent tout message envoyé à ce *topic*.

Une plate-forme MOM est composée d'un ou plusieurs serveurs de messages ou *brokers* avec lesquels les applications clientes communiquent. Un serveur de messages gère les files de messages et les sujets de discussion et fournit les services nécessaires à la gestion des connexions et des échanges de messages entre applications clientes. De façon simplifiée, configurer un serveur de messages consiste à définir ses paramètres de configuration internes, à décrire la configuration des services, des *queues* et des *topics* qu'il héberge.

IV.2. **Présentation détaillée des concepts constructeurs**

IV.2.1. **Définition de modèles de configuration**

La partie a) de la Figure 23 montre les concepts constructeurs de MeCSV dédiés à la définition de modèles de configuration. Ces concepts offrent une capacité de représentation de haut niveau des informations de configuration d'un système géré. Cette représentation est primordiale pour que la fonction de vérification que nous proposons puisse comprendre et traiter de façon générique, les instances de configuration qui lui seront envoyées. Dans ce qui suit, nous détaillons les concepts constructeurs de MeCSV et les illustrons sur l'exemple du MOM.

– Description des concepts constructeurs

L'unité de base de la définition d'un modèle de configuration est la classe Configuration. Cette classe est un conteneur logique qui permet de rassembler de façon cohérente des paramètres de configuration d'éléments gérés ainsi que des métadonnées nécessaires au processus de reconfiguration.

- Le paramètre de configuration est l'unité d'information de configuration permettant de définir le comportement désiré du système. Dans MeCSV, le paramètre de configuration est un attribut de classe modélisé avec le constructeur `ConfigurationParameter`.
- La métadonnée de configuration est une information qui permet de qualifier les configurations elles-mêmes, par exemple, qualifier leur état ou leur rôle dans un scénario d'utilisation donné ou par rapport au processus de reconfiguration. Ce sont par exemple, des informations sur le type de configuration, c'est le cas de l'attribut «par défaut» (*default*) ou «courant» (*current*) dans CIM et dans YANG pour qualifier les instances de configuration par défaut ou courantes. Dans MeCSV, la métadonnée de configuration est un attribut de classe représenté par le constructeur `ConfigurationMedata`.

Nous avons dégagé de l'étude de différents langages de configuration [Cons & Poznanski, 2002 ; Anderson & Scobie, 2002 ; Kanies, 2006 ; Goldsack et al., 2009 ; RFC6020] et de configurations existantes de systèmes, deux relations fondamentales pour la définition de modèles de configuration, à savoir : la relation de dépendance structurelle entre configurations et la relation de composition de configurations.

- Relation de dépendance : une relation de dépendance structurelle entre deux configurations reflète une relation de dépendance entre les éléments gérés qu'elles caractérisent. Plus précisément, elle signifie que les paramètres de configuration de l'une référencent des paramètres de configuration de l'autre. Une association `ConfigurationReference` entre deux classes `Configuration` permet de modéliser ce type de relation.

- Relation de composition : une relation de composition permet le découpage d'une configuration en plusieurs sous-configurations, par exemple pour des raisons de modularité et de réutilisabilité. Une relation d'agrégation `ConfigurationComposition` entre classes `Configuration` permet de représenter une relation de composition.

– Application : définir le modèle de configuration d'un serveur de messages avec MeCSV

La Figure 24 montre l'utilisation de MeCSV pour modéliser l'information de configuration d'un serveur de messages à travers un diagramme de classes UML simplifié. Des stéréotypes sont appliqués aux éléments du diagramme pour la représentation des concepts constructeurs introduits précédemment. Ces stéréotypes sont présentés en détail dans la section IV.3.

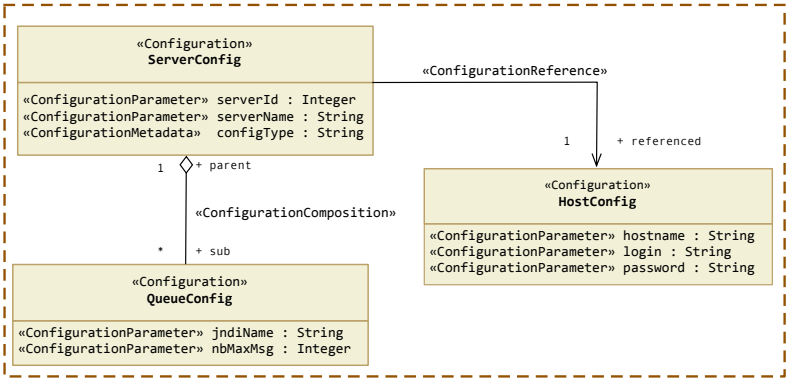

Figure 24 - Extrait du modèle de configuration d'une plate-forme de type MOM

La configuration du serveur de message (`ServerConfig`) est caractérisée par ses paramètres de configuration internes tels que son identifiant (`serverId`) ou son nom (`serverName`) et une métadonnée (`configType`) qui qualifie son type, par exemple «configuration par défaut». La configuration du serveur contient également une sous-configuration, celle de ses files de messages (`QueueConfig`). La configuration du serveur référence la configuration de la machine-hôte (`HostConfig`) qui l'héberge.

IV.2.2. Représentation d'informations d'état opérationnel

La représentation d'informations d'état opérationnel découle de la nécessité d'avoir accès à la situation opérationnelle du système géré pour la vérification de l'applicabilité opérationnelle. Pour ce faire, MeCSV propose deux constructions, `ManagedElement` et `StateParameter`, illustrées dans la partie b) de la Figure 23.

– Description des concepts constructeurs

Le concept d'élément géré est traditionnellement présent au sein des modèles d'information et de données de gestion et représente la vue logique d'une ressource gérée [Stallings, 1999 ; Clemm, 2006]. Ce concept est également présent dans MeCSV à travers la classe `ManagedElement`. Il s'est révélé essentiel afin de lier les configurations à l'élément géré qu'elles caractérisent (relation `ElementConfiguration`) et d'avoir accès aux informations de son état opérationnel.

Les paramètres d'état opérationnel sont qualifiés par le constructeur `StateParameter` et sont des attributs de la classe `ManagedElement`. Un `StateParameter` représente l'unité d'information d'état opérationnel, variable dans le temps *at runtime*. Il doit donc provenir de l'observation du fonctionnement de l'élément géré par le module de supervision existant.

Il ne s'agit pas de représenter une vue opérationnelle complète du système géré mais de renseigner les paramètres d'état opérationnel qui peuvent influencer des valeurs de configuration.

Ces deux constructions permettent la modélisation des informations d'état opérationnel utiles à la vérification opérationnelle. Le système de vérification peut ainsi communiquer avec le module de supervision du système de gestion afin de procéder à l'évaluation de l'applicabilité opérationnelle des configurations.

– Application : spécifier l'état opérationnel d'un serveur de messages avec MeCSV

La Figure 25 illustre l'utilisation de MeCSV pour modéliser la vue opérationnelle d'un serveur de messages et d'une *queue* à travers un diagramme de classe UML stéréotypé.

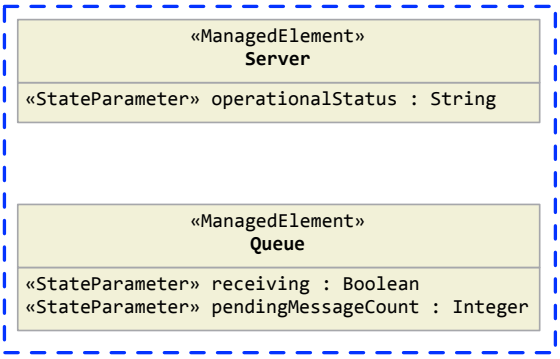

Figure 25 - Vue des paramètres d'état opérationnel concernés

Cette vue opérationnelle indique que le concepteur a référencé le statut opérationnel courant d'un serveur de messages (`operationalStatus`), l'état d'utilisation de ses files de messages (`receiving`) et le nombre courant de messages stockés en mémoire (`pendingMessageCount`) comme des informations d'état opérationnel susceptibles d'influencer la validité d'une configuration candidate *at runtime*. Ils serviront à exprimer des contraintes.

IV.2.3. Expression de contraintes

La partie c) de la Figure 23 présente les concepts constructeurs qui permettent d'exprimer et de gérer les contraintes à vérifier sur les configurations. Ces concepts constructeurs sont enrichis d'attributs supplémentaires pour supporter l'adaptation et l'évolution des spécifications de contraintes en fonction des scénarii de fonctionnement.

– Description des concepts constructeurs

Le constructeur `Constraint` est le concept de base pour l'expression de contraintes. Afin de supporter la vérification de configurations, les éléments de configuration, à savoir les paramètres de configuration (`ConfigurationParameter`), les configurations (`Configuration`) et les métadonnées de configuration (`ConfigurationMetadata`) sont rendus «contraignables» (constructeur `ConstrainableElement`), c'est-à-dire que leur valeur peut être restreinte par une contrainte.

Une contrainte est définie dans le cadre d'une configuration qui lui sert de contexte d'évaluation (relation `evaluationContext - ownedConstraint`).

Elle est exprimée sous forme textuelle (`expression`) dans un langage choisi (`langage`). Ce langage peut être un langage de programmation général (ex. Java) ou un langage de contraintes (ex. OCL).

Le constructeur `Constraint` se distingue en deux sous-types : les contraintes *offline* (constructeur `OfflineConstraint`) et les contraintes *online* (constructeur `OnlineConstraint`), dédiées respectivement à la vérification de l'intégrité structurelle et à la vérification de l'applicabilité opérationnelle de configurations.

Définition 23 ***Contrainte offline*** *: une contrainte offline restreint les valeurs possibles d'un élément de configuration par rapport à d'autres éléments de configuration ou à des constantes fixées. Son évaluation ne nécessite aucune acquisition d'informations d'état opérationnel. Les contraintes offline encadrent la conformité structurelle d'une configuration, à savoir le respect des types de données, de l'intervalle de valeurs autorisées, la consistance de la composition de paramètres et des dépendances. Leurs violations participent aux erreurs illégales [Oppenheimer et al., 2003 ; Gençay et al., 2008].*

Les contraintes *offline* sont qualifiées par le constructeur `OfflineConstraint`. Elles sont vérifiables pour la plupart en phase de conception ou avant la mise en service, notamment dans le cas de configurations prédéfinies.

Définition 24 ***Contrainte online*** *: une contrainte online est une contrainte qu'on ne peut vérifier qu'en cours d'exécution parce qu'elle confronte les valeurs possibles d'un élément de configuration à des données d'état opérationnel. Les contraintes online encadrent l'applicabilité opérationnelle, elles permettent de vérifier les valeurs de configuration par rapport à l'état courant du système géré en cohérence avec les objectifs de service. Ces contraintes permettent de prévenir les erreurs dites légales.*

Le constructeur `OnlineConstraint` permet de les exprimer. Une contrainte qualifiée de `OnlineConstraint` fait obligatoirement intervenir un paramètre d'état opérationnel dans son expression.

Gestion de cycle de vie des contraintes

Nous avons enrichi le concept de contrainte de deux attributs pour permettre une expression de contraintes au plus près du mécanisme de reconfiguration et du fonctionnement du système géré. Ces deux constructeurs permettent notamment de renforcer ou de relâcher des contraintes.

- L'attribut `active` ajoute le concept d'état à une contrainte. Une contrainte possède désormais un état, elle peut être active ou inactive. L'ajout de ce constructeur provient du constat que la pertinence de certaines contraintes dépend des scénarii de fonctionnement. Ce constructeur offre la possibilité d'activer ou de désactiver des contraintes et donc d'adapter leur vérification à un contexte opérationnel donné.

- L'attribut `level` permet de moduler les niveaux de sévérité ou de priorité d'une contrainte. A l'inverse de la vision habituelle de la contrainte comme une règle statique dont l'évaluation renvoie vrai ou faux, nous avons remarqué que certaines contraintes sont plutôt des préférences dont le niveau de sévérité peut être modulé en fonction des objectifs de métier. Ce constructeur permet d'augmenter l'expressivité d'une contrainte au delà des formes les plus courantes à savoir «optionnel» et «obligatoire», et donc de personnaliser leur sévérité et/ou priorité.

– **Application : exprimer des contraintes sur un serveur de messages avec MeCSV**

La Figure 26 montre une représentation de contraintes *offline* et *online* sur la configuration de la file de messages de la Figure 24 à l'aide du langage OCL.

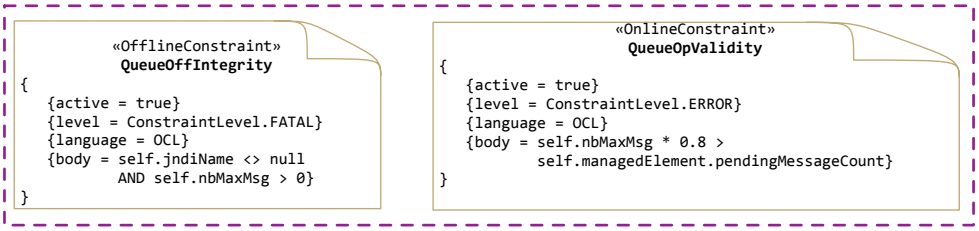

Figure 26 - Exemples de contraintes *offline* et *online*

La contrainte `QueueOffIntegrity` est un exemple de contrainte *offline*, elle vise à préserver l'intégrité structurelle de la configuration d'une *queue*. Dans cet exemple, l'attribut `jndiName` (nom de la *queue* dans l'annuaire de noms) doit être renseigné, ou encore l'attribut `nbMaxMsgs` fixant le nombre maximum de messages que la *queue* peut stocker en mémoire doit être supérieur à 0. Par exemple, sans système de vérification, l'application d'une configuration de *queue* violant ces contraintes rend la *queue* inutilisable.

Quant à la contrainte `QueueOpValidity`, elle empêche les configurations candidates de file de messages d'avoir un nombre de messages courant en mémoire supérieur à 80% de la limite autorisée. C'est un exemple de contrainte *online*. Cette contrainte compare la nouvelle valeur du paramètre de configuration `nbMaxMsgs` à appliquer à la valeur courante du nombre de messages (`pendingMessageCount`) afin de prévenir toute inconsistance ou problème de performance. La contrainte `QueueOpValidity` est un exemple de préférence, elle peut être importante dans un contexte d'optimisation de la performance du système géré. L'attribut `level` permet de moduler sa sévérité suivant les contextes opérationnelles. Elle peut également être désactivée.

IV.2.4. **Modèle de référence résultant**

Un extrait du modèle de référence obtenu est présenté dans la Figure 27. Il contient les trois catégories de modèles présentées respectivement dans les figures 24, 25 et 26, à savoir d'une part le modèle de configuration de la plate-forme MOM et le modèle des informations d'état opérationnel qui permettent de structurer les échanges d'informations de gestion manipulées (configurations candidates, valeurs opérationnelles) et d'autre part les contraintes *offline* et *online* qui encadrent la validité des configurations candidates.

Grâce à MeCSV, le modèle de référence ainsi défini va servir de support des échanges d'informations entre le système de vérification et un système de gestion donné de façon indépendante des plates-formes technologiques. Quels que soient les domaines applicatifs modélisés et les systèmes de gestion considérés, la généricité de notre approche est maintenue parce que tout modèle de référence défini est manipulable comme tel par le système de vérification que nous proposons.

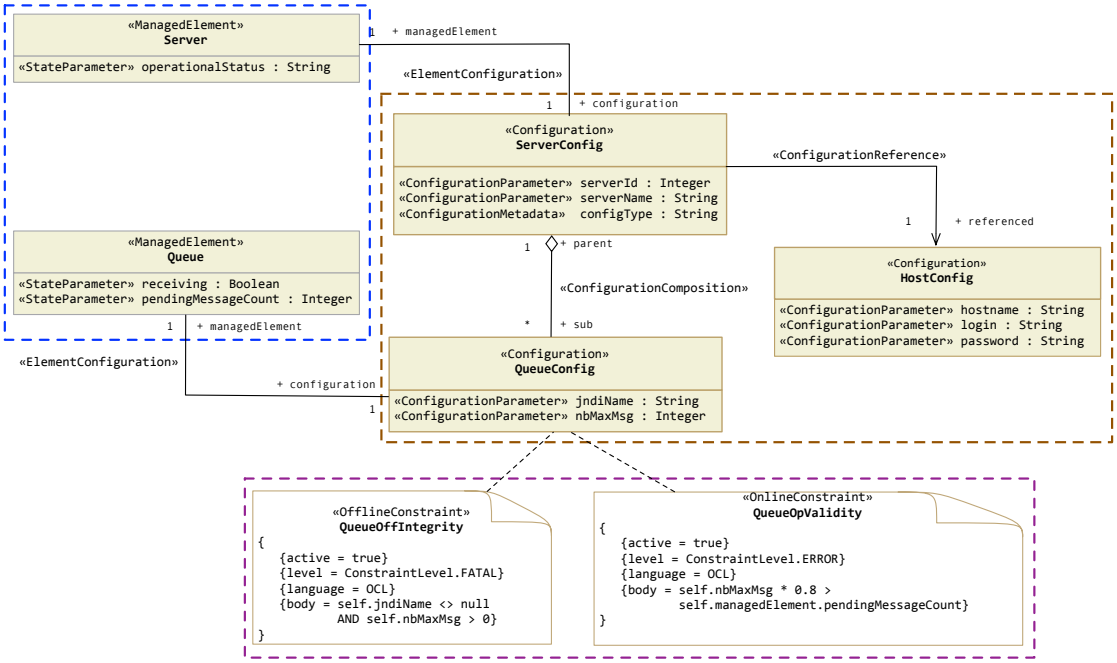

Figure 27 - Extrait du modèle de référence résultant

IV.3. Outillage et implémentation

Afin de tester les constructions du langage et de valider leur capacité à couvrir les besoins de spécification de configurations réelles, nous avons implémenté le métamodèle MeCSV sous forme de profil UML. Cette section présente cette mise en œuvre ainsi que les outils et techniques sur lesquels nous nous sommes appuyés pour ce faire.

De façon générale, chaque concept constructeur a été implémenté sous forme de stéréotype de même nom. Des contraintes OCL encadrent le bien-formé et la sémantique d'utilisation des éléments du profil MeCSV ainsi défini. Pour la mise en œuvre technique du profil, nous avons utilisé la plate-forme Eclipse MDT. Le profil UML de MeCSV est disponible sous forme de *plugin* Eclipse. Son utilisation a été validée dans des éditeurs de modèles compatibles tels que `Eclipse MDT` [Eclipse-MDT] ou `TopCased` [TOPCASED].

IV.3.1. Concepts et terminologies

– Mécanisme de profil UML

UML est le langage de modélisation le plus utilisé dans la conception orienté-objet d'applications logicielles. Standardisé par l'OMG en 1997, UML est à la version 2.4. UML propose une technique de métamodélisation via le mécanisme de profils UML.

Un profil UML est un mécanisme d'extension qui permet d'adapter les concepts constructeurs généraux du métamodèle UML à un domaine métier. Pour ce faire, le métamodèle UML introduit des constructeurs de base comme les stéréotypes qui permettent d'étendre les méta-classes UML de base avec des concepts métiers. Les stéréotypes peuvent éventuellement posséder des attributs encore appelés méta-attributs ou valeurs étiquetées (*taggued values*). La définition de contraintes additionnelles sur les méta-classes UML permet de compléter et de garantir la sémantique du langage de modélisation dédié ainsi obtenu.

Plusieurs profils UML ont été définis, standardisés et largement utilisés, par exemple le profil UML des EJB (Enterprise Java Beans) pour les architectures JEE (Java Enterprise Edition), le

profil UML des systèmes embarqués temps-réels MARTE (Modeling and Analysis of Real Time Embedded Systems) [OMG-UMLProfiles, 2013] ou encore le profil UML pour CIM [DMTF-CIMUMLProfile, 2009].

La Figure 28 montre un exemple de stéréotype (`<<Configuration>>`) qui étend une méta-classe (`Class`) du métamodèle UML. On dit aussi que le stéréotype `<<Configuration>>` a pour classe de base la méta-classe UML `Class`. Cela signifie que toute classe d'un diagramme de classes peut être annotée du stéréotype `<<Configuration>>`. La classe ainsi annotée représente une configuration.

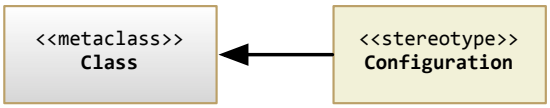

Figure 28 - Exemple montrant un stéréotype qui étend une méta-classe

La définition de profils UML permet à l'utilisateur final de manipuler directement les concepts métier dont il a la maîtrise sans changer les modeleurs UML dont il a l'habitude. Il nous a paru pertinent d'implémenter le métamodèle MeCSV sous forme de profil UML afin de bénéficier de la popularité d'UML et permettre ainsi l'utilisation de MeCSV dans les modeleurs UML existants.

– Présentation du langage OCL

OCL est un langage de spécification de contraintes standardisé par l'OMG pour répondre au manque de précision des diagrammes UML. Partant du constat que les langages formels traditionnels sont difficiles à appréhender pour l'utilisateur moyen (qui ne possède pas de solides bagages mathématiques), OCL a été développé en vue de fournir un langage mathématique rigoureux et simple pour la définition de contraintes sur des modèles UML. «*C'est un langage formel qui se veut simple et facile à lire et écrire*» [OCL, 2006]. En tant que langage de spécification, son évaluation n'a pas d'effets de bord, elle renvoie simplement vrai ou faux. La version courante d'OCL est la version 2.3.
Notre choix d'un langage de contraintes s'est donc porté sur OCL du fait de sa simplicité et parce qu'il est spécialement dédié à la spécification rigoureuse de modèles UML.

IV.3.2. Définition du profil UML pour MeCSV

IV.3.2.1. Présentation détaillée des stéréotypes et des méta-classes

Les méta-classes UML suivantes `Class`, `Association`, `Property` et `Constraint` ont été utilisées dans la définition du profil UML pour MeCSV. Des contraintes supplémentaires OCL ont été exprimées sur ces méta-classes pour les adapter à la définition de modèles de référence MeCSV. Le profil UML pour MeCSV est illustré dans la Figure 29.

De façon générale, un modèle de classes UML après application du profil MeCSV est un modèle de référence MeCSV si :

- ses éléments représentent des classes, des associations, des attributs de classe et des contraintes.
- ses éléments sont annotés des stéréotypes présentés ci-après.
- et ses éléments satisfont les contraintes, exprimées ci-dessous en langage naturel, encadrant l'utilisation du métamodèle MeCSV.

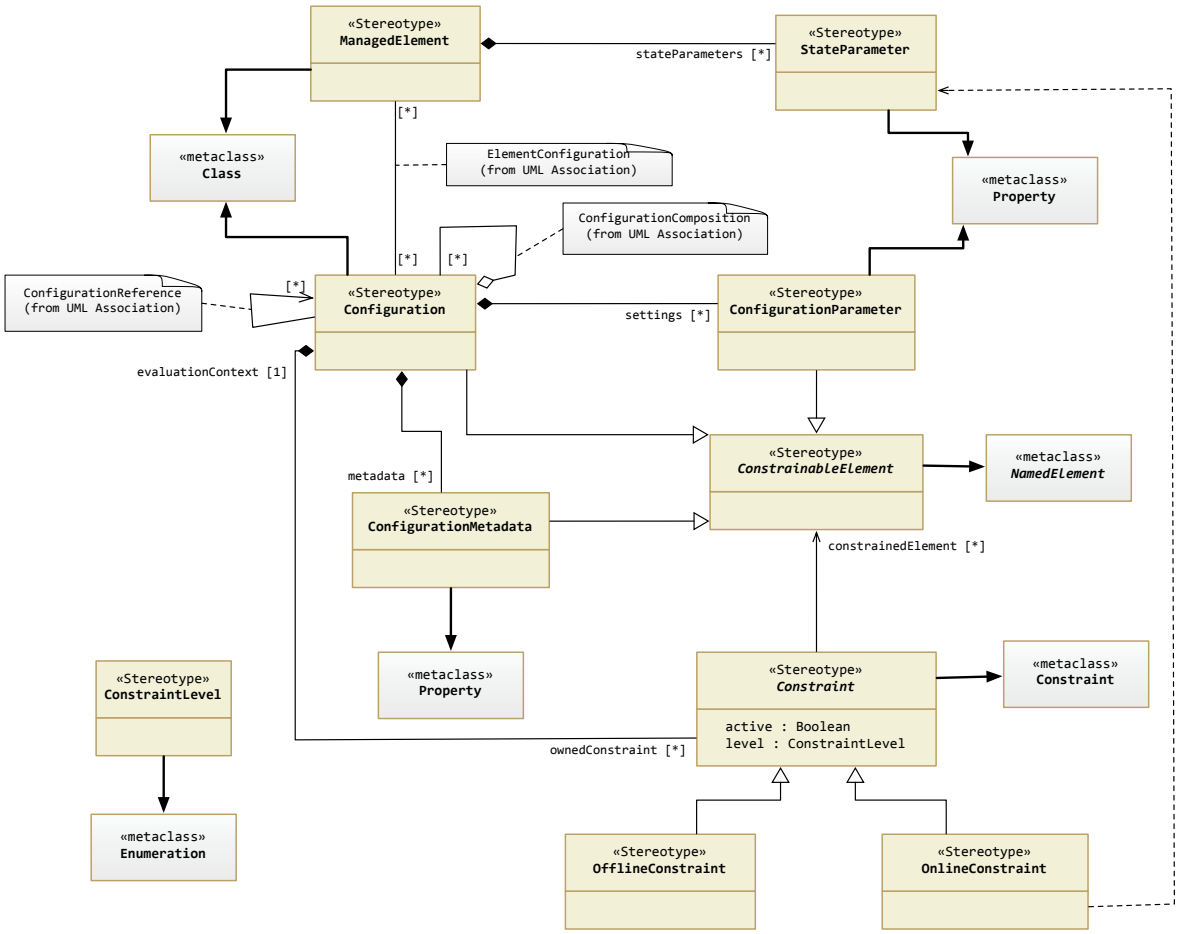

Figure 29 - Profil UML pour MeCSV

– Définition des modèles d'information de configuration

Le stéréotype `<<Configuration>>` étend la méta-classe UML `Class` et permet de représenter des configurations. Les stéréotypes `<<ConfigurationParameter>>` et `<<ConfigurationMetadata>>` étendent la méta-classe UML `Property`, ils indiquent respectivement qu'un attribut de classe est un paramètre ou une métadonnée de configuration.

La définition des modèles d'information de configuration est encadrée par les contraintes suivantes (exprimées en langage naturel) :

- Seules les classes annotées `<<Configuration>>` peuvent posséder des attributs stéréotypés `<<ConfigurationParameter>>` ou `<<ConfigurationMetadata>>`.
- Le stéréotype `<<ConfigurationComposition>>` et le stéréotype `<<ConfigurationReference>>` ont pour classe de base la méta-classe `Association`.
- Les associations étiquetées `<<ConfigurationComposition>>` et `<<ConfigurationReference>>` ne peuvent être définies qu'entre des classes annotées `<<Configuration>>`.
- Une association tagguée `<<ConfigurationComposition>>` est obligatoirement une agrégation.
- Une association annotée `<<ConfigurationReference>>` est navigable de la classe `<<Configuration>>` vers la classe `<<Configuration>>` `destination`.

– **Représentation des informations d'état**

Le stéréotype <<ManagedElement>> a pour classe de base la méta-classe Class. Le stéréotype <<StateParameter>> étend la méta-classe Property.

La représentation des informations d'état avec le profil MeCSV doit respecter les contraintes suivantes :

- Les attributs étiquetés <<StateParameter>> sont obligatoirement des attributs de classes étiquetées <<ManagedElement>>.
- Une association annotée <<ElementConfiguration>> ne peut être définie qu'entre une classe annotée <<Configuration>> et une autre annotée <<ManagedElement>>.

– **Expression des contraintes**

Le stéréotype <<Constraint>> étend la méta-classe UML Constraint et permet de représenter des contraintes. Il est enrichi des attributs active et level qui ont été implémentés sous forme de méta-attributs. Ce stéréotype est abstrait, il se subdivise en stéréotypes <<OfflineConstraint>> et <<OnlineConstraint>>.

L'expression de contraintes doit obéir aux règles suivantes :

- Le contexte d'évaluation d'une contrainte doit être une classe stéréotypée <<Configuration>>.
- Seuls les éléments de modèles étiquetés <<ConfigurationParameter>>, <<ConfigurationMetadata>> et <<Configuration>> peuvent être contraints.
- Les contraintes annotées <<OfflineConstraint>> ne doivent pas référencer d'attribut stéréotypé <<StateParameter>>.
- Les contraintes stéréotypées <<OnlineConstraint>> doivent référencer au moins un attribut stéréotypé <<StateParameter>> dans leur expression.

IV.3.2.2. Mise en oeuvre technique

L'implémentation technique a été réalisée avec la plate-forme Eclipse MDT - Papyrus via la technologie des profils statiques EMF/Ecore.

Eclipse MDT est un projet de la plate-forme open-source Eclipse qui porte le développement, et la promotion d'outils pour exploiter l'approche IDM. Cette plate-forme est construite sur un cadriciel de base appelé EMF (Eclipse Modeling Framework).

Outil emblématique de l'IDM, EMF est un cadre de métamodélisation qui offre des librairies et outils logiciels pour la manipulation et l'utilisation de modèles comme *inputs* au développement et intégration d'outils logiciels. EMF définit un métamodèle Ecore qui sert de base aux outils et librairies que le cadre EMF fournit. Il existe par exemple des implémentations Ecore des standards UML et du langage de contraintes OCL. Les modèles UML ou Ecore ainsi que les contraintes OCL deviennent des éléments productifs manipulables et exécutables.

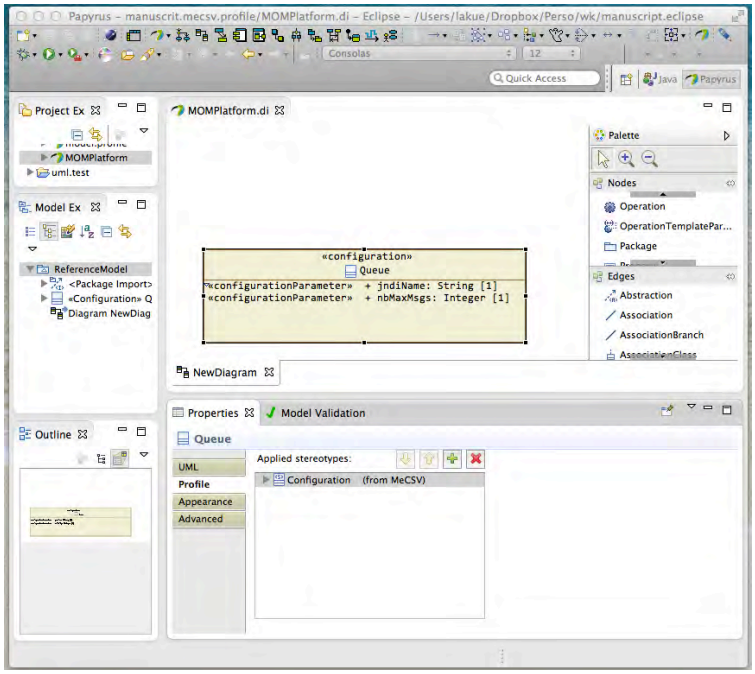

Figure 30 - Ecran de définition d'un modèle de référence avec le profil UML

EMF propose également un éditeur de modèle qui permet une implémentation de profils UML. Nous l'avons utilisé pour modéliser les divers stéréotypes, les méta-classes ainsi que les méta-attributs. Le modèle ainsi obtenu a été enrichi de contraintes OCL correspondant aux règles d'utilisation précédemment énoncées. Le profil UML résultant a été exporté sous forme de *plugin* Eclipse. La Figure 30 est une impression d'écran de l'utilisation du profil dans Eclipse MDT. L'installation du *plugin* rajoute des menus contextuels au modeleur UML existant qui permettent l'application de stéréotypes, comme l'application du stéréotype `<<Configuration>>` dans la Figure 30.

IV.4. **Conclusion**

Le cadriciel que nous proposons est fondé sur le métamodèle MeCSV, métamodèle que nous avons créé et dédié à la spécification et à la vérification de contraintes. Ce chapitre nous a permis d'en présenter les concepts constructeurs principaux.

Le métamodèle MeCSV propose trois catégories de concepts constructeurs de haut niveau dédiées à trois objectifs spécifiques : la définition de modèles d'information de configuration, la représentation d'informations d'état opérationnel et l'expression de contraintes *offline* et *online*. MeCSV permet ainsi la définition de modèles de référence pour divers environnements gérés et donc potentiellement pour divers domaines applications.

Afin de tester les concepts constructeurs proposés, le métamodèle a été mis en œuvre sous forme de profil UML. Nous avons notamment utilisé la plate-forme `Eclipse MDT` et son framework `EMF/Ecore` pour définir un profil UML pour MeCSV disponible sous forme de *plugin* Eclipse. L'installation de ce *plugin* dans un modeleur UML compatible, permet de définir des modèles de référence UML productifs, manipulables comme tel en phase d'exécution à des fins d'analyse et de vérification.

Le profil UML pour MeCSV a été validé expérimentalement : nous avons pu construire divers modèles de référence pour différents environnements gérés. Ce processus de définition de modèles de référence est détaillé dans le chapitre suivant.

Chapitre 5

Production outillée de modèles de référence MeCSV

Nous avons présenté dans le chapitre précédent les concepts constructeurs du métamodèle MeCSV permettant de définir des modèles de référence pour des domaines applicatifs donnés. Le modèle de référence est un élément fondamental dans la mise en œuvre de la vérification opérationnelle de configurations. Sa définition permet de représenter à un niveau abstrait et pour un domaine applicatif donné, les informations utiles à la mise en œuvre de la vérification opérationnelle de configurations. Ce chapitre présente le processus de définition de modèles de référence. En suivant ce processus, nous avons défini des modèles de référence pour deux environnements gérés. Pour le premier environnement, le modèle de référence a été construit de toute pièce. Pour le deuxième environnement, le modèle de référence a été obtenu par transformation de modèles de gestion existants CIM.

V.1. **Processus de définition de modèles de référence MeCSV**

L'approche de vérification de la configuration que nous proposons va au-delà de son aspect structurel seul, en identifiant par ailleurs, les paramètres d'état opérationnel qui peuvent influencer sa validité. Autrement dit, nous avons rajouté à la notion de validité d'une configuration, sa validité vis-à-vis de l'état courant de l'environnement géré. Dans ce sens, le modèle de référence, qui va être support de cette vérification est structuré en trois parties :

- le modèle de configuration qui décrit la structure de l'information de configuration du domaine d'application,
- le modèle d'état opérationnel qui identifie et représente les paramètres d'état opérationnel qui vont servir à l'écriture des contraintes *online*,
- le modèle de contraintes qui contient les contraintes *offline* et *online* qui devront être respectées.

Concrètement, les modèles de configuration et d'état opérationnel sont composés d'un ensemble de classes et d'associations au sens UML : ils représentent la vue structurelle du modèle de référence.

La définition du modèle de référence consiste à modéliser chaque partie, elle se décompose donc en trois étapes : ① la définition du modèle de configuration, ② la définition du modèle d'état opérationnel et ③ l'expression du modèle de contraintes.

V.1.1. **Processus de définition de modèles de référence MeCSV ex nihilo**

Cette partie détaille les trois étapes de définition de modèles de référence. Les deux premières étapes correspondent à la modélisation de la vue structurelle du modèle de référence. Concrètement, il s'agit d'utiliser un éditeur de modèles UML et de définir les diagrammes de classes sur lesquelles vont être appliqués les différents stéréotypes du profil UML pour MeCSV dé-

crits dans le chapitre précédent. Ce processus de définition de modèles de référence en trois étapes est illustré dans la Figure 31 sous forme d'un diagramme d'activité utilisant les formalismes SPEM (Software & Systems Engineering Metamodel) [SPEM, 2008].

– **Définition du modèle de configuration**

Cette étape consiste à modéliser les classes de configuration correspondant aux ressources dont on veut vérifier les configurations, leurs paramètres de configuration et les associations de dépendances et de composition entre elles. Les classes de configuration peuvent être enrichies de métadonnées utiles au processus de reconfiguration et/ou de vérification. Le résultat de cette modélisation est une première partie du diagramme de classes de la vue structurelle du modèle de référence, avec application des stéréotypes propres à la configuration : `Configuration`, `ConfigurationParameter`, `ConfigurationMetadata`, `ConfigurationReference` et `ConfigurationComposition`. Le modèle obtenu doit représenter la structure de l'information de configuration de l'environnement géré.

– **Définition du modèle d'état opérationnel**

Dans cette étape, on représente la vue logique des ressources et on identifie les paramètres d'état opérationnel dont l'évolution *at runtime* est susceptible d'influencer les valeurs de configuration. Le résultat de cette modélisation est la seconde partie du diagramme de classes de la vue structurelle du modèle de référence, avec application des stéréotypes relatifs à l'état opérationnel à savoir `ManagedElement` et `StateParameter`. Ces classes définissent les éléments qui vont servir à l'expression des contraintes *online*.

– **Expression des contraintes**

Cette étape consiste à exprimer, dans un langage de spécification choisi, les contraintes qui devront être vérifiées pour l'intégrité structurelle comme pour l'applicabilité opérationnelle des configurations candidates. Le résultat de cette étape doit fournir un ensemble de contraintes *offline* et *online* avec mention de leur niveau de sévérité et de leur état d'activité.

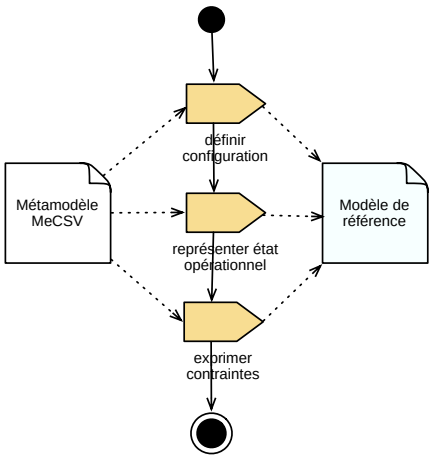

Figure 31 - Processus de définition de modèles de référence « de toute pièce »

Ce processus change légèrement lorsqu'il y a un modèle de gestion existant. La section suivante présente la démarche de définition de modèles de référence à partir de modèles de gestion existants.

V.1.2. **Processus de définition de modèles de référence MeCSV à partir d'un existant**

Le processus de définition présenté dans la partie précédente, permet de construire un modèle de référence de toute pièce et est plutôt adapté lorsqu'on part de rien. Dans le cas où un modèle d'information de gestion est disponible, il est important de pouvoir réutiliser une partie de cet existant dans la définition du modèle de référence MeCSV. Le processus correspondant est illustré dans la Figure 32 : ① la spécification de règles de transformation entre le modèle d'information de gestion et MeCSV, ② la génération de la vue structurelle du modèle de référence ③ l'expression des contraintes.

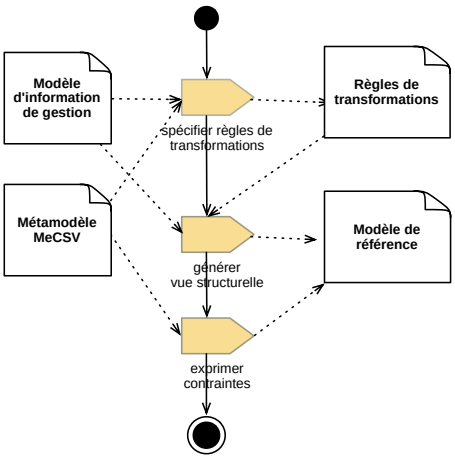

Figure 32 - Processus de définition d'un modèle de référence à partir d'un existant

V.1.2.1. **Spécification de transformation de modèles**

La prise en compte d'un modèle de gestion existant se fait par spécification de règles de transformation de modèles, c'est-à-dire des règles de correspondance ou de transformation entre les constructeurs du langage d'information de gestion existant et le métamodèle MeCSV.

– Transformation de modèles

Une transformation de modèles permet d'établir des règles de correspondance (*mappings*) et de traduction de concepts entre un modèle source et un modèle cible.

Définition 25 ***Transformation de modèles*** *: La transformation de modèles consiste en la génération d'un modèle cible à partir d'un modèle source ; les deux modèles représentent le même système. Ceci peut être fait de manière manuelle, semi-automatique, ou encore totalement automatique [OMG, 2003].*

Il existe trois approches de transformation de modèles, l'approche par programmation, l'approche par *template* et l'approche par modélisation [Diaw et al., 2010].

- L'approche par programmation consiste à programmer les transformations de modèles à l'aide de langages de programmation existants comme le langage Java. La transformation est un programme informatique. Les données d'entrée et de sortie représentent respectivement les modèles source et cible.

- L'approche par *template* consiste à définir des modèles *template* ou gabarits de modèles cibles paramétrés. L'exécution d'une transformation consiste à prendre un gabarit de modèles cibles paramétrés et à remplacer les divers paramètres par les valeurs du modèle source correspondant.

- l'approche par modélisation consiste à spécifier les règles de transformation en s'appuyant sur les concepts de l'IDM. L'objectif poursuivi est de modéliser les règles de transformation et de les rendre pérennes et productives en exprimant leur indépendance vis-à-vis des plates-formes d'exécution. L'OMG a défini un standard QVT (Query/View/Transform) [QVT, 2008] dédié à la spécification et la transformation de modèles.

Nous avons vu dans la première partie de ce manuscrit, qu'il y avait quatre grands types d'information de gestion dont les informations d'état opérationnel et les informations de configuration. Pour un modèle de gestion donné, des règles de transformation entre son langage de définition et le métamodèle MeCSV pourront permettre d'obtenir de façon automatique la vue structurelle du modèle de référence MeCSV correspondant.

Dans la lignée de l'approche IDM, nous préconisons la dernière approche de transformation à savoir la transformation par modélisation. Comme l'illustre la Figure 33, la définition de *mappings* au niveau métamodèle entre des constructeurs de langages d'information de gestion comme SMIng, CIM ou YANG et le métamodèle MeCSV permettront de réutiliser les modèles de gestion métier existants définis dans ces langages. Nous avons notamment spécifié des règles entre CIM et MeCSV qui sont présentées dans la section V.3.

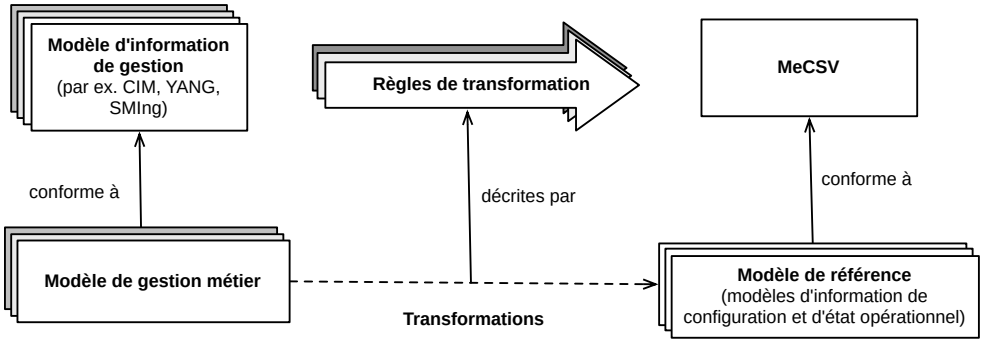

Figure 33 - Prise en compte de l'existant par transformation de modèles

V.1.2.2. Génération de la vue structurelle du modèle de référence

Cette étape correspond à l'exécution des règles de transformation de modèles définies précédemment sur le modèle de gestion existant. Le résultat de cette exécution fournit la partie structurelle du modèle de référence MeCSV à savoir un modèle de classes stéréotypées `Configuration` et `ManagedElement` avec leurs différents attributs et relations, conformément au métamodèle MeCSV.

V.1.2.3. Expression des contraintes

L' étape d'expression de contraintes demeure inchangée par rapport au processus de la section V.1.1. Il s'agit de spécifier des contraintes *offline* et *online* sur la vue structurelle obtenue.

Ces deux méthodes de définition de modèles de référence ont été expérimentées dans deux contextes applicatifs différents : la gestion d'une plate-forme de type MOM appelée `JORAM` et la gestion de machines virtuelles dans un environnement CIM/WBEM. Dans le premier cas, présenté dans la section V.2, le modèle de référence MeCSV de la plate-forme `JORAM` a été construit de toute pièce. Dans le second cas, section V.3, la spécification de règles de transformation entre CIM et le métamodèle MeCSV nous a permis de générer automatiquement la vue structurelle du modèle de référence MeCSV correspondant.

V.2. Définition d'un modèle de référence ex nihilo

Cette section présente la démarche de création de toute pièce d'un modèle de référence. Le cas d'étude MOM du chapitre précédent a été instancié sur un système réel nommé JORAM, une plate-forme MOM open source. L'objectif des points suivants est de détailler le processus de création du modèle de référence de JORAM, les entrées et sorties du processus et les outils utilisés. Le modèle de référence obtenu a été expérimenté dans plusieurs scénarii de vérification opérationnelle de configurations de serveurs de messages présentés dans le Chapitre 7.

V.2.1. Système géré cible

V.2.1.1. Présentation de la plate-forme JORAM

JORAM (Java Open Reliable Asynchronous Messaging) [JORAM] est un MOM open source utilisé dans des environnements opérationnels très variés comme le secteur bancaire, les secteurs de la santé, les secteurs administratifs. JORAM est une implémentation 100% Java de la spécification JMS 1.1. Il est exploité soit comme un MOM autonome entre des applications JMS développées pour des environnements variés (de JEE à JME), soit comme un MOM intégré dans un serveur d'applications JEE (en l'occurrence, JORAM est une brique du serveur d'applications JEE JONAS). Comme toute plate-forme JMS, JORAM est structuré en deux parties : une partie serveur qui gère les objets JMS (*files, topics* et connexions) et une partie cliente qui est en relation avec les applications.

Pour les besoins de gestion et de reconfiguration, nous nous concentrons sur la partie serveur de JORAM, c'est à dire la partie qui fournit les fonctionnalités supportant les échanges de messages entre applications.

V.2.1.2. Configuration et reconfiguration d'une plate-forme JORAM

L'architecture d'une plate-forme JORAM est constituée d'un ou plusieurs serveurs interconnectés proposant des services de connexions et des destinations pour l'échange de messages entre applications. Une plate-forme JORAM peut être mise en œuvre de façon centralisée ou de façon distribuée. Une configuration centralisée signifie que la plate-forme n'est constituée que d'un seul serveur de messages qui fournit tous les services. Dans une configuration distribuée, la plate-forme est constituée de deux ou plusieurs serveurs de messages. Les serveurs de messages sont regroupés dans un ou plusieurs domaines réseaux.
Les serveurs JORAM supportent différents protocoles de communication comme TCP/IP, HTTP, SOAP.

La configuration d'une plate-forme JORAM se fait en deux temps : JORAM sépare la configuration de l'architecture de la plate-forme de la configuration des objets JMS (fabriques de connexion, *files*, *topics*, droits d'accès utilisateurs), également appelés «objets administrés».

- Dans un premier temps, il faut configurer l'architecture de la plate-forme, c'est-à-dire décrire les serveurs, leur nombre, leur localisation (machines hôtes) et les services qu'ils fournissent (gestionnaires de connexions, services protocolaires, annuaires de nom). Cette première configuration doit être décrite dans un fichier de configuration XML appelé `a3servers.xml` selon un schéma DTD fourni. Le déploiement de ce fichier démarre les serveurs de message de la plate-forme et les services proposés selon la configuration spécifiée. La plate-forme est prête à recevoir la configuration des objets administrés.

- Dans un second temps, il faut définir les objets JMS pour chaque serveur. Cette configuration se fait dans un second fichier de configuration appelé `joramAdmin.xml` par rapport à un schéma DTD qui est également fourni. L'exécution de ce fichier crée les objets JMS conformément à la configuration spécifiée.

La Figure 34 montre un exemple complet de configuration `JORAM` :

- entre les balises `<config></config>`, est définie la configuration d'un serveur de messages `S0`. Le serveur `S0` s'exécute sur la machine `localhost` et propose les services suivants : un gestionnaire de connexion (`ConnectionManager`), un service TCP (`TCPProxyService`), et un service d'annuaire de noms JNDI (Java Naming and Directory Interface) `JndiServer`,
- entre les balises `<JoramAdmin></JoramAdmin>`, sont spécifiées les configurations d'objets administrés : le server `S0` fournit une fabrique de connexion (`ConnectionFactory`), deux files de messages `myQueue` et `dmQueue` et permet un accès utilisateur anonyme (`anonymous`).

```
<?xml version="1.0"?>
<config name="Simple_Config">
 <server id="0" name="S0" hostname="localhost">
  <service class="org.[…].ConnectionManager" args="root root"/>
  <service class="org.[…].TcpProxyService" args="16010"/>
  <service class="fr.[…].JndiServer" args="16400"/>
 </server>
</config>
<JoramAdmin>
  <InitialContext> […] </InitialContext>
  <ConnectionFactory> […] </ConnectionFactory>
  <Queue name="myQueue" serverId= "0"
         nbMaxMsg="200" dmq="dmqueue">
    <freeReader/> <freeWriter/>
    <jndi name="myQueue"/>
  </Queue>
  <User name="anonymous" password="passwd" serverId="0"/>
  <DMQueue name="dmqueue" serverId = "0">
     <freeReader /><freeWriter />
  </DMQueue>
</JoramAdmin>
```

Figure 34 - Exemple de configuration d'un serveur de messages JORAM

La reconfiguration d'une plate-forme déployée se fait soit par modification de ces deux fichiers de configuration, soit par des appels d'opérations via une interface d'administration qui vont modifier les fichiers de configuration correspondants. Le passage d'une architecture centralisée à une architecture distribuée (et vice-versa), la création de nouvelles files de messages ou la migration de files de messages existantes entre serveurs, l'ajustement des paramètres de configuration tels le nombre maximum de connexions clients, le nombre maximum de messages en mémoire, les droits d'accès utilisateurs sont des exemples de reconfiguration.

V.2.2. **Création du modèle de référence correspondant**

Cette section décrit les trois étapes de création du modèle de référence de `JORAM`, d'abord la définition du modèle de configuration de la plate-forme, puis la représentation du modèle d'état opérationnel et enfin l'expression des contraintes. Le résultat de la modélisation de ces trois étapes a produit le modèle de référence MeCSV de la plate-forme `JORAM`.

V.2.2.1. Définition du modèle de configuration

Cette étape a consisté à modéliser la structure de la configuration d'une plate-forme JORAM, c'est-à-dire à traduire les différents concepts des grammaires DTD de configuration en classes, attributs et associations stéréotypées MeCSV. Nous avons étudié ces grammaires, puis établi les correspondances avec les stéréotypes MeCSV. Le résultat de ce travail est un modèle de configuration conforme à MeCSV dont un extrait est illustré dans le diagramme de classes UML de la Figure 35. Cet extrait de diagramme de classes permet de représenter l'exemple de configuration de la figure précédente (Figure 35).

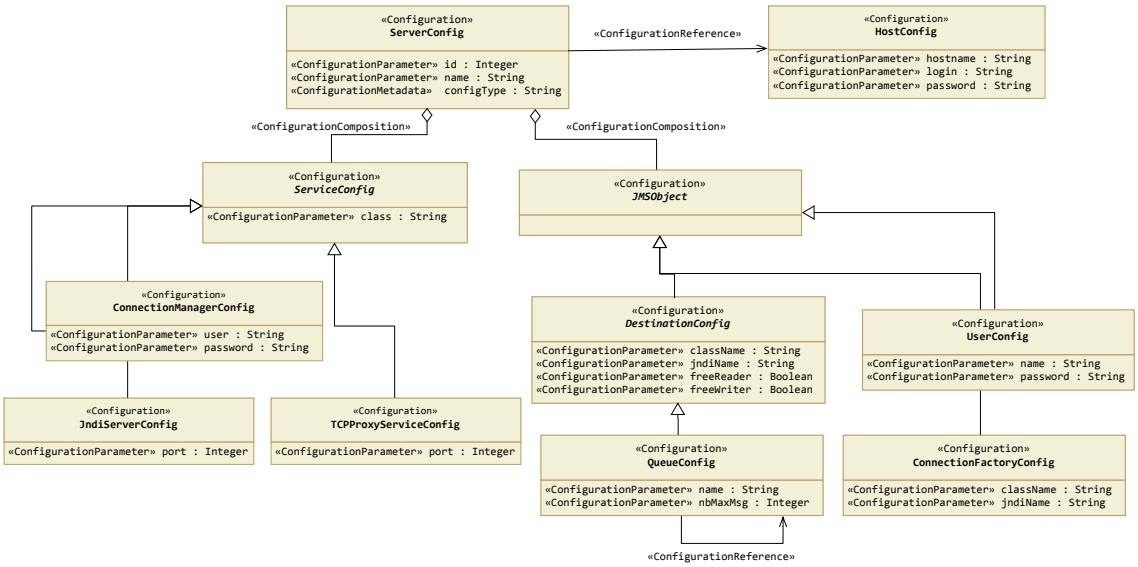

Figure 35 - Extrait du modèle de configuration résultant

V.2.2.2. Représentation des informations d'état opérationnel

Cette étape a consisté à identifier les éléments gérés et les informations opérationnelles importantes. Ces informations d'état opérationnel sont tributaires des contraintes qu'on veut exprimer.

Il n'existait pas de lignes directrices concernant les paramètres d'état sensibles à surveiller. Nous avons étudié diverses problématiques de gestion des intergiciels orientés messages, via les documentations existantes, les retours d'expériences et les forums de discussion.

Les intergiciels que nous avons étudiés sont `ActiveMQ` [ActiveMQ], un MOM open source de la fondation APACHE, il est très populaire et utilisé dans plusieurs serveurs d'applications existants ; `OpenMQ` [OpenMQ], un MOM open source inclus dans le serveur d'applications `GlassFish` de Sun, et `JORAM` lui-même.

Nous nous sommes intéressés aux objectifs métiers mis en œuvre. Ce sont pour l'essentiel, des objectifs de gestion de la performance et de la stabilisation opérationnelle des plates-formes tels que l'optimisation du débit de messages (nombre de messages traités par seconde), la réduction du délai de latence (temps entre la production et la consommation d'un message) ou la gestion des surcharges et des pannes.

Ces études nous ont permis d'identifier des paramètres d'état opérationnel clés, susceptibles de contribuer ou non au succès d'une reconfiguration. Des exemples de ces paramètres sont le statut opérationnel des serveurs et des services, le nombre de connexions actives, le nombre de messages stockés dans les destinations, le pic ou la moyenne d'utilisation des ressources.

La modélisation de ces informations est illustrée dans l'extrait de diagramme UML de la Figure 36.

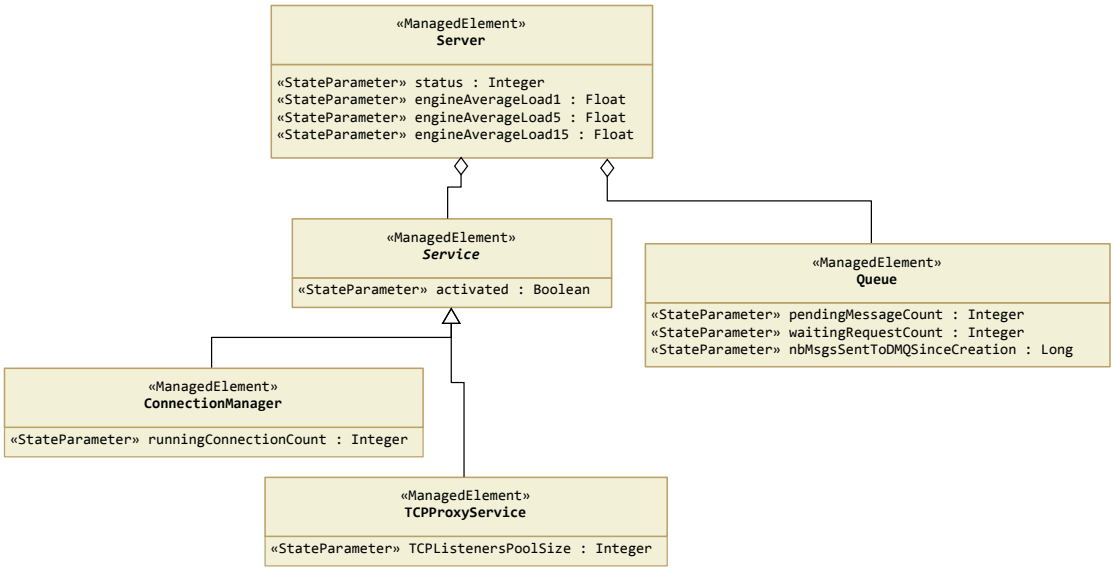

Figure 36 - Extrait du modèle d'état opérationnel résultant

V.2.2.3. Expression des contraintes

L'identification des contraintes *offline* a été facilitée par l'étude de la documentation technique de `JORAM` ainsi que par l'étude des différentes exceptions prévues dans les interfaces d'administration que la plate-forme propose. Par exemple, `JORAM` définit un mécanisme de lancement d'exceptions et d'erreurs. Les erreurs de gestion génèrent des exceptions `AdminException`, dont héritent des exceptions plus spécifiques comme `ServerIdAlreadyUsedException` qui est levée lorsqu'il y a doublon dans les identifiants des serveurs (les identifiants des serveurs doivent être uniques pour que le système fonctionne correctement), ou bien les erreurs `JndiError` qui permettent de lancer des exceptions telles que `NamingException` lorsque le nom d'une file de messages n'est pas enregistré dans l'annuaire de noms.

Certaines contraintes *online* ont été également identifiées dans la documentation, par exemple les exceptions levées lorsqu'on veut reconfigurer un serveur ou un service qui n'est pas opérationnel. D'autres dépendent du niveau de service désiré et des objectifs métiers supportés en matière de performance ou de sécurité, par exemple le débit de messages, le temps de latence et la charge moyenne. Ces contraintes introduisent une confrontation de paramètres de configuration à des paramètres opérationnels, par exemple le nombre maximum des messages stockés en mémoire d'une file de messages versus le nombre courant de messages stockés en mémoire ou encore le nombre maximum de connexions supportées versus le nombre courant de connexions actives.

Nous avons choisi le langage OCL pour l'expression des contraintes. Les contraintes *offline* et *online* que nous avons définies encadrent les exigences suivantes :

- respect de la structure de la configuration : toute configuration respecte l'architecture de la plate-forme ainsi que les relations de dépendances entre paramètres de configuration (E1). E1 se traduit par un ensemble de contraintes *offline* par exemple, les serveurs de messages doivent posséder un identifiant unique ou encore un serveur de messages doit

fournir au moins une destination. Ces contraintes garantissent que les configurations candidates de serveurs préservent le fonctionnement de la plate-forme.

- présence d'un service de nommage : les fabriques de connexions et les destinations doivent être accessibles via un service de nom (E2). E2 se traduit à la fois par des contraintes *offline* et *online*. Les contraintes *offline* garantissent que la plate-forme fournit un service de nommage (`JndiServer`) et que les fabriques de connexion et les destinations possèdent un nom pour leur enregistrement (attribut `jndiName`). Les contraintes *online* assurent que le service de nommage est opérationnel.

- optimisation de la mémoire : la mémoire d'une file de messages ne doit pas être dans un état dégradé, c'est-à-dire qu'une file de messages ne doit pas être remplie au-delà de 80% de sa capacité (E3). E3 se traduit par une contrainte *online* qui confronte la mémoire courante d'une file de messages (attribut `pendingMessageCount`) à sa mémoire maximale (attribut `nbMaxMsg`).

Ces contraintes ont été exprimées dans un fichier OCL dont un extrait est illustré dans la Figure 37.

```
context ServerConfig
inv  UniqueServerIds_offline_active_fatal: ServerConfig.allInstances()->forAll(s1, s2  | s1 <> s2 implies  s1.id <>  s2.id)

context ServerConfig
inv  MustBePositiveServerId_offline_active_fatal: self.id > 0

context ServerConfig
inv  UniqueServerName_offline_inactive_fatal: ServerConfig.allInstances()->forAll(s1, s2  | s1 <> s2 implies  s1.name<>  s2.name)

context ServerConfig
inv MustExistJndiServer_offline_active_fatal: ServerConfig.allInstances()->exists(s  | s.configSub->exists(c | c.oclIsTypeOf(JndiServerConfig))

context ServerConfig
inv MustHaveQueue_offline_active_error: self.configSub->exists( q | q.oclIsTypeOf(QueueConfig))                    E1

context QueueConfig
inv ForbiddenDefaultNbMaxMsg_offline_active_error: self.oclIstTypeOf(DeadMQueueConfig) = false implies self.nbMaxMsg <> -1

context QueueConfig
inv MustHaveMemoryLimitBehavior_offline_active_warning: self.nbMaxMsg <> -1 implies (self.referenced->exists(q2 |
q2.oclIsTypeOf(DeadMQueueConfig) = true )


context ServerConfig
inv ShouldBeActivatedJndiServer_online_active_error: ServerConfig.allInstances()->exists(s  |  s.configSub->exists(c |
c.oclIsTypeOf(JndiServerConfig) and c.oclAsType(JndiServerConfig).managedElement->first().oclAsType(JndiServer).activated = true)

context ServerConfig
inv MustHaveAccessibleQueue_offline_active_error: self.configSub->select(q1 | q1.oclIsTypeOf(QueueConfig) = true)->forAll( q2 |
q2.oclAsType(QueueConfig).jndiName <> '')
                                                                                                                    E2
context QueueConfig
inv MustBeAccessible_offline_active_error: self.jndiName <> ''


context QueueConfig                                                                                                 E3
inv OperationalApplicableNbMaxMsg_online_active_error :
(self.managedElement->notEmpty() = true) implies self.nbMaxMsg*0.8 > self.managedElement->first().oclAsType(Queue).pendingMessageCount
```

Figure 37 - Extrait des contraintes *offline* et *online* en OCL

V.3. Définition de modèle de référence à partir d'un existant

Cette section décrit la mise en œuvre de règles de transformation de modèles QVT entre le standard CIM et le métamodèle MeCSV. L'application de ces règles sur le modèle CIM d'un système virtuel nous a permis de générer automatiquement la vue structurelle du modèle de référence correspondant. La vue structurelle générée a été ensuite complétée par des contraintes *offline* et *online*. Le modèle de référence ainsi obtenu, a été expérimenté dans plusieurs scénarii de vérifications opérationnelles de configurations candidates de machines virtuelles qui sont présentés dans le Chapitre 7.

V.3.1. Système géré cible

V.3.1.1. Le patron de configuration CIM_SettingData

Comme nous l'avons précédemment vu au Chapitre 2, CIM est le modèle informationnel de base de l'architecture WBEM. CIM et WBEM sont des standards de gestion issus du DMTF. Ils permettent d'intégrer des environnements de gestion hétérogènes. De plus, CIM fournit des schémas standardisés pour différents types d'environnements et de domaines de gestion. En l'occurrence, CIM définit un schéma standard appelé «*CIM Core Schema*» qui propose un patron de configuration pour décrire la configuration (classe `CIM_SettingData`) d'un élément géré donné (classe `CIM_ManagedElement`). Ce patron est présenté dans la Figure 38.

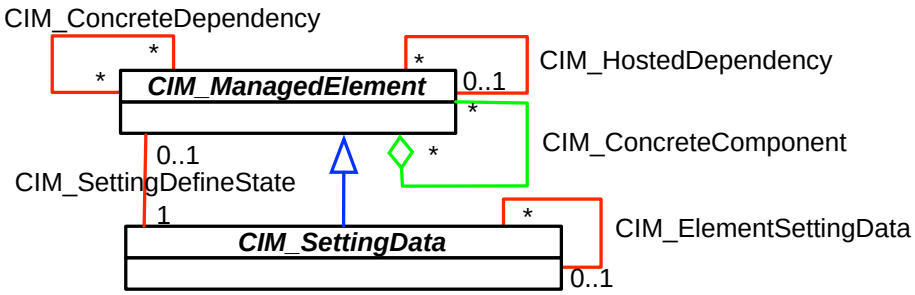

Figure 38 - *Pattern* de classes CIM (version 2.3) pour la transformation de modèles

- La classe `CIM_ManagedElement` est la classe mère de toutes les classes CIM. Elle est abstraite. Elle permet de représenter la vue logique d'une entité gérée, notamment sa vue opérationnelle. Habituellement, la classe `CIM_ManagedElement` mélange les paramètres d'état opérationnel avec les paramètres de configuration.

- La classe `CIM_SettingData` supporte plusieurs usages dont la définition de configurations d'éléments gérés. Elle est abstraite. Les instances de type `CIM_SettingData` peuvent représenter plusieurs types de configuration pour un élément géré, par exemple des configurations persistantes, des configurations en cours d'application ou des configurations par défaut.

La spécification de ce patron précise qu'une instance de type `CIM_SettingData` qui définit la configuration d'un élément géré ne doit pas contenir de valeurs d'état opérationnel. Les valeurs courantes d'état opérationnel correspondantes doivent être reflétées par des attributs définis dans la vue logique de l'élément géré lui-même (instances de classes héritant de `CIM_ManagedElement`). La classe `CIM_SettingData` poursuit donc sémantiquement le même objectif que la classe `Configuration` de MeCSV. De même, toute classe qui hérite de `CIM_ManagedElement` et dont la configuration est externalisée dans une classe fille de `CIM_SettingData` est équivalente à une classe `ManagedElement` de MeCSV.

V.3.1.2. Présentation du standard QVT

L'approche par modélisation de la transformation de modèles est une technique fondamentale dans l'IDM pour rendre les modèles opérationnels. QVT [OMG-QVT, 2008] est un standard défini par l'OMG, dans ce but, pour modéliser des transformations de modèles. La spécification QVT définit un ensemble de formalismes pour exprimer des règles de transformation de modèles, structuré en deux parties : une partie déclarative supportée par les langages standards `Relations`, `Core` et une partie impérative proposant un langage standard `Operational Mappings` et un mécanisme non-standard appelé *Black Box*.

- `Relations` est un langage déclaratif de haut niveau, convivial qui permet d'exprimer des correspondances complexes entre différents éléments de (méta-) modèles. `Relations` supporte la génération implicite de traces d'exécution des transformations. Il possède une syntaxe textuelle et graphique.
- `Core` est un langage déclaratif aussi puissant que `Relations` qui permet d'exprimer des correspondances simples entre différents éléments de (méta-) modèles. Cependant c'est un langage technique de bas niveau, possédant une syntaxe textuelle.
- Le langage `Operational Mappings` étend les langages `Relations` et `Core` avec des constructions impératives et des extensions OCL à effet de bord pour une spécification de transformation plutôt procédurale.
- QVT propose également un mécanisme dit *Black Box* qui permet d'appeler des opérations de transformations spécifiées dans des programmes externes.

Nous avons utilisé le langage `Operational Mappings` du standard `QVT`, pour définir des règles de transformations entre CIM et MeCSV. Plus précisément, nous avons exploité l'implémentation `Eclipse QVT Operational` (`QVTo`) [Eclipse-QVTo] du langage `Operational Mappings`. `QVTo` fournit une implémentation du langage `Operational Mappings` pour la description de règles de transformation et un moteur de transformation pour l'exécution de ces règles. `Eclipse QVTo` est un *plugin* Eclipse, il peut être également utilisé en mode *standalone*.

V.3.1.3. Mise en œuvre de la transformation

La Figure 39 illustre la mise en œuvre de la transformation de modèles entre CIM et MeCSV. Le DMTF fournit à ses membres industriels et académiques, une version du modèle informationnel CIM et de ses schémas standardisés en UML. Nous avons utilisé la version UML du schéma «*CIM Core Schema*». Nous avons décrit des règles de transformation entre les classes du patron de configuration de CIM et celles du profil UML de MeCSV.

L'exécution de ces règles de transformation sur un modèle CIM source génère son pendant en modèle de référence (partie structurelle). Cette partie du modèle de référence est complétée dans un deuxième temps par les contraintes qui doivent être respectées.

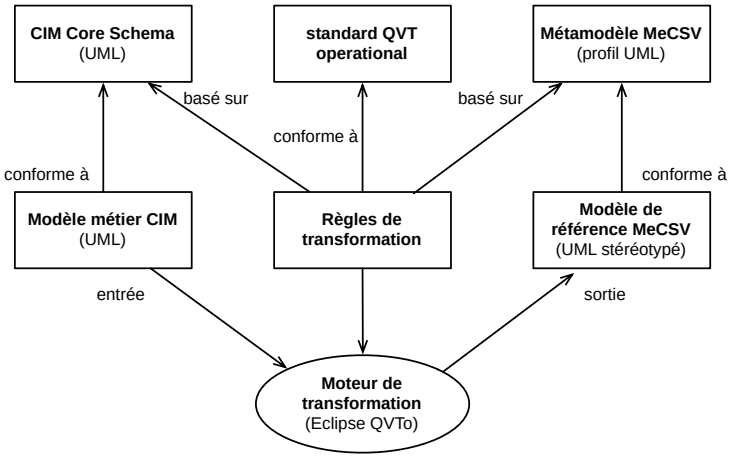

Figure 39 – Transformation de modèles CIM vers MeCSV

V.3.2. Création du modèle de référence correspondant

V.3.2.1. Spécification des règles de transformation de modèles

Comme le montre l'extrait de la Figure 40, la mise en oeuvre des règles de transformation de modèles CIM a consisté à spécifier des traductions entre les éléments du patron de configuration CIM et leurs équivalents dans MeCSV et à leur appliquer les stéréotypes du profil UML de MeCSV.

```
transformation CIM2MeCSVProfile(in cimModel :uml, in MeCSVProfile : uml, out referenceModel:uml);
main() {
    -- Preparation du modele destination
    var _result:= cimModel.rootObjects()[Model]->deepclone();
    _result[Model]->forEach(model) {

        -- Application du profil MeCSVProfil
        var profile : Profile := MeCSVProfile.objectsOfType(Profile)![name="MeCSVProfile"];
        model.applyProfile(profile);
```

Figure 40 - Transformation de CIM vers le profil UML de MeCSV

- **Transformation de la classe `CIM_SettingData`**

Nous avons défini des règles qui transforment toute classe héritant de `CIM_SettingData` d'un modèle source CIM en classe stéréotypée «`Configuration`» dans le modèle destination généré. Les attributs de ladite classe sont automatiquement transformés en attributs de classe «`ConfigurationParameter`». Un extrait des règles est présenté dans la Figure 41.

```
-- Traduction des elements du modele source
model.packagedElement->forEach(member){
    -- traduction des classes
    if(member.oclIsTypeOf(Class) and (member.oclIsTypeOf(AssociationClass)=false)) then {
        log("treating Class" + member.name);
        var clazz : Class := member.oclAsType(Class);

        -- Vérifier si la classe source est candidate à la transformation en <Configuration>
        -- Si oui appliquer le stereotype <Configuration>
        if(clazz.isEligibleToConfig()) then {
            var stereoConfig : Stereotype = MeCSVProfile.objectsOfType(Stereotype)![name = "Configuration"];
            clazz.applyStereotype(stereoConfig);

        -- Transformer ses paramètres en <ConfigurationParameter>
            clazz.attribute->forEach(attr){
                log("-- treating attributes" +attr.name);

                var stereoConfigParam : Stereotype = MeCSVProfile.objectsOfType(Stereotype)![name = "ConfigurationParameter"];
                attr.applyStereotype(stereoConfigParam);
            }

        }endif;
```

Figure 41 - Règle de transformation des classes de configuration

- **Transformation de la classe `CIM_ManagedElement`**

Nous avons spécifié des règles de correspondance qui transforment toute classe héritant de `CIM_ManagedElement` et n'héritant pas de `CIM_SettingData` d'un modèle source CIM en classe stéréotypée «`ManagedElement`» dans le modèle destination correspondant. Les attributs d'une telle classe sont transformés en attributs de classe annotés «`StateParameter`» (Figure 42).

```
-- Vérifier si la classe source est candidate à la transformation en <ManagedElement>
-- Si oui appliquer le stereotype <ManagedElement
if(clazz.isEligibleToState()) then {
    var stereoState : Stereotype = MeCSVProfile.objectsOfType(Stereotype)![name = "ManagedElement"];
    clazz.applyStereotype(stereoState);

    -- Transformer ses paramètres en <StateParameter>
    clazz.attribute->forEach(attr){
        log("-- treating attributes" +attr.name);

        var stereoStateParam : Stereotype = MeCSVProfile.objectsOfType(Stereotype)![name = "StateParameter"];
        attr.applyStereotype(stereoStateParam);
    }

}endif;
```

Figure 42 – Règle de transformation des classes d'état opérationnel

– Transformation des associations

Les règles de transformation traduisent toute association éligible du modèle source, c'est-à-dire toute association qui représente une dépendance entre configurations, une composition de configurations ou encore un lien entre configuration et élément géré , en association stéréotypée avec les constructeurs correspondants de MeCSV. Un extrait de ces règles est présenté dans la Figure 43.

```
-- Traduction des associations
if(member.oclIsKindOf(Association)) then {
    var asso : Association:=member.oclAsType(Association);
    -- Application de stereotypes adequats
    if(asso.isEligibleConfigurationRelationship()) then {
    -- Cas <ConfigurationReference>
        if(asso.isEligibleConfigurationReference()) then
        {
            var stereoAssoRef : Stereotype = MeCSVProfile.objectsOfType(Stereotype)![name = "ConfigurationReference"];
            asso.applyStereotype(stereoAssoRef);
            }else{
            -- Cas <ConfigurationComposition>
            var stereoAssoComp : Stereotype = MeCSVProfile.objectsOfType(Stereotype)![name = "ConfigurationComposition"];
            asso.applyStereotype(stereoAssoComp);
        }endif;
    }else {
    -- Cas <ElementConfiguration>
        if(asso.isEligibleElementConfiguration()) then {
            var stereoAssoMe : Stereotype = MeCSVProfile.objectsOfType(Stereotype)![name = "ElementConfiguration"];
            asso.applyStereotype(stereoAssoMe);
        }endif;
    }endif;
```

Figure 43 - Règle de transformation des associations

L'exécution de ces règles de transformation sur un modèle CIM qui vérifie le patron de configuration `CIM_SettingData` fournit la partie structurelle d'un modèle de référence. Cette étape est présentée dans la section suivante.

V.3.2.2. Génération de la vue structurelle du modèle de référence

Dans cette étape, les règles de transformation définies précédemment sont exécutées par le moteur de transformation d'`Eclipse QVTo` sur un modèle métier CIM source. En l'occurrence, nous avons testé les règles de transformation sur un modèle CIM tiré du profil de gestion «Virtual System» du DMTF [DMTF-VirtualSystemProfile, 2010] et avons obtenu la vue structurelle du modèle de référence lui correspondant.

– Présentation du profil de gestion de systèmes virtuels du DMTF

Définition 26 ***Profil de gestion du DMTF*** *: un profil de gestion est un ensemble de documents, défini par le DMTF ou ses partenaires, qui spécifie pour un domaine de gestion particulier, un modèle CIM métier et un guide pour son utilisation.*

La définition de profils de gestion est une initiative du DMTF dont le but est de fournir des descriptions unifiées de différents domaines de gestion en CIM, afin d'améliorer l'interopérabilité entre systèmes de gestion hétérogènes. On trouve par exemple des profils de gestion dédiés à la gestion de services réseaux (DNS, DHCP,) de systèmes d'exploitation ou encore d'éléments matériels (CPU, batteries, port Ethernet).

Le profil de gestion *Virtual System* est dédié à la gestion des systèmes virtuels. Il définit un modèle CIM minimal pour l'inspection et le contrôle de fonctionnalités de virtualisation comme les assignations de machines virtuelles sur des machines hôtes et les allocations de ressources. Ce profil identifie les éléments nécessaires pour la représentation et la gestion de machines virtuelles et de ressources virtuelles telles que la mémoire, l'espace disque ou encore la capacité du processeur (CPU).
La Figure 44 montre le modèle CIM issu de ce profil que nous avons utilisé comme modèle source pour l'exécution de la transformation.

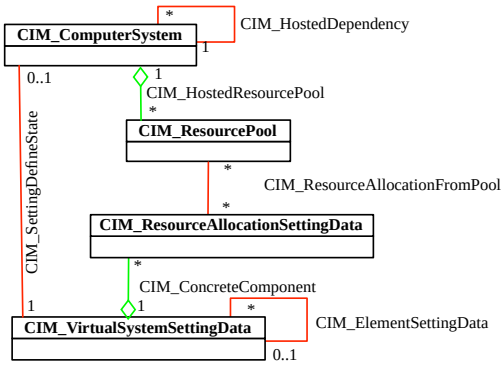

Figure 44 – Modèle CIM issu du profil Virtual System

- La classe `CIM_ComputerSystem` : cette classe permet de représenter la vue opérationnelle des systèmes virtuels. Un exemple de paramètre est son statut opérationnel (`OperationalStatus`) ou encore son état d'activité (`EnabledState`) qui renseigne la capacité de l'élément à être sollicité.

- La classe `CIM_ResourcePool` : cette classe est une entité logique qui permet de modéliser le pool de ressources, offertes par une machine-hôte, à allouer à ses machines virtuelles. Des exemples de paramètres d'état opérationnel sont la réservation courante (attributs `Reserved` et `CurrentlyConsumedResource`) c'est-à-dire la somme de toutes les allocations pour un type de ressource donné (mémoire, espace disque, CPU).

- La classe `CIM_ResourceAllocationSettingData` : cette classe permet de définir la configuration, c'est-à-dire l'ensemble des paramètres de configuration qui caractérisent une ressource allouée. Le type de ressource (attribut `ResourceType`) est un exemple de paramètre de configuration, le nombre de Giga-octets est un exemple de paramètre de configuration pour une ressource de type espace disque (attribut `Reservation`).

- La classe `CIM_VirtualSystemSettingData` : cette classe permet de modéliser la configuration de systèmes virtuels. Il existe une relation de composition entre la classe `CIM_VirtualSettingSystemData` et la classe `CIM_ResourceAllocationSettingData`. Cela traduit le fait que la configuration d'un système virtuel doit comprendre les configurations des différentes ressources qui lui sont allouées.

– **Concepts généraux d'allocation de ressources virtuelles**

De façon générale, une machine hôte (`CIM_ComputerSystem`) expose un pool de ressources (`CIM_ResourcePool`) à allouer à ses machines virtuelles (`CIM_ComputerSystem`). Les objectifs de gestion de ces machines virtuelles consistent à définir différents schémas d'allocation de ressources (`CIM_ResourceAllocationSettingData`) pour les machines virtuelles hébergées (`CIM_VirtualSystemSettingData`).

La reconfiguration des allocations de ressources pour des machines virtuelles est un exemple de reconfiguration dynamique caractérisée par une dynamicité forte. Il faut non seulement garantir la bonne définition de la configuration de nouvelles machines virtuelles ou de machines virtuelles existantes, mais il faut aussi s'assurer que les reconfigurations des allocations de ressources restent cohérentes par rapport aux ressources réelles utilisées. Une reconfiguration qui échoue à allouer proprement les ressources disponibles peut provoquer un fonctionnement dégradé voire une impossibilité de mise en route d'une machine virtuelle.

– **Exécution de la transformation**

En suivant les recommandations du profil de gestion *Virtual System*, nous avons extrait dans un premier temps, le diagramme de classes UML correspondant au modèle CIM de la Figure 44. Le diagramme de classes UML obtenu a servi d'entrée au moteur d'exécution de transformations de l'outil `Eclipse QVTo`. La vue structurelle du modèle de référence équivalent avec les stéréotypes adéquats (Figure 45) a été générée avec succès. Cette possibilité nous a permis de valider les règles de transformation.

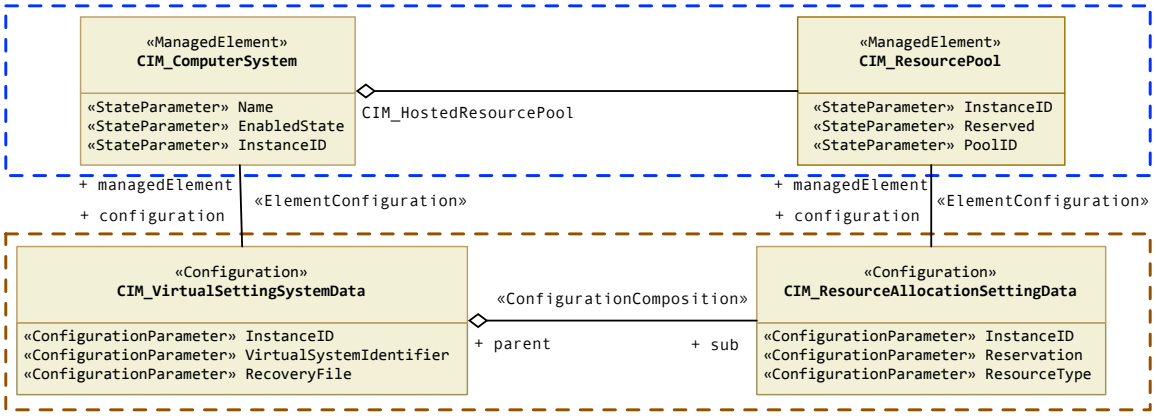

Figure 45 - Extrait de la vue structurelle conforme à MeCSV obtenue

V.3.2.3. Expression des contraintes

Après avoir identifié les classes candidates pour la transformation de modèles et généré la vue structurelle du modèle de référence, nous avons exprimé les contraintes *offline* et *online* nécessaires à la validité opérationnelle des configurations candidates.

Le profil de gestion signale des contraintes structurelles équivalant aux contraintes *offline*. Par exemple :

- la configuration d'une machine virtuelle doit préciser un identifiant unique pour chaque machine virtuelle (C1). Cette contrainte permet de garantir qu'une machine virtuelle possède un identifiant unique. Cette contrainte est primordiale notamment dans le cas de clusters de machines.

- la configuration d'une machine virtuelle doit mentionner des allocations positives pour les types de ressources suivants : mémoire, espace disque et CPU (C2). Cette contrainte permet de garantir qu'une machine virtuelle est utilisable.

Nombre de préconisations signalées dans la documentation correspondent à notre concept de contraintes *online*. Par exemple :

- l'allocation d'une ressource ne doit pas être supérieure à la capacité disponible (C3). Cette contrainte confronte le paramètre de configuration `Reservation` au paramètre d'état `Reserved`. La valeur de ce paramètre n'est disponible qu'en phase d'exécution (*at runtime*) et peut varier selon les conditions opérationnelles. Cette contrainte online permet de garantir une allocation consistante avec l'état courant de la ressource.

- la somme totale des allocations pour un type de ressource donné ne doit pas dépasser 90% de la capacité totale du pool de ressources (C4). Cette contrainte est une variante de la précédente, elle permet de prévenir les situations de fonctionnement dégradé.

Ces contraintes ont été exprimées dans un fichier OCL dont l'extrait est présenté dans la Figure 46.

```
context CIM_VirtualSystemSettingData
inv uniqueVirtualSystemsIds_offline_active_fatal: CIM_VirtualSystemSettingData.allInstances()->forAll(v1, v2 | v1<>v2 implies
v1.VirtualSystemIdentifier <> v2.VirtualSystemIdentifier)                                                                    C1

context CIM_VirtualSystemSettingData
inv PositiveReservationVSSD_offline_active_error: self.sub->select( r | r.oclIsTypeOf(CIM_ResourceAllocationSettingData)=true)->forAll( r2 |
r2.oclAsType(CIM_ResourceAllocationSettingData) (CIM_ResourceAllocationSettingData))


context CIM_VirtualSystemSettingData
inv MandatoryMemoryAllocation_offline_active_error:
self.sub->select( r | r.oclIsTypeOf(CIM_ResourceAllocationSettingData)=true)->exists(r2 |
r2.oclAsType(CIM_ResourceAllocationSettingData).ResourceType = 4)

context CIM_VirtualSystemSettingData                                                                                         C2
inv MandatoryDiskAllocation_offline_active_error:
self.sub->select( r | r.oclIsTypeOf(CIM_ResourceAllocationSettingData)=true)->exists(r2 |
r2.oclAsType(CIM_ResourceAllocationSettingData).ResourceType = 19)


context CIM_ResourceAllocationSettingData                                                                                    C3
inv OnlineValidRervationRASD_online_active_error: self.Reservation < self.managedElement->first().capacity - self.managedElement-
>first().Reserved


context CIM_VirtualSystemSettingData
def memRASD : Set(CIM_ResourceAllocationSettingData) =  self.sub->select(r | r.oclIsTypeOf(CIM_ResourceAllocationSettingData) = true and
r.oclAsType(CIM_ResourceAllocationSettingData).ResourceType = 4)                                                             C4
def memPool : Set(CIM_ResourcePool) = memRASD.managedElement->select( p | p.oclIsTypeOf(CIM_ResourcePool)=true)
inv OnlineTotalResourcePoolReservation_online_active_warning:
memRASD->collect(Reservation)->sum() <= memPool->collect(Capacity)->sum()  * 0.9
```

Figure 46 - Extrait des contraintes *offline* et *online* en OCL

V.4. Conclusion

Dans ce chapitre nous avons présenté le processus de création outillée de modèles de référence pour un environnement de gestion donné. Deux méthodes sont offertes pour la définition des modèles de référence MeCSV :

- la première consiste à définir un modèle de référence MeCSV de toute pièce. Elle est plus adéquate lorsqu'il n'existe pas de modèles de gestion de l'environnement considéré. Dans un éditeur de modèles où le profil UML de MeCSV est installé, elle consiste à modéliser un diagramme de classes représentant les informations de configuration et d'état opérationnel d'une part et à exprimer dans un fichier OCL, les contraintes nécessaires d'autre part et à appliquer les stéréotypes du profil UML de MeCSV. Ce modèle de référence est construit une fois pour toutes, il demeure toutefois éditable en phase d'exécution, notamment pour rajouter ou modifier des contraintes.

- la seconde permet de prendre en compte les modèles de gestion existants. Cette méthode consiste à spécifier des règles de transformation entre un modèle d'information existant et MeCSV. Ces règles de transformation exécutées sur un modèle métier source permettent de générer automatiquement la vue structurelle du modèle de référence lui correspondant (diagrammes de classes des informations de configuration et d'état opérationnel). Cette vue est ensuite complétée avec les contraintes nécessaires.

Ces deux méthodes ont été expérimentées dans la définition de modèles de référence pour deux systèmes réels.

- Nous avons démontré la capacité des constructeurs de MeCSV à couvrir la définition d'un modèle de référence pour une plate-forme MOM appelée `JORAM`. Nous avons construit de toute pièce un modèle de référence complet.

- Nous avons également démontré la faisabilité d'une prise en compte de modèles de gestion existants en définissant des transformations de modèles entre CIM et MeCSV. Ces règles ont été exécutées sur un modèle métier CIM extrait du profil de gestion *Virtual System* du DMTF. Cette transformation a produit en sortie une vue structurelle du modèle de référence correspondant que nous avons ensuite complétée par des exemples de contraintes *offline* et *online*.

Favoriser l'intégration de modèles CIM en phase de conception, c'est-à-dire pouvoir générer automatiquement des modèles de référence pour modèles CIM est une contribution importante : cela permet de proposer d'emblée notre service de vérification opérationnelle de configurations à un grand nombre d'environnements de gestion.

Selon nous, les modèles de référence sont suffisants pour porter la vérification de configurations candidates *at runtime*. Afin d'exécuter la vérification, le système de gestion d'un système existant doit être capable d'envoyer des instances conformes à la partie structurelle du modèle de référence défini. C'est-à-dire que le système de gestion doit envoyer les instances de configuration à vérifier et les valeurs courantes des paramètres d'état opérationnel nécessaires à l'évaluation de contraintes *online*. Pour vérifier cette hypothèse, nous avons conceptualisé et implémenté une architecture de vérification opérationnelle de configurations qui est présentée dans le chapitre suivant.

Chapitre 6

Un service de vérification opérationnelle

Dans le chapitre précédent, nous avons présenté le métamodèle MeCSV qui permet, en phase de conception, une spécification unifiée de modèles de référence qui vont servir à vérifier les configurations candidates en phase d'exécution. Pour ce faire, nous avons architecturé un service de vérification opérationnelle qui peut être invoqué par tout système de gestion proposant des fonctionnalités de supervision et de reconfiguration. Ce service est implanté et organisé autour de deux composants internes : un moteur d'exécution de la vérification et une base de modèles où sont stockés les modèles de référence. Le service de vérification fournit des interfaces bien définies d'invocation de la vérification et d'édition des modèles de références MeCSV. Afin de favoriser la prise en compte de l'existant, nous avons également spécifié l'architecture d'un composant externe d'intégration, le composant adaptateur, qui permet d'intégrer les plates-formes de gestion métier avec le service de vérification *at runtime*. Ces éléments sont présentés dans ce chapitre.

VI.1. **Présentation générale du service de vérification**

D'après les propriétés que nous avons identifiées pour une solution de vérification opérationnelle de configurations (Section II.2.3 du Chapitre 2), trois capacités sont voulues pour ce service de vérification : la capacité de supporter une vérification opérationnelle d'instances de configuration basée sur les modèles de référence MeCSV, la capacité de prendre en compte des modifications des éléments de modèles de référence en cours d'utilisation et la capacité d'intégrer les systèmes de gestion existants.

- La capacité de vérification en ligne d'instances de configuration : le service de vérification doit pouvoir instrumenter les modèles de référence MeCSV préalablement définis et évaluer les contraintes *offline* et *online* sur des configurations candidates en phase d'exécution. Cette capacité nous permet de répondre au besoin de vérification opérationnelle à savoir la vérification de la conformité structurelle et la vérification de l'applicabilité opérationnelle des configurations proposées. De plus, cette vérification doit pouvoir être modulable pour, par exemple, restreindre le périmètre à évaluer, ne vérifier que certains types de contraintes en fonction des besoins courants d'assurance.

- La capacité d'édition des éléments des modèles de référence : cette capacité répond au besoin d'évolutivité des spécifications en phase d'exécution. Le service de vérification doit pouvoir supporter la mise-à-jour des modèles de référence à tout moment de son utilisation, de concert avec les évolutions de l'architecture du système géré ou des objectifs de gestion.

- La capacité d'intégration des systèmes de gestion existants : le choix de construire une vérification opérationnelle basée sur MeCSV permet d'obtenir un service indépendant des

plates-formes techniques existantes. Cependant, il faut également prendre en compte l'existant, le service de vérification doit pouvoir intégrer les systèmes de gestion métier.

Afin de supporter ces trois capacités, nous avons défini l'architecture d'un service de vérification, présentée dans la Figure 47, suivant deux plans :

- un plan opérationnel qui concerne l'utilisation du service pour exécuter la vérification de configurations candidates. Ce plan est porté par l'interface de vérification. L'interface de vérification offre des méthodes d'accès à un moteur d'exécution de la vérification. Ce moteur est capable d'analyser des instances de modèles conformes à la structure d'un modèle de référence MeCSV donné, d'évaluer des contraintes *offline* et *online* sur ces instances et de notifier les violations constatées. Le moteur de vérification s'appuie sur une base de modèles de référence où sont stockés les modèles de référence des différents types d'environnements gérés.
- un plan de contrôle qui permet la gestion de l'évolution des modèles de références. Le plan de contrôle s'appuie sur une interface d'édition de modèles de référence. Cette interface pose les bases d'une gestion du cycle de vie des modifications dynamiques des modèles de référence. En cours d'exécution, un administrateur ou tout expert (humain ou système) ayant la maîtrise de la configuration du système géré peut modifier le modèle de référence, par exemple ajouter des contraintes, modifier leur sévérité, les activer ou les désactiver.

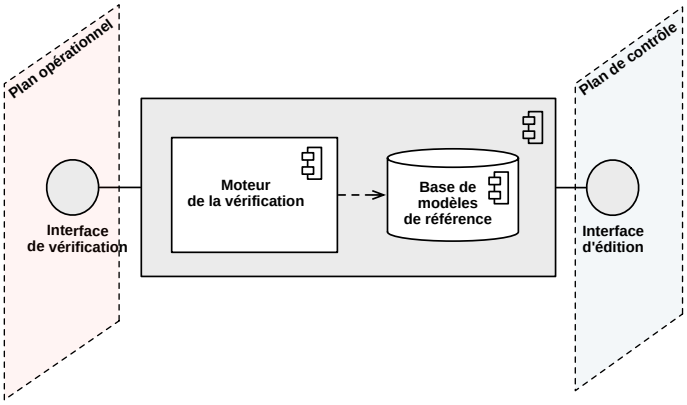

Figure 47 - Architecture du service de vérification

VI.2. Les interfaces du service

VI.2.1. Interface de vérification (plan opérationnel)

Nous avons formalisé une interface qui permet à un système de gestion de solliciter une vérification de configurations candidates. Les différentes méthodes qui la composent, sont spécifiées dans la Figure 48 à l'aide du langage IDL (Interface Description Langage) de l'OMG.

Cette interface propose un ensemble de méthodes permettant d'invoquer une vérification complète (Figure 48 - ①) ou sélective (Figure 48 - ②, ③ et ④) d'instances de configuration. Pour ce faire, le système de gestion doit envoyer au serveur de vérification des instances de configuration candidates et des instances d'état courant décrites dans un format conforme à un modèle de référence MeCSV donné.

```
module RuntimeConfigurationVerification {

    typedef string constraintLevel;
    typedef boolean constraintStatus;

    enum constraintType {offline, online};

    struct verificationError {
      string evaluationContext;
      constraintLevel level;
      constraintType type;
    };

    typedef sequence<verificationError> verificationErrorList;

    struct verificationResult {
      boolean result;
      verificationErrorList errorList;
    };

    typedef string referenceModelStructure;

    struct referenceModelInstance{
      string configInstanceModel;
      string stateInstanceModel;
    };


  //spécification du plan opérationnel

  interface VerificationOperation {
    // Aucun modèle de référence correspondant n'a été trouvé
    exception NotFoundReferenceModelException { string reason; };

     // L'instance envoyée en vérification est mal formée par rapport  à son modèle de référence
    exception MalformedInstanceException { string reason; };

     // L'instance ne mentionne aucune valeur de paramètres d'état opérationnel
    exception NoStateParameterValuesException;

     // Le niveau de sévérité demandé n'existe pas
    exception IllegalConstraintLevelException;


    // Vérification complète

    verificationResult validateAll (in referenceModelInstance instancePath)                    (1)
        raises (NotFoundReferenceModelException, MalformedInstanceException, NoStateParameterValuesException);

   // Vérification sélective

   // filtrée par rapport au type de contrainte (online ou offline).                            (2)

   verificationResult validateByConstraintType (in referenceModelInstance instancePath, in constraintType type);
       raises (NotFoundReferenceModelException, MalformedInstanceException,  NoStateParameterValuesException);

   // filtrée par rapport au niveau de sévérité.                                                (3)

   verificationResult validateByConstraintLevel(in referenceModelInstance instancePath, in constraintLevel level);
       raises (NotFoundReferenceModelException, MalformedInstanceException,  NoStateParameterValuesException,
       IllegalConstraintLevelException);

  // filtrée par rapport au type de contrainte et au niveau de sévérité.                        (4)

   verificationResult validateByConstraintFeatures(in referenceModelInstance instancePath, in constraintType type,
       in constraintLevel level);
       raises (NotFoundReferenceModelException, MalformedInstanceException,  NoStateParameterValuesException,
       IllegalConstraintLevelException);
  };
```

Figure 48 - Interface de vérification (plan opérationnel)

La vérification complète (méthode `validateAll`) permet d'évaluer toutes les contraintes actives du modèle de référence peu importe leur type (*offline* ou *online*) et leur niveau de sévérité. La vérification partielle permet de personnaliser la requête de vérification en sélectionnant les contraintes à évaluer. On peut choisir d'évaluer uniquement un certain type de contraintes (méthode `validateByConstraintType`), par exemple n'évaluer que les contraintes *online*. On peut ne vérifier que des contraintes d'un niveau de sévérité précisé (méthode `validateByConstraintLevel`), par exemple ne valider que les contraintes critiques. On peut également combiner les deux filtres (méthode `validateByConstraintFeatures`), par exemple ne valider que les contraintes qui sont *online* et critiques.

En tout cas, comme sortie, le service de vérification fournit le résultat de la vérification (type `verificationResult`) qui comprend la liste des erreurs de configuration illégales ou légales éventuelles.

L'une des problématiques dans le débogage d'erreurs de configuration réside dans la capacité des systèmes de vérification à retourner des résultats de vérification compréhensibles facilitant la résolution des violations [Sinz et al., 2005 ; Yin et al., 2011]. Nous avons pris soin de définir des messages d'erreurs riches mentionnant avec précision la contrainte non respectée, les éléments de configuration impactés et le niveau de sévérité de la violation constatée (type `verificationError`).

VI.2.2. **Interface d'édition (plan de contrôle)**

Une interface d'édition des éléments du modèle de référence a été définie pour permettre une gestion du cycle de vie des modifications des modèles de référence MeCSV (Figure 49). Cette interface répond au besoin fondamental de l'évolution opérationnelle des spécifications (structure du modèle de référence et contraintes).

```
  //module RuntimeConfigurationVerification (suite)

  // spécification du plan de contrôle
  interface VerificationControl{

     // Aucun modèle de référence correspondant n'a été trouvé
     exception NotFoundReferenceModelException {string reason;};

     // La contrainte demandée n'existe pas
     exception NotFoundConstraintException {string reason;};

    // Enregistrement d'un modèle de référence

     void registerReferenceModel(in string structureModelPath, in string constraintModelPath) raises   (1)
         (NotFoundReferenceModelException);

   // Edition de contraintes

     void updateConstraintStatus (in string constraintName, in constraintStatus status)
         raises (NotFoundConstraintException);
                                                                                                        (2)
     void updateConstraintLevel (in string constraintName, in constraintLevel level) raises
         (NotFoundConstraintException);

     void updateConstraintFeatures (in string constraintName, in constraintStatus status, in constraintLevel level)
         raises (NotFoundConstraintException);
  };
};
```

Figure 49 - Interface d'édition de modèles de référence MeCSV

Une fonctionnalité importante est le chargement de modèles de référence. Il s'agit, pour le paramétrage du service de vérification, d'enregistrer les modèles de références, en vue de son initialisation. Ces modèles peuvent être rechargés pendant l'exécution du service, par exemple recharger un modèle de référence MeCSV après modification. L'interface d'édition supporte ainsi l'ajout et la suppression d'éléments des modèles de référence, par exemple l'ajout de nouveaux paramètres d'états à surveiller, ainsi que la modification d'éléments existants, par exemple, la modification des caractéristiques des contraintes (Figure 49 – ②).

VI.3. **Description des composants de l'architecture**

L'architecture du service est constituée de composants internes qui réalisent la vérification et de composants externes dont l'objectif est d'intégrer l'existant. Les composants internes sont le moteur d'exécution de la vérification et la base de modèles de référence. Les composants externes consistent en des adaptateurs qui permettent la traduction de formalismes métier vers le formalisme MeCSV.

VI.3.1. Les composants supports de la vérification

VI.3.1.1. Le moteur d'exécution de la vérification

Le moteur de vérification est le cœur de la mise en œuvre opérationnelle de la vérification de configurations. Selon l'opération de vérification de l'interface invoquée, il évalue les instances de configuration reçues par rapport au modèle de référence du système qui a été défini. La Figure 50 illustre les blocs fonctionnels de ce moteur :

- le module de traitement de modèles : il permet l'analyse de modèles conformes au format MeCSV. Il se base sur le premier argument des opérations ①, ②, ③ et ④ de l'interface de vérification (Figure 48). Dans un premier temps, il recherche le modèle de référence MeCSV correspondant aux instances de configurations. Lorsqu'il le trouve, il vérifie le bien-formé des instances vis-à-vis de la structure du modèle de référence. Si tout va bien, il peut alors transmettre les instances reçues et le modèle de référence trouvé au module d'évaluation de contraintes. Si au contraire, aucun modèle de référence n'est trouvé ou bien que les instances sont mal formées, il lance les exceptions `NotFoundReferenceModel` et `MalformedInstanceException`.

- l'évaluateur de contraintes : il gère les requêtes de vérification. Il se base sur la nature de l'opération invoquée et donc sur les valeurs des arguments `type` et `level` dans le cas des requêtes de vérification sélective. Ce module analyse les contraintes du modèle de référence correspondant et les évalue sur les instances de modèles reçues. L'évaluation d'une contrainte se fait selon son statut, seules les contraintes actives sont considérées.

- le module de *reporting* : c'est un module de notification de résultats de vérification (type `verificationResult` de la Figure 48). Les résultats notifiés vont au delà de «vrai» ou «faux» et incluent les notions de sévérité de la violation et des éléments de configurations concernés.

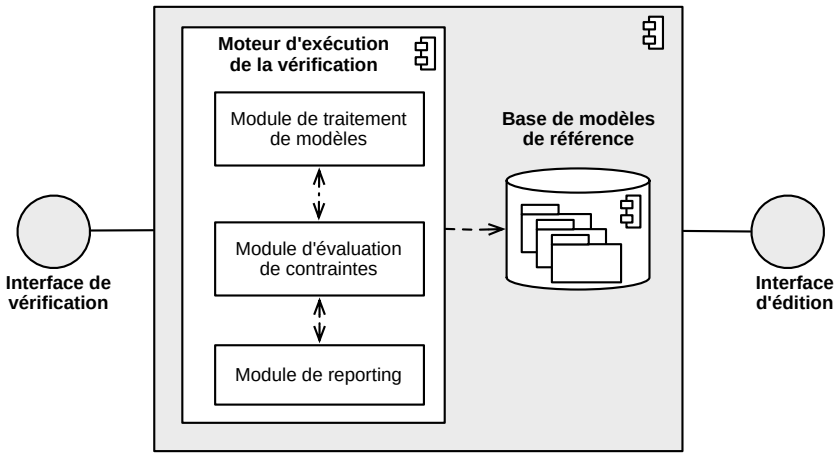

Figure 50 – Composants internes du service de vérification

VI.3.1.2. La base de modèles de référence

Le service de vérification s'adresse à plusieurs domaines applicatifs et donc à plusieurs systèmes de gestion. De ce fait, le composant «base de modèles de référence» a été conçu pour la sauvegarde des différents modèles de référence manipulés. Cette base supporte la création, la modification, la suppression, et l'interrogation des éléments de modèles que l'on retrouve habituellement dans une base de données.

Ceci permet d'une part leur consultation (par le moteur de vérification) et d'autre part leur mise à jour en cours d'opération, notamment par un administrateur via l'interface d'édition.

VI.3.2. Les composants d'intégration de l'existant at runtime

VI.3.2.1. Rôle d'un composant adaptateur

Dans une démarche de séparation des préoccupations, nous avons choisi d'externaliser la fonction de vérification afin de la découpler d'un système ou d'un protocole de gestion particulier. Ce choix répondait au besoin de favoriser une vérification en ligne de configurations indépendante des plates-formes face à la problématique d'un existant hétérogène. Afin d'assurer la prise en compte de l'existant *at runtime*, nous proposons d'intercaler entre le service de vérification et un système de gestion existant, un composant adaptateur spécifique. Ce composant intermédiaire d'intégration, est capable de récupérer et de traduire les instances de configuration et d'état opérationnel métier en instances de modèles de référence MeCSV.

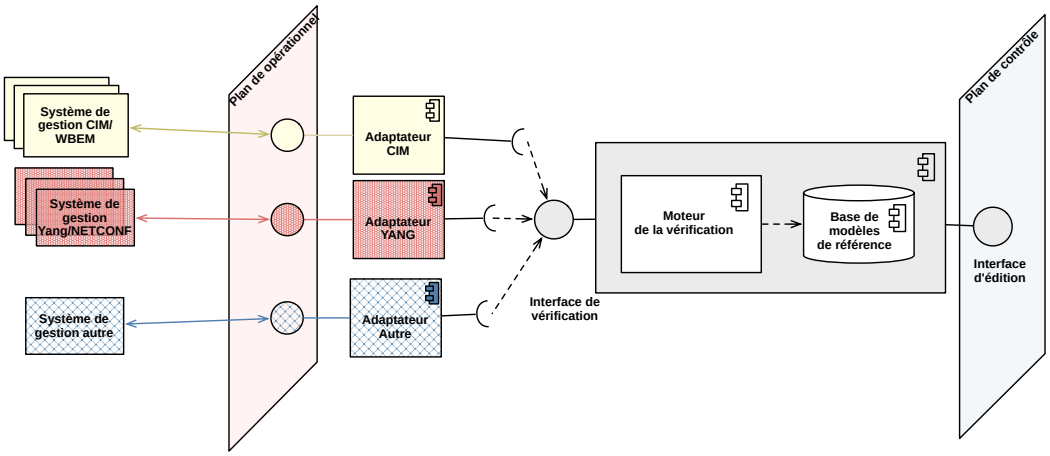

Figure 51 - Rôle des composants adaptateurs

Comme le montre la Figure 51, un composant adaptateur transforme, pour un système de gestion donné, l'interface de vérification du service basée sur MeCSV en interface de vérification basée sur les formalismes spécifiques du système de gestion. Il permet ainsi la traduction de requêtes de vérification «métier» en requêtes conformes à l'interface du service de vérification. Le développement de composants adaptateurs pour différentes plates-formes de gestion, en particulier pour les plates-formes de gestion standard, telles que celles basées sur CIM/WBEM ou YANG/NETCONF, permettraient une utilisation intégrée de notre service.

VI.3.2.2. Architecture d'un composant adaptateur

Afin de prendre en compte l'existant, nous avons défini l'architecture minimale d'un composant adaptateur illustrée dans la Figure 52.

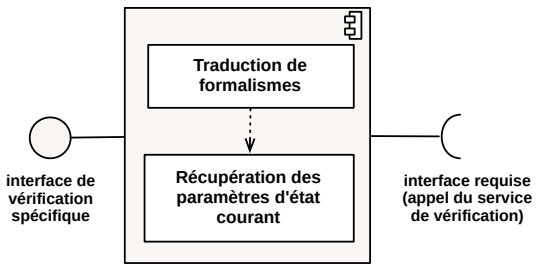

Figure 52 - Architecture d'un composant adaptateur

Le composant adaptateur expose une interface de vérification spécifique au module de décision qui le sollicite, et fournit des capacités de traduction de formalismes et de récupération des paramètres d'état courant.

– Interface de vérification spécifique d'un composant Adaptateur

L'interface de vérification du composant adaptateur permet au module de décision existant de solliciter une vérification de configurations dans son formalisme spécifique. C'est une adaptation de l'interface du service de vérification, compatible avec le modèle d'information de gestion métier. Le format attendu par le service de vérification (instance de modèle de référence MeCSV) reste transparent pour le module de décision client.

Comme le montre la Figure 53, cette interface reprend l'ensemble des méthodes du service de vérification à la différence près que le type de configurations candidates en entrée dépend du modèle de gestion métier. Les requêtes de vérification de configuration candidates métier seront traduites en requêtes de vérification conformes à l'interface du service de vérification.

```
interface MeCSVAdapter {


 // Vérification de base de configurations exprimées dans leur formalisme d'origine (spécifique)
  verificationResultList validateAll (in sequence <any> configurationData)
       raises (NotFoundReferenceModelException, MalformedInstanceException, NoStateParameterValuesException);

 // Vérification partielle de configurations exprimées dans leur formalisme d'origine (spécifique)

  // filtre par rapport au type de contrainte (online ou offline).

  verificationResultList validateByConstraintType (in sequence<any> configurationData, in constraintType type);
     raises (NotFoundReferenceModelException, MalformedInstanceException,  NoStateParameterValuesException);

  // filtre par rapport au niveau de sévérité.

  verificationResultList validateByConstraintLevel(in sequence<any> configurationData, in constraintLevel level);
     raises (NotFoundReferenceModelException, MalformedInstanceException,  NoStateParameterValuesException,
     IllegalConstraintLevelException);

  // filtre par rapport au type de contrainte et au niveau de sévérité.

 verificationResult validateByConstraintFeatures(in sequence<any> configurationData, in constraintType type,
     in constraintLevel level);
     raises (NotFoundReferenceModelException, MalformedInstanceException,  NoStateParameterValuesException,
     IllegalConstraintLevelException);

 };
```

Figure 53 - Interface de vérification de l'adaptateur

– Module de récupération des paramètres d'état

Pour les besoins de la vérification opérationnelle, le service de vérification a besoin d'avoir accès aux valeurs de paramètres d'état courant. Le module de récupération des paramètres d'état anticipe ce besoin. Il gère la connexion au module de supervision existant et récupère les valeurs courantes des paramètres d'état du modèle de référence MeCSV correspondant.

– Module de traduction de formalismes

Le service de vérification a besoin de recevoir des instances de gestion conformes à la vue structurelle d'un modèle de référence MeCSV existant. Le module de traduction de formalismes est donc responsable de la transformation des informations de gestion spécifiques (configurations candidates et valeurs d'état courant) en des instances conformes à un modèle de référence MeCSV. Concrètement, cela nécessite des correspondances entre modèles de gestion métier et modèles de référence MeCSV équivalents.

VI.4. **Déroulement du processus de vérification**

Le service de vérification peut ainsi supporter deux types de vérification : une vérification native basée sur le formalisme MeCSV, le système de gestion sollicite directement le service de vérification et une vérification intégrée, le système de gestion sollicite le service de vérification via un composant adaptateur.

VI.4.1. **Vérification native de configurations**

Le système de gestion envoie des requêtes de vérification directement au service. Ces requêtes contiennent des instances de configuration et d'état courant dans un format conforme à un modèle de référence MeCSV existant.

La Figure 54 présente les interactions entre un système de gestion, en particulier ses modules de décision et de supervision, et le service de vérification. Deux types d'interactions peuvent être observées : des interactions internes au système de gestion (interactions i et ii) et des interactions externes entre le système gestion et le service de vérification (interactions 1 et 2).

Lorsque le module de décision choisit une configuration candidate qu'il souhaite vérifier :

- il se charge de récupérer auprès du module de supervision les valeurs courantes des paramètres d'état du modèle de référence MeCSV correspondant (Figure 54 - i),
- puis il transforme la configuration candidate et les valeurs d'état collectées en instances de modèle de référence MeCSV (Figure 54 – ii),
- qu'il soumet au service de la vérification (Figure 54 - 1),
- le service de vérification évalue la configuration candidate à la fois structurellement et en fonction des informations opérationnelles récupérées à l'étape i et retourne les résultats de la vérification (Figure 54 - 2).

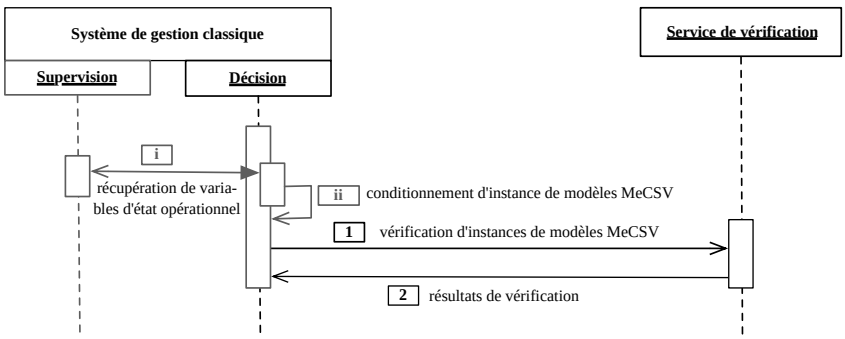

Figure 54 – Processus de vérification native de configurations

VI.4.2. **Vérification de configurations intégrée**

Le système de gestion sollicite le service de vérification via un composant adaptateur : son module de décision envoie des requêtes de vérification métier, c'est-à-dire contenant des configurations candidates dans son formalisme propre. L'adaptateur se charge de récupérer les informations nécessaires d'état courant et de transformer les informations de gestion métier en instances de modèle de référence MeCSV qu'il envoie ensuite au service de vérification.

La Figure 55 présente les interactions entre un système de gestion existant, son adaptateur et le service de vérification. Lorsque le module de décision choisit une configuration candidate qu'il souhaite vérifier :

- il l'envoie au composant adaptateur à qui il délègue la sollicitation de sa vérification (Figure 55 – 1a).
- afin de permettre la vérification opérationnelle, le composant adaptateur se connecte au module de supervision existant pour la collecte des valeurs courantes des paramètres d'état opérationnel (Figure 55 – 1a').
- le composant adaptateur transforme ces deux informations dans un formalisme compatible avec un modèle de référence existant (Figure 55 – 1a'') et le soumet au service de vérification (Figure 55 – 1b)
- le service de vérification évalue la configuration candidate à la fois structurellement et en fonction des informations opérationnelles récupérées à l'étape 1a' et retourne les résultats de la vérification (Figure 55 – 2a et 2b).

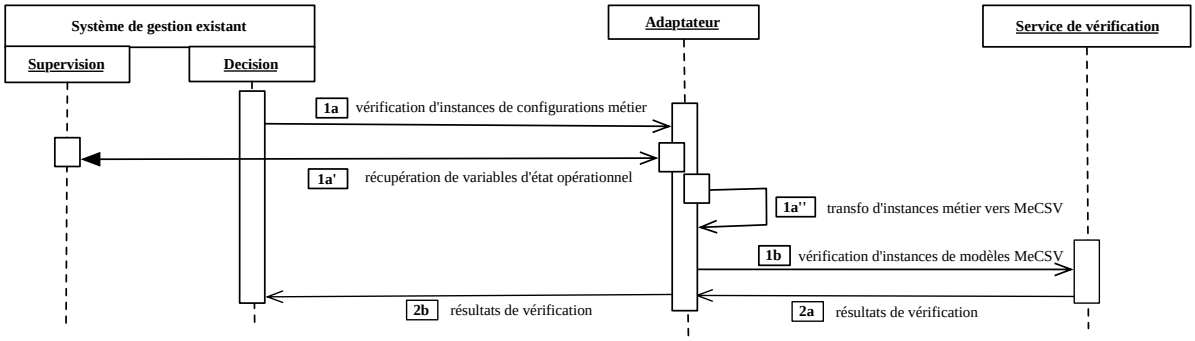

Figure 55 – Processus de vérification de configurations intégrée

VI.5. **Conclusion**

Dans ce chapitre, nous avons présenté l'architecture du service de vérification qui permet d'exploiter les modèles de référence MeCSV pour la vérification en ligne de configurations candidates. La conception du service de vérification représentait la deuxième étape pour la création du cadriciel que nous proposons. Le service a été mis en œuvre suivant deux plans, un plan opérationnel et un plan de contrôle.

- Le plan opérationnel permet l'utilisation du service de vérification et repose sur une interface de vérification qui offre une série de méthodes d'accès à un moteur d'exécution de la vérification. Ce dernier s'appuie sur le composant « base de modèle de référence » qui permet la persistance des divers modèles de référence à manipuler et supporte ainsi leur instrumentation.
- Le plan de contrôle permet de supporter l'évolution du modèle de référence et s'appuie sur une interface d'édition qui permet la gestion du cycle de vie des éléments du modèle de référence en cours d'exécution.

Nous avons également défini une architecture de composants adaptateurs pour favoriser la prise en compte de l'existant *at runtime*. Un composant adaptateur agit comme une passerelle entre un système de gestion donné et le service de vérification. Dès lors le module de décision existant peut requérir une vérification de configurations dans son formalisme propre.

Afin de valider expérimentalement l'architecture du service proposée, nous avons implémenté un prototype du service et de ses composants. Ce prototype a supporté avec succès la vérification en ligne de configurations de serveurs de messages `JORAM` et de machines virtuelles modélisées en CIM. Le prototypage et les expérimentations font l'objet du prochain chapitre.

Chapitre 7

Prototype et cas d'expérimentation

L'objectif de ce dernier chapitre est de présenter le prototype du service de vérification. Nous avons implémenté un prototype du service de vérification et de ses composants. Ce prototype a été expérimenté dans la vérification en ligne de configurations issues de deux domaines applicatifs différents : celui de la reconfiguration dynamique de serveurs de messages `JORAM` et celui de la reconfiguration dynamique de machines virtuelles en environnement CIM/WBEM. Dans le cas `JORAM`, nous avons développé un système de gestion *ad hoc* capable de solliciter directement le service de vérification. Dans le cas CIM/WBEM, nous avons implémenté un adaptateur dédié qui fait le lien entre un serveur WBEM et le service de vérification.

VII.1. **Prototypage du service de vérification**

L'approche dirigée par les modèles a été poursuivie dans l'implémentation du service de vérification. Nous avons fait le choix du langage OCL pour la réalisation du prototype et nous nous sommes appuyés à cet effet, sur la suite de librairies open source d'évaluation de contraintes OCL, `Dresden OCL` [Dresden-OCL].

Les interfaces de vérification et d'édition ont été implémentées sous forme d'interfaces Java. Les sections suivantes détaillent l'implémentation des différents composants du service, à savoir le moteur de vérification, la base de modèles de référence. Nous avons également implémenté un adaptateur de base à étendre pour l'implémentation d'adaptateurs spécifiques.

VII.1.1. **Implémentation du moteur de vérification**

VII.1.1.1. **Présentation de la suite d'outils logiciels Dresden OCL**

`Dresden OCL` est une suite d'outils logiciels développée à l'Université Technologique de Dresden en Allemagne, pour l'évaluation de contraintes OCL sur différents types de modèles comme UML, Ecore et Java. `Dresden OCL` se présente sous la forme d'une architecture modulaire organisée en cinq couches (Figure 56) :

- La couche `API` fournit une façade d'accès aux outils de `Dresden OCL`.
- La couche `Tools` contient les outils logiciels de la suite à savoir un parseur de contraintes OCL (`OCL Parser`), un interprétateur OCL (`OCL Interpreter`), un parseur de modèles (`Model Browser`) permettant d'instrumenter les modèles et les instances de modèles manipulés et des générateurs de code OCL vers Java ou SQL.
- La couche `Registry` propose un bus de modèles qui permet de gérer les différents métamodèles associés aux modèles et instances de modèles manipulés.
- La couche `OCL` contient la syntaxe et la sémantique OCL utilisées par les outils logiciels.

- La couche `Variability` offre la possibilité d'adapter `Dresden OCL` à des métamodèles et des modèles utilisateurs.

Nous avons utilisé la version 3.1 de `Dresden OCL` conforme à la spécification OCL 2.2.

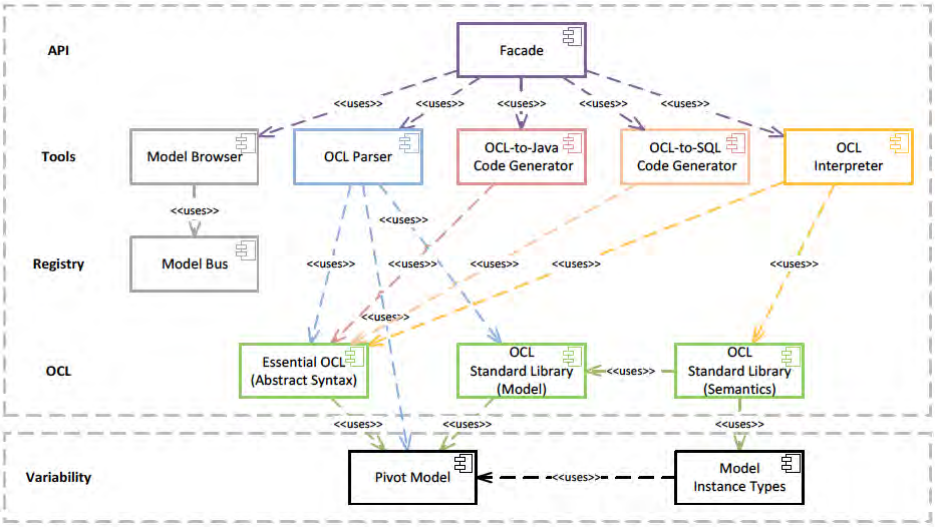

Figure 56 - Architecture de Dresden OCL

VII.1.1.2. Utilisation et adaptation de Dresden OCL

Le moteur de vérification que nous avons développé, utilise l'API de `Dresden OCL` pour accéder aux outils de navigation de modèles (`Model Browser`), d'analyse (`OCL Parser`) et d'interprétation d'OCL (`OCL Interpreter`).

– Implémentation du module de traitement de modèles

La fonctionnalité attendue du module de traitement de modèles correspond à la fonctionnalité offerte par le composant `Model Browser` de `Dresden OCL`. Le composant `Model Browser` a été utilisé tel quel en tant que module de traitement de notre moteur de vérification.

Les modèles issus de l'application du profil UML pour MeCSV, pour des raisons techniques, n'ont pas pu être exploités directement avec les outils de `Dresden OCL`. L'implémentation du modèle de référence s'est traduite par un modèle `Ecore` simplifié pour sa partie structurelle (éléments de configuration et d'état opérationnel) et un fichier OCL pour ses contraintes. Dans le cadre du prototype, tout modèle de référence défini avec le profil UML passe par une phase de transformation en modèle Ecore pour les besoins de la vérification.

– Implémentation du module d'évaluation de contraintes

Les outils `OCL Parser` et `OCL Interpreter` de `Dresden OCL` permettent d'analyser et d'interpréter des contraintes OCL sur des modèles et sont donc candidats à l'implémentation du module d'évaluation de contraintes. Toutefois, les contraintes définies dans MeCSV possèdent des sémantiques additionnelles par rapport à une contrainte OCL de base, elles nécessitent un traitement spécifique inexistant dans les outils de `Dresden OCL`.

Dans un premier temps, nous avons utilisé les capacités de la couche `Variability` pour étendre la sémantique de base d'une contrainte OCL à la notion de type *online* et *offline*, de statut et de niveau de sévérité,. Dans un deuxième temps, nous avons étendu les outils `OCL Parser` et `OCL Interpreter` pour qu'ils puissent gérer ces nouveaux attributs.

– **Implémentation du module de *reporting***

`Dresden OCL` fournit une interface de notification de résultat de vérification (`IIResult`) qui permet de retourner vrai ou faux en cas d'erreurs ainsi que des informations sur la contrainte évaluée. Nous l'avons également étendue pour des résultats de vérification plus expressifs notamment avec la mention du type de contrainte évaluée, du niveau de sévérité de sa violation et de la configuration affectée.

VII.1.2. Implémentation de la base de modèles de référence

Actuellement, la base de modèles est un système de fichiers organisé en répertoires qui s'appuie sur les capacités de persistance et de sérialisation du format `XMI` [XMI, 2011].

L'organisation se fait par modèle de référence comme le montre la Figure 57. Une fois le modèle de référence MeCSV défini en phase de conception, il est chargé dans la base de modèles de référence où est initialisé un dossier portant son nom et contenant trois sous-dossiers : le premier pour la partie structurelle du modèle de référence (répertoire `model`), le deuxième pour les contraintes (répertoire `constraints`) et le troisième qui servira à stocker les instances reçues à chaque requête de vérification (répertoire `modelinstance`).

La modification dynamique du modèle de référence se fait via l'utilisation d'un éditeur de modèle comme Eclipse MDT.

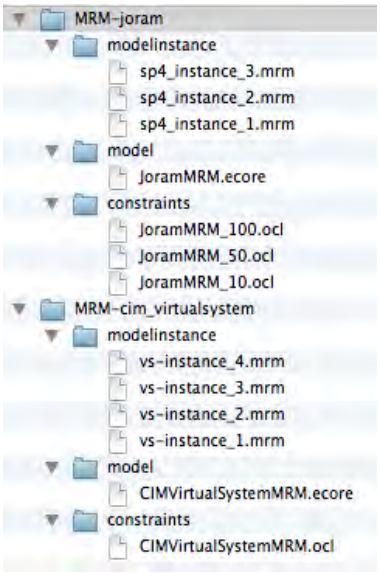

Figure 57 – Base de modèles de référence

VII.1.3. Implémentation d'un adaptateur de base

Nous nous sommes appuyés sur le patron de conception logicielle «Patron de méthodes» (Template Method) [Gamma et al., 1995] pour élaborer le prototype d'un adaptateur de base que vont étendre les adaptateurs spécifiques de chaque plate-forme de gestion.

Le patron de conception «Patron de méthodes» permet de définir le squelette d'un algorithme à l'aide d'opérations abstraites dont le comportement concret est délégué aux sous-classes qui implémenteront ces opérations. Son utilisation se prête au développement des composants adaptateurs : en effet, les adaptateurs ont un objectif commun de supporter une vérification de configurations dans le formalisme d'origine, à la différence près que les détails de connexion

au module de supervision source et de traduction des formalismes sont spécifiques à chaque plate-forme de gestion métier.

Nous avons implémenté un prototype de l'adaptateur de base qui se présente sous la forme d'une classe abstraite Java. Cette classe définit le squelette d'un algorithme utilisant deux méthodes abstraites que les sous-classes (adaptateurs spécifiques) devront implémenter pour chaque plate-forme de gestion à intégrer :

- la première (`getStateParameter`) permet de se connecter au module de supervision et de récupérer des informations d'état courant à partir de noms de classes `ManagedElement` et d'attributs `StateParameter` d'un modèle de référence MeCSV donné,
- la deuxième (`transformToMeCSVCompliant`) permet de transformer les instances de modèles de gestion métier en instances d'un modèle de référence MeCSV donné.

La Figure 58 illustre l'utilisation du patron de conception. La classe abstraite `AbstractMeCSVAdapter` propose des méthodes concrètes d'invocation de la vérification qui reçoivent toutes en premier argument les instances de configurations spécifiques. L'algorithme de ces méthodes fait appel aux deux méthodes abstraites de la façon suivante :

- à la réception d'une requête de vérification de configurations spécifiques, on analyse d'abord la nature de la vérification demandée, si cette vérification inclut l'évaluation de contraintes *online*, alors on appelle la méthode `getStateParameter` pour la collecte des informations d'état courant auprès du module de supervision.
- muni des informations de configuration et d'état opérationnel, on appelle la méthode `transformToMeCSVCompliant` qui retourne une instance conforme à un modèle de référence MeCSV qu'on peut dès lors soumettre au service de vérification.

A partir de cet adaptateur de base, nous pouvons donc développer un adaptateur concret, par exemple, nous avons développé en Java, un adaptateur pour l'intégration des environnements CIM/WBEM qui est présenté dans la section VII.2.2.1. Cet adaptateur est une classe Java qui hérite de l'adaptateur de base. L'implémentation de la méthode `getStateParameter` permet la connexion à un serveur WBEM et la consultation du `CIM Repository` pour la rcollecte de paramètres d'état. L'implémentation de la méthode `transformToMeCSVCompliant` favorise la transformation d'instances CIM en instances MeCSV.

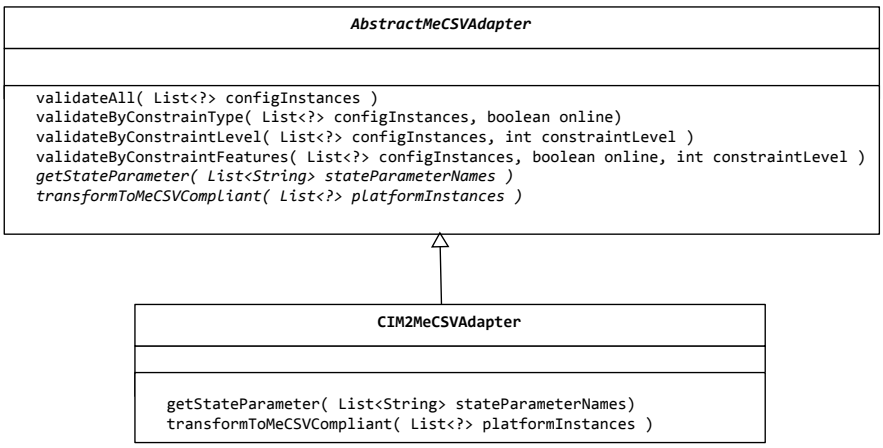

Figure 58 - Patron de conception d'adaptateurs à partir de l'adaptateur de base

Comme cela a déjà été souligné dans la section V.1.2.1 pour la transformation de modèles de gestion existants en phase de conception, nous recommandons la définition de ces transformations au niveau métamodèle, c'est-à-dire directement entre les constructeurs du langage de

gestion métier et MeCSV. Cette possibilité permettrait de définir un adaptateur une fois pour toutes et le réutiliser avec tout système de gestion basé sur ce langage. C'est le cas par exemple des systèmes de gestion utilisant les modèles d'information CIM ou YANG.

VII.2. Présentation des cas d'expérimentation

VII.2.1. Vérification native de configurations

Afin de tester la faisabilité de l'utilisation du service de vérification et évaluer le surcoût temporel introduit, nous avons expérimenté le prototype dans plusieurs scénarii de reconfiguration dynamique de la plate-forme MOM `JORAM` présentée dans la section V.2.1.1. Pour ce faire, nous avons développé un système de gestion *ad hoc* pour JORAM composé d'un module de supervision et d'un module de décision. Nous avons également prédéfini quatre configurations candidates. En phase d'exécution, le composant de décision choisit aléatoirement une configuration candidate et sollicite sa vérification avant application.

VII.2.1.1. Développement d'un système de gestion *ad hoc* pour JORAM

La plate-forme `JORAM` ne dispose pas de système de gestion, elle offre cependant une interface d'administration basée sur JMX. Les composants de `JORAM` (serveur, service, destination, usine de connexion) implémentent à cet effet, l'interface `MBean` et fournissent des méthodes pour leur supervision et leur reconfiguration. Nous avons développé, pour les besoins de l'expérimentation, un système de gestion *ad hoc* pour la plate-forme `JORAM`. Ce système de gestion s'appuie sur l'interface d'administration JMX et comprend un module de supervision et un module de décision.

- Le module de supervision est capable de surveiller un environnement `JORAM` par un mécanisme de *polling*. Il comprend un ensemble d'opérations de consultation qui, à partir de la spécification d'une ou plusieurs cibles (composants `JORAM`), peut récupérer les valeurs courantes des paramètres d'état opérationnel de ces cibles.

- Le module de décision est capable de choisir une configuration dans une base de configurations prédéfinies et de solliciter son évaluation par le service de vérification. Il est également capable de créer, à partir d'une configuration et de valeurs d'état opérationnel, des instances de modèles conformes à la partie structurelle du modèle de référence.

VII.2.1.2. Architecture du système géré cible

Nous avons prédéfini quatre configurations candidates conformes au modèle de référence MeCSV de la plate-forme, différentes en taille (nombre de paramètres de configurations) et en complexité (composition et dépendances entre classes de configurations).

- Première configuration (cas 1) : cette configuration décrit une architecture centralisée composée d'un serveur de messages qui fournit un service de gestion de connexions, un service de proxy TCP et un service de nom. Ce serveur propose également une usine de connexions, une file de messages, une file de messages de type Dead Message Queue ou DMQ (sorte de file de messages «poubelle» pour les messages non délivrés) et un accès utilisateur anonyme pour un total de 57 paramètres de configuration.

- Deuxième configuration (cas 2) : cette configuration définit une architecture distribuée de deux serveurs de messages, chacun proposant les services de connexions habituels, une usine de connexion, une file de messages, une file DMQ et un accès utilisateur anonyme.

Le premier serveur fournit également un service de nom pour un total de 110 paramètres de configuration.

- Troisième configuration (cas 3) : cette configuration spécifie une architecture distribuée de trois serveurs de messages pour un total de 197 paramètres de configurations.

- Quatrième configuration (cas 4) : afin d'apprécier le passage à l'échelle de la vérification que nous proposons, nous avons défini un quatrième cas de 1970 paramètres de configuration obtenu en dupliquant dix fois la troisième configuration. Ce cas n'a pas été expérimenté sur une plate-forme réelle. Nous avons programmé un simulateur qui fait varier de façon aléatoire les valeurs de paramètres d'état opérationnel.

Chaque configuration possède une version correcte qui sert de «configuration témoin» et une version où des erreurs structurelles violant des contraintes *offline* ont été introduites.

VII.2.1.3. Scénario de reconfiguration et processus de vérification

La plate-forme JORAM est initialisée avec la première configuration. Afin de simuler des environnements d'utilisation réels et d'influer sur les conditions opérationnelles, des applications clientes (que nous avons programmées) génèrent une charge importante via des échanges de messages fictifs.

Le composant de décision choisit de façon périodique et aléatoire, une des configurations candidates, demande la récupération des valeurs courantes des paramètres d'état opérationnel précisés dans le modèle de référence et crée une instance du modèle de référence qu'il soumet au service de vérification. Le nombre de paramètres d'état surveillés croît avec la taille de la configuration.

A réception de la requête, le service de vérification vérifie la conformité de l'instance reçue vis-à-vis du modèle de référence MeCSV qu'il instrumente et procède à son évaluation par rapport aux contraintes actives. Nous avons procédé à l'exécution de chaque requête de vérification avec 10, 50 et 100 contraintes (un extrait de ces contraintes est présenté dans la section V.2.2.3). Le nombre de contraintes *online* représente 20% du nombre total de contraintes.

Le Tableau 3 récapitule les données de l'expérimentation, à savoir le nombre d'instances de classes de configuration (`Configuration`) et de classes d'état opérationnel (`ManagedElement`) et le nombre de paramètres de configuration (`ConfigurationParameter`) et de paramètres d'état (`StateParameter`) dont les valeurs ont été traitées pour chaque vérification.

Cas	*Nombre d'instances de classes configuration*	*Nombre de paramètres de configurations*	*Nombre d'instances de classes d'état*	*Nombre de paramètres d'état*
Cas 1	9	57	4	25
Cas 2	18	110	10	64
Cas 3	30	197	16	103
Cas 4	300	1970	16	103

Tableau 3 – Récapitulatif des données de l'expérimentation

Pour chaque cas de configuration et de contraintes, nous avons effectué 100 mesures du temps d'exécution de la vérification T_{verif} (Figure 59).

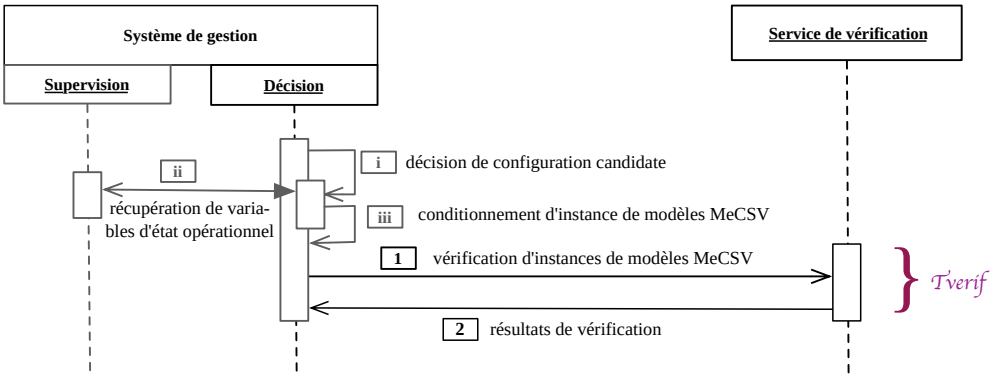

Figure 59 - Temps d'exécution de la vérification T_{verif}

Le temps T_{verif} comprend le temps que prend le module de traitement des modèles pour vérifier la conformité de l'instance reçue vis-à-vis du modèle de référence MeCSV, le module d'évaluation de contraintes pour évaluer les contraintes du modèle de référence MeCSV sur les instances reçues et le temps que prend le module de *reporting* pour préparer les résultats de vérification. Le temps de *reporting* est négligeable.

Les tests ont été exécutés sur une machine Intel® Core™ 2 Duo de 2,66GHz et 4 Giga-octets de mémoire RAM. Le système de gestion, le système `JORAM` et le service de vérification s'exécutent sur la même machine.

VII.2.1.4. Résultats et discussions

VII.2.1.4.1. Faisabilité de la vérification

Le service de vérification a évalué les instances avec succès et retourné des erreurs pour les configurations invalides. Un exemple de rapport d'erreurs généré est illustré dans la trace d'exécution de la Figure 60. Il correspond au résultat de l'évaluation de dix contraintes sur une instance de modèle issue du cas 3.

On y trouve des indications sur la nature des violations constatées: le type de contraintes non respectées (*offline*, *online*), leur niveau de sévérité, (`WARNING`, `ERROR`, `FATAL`), l'élément de configuration affecté (un serveur, une file de messages). Un résultat de vérification possède également un attribut booléen `result` qui signale si des erreurs ont été détectées ou non.

```
INFO [main] -  >>>> Registering the MeCSV Reference Model
INFO [main] -  >>>> structure :ReferenceModelRepository/MRM-joram/model/JoramMRM.ecore
INFO [main] -  >>>> constraints : ReferenceModelRepository/MRM-joram/constraints/JoramMRM_10.ocl
INFO [main] -  >> Validator : Registering the MeCSV reference model structure...
INFO [main] -  >> Validator : Registering the MeCSV reference model constraint...
INFO [main] -  >> Validator : service initialized...
INFO [main] -  >>>> Candidate Instance for validation : ReferenceModelRepository/MRM-joram/modelinstance/sp4_instance_3.mrm
INFO [main] -  >>>> Begin validation request
INFO [main] -  >> Validator : validation request received : validateAll...
INFO [main] -  >> Validator | Reporting module : output...
verificationResult {
   result : false
   verificationErrors : [
      { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s1 }
      { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s0 }
      { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : WARNING, evaluationContext : Queue  q0 }
      { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : WARNING, evaluationContext : Queue  q1 }
      { constraintName : MustBeRegisteredConnectionFactory, constraintType : offline, constraintLevel : ERROR, evaluationContext : ConnectionFactory null }
      { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s2 }
   ]
}
INFO [main] -  >>>> End validation request
```

Figure 60 - vérification de configurations issues du cas 3 avec 10 contraintes

La détection de contraintes *online* (contrainte *online* `OperationalApplicableNbMaxMsg` dans la Figure 60) vient renforcer le besoin que nous avons identifié, de la prise en compte de l'état courant lors de vérifications en ligne de configurations.

Ce premier cas d'expérimentation montre la faisabilité de notre approche de vérification notamment la capacité du prototype de service de vérification que nous avons implémenté, à effectuer des vérifications de contraintes à la fois *offline* et *online*. Aucune configuration présentant des violations de contraintes n'a été appliquée.

Nous avons précédemment montré, dans la section V.2 du Chapitre 5, la capacité du métamodèle MeCSV à couvrir la représentation de concepts liés à la vérification opérationnelle de configurations d'intergiciels orientés messages. Ce nouveau résultat vient confirmer la capacité de MeCSV à servir de base à la mise en œuvre effective d'une vérification en ligne de configurations réelles de systèmes JORAM.

VII.2.1.4.2. Surcoût temporel de la vérification

La mesure du surcoût temporel de la vérification est un résultat important, en particulier la mesure de l'impact du nombre de paramètres de configurations à vérifier et de celui du nombre de contraintes sur le temps d'exécution de la vérification.

Les résultats présentés dans la Figure 61 proviennent du calcul de la moyenne arithmétique de 100 mesures du temps d'exécution de la vérification T_{verif} pour chaque cas de configurations (Cas 1, 2 et 3) et de nombre de contraintes (10, 50 et 100).

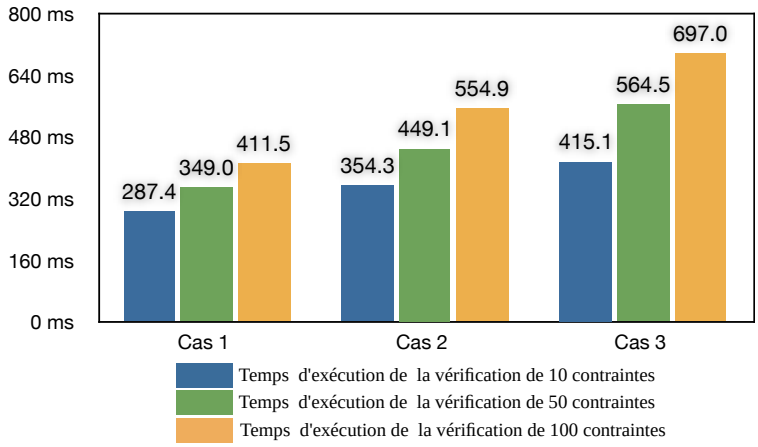

Figure 61 – Mesures du temps d'exécution de la vérification (T_{verif})

Le temps de vérification pour les trois cas de configuration testés dans des conditions réelles est en dessous des 700 millisecondes. Comme le montrent ces résultats, le temps d'exécution croît en fonction du nombre de paramètres de configuration présentés et du nombre de contraintes à vérifier. Cependant ce temps n'est ni proportionnel au nombre de paramètres de configuration, ni au nombre de contraintes. Par exemple, alors que le nombre de paramètres de configuration croît de 346% entre le cas 1 (57 paramètres) et le cas 3 (197 paramètres), le temps de vérification augmente de 159%. De façon similaire, alors que le nombre de contraintes passe de 10 à 100 soit un facteur de 10, le temps moyen de vérification montre un facteur de multiplication inférieur à 1.6. Le temps observé présente une tendance linéaire.

Nous pouvons en conclure que dans le cas de systèmes à taille modérée, ni le nombre de contraintes, ni le nombre de paramètres de configuration n'influe significativement sur le temps d'exécution de la vérification.

– **Passage à l'échelle**

Le cas 4 donne un premier aperçu du passage à l'échelle de l'approche de vérification (Figure 62). Le temps pour les trois jeux de contraintes est en dessous de 1800 millisecondes avec des mesures qui confirment les premiers résultats, à savoir : alors que le nombre de paramètres de configuration passe de 57 au cas 1 à 1970 au cas 4, soit un facteur de 34.56, le temps de vérification reste multiplié en moyenne par 3.8.

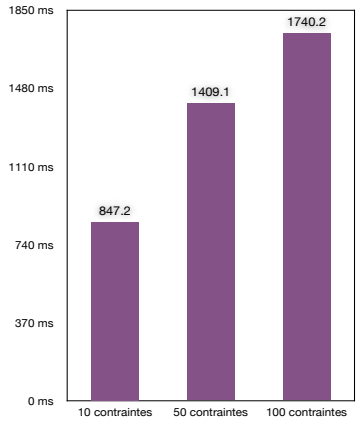

Figure 62 - Mesures du temps d'exécution (T_{verif}) pour le cas 4

VII.2.1.4.3. Discussion des résultats

Ces premiers résultats confirment la nécessité d'une vérification opérationnelle de configurations et la faisabilité d'une telle vérification à partir du métamodèle MeCSV et du service de vérification que nous avons conçu. La vérification en ligne a généré un surcoût temporel plutôt faible si on considère des configurations de taille modérée dans un environnement non contraint tel que celui de cette expérimentation. Ce surcoût engendré est surtout à confronter au nombre d'erreurs évitées. Quand bien même les configurations candidates seraient contrôlées et vérifiées d'avance, l'impossibilité de prévoir les conditions opérationnelles, variables dans le temps, reste une source d'instabilité qui peut pénaliser la réussite d'une reconfiguration dynamique. L'utilisation de notre cadriciel est une solution pour prévenir ces instabilités.

VII.2.2. Vérification intégrée de configurations

Afin d'apprécier la faisabilité de l'intégration par la définition de composants adaptateurs, nous avons implémenté un adaptateur pour les environnements CIM/WBEM que nous avons expérimenté dans différents cas de reconfiguration dynamique de machines virtuelles modélisées dans le Chapitre 5. `OpenPegasus` [OpenPegasus] est le serveur de gestion WBEM qui a été utilisé. Nous avons travaillé directement sur les modèles CIM stockés dans la base d'informations «`CIM Object Repository`».

VII.2.2.1. Implémentation d'un adaptateur pour CIM/WBEM

A partir de l'adaptateur de base, nous avons développé un prototype d'adaptateur pour les environnements de gestion CIM/WBEM appelé `CIM2MeCSVAdapter`. Il a été implémenté en tant que client WBEM via l'API SBLIM. Ce rôle est justifié par le fait qu'il a besoin d'accéder à des informations d'état opérationnel et donc de se connecter au module de supervision d'un serveur WBEM.

Ce composant propose une interface de vérification qui permet à un environnement de gestion basé sur les standards CIM/WBEM d'invoquer directement les diverses opérations de vérification sur des instances de configuration (instances de `CIM_SettingData`). Nous avons implémenté les différentes méthodes abstraites de l'adaptateur de base (section VII.1.3) pour la communication avec le module de supervision source et la transformation des instances CIM en instances conformes à un modèle de référence MeCSV donné.

- Traduction des formalismes : l'adaptateur `CIM2MeCSVAdapter` implémente la méthode abstraite `transformToMeCSVCompliant` de l'adaptateur de base (Figure 58) pour la traduction d'instances CIM en instances de modèles de référence MeCSV. A l'image des règles de transformation de modèles CIM vers MeCSV que nous avons spécifiées dans la section V.3.1.1, notre choix d'implémentation de cette traduction a consisté à programmer des transformations en Java relativement au patron de configuration `CIM_SettingData` et aux constructeurs de MeCSV. Le cadre EMF fournit une API de création dynamique d'instances `Ecore` que nous avons également exploitée. Ainsi toute instance CIM peut être automatiquement transformée en instance de modèle de référence.

- Récupération de paramètres d'état : le composant adaptateur `CIM2MeCSVAdapter` implémente la méthode abstraite `getStateParameter` de l'adaptateur de base pour l'acquisition de variables d'état opérationnel. Via l'utilisation de l'API SBLIM, cette méthode permet à l'adaptateur CIM2MeCSVAdapter de se connecter à un serveur WBEM et, de lancer des requêtes de consultation et de récupération d'instances CIM, à partir des noms de `ManagedElement` et de `StateParameter` définis dans un modèle de référence MeCSV. La version actuelle de l'adaptateur récupère toutes les informations opérationnelles, il ne les filtre pas selon les contraintes *online* actives ou inactives.

- Connaissance du modèle de référence : le composant adaptateur a accès aux classes du modèle de référence MeCSV correspondant via son identifiant uniforme de ressources URI (Uniform Resource Identifier). Il travaille ainsi avec des versions de modèles de référence à jour.

VII.2.2.2. Mise en œuvre d'un système de gestion basé sur OpenPegasus

– Présentation d'OpenPegasus

`OpenPegasus` est une implémentation open-source des standards CIM et WBEM du DMTF. Il reprend l'architecture WBEM et permet de disposer d'un `CIMOM`, d'une base de gestion de modèles et d'instances CIM, le `CIM Repository` ainsi que des interfaces de programmation et d'accès à des *providers*, clients et *listeners* CIM. L'architecture WBEM et ses composants sont détaillés dans la section I.2.2.2.2 du Chapitre 1.

– Implémentation d'un client WBEM pour le module de décision

Nous avons implémenté un module de décision dont le rôle est de sélectionner des configurations dans le `CIM Repository` selon des scénarii de reconfiguration et d'utiliser l'interface de vérification spécifique de l'adaptateur `CIM2MeCSVAdapter` pour demander leur vérification. La Figure 63 présente l'architecture de vérification intégrée de configurations CIM :

- le module de décision invoque les opérations de vérification de configurations (instances de classes héritant de `CIM_SettingData`) sur l'interface de vérification de l'adaptateur `CIM2MeCSVAdapter`.

- A la réception d'une demande de vérification, l'adaptateur déclenche la récupération d'informations opérationnelles complémentaires auprès du `CIMOM` (instances de classes héritant de `CIM_ManagedElement` mais n'héritant pas de `CIM_SettingData`)
- L'adaptateur transforme ces informations de gestion métier en instances conformes à un modèle de référence MeCSV puis les soumet au service de vérification.
- Le service de vérification vérifie ces instances et renvoie les résultats de vérification.

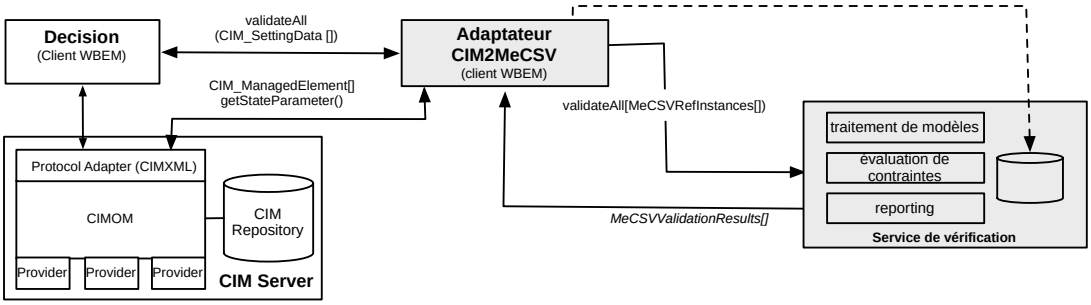

Figure 63 – Architecture de vérification intégrée de configurations CIM basée sur OpenPegasus

VII.2.2.3. Architecture du système géré cible

Nous avons expérimenté la vérification intégrée sur trois cas de configurations de systèmes virtuels variant en taille et en complexité de dépendances (instances du modèle de la Figure 44). Nous avons rajouté au `CIM repository` des instances `CIM_SettingData` et `CIM_ManagedElement` correspondantes (au format MOF du DMTF).

- Dans la première configuration (Cas 1), une machine physique héberge une machine virtuelle avec trois types de pools de ressources (mémoire RAM, espace disque, CPU) pour un total de 180 valeurs de paramètres de configuration à manipuler.
- Dans la deuxième configuration (Cas 2), la machine physique héberge deux machines virtuelles avec trois différents types de pools de ressources pour un total de 360 paramètres de configuration à manipuler.
- Dans la troisième configuration (Cas 3), nous avons 3 machines virtuelles avec 3 différents types de ressources pour un total de 800 paramètres de configuration.

VII.2.2.4. Scénario de reconfiguration et processus de vérification

– Déroulement de la vérification

Le processus de vérification s'enclenche lorsque le composant adaptateur reçoit une requête de vérification contenant la ou les instances de configuration à vérifier (instances de `CIM_SettingData`). Cette requête déclenche la récupération de valeurs de paramètres d'état nécessaires, dans le cas présent, via la consultation du `CIM Repository` d'`OpenPegasus`. Chaque requête entraîne la consultation supplémentaire de 17 instances (`CIM_ComputerSystem` et `CIM_ResourcePool`), soit 140 paramètres d'état opérationnel dont il faut récupérer les valeurs.
Une fois les instances d'informations d'état opérationnel récupérées, l'adaptateur transforme l'ensemble des instances CIM (configurations et état opérationnel) en instances conformes au modèle de référence MeCSV de la plate-forme du système virtuel et les envoie au service de vérification. Ce dernier évalue ces instances conformément aux contraintes définies dans le modèle de référence.

De plus, pour chaque demande de vérification, le service procède à une vérification avec 10, 50 et 100 contraintes, un extrait de ces contraintes a été présenté à la section V.3.2.3 du Chapitre 5. Le nombre de contraintes *online* représente 30% du nombre total de contraintes.

Le Tableau 4 récapitule les données de l'expérimentation : le nombre d'instances de classes héritant de `CIM_SettingData`, le nombre d'instances de classes héritant purement de `CIM_ManagedElement` et le nombre de paramètres de configuration et d'état opérationnel résultant traité pour chaque vérification.

Cas	*Nombre d'instances de classes configuration (CIM_SettingData)*	*Nombre de paramètres de de configuration*	*Nombre d'instances de classes d'état (CIM_ManagedElement)*	*Nombre de paramètres d'état*
Cas 1	8	180	17	140
Cas 2	16	360	17	140
Cas 3	30	800	17	140

Tableau 4 – Récapitulatif des données de l'expérimentation

– Eléments de performance

Nous avons inséré des points de mesure à chaque étape du processus de vérification afin d'évaluer le surcoût temporel de cette intégration (Figure 64).

- Nous avons mesuré le temps total mis par l'adaptateur T_{adapt}. Cette mesure comprend le temps que l'adaptateur met pour récupérer les valeurs d'état opérationnelles nécessaires (T_{etat}) et pour traduire les informations CIM en informations conformes au modèle de référence correspondant T_{conv} ($T_{adapt} = T_{etat} + T_{conv}$).

- Nous avons également mesuré le temps d'exécution de la vérification proprement dit T_{verif}. La somme du temps de l'adaptateur (T_{adapt}) et du temps de la vérification (T_{verif}) donne le temps total d'exécution de la vérification intégrée $T_{total} = T_{adapt} + T_{verif}$.

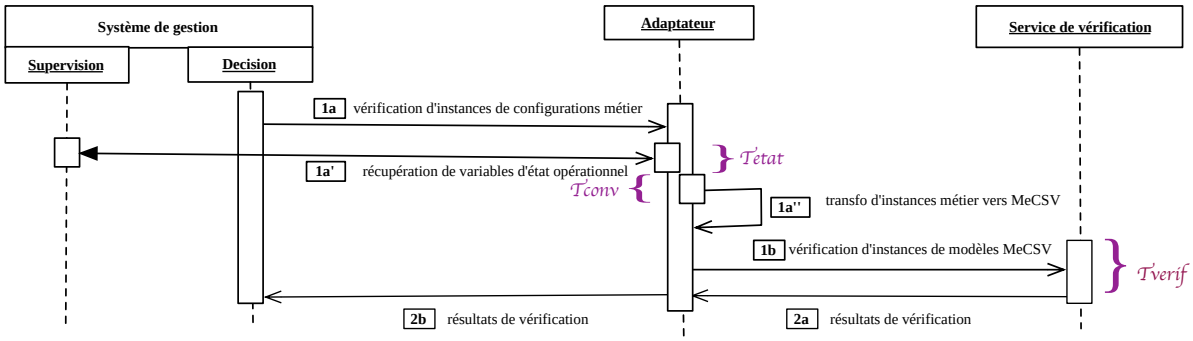

Figure 64 - Différents points de mesures du surcoût temporel

Les tests ont été exécutés sur une machine Intel® Core ™ 2 Duo de 2,66 GHz et 4 Giga-octets de mémoire RAM où s'exécutent le service de vérification et l'adaptateur `CIM2MeCSVAdapter`. La machine de test et le serveur WBEM `OpenPegasus` se trouvent dans le même réseau local, reliés par une connexion de 100 Mbits par seconde.

VII.2.2.5. Résultats et discussions

L'adaptateur CIM/WBEM a été implémenté et expérimenté pour montrer la faisabilité d'une intégration de système de gestion existant *at runtime* et évaluer le surcoût temporel de cette vérification intégrée.

Les résultats présentés dans les sections qui suivent proviennent du calcul de la moyenne arithmétique de 100 mesures du temps d'exécution de chaque demande de vérification. Dans chaque cas, le service de vérification a évalué les instances avec succès et retourné des erreurs pour les configurations invalides.

VII.2.2.5.1. Faisabilité de la vérification

Comme dans l'expérimentation du cas `JORAM`, le service de vérification a évalué les instances avec succès et retourné des erreurs pour les configurations invalides. Un exemple de rapport d'erreurs généré est illustré dans la trace d'exécution de la Figure 65. Il correspond au résultat de l'évaluation de dix contraintes sur la configuration du cas 1. On y trouve des indications sur la nature des violations constatées : le type des contraintes non respectées (*offline*, online), leur niveau de sévérité (`WARNING`, `ERROR`), l'élément de configuration affecté (ici, la configuration d'une machine virtuelle).

```
 INFO [main] -  >>>> Candidate Instance for validation : ReferenceModelRepository/MRM-cim_virtualsystem/modelinstance/vs_instance_1.mrm
 INFO [main] -  >>>> Begin validation request
 INFO [main] -  >> Validator : validation request received : validateAll...
DEBUG [main] -  >> Validator | Treatment module : loading a MeCSV reference instance...
DEBUG [main] -  >> Validator | Evaluation Module : interpreting constraints...
DEBUG [main] -  >> Validator | Reporting module : preparing result notification...
 INFO [main] -  >> Validator | Reporting module : output...
 verificationResult {
    result : false
    verificationErrors : [
      { constraintName : DiskAllocMustExist, constraintType : offline, constraintLevel : ERROR, evaluationContext : CIM_VirtualSystemSettingData  conf2vm1 }
      { constraintName : ValidMemoryAllocation, constraintType : online, constraintLevel : WARNING, evaluationContext : ReferenceModel null }
      { constraintName : DiskAllocMustExist, constraintType : offline, constraintLevel : ERROR, evaluationContext : CIM_VirtualSystemSettingData  statevm1 }
      { constraintName : MemoryAllocMustExist, constraintType : offline, constraintLevel : ERROR, evaluationContext : CIM_VirtualSystemSettingData  statevm1 }
      { constraintName : ValidTotalDiskReservation, constraintType : online, constraintLevel : WARNING, evaluationContext : ReferenceModel null }
      { constraintName : ValidTotalMemoryReservation, constraintType : online, constraintLevel : WARNING, evaluationContext : ReferenceModel null }
      { constraintName : MemoryAllocMustExist, constraintType : offline, constraintLevel : ERROR, evaluationContext : CIM_VirtualSystemSettingData  conf2vm1 }
      { constraintName : ValidDiskAllocation, constraintType : online, constraintLevel : WARNING, evaluationContext : ReferenceModel null }
    ]
 }
```

Figure 65 - vérification de configurations issues du cas 1 avec 10 contraintes

Ce résultat confirme l'applicabilité de notre approche notamment la capacité du prototype du service de vérification, à supporter des vérifications de contraintes *offline* et *online*.

VII.2.2.5.2. Mesures du surcoût temporel de l'intégration opérationnelle

La Figure 66 montre les résultats de la phase d'intégration de la plate-forme de gestion basée sur `OpenPegasus` avec le service de vérification, soit la phase de récupération d'instances d'état opérationnel T_{etat} et la traduction des instances de gestion CIM en instances conformes au modèle de référence MeCSV, T_{conv}.

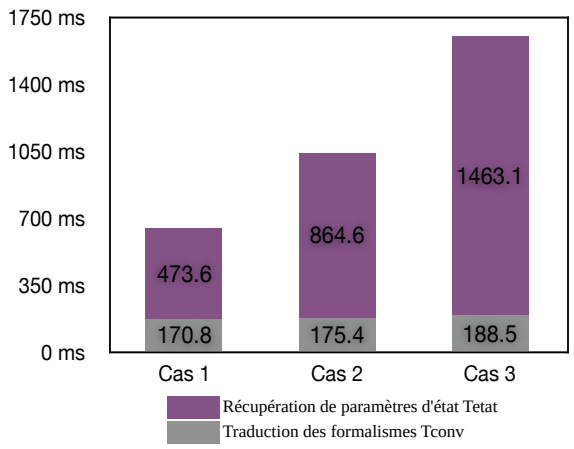

Figure 66 - Surcoût temporel T_{adapt} introduit par le composant adaptateur

Ces résultats soulignent le surcoût important introduit par la récupération de paramètres d'état opérationnel au niveau d'`OpenPegasus`. Le temps de traduction des formalismes par l'adaptateur T_{conv} est presque constant et négligeable dans l'ensemble, il est inférieur à 200 millisecondes. Le temps de récupération des informations d'état T_{etat} est le contributeur essentiel à ce surcoût, il représente en moyenne 81.74% de T_{adapt}.

Ce surcoût important est dû à l'implémentation du serveur `OpenPegasus` lui-même. Il couvre le temps de connexion au serveur WBEM, la manipulation des modèles CIM et les différentes requêtes de récupération des instances auprès du `CIM Repository`.

Pour les deux derniers points, ce temps est surtout impacté par la navigation des instances d'associations CIM. Les associations CIM sont en réalité des classes-associations possédant au minimum deux attributs, respectivement du type de la classe CIM source et de la classe CIM destination. Elles sont instanciées au même titre que les classes CIM. Ainsi pour déterminer les relations courantes entre deux instances de classes CIM, il faut identifier les instances d'association auxquelles elles participent.

Il ne suffit donc pas de récupérer les instances de `CIM_ManagedElement` qui permettent de remonter les conditions opérationnelles courantes, la consultation des associations entre éléments `CIM_ManagedElement` et `CIM_SettingData` est également nécessaire afin d'associer correctement les instances de configuration aux instances d'état opérationnel.

Par ailleurs, les valeurs des métadonnées qualifiant le type de configuration CIM (courant, par défaut, candidat) sont précisées au niveau d'instances d'associations CIM (`CIM_ElementSettingData`). Leur transformation nécessite donc la consultation supplémentaire d'instances de `CIM_Association`.

Nous avons remarqué que T_{etat} croit en fonction de la taille et de la configuration considérée (nombre de classes et associations entre classes). Par exemple, T_{etat} triple entre le cas 1 (8 instances de configuration) et le cas 3 (30 instances de configurations) par rapport à un nombre d'instances d'état opérationnel constant (17 instances).

VII.2.2.5.3. Mesures du surcoût temporel de la vérification

Comme le montrent les résultats de la phase de vérification (Figure 67), le temps d'exécution croît avec l'évolution du nombre d'éléments de configuration et de contraintes. Toutefois, ce temps ne croît pas proportionnellement, il suit une évolution linéaire.

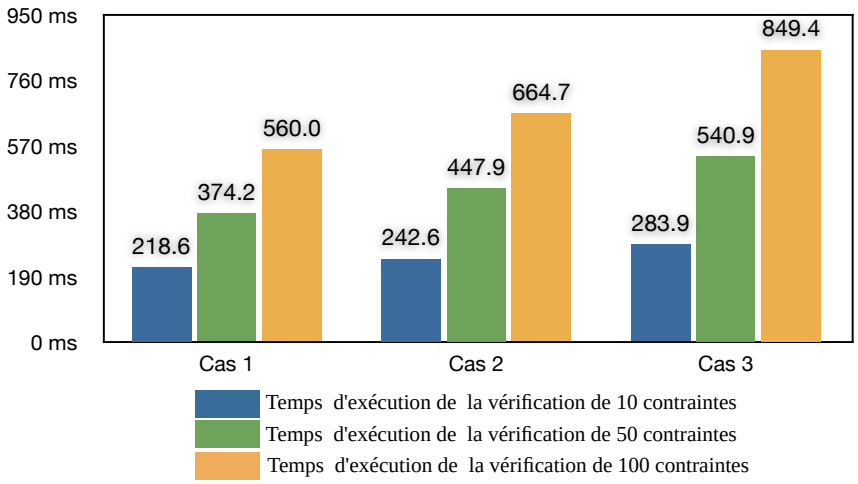

Figure 67 - Mesures du temps d'exécution de la vérification (T_{verif})

En effet alors que le ratio entre le cas 1 et le cas 2 est de 2 et le ratio entre le cas 1 et le cas 3 est de 4.4, les temps d'évaluation de contraintes correspondants sont respectivement 1.3 et 1.8. De la même façon, lorsqu'on passe de l'évaluation de 10 contraintes à l'évaluation de 100 contraintes, soit un ratio de 10, le temps d'exécution ne progresse que 2.8. Ces résultats confirment les précédents obtenus sur la vérification de configurations de la plate-forme `JORAM`.

Le Figure 68 résume, pour chaque jeu de contraintes, la répartition en pourcentage du temps d'exécution de l'adaptateur T_{adapt} par rapport au temps d'exécution de la vérification T_{verif}.
Le temps de l'adaptateur T_{adapt} occupe 54% à plus de 85% du surcoût total de la vérification.

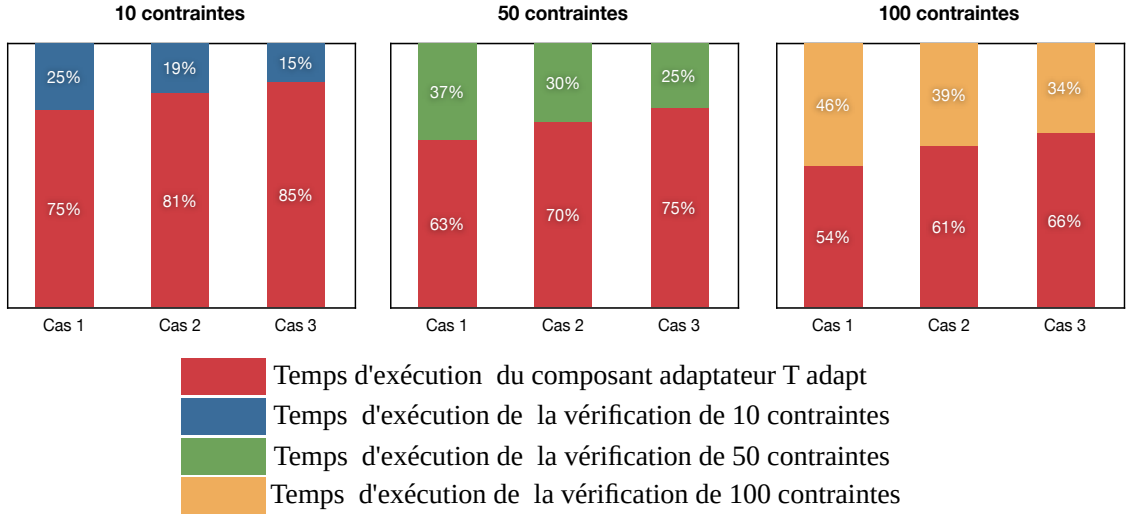

Figure 68 - Répartition du T_{adapt} et de T_{verif} par rapport au temps total T_{total}

VII.2.2.5.4. Discussion des résultats

L'objectif de cette expérimentation était de tester la faisabilité d'une vérification intégrée de configurations via l'adaptateur `CIM2MeCSVAdapter` et de mesurer le surcoût généré. Les résultats obtenus ont montré la faisabilité d'une intégration d'un environnement CIM/WBEM basé sur le serveur `OpenPegasus`. Le module de décision existant a pu solliciter la vérification d'instances de classes `CIM_VirtualSettingData` et `CIM_ResourceAllocationSettingData` et les configurations invalides ont donné lieu à des levées d'erreurs.

Ce cas d'expérimentation montre qu'il est possible d'implémenter un adaptateur pour un environnement de gestion donné à partir de l'adaptateur de base que nous avons conçu.

Les divers choix de conception que nous avons retenus (extension de l'adaptateur de base, programmation de transformations entre patron `CIM_SettingData` et MeCSV, client WBEM via l'API SBLIM) ont réduit l'effort de développement de l'adaptateur `CIM2MeCSVAdapter` : sa version actuelle consiste en 192 lignes de code. De plus, ce prototype est implémenté une fois pour toute, c'est-à-dire qu'il peut être réutilisé pour tout modèle CIM conforme au patron `CIM_SettingData` et pour toute implémentation de serveur WBEM donné. Tout environnement de gestion basé sur CIM/WBEM peut être ainsi intégré au service de vérification.

La vérification en ligne a généré un surcoût temporel important si l'on considère les résultats obtenus dans le cas d'expérimentation `JORAM`. Ce surcoût est essentiellement dû au temps mis par l'adaptateur pour récupérer les informations d'état supplémentaires auprès du `CIM Repository`. Ce temps est tributaire de l'implémentation du serveur `OpenPegasus`, notamment la façon dont il traite les associations CIM.

VII.2.3. Pilotage de la vérification *at runtime*

Le pilotage de la vérification représente une exigence essentielle de la vérification de configurations dans le contexte de la reconfiguration dynamique. Nous avons également procédé à des tests de faisabilité d'une vérification flexible de configurations et d'une vérification évolutive prenant en compte l'édition de contraintes *at runtime*.

VII.2.3.1. Rappel du résultat de vérification complète du cas 3 de JORAM

La Figure 69 rappelle le rapport d'erreurs généré par la vérification complète (`validateAll`) de dix contraintes sur le troisième cas de configuration candidate de `JORAM` (présenté à la section VII.2.1.2). Cette configuration va être réutilisée, telle quelle, dans les sections suivantes pour présenter les résultats de diverses séries de vérification sélective et évolutive.

```
INFO [main] -  >>>> Registering the MeCSV Reference Model
INFO [main] -  >>>> structure :ReferenceModelRepository/MRM-joram/model/JoramMRM.ecore
INFO [main] -  >>>> constraints : ReferenceModelRepository/MRM-joram/constraints/JoramMRM_10.ocl
INFO [main] -  >> Validator : Registering the MeCSV reference model structure...
INFO [main] -  >> Validator : Registering the MeCSV reference model constraint...
INFO [main] -  >> Validator : service initialized...
INFO [main] -  >>>> Candidate Instance for validation : ReferenceModelRepository/MRM-joram/modelinstance/sp4_instance_3.mrm
INFO [main] -  >>>> Begin validation request
INFO [main] -  >> Validator : validation request received : validateAll...
INFO [main] -  >> Validator | Reporting module : output...
verificationResult {
   result : false
   verificationErrors : [
      { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s1 }
      { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s0 }
      { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : WARNING, evaluationContext : Queue  q0 }
      { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : WARNING, evaluationContext : Queue  q1 }
      { constraintName : MustBeRegisteredConnectionFactory, constraintType : offline, constraintLevel : ERROR, evaluationContext : ConnectionFactory null }
      { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s2 }
   ]
}
INFO [main] -  >>>> End validation request
```

Figure 69 - Rappel de la vérification de 10 contraintes sur le cas 3 JORAM

VII.2.3.2. Vérification flexible de configurations

Nous avons évalué la capacité de la vérification à être modulée, par exemple, pouvoir solliciter une vérification partielle. Nous avons ainsi sollicité des séries de vérification sélective filtrées par type de contrainte (*offline* ou *online*), par niveau de sévérité ou par combinaison de ces deux critères (appel des opérations `validateByConstraintType`, `validateByConstraintSeverity` et `validateByConstraintFeatures` présentées dans le Chapitre 6).

– Vérification partielle de contraintes *online* seules

La Figure 70 montre le rapport d'erreurs résultant d'une vérification partielle de contraintes *online* seules sur la configuration du cas 3. A la différence des résultats issus de la vérification complète, on peut observer que seules les violations de contraintes online ont été notifiées.

```
INFO [main] -  >>>> Candidate Instance for validation : ReferenceModelRepository/MRM-joram/modelinstance/sp4_instance_3.mrm
INFO [main] -  >>>> Begin validation request
INFO [main] -  >> Validator : validation request received : validateByConstraintType (online = true)...
INFO [main] -  >> Validator | Reporting module : output...
verificationResult {
   result : false
   verificationErrors : [
      { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : WARNING, evaluationContext : Queue  q0 }
      { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : WARNING, evaluationContext : Queue  q1 }
   ]
}
INFO [main] -  >>>> End validation request
```

Figure 70 - Vérification partielle de contraintes *online* seules

Dans le cas présent, la contrainte *online* (`OperationalApplicableNbMsgMax`) n'est pas respectée par les configurations des files de messages `q0` et `q1`. Cette contrainte garantit que la mémoire d'une file de message n'est pas dans un état dégradé, c'est-à-dire que sa capacité maximale de stockage de messages est suffisamment supérieure à son nombre de messages courants stockés. Le non-respect de ce type de contrainte qui n'est vérifiable qu'en ligne peut provoquer des comportements indésirables, tels que l'envoi de messages en attente dans la file DMQ, la terminaison de connexions d'applications clientes en cours.

Ce résultat prouve la faisabilité d'une vérification partielle de contraintes *online* seules, notamment la capacité du prototype du service de vérification à supporter une vérification partielle de configurations filtrée sur le type de contrainte.

– **Vérification partielle selon un niveau de sévérité (ERROR)**

Le module de décision sollicite une vérification sélective de la configuration du cas 3 selon le niveau de sévérité `ERROR`. Le rapport d'erreurs généré est illustré dans la Figure 71. On peut remarquer que seules les violations de contraintes de niveau de sévérité `ERROR` ont été rapportées. Il n'y a qu'une seule violation qui répond à ce critère, celle de la contrainte `MustBeRegisteredConnectionFactory` de sévérité `ERROR`. Cette contrainte vérifie qu'une usine de connexion est enregistrée dans le service de noms. Cet enregistrement permet normalement aux applications clientes de rechercher les usines de connexion dans le service de noms afin de pouvoir se connecter aux serveurs de messages.

```
INFO [main] -  >>>> Candidate Instance for validation : ReferenceModelRepository/MRM-joram/modelinstance/sp4_instance_3.mrm
INFO [main] -  >>>> Begin validation request
INFO [main] -  >> Validator : validation request received : validateByConstraintSeverity (severity = ERROR)...
INFO [main] -  >> Validator | Reporting module : output...
verificationResult {
   result : false
   verificationErrors : [
      { constraintName : MustBeRegisteredConnectionFactory, constraintType : offline, constraintLevel : ERROR, evaluationContext : ConnectionFactory null }
   ]
}
INFO [main] -  >>>> End validation request
```

Figure 71 - Vérification partielle de contraintes par niveau de sévérité

Ce résultat prouve la faisabilité d'une vérification partielle selon le niveau de sévérité des contraintes, en l'occurrence la capacité du prototype du service de vérification à supporter une vérification partielle de configurations filtrée sur le niveau de sévérité de la contrainte.

– **Vérification partielle de contraintes *offline* seules du plus haut niveau de sévérité (FATAL)**

Dans ce test, le module de décision sollicite une vérification sélective de la configuration du cas 3 selon le type de contraintes (*offline*) et le niveau de sévérité `FATAL`. La Figure 72 présente le rapport d'erreurs produit.

```
INFO [main] -  >>>> Candidate Instance for validation : ReferenceModelRepository/MRM-joram/modelinstance/sp4_instance_3.mrm
INFO [main] -  >>>> Begin validation request
INFO [main] -  >> ValidationService : validation request received : validateByConstraintFeatures (online =  false, severity = FATAL)...
INFO [main] -  >> Validator | Reporting module : output...
verificationResult {
   result : false
   verificationErrors : [
      { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s2 }
      { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s0 }
      { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s1 }
   ]
}
INFO [main] -  >>>> End validation request
```

Figure 72 - Vérification sélective combinée par type de contrainte et par niveau de sévérité

On peut constater que seules les violations de contraintes *offline* et de niveau de sévérité `FATAL` ont été remontées. C'est le cas de la contrainte `UniqueServerIds` qui n'est pas respectée par les trois serveurs de messages de la plateforme. Cette contrainte contrôle l'unicité des identifiants de serveurs. L'unicité des identifiants de serveurs est nécessaire au fonctionnement d'une plate-forme `JORAM`. Sa violation entraîne une rupture de service.

Ce résultat montre la faisabilité d'une vérification partielle selon le type de contraintes et le niveau de sévérité, en l'occurrence la capacité du service à favoriser une vérification partielle de configurations selon le type de contrainte et le niveau de sévérité.

VII.2.3.3. Vérification évolutive de configurations

Nous avons également expérimenté l'édition de contraintes en cours d'exécution et la capacité du service à prendre en compte ces modifications. Dans la version actuelle du prototype, l'édition de contraintes consiste à modifier directement le fichier de contraintes OCL du modèle de référence correspondant et à solliciter le rechargement du fichier concerné.

– Ajout de contraintes

Nous avons rajouté la contrainte *offline* suivante à la liste des contraintes du modèle de référence JORAM (Figure 73). Cette contrainte permet de vérifier que la capacité maximale d'une file de message est supérieure à 10000 messages.

AJOUT - CONTRAINTES

```
-- La capacité maximale d'une file de message doit être supérieure à 10000 messages
context Queue
inv NbMaxMsgMustBeGreaterThan10000_offline_active_error: self.nbMaxMsg >= 10000
```

Figure 73 - Ajout d'une nouvelle contrainte

Le lancement d'une vérification complète de la configuration (cas 3) après ajout de cette contrainte produit le résultat illustré dans la Figure 74. On peut constater que la contrainte (contrainte `NbMaxMsgMustBeGreaterThan10000`) a été bien prise en compte. De plus, elle n'est pas respectée par les files de message de la plate-forme.

```
INFO [main] -  >>>> Candidate Instance for validation : ReferenceModelRepository/MRM-joram/modelinstance/sp4_instance_3.mrm
INFO [main] -  >>>> Begin validation request
INFO [main] -  >> Validator : validation request received : validateAll...
INFO [main] -  >> Validator | Reporting module : output...
verificationResult {
   result : false
   verificationErrors : [
     { constraintName : NbMaxMsgMustBeGreaterThan10000, constraintType : offline, constraintLevel : ERROR, evaluationContext : Queue  q2 }
     { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s0 }
     { constraintName : NbMaxMsgMustBeGreaterThan10000, constraintType : offline, constraintLevel : ERROR, evaluationContext : Queue  q0 }
     { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : WARNING, evaluationContext : Queue  q1 }
     { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s2 }
     { constraintName : MustBeRegisteredConnectionFactory, constraintType : offline, constraintLevel : ERROR, evaluationContext : ConnectionFactory null }
     { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s1 }
     { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : WARNING, evaluationContext : Queue  q0 }
     { constraintName : NbMaxMsgMustBeGreaterThan10000, constraintType : offline, constraintLevel : ERROR, evaluationContext : Queue  q1 }
   ]
}
INFO [main] -  >>>> End validation request
```

Figure 74 - Prise en compte de l'ajout de contraintes

Ce résultat montre la faisabilité d'un ajout de contraintes en cours d'exécution du service et sa capacité à prendre en compte la nouvelle contrainte ajoutée de façon immédiate, sans recodage ou recompilation du prototype ou redéfinition complète du modèle de référence MeCSV.

– Modification de niveau de sévérité

Le niveau de sévérité de la contrainte `OperationalApplicableNbMaxMsg` (qui prévient le passage de la mémoire file dans un état dégradé) a été élevé à un niveau critique (`CRITICAL`). Cette modulation de la sévérité peut faire partie d'un scénario d'optimisation de la performance des échanges où il devient critique de renforcer cette contrainte afin de prévenir sa violation (Figure 75).

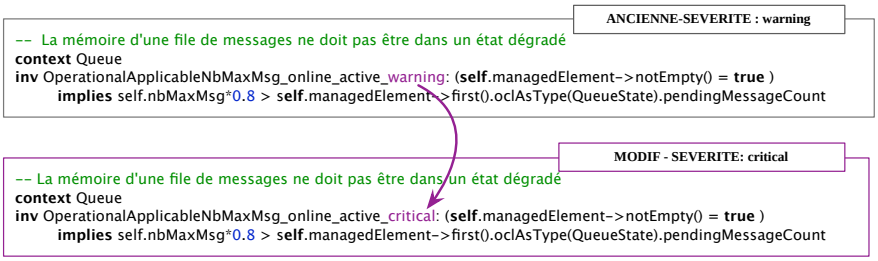

Figure 75 – Modification du niveau de sévérité d'une contrainte existante

La Figure 76 illustre le résultat de la vérification complète de la même configuration après cette modification du niveau de sévérité. Ce changement de sévérité est reflété dans le rapport d'erreurs généré.

Ce résultat met en évidence la faisabilité de la modification en ligne du niveau de sévérité d'une contrainte existante et la capacité du service de vérification à la supporter et la à prendre en compte en cours d'exécution.

```
INFO [main] -  >>>> Candidate Instance for validation : ReferenceModelRepository/MRM-joram/modelinstance/sp4_instance_3.mrm
INFO [main] -  >>>> Begin validation request
INFO [main] -  >> Validator : validation request received : validateAll...
INFO [main] -  >> Validator | Reporting module : output...
verificationResult {
   result : false
   verificationErrors : [
     { constraintName : NbMaxMsgMustBeGreaterThan10000, constraintType : offline, constraintLevel : ERROR, evaluationContext : Queue  q1 }
     { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : CRITICAL, evaluationContext : Queue  q0 }
     { constraintName : NbMaxMsgMustBeGreaterThan10000, constraintType : offline, constraintLevel : ERROR, evaluationContext : Queue  q0 }
     { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : CRITICAL, evaluationContext : Queue  q1 }
     { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s0 }
     { constraintName : MustBeRegisteredConnectionFactory, constraintType : offline, constraintLevel : ERROR, evaluationContext : ConnectionFactory null }
     { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s2 }
     { constraintName : UniqueServerIds, constraintType : offline, constraintLevel : FATAL, evaluationContext : Server  s1 }
     { constraintName : NbMaxMsgMustBeGreaterThan10000, constraintType : offline, constraintLevel : ERROR, evaluationContext : Queue  q2 }
   ]
}
INFO [main] -  >>>> End validation request
```

Figure 76 - Prise en compte du nouveau niveau de sévérité

– Désactivation de contraintes

Comme le montre la Figure 77, la contrainte `UniqueServerIds` été désactivée. Cette contrainte permettait de s'assurer que tous les serveurs possédaient un identifiant unique. Lorsque le module de décision requiert une vérification complète de la même configuration, le rapport d'erreurs produit, Figure 78, montre que la violation de cette contrainte n'a plus été remontée, elle n'a plus été évaluée.

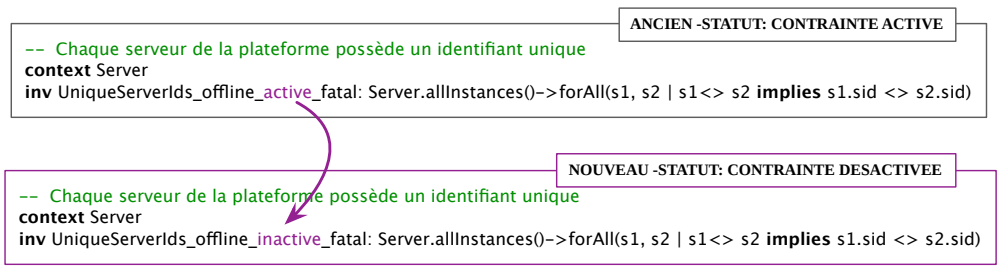

Figure 77 – Désactivation d'une contrainte

Ce résultat montre la faisabilité de la désactivation d'une contrainte existante et la capacité du service de vérification à supporter et à prendre en compte cette désactivation.

```
INFO [main] -  >>>> Candidate Instance for validation : ReferenceModelRepository/MRM-joram/modelinstance/sp4_instance_3.mrm
INFO [main] -  >>>> Begin validation request
INFO [main] -  >> Validator : validation request received : validateAll...
INFO [main] -  >> Validator | Reporting module : output...
verificationResult {
   result : false
   verificationErrors : [
     { constraintName : NbMaxMsgMustBeGreaterThan10000, constraintType : offline, constraintLevel : ERROR, evaluationContext : Queue  q1 }
     { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : CRITICAL, evaluationContext : Queue  q1 }
     { constraintName : MustBeRegisteredConnectionFactory, constraintType : offline, constraintLevel : ERROR, evaluationContext : ConnectionFactory null }
     { constraintName : NbMaxMsgMustBeGreaterThan10000, constraintType : offline, constraintLevel : ERROR, evaluationContext : Queue  q0 }
     { constraintName : OperationalApplicableNbMaxMsg, constraintType : online, constraintLevel : CRITICAL, evaluationContext : Queue  q0 }
     { constraintName : NbMaxMsgMustBeGreaterThan10000, constraintType : offline, constraintLevel : ERROR, evaluationContext : Queue  q2 }
   ]
}
INFO [main] -  >>>> End validation request
```

Figure 78 - Prise en compte de la désactivation de contraintes par le service de vérification

VII.2.3.4. Discussion des résultats

L'objectif de ces expérimentations était d'évaluer la faisabilité d'un pilotage de la vérification *at runtime* à savoir la possibilité de solliciter une vérification partielle de configurations et la capacité du service de vérification à prendre en compte les modifications des éléments de modèles de référence MeCSV. Les résultats obtenus montrent la capacité du prototype à supporter une vérification en ligne, sélective de configurations et à prendre en compte les différentes modifications de contraintes (ajout, désactivation, modification de sévérité).

Ces expérimentations montrent également qu'à partir du métamodèle MeCSV (via la richesse de ses constructeurs de contraintes) et du service de vérification, il est possible de mettre en œuvre une vérification en ligne de configurations dont les contraintes peuvent être évoluées en cours d'exécution, ajoutées, renforcées ou relâchées. Cette flexibilité et évolutivité peuvent permettre de personnaliser la vérification souhaitée et de l'adapter aux objectifs et scénarii d'utilisation *at runtime*.

VII.3. Conclusion

Dans ce chapitre, nous avons présenté des aspects validation du service de vérification proposé à travers la réalisation d'un prototype du service et de ses composants, puis des expérimentations de ce prototype dans deux contextes applicatifs différents : la gestion d'un *middleware* orienté messages appelée JORAM dans un environnement JMX, la gestion de machines virtuelles dans un environnement CIM/WBEM.

Les résultats obtenus ont montré l'applicabilité du prototype dans les deux cas d'expérimentation. Les différents scénarii de tests ont illustré :

- la capacité du service de vérification à être sollicité par différents modules de décision et procéder à une vérification en ligne de configurations vis-à-vis de valeurs opérationnelles courantes remontées par un module de supervision existant,
- sa capacité à être intégré à un environnement de gestion existant grâce au développement d'un composant adaptateur,
- sa capacité à supporter des invocations de vérifications sélectives ainsi que prendre en compte des modifications de contraintes *at runtime*.

Ce dernier chapitre vient confirmer l'applicabilité du cadriciel que nous proposons en phase d'exécution, notamment sa capacité à supporter une vérification opérationnelle de configurations flexible et évolutive pouvant prendre en compte l'existant.

Conclusion générale

Rappel de la problématique

Disposer *at runtime* de solutions simples et intégrées de vérification et de validation est une condition essentielle si nous voulons construire des systèmes de gestion autonomes, fiables et sûrs. Les travaux de cette thèse s'inscrivaient dans ce contexte, ils visaient la construction d'un cadriciel de vérification en ligne de configurations pour les systèmes de gestion autonomes et adaptatifs.

A partir de l'état de l'art sur la reconfiguration dynamique, les erreurs de configuration et les travaux existants de vérification de configurations, nous avons identifié les propriétés clés d'une solution idéale de vérification de configurations :

- face aux caractéristiques de la reconfiguration dynamique, la solution devait supporter une vérification en ligne tenant compte de l'état courant du système géré,
- face à la dynamicité du système géré, la solution devait permettre une vérification flexible et évolutive,
- face à l'hétérogénéité des environnements de gestion, la solution devait être non seulement indépendante des plates-formes et des protocoles de gestion, mais elle devait également favoriser la prise en compte de l'existant.

Ces trois propriétés nous ont conduit à la définition d'un cadriciel dédié. Nous nous sommes inspirés de l'approche de l'Ingénierie Dirigée par les Modèles (IDM) et avons utilisé les modèles comme bases de notre raisonnement.

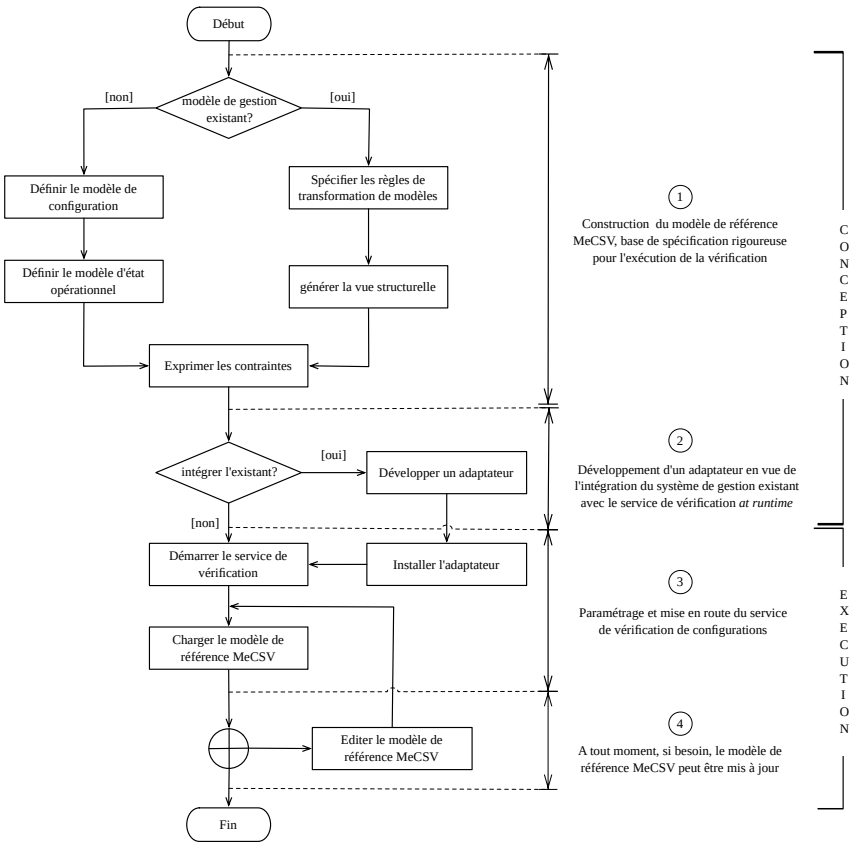

Figure 79 - Processus d'utilisation du cadriciel

Synthèse des contributions

La Figure 86 décrit le processus d'utilisation de notre cadriciel pour un système de gestion donné :

1. **modélisation de configurations et de contraintes de haut niveau** : comme le montre la Figure 86 – ①, il s'agit d'abord de constituer une spécification rigoureuse des configurations des environnements gérés et des contraintes qui doivent être respectées. Nous avons défini à cet effet le métamodèle MeCSV qui a été présenté dans le Chapitre 4. MeCSV propose des concepts constructeurs de haut niveau permettant de représenter un modèle d'information de configuration, un modèle d'information d'état opérationnel et un modèle de contraintes *offline* et *online* qui doivent être vérifiées. L'ensemble de ces modèles constitue le modèle de référence MeCSV pour un domaine applicatif donné. Une méthodologie de production outillée de modèles de référence MeCSV a été développée dans le Chapitre 5 et éprouvée dans la construction de modèles de référence avec et sans considération de modèles de gestion existants.

2. **développement d'un composant d'intégration de l'existant** : il est également possible d'implémenter, si besoin, un adaptateur (Figure 86 – ②), composant d'intégration qui servira de passerelle *at runtime*, entre le système de gestion et notre service de vérification. Nous avons défini, dans le Chapitre 6, l'architecture d'un tel composant. De plus, nous avons proposé dans le Chapitre 7, le prototype d'un adaptateur de base en Java, à étendre pour l'implémentation d'adaptateurs spécifiques. Cet adaptateur de base a par ailleurs servi à l'implémentation d'un adaptateur pour un environnement de gestion basé sur les standards CIM/WBEM du DMTF. L'adaptateur fait partie de l'architecture complète d'un service de vérification en ligne que nous avons aussi conçu.

3. **utilisation du service de vérification en ligne** : nous avons défini un service de vérification qui exploite les modèles de référence MeCSV pour vérifier les configurations qui lui sont soumises. L'architecture de ce service est présentée dans le Chapitre 6. Il est constitué de deux composants internes, supports de la vérification : un moteur de vérification et une base de modèles. Sa mise en route nécessite une étape de paramétrage qui consiste à charger les modèles de référence MeCSV dans la base de modèles (Figure 86 – ③). Il peut dès lors être sollicité par un système de gestion pour la vérification de configurations.

Le service de vérification a été mis en œuvre suivant deux plans : un plan opérationnel qui en permet l'utilisation et un plan de contrôle qui permet la gestion du cycle de vie des modèles de référence MeCSV en cours d'exécution.

– Le plan opérationnel est porté par une interface d'opérations permettant de solliciter une vérification selon plusieurs niveaux de granularités, ce qui permet une vérification de configurations souple et modulables au plus près des scénarii d'usage. Un système de gestion «client» peut invoquer ces opérations directement via l'interface du service ou via l'adaptateur implémenté pour le système de gestion (voir 2). Cette possibilité permet une vérification permettant la prise en compte des systèmes de gestion existants.

– Le plan de contrôle est supporté par une interface d'opérations favorisant la mise à jour des éléments de modèles de référence à tout moment de l'utilisation du service de vérification (Figure 86 – ④). Cette capacité permet une vérification tenant compte de l'évolution des environnements gérés et des objectifs de gestion.

Le service de vérification a fait l'objet d'un prototype, que nous avons détaillé dans le Chapitre 7. Ce prototype a été validé sur divers cas d'expérimentations issus de deux contextes applicatifs différents : la gestion d'intergiciels orientés messages en environnement JMX et la gestion de systèmes virtuels en environnements CIM/WBEM.

Les résultats de l'ensemble des expérimentations ont confirmé la faisabilité de notre approche et ont montré que le prototype défini permet de couvrir les nouvelles exigences de la vérification en ligne de configurations en situation de reconfiguration dynamique.

D'un point de vue utilisation du cadriciel, une fois le modèle de référence crée (1) et le composant d'intégration développé (2), le service de vérification peut être utilisé par le système de gestion pour vérifier toute nouvelle configuration sans effort supplémentaire.

Perspectives

Les travaux réalisés dans cette thèse ouvrent deux grands axes de perspectives : un axe à court terme qui concerne l'évolution du prototype développé et la consolidation des validations expérimentales et un axe à plus long terme relatif à la dérivation automatique des contraintes et l'extension du cadriciel à la vérification du déploiement des configurations candidates.

– Evolution du prototype

Dans la version actuelle du prototype, nous avons privilégié l'intégration d'outils provenant de l'IDM. Il serait intéressant d'approfondir cette intégration au niveau de certaines briques du prototype. Par exemple, l'utilisation d'une véritable base de modèles (*model repositories*) telles que `EMFStore` ou encore `CDO` pour le composant «base de modèles de référence MeCSV» du service permettrait de bénéficier d'une gestion du cycle de vie des modèles davantage évoluée. En outre, ces bases de modèles offrent des possibilités de gestion automatique de versions et de droits d'accès qui sont importants pour l'outil que nous proposons.

Une autre perspective consisterait à étendre le service de vérification lui-même. Il serait profitable, par exemple, de lui rajouter une interface d'abonnement aux paramètres d'état du système géré. Les contraintes *online* pourraient ainsi être exploitées à des fins de *monitoring* de configurations courantes pour renforcer la détection de leur éventuelle invalidation.

– Poursuite du plan de validation et passage à l'échelle

Des tests d'intégration de nouveaux domaines applicatifs et de cas d'expérimentations plus complexes permettraient de poursuivre le plan de validation. Notamment approfondir les tests de passage à l'échelle du service. Ces validations sont à enrichir d'aspects méthodes exploitant les avantages de la flexibilité et de l'évolutivité de la vérification apportée.

– Dérivation automatique de contraintes

La spécification de contraintes est un élément primordial pour l'utilisation de notre cadriciel. Aider les utilisateurs dans cette tâche et la rendre plus facile est un besoin clé pour l'exploitation effective du cadriciel. A la lumière des travaux sur l'ingénierie des exigences dans les systèmes auto-adaptatifs [Cheng et al., 2007 ; Whittle et al., 2008], une perspective intéressante serait la déduction automatique de contraintes des exigences ou des contrats de service, de la documentation du système géré voire de l'historique des violations.

– Vérification d'actions de reconfigurations

La reconfiguration dynamique se fait en deux temps : la décision de la nouvelle configuration, plus l'application de celle-ci. La contribution de nos travaux s'est inscrite dans la vérification de configurations candidates et non dans la vérification du processus de leur application, par exemple, la bonne orchestration des actions de reconfiguration, la validité de leur composition. Un travail essentiel, dans la lignée de cette thèse, réside dans l'étude des caractéristiques des processus de reconfiguration dynamique afin d'en proposer la vérification.

Bibliographie

[Abrial, 1996] Abrial, J.-R. (1996). The B-Book: assigning programs to meanings. 1996. Cambridge University.

[ActiveMQ] ActiveMQ, November, 2013. [Online]. Available : http://activemq.apache.org/.

[Agrawal et al., 2004] Agrawal, D., Giles, J., and Lee, K. W., Voruganti, K., & Filali-Adib, K.

(2004). Policy-based validation of SAN configuration. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on* (pp. 77-86).

[Agrawal et al., 2005] Agrawal, D., Lee, K.-W., and Lobo, J. (2005). Policy-based management of networked computing systems. *Communications Magazine*, IEEE, 43(10), pp 69–75.

[Anderson and Scobie, 2002] Anderson, P. and Scobie, A. (2002). LCFG - the Next Generation. In *UKUUG Winter Conference*. UKUUG (pp. 4-7).

[Anderson and Smith, 2005] Anderson, P. and Smith, E. (2005). Configuration tools: Working together. In *LISA*, pp 31–37.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-c. J., Randell, B., Landwehr, C., and Member, S. (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, *1*(1), pp 11–33.

[Bahati and Bauer, 2008] Bahati, R. M. and Bauer, M. a. (2008). Adapting to Run-Time Changes in Policies Driving Autonomic Management. *4th International Conference on Autonomic and Autonomous Systems (ICAS'08)*, pp 88–93.

[Boehm, 1984] Boehm, B. W. (1984). Verifying and validating software requirements and design specifications. *Software, IEEE, 1*(1), *pp* 75–88.

[Boutaba and Aib, 2007] Boutaba, R. and Aib, I. (2007). Policy-based management: A historical perspective. *Journal of Network and Systems Management*, *15*(4),pp 447-480.

[Buchmann, 2008] Buchmann, D. (2008). Verified Network Configuration: Improving Network Reliability. PhD thesis, Universitat Freiburg (Schweiz).

[Burgess and Couch, 2006] Burgess, M. and Couch, A. L. (2006). Modeling Next Generation Configuration Management Tools. In LISA, pages 131–147.

[Cheng et al., 2009] Cheng, B. H., De Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., ... & Whittle, J. (2009). Software engineering for self-adaptive systems: A research roadmap. In *Software engineering for self-adaptive systems* (pp. 1-26). Springer Berlin Heidelberg.

[Cheng and Atlee, 2007] Cheng, B. H. C. and Atlee, J. M. (2007). Research directions in requirements engineering. In *2007 Future of Software Engineering*, pp 285–303. IEEE Computer Society.

[CIM, 2011] CIM Schema version 2.29.1 - CIM Core. (2011). Distributed Management Task Force (DMTF).

[CIM-V3] Common Information Model (CIM) Metamodel. (2012). Distributed Management Task Force (DMTF).

[CIM-WP, 2000] Common Information Model (CIM) Core Model White Paper. (2000). Distributed Management Task Force (DMTF).

[Clemm, 2006] Clemm, A. (2006). Network management fundamentals. Cisco Press.

[Combemale, 2008] Combemale, B. (2008). Ingénierie Dirigée par les Modèles (IDM)--État de l'art. pp 1–19.

[Cons and Poznański, 2002] Cons, L. and Poznański, P. (2002). Pan: A high-level configuration language. In *LISA* (Vol. 2, pp. 83-98).

[Delaet et al., 2008] Delaet, T., Anderson, P., and Joosen, W. (2008). Managing Real-World System Configurations with Constraints. *7th International Conference on Networking (ICN 2008)*, pp 594-601.

[Delaet and Joosen, 2007] Delaet, T. and Joosen, W. (2007). PoDIM: A Language for High-Level Configuration Management. In *LISA* (Vol. 7, pp. 1-13).

[Derbel et al., 2009] Derbel, H., Agoulmine, N., and Salaün, M. (2009). ANEMA: Autonomic network management architecture to support self-configuration and self-optimization in IP networks. *Computer Networks, 53*(3), pp 418–430.

[Desertot, 2007] Desertot, M. (2007). Une architecture flexible et adaptable pour les serveurs d'applications. PhD thesis, Université Joseph Fourier de Grenoble.

[Diaw et al., 2010] Diaw, S., Lbath, R., and Coulette, B. (2010). Etat de l'art sur le développement logiciel dirigé par les modèles. *Technique et Science Informatiques*, *29*(4-5), 505-536.

[DMTF-UMLProfile, 2009] UML Profile for CIM. (2009). Distributed Management Task Force (DMTF).

[DMTF-VirtualSystemProfile, 2010] Virtual System Profile (DSP 1057 version 1.0.0). (2010). Distributed Management Task Force (DMTF).

[Dresden-OCL] Dresden OCL. TU Dresden, Software Technology Group, November, 2013. [Online]. Available : http://www.dresden-ocl.org/.

[Eclipse-MDT] Eclipse Modeling - MDT, November, 2013. [Online]. Available : http://www.eclipse.org/modeling/mdt/.

[Eclipse-QVTo] QVT Operational, November, 2013. [Online]. Available : http://projects.eclipse.org/projects/modeling.

[Festor and Andrey, 2003] Festor, O. and Andrey, L. (2003). Chapitre 6: JMX: un standard pour la gestion Java. *Standards pour la gestion des réseaux et des services*, pp 213–251.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Design patterns: Elements of reusable object-oriented design. Addison-Wesley Reading.

[Gençay et al., 2008] Gençay, E., Sinz, C., Küchlin, W., and Schäfer, T. (2008). SANchk: SQL-based SAN configuration checking. *Network and Service Management, IEEE Transactions on, 5*(2), pp 91–104.

[Goldsack et al., 2003] Goldsack, P., Guijarro, J., Lain, A., and Mecheneau, G. (2003). SmartFrog: Configuration and automatic ignition of distributed applications. *HP OVUA*, pp 1–9.

[Goldsack et al., 2009] Goldsack, P., Guijarro, J., Loughran, S., Coles, A., Farrell, A., Lain, A., Murray, P., and Toft, P. (2009). The SmartFrog configuration management framework. *ACM SIGOPS Operating Systems Review*, *43*(1), pp16-25.

[Hewson and Anderson, 2011] Hewson, J. and Anderson, P. (2011). Modelling System Administration Problems with CSPs. In *Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation (Mod-Ref'11)*, pp. 73-82.

[Hewson et al., 2013] Hewson, J., Anderson, P., and Gordon, A. Constraint-Based Autonomic Reconfiguration. In *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on* (pp. 101-110). IEEE.

[Horn, 2001] Horn, P. (2001). Autonomic Computing: IBM's Perspective on the State of Information Technology, IBM Corporation.

[ISO 9000] International Standards Organisation (2005). Quality management systems-Fundamentals and vocabulary (ISO 9000: 2005).

[ISO10040, 1998] ISO/IEC 10040: 1998. Information technology–Open Systems Interconnection–Systems management overview. ITU-T.

[Jackson, 2002] Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM), 11*(2), pp 256–290.

[Jones, 1990] Jones, C. B. (1990). Systematic software development using VDM, volume 2. Prentice Hall Englewood Cliffs.

[JORAM] Java Open Reliable Asynchronous Messaging (JORAM). OW2 Consortium, November, 2013. [Online]. Available : http://joram.ow2.org/.

[Juniper Networks, 2008] Juniper Networks (2008). What 's Behind Network Downtime ? Proactive Steps to Reduce Human Error and Improve Availability of Networks. November, 2013. [Online]. Available : http://www-05.ibm.com/uk/juniper/pdf/200249.pdf.

[Kanies, 2006] Kanies, L. (2006). Puppet: Next-generation configuration management. The USENIX Magazine. v31 i1, pp 19–25.

[Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *Computer, 36*(1), pp 41–50.

[KETFI, 2004] Ketfi, A. (2004). Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels. PhD thesis, Université Joseph Fourier de Grenoble.

[Kiciman and Wang, 2004] Kiciman, E. and Wang, Y. (2004). Discovering correctness constraints for self-management of system configuration. In *Autonomic Computing, 2004. Proceedings. International Conference on* (pp. 28-35). IEEE.

[Konstantinou et al., 2002] Konstantinou, A. V., Florissi, D., and Yemini, Y. (2002). Towards Self-Configuring Networks. In *DARPA Active NEtworks Conference and Exposition, 2002. Proceedings* (pp. 143-156). IEEE.

[Lindblad, 2013] Lindblad, J. (2013). Tutorial : NETCONF and YANG. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on.* IEEE

[Lymberopoulos et al., 2003] Lymberopoulos, L., Lupu, E., and Sloman, M. (2003). Using CIM to Realize Policy Validation within the Ponder Framework. *In DMTF Global Management Conference*.

[Mahajan et al., 2002] Mahajan, R., Wetherall, D., and Anderson, T. (2002). Understanding BGP misconfiguration. In *ACM SIGCOMM Computer Communication Review 32*(4), pp. 3-16.

[Meyer, 1992] Meyer, B. (1992). Eiffel: the language, volume 66. Prentice-Hall.

[Miller, 2005] Miller, B. (2005). The autonomic computing edge: Can you CHOP up autonomic computing. pages 1–7.

[Morin et al., 2009] Morin, B., Barais, O., Nain, G., and Jezequel, J. (2009). Taming Dynamically Adaptive Systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering* (pp. 122-132). IEEE Computer Society.

[Narain et al., 2008] Narain, S., Levin, G., Malik, S., and Kaul, V. (2008). Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management, 16*(3), 235-258.

[Narain et al., 2010] Narain, S., Talpade, R., and Levin, G. (2010). Network Configuration Validation. In*Guide to Reliable Internet Services and Applications* (pp. 277-316). Springer London.

[IDL, 2012] IDL Syntax and Semantics. *In Common Object Request Broker Architecture (CORBA) Specification, Version 3.3, November 2012, chapter 7*. OMG.

[OCL, 2006] Object Constraint Language (OCL), Version 2.0. (2006). Object Management Group (OMG).

[OML-UMLProfile, 2013] OMG UML Profile Specifications, November, 2013. [Online]. Available : http://www.omg.org/spec/.

[OMG, 2003] Object Management Group (OMG) (2003). MDA Guide Version 1.0. 1. Object Management Group, (OMG).

[**OpenMQ]** Open Message Queue, November, 2013. [Online]. Available : https://mq.java.net/.

[OpenPegasus] The OpenGroup — OpenPegasus, November, 2013. [Online]. Available http://www.opengroup.org/subjectareas/management/openpegasus.

[Oppenheimer et al., 2003] Oppenheimer, D., Ganapathi, A., and Patterson, D. A. (2003). Why do internet services fail and what can be done about it? *In 4th Usenix Symposium on Internet Technologies and Systems Vol. 3*.

[Pappas et al., 2004] Pappas, V., Xu, Z., Lu, S., Massey, D., Terzis, A., and Zhang, L. (2004). Impact of configuration errors on DNS robustness. *In ACM SIGCOMM Computer Communication Review, volume 34*, pp 319–330. ACM.

[Patterson, 2002] Patterson, D. (2002). A Simple Way to Estimate the Cost of Downtime. In *LISA* (Vol. 2, pp. 185-188).

[Patterson et al., 2002] Patterson, D., Brown, A., Broadwell, P., and Candea, G. (2002). Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science.

[Pavlou, 2007] Pavlou, G. (2007). On the evolution of management approaches, frameworks and protocols: a historical perspective. *In Journal of Network and Systems Management 15*(4), pp 425–445.

[Pressman, 2001] Pressman, R. S. (2001). Software Engineering – A Practitioner's Approach. McGraw-Hill Higher Education.

[QVT, 2008] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0. (2008). Object Management Group (OMG).

[Ramachandran et al., 2009] Ramachandran, V. and Gupta, M., Sethi, M., & Chowdhury, S. R. (2009). Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications. In *Proceedings of the 6th international conference on Autonomic computing* (pp. 169-178).

[Ramshaw et al., 2006] Ramshaw, L., Sahai, A., Saxe, J., and Singhal, S. (2006). Cauldron: a policy-based design tool. *In 7th IEEE International Workshop on Policies for Distributed Systems and Networks*, pp 113–122.

[RFC 2748] Durham, D., Boyle, J., Cohen, R., Herzog, S., Rajan, R., and Sastry, A. (2000). The COPS (Common Open Policy Service) Protocol. Internet Engineering Task Force (IETF), RFC 2748.

[RFC 2753] Yavatkar, R., Pendarakis, D., and Guerin, R. (2000). A Framework for Policy-based Admission Control. Internet Engineering Task Force (IETF), RFC 2753.

[RFC 3410] Case, J., Mundy, R., Partain, D., and Stewart, B. (2002). Introduction and Applicability Statements for Internet-Standard Management Framework. RFC 3410.

[RFC 3416] Presuhn, R., Case, J., McCloghrie, K., Rose, M., and Waldbusser, S. (2002). Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP) Internet Engineering Task Force (IETF), RFC 3416.

[RFC 3512] Claise, B. and Ersue, M. (2012). An Overview of the IETF Network Management Standards. Internet Engineering Task Force (IETF), RFC 3512.

[RFC 3535] Schoenwaelder, J. (2003). Overview of the 2002 IAB Network Management Workshop. nternet Engineering Task Force (IETF),RFC 3535.

[RFC 6020] Bjorklund, M. (2010). YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). Internet Engineering Task Force (IETF), RFC 6020.

[RFC 6241] Enns, R., Bjorklund, M., Schönwälder, J., and Bierman, A. (2011). NETCONF Configuration Protocol. Internet Engineering Task Force (IETF), RFC 6241.

[RFC 6632] MacFaden, M., Partain, D., Saperia, J., and Tackabury, W. (2003). Configuring Networks and Devices with Simple Network Management Protocol (SNMP)nternet Engineering Task Force (IETF), RFC 6632.

[Samaan and Karmouch, 2009] Samaan, N. and Karmouch, A. (2009). Towards autonomic network management: an analysis of current and future research directions. *Communications Surveys & Tutorials, IEEE, 11*(3), 22-36.

[Siddiqi, 2001] Siddiqi, A. (2001). Reconfigurability in Space Systems: Architecting Framework and Case Studies. PhD thesis, Massachussets Institute of Technology.

[Sinz et al., 2003] Sinz C, Khosravizadeh A, Kuchlin W, Mihajlovski V (2003). Verifying CIM models of Apache Web-server configurations. In *Quality Software, 2003. Proceedings. Third International Conference on* (pp. 290-297). IEEE.

[Sloman, 1994] Sloman, M. (1994). Policy driven management for distributed systems. *Journal of network and Systems Management, 2*(4), pp 333–360.

[SPEM, 2008] Software & Systems Process Engineering Meta-Model Specification. (2008), OMG.

[Spivey, 1989] Spivey, J. (1989). An introduction to Z and formal specifications. Software Engineering Journal.

[Stallings, 1999] Stallings, W. (1999). SNMP, SNMPv2, SNMPv3, and RMON 1 and 2. Addison-Wesley Longman Publishing

[Stawowski, 2011] Stawowski, M. (2011). Avoiding IT Services Failures with Change Management Automation and Configuration Optimization. *ISSA Journal, 9*(12), pp 24–28.

[Stumptner et al., 1998] Stumptner, M., Friedrich, G. E., and Haselböck, A. (1998). Generative Constraint-based Configuration of Large Technical Systems. *Artificial Intelligence for Engineering Design Analysis and Manufacturing, 12*(4), pp 307–320.

[Sun, 2006] Sun, Y. (2006). Complexity of system configuration management. PhD thesis, Tufts University.

[TOPCASED] Topcased the Open Source Toolkit for Critical Systems, November, 2013. [Online]. Available : URL http://www.topcased.org.

[Uchiumi et al., 2012] Uchiumi, T., Kikuchi, S., and Matsumoto, Y. (2012). Misconfiguration Detection for Cloud Datacenters using Decision Tree Analysis. In *Network Operations and Management Symposium (APNOMS), 2012 14th Asia-Pacific* (pp. 1-4).

[UML, 2007] OMG Unified Modeling Language (OMG UML), Superstructure V2.1.2. (2007). OMG.

[Walsh et al., 2004] Walsh, W., Tesauro, G., Kephart, J., and Das, R (2004). Utility functions in autonomic systems. *In Autonomic Computing, 2004. Proceedings. International Conference on (pp. 70-77). IEEE.*

[Whittle et al., 2008] Whittle, J., Sawyer, P., Bencomo, N., and Cheng, B. H. C. (2008). A language for self-adaptive system requirements. In *International Workshop on Service-Oriented Computing: Consequences for Engineering Requirements, 2008. SOCCER'08*., pp 24–29.

[Woodcock and Larsen, 2009] Woodcock, J. and Larsen, P. (2009). Formal Methods : Practice and Experience. *ACM Computing Surveys (CSUR), 41*(4), pp 1–40.

[Wuttke, 2010] Wuttke, J. (2010). Automatically generated runtime checks for design-level constraints. PhD thesis, Università della Svizzera Italiana.

[XMI, 2011] OMG MOF 2 XMI Mapping Specification. (2011) OMG.

[Yin et al., 2011] Yin, Z., Ma, X., and Zheng, J. (2011). An Empirical Study on Configuration Errors in Commercial and Open source Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (pp. 159-172).

Publications

Akue, L. ; Lavinal, E. ; Desprats, T ; Sibilla M., "Integrating an Online Configuration Checker with Existing Management Systems : Application to CIM/WBEM Environments", Actes de 7th International DMTF Workshop on Systems and Virtualization (SVM 2013), Zurich, Suisse, 18 Oct. 2013.

Akue, L. ; Lavinal, E. ; Desprats, T ; Sibilla M., "Runtime Configuration Validation for Self-configurable Systems", Actes de 13th International Symposium on Integrated Network Management (IM 2013), pp. 712–715, Ghent, Belgique, 27 – 31 Mai 2013.

Akue, L. ; Lavinal, E. ; Sibilla M., "A Model-Based Approach to Validate Configurations at Runtime ", Actes de 4th International Conference on Advances in System Testing and Validation Lifecycle (VALID 2012), pp. 133–138, Lisbonne, Portugal. (Best Paper Award).

Akue, L. ; Lavinal, E. ; Sibilla M., "A Dynamic Configuration Validation Language ", Actes de 4th Symposium On Configuration Analytics and Automation (SAFECONFIG 2011), pp. 1–2, Arlington VA, Etats-Unis, 31 Oct 2011 – 1 Nov 2011.

Akue, L. ; Lavinal, E. ; Sibilla M., "Towards a Validation Framework for Dynamic Reconfiguration", Actes de International Conference on Network and Service Management (CNSM 2010), pp. 314–317, Niagara Falls, Canada, 25 – 29 Oct 2010.