IT infrastructure modeling for risk identification and prevention

Benjamin Somers

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE
PAYS DE LA LOIRE – IMT ATLANTIQUE

ÉCOLE DOCTORALE N° 648
Sciences pour l'Ingénieur et le Numérique
Spécialité : *Informatique*

Par
Benjamin SOMERS

IT infrastructure modeling for risk identification and prevention

Thèse présentée et soutenue à IMT Atlantique, Brest, le 15 mai 2024
Unité de recherche : Lab-STICC
Thèse N° : 2024IMTA0401

Rapporteurs avant soutenance :

Catherine DUBOIS Professeur des universités à l'ENSIIE
Régine LALEAU Professeur des universités à l'Université Paris-Est Créteil

Composition du Jury :

Présidente :	Isabelle BORNE	Professeur des universités à l'Université Bretagne Sud
Examinateurs :	Catherine DUBOIS	Professeur des universités à l'ENSIIE
	Régine LALEAU	Professeur des universités à l'Université Paris-Est Créteil
	Thomas LEDOUX	Professeur à IMT Atlantique
Dir. de thèse :	Fabien DAGNAT	Maître de conférences HDR à IMT Atlantique
Encadrant :	Jean-Christophe BACH	Maître de conférences à IMT Atlantique

Invités :

Philippe LE GOFF Responsable sécurité opérationnelle au Crédit Mutuel Arkéa
Yohan BELLEGUIC Architecte *cloud* au Crédit Mutuel Arkéa

Remerciements

Avant de présenter mes travaux, je tiens à débuter ce mémoire en remerciant celles et ceux qui m'ont aidé à arriver là où je suis aujourd'hui, et qui ont fait de cette période de ma vie une expérience hors du commun.

Je souhaite tout d'abord remercier très chaleureusement les personnes qui ont encadré ce projet : Fabien Dagnat et Jean-Christophe Bach côté académique, et Philippe Le Goff, Nicolas Dupeux et Yohann Belleguic côté entreprise. J'ai énormément appris à vos côtés, tant du point de vue technique que scientifique ou humain. Je sais que je n'ai pas toujours été l'élément le plus simple à encadrer et que je nous ai imposé des rythmes de travail pour le moins inconfortables ; vous voilà libérés ! Je tiens également à remercier les membres de mon comité de suivi individuel, Alain Plantec et Stefano Zacchiroli, pour leur bienveillance et leurs précieux conseils durant nos trois rencontres. Vos retours et votre regard extérieur sur mes travaux m'ont permis de prendre le recul dont je manquais au début de ma thèse. Je remercie mes rapporteurs, Catherine Dubois et Régine Laleau, dont les retours sur mon manuscrit ont mis en lumière de nombreux axes d'amélioration que j'espère avoir su au mieux explorer. Je remercie également les autres membres de mon jury, Isabelle Borne et Thomas Ledoux, pour leur intérêt dans mes travaux et nos échanges lors de ma soutenance.

Je dois mon parcours en partie au hasard de quelques rencontres marquantes, aussi je tiens à saluer deux personnes qui ne liront probablement jamais ces lignes, mais qui ont certainement influencé celui-ci bien plus qu'ils ne pouvaient alors l'imaginer. À vous d'abord M. Bouhnif ; être votre élève fut une expérience aussi passionnante qu'elle fut terrifiante, mais vous êtes la personne qui m'a le plus tôt inspiré à faire des études supérieures longues. Vous m'aviez dit lorsque j'avais 13 ans que je deviendrais chercheur ; je pense ne pas vous avoir fait mentir. Et à toi Clément ; tu m'as aidé à faire mes marques à Pasquet, tu avais les mêmes intérêts bizarres que moi, et tu as su m'inspirer le sérieux dont je manquais pour faire des études supérieures. Et surtout, c'est toi qui m'as fait découvrir les classes préparatoires en général, et plus précisément le Lycée du Parc, où je t'ai suivi deux ans plus tard ; merci.

J'ai eu la chance, durant ces études supérieures, d'être accompagné par la Fondation Georges Besse, sans le soutien de laquelle celles-ci auraient été autrement plus fastidieuses. Être Lauréat de la Fondation m'a ouvert sur l'autre et donné un regard nouveau sur mon environnement. J'y ai découvert une grande famille aux valeurs profondément humaines qui me portent aujourd'hui encore au quotidien. Je remercie tout particulièrement Sylvie et Françoise qui m'ont aidé à me sentir à ma place à la Fondation. Merci enfin à Gilbert, Madame Larras et Messieurs Elbrahimi, Bertello, Gonnord et Chapey pour votre soutien dans ma candidature.

Mon arrivée à Télécom Bretagne m'a permis de découvrir des univers jusqu'alors inexplorés. J'ai très rapidement pris mes marques au sein du Club Troll, qui a été d'une grande importance pour moi au long de mon parcours étudiant. Ces soirées et nuits passées à jouer à Twilight Imperium, Ascension et autres Donjons et Dragons m'ont permis de faire connaissance avec de nombreux amis, qui m'ont peu à peu fait découvrir le monde de la recherche dès ma première année. Je leur en suis reconnaissant, en particulier Nicolas, avec lequel j'ai eu de l'occasion de travailler durant mes études.

En parallèle, j'ai pu m'impliquer dans l'Association ResEl, à qui je dois la quasi-intégralité de l'expertise technique que j'ai aujourd'hui. Le ResEl fut un terrain d'expérimentation unique, qui a beaucoup inspiré de nombreux axes d'exploration de mes travaux de thèse.

Jeg ønsker å takke kameratene mine i Oslo for den hjertelige velkomsten. I Norge har jeg oppdaget en annen måte å vurdere livet på, og jeg kunne også virkelig finne meg selv. Så vanskelig det var å komme tilbake til Frankrike!

Enfin, j'ai dédié une grande partie de ma vie associative durant ma thèse à FedeRez, dont la gestion au quotidien a été riche en enseignements. J'ai pu rencontrer des personnes formidables grâce à cette association et ai plaisir d'avoir à ma manière contribué à ce qu'elle est devenue aujourd'hui.

Du fait du contexte sanitaire pour le moins délicat, le quotidien de ma thèse aurait été bien plus compliqué sans mes colocataires du Pavillon. Je souhaite donc remercier Damien et Jonathan pour ces dégustations de bières sans fin, ces barbecues du dimanche qui commençaient à 11 h et qui finissaient à 22 h, ces expériences pyrotechniques et autres idées ridicules que nous avions trop souvent tendance à pousser à l'extrême. Cette période à vos côtés était incroyable !

Ces derniers remerciements vont à ma famille, qui a toujours été là pour moi, qui m'a toujours encouragé et m'a toujours poussé à donner le meilleur de moi-même. Je tiens à remercier tout particulièrement ma mère, dont les efforts m'ont porté tout au long de mon parcours.

Chapitre

Résumé en français

Table des matières

.1 Introduction v
.2 Contributions vi
.3 Gestion du risque IT vii
.4 Vérification d'infrastructures viii
.5 Déploiement et maintenance d'infrastructures ix
.6 Intégration de nos travaux ix
.7 Conclusion x

.1 Introduction

Les infrastructures informatiques sont omniprésentes dans notre vie quotidienne. Véritables colonnes vertébrales des systèmes modernes, elles assurent le bon fonctionnement de nos centrales électriques, systèmes de transport, institutions bancaires et autres systèmes vitaux. Ces systèmes doivent inspirer une certaine confiance, laquelle est renforcée par le respect d'un grand nombre d'exigences (techniques, fonctionnelles, légales...).

Quand l'on parle d'infrastructure informatique, sont usuellement entendus les serveurs, l'équipement réseau, les câbles ou encore les composants logiciels qu'une entreprise exploite. Mais il serait bien maladroit de s'y limiter ; ces équipements ont besoin d'électricité et de refroidissement, les sociétés délèguent un certain nombre de responsabilités à des sous-traitants, et les employés interagissent avec tout cet environnement au quotidien. C'est de cet écosystème complexe que naît le risque, dont l'étude et la gestion doivent alors couvrir un grand nombre d'aspects de l'entreprise.

Pour comprendre les interactions en jeu dans ces entreprises, une bonne *connaissance* de leurs infrastructures est nécessaire. Si cette connaissance est facilement mobilisable dans une petite structure avec peu de départements et d'employés, ce n'est pas le cas dans de grandes compagnies où chaque domaine d'expertise ne représente qu'une petite fraction de l'infrastructure complète. Dans ce mémoire, nous adoptons une approche dirigée par les *modèles* et nous prenons le parti de dire que *tout* peut fondamentalement en constituer un. Ainsi, chaque artéfact de conception, chaque représentation mentale que se fait un employé d'une entreprise, correcte ou incorrecte, constitue un modèle dont la *fédération* avec d'autres modèles permet de tirer de nouvelles connaissances.

Durant nos travaux, nous nous sommes posé des questions telles que « en quoi la mise à jour de tel équipement réseau affecte-t-elle la sûreté et la sécurité de notre infrastructure ? », « comment pouvons-nous configurer cette application pour répondre à nos besoins métiers ? » ou encore « considérant les objectifs de l'entreprise, est-il avisé de procéder à tel changement architectural ? ». Tout d'abord, nous faisons un tour d'horizon du processus de gestion du risque pour les infrastructures informatiques en section .3 (chapitre IV du mémoire). Dans un deuxième temps, nous abordons la modélisation et la vérification formelle de modèles en section .4 (chapitre V du mémoire). Ensuite, nous présentons CL/I, un langage de modélisation d'infrastructures au cœur de notre travail de thèse, en section .5 (chapitre VI du mémoire). Nous terminons notre développement par une prise de recul sur l'intégration de nos travaux dans un contexte industriel en section .6 (chapitre VII du mémoire). Enfin, nous concluons sur les apports de nos travaux et leurs perspectives en section .7 (chapitre VIII du mémoire).

.2 Contributions

Nous avons apporté diverses contributions, tant scientifiques que techniques, tout au long de cette thèse.

Publications scientifiques

- Benjamin Somers, Fabien Dagnat, Jean-Christophe Bach. How IT Infrastructures Break: Better Modeling for Better Risk Management. 17th International Conference on Risks and Security of Internet and Systems (CRiSIS 2022), Décembre 2022, Sousse, Tunisie. pp. 169–184. ⟨hal-03801086⟩. ⟨10.1007/978-3-031-31108-6_13⟩.
- Benjamin Somers, Fabien Dagnat, Jean-Christophe Bach. Modeling heterogeneous IT infrastructures: a collaborative component-oriented approach. 28th International working conference on Exploring Modeling Methods for Systems Analysis and Development (EMMSAD 2023), Juin 2023, Saragosse, Espagne. pp. 227–242. ⟨hal-04083449⟩. ⟨10.1007/978-3-031-34241-7_16⟩.

Productions techniques

- Au chapitre IV,
  - Une formalisation du processus de gestion du risque (section IV.1.1),
  - La traduction des référentiels CAPEC, CVE et CWE du MITRE en ontologies, pour effectuer des analyses sémantiques poussées du risque (section IV.2.3),
  - Une rosace du risque, représentant la prévalence des familles de faiblesses dans les vulnérabilités cyber identifiées à ce jour (figure IV.17),
  - Diverses recommandations pour procéder à l'analyse des risques sur des infrastructures informatiques modernes (section IV.4),
  - Un *framework* pour partager les analyses de risque (section IV.5) ;
- Au chapitre V,
  - Divers modèles formels et techniques pour une petite infrastructure informatique (sections V.2 et V.3),
  - Des recommandations pour la modélisation d'infrastructures informatiques (sections V.3 et V.4) ;

- Au chapitre VI,
  - CL/I, un langage de description d'infrastructures (section VI.2),
  - Un *backend* Z3 pour CL/I (section VI.3),
  - Des études de cas couvrant divers domaines (section VI.4) ;
- Au chapitre VII,
  - Un métamodèle orienté composants et responsabilités pour les infrastructures informatiques (section VII.1),
  - Des recommandations pour la modélisation collaborative d'infrastructures (sections VII.3 et VII.4).

🐓.3 Gestion du risque IT

Du fait de leur importance dans notre quotidien, les infrastructures informatiques sont sujettes à de nombreuses normes et régulations provenant de diverses entités tout autour du globe. Bien que le risque cyber (tel que les *malwares*, les attaques par déni de service, les fuites de données...) représente une part importante du risque IT, il ne saurait s'y résumer. La typologie du risque est beaucoup plus large et prend en compte les événements naturels, les pertes financières, la sûreté de fonctionnement des infrastructures...

Nous développons dans le chapitre IV le processus de gestion du risque informatique au travers d'un formalisme: le cycle du risque, que nous raffinons tout au long de ce mémoire. Ce cycle comporte cinq grandes phases:

- L'évaluation des risques, consistant à identifier un ensemble de risques auxquels une infrastructure est sujette ;
- Le filtrage, consistant à exclure le risque jugé acceptable ou pour lequel aucune remédiation n'est envisageable ;
- L'établissement d'exigences de sûreté et de sécurité grâce notamment à des normes, réglementations, ou contrats ;
- La définition de contraintes sur les infrastructures pour appliquer les exigences ;
- La réification des infrastructures.

Afin d'évaluer le risque, nous étudions au chapitre IV différentes taxonomies du risque, et développons une ontologie à partir des référentiels du MITRE afin de systématiser notre étude. Nous proposons au chapitre V de faire que cette évaluation découle d'un ensemble de propriétés formelles vérifiées à partir de différents modèles d'infrastructure. Une première phase de modélisation permet alors d'associer des portions d'infrastructure à des modèles formels et techniques et une seconde phase de vérification permet de déduire, à partir de prouveurs formels, des propriétés. Nous présentons ce processus plus en détail dans ce résumé en section 🐓.4.

Pour filtrer le risque acceptable, nous nous intéressons à différents *frameworks* d'analyse de risque et étudions dans quelle mesure ceux-ci peuvent être adaptés aux infrastructures informatiques, et en particulier aux infrastructures *cloud* modernes. Cette étude nous permet de déduire certaines recommandations pour l'analyse de risque et de discuter des critères d'acceptabilité de ce risque.

Les infrastructures présentant des risques de sûreté et de sécurité sont soumises à un certain nombre d'exigences liées aux secteurs d'activité des entreprises. Ces exigences vont de la simple recommandation, avec notamment le [GR-63-CORE] (donnant des consignes pour la protection physique des infrastructures de télécommunications), à l'obligation à l'échelle supranationale, avec notamment le RGPD ([GDPR], cadrant la collecte et l'utilisation des données personnelles), en passant par l'exigence sectorielle, avec notamment [PCI DSS] (visant à réduire la fraude monétique).

Ces exigences sont ensuite traduites en un ensemble de contraintes sur l'infrastructure. Pour les risques liés à la sécurité, ces contraintes visent à réduire la surface d'attaque, notamment en assurant un bon cloisonnement des réseaux et en implémentant des contrôles d'accès. Pour les risques liés à la sûreté, ces contraintes visent à fiabiliser l'infrastructure, notamment en garantissant une bonne réplication des divers services et en mettant en place des mécanismes de surveillance de l'intégrité des composants.

Finalement, les infrastructures ainsi contraintes sont concrétisées, soit par la création d'une nouvelle infrastructure, soit en procédant à la mise à niveau d'une infrastructure déjà existante. Les cinq étapes de ce processus de gestion du risque sont manuelles, aussi nous nous attachons dans ce mémoire à fournir diverses pistes d'amélioration pour son automatisation.

.4 Vérification d'infrastructures

Une fois notre cadre théorique posé, nous nous attachons dans le chapitre V à raffiner l'étape d'évaluation des risques. Lorsque l'on cherche à étudier une infrastructure informatique complète, de nombreux désalignements sémantiques apparaissent : l'infrastructure a très souvent des composantes techniques, humaines ou encore organisationnelles, qui sont exprimées à des niveaux d'abstraction différents, avec des vocabulaires différents. Ainsi, si l'on considère une infrastructure sous son aspect matériel, de très bas niveau, beaucoup de concepts ne sont pas reflétés (comme les câbles ou l'électricité) dans une vue fonctionnelle, de très haut niveau, de la même infrastructure.

Les connaissances des employés d'une entreprise étant limitées à leurs domaines d'expertise respectifs, de nombreuses représentations mentales de son infrastructure peuvent coexister, à différents niveaux d'abstraction. Dès lors, une vision globale ne peut être obtenue que par l'établissement d'un modèle fédérant les représentations de chacun ; nous présentons cela plus en détail en section .6.

L'évaluation des risques est un processus principalement manuel, aussi celle-ci est directement influencée par la manière dont les faits sont présentés aux auditeurs et par les aspects de l'infrastructure qu'ils choisissent d'aborder. Pour bénéficier d'une étude des risques plus objective, nous cherchons donc à réduire l'appréciation humaine en automatisant ce processus en trois étapes :

- Le développement de modèles d'infrastructure ;
- La vérification de propriétés sur ces modèles ;
- L'interprétation de ces propriétés dans une taxonomie du risque.

Nous construisons pas à pas dans ce chapitre une étude de cas représentant une infrastructure technique simple, mobilisant des modèles sur plusieurs niveaux d'abstraction (architecture physique, topologie réseau, comportement dynamique). Nous montrons comment composer des modèles techniques avec des modèles plus formels, et procédons à la modélisation de propriétés en logique temporelle et

à leur vérification à l'aide du logiciel UPPAAL. Tout au long de notre étude de cas, nous mettons en évidence divers obstacles qui peuvent survenir dans le processus de modélisation et dans la composition et la vérification consécutives des modèles produits.

Nous concluons ce chapitre en discutant des possibilités d'automatisation du processus et proposons diverses pistes de développements futurs.

.5 Déploiement et maintenance d'infrastructures

La complexité des infrastructures informatiques ne cesse de croître, et cette tendance s'est accélérée depuis l'avènement des technologies du *cloud*. Cette complexité engendre des difficultés pour garantir le bon fonctionnement des infrastructures et le respect des exigences spécifiées. Les facteurs pouvant entraîner des non-conformités sont nombreux, et leur présence dans des infrastructures critiques peut être catastrophique.

Dans le chapitre VI, nous nous concentrons sur la triade exigences–configuration–exécution et étudions les non-conformités pouvant exister entre (et dans) chacun de ces trois domaines. Plus précisément, nous nous intéressons à comment les exigences sont traduites dans des configurations d'infrastructure, comment ces configurations se reflètent dans l'exécution de l'infrastructure, et comment cette exécution respecte ou non les exigences initiales. Dans ce chapitre, nous entendons le terme « configuration » au sens large, à savoir la configuration des logiciels, l'architecture des systèmes, et plus généralement tous les éléments susceptibles de contrôler la structure ou le comportement des composants d'infrastructure.

Pour nous assister dans notre étude, nous avons développé un langage de description d'infrastructures, CL/I, permettant de modéliser structurellement les infrastructures, de représenter les exigences sous forme de prédicats logiques, et d'instancier les modèles. Nous avons bâti autour de CL/I un écosystème de compilation permettant de relier nos modèles d'infrastructures à divers *model checkers* et moteurs d'exécution. Nous présentons dans ce mémoire le langage au travers d'exemples concrets et formalisons sa compilation en commandes SMT-LIB pour Z3.

Enfin, nous décrivons deux études de cas ; l'une autour de l'hyperviseur [Proxmox VE], où nous exhibons un exemple de désalignement entre exigences et contraintes et identifions la violation de certaines propriétés à l'exécution ; l'autre où nous analysons les droits d'accès des employés d'une entreprise en fonction de leurs compétences et responsabilités. Notre langage est en constante évolution et certaines de ses constructions ne sont pas encore figées ; de plus, l'implémentation de notre compilateur n'est pas complète vis-à-vis du langage. Malgré la nature de prototype du compilateur, les résultats que nous avons obtenus sont prometteurs et nous envisageons de mener des études à plus grande échelle pour le valider.

.6 Intégration de nos travaux

Le dernier chapitre du développement de notre mémoire se concentre sur l'intégration dans un contexte industriel des travaux que nous avons présentés. Nous faisons une passerelle entre nos travaux et la discipline de l'*Enterprise Modeling* pour étendre nos modèles d'infrastructures techniques à des modèles plus génériques d'entreprises. Nous encourageons la collaboration dans le processus de modélisation et proposons un métamodèle centré sur les composants et les responsabilités pour représenter les infrastructures et leurs acteurs.

Chaque employé ayant sa vision locale de l'entreprise, avec son jargon et ses abstractions, il est crucial de fédérer le savoir pour construire une vision d'ensemble, une *big picture* de l'entreprise. Mais il ne s'agirait pas de se contenter pour les parties prenantes de modéliser chacune de son côté sa vue de l'infrastructure et d'espérer qu'une sorte d'algorithme d'agrégation crée cette *big picture* automatiquement. Les modèles peuvent comporter des erreurs, des imprécisions ou des inconnues que seule la confrontation des équipes peut permettre de lever. Nous proposons ainsi diverses pistes pour guider la modélisation collaborative.

Notre approche ne cherche pas à remplacer des *frameworks* de modélisation bien établis au sein des entreprises, mais bien de les rassembler, de les fédérer, afin de mener des analyses plus complexes, couvrant différents modèles dans différents départements des entreprises. Au travers de la fédération de modèles, nous pensons que la modélisation des entreprises peut rassembler de nombreuses parties prenantes, tout en préservant les outils et modèles que celles-ci sont habituées à exploiter.

🐓.7 Conclusion

Nous avons durant cette thèse construit des liens entre trois communautés scientifiques : gestion des risques, méthodes formelles et modélisation d'entreprise. Ces liens se sont organiquement développés à mesure que nous avons suivi le cycle du risque, formalisé en section 🐓.3. Il ressort de ce tour d'horizon du risque que la discipline demeure très manuelle, malgré un cadre technique où l'automatisation est devenue la norme. De plus, les méthodes développées par les différentes communautés techniques et scientifiques manquent d'interopérabilité, et rares sont les initiatives actuelles traitant du problème.

Nous voyons nos différents travaux autour du cycle du risque comme une contribution à un socle commun réunissant les communautés que nous avons visées durant cette thèse. Nous présentons plus en détail nos perspectives au chapitre VIII.

Table of Contents

Remerciements **iii**

Chapitre 🐓 Résumé en français **v**

🐓.1 Introduction v
🐓.2 Contributions vi
🐓.3 Gestion du risque IT vii
🐓.4 Vérification d'infrastructures viii
🐓.5 Déploiement et maintenance d'infrastructures ix
🐓.6 Intégration de nos travaux ix
🐓.7 Conclusion x

Table of Contents **xi**

Chapter I Introduction **1**

I.1 Context 1
I.1.1 What exactly is an IT infrastructure? 2
I.1.2 IT infrastructures then and now 3
I.1.3 IT professions 4
I.2 Problem statement 6
I.3 Contributions and outline 7
I.4 Funding 7

Chapter II State of the Art **9**

II.1 Managing risk in IT infrastructures 9
II.1.1 Requirements 11
II.1.2 Environment 12
II.1.3 Approaches to risk 13
II.1.4 Our position 14
II.2 Modeling and checking infrastructures 14
II.2.1 Infrastructure modeling 16
II.2.2 Model checking 16
II.2.3 Our position 18
II.3 IT Infrastructures dynamics 18

II.3.1 Infrastructure life cycle . . . 18
II.3.2 Deployment . . . 19
II.3.3 Monitoring . . . 20
II.3.4 Our position . . . 21
II.4 Conclusion . . . 21

Chapter III Reader's Guide **23**
III.1 Approach . . . 23
III.2 Progress . . . 24
III.3 Big picture . . . 24

Chapter IV Managing Risk in IT Infrastructures **25**
IV.1 The risk cycle . . . 27
IV.1.1 Formalism . . . 27
IV.1.2 Properties . . . 29
IV.1.3 Iteration . . . 30
IV.1.4 Change . . . 31
IV.1.5 Approach . . . 33
IV.2 Risk classification . . . 33
IV.2.1 Taxonomy efforts . . . 33
IV.2.2 The case of MITRE . . . 37
IV.2.3 An ontology over MITRE . . . 38
IV.3 Risk analysis frameworks . . . 44
IV.3.1 Traditional frameworks . . . 44
IV.3.2 Modern initiatives and IT infrastructures . . . 46
IV.4 Risk assessment and tolerance criteria . . . 47
IV.4.1 Analyzing parts . . . 47
IV.4.2 Analyzing systems . . . 48
IV.4.3 Closing the loop . . . 49
IV.5 Sharing analyses . . . 51
IV.5.1 Building open analyses . . . 52
IV.5.2 Composing analyses . . . 52
IV.6 Conclusion . . . 52

Chapter V Checking IT Infrastructures **55**
V.1 Rethinking risk assessment . . . 56
V.1.1 Formalism . . . 56
V.1.2 Risk and properties . . . 58
V.2 Modeling IT Infrastructures . . . 58
V.2.1 From the technical world... . . . 60
V.2.2 ... to the formal one . . . 62
V.2.3 Case study . . . 64
V.3 Model checking . . . 69

V.3.1 Properties and checkers . . . 70
V.3.2 The need for proper abstractions . . . 72
V.3.3 Going back to our case study . . . 72
V.4 Automating risk assessment . . . 75
V.4.1 Expressing formal properties... . . . 75
V.4.2 ... and combining models together . . . 76
V.5 Conclusion . . . 77

Chapter VI Deploying and Maintaining IT Infrastructures 79
VI.1 Requirements–configuration–execution triad . . . 80
VI.1.1 Inconsistencies . . . 81
VI.1.2 Change . . . 82
VI.1.3 Formalization . . . 83
VI.1.4 Approach . . . 84
VI.2 The CL/I language . . . 84
VI.2.1 Another language? . . . 84
VI.2.2 Modeling in CL/I . . . 85
VI.2.3 Syntactic processing . . . 86
VI.2.4 Semantic processing . . . 88
VI.2.5 Extensions . . . 90
VI.3 Mapping into Z3 . . . 91
VI.3.1 Translation rules . . . 92
VI.3.2 Conformance checking . . . 95
VI.4 Case studies . . . 95
VI.4.1 Virtual environment model . . . 95
VI.4.2 Proxmox VE configuration and execution . . . 97
VI.4.3 Model checking . . . 98
VI.4.4 Scaling . . . 99
VI.4.5 A more complete case study . . . 100
VI.5 Conclusion . . . 101

Chapter VII Integrating our Approach 103
VII.1 Theoretical framework . . . 104
VII.1.1 Actors and responsibilities . . . 104
VII.1.2 Components and instances . . . 106
VII.1.3 Metamodel links . . . 106
VII.2 Collaborative enterprise modeling . . . 106
VII.2.1 Enterprise modeling . . . 107
VII.2.2 Collaborative modeling . . . 108
VII.3 Federating models . . . 109
VII.3.1 Modeling guidelines . . . 110
VII.3.2 Scaling infrastructures . . . 112
VII.4 Integration guidelines . . . 113

VII.4.1 Component catalogs . . . 113
VII.4.2 *A posteriori* modeling . . . 114
VII.4.3 *A priori* modeling . . . 116
VII.5 Case study . . . 116
VII.5.1 Heterogeneous models... . . . 116
VII.5.2 ... linked together . . . 118
VII.5.3 Exploiting the model . . . 118
VII.6 Conclusion . . . 120

Chapter VIII Conclusion **121**
VIII.1 Synthesis of contributions . . . 121
VIII.2 Limitations and perspectives . . . 123
VIII.2.1 Risk management . . . 123
VIII.2.2 Modeling . . . 124
VIII.2.3 CL/I . . . 124
VIII.2.4 Enterprise integration . . . 124

Bibliography **125**

Appendix A `mitre2owl` Algorithm **143**
A.1 Detail of the algorithm . . . 143
A.1.1 Parsers_S . . . 143
A.1.2 Parsers_D . . . 145
A.2 Semantic transformation . . . 147

Appendix B UPPAAL Model **149**
B.1 Source code . . . 149
B.1.1 Common functions . . . 149
B.1.2 Corosync cluster . . . 149
B.1.3 Multi-quorum Corosync cluster . . . 150
B.1.4 Corosync node . . . 152
B.1.5 Network node above Corosync . . . 153
B.1.6 Declarations for Corosync . . . 153
B.2 Traces . . . 156
B.2.1 Scenario 1 . . . 156
B.2.2 Scenario 2 . . . 157
B.2.3 Scenario 3 . . . 158

Appendix C CL/I **159**
C.1 Language grammar . . . 160
C.2 AST construction rules . . . 168
C.3 Transformation from CL/I's AST to the CLIR . . . 170
C.3.1 AST . . . 170
C.3.2 Structure . . . 171

C.3.3 Right values 174
C.3.4 RTRDot 175
C.3.5 RInit 176
C.3.6 Right types 177

List of Figures and Tables

Figure I.1 Infrastructures and their environment . . . 2
Figure I.2 Traditional IT infrastructure . . . 3
Figure I.3 Modern cloud infrastructure . . . 4
Figure I.4 Comparison between on-site and XaaS strategies . . . 5

Figure II.1 Different states for systems, according to IEC 60050-192 . . . 10
Figure II.2 UEML object projected as two models onto two operational languages . . . 15
Figure II.3 The three dimensions of specification according to RM-ODP . . . 15
Figure II.4 Model checking process, according to [Baier08] . . . 17
Figure II.5 Development life cycle . . . 19

Figure III.1 Contributions of this dissertation to each scientific community . . . 24

Figure IV.1 ISO 31000 iterative process . . . 26
Figure IV.2 The risk cycle . . . 27
Figure IV.3 Risk management process . . . 28
Figure IV.4 Graphical illustration of the reify function . . . 29
Figure IV.5 Risk management cycle . . . 32
Figure IV.6 Hierarchy of CWEs . . . 32
Figure IV.7 Excerpt from OWASP ASVS . . . 34
Figure IV.8 Excerpt from NIST SP 800-53 . . . 34
Figure IV.9 Excerpt from ENISA's Threat Taxonomy . . . 35
Figure IV.10 Screenshot of SCAP Workbench . . . 36
Figure IV.11 Relationships between `CAPEC-25`, `CVE-2009-1388` and `CWE-833`. . . . 37
Figure IV.12 Structure of `mitre2owl` . . . 39
Figure IV.13 Algorithm for parsing MITRE XML schemas . . . 40
Figure IV.14 Algorithms for parsing MITRE XML types and attributes . . . 40
Figure IV.15 Integration of our ontologies in an industrial process . . . 41
Figure IV.16 Exploration of the Deadlock CWE . . . 42
Figure IV.17 Vulnerability–Weakness rosette . . . 43
Table IV.1 Failure Mode and Effects Analysis of a server . . . 45
Figure IV.18 Fault-Tree Analysis of a server not powering up . . . 45
Figure IV.19 Feature matching when assembling components . . . 49

Figure IV.20 Typical server motherboard . . . 49
Figure IV.21 Excerpt from PCI DSS 4.0 . . . 50
Figure IV.22 Generic model for risk analysis . . . 51
Figure IV.23 Excerpt from a server manual . . . 53
Figure IV.24 Multi-level open risk analysis . . . 53

Figure V.1 Model of a model and its properties . . . 57
Figure V.2 Refinement of the `assess` function . . . 57
Figure V.3 Composition of models . . . 57
Figure V.4 A scribble, an automaton, a hardware inventory: three models . . . 59
Figure V.5 NetBox model of a datacenter rack . . . 60
Figure V.6 Excerpt from an SNMP walk over a network switch . . . 61
Figure V.7 Excerpt from the `Win32_Process` instances of a Windows 7 system . . . 61
Figure V.8 UPPAAL model of a lock and processes able to acquire it . . . 62
Figure V.9 Overview of a datacenter with three rooms . . . 63
Figure V.10 Physical infrastructure modeled with NetBox . . . 63
Figure V.11 Network infrastructure extracted from NetBox . . . 65
Figure V.12 Loss of a 5-node quorum with 3 votes . . . 66
Figure V.13 UPPAAL model of Corosync . . . 66
Figure V.14 Trace of the loss of a 5-node quorum with 3 votes . . . 67
Figure V.15 Linking technical models to a formal model . . . 68
Figure V.16 UPPAAL transitional models . . . 68
Figure V.17 Linking technical models to a formal model thanks to transitional models . . . 68
Figure V.18 Execution of our lock model in the UPPAAL simulator . . . 71
Figure V.19 Verification time as a function of model size . . . 71
Figure V.20 State space exploration strategies . . . 71
Figure V.21 Composed model with its properties . . . 73
Figure V.22 Simplified UPPAAL model for our system . . . 74
Figure V.23 Parse tree for a textual property . . . 76

Figure VI.1 *Requirements–configuration–execution* triad: technical point of view . . . 81
Figure VI.2 Deployment graphs . . . 83
Figure VI.3 *Requirements–configuration–execution* triad: mathematical point of view . . . 83
Figure VI.4 *Requirements–configuration–execution* triad: our approach . . . 84
Figure VI.5 Our language as a pivot between formal and informal tools . . . 85
Figure VI.6 CL/I components for a user, a permission triple and a file . . . 86
Figure VI.7 CL/I instances of the components in figure VI.6 . . . 86
Figure VI.8 Syntactic transformation of the `User` component and the `file_1` instance . . . 87
Figure VI.9 Translation from CL/I to the CLIR . . . 89
Figure VI.10 Simplified semantic rules from CL/I to the CLIR . . . 90
Figure VI.11 Two-stage processing of CL/I models . . . 90
Figure VI.12 Helper functions for our translation rules . . . 92
Figure VI.13 Translation rules . . . 93

Figure VI.14 Translation from CL/I to the CLIR to Z3 . . . 94
Figure VI.15 Model for `Node`, `Group` and `Cluster` . . . 96
Figure VI.16 Model for `Virt` and affinity and antiaffinity rules . . . 97
Figure VI.17 Proxmox VE configuration translation . . . 98
Table VI.1 Summary of relations . . . 98
Figure VI.18 Verification time (log scale) as a function of the number of `Virts` . . . 99
Figure VI.19 Multi-domain model . . . 100

Figure VII.1 Component- and responsibility-oriented metamodel . . . 105
Figure VII.2 Specialization of the metamodel for ReMoLa compatibility . . . 105
Figure VII.3 Representation of the ISO 19439 standard . . . 108
Figure VII.4 Unification, composition, federation: three approaches to collaborative models 109
Figure VII.5 Inconsistencies by addition, approximation and mistake and their resolution . 109
Figure VII.6 Model for a generic service checker . . . 111
Figure VII.7 Model reconciliation with three points of view . . . 112
Figure VII.8 An example of model reuse . . . 114
Figure VII.9 *A posteriori* modeling by federation . . . 115
Figure VII.10 *A priori* modeling . . . 115
Figure VII.11 eBank's organizational structure . . . 117
Figure VII.12 BPMN diagram for purchasing new hardware . . . 117
Figure VII.13 Task catalog . . . 117
Figure VII.14 ePay's big picture . . . 119
Figure VII.15 Impact tree from application slowness to financial impact . . . 119

Figure VIII.1 Refinement of the risk cycle throughout this dissertation . . . 122

Figure B.1 Trace for the scenario 1 . . . 156
Figure B.2 Trace for the scenario 2 . . . 157
Figure B.3 Trace for the scenario 3 . . . 158

Table C.1 Presentation of CL/I's AST, along with a few examples to build each node . . 168

Chapter I

Introduction

Contents

I.1 Context 1
I.1.1 What exactly is an IT infrastructure? 2
I.1.2 IT infrastructures then and now 3
I.1.3 IT professions 4
I.2 Problem statement 6
I.3 Contributions and outline 7
I.4 Funding 7

I.1 Context

Computers are all around us. Over the years, IT infrastructures have become the backbone of businesses, the heart of global communications, and a vital part of our day-to-day lives. They drive power plants, transportation systems, banking institutions, life-critical systems..., all of which must inspire confidence in their safety and security. But beyond inspiring trust, all of them must work in accordance with a set of specifications.

Modern companies often rely on very diverse IT infrastructures to support their activities. As they sell products and services to customers, businesses attach importance to maintaining a certain level of quality and meeting contractual obligations. These requirements call for a set of safety and security properties on the infrastructure to be met and preserved over time, which can require the allocation of considerable resources (human, technical, organizational, *etc.*). To comply with legal and industrial regulations related to such properties, companies resort to internal and external audits. The former mobilize the company's internal resources, but may lack detail or objectivity. The latter are carried out by auditing firms leading to the issuance of certifications after a review process that is often very costly.

Whether audited or not, a good knowledge of the infrastructure is necessary to understand the interactions at work and better assess the risks to which the company is exposed. While small businesses with only a few departments may have employees with a good grasp of the big picture, this is hardly the case in large companies, in which each employee focuses on their own area of expertise (often a small part of the overall infrastructure). By interacting with these infrastructures on a daily basis, through specialized tools and business knowledge, employees develop their own local perspective of

the company, specific to their domain (with sometimes wrong assumptions on other domains). These perspectives can be of a very high level, notably for decision-makers who need to have a clear vision of the company as a whole, as they can be of a very low level, notably for technicians qualified in very specific areas.

Because of the diversity of IT infrastructures, many abstractions are used to enable their standardization and regulation. As a result, safety and security properties are often expressed using a generic, high level terminology. Verifying that these properties are satisfied means identifying a number of (often lower level) perimeters in which they apply, and defining evaluation criteria. Such perimeters may cover several business domains and call on the knowledge of many different employees, requiring the use of a common vocabulary to share information. In a way, assessing properties across an entire infrastructure consists in federating heterogeneous knowledge into an *ad hoc* formalism bridging the gap between the abstract and the concrete views.

I.1.1 What exactly is an IT infrastructure?

When we talk about IT infrastructures, we are considering a whole range of areas, both technical and non-technical. To set the scene for this introduction, we can draw on the definition from the ITIL 4 Foundation [ITIL19]: an IT infrastructure is "[a]ll of the hardware, software, networks, and facilities that are required to develop, test, deliver, monitor, manage, and support IT services." What usually comes to mind to illustrate this definition are servers and switches for the hardware side, operating systems and specialized applications for the software side, and cables and fibers connected between devices for the network side.

But we cannot reduce IT infrastructures to these elements. A server needs electricity and cooling to operate and is typically placed in a rack. The choice of a particular operating system is the consequence of a need and the cause of other subsequent choices. Network links, say between a company's subsidiaries in Paris and Los Angeles, pass through a set of exchange points outside the company's control. Rather than simply compiling an inventory of what the company owns or manages to characterize its infrastructure, we need to look at the entire environment that enables these assets to operate. A view of an infrastructure and its environment is given in figure I.1.

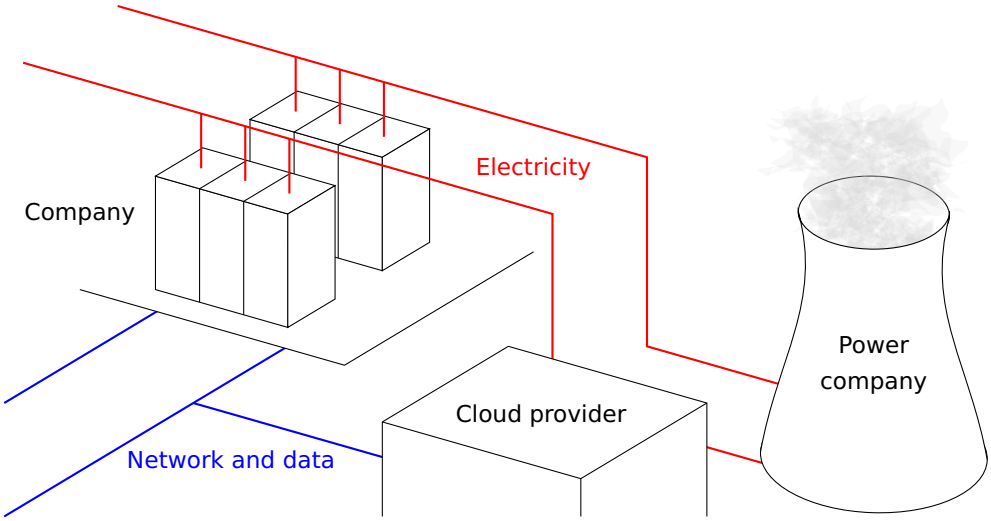


Figure I.1: Infrastructures and their environment

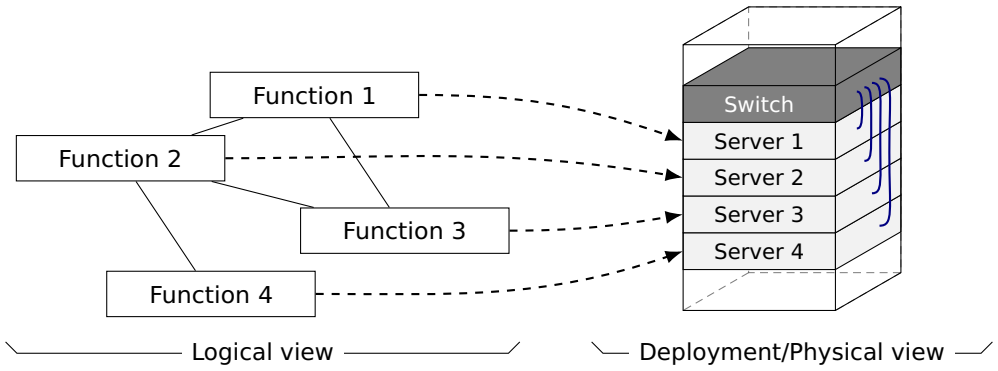


Figure I.2: Traditional IT infrastructure, where the deployment view generally corresponds to the physical view

Leaving aside any software aspect, most IT infrastructures are an assembly of numerous generic components, which specialize according to their use. Where an aircraft, a car, or even a factory is made up of hundreds and thousands of specialized components, the essence of hardware IT infrastructures can be summed up as what can compute, what can store and what can carry information. At a lower level also appears what can supply energy, what can regulate temperature and what can physically host the hardware, as previously mentioned. What sets IT infrastructures apart from others is that most of these components are freely interchangeable, thanks to the standardization of their interfaces over time. And over time have these infrastructures undergone major paradigm shifts.

I.1.2 IT infrastructures then and now

IT infrastructures have experienced significant changes over the past few decades. During the early days of business computing, mainframes were prevalent due to their large storage and processing capabilities. Despite their advantages, these systems had a high cost, took up a substantial amount of physical space, and provided limited interoperability between different hardware and software systems due to their proprietary nature.

The introduction of personal computers and servers in the 1980s and 1990s made computational resources more accessible to people. One critical aspect of this era was the introduction of standardized component interfaces and normalized architectures. Standardization made it possible for hardware components from different vendors to communicate with each other seamlessly and for software to run on many types of hardware. This period marked a shift from the traditionally vertical approach to infrastructures, with a few large machines having abundant resources, to a horizontal approach, with many machines having fewer resources.

In traditional settings, properties such as location, quantity, allocated resources and network were quite stable over time. Services did not “move” from one server room to another, scaling was not an automated process, hardware for “always-on” services was not frequently replaced, and network topology did not significantly change. To illustrate such a traditional infrastructure, a simplified deployment where functional components are directly mapped to physical servers is shown on figure I.2.

In the early 21st century, the Internet gained in importance and demand for computing resources exploded, with datacenters housing several thousands of servers. The increasing load on infrastructures led to a rethinking of their architecture, marking a transition from static to dynamic systems. These changes

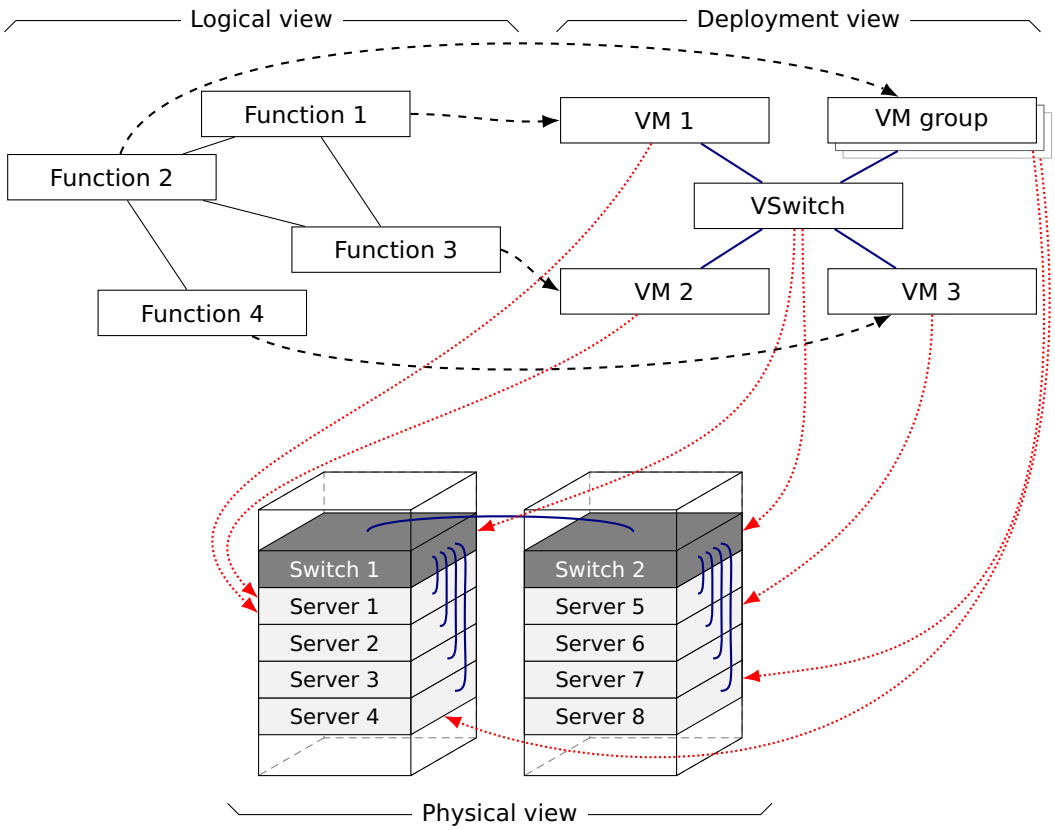


Figure I.3: Modern cloud infrastructure, where there is a clear separation between the virtual and the physical infrastructures

were driven by the democratization of cloud computing paradigms, pushing towards the abstraction of physical aspects to focus on the functional requirements. The introduction of these paradigms allowed for on-demand provisioning and reconfiguration of resources, leading to unprecedented flexibility and adaptability.

In contrast to traditional infrastructures, modern virtualized infrastructures can only guarantee the stability factors we mentioned previously for so long, sometimes hours or even minutes. The tight coupling between the logical and the physical layers does not exist anymore, as shown on figure I.3 with an added, virtual, “deployment layer”.

Pushed to the extreme, these abstractions have given rise to the concepts of Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS), which we summarize as XaaS. They correspond cumulatively to the abstraction of hardware, network, storage and virtualization (IaaS), of basic software stacks (PaaS) and of complete applications (SaaS), as shown on figure I.4. These XaaS solutions often entail relocating resources to so-called cloud providers, separating technical expertise from business expertise and outsourcing the former to external companies.

I.1.3 IT professions

Much like we structure communication systems in layers from the lower, physical levels, to the higher, functional levels (the famous OSI layers), people working with IT infrastructures have activities that we can group in low- and high-level views. These professions deal with very different aspects of

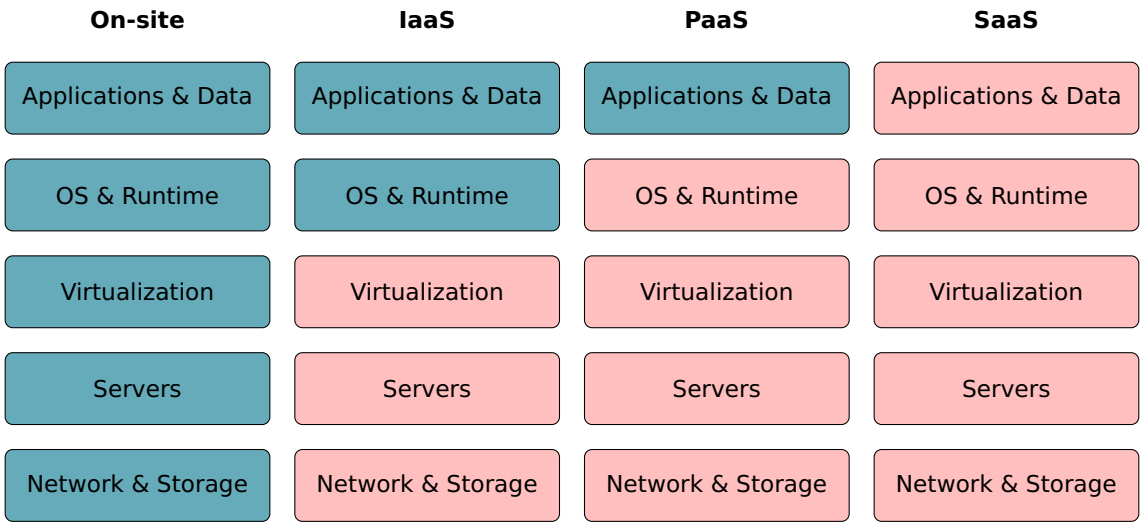


Figure I.4: Comparison between on-site and XaaS strategies
Legend. ■ Managed by the company, ■ Managed by the service provider.

infrastructures, and therefore carry different responsibilities and require tools and representations adapted to their specific view. It is important not to confuse layers and views at this point: while layers can be seen as distinct strata, views can overlap. For example, a company's financial department, while featuring what qualifies to us as high-level professions, is involved in many other views, from low-level hardware aspects (notably by planning equipment budgets), to high-level support functions (notably by managing the company's payroll).

Hardware

IT infrastructures are built on considerable physical foundations. At this level, datacenter employees have a good (local) vision of the company's physical infrastructure (servers, network equipment, cables...), but do not necessarily have a good understanding of the higher-level infrastructures that depend on it. Data center technicians are responsible for mounting equipment, replacing faulty parts, and managing physical connections to the electrical and various logical networks.

At this level, we can also find non-IT professions in the HVAC (heating, ventilation, air conditioning) and electrical sectors, both critical low-level elements of any datacenter.

Backbone

On top of these physical considerations lie the logical aspects of IT infrastructures. It would be impossible to design a data center without good storage management, and even less so without solid network foundations. This is where network and storage engineers come in. Network technicians are responsible for the company's network infrastructure. Storage technicians are responsible for commissioning resilient, high-performance storage solutions, by carefully designing distributed disk clusters. They have both a higher-level perspective, since they manage an infrastructure view dependent on the network, and a lower-level view, since they are familiar with the physical characteristics of the storage infrastructure.

Logical foundations

Above storage and network lie what we call the logical foundations. In so-called "on-premises" infrastructures, systems technicians install and maintain operating systems and runtime environments. In IaaS infrastructures, where all management of cloud foundations is delegated to an external party, their task is to manage the various virtualized resources. At this level, the details of the lower views are usually abstracted.

Finally, at the top of the chain are the employees who manage the applications installed on the end systems, that are used within the company, or even sold to clients. All these professions provide a multifaceted picture of IT infrastructures, which we want to exploit.

I.2 Problem statement

From this introduction emerges an important factor: the *knowledge* of infrastructures. From facts ("this element works as such") to beliefs ("this element seems to work as such") to requirements ("this element must work as such"), every stakeholder draws their own picture of the infrastructure. But this picture is at best limited, at worst false: facts are not always tangible, beliefs are not always correct, and requirements are not always met. In this dissertation, we defend the position that only through the federation of knowledge we can derive meaning from each others' views. We built upon this premise to explore the three lines of research we present here.

"How does updating this network router affect our safety and security?" IT infrastructures are not implemented without a goal in mind; they respond to requirements laid out in functional specifications. In addition to these, external regulatory bodies impose their own requirements, such as laws or standards, which usually limit flexibility and impose operating constraints on companies. Risk assessment on such infrastructures is a difficult exercise, due to their sizes, their inherently dynamic structure, and the semantic gap which can be considerable between the expression of requirements and field reality. Part of our work focuses on how to simplify, systematize and automate risk analysis, to increase confidence in infrastructures and reduce the cost of their audit and certification.

"How can we configure this application to fulfill our business needs?" Companies make extensive use of commercially available off-the-shelf components in their IT infrastructures. To best meet their needs, these components can offer configuration options that alter their default operation. Correctly configuring such components can be a challenging task and can lead to unexpected behaviors. When done wrong, *i.e.* in case of human configuration error, safety and security properties may no longer hold and potentially impactful edge conditions can arise during their operation. To better predict such behavior and help configure components more safely, we have explored verification of formal properties on them.

"Given our objectives, is it wise to make this architectural change?" IT infrastructures span across many business domains, from supporting corporate functions to customer services. A small change in one area can have far-reaching consequences in others; evolving needs and resources call for a good knowledge of the underlying infrastructure. This knowledge is disseminated throughout the company, but the presence of large teams and numerous domains leads to its fragmentation. Ultimately,

this can result in a poor understanding of the close relationships between infrastructure components, and require time-consuming meetings to align everyone's views. We have focused on the industrialization and scaling-up of our approach in highly heterogeneous domains to simplify cooperation through infrastructure modeling guidelines.

I.3 Contributions and outline

In this dissertation, we explore in breadth the topics related to risk management for IT infrastructures. Along our journey through the risk management process, which we set out to formalize throughout this dissertation, we attempt to connect various technical and scientific communities, leading to several productions and publications.

In chapter II, we present the state of the art of the subjects we address in this dissertation, which is further refined in each of the chapters developing our thesis. We provide a detailed reader's guide in chapter III, setting out the big picture of our work and presenting these developments.

Starting from an existing infrastructure, we explore different risk taxonomies in chapter IV and focus on MITRE's CAPEC, CVE and CWE projects, which we translate into ontologies in section IV.2.3. These ontologies allow us to better classify risk when assessing concrete software infrastructures. To help with the assessment phase, we provide guidelines for performing risk analyses on modern IT infrastructures in section IV.4 and advocate more interoperable risk analyses in section IV.5 by proposing a composable model for open analyses.

We propose to further refine the risk assessment step of the risk management process in chapter V, by considering it through the prism of modeling and model checking. We describe in section V.2 through a case study a number of technical and formal models, which we link in section V.3, where we use a trial-and-error approach to highlight the difficulties that arise when verifying concrete infrastructures. This study allows us to draw guidelines for IT infrastructure modeling and the automation of the whole process, discussed in section V.4.

To support this automation, we introduce in chapter VI an infrastructure description langage, CL/I, which represents the most important contribution of our work. CL/I acts as a pivot language, interacting with a variety of data sources within a company, as well as with formal provers and verifiers, and aims to systematize the use of formal methods. We present in section VI.3 the compilation of our language into Z3 scripts, and give two case studies in section VI.4 showing the use of the language and its integration.

We discuss in chapter VII the implementation of our approach in an industrial context with numerous stakeholders in a variety of roles and responsibilities. We introduce a component- and responsibility-oriented metamodel for IT infrastructures in section VII.1 to serve as a framework for our study and provide guidelines for collaborative enterprise modeling in sections VII.3 and VII.4. We illustrate this approach with a case study in section VII.5.

Finally chapter VIII summarizes our work and concludes this dissertation.

I.4 Funding

This PhD was conducted under a French *Cifre* partnership between the Crédit Mutuel Arkéa, a French bancassurance group, and IMT Atlantique, a leading French *Grande École*.

Chapter II

State of the Art

Contents

II.1 Managing risk in IT infrastructures . . . 9
II.1.1 Requirements . . . 11
II.1.2 Environment . . . 12
II.1.3 Approaches to risk . . . 13
II.1.4 Our position . . . 14
II.2 Modeling and checking infrastructures . . . 14
II.2.1 Infrastructure modeling . . . 16
II.2.2 Model checking . . . 16
II.2.3 Our position . . . 18
II.3 IT Infrastructures dynamics . . . 18
II.3.1 Infrastructure life cycle . . . 18
II.3.2 Deployment . . . 19
II.3.3 Monitoring . . . 20
II.3.4 Our position . . . 21
II.4 Conclusion . . . 21

In this dissertation, we cover a wide range of topics related to risk in IT infrastructures. Before presenting our study, this chapter introduces the work that has shaped our areas of interest. First, we give an overview of the literature on risk management. We then move on to modeling and model checking and their practical application to IT infrastructure. We finally consider the dynamics of IT infrastructures before concluding this chapter. This state of the art is further specialized and refined throughout the chapters of this dissertation.

II.1 Managing risk in IT infrastructures

Despite all the abstractions from physical reality that we discussed in the introduction, IT infrastructures remain subject to the constraints of the physical world. As a result, and because of the potentially high value of the activities they support, these infrastructures are inherently subject to risk. Traditionally, we distinguish between two main categories of risk [Burns92]: security risk, which covers acts of human malice, and safety risk, which encompasses everything else (natural causes, accidents, *etc.*). While both

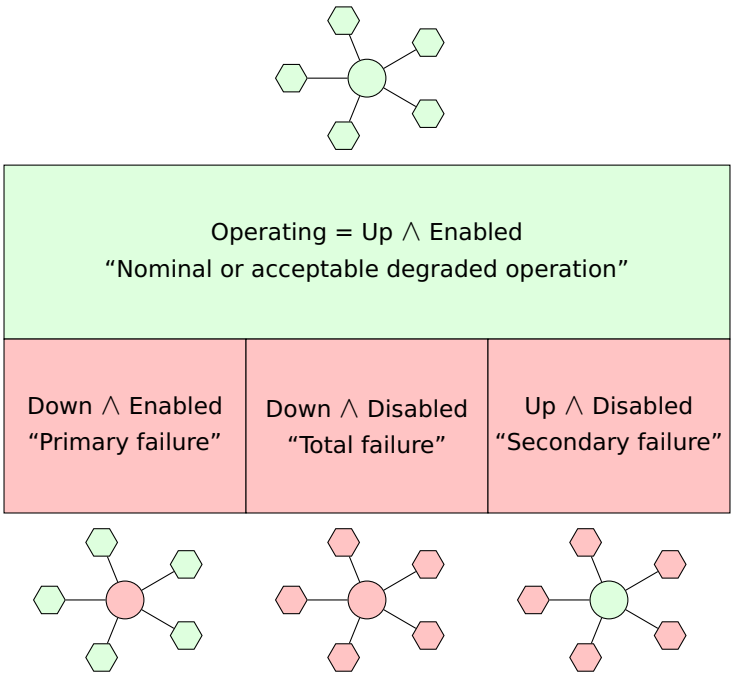


Figure II.1: Different states for systems, according to IEC 60050-192
Legend. □ Up, ▭ Down, ○ Studied component, ⬡ Dependency

categories of risk are often considered separately, the literature shows efforts to combine and study these within a common framework [Amundrud17; Boustras20], albeit with little practical implementation. In this dissertation, we take a generic view of risk, drawing lessons from this traditional separation.

When working in IT teams, it is not uncommon to consider questions such as “is the service up?”, but the precise definition of a functioning service needs to be elaborated. According to the [IEC 60050]-192, which presents a common terminology in the field of dependability, a system, say an online store, may be in an *operating* or a *non-operating* state, depending on whether it is performing as required or not. An individual component inside the system, say this online store’s web engine, may either be in an *up* or *down* state, whether it is able, modulo external dependencies, say the store’s databases, to perform as required or not. Independently of the *up*/*down* state, these external dependencies put the system in a so-called *enabled* state if all of them are performing as required (if the databases are in an *operating state*), and in a *disabled* state otherwise. Figure II.1 summarizes the various possible situations, with a system (the online store in our example) made of a component (the web engine in our example) and five dependencies (the databases in our example). The notion of a functioning or non-functioning service thus hides a whole chain of dependencies likely to cause the service to become inoperative.

From these simple definitions come two important aspects of the study of infrastructures which we discuss in this section:

- Systems are built according to a set of requirements, against which their proper operation is measured;
- There is potentially a strong coupling between systems and their environment, as a malfunction on an external dependencies can affect the operating state of the whole system.

The study of these aspects sheds light on the first issue of our problem statement, concerning the close links between requirements and risks.

II.1.1 Requirements

Ensuring the safety and security of complex IT infrastructures has gained significant traction in the research community over the years [Uchenna17; Maniah22]. Such a complexity often calls for the use of advanced techniques and tools to address deviations and ensure conformance to desired requirements[1]. In [Somers23.1], we categorize these requirements as such:

- External requirements, *i.e.* requirements that are imposed by external bodies or by the companies themselves when interfacing with the outside:
  - Legal and regulatory, with notably the *General Data Protection Regulation* [GDPR], governing the use and collection of personal information or the *Payment Card Industry Data Security Standard* [PCI DSS], to reduce credit card fraud;
  - Contractual, with notably payment processor requirements [Mastercard21; Visa22], applying financial penalties in case of non-compliance with quality standards, or service-level agreements (SLAs) negociated by the company with its business clients [He18].
- Internal requirements, *i.e.* requirements that are self-imposed by the company:
  - Technical, with notably architectural decisions;
  - Functional, with notably performance criteria.

These requirements have a direct impact on the risk of infrastructures and the way they are designed, which is one of the main focuses of this dissertation.

Requirements can be expressed in a number of ways, which we explore in more detail in section IV.2, but we find them most often in technical specifications and legal notices, in a textual form. A broad range of propositions to describe requirements in a more formal way are found in the literature with notably ArchCNL and RQCODE.

ArchCNL [Schröder18] is a controlled natural language (CNL) for describing architectural rules and checking source code conformance to such rules. Although the use of a CNL enables requirements to be expressed in a language close to how they are usually expressed (and thus be "self-documenting"), two obstacles stand out. First, some effort is required to translate existing requirements in a controlled vocabulary, which may lack expressiveness. Second, such a high-level approach requires writing *ad-hoc* mappings between the controlled vocabulary and the code.

Recent initiatives such as Requirements as Code (RQCODE) [Nigmatullin23], which are closer to the developer, propose to support automated conformance checking by writing requirements in a programmatic way. While such approaches involve a greater rewriting effort for requirements, they can benefit from a more comprehensive tooling across various technical communities and thus greater industrial acceptance.

To answer the industrial need around our work, we have chosen to stay in line with this desire to automate infrastructure risk management through code in this dissertation.

[1] The term "rule" is sometimes used in the literature.

II.1.2 Environment

IT infrastructures do not exist independently of a context. For a hardware infrastructure, the context corresponds for example to the electrical grids on which it depends and its geographical location. For a software infrastructure, it corresponds for example to its execution environment and its software dependencies.

Such a context creates strong interdependencies between infrastructures, that are represented in the literature as multiplex networks [DeDomenico13] and interdependent graphs. For [Rinaldi01], the interdependencies can be of four types:

- Informational, if there is a logical reliance on information transfer between infrastructures;
- Physical, if there is a physical reliance on material flow (electricity, water) from one infrastructure to another;
- Geographical, if a local environmental event can affect components across several infrastructures due to physical proximity;
- Procedural, for other kinds of interdependencies.

A complete risk analysis therefore requires an in-depth study of each of these interdependencies, which we find is lacking in the literature, where these types of interdependencies are often considered separately.

Informational

Software and its execution environment present strongly coupled risks. If a piece of software has a security flaw, an attacker could modify its environment, which in turn could have an effect on other systems. At the same time, software behavior can be indirectly modified by an attack on its environment [Vaidya19]. As IT gets more and more ubiquitous, we observe a multiplication of entry points, inducing a much larger attack surface [Hannousse21], which requires proper attention by companies. Finally, the opening up of the software industry to open source collaboration has intensified attacks on the supply chain, notably on package repositories [Ohm20]. Our work argues that a better understanding of infrastructures and a greater involvement of all stakeholders in their mapping and modeling enables to better apprehend these new attack vectors.

Physical

A failure at a critical point in an infrastructure, such as the electrical grid, can lead to a cascade of failures in many other components [Buldyrev10], requiring proper monitoring and mitigation strategies [Ten10]. But physical interdependency involves more than electricity provision. Within an IT infrastructure, material flows can be very diverse, from sourcing technical equipment [Voas21] to supplying fluids to cooling systems.

Geographical

The geographical location of hardware and buildings has a major impact on infrastructures at every scale. On an industrial site level, proximity to so-called Seveso establishments or the presence of a seismic risk call for specific infrastructure design and management measures [Esen22]. On a smaller

level, very localized events such as ambient vibrations can have serious consequences on storage performance [Turner10]. Along with physical interdependencies, geographical interdependencies demonstrate a strong coupling between events of a physical nature and measurable degradation at a logical level.

Procedural

Finally, a number of other factors can contribute to infrastructure interdependencies, including economic considerations and executive decisions. From individual businesses to intergovernmental organizations, many entities take part in the decisional and technical chains, as extensively presented by [Hasan15]. These stakeholders assume a wide range of responsibilities that are important to consider, which we integrate into our approach in chapter VII.

II.1.3 Approaches to risk

Risk analysis is a discipline that developed extensively during the 20th century in the aerospace and automotive industries. Whether in these fields or in critical infrastructures, failures can lead to fatalities [Yates14]. Risk analysis is therefore a discipline that should not be taken lightly.

Generic methods such as FTA and FMEA [ARP4761], HAZOP [IEC 61882] or STPA [Leveson12] have been devised to increase confidence in infrastructures and identify risk scenarios. These methods follow rigorous, systematic processes, but are fairly low-level and hardware-oriented, due to their original target fields. We discuss their applicability to IT infrastructures in section IV.3.

In parallel, various industries have been developing more specific safety and security regulations to guide such risk analyses. In aerospace, [AS9100D] (Quality Management Systems – Requirements for Aviation, Space, and Defense Organizations) describes quality standards to increase the safety of the supply chain and final products. In the automotive industry, [ISO 26262] (Road vehicles – Functional safety) defines so-called Automotive Safety Integrity Levels (ASIL) to classify risks and treat them with different degrees of care according to their criticality for human life. Finally, to return to our main focus, [ISO/IEC 27005] (Information security, cybersecurity and privacy protection — Guidance on managing information security risks) provides guidance on how to assess and treat risk on information systems. These various regulations, while not incompatible, have differences in nomenclature that limit their interoperability. In our work, we have sought to explore methods from a variety of fields in order to gain insight into their applicability to IT infrastructures.

For these infrastructures in particular, many integrated frameworks offer a structured way to analyze and manage risks related to information security, some of them implementing ISO/IEC 27005, with specific nuances and areas of focus. [Abbass15] gives a detailed comparison between [OCTAVE], [EBIOS-RM] (on which we have chosen to focus in this dissertation), [MEHARI], [CRAMM] and [CORAS] and discusses their advantages and drawbacks. The article presents four strategies to risk mitigation, stressing the fact that eliminating the risk is not the only solution:

- Risk avoidance, where the source of the risk is eliminated;
- Risk limitation, where the exposure to the risk is reduced;
- Risk transfer, where the responsibility for the risk is delegated to an external party;
- Risk acceptance, where the risk is not dealt with any further.

We discuss the acceptability of such a residual risk (*i.e.* risk that is not eliminated) in section IV.4.3.

In a bid to integrate risk management seamlessly into development and operational processes, the DevSecOps movement has gained attention in the scientific community [Myrbakken17]. It aims to embed security practices within the DevOps methodology, ensuring that risk assessment isn't an afterthought but an integral part of the IT development life cycle. But this kind of modern approach is not enough to guarantee system safety and security, and [Neumann19] notes in particular that:

- It is impossible to build trustworthy applications on flawed systems; even if the application is designed to high standards, the risk analysis must not neglect the execution environment;
- Properties, sometimes adverse, appear when composing systems together; risk analysis must be carried out not only on individual components, but also at their interface;
- Relocating an infrastructure on a public cloud does not magically make it more safe and secure; we need to question our trust in third parties.

II.1.4 Our position

The state of the art in risk management is very extensive in the historical fields in which the discipline has been developed. However, our experience of risk analysis on complex IT infrastructures has shown us that most of these methods are not adapted to our area of study. First of all, it appears that the traditional, textual and verbose format of risk reference frameworks runs counter to the modern automation principles advocated by IT practitioners. This idea is accurately captured by Nancy Leveson, who claims that "[h]uman error is a symptom of a system that needs to be redesigned" [Leveson19]. Secondly, the multi-level nature of IT infrastructures requires very careful consideration of the risks associated with interdependencies, which specialised frameworks are not designed to capture. In our work, we advocate a more collaborative approach to risk management that takes account of the specific characteristics of IT infrastructures. This collaboration involves a thorough modeling of infrastructures, which we discuss next.

II.2 Modeling and checking infrastructures

As we mentioned in the introduction of this dissertation, numerous professions are involved in IT infrastructures, each with their own tools, languages and representations, from hardware to software, including networks and processes. A datacenter can be described by rack diagrams, illustrating the layout of servers and network components. A software can be represented using UML diagrams [UML], to show its structure and the different interactions at work, or can be described by its code. A network topology can be seen as mathematical objects [Park00], described by switch configurations, or as code in *Software-Defined Networking* (SDN) [Masoudi16], using technical tools as well as formally defined frameworks such as Netkat [Anderson14]. However, most of these tools are not designed to interact with one another. Asking simple questions such as "what happens if I unplug this cable?" is often enough to highlight this problem [Neville22]. In our work, we attempt to use the knowledge distributed throughout these representations to derive new risk-related properties and help configure and maintain systems. In addition to our first issue, modeling and model checking provide crucial insight on the second issue of our problem statement, regarding the conformance of infrastructures to requirements.

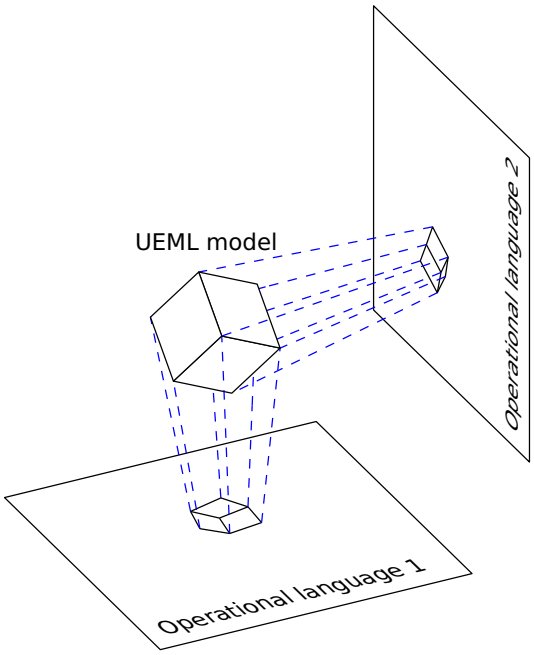


Figure II.2: UEML object projected as two models onto two operational languages

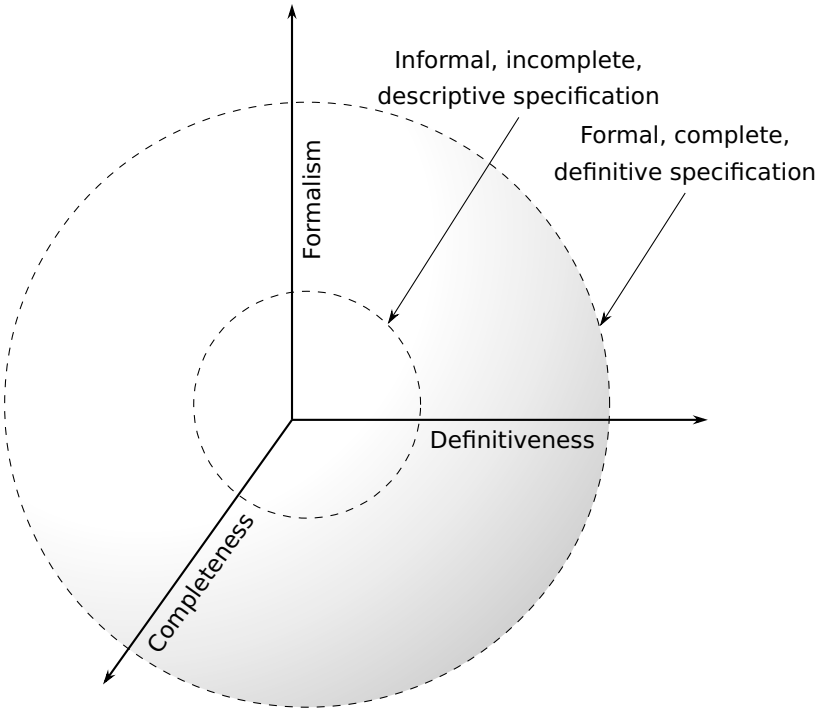


Figure II.3: The three dimensions of specification according to RM-ODP

II.2.1 Infrastructure modeling

All the infrastructure representations we have mentioned above can be considered as models [Sandkuhl18], that is, abstractions of systems for pragmatic use. In the domain of Enterprise Modeling [Vernadat20], an approach to unify these models in a kind of modeling Esperanto [Vallespir18] has been devised in order to alleviate the problem of interaction between models: UEMLs [Vernadat02] (for Unified Enterprise Modelling Language). The idea was rather bold: a UEML model was a kind of "super-model" that could be projected onto so-called operational languages (where processing would actually happen) as models in these languages. We have represented the concept on figure II.2. However, due to a lack of technical adoption, the framework has outlived its purpose. In this dissertation, we defend a "rival" approach: model federation [Golra16], where rather than building operational models from a big holistic model, we start with small operational models and build links between them to derive a bigger picture.

Other approaches in enterprise modeling such as [Archimate] and [TOGAF] focus more on the processes around IT infrastructures and the people who work with them, to help with high-level decision making. They offer a comprehensive approach for designing, planning, implementing, and governing enterprise IT architecture, enhancing alignment with business goals, which we discuss in chapter VII.

Infrastructure models come in a wide variety of types. According to [RM-ODP], they[2] vary according to three dimensions, represented on figure II.3:

- Formalism, whether the specification is supported by formal methods and mathematical foundations or not,
- Completeness, whether the specification deals with whole systems or only parts,
- Definitiveness, whether the specification is descriptive, *i.e.* leaving room for interpretation, or "definitive".

In this dissertation, we seek to combine formal and informal models and we deduce that completeness is not always desirable in chapter V, since excessive precision hinders formal analyses. Our practice of IT infrastructure modeling leads us to argue that that we should not always establish definitive models and rather leave room for the unknown, which we discuss in chapter VII. We further refine this categorization of models in section V.2. As the formal dimension of models often makes it possible to take advantage of verification tools [Vallespir18], this is what we explore in the rest of this section.

II.2.2 Model checking

Formal models can describe possible system behavior in an unambiguous, mathematical way. Model checking [Clarke09; Agha18] formally guarantees that a specification (often expressed in temporal logic) is actually satisfied by the designed system. The tool that performs model checking is called a model checker. It examines (traditionally all) system states and checks whether or not they satisfy specified properties. If a state violates a property, the model checker can provide a counter-example, *i.e.* a trace from an initial state to the error state. Figure II.4 represents the process, as described by [Baier08].

Model checking is implemented in numerous tools, among which we can cite SPIN [Holzmann97], OBP [Dhaussy12] and UPPAAL [Larsen97], the latter of which we use extensively in sections V.2 and V.3. In our work, we have also been particularly interested in SMT solvers [Monniaux16], able to check the satisfiability of first-order logic formulae, and have exploited Z3 [deMoura08] in section VI.3. Over

[2]The standard talks about specifications, but the concepts can be extended to models.

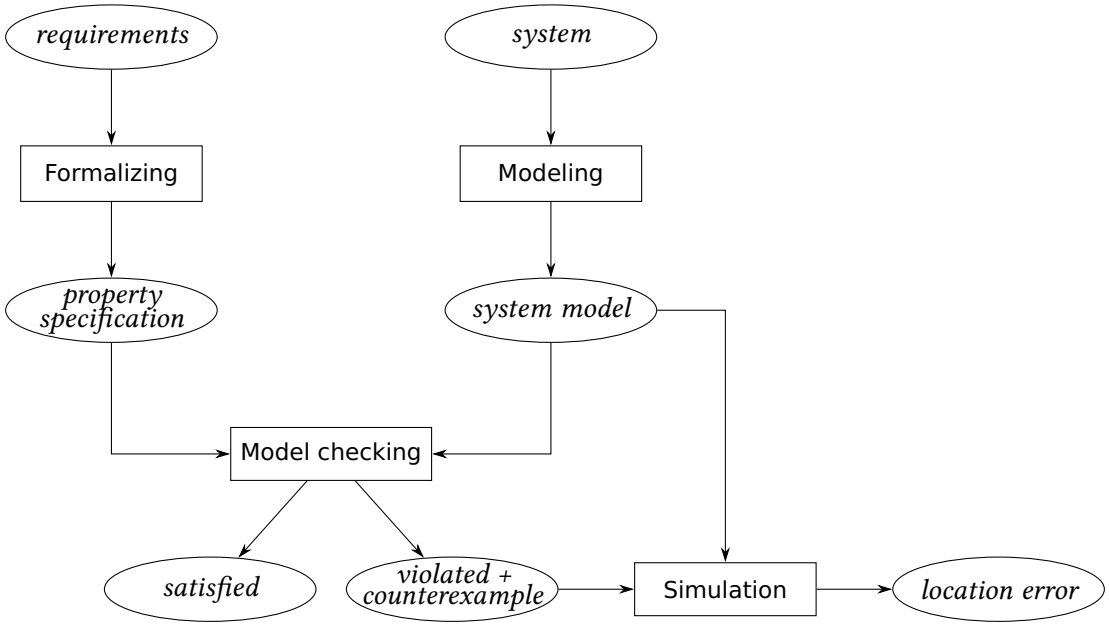


Figure II.4: Model checking process, according to [Baier08]

time, the tools have greatly improved, enabling them to be used on larger scales and even in corporate contexts. For example, Amazon Web Services engineers have been defining formal specifications for their complex mission-critical systems since 2011 [Newcombe14]. The adoption of formal methods in business is nevertheless not the norm [Lecomte17], and we seek in this dissertation to open up avenues for simplifying entry into the field. We also try to minimize the semantic gap between reality, formal methods and formal properties to be checked, by discussing automation strategies for modeling and expressing such properties in section V.4.

The world of verification is however not limited to formal models, with techniques such as property-based testing [Hughes10] and even formal source code verification [Kirchner15; Leroy09]. Such techniques, which can directly adapt to the workflows of practitioners, can provide the automation strategies we want to advocate. [Caracciolo15] notes however that most stakeholders adopt non-automated techniques or avoid testing completely, because of:

- Fragmented tool support: most tools are highly specialized and can only handle specific architectural constraints;
- Tool incompatibility: tools operate according to their own technical and theoretical assumptions and lack a common specification;
- Steep learning curve: many tools require a considerable amount of time to be properly used

Caracciolo proposes in this article to formally verify informal models with his rule specification language, Dictō, but we have found the approach to be too low-level for most practitioners. The hindsight we have now on UEML [Vernadat20] leads us to believe that a tool that seeks unification at all costs risks meeting ultimately the same fate.

II.2.3 Our position

Within a company, every employee produces models, whether voluntarily or not, notably by representing systems or using technical software. It seems unreasonable to reduce the modeling effort to a few architects or modeling experts when the company's fields of study are so diverse and numerous. We argue that this diversity of models should be brought together to provide a more complete overall picture of a company and its infrastructure, enabling more comprehensive studies to be carried out across several business domains. Model verification remains a discipline that is quite closed to non-experts; we think however that the richness of business models can benefit formal studies. In our work, we promote a wider use of formal methods and model checking by facilitating their interoperability with technical models. However, formal models are often static, checked once before a system is deployed on a concrete infrastructure. In dynamic systems that require monitoring during their lifetime, this approach is not always appropriate and this is the subject of the last section of this chapter.

II.3 IT Infrastructures dynamics

The presence of IT systems in critical infrastructure has grown steadily over the past few decades [Merabti11]. This growth has led to unprecedented opportunities across numerous business sectors, but it has also brought its own share of risks. Complex virtualized infrastructures are now commonplace, with frequent topological changes making system interactions increasingly difficult to study [Neville22] and risk assessment more challenging [Lv18]. In this section, we study the life cycle of such infrastructures, then focus on their deployment, to finally study how they are monitored. This aspect of our state of the art supports the third issue of our problem statement concerning the evolution of infrastructures.

II.3.1 Infrastructure life cycle

If we set aside the issues surrounding their decommissioning, IT infrastructures follow a continuous cycle during their lifetime. The literature displays a number of so-called life cycles, some of which being described in detail in [Ruparelia10], but their main steps remain essentially similar. For software infrastructures, we traditionally identify six stages of this life cycle, which we can extend to IT infrastructures in general:

1. Requirements definition, to understand and document the needs and expectations of stakeholders and define what the system shall do;
2. System design, where diagrams, design documents, data models, and interface prototypes are produced;
3. Development, where the actual coding and implementation take place;
4. Testing, to check for correctness and validate the system;
5. Deployment, where the system is installed, configured and put into service;
6. Review and maintenance, to evaluate the system's operation and performance and gather feedback.

These steps, also known as the *development life cycle*, are depicted in figure II.5. Section II.1 deals with the requirements definition stage and section II.2 is focused on stages 2 to 4. In this section, we are particularly interested in the last two stages, which are examined in detail in chapter VI. Along this dissertation, we focus on each step of the process, in which we discuss the potential for risk analysis.

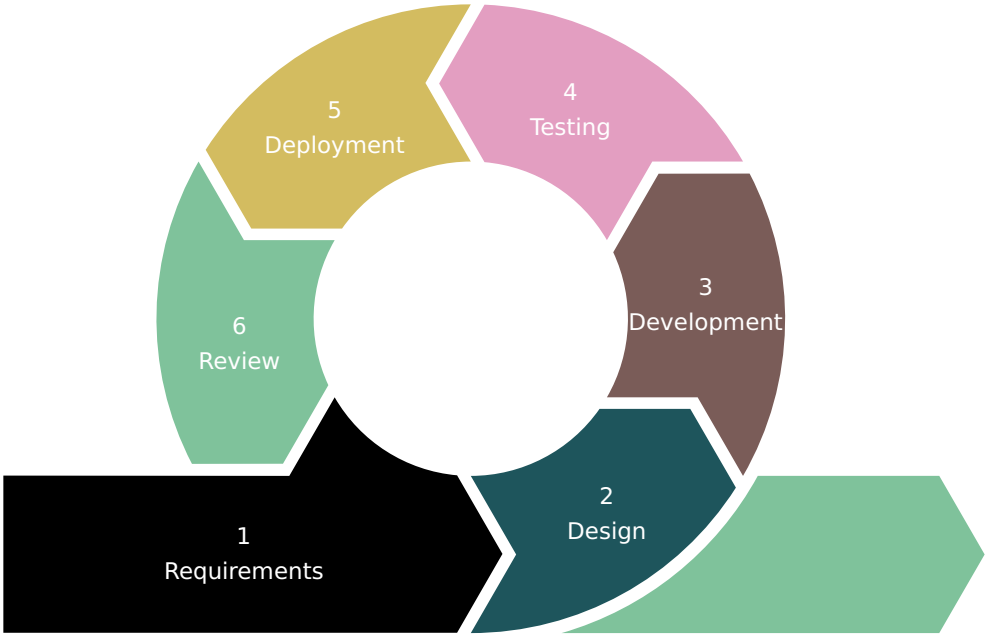


Figure II.5: Development life cycle

II.3.2 Deployment

After testing infrastructures comes their deployment. This step involves installing and configuring applications on systems that can host them. This process is time-consuming and error-prone, as it sometimes involves the proper orchestration of dozens of components. [Xu15] identifies the following issues with system configuration:

- Complexity: systems sometimes expose many configuration parameters, with non-trivial constraints and dependencies;
- Invisibility: without access to a system's implementation, it is sometimes impossible to understand the undocumented effects of configuration parameters;
- Dynamics: the configuration of systems may change over time (we explore this in more detail in section II.3.3)
- Bad design and implementation: little guidance on making configuration decisions are given to users and error messages can sometimes be unclear.

To automate the process, a number of tools and techniques are now available, from configuration management to Infrastructure as Code (IaC) [Kumara21]. These approaches are however not exempt from the issues mentioned above, and rather shift them from application configuration to infrastructure configuration. In particular, IaC code smells have been explored in detail in the literature [Rahman21; Schwarz18; Sharma16].

Configuration management systems (usually) adopt an imperative, step-by-step strategy to configure systems. Tools such as Ansible [Hochstein17] and Chef [Pandya22] allow to specify sequences of configuration steps to deploy and configure existing systems (mutable infrastructure). On the other hand, IaC approaches tend to follow more descriptive models. In tools like Terraform [Brikman19] and Pulumi [Campbell20], we describe the resources to be deployed and configured and a specialized agent automates deployment and configuration of new systems (immutable infrastructure).

These methods enable:

- Infrastructure reproducibility [Vaillancourt20], guaranteeing the validity of initial conditions for each deployment;
- Infrastructure versioning [Opdebeeck20], allowing better assignment of responsibilities and error identification.

The formalization of infrastructures using these approaches enables the use of more advanced verification tools than in traditional infrastructures. In the scientific community, these concepts have been further explored and formal deployment models such as Aeolus [DiCosmo14] or Madeus [Chardet18] have been developed to efficiently deploy software infrastructures and their dependencies. They are relevant approaches towards the automation of large-scale infrastructure deployment.

II.3.3 Monitoring

Ensuring infrastructure properties at a specific moment increases confidence in systems, but the rapid and frequent evolution of virtualized infrastructures raises questions about the long-term validity of these properties. Indeed, in such infrastructures, components and their dependencies move from a server to another and are frequently (and sometimes automatically) updated. Without going into the well-known complexity of managing dependencies [Burns16], adverse effects such as software aging, where safety invariants on memory are violated during the execution of a program over a long period of time, are described in the literature [Li02; Grottke08; Pietrantuono20].

Complex infrastructures therefore require continuous system checks during their execution: monitoring. The traditional way to monitor infrastructures, which is still the most commonly used nowadays, is for external observers to collect facts on these systems and compare them to predefined thresholds. Monitoring solutions are very varied, and the comparative literature on the subject is extensive [Grati15; daCunhaR16]. Approaches such as cloud orchestration or SDN enable automatic generation of checks from infrastructure descriptions. In these fields, research in change management has explored methods for analyzing security policy violations linked to configuration changes (called *configuration drift*). Systems like Weatherman [Bleikertz15] can monitor runtime changes in cloud infrastructures using graph transformations and verify properties on infrastructures using graph matching. In the realm of SDN, systems such as VeriFlow [Khurshid13] can check invariants on large-scale live network infrastructures. Both systems suffer however from two main drawbacks which are common for real-time verification in the literature:

- They require a common model on which checks are performed (graph structures, network *data plane* internal representations);
- They require modifications to the runtime systems to capture events.

The approach we defend in chapter VI is quite different from these, since we use a pivot language in which requirements, static models and dynamic observations of systems are represented, to delegate the verification of properties to external solvers.

Finally, a more recent approach, *chaos engineering* [Netflix10], consists in injecting faults into production systems and measuring their impact in real-life situations, which software testing does not allow. In the case of Netflix, this is implemented as a controlled environment testing, where 50 % of

clients go into an instrumented sane environment and the 50 % others go into an instrumented faulty environment to measure deviations [Jones17]. This approach is what actually inspired this doctoral project, but as do configurations in IT infrastructures, we can say that the idea drifted.

II.3.4 Our position

Formal verification of infrastructure properties is important to guarantee safety and security. However, there can be significant differences between models and real infrastructures. These differences may arise during the configuration and deployment of systems due to mismatches between the requirements and their implementation; they may also appear as a result of a drift since their deployment. We advocate an approach that moves away from the simple configuration-oriented description of systems, based on a formal description of requirements and its translation into configuration. In particular, we consider infrastructure monitoring as a special case of model checking and regard the generation of system checks as an artifact of the compilation of IaC models.

II.4 Conclusion

The state of the art that we have presented in this chapter, which we specialize throughout this dissertation, enables us to position our work with regard to the problem statement set out in the introduction. First, a good risk management of infrastructures is necessary to guarantee their proper operation and compliance with requirements. The literature mainly describes manual risk management processes, which are however not in phase with the automation principles of modern infrastructures. Then, risk management requires a better understanding of infrastructures, which we think can only happen through modeling and formal, automated verification of safety and security properties. However, the barrier to entry for formal methods can be high in areas where technical models are the norm. Finally, infrastructures and the requirements imposed on them evolve, and safety and security properties need to be maintained over time. However, there is no formalised approach combining requirements, configuration and real-time verification for studying IT infrastructures in the broad sense.

Chapter III

Reader's Guide

Contents

III.1 Approach 23
III.2 Progress 24
III.3 Big picture 24

Our research is conducted in an industrial banking context. In this sector, institutions are subject to a number of requirements, laid down by national, transgovernmental and sector-specific directives. These environments are critical to our daily lives, hosting sensitive customer data, whose safety and security must be ensured to preserve the integrity of the global system. In this context, our work focuses on risk as a whole.

III.1 Approach

Risk is a subject of study that spans across several scientific communities, and that can be approached in a multitude of ways. We can look at risk from the bottom up, studying technical infrastructures and working our way up to a more global vision. We can also consider it as something very high-level, organizational, which we decline downwards when we build infrastructures. It is under these different visions that we approach this dissertation, whose ideas have developed organically by attempting to link them around the concept of infrastructure modeling.

Our project aims to take a more proactive approach to incidentology, in order to better assess risk and predict sensitive events. Through our exploration of risk, we have involved ourselves in three scientific communities, in which our work contributes. These communities are:

C1. The risk management community, which focuses on the organizational and technical aspects of risk analysis and remediation;
C2. The formal methods community, which opens the door to more mathematical and theoretical tools for our approach to risk management and safety and security properties on infrastructures;
C3. The enterprise modeling community, which considers the enterprise as a whole, both in terms of infrastructure and of its stakeholders and their responsibilities, without which it could not exist.

Our work aims to bring together and build bridges between these communities. This approach makes sense in the context of our enterprise (C3), subject to numerous regulations (C1) and in which we are determined to implement a rigorous approach (C2) to risk. To this end, we have placed ourselves along the entire risk management process, exploring it broadly through the prism of the three communities.

III.2 Progress

In chapter IV, we begin our exploration by focusing on the risk management community, which we seek to bring to the frontier of formal methods. We first introduce the risk management process that we follow throughout our dissertation, and set out to provide a more formal vision of it. We propose to refine the first steps of this process, with the ultimate goal of this dissertation being its complete and comprehensive exploration throughout the other chapters.

In chapter V, we continue to refine this process by bringing modeling and verification from the formal methods world to risk management. We present a case study of a modest IT infrastructure and integrate our formal elements, while demonstrating the limits of formal methods applied to IT infrastructures.

Chapter VI seeks to strengthen the bridge between the first two communities we have built so far. It focuses on the links between safety and security requirements, infrastructure configuration and deployment, and infrastructure execution, using formal methods. To this end, we present a modeling language, CL/I, that connects infrastructure models and component configuration, while using verification tools to determine conformance criteria for these requirements.

Finally, chapter VII focuses on the corporate aspects studied by the third scientific community, detailing how the approach presented in our dissertation can be integrated into such a context. This integration defends the federation of knowledge and models, valued by the formal community, while taking a step back from the risk management process as a whole.

III.3 Big picture

Figure III.1 illustrates the contributions of each chapter to the three communities. The curious reader may wish to consult figure VIII.1 in the course of reading this dissertation, to better appreciate where each chapter fits into the overall risk management process. We also recall our position in this big picture through brief diagrams at the heading of each chapter developing our thesis.

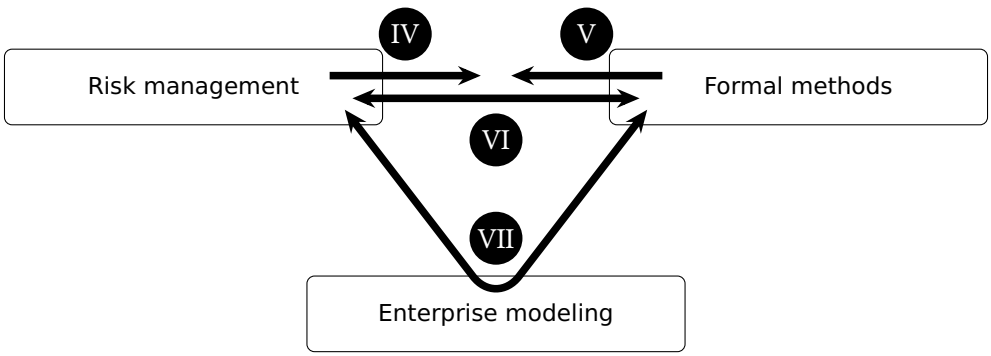
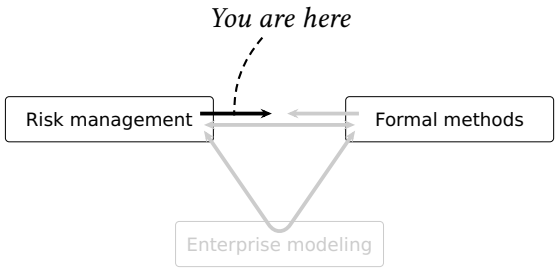


Figure III.1: Contributions of this dissertation to each scientific community
Legend. (x) Developed in chapter x, □⟶□ Bridge to, □⟷□ Bridge between

Chapter IV

Managing Risk in IT Infrastructures



> Laura Roslin — In this situation, you're putting your pilots at risk and you're exposing the entire fleet to possible attack every moment we stay here.
>
> We've been at risk of an attack since day one. The Cylons won't be missing their patrol for at least one more day. — William Adama
>
> Laura Roslin — Colonel Tigh, how much aviation fuel has been expended in this operation?
>
> Forty-three percent of reserves. — Saul Tigh
>
> Laura Roslin — Almost half. That's unacceptable.
>
> *Battlestar Galactica – You Can't Go Home Again*

Contents

IV.1 The risk cycle . . . 27
- IV.1.1 Formalism . . . 27
- IV.1.2 Properties . . . 29
- IV.1.3 Iteration . . . 30
- IV.1.4 Change . . . 31
- IV.1.5 Approach . . . 33

IV.2 Risk classification . . . 33
- IV.2.1 Taxonomy efforts . . . 33
- IV.2.2 The case of MITRE . . . 37
- IV.2.3 An ontology over MITRE . . . 38

IV.3 Risk analysis frameworks . . . 44
- IV.3.1 Traditional frameworks . . . 44
- IV.3.2 Modern initiatives and IT infrastructures . . . 46

IV.4 Risk assessment and tolerance criteria . . . 47
- IV.4.1 Analyzing parts . . . 47
- IV.4.2 Analyzing systems . . . 48
- IV.4.3 Closing the loop . . . 49

IV.5 Sharing analyses . . . 51
- IV.5.1 Building open analyses . . . 52
- IV.5.2 Composing analyses . . . 52

IV.6 Conclusion . . . 52

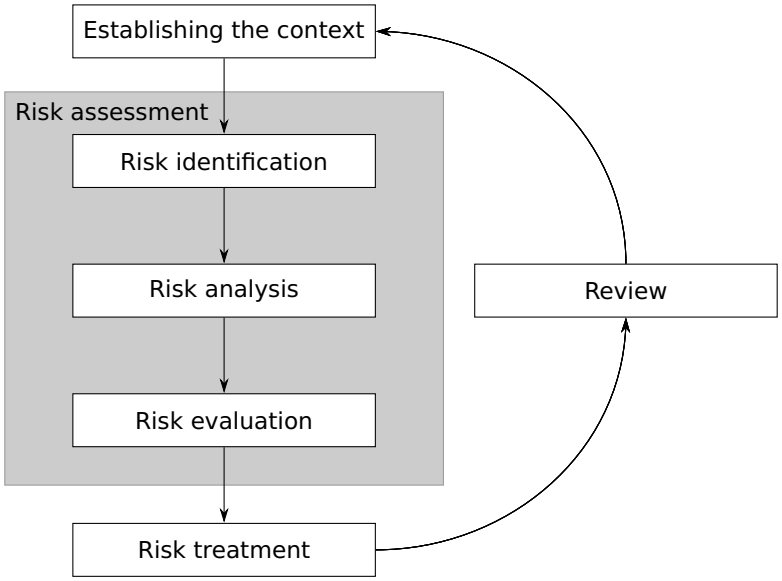


Figure IV.1: ISO 31000 iterative process

IT infrastructures, due to their pervasive nature in today's world, are subject to numerous regulatory standards and guidelines emanating from various entities across the globe. These entities, ranging from national agencies to international organizations and industry-specific bodies, define the areas of acceptability for IT infrastructure operations, and thus shape the practice of risk analysis.

When thinking about risk in IT infrastructures, the first thing that often comes to mind is the cyber threat, such as malware intrusions, distributed denial-of-service attacks or data breaches. However, the typology of risk is much wider, taking into account, in addition to cyber threats, security in the broader sense within datacenters (armed attacks, sabotage...), as well as functional safety of infrastructures (resilience, availability...), financial or even environmental risks. The practice of risk assessment shows that an in-depth analysis of infrastructures must carefully take account of this multidimensionality, without overlooking their operating environment.

The [ISO 31000] (*Risk management – Guidelines*) standard defines the risk management discipline as a five-step process:

1. Establishing the context
2. Risk identification
3. Risk analysis
4. Risk evaluation
5. Risk treatment

The first step defines the context of the study: *what* parts of the infrastructure are considered and *when*, *where* the system is, *who* is involved, and *why* and *how* the study is being done. The next three steps are called the "risk assessment" by the standard. The first one consists in identifying risks and their causes and impacts. Then, a qualitative and quantitative risk analysis is carried out to assess likelihoods, confidence levels and magnitude of consequences. The evaluation step compares the results and leads to decisions on the infrastructure. Finally, the risk treatment phase addresses risks by modifying or eliminating what led to it, attempting to reduce risk likelihoods and mitigating the consequences. The whole process is iterative, as depicted in figure IV.1

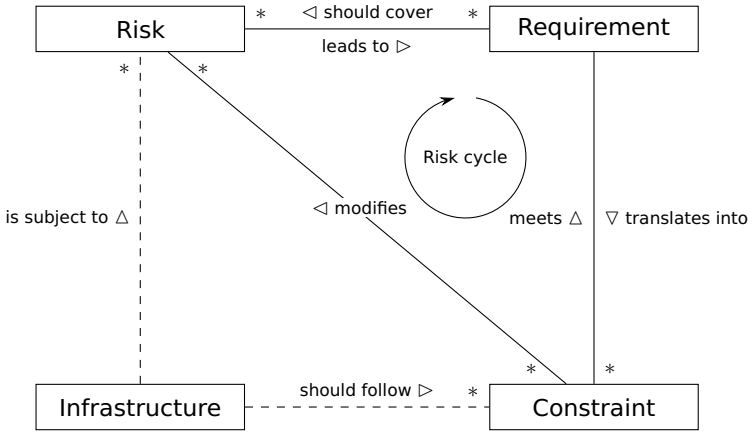


Figure IV.2: The risk cycle

In this chapter, we begin by introducing risk management in section IV.1 through a formalism that we further develop over the course of this dissertation. We then outline different ways of classifying risk in order to process it in a systematic way and propose an ontology adapted to our study in section IV.2. After that, we look at how different risk analysis frameworks can be specifically adapted to IT infrastructures and the challenges related to modern cloud infrastructures in section IV.3. We then provide guidelines for risk analyses and discuss risk tolerance in section IV.4. Next, we consider the possibility of sharing risk analyses using a common exchange format in section IV.5. Finally, we conclude this chapter in section IV.6.

IV.1 The risk cycle

Risk can manifest in many ways in IT infrastructures, whether in the technologies used, in the decision-making processes or in the people who interact with them. To ensure proper risk coverage, regulators and companies themselves impose a number of requirements that must be met. These requirements are translated into infrastructure constraints which can have an impact, positive or negative, on the risk. The risk remaining after an iteration of constraints, which frameworks such as [EBIOS-RM] call the *residual risk* is then evaluated and gives rise to new requirements, depending on whether it is deemed unacceptable. This is what we call the *risk cycle* (represented on figure IV.2).

This risk cycle is a central element of our work, that we further develop and refine throughout this dissertation. To introduce the concepts we discuss here and in the following chapters, we propose first to present our formalization of the process, and take the opportunity to prove a few of its properties.

IV.1.1 Formalism

Let $\mathcal{I}$ be the set of all infrastructures and $\mathcal{R}$ the set of risks. Risk assessment can be seen as a mapping between an infrastructure and the various risks to which it is exposed. We represent it as a function $\mathsf{assess} : \mathcal{I} \to \mathbb{P}(\mathcal{R})$. For example, a datacenter built in a seismically active area is subject to seismic risk, which can be broken down into a risk for the integrity of buildings or for the proper operation of hardware components, among others.

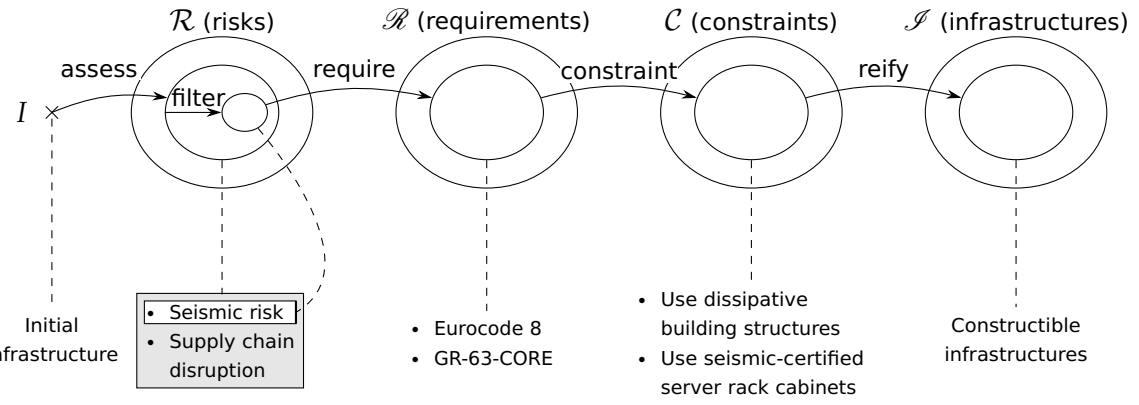


Figure IV.3: Risk management process: assessing risk, filtering what is out of scope/below tolerance criteria, translating risk into requirements, then constraints, and reifying the infrastructure.

Some of these risks may be deemed acceptable by a company, according to several criteria. We represent this "filtering" of risk as a function $\mathsf{filter} : \mathbb{P}(\mathcal{R}) \to \mathbb{P}(\mathcal{R})$ that takes a set of identified risks and gives a subset of unacceptable risks. For example, if a company keeps a large stock of equipment or has properly diversified its sources of supply, it can cope with the risks of supply chain disruption, and thus exclude it from its analysis.

Let us define $\mathscr{R}$ the set of requirements that apply on infrastructures. Given an infrastructure i, a number of requirements, both external and internal to a company, must be met. We can represent the listing of such requirements as a function $\mathsf{require}_i : \mathbb{P}(\mathcal{R}) \to \mathbb{P}(\mathscr{R})$ that maps a set of risks on i into a set of requirements to implement for i. In our example of seismic risk, standards such as [Eurocode 8] (*Design of structures for earthquake resistance*) provide a set of requirements for the construction of seismically-rated buildings. Additionally, section 4.4 (*Earthquake, Office Vibration, and Transportation Vibration*) of [GR-63-CORE] (*NEBS Requirements: Physical Protection*) gives generic requirements for network equipment in seismic environments.

Once these requirements are identified, we need to implement them in our infrastructure. We define $\mathcal{C}$ the set of so-called infrastructure constraints. Requirements are implemented as infrastructure constraints through a function $\mathsf{constraint}_i : \mathbb{P}(\mathscr{R}) \to \mathbb{P}(\mathcal{C})$ that maps a set of requirements to a set of possible constraints on i. For example, constraints associated to seismic requirements can range from the choice of equipment to the way buildings are constructed to the layout of equipment in each room.

These constraints need to be reified into a concrete infrastructure. We define $\mathscr{I} \subset \mathcal{I} \times \mathbb{P}(\mathcal{C})$ the set of infrastructures constructible under constraints. This definition considers that not all constrained infrastructures are necessarily constructible, particularly in the case of a contradiction between two constraints. Given $C \subset \mathcal{C}$ a set of prior constraints and $C' \subset \mathcal{C}$ a set of additional constraints, we define the function $\mathsf{reify}_C : \mathbb{P}(\mathcal{C}) \to \mathbb{P}(\mathscr{I})$ that maps a set of constraints into a set of constructible infrastructures, such that $\mathsf{reify}_C(C') = \{(i, C'') \in \mathscr{I} \mid C'' \supset C \cup C'\}$. Intuitively, this function gives the set of constructible infrastructures that respect at least both the prior constraints C and the new constraints C' we are seeking to implement.

We illustrate the process with our running example in figure IV.3.

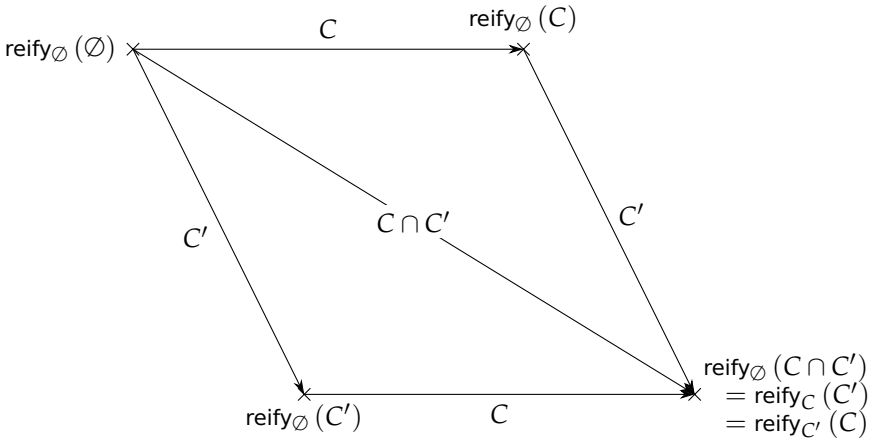


Figure IV.4: Graphical illustration of the reify function

IV.1.2 Properties

From the definition of the reify function, we can deduce a number of properties that illustrate the use of our formalism. The first two lemmas, directly derived from the definition of reify, highlight the iterative aspect of risk management. Figure IV.4 illustrates both lemmas. The last lemma is only used as a proof aid for the next subsection. It proves the trivial intuition that looking for an infrastructure that implements two sets of constraints is equivalent to looking among the infrastructures that implement the first set for those that implement the second.

Lemma 1 (Initial freedom). $\mathit{reify}_{\emptyset}(\emptyset) = \mathscr{I}$ *(reifying an unconstrained infrastructure can result in any constructible infrastructure).*

Lemma 2 (Connectivity). $\mathit{reify}_{C}(C') = \mathit{reify}_{\emptyset}(C \cup C')$ *(infrastructures can be constrained successively or at once).*

Lemma 3 (Separation). $\mathit{reify}_{C}(C') = \mathit{reify}_{\emptyset}(C) \cap \mathit{reify}_{\emptyset}(C')$ *(this lemma is only used as a proof aid).*

Proof.

$$
\begin{aligned}
\mathsf{reify}_{C}(C') &= \{(i, C'') \in \mathscr{I} \mid C'' \supset C \cup C'\} && \text{(definition)}\\
&= \{(i, C'') \in \mathscr{I} \mid C'' \supset C \wedge C'' \supset C'\}\\
&= \{(i, C'') \in \mathscr{I} \mid C'' \supset C\} \cap \{(i, C'') \in \mathscr{I} \mid C'' \supset C'\}\\
&= \mathsf{reify}_{\emptyset}(C) \cap \mathsf{reify}_{\emptyset}(C') && \text{(definition)}
\end{aligned}
$$

□

The following proposition reflects the fact that adding constraints to an infrastructure reduces the choices available for reifying it.

Proposition 4 (Antitony). $C \supset C' \Rightarrow \mathit{reify}_{\emptyset}(C) \subset \mathit{reify}_{\emptyset}(C')$ *(more constraints on an infrastructure lead to fewer choices).*

Proof.

$$
\begin{aligned}
C \supset C' \Leftrightarrow \quad & C = C' \cup (C \setminus C') \\
\Leftrightarrow \quad & \mathsf{reify}_\emptyset(C) = \mathsf{reify}_\emptyset\left(C' \cup (C \setminus C')\right) \\
\Leftrightarrow \quad & \mathsf{reify}_\emptyset(C) = \mathsf{reify}_\emptyset(C') \cap \mathsf{reify}_\emptyset(C \setminus C') \qquad \text{(lemmas 2 and 3)} \\
\Rightarrow \quad & \mathsf{reify}_\emptyset(C) \subset \mathsf{reify}_\emptyset(C')
\end{aligned}
$$

□

IV.1.3 Iteration

Risk analysis is an iterative process that can require several refinement steps to reach the desired goals. We have shown in figure IV.3 (page 28) a simple iteration of the process, from an initial infrastructure to a set of final infrastructures. Adding constraints to an infrastructure may introduce additional risks, requiring a new iteration of the process to remedy the situation. For example, if we consider a small infrastructure for sending and receiving e-mails in a company, we can identify the risk for employees to suffer from phishing attacks. One remediation solution may be to install filtering software on the infrastructure. However, this introduces the risk of employees no longer receiving legitimate, business-critical e-mail. A second iteration of the process becomes necessary to handle this new risk.

We assume here that the risk management process only adds risks, requirements and constraints at each iteration, without any removal. Although this approximation may not hold over the lifetime of major risk management projects, it remains true locally. We define the function iter to capture a single step of this process as such:

$$
\begin{aligned}
\mathsf{iter} : \quad & \mathbb{P}(\mathscr{I}) \to \mathbb{P}(\mathscr{I}) \\
& S \mapsto \bigcup_{(i,C)\in S} \mathsf{reify}_C \circ \mathsf{constraint}_i \circ \mathsf{require}_i \circ \mathsf{filter} \circ \mathsf{assess}(i)
\end{aligned}
$$

Although the risk management process is widely used in corporate environments, its recursive nature leaves us no guarantee of termination. This is the purpose of this subsection. To our knowledge, such a formalization aimed at proving the termination of the process is novel. First, we introduce a brief linearity lemma derived from the definition of iter to help us with our final proof.

Lemma 5 (Linearity). $\mathit{iter}(S \cup S') = \mathit{iter}(S) \cup \mathit{iter}(S')$

Proposition 6 (Limit). $\mathit{iter}^\infty \overset{\text{def}}{=} \lim_{n\to+\infty} \mathit{iter}^n$ *exists and is finite (the risk cycle eventually converges).*

Proof. In this proof, variables in blue denote metavariables local to each term of unions. Before getting to the heart of the proof, it is useful to prove the following goal. Intuitively, it means that a process iteration in which no constraint is added does nothing.

$$
\begin{aligned}
\mathsf{iter}\left(\mathsf{reify}_\emptyset(C)\right) &= \bigcup_{(i,C')\in\mathsf{reify}_\emptyset(C)} \mathsf{reify}_{C'}\left(C_i''\right) && (C_i'' \text{ not developed for clarity}) \\
&= \bigcup_{(i,C')\in\mathsf{reify}_\emptyset(C)} \mathsf{reify}_\emptyset(C') \cap \mathsf{reify}_\emptyset\left(C_i''\right) && \text{(lemma 3)}
\end{aligned}
$$

$$
\begin{aligned}
&\subset \bigcup_{(\cdot,C')\in \mathsf{reify}_\emptyset(C)} \mathsf{reify}_\emptyset\left(C'\right) \\
&\subset \bigcup_{(\cdot,C')\in \mathsf{reify}_\emptyset(C)} \mathsf{reify}_\emptyset\left(C\right) && \text{(definition, proposition 4)} \\
&\subset \mathsf{reify}_\emptyset\left(C\right) && \square_1
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{iter}^2\left(S\right) &= \mathsf{iter}\left(\mathsf{iter}\left(S\right)\right) \\
&= \mathsf{iter}\left(\bigcup_{(i,C)\in S} \mathsf{reify}_C\left(C'_i\right)\right) && (C'_i \text{ not developed for clarity}) \\
&= \bigcup_{(i,C)\in S} \mathsf{iter}\left(\mathsf{reify}_C\left(C'_i\right)\right) && \text{(lemma 5)} \\
&= \bigcup_{(i,C)\in S} \mathsf{iter}\left(\mathsf{reify}_\emptyset\left(C \cup C'_i\right)\right) && \text{(lemma 2)} \\
&\subset \bigcup_{(i,C)\in S} \mathsf{reify}_\emptyset\left(C \cup C'_i\right) && \text{(goal 1)} \\
&\subset \bigcup_{(i,C)\in S} \mathsf{reify}_C\left(C'_i\right) && \text{(lemma 2)} \\
&\subset \mathsf{iter}\left(S\right)
\end{aligned}
$$

$$
\begin{aligned}
\forall S \subset \mathscr{I}, \forall n \geq 1, \mathsf{iter}^{n+1}\left(S\right) &= \mathsf{iter}^2 \circ \mathsf{iter}^{n-1}\left(S\right) \\
&\subset \mathsf{iter} \circ \mathsf{iter}^{n-1}\left(S\right) \\
&\subset \mathsf{iter}^n\left(S\right)
\end{aligned}
$$

Thus, $\lim_{n\to+\infty} \mathsf{iter}^n$ exists and is finite.

□

The convergence of the process comes as good news, but we should not celebrate too quickly. We do not want to find ourselves in a situation where the process converges on $\emptyset$, which would mean that no infrastructure can be built. Care must then be taken when defining the functions applied at each iteration, notably the filter function. If too many risks are filtered, more options become available for building infrastructures, at the cost of increased sensitivity to risk. If, on the other hand, too many risks are preserved, the chances of the infrastructure not being constructible ($\mathsf{iter}^\infty\left(S\right) = \emptyset$) increase. Our work does not offer a solution to this dilemma, which we see as something only an expert analyst can resolve.

IV.1.4 Change

Once the risk management process has given us a number of candidate infrastructures to implement, we need to choose one. Our formalism does not propose a way of making this decision. This choice is very often made by a strategy of cost minimization, defined by the company. These costs may be financial (if we wish to reduce equipment purchases), temporal (if we wish to reduce the amount of

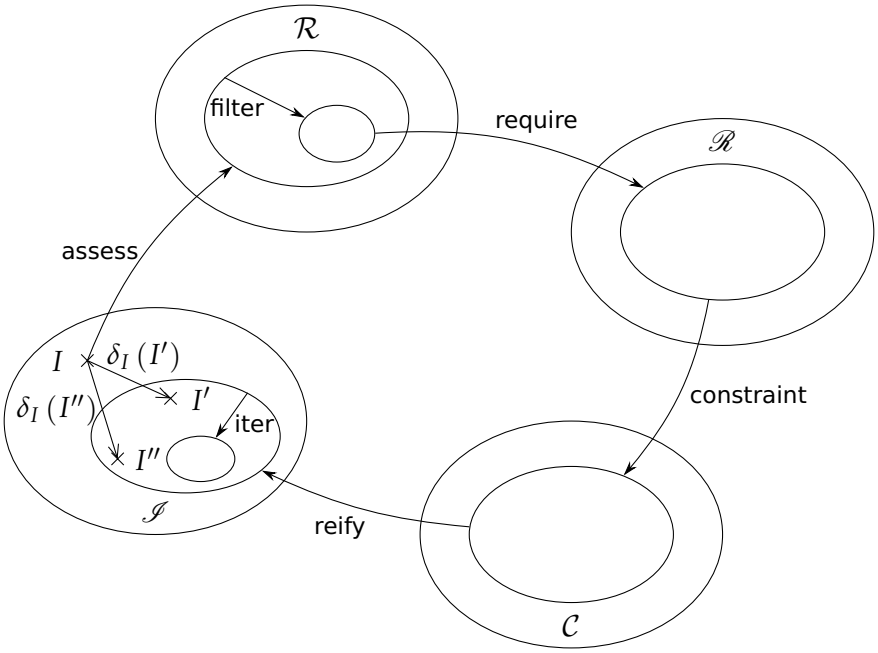


Figure IV.5: Risk management cycle

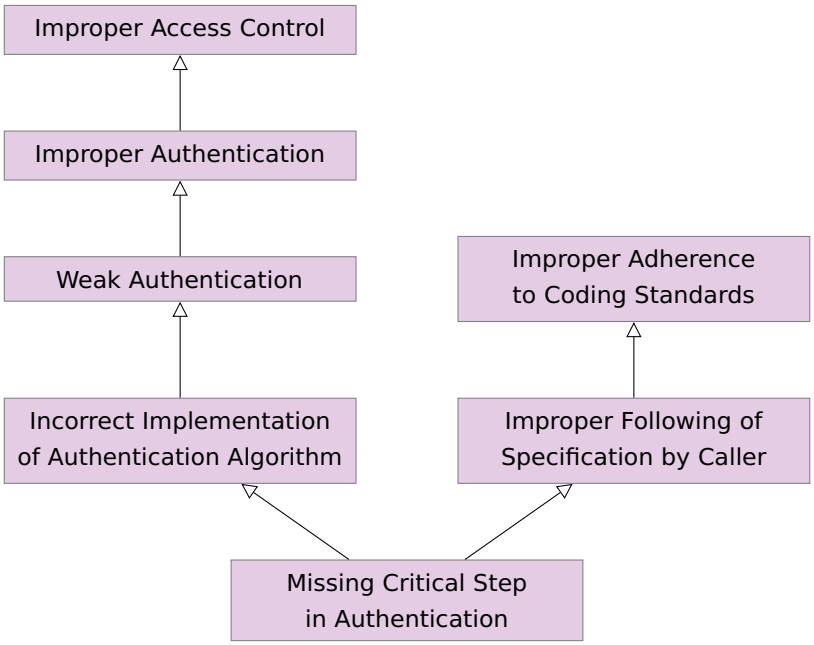


Figure IV.6: Hierarchy of the "Missing Critical Step in Authentication" CWE weakness. The arrows read as *child of* or *is a* (a Weak Authentication *is an* Improper Authentication).

human time spent), or ecological (if we wish to minimize the carbon footprint of the change) for example. For two constrained infrastructures $I = (i, C), I' = (i', C') \in \mathscr{I}$, let us denote $\delta_I(I')$ the changes needed to move from I to I'. The cost of such change can be seen as a function cost. To complete our formalism, we can consider that the optimal choice of infrastructure, given by the minimization of this cost, is $\operatorname{argmin}_{\mathsf{reify}_C(C')}(\mathsf{cost} \circ \delta_I)$ (where $\operatorname{argmin}_S f = \{x \in S \mid f(x) \leq f(x') \, \forall x' \in S\}$ is intuitively the infrastructure for which the cost of change is minimal).

We explore the notion of change in further details in section VI.1.2.

IV.1.5 Approach

Figure IV.5 summarizes this formalization. In this chapter, we focus on the workflow related to the functions assess, filter and require.

IV.2 Risk classification

IT risk is a very broad field, with a wide variety of causes, various types of consequences and many possible courses of action. In this context, risk classification can play a multi-faceted role. First and foremost, it aids in correctly communicating about issues: a proper classification can provide experts, managers and clients with information that they can more easily interpret and process. By understanding the nature and type of a risk, stakeholders can discern its potential impacts and the appropriate mitigation strategies. Multi-level taxonomies can serve as a foundation for communication between employees working in different domains, with different concepts and abstractions, within a company. Figure IV.6 gives an example of a hierarchical typology as seen in MITRE's CWE.

This foundation can be leveraged to help assessing the risk, filtering the acceptable risk and giving guidelines on safety and security requirements. For instance, if a tool is aware of a particular class of application-layer vulnerabilities, it can be programmed to scan for patterns associated with that class, thereby improving its effectiveness. If categories of risk are considered to have little impact by domain experts, they can be ignored. Finally, a good risk classification can lead to the definition of quality standards and best practices to be respected for addressing each risk.

IV.2.1 Taxonomy efforts

Several collective initiatives have emerged over time to classify risk. Here, we delve into four of them, which are distinguished both by their nature and by the spectrum they cover.

The *Open Web Application Security Project* (OWASP) is a global non-profit that works towards enhancing the security of software applications. The *Application Security Verification Standard* ([OWASP ASVS]) serves as a robust framework ensuring software applications are built with security embedded from the foundation. Through a categorized set of implementation guidelines, it offers developers with a solid baseline to create secure-by-design applications. Being a catalog of low-level guidelines, it is inherently opinionated and may require adjustments. In figure IV.7, we show OWASP ASVS guidelines for authentication systems.

The *National Institute of Standards and Technology* (NIST) is instrumental in setting standards in various fields, including cybersecurity. Their *Security and Privacy Controls for Federal Information Systems and Organizations* ([NIST SP 800-53]) is a notable framework that offers an organized collection of

V1.2 Authentication Architecture

When designing authentication, it doesn't matter if you have strong hardware enabled multi-factor authentication if an attacker can reset an account by calling a call center and answering commonly known questions. When proving identity, all authentication pathways must have the same strength.

#	Description	L1	L2	L3	CWE
1.2.1	Verify the use of unique or special low-privilege operating system accounts for all application components, services, and servers. (C3)		✓	✓	250
1.2.2	Verify that communications between application components, including APIs, middleware and data layers, are authenticated. Components should have the least necessary privileges needed. (C3)		✓	✓	306
1.2.3	Verify that the application uses a single vetted authentication mechanism that is known to be secure, can be extended to include strong authentication, and has sufficient logging and monitoring to detect account abuse or breaches.		✓	✓	306
1.2.4	Verify that all authentication pathways and identity management APIs implement consistent authentication security control strength, such that there are no weaker alternatives per the risk of the application.		✓	✓	306

Figure IV.7: "Authentication Architecture", OWASP Application Security Verification Standard 4.0.3, page 18, license CC-BY-SA 4.0

AC-11 DEVICE LOCK

Control:

a. Prevent further access to the system by [*Selection (one or more): initiating a device lock after* [*Assignment: organization-defined time period*] *of inactivity; requiring the user to initiate a device lock before leaving the system unattended*]; and

b. Retain the device lock until the user reestablishes access using established identification and authentication procedures.

Discussion: Device locks are temporary actions taken to prevent logical access to organizational systems when users stop work and move away from the immediate vicinity of those systems but do not want to log out because of the temporary nature of their absences. Device locks can be implemented at the operating system level or at the application level. A proximity lock may be used to initiate the device lock (e.g., via a Bluetooth-enabled device or dongle). User-initiated device locking is behavior or policy-based and, as such, requires users to take physical action to initiate the device lock. Device locks are not an acceptable substitute for logging out of systems, such as when organizations require users to log out at the end of workdays.

Related Controls: AC-2, AC-7, IA-11, PL-4.

Figure IV.8: "Device Lock", NIST SP 800-53, Rev. 5, republished courtesy of the National Institute of Standards and Technology

security and privacy controls, structured to address a large spectrum of cybersecurity requirements. Both requirements and controls are supported by U.S. laws and directives, making the document a reference for U.S. companies. Figure IV.8 shows the standard's view on screen locking. While the ASVS describes low-level concepts, the SP 800-53 is much more descriptive and open to interpretation.

The *European Union Agency for Cybersecurity* (ENISA) offers an established [Threat Taxonomy][1]. It gives a structured methodology to classify risks for IT systems, including safety risks (natural disasters, data loss...) and physical considerations (hardware malfunction, outages...) in addition to software security risks. Because it has not been updated since 2016, we think it may be unsuited for cybersecurity analyses, as the domain evolves rapidly. However, the safety aspects, neglected in other generic frameworks are a notable strength of the document. We have extracted some of the risks it identifies and show them in figure IV.9.

[1]There is a significant overlap between the notions of risks and threats and definitions may differ from one organization to another. We consider ENISA's threats as risks.

Threat number	High Level Threats	Threats	Threat details
44	Disaster (natural, environmental)		
45		*Disaster (natural earthquakes, floods, landslides, tsunamis, heavy rains, heavy snowfalls, heavy winds)*	
46		*Fire*	
47		*Pollution, dust, corrosion*	
48		*Thunder stroke*	
49		*Water*	
50		*Explosion*	
51		*Dangerous radiation leak*	
52		*Unfavorable climatic conditions*	
53			Lost of data or accessibility of IT infrastructure in result of extensive humidity
54			Lost of data or accessibility of IT infrastructure in result of extensive temperature
55		*Major events in the environment*	
56		*Threats from space / Electromagnetic storm*	
57		*Wildlife*	
58	Failures/ Malfunction		
59		*Failure of devices or systems*	
60			Failure of defective data media
61			Hardware failure

Figure IV.9: Excerpt from ENISA's Threat Taxonomy, entries 44 to 61, republished courtesy of the European Union Agency for Cybersecurity

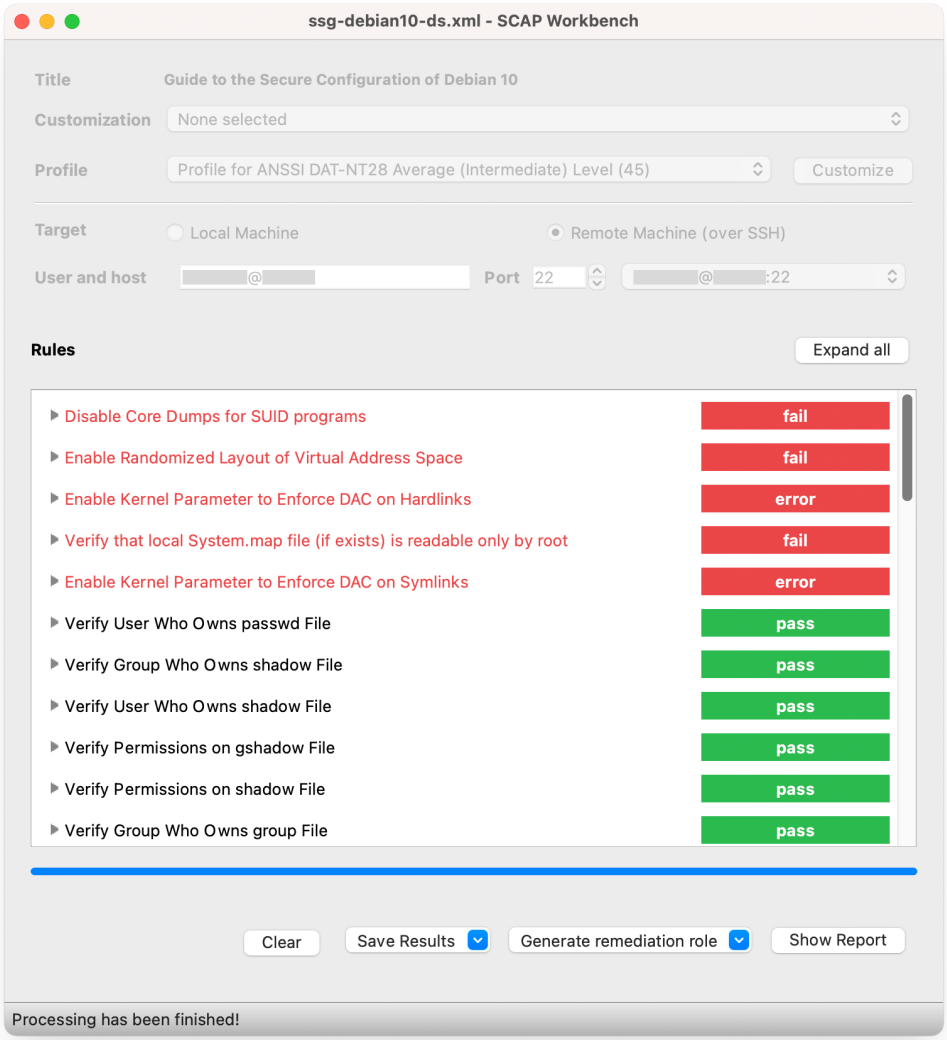


Figure IV.10: SCAP Workbench performing checks recommended by [ANSSI DAT-NT-028] on a Debian 10 system

Finally, a *Security Technical Implementation Guide* (STIG) is a set of configuration guidelines for ensuring system security. Originally developed at the U.S. *Defense Information Systems Agency*, these guidelines have also gradually been written by software publishers. The *Security Content Automation Protocol* (SCAP), a method used to automate vulnerability management and policy compliance evaluation, works hand in hand with STIGs. It facilitates the automated vulnerability checking, technical control compliance activities, and security measurement, ensuring that the guidelines stipulated in STIGs are adhered to. Tools such as SCAP Workbench, shown in figure IV.10 allow to perform analyses on servers. This opening up to automation is a significant step forward, with large actors such as the US Department of Defense or Red Hat sharing public compliance checklists[2], and some security vendors integrating the protocol into their proprietary solutions, such as [Tenable Nessus] and [Trellix ePO].

[2]Available online (`https://public.cyber.mil/stigs/`)

The first three frameworks are text-based, backed by a number of external (and often also textual) references. Their systematic use therefore requires human judgement to implement them in a given infrastructure. For STIG and SCAP, the data can be used directly by a computer, but the field of action is limited to system configuration.

IV.2.2 The case of MITRE

Given the limitations of the initiatives cited earlier, we have decided to focus on several projects from the MITRE Corporation, namely [CAPEC], [CVE] and [CWE] as a base for our work. Thanks to their maturity, popularity and structural properties, these projects provide a solid foundation for risk classification, helping to assess and filter risk.

CAPEC (for *Common Attack Pattern Enumeration and Classification*) is a comprehensive dictionary and classification taxonomy of known attacks that can be used to exploit applications. The main purpose of CAPEC is to provide a standardized way of identifying, sharing, and documenting various attack patterns that threat actors might use. By understanding the patterns and techniques used in different attacks, developers, testers, and consumers can devise strategies to better protect their systems. For example, `CAPEC-25` (Forced Deadlock) describes an attack pattern in which an attacker manages to cause a denial of service by exploiting a situation leading to a deadlock in a system.

CVE (for *Common Vulnerabilities and Exposures*) is a list of publicly known cybersecurity vulnerabilities and exposures. Each entry in the CVE list contains an identification number, a description, and at least one public reference. This system allows the security community to access and share data about vulnerabilities effectively, facilitating better defense strategies and quicker responses to new vulnerabilities. It is widely used by organizations to track and manage vulnerabilities in their systems. For example, `CVE-2009-1388` describes a vulnerability in the Linux kernel triggering a race condition, eventually leading to a deadlock.

Finally, CWE (for *Common Weakness Enumeration*) is a community-driven list of common software and hardware weakness types that have security ramifications. The main aim of CWE is to serve as a common language for describing software security weaknesses, as a standard measurement for software assurance tools and as a baseline for identification, mitigation, and prevention of weaknesses. For example, `CWE-833` (Deadlock) describes what a deadlock is and its relations to other weaknesses.

All three projects are tightly coupled: weaknesses in the CWE are the root causes for vulnerabilities (in the CVE when known), exploitable by an attacker using CAPEC techniques. We illustrate the links between our three previous examples in figure IV.11. In addition, they create a bridge to external initiatives, by making numerous references to OWASP datasets and NIST publications for example. Finally, many software applications use these projects, whose identifiers (`CAPEC-XXX`, `CVE-XXX`, `CWE-XXX`) can then be used as a common vocabulary for thorough analyses.

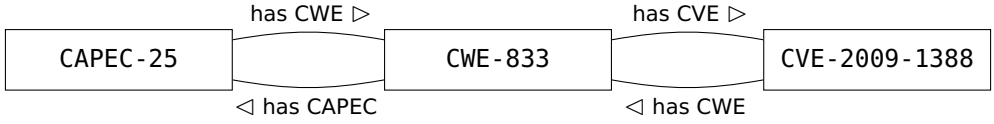


Figure IV.11: Relationships between `CAPEC-25`, `CVE-2009-1388` and `CWE-833`.

However, linking these three reference frameworks is a costly task. The contents from CAPEC and CWE are formally specified using XML Schema Definitions, and the content from CVE is formally specified using a JSON Schema[3]. Both formats allow for structured, expressive documents, while enabling automated data processing. We claim, however, that the format used is not up to the high coupling of the data and does not allow for complex reasoning. A question we might ask could be "what are the vulnerabilities (before 2010) and attack patterns related to a deadlock, and what are the links between these attack patterns and other weaknesses?". This question cannot be answered easily using the original XML/JSON representation and tools available, and the work we present in the following subsection seeks to fill this gap.

IV.2.3 An ontology over MITRE

To reduce human intervention in the process of identifying risk in infrastructures, it is important to have a solid basis of reasoning that facilitates its automation. To take advantage of the richness of the MITRE projects and enable complex reasoning both between each project and between the tools exploiting them, we have decided to develop an ontology based on the XML and JSON data available.

> **XML primer.**
>
> XML (eXtensible Markup Language) is a markup language for data representation. It comprises elements with start and end tags, which can have nested elements and attributes. A 30-year-old John Doe can be represented as such:
>
> ```
> <person age="30">
>   <name>John Doe</name>
> </person>
> ```
>
> The structure of XML documents can be controlled by XML Schema Definitions (XSD). XSD files, also written in XML, describe the structure of XML documents and allow for their validation, ensuring their coherence. The following schema can fit our XML example:
>
> ```
> <xs:element name="person">
>   <xs:complexType><!-- An element with tag ‘person‘ -->
>     <xs:element name="name" type="xs:string"/><!-- An element with tag ‘name‘ -->
>     <xs:attribute name="age" type="xs:integer"/><!-- An attribute with name ‘age‘ -->
>   </xs:complexType>
> </xs:element>
> ```
>
> The data tree of XML files can be explored with the XPath language. It provides a way to access elements and attributes using so-called path expressions. For example, `//person/name` would retrieve the `name` element `John Doe` and `//person/@age` would retrieve the `age` attribute `30` from the previous document. The special path `.` designates the current node, and functions `text()` and `name()` respectively get the current element's textual representation and tag name.

[3] An XML version of the CVE is also available, but we have found it to be lacking critical information. It is moreover scheduled for deprecation in 2024.

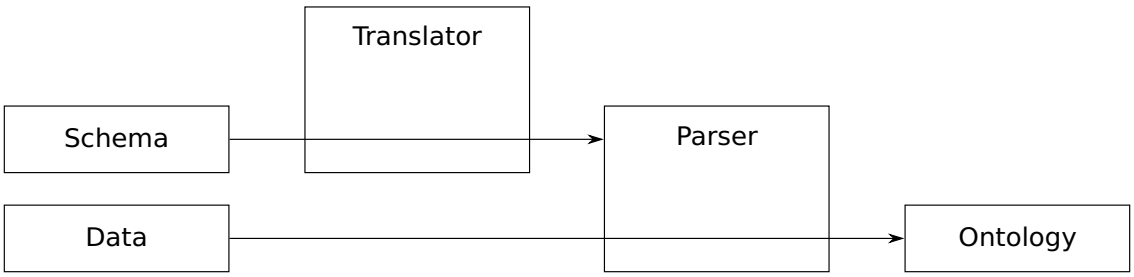


Figure IV.12: Structure of `mitre2owl`. The schema is used by the translator to build a parser, which can parse the data to build the ontology.

We have developped a tool, `mitre2owl`[4], which translates CAPEC, CVE and CWE schemas and data into ontologies. For each of these, the tool analyzes the schemas and creates specialized parsers to translate the XML and JSON data into an OWL/XML ontology. The stucture of `mitre2owl` is shown on figure IV.12.

> **Ontology primer.**
>
> An ontology is a formal representation of knowledge within a domain. It describes concepts, relationships, and the rules that govern them, enabling the creation of a shared and common understanding of a subject. OWL is W3C-endorsed standard for creating and defining ontologies of the form `<subject> <predicate> <object>`, allowing for rich representation of knowledge using triples.
>
> Turtle (for Terse RDF Triple Language) is one of the syntaxes to express an OWL ontology. In Turtle, we could represent our 30-year-old John Doe as such (in the namespace `ex` representing our example):
>
> ```
> ex:John a ex:Person. # John is a person
> ex:John ex:hasAge "30"^^xsd:integer. # John's age is 30
> ex:John ex:hasName "John Doe". # John's name is John Doe
> ```
>
> Ontologies can be queried using SPARQL (for SPARQL Protocol and RDF Query Language). To fetch all the persons named John Doe, we would run:
>
> ```
> select ?person where {
>     ?person a ex:Person.
>     ?person ex:hasName "John Doe".
> }
> ```

The algorithm

Here we describe the main elements of the XML/XSD part of our algorithm (the JSON part is substantially similar). The actual algorithm is more complex, as it handles data conversion, several semantic adaptations to make the final structure better suited to an ontology, and adds reasoning rules. For convenience, we suffix the words derived from "parse" with S and D to remove any ambiguity as to whether we are parsing the schema (S) or the data (D).

[4]Available publicly on Github (`https://github.com/CAPRICA-Project/mitre2owl`)

Algorithm: Parser builder (`Schema`)

```
Data: xs:schema element
types ← {};
elements ← {};
foreach n ∈ /*/xs:complexType do
  | types[n] ← ComplexType(n)
end
foreach n ∈ /*/xs:simpleType do
  | types[n] ← SimpleType(n)
end
foreach /*/xs:element do
  | elements[@name] ← Element(.)
end
```

Figure IV.13: `xs:schema` parser$_S$. `expression` executes the XPath expression "`expression`" on the XML node in scope. The default scope is the algorithm input, and **if** and **foreach** combinators locally change the scope to the XML nodes matched by their respective XPath expressions.

Algorithm: `Attribute.parse`

Data: XML element
Result: `<?> <has name()> types[type].parse(.).`

Algorithm: `SimpleType.parse`

Data: XML element
Result: `< text()> a name().`

Figure IV.14: Algorithms for the attributes and simple types parsers$_D$. The semantics for the composition of results is presented in appendix A.2.

The first step of our algorithm is to load the XSD schema to parse$_S$ its contents. Schemas define custom types and values that influence the way data is parsed$_D$ in XML files. These types are very varied, notably describing the structure of a vulnerability or an attack, and the values can describe elements such as an operating system or processor architecture. We first parse$_S$ types defined globally, so that we know how to parse$_D$ our data when we encounter them. Then, we parse$_S$ the XSD's elements to learn the structure of the data. This is shown in figure IV.13. Appendix A.1.1 shows the other parsers$_S$ that we use (to parse for example `xs:complexTypes` or `xs:attributes`). It is important to note that, to simplify the implementation, our parsers are not complete with respect to XML schemas, but the subset we are processing is complete with respect to the current MITRE datasets.

After this step, we have built several parsers$_D$ that we use to construct our ontology. When parsing$_D$ elements from the XML data, we create so-called assertions of the following form (appendix A.1.2 shows all the parsers$_D$ for the XML data):

- `<subject> <has` *attribute name*`> <` *attribute value*`>` for XML attributes (shown on figure IV.14 as `Attribute.parse`);
- `<subject> a <` *type name*`>` for typing XML elements (an example for `xs:simpleTypes` is shown on figure IV.14 as `SimpleType.parse`);
- `<subject> <has` *child name*`> <` *child identifier*`>` for XML elements having children (same logic as `Attribute.parse`).

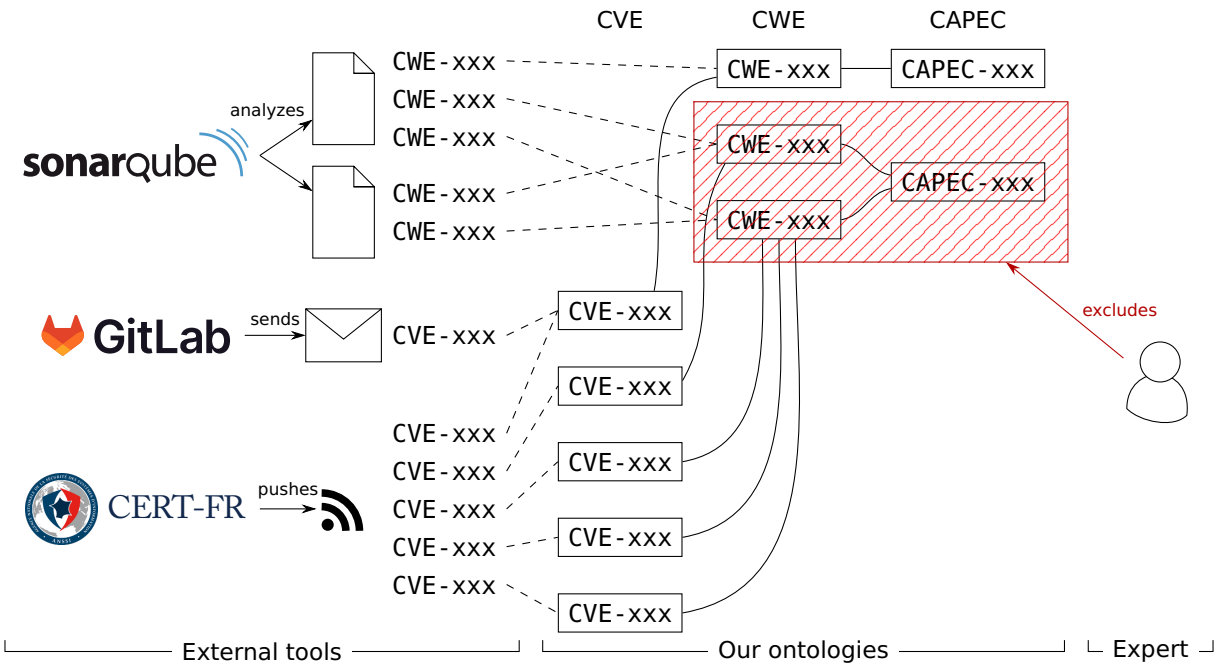


Figure IV.15: Integration of our ontologies in an industrial process. SonarQube, GitLab and ANSSI logos respectively © SonarSource SA, GitLab Inc. and Agence nationale de la sécurité des systèmes d'information.

There have been a few attempts in the literature to formalize the MITRE datasets in the form of an ontology. We can cite [Dimitrov23] in particular which describes an ontology for the CWE, but these approaches are mainly *ad hoc* developments, and we have not found any set of interoperable ontologies for the three MITRE datasets we are studying. In addition to filling a semantic gap, we consider our approach to be more correct and complete, since we are building our ontologies directly from the datasets, with limited *ad hoc* developments[5]. We have set up an automatic generation system for `mitre2owl` ontologies based on Github actions; a daily update is publicly available on Github[6].

Using the ontologies

We see our ontologies fitting into a more interactive and iterative risk assessment process. Naturally, since the MITRE projects are mainly concerned with security issues, our ontologies are limited to these, but we can imagine their extension to safety concerns. For future developments, we are considering bringing together around our ontologies various software applications that rely on MITRE datasets, such as the [SonarQube] and Astrée [Cousot05] code verifiers, to enable more advanced reasoning between tools. We show in figure IV.15 an example of a workflow in which our ontologies can be used. On the left-hand side of the figure, we represent external tools linked to MITRE identifiers that report security alerts on a company's systems. Our ontologies are placed in the middle of the figure, and an operator can use an ontology exploration tool to unfold a risk assessment on infrastructures. Finally, on the right-hand side is the verdict of acceptability for each risk, produced by a domain expert. When a

[5]Ours are here mostly for cosmetic purposes and ease of use

[6]Due to size limitations, the CVE ontology cannot be versioned on Github. The CAPEC ontology is available on `https://github.com/CAPRICA-Project/CAPEC.owx` and the CWE ontology is available on `https://github.com/CAPRICA-Project/CWE.owx`.

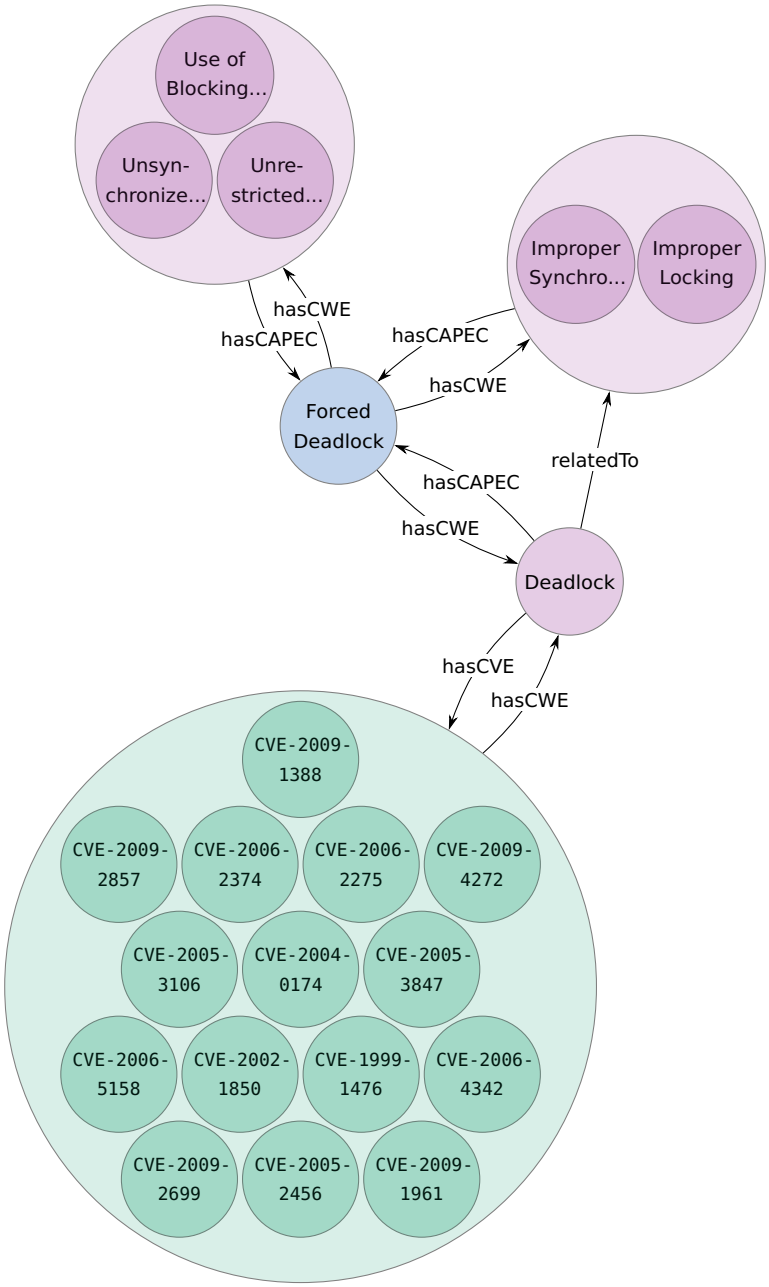


Figure IV.16: Exploration of Deadlock, obtained from the following SPARQL query:

```
construct where {
    cwe:CWE-833 cwe:hasCAPEC ?capec;
                cwe:hasCVE ?cve;
                cwe:relatedTo ?cwe2.
    ?cve cve:hasName ?cve_name.
    ?capec capec:hasCWE ?cwe.
    filter (?cve_name < "CVE-2010").
}
```

Figure IV.17: Vulnerability–Weakness rosette. Visualization obtained with Gephi, using the Dual Circle Layout, with custom coloring and categorization. Full version and dataset available on `https://github.com/CAPRICA-Project/CWE-Rosette`.

security alert is received or a vulnerability is identified in the company's systems, it is then possible to use a semantic reasoner to identify families of attacks that could exploit these vulnerabilities, and thus dynamically update the risk analysis of the overall infrastructure.

At the end of section IV.2.2, we asked the question "what are the vulnerabilities (before 2010) and attack patterns related to a deadlock, and what are the links between these attack patterns and other weaknesses?". We give a visual answer to this question in figure IV.16, along with the SPARQL query using our ontologies which generated the graph.

Finally, with the help of our ontologies, we were able to carry out a study of the most frequently encountered weaknesses and represented the distribution of CWE classes in the form of a rosette, as shown in figure IV.17. Each of the 64,000+ lines represents the association of a CVE (in the center, hidden for cosmetic reasons) with a CWE class (on the outside). CWE classes are grouped into so-called pillars, that represent main CWE categories. Looking at the various computer attacks carried out over the last few years, it comes as no surprise that the most exploited pillars are `CWE-707` (Improper Neutralization) and `CWE-664` (Improper Control of a Resource Through its Lifetime). Such representations of risk can provide a high-level view of a company's overall exposure for decision-makers, and help focus remediation objectives within the company.

IV.3 Risk analysis frameworks

In a landscape of rapidly evolving technology, the critical role of robust IT infrastructures cannot be understated; risk management is an important element in the life cycle of IT systems. A sound risk taxonomy, whether derived from the MITRE in our case, or other reference frameworks, can help guide risk analysis workshops into coherent, prioritizable topics. In this section, we look at several risk analysis frameworks that we think are useful for studying IT infrastructures, and examine their relevance to modern cloud infrastructures.

IV.3.1 Traditional frameworks

Many general-purpose implementations of risk management (ISO 31000) have been developed over the years, the most notable for our work being *Fault Tree Analysis* (FTA), *Failure Mode and Effects Analysis* (FMEA), *Hazard and Operability study* (HAZOP) and *System-Theoretic Process Analysis* (STPA).

FTA [ARP4761] is the leading top-down approach, developed at Bell Laboratories as an early effort to incorporate systematic safety and reliability analysis in complex systems. In this method, main events are first identified and their (advisably independent) causes are recursively analyzed and combined using boolean logic gates. Individual causes leading to a particular event are put in OR gates and collective causes are put in AND gates. It is a deductive approach carried out by repeatedly asking "how can this event happen and what are its causes?". Figure IV.18 gives an example of such an analysis on a physical server.

In contrast, FMEA [ARP4761] focuses on single failures, whether they have a higher-level impact or not. FMEA is a bottom-up, inductive method developed by the U.S. Military to classify failure modes according to their impact on mission success and personnel safety. It is used to identify low-level failures and study the effects on the higher levels. Table IV.1 gives an example of such an analysis on a physical server.

ID	Component	Failure mode	Failure effect	Cause
1	Power subsystem	No power suply powers up	The server does not power up	PSU absent, dysfunctional or not properly seated
2		One power supply does not power up	The orange indicator light flashes	
3	CPU controller	All CPU failed	The server cannot boot	CPU absent, dysfunctional or not properly seated
4		One CPU failed	The CPU is not used and its RAM is not recognised	
5	Drive controller	All drives failed	Loss of data, OS cannot boot	Drives dysfunctional or not properly seated
6		Some drives failed	I/O operations slower than usual	
7	RAM controller	Some RAM sticks failed	The server may not boot	RAM dysfunctional or not properly seated
8		Wrong RAM configuration	Some RAM sticks are not used	RAM sticks seated at wrong positions

Table IV.1: Failure Mode and Effects Analysis of a server

FTA and FMEA are often used together thanks to their opposite views, but are very time-consuming to apply thoroughly, therefore the analyses are often incomplete [Cristea17]. Due to the iterative nature of their development, FTA and FMEA analyses can be easily composed and specialized.

HAZOP [IEC 61882], created by the Imperial Chemical Industries, works very differently from the other two methods. It starts by defining system parameters (such as “electricity”, “heat”...) and using standardized *guidewords* (“no”, “more”, “less”...) to study deviations in these parameters from a design intent (“no electricity”, “more heat than expected”...). The technique is a qualitative way to assess complex processes and focuses on structured discussions. In some cases, this format may however provide a false sense of security by being too guided [Baybutt15].

In these frameworks, interactions between components are often not taken into account [Sulaman19]. FTA and HAZOP indeed analyze separately each part to provide a synthetic assessment for the system using a divide-and-conquer approach.

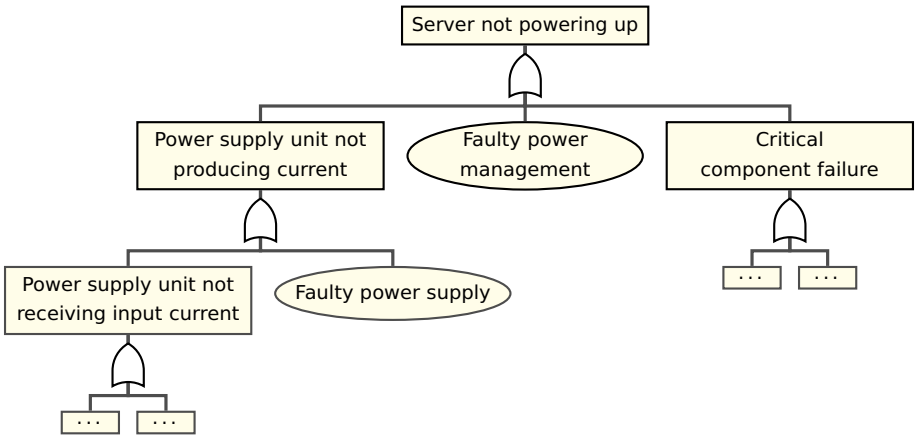


Figure IV.18: Fault-Tree Analysis of a server not powering up. Rectangles represent compound events, ellipses represent basic events, and the gates between them are OR operators.

IV.3.2 Modern initiatives and IT infrastructures

The majority of these frameworks have been developed over 40 years ago, and are not always well suited for analyzing modern complex systems, be it IT infrastructures or systems including many control loops. To overcome the limitation of the previously mentioned frameworks, the STPA [Leveson12] method, created by MIT Professor Nancy Leveson, adopts an approach focused on unsafe interactions between components. Instead of focusing on individual failures, it focuses on the control actions being the real cause of accidents. It is a top-down method to study functional control instead of physical components. Four main steps are identified:

1. Define the purpose of the analysis, what the analysis aims to prevent;
2. Model the control structure, show the relationships and interactions at play;
3. Identify unsafe control actions, how they can lead to losses defined in the first step, and create functional requirements and constraints for the system;
4. Identify loss scenarios, understand why unsafe control occurs and which and why components lead to a loss.

IT infrastructures also have their own analysis frameworks. EBIOS Risk Manager [EBIOS-RM] is a method maintained by the French cyber agency ANSSI proposing a cyclic approach for risk management. In addition to implementing ISO 31000, it also implements [ISO/IEC 27005] (*Information security, cybersecurity and privacy protection – Guidance on managing information security risks*), dealing with information security risk management. It consists of five workshops identifying high level, textual risk scenarios and their resolutions with security measures:

1. Define the scope of the study and the security baseline;
2. Identify the origins of risk, who can adversely affect the system and why;
3. Get a good view of the infrastructure and devise strategic high-level attack scenarios;
4. Build low-level operational scenarios and identiy the critical supporting assets;
5. Summarize risk scenarios and define a risk treatment strategy.

If we set aside EBIOS-RM, the frameworks mentioned in this dissertation have proved effective in industries like aerospace or automotive, but IT infrastructures present unique challenges that we think call for several adaptations of the methods. Systems traditionally targeted by general-purpose risk analysis frameworks tend to be made up of many distinct components, each with a specific function and internal behavior. IT infrastructures are made of many commercially available off-the-shelf (COTS) components with many common failure modes and effects. This specificity has a direct impact on risk analyses insofar as, while there are many components to analyze in large infrastructures, many of them are identical or perform equivalent functions. In the case of EBIOS-RM, its strong emphasis on cybersecurity often overshadows another essential facet: safety. In data centers, for example, physical safety from fires or electrical malfunctions is as crucial as cybersecurity.

Furthermore, these frameworks work well on environments that do not regularly evolve: risk analysis is a costly process requiring large workshops involving numerous stakeholders, often performed once and for all for a product. Cloud infrastructures on the other hand have an inherently dynamic nature, allowing for automatic scaling, reconfiguration and deployment of resources. This leads to infrastructures whose temporal stability is too short (minutes, hours) compared with the lifespan of conventional risk analyses (months, years).

All in all, some methods are better suited to specific domains, and we feel it is important to let the different stakeholders in the IT infrastructure choose which ones they want to use. As pointed out by [Alturkistani14], collaboration plays a vital role, and we reckon that federating these methods is beneficial for the domain. We take a closer look at the concept in section VII.3.

IV.4 Risk assessment and tolerance criteria

As we have shown on figure IV.18 and table IV.1 (page 45), generic risk analysis frameworks can be used on IT infrastructures, and their combination can be useful in identifying unexpected risk. But over the years, IT infrastructures have transitioned from their traditional model. While so-called legacy systems retain their importance in certain critical business operations (banking, healthcare...), the advent of virtualization brings along a whole new paradigm. In modern infrastructures, we can identify three layers with different stability profiles:

- The physical layer which is the most stable, where traditional analyses can still be performed (as long as hardware is under control of the company);
- The logical layer, which is as stable as the evolution of the company's projects;
- The deployment layer, which presents challenges for traditional approaches.

In this section, we present lessons learned from risk analysis of IT infrastructures through guidelines and suggestions for future developments.

IV.4.1 Analyzing parts

While automotive and aeronautical systems are made up of thousands of specialized components, IT infrastructures benefit from a standardization effort driven by the desire for component interoperability. For physical systems, this translates into the use of COTS components [Rose03], such as processors for computing power, RAM strips for fast volatile memory, or disks for permanent storage. For software systems, this translates into the use of common APIs and exchange formats, open source libraries, and well-established design patterns. Finally, for networked systems, this translates into the use of standardized protocols and architectures.

A company-wide inventory of infrastructure assets can show thousands of different components (hardware, software...), designed by hundreds of actors (companies, developer communities...). A thorough analysis of all the components, while theoretically desirable, is not feasible within a reasonable timeframe. When considering a system, it is crucial to choose the right level of abstraction, based on:

- The knowledge of the system;
- The ability to act upon the system;
- The resources (human, financial...) to act upon the system.

It seems unlikely to have absolute knowledge of an entire infrastructure. Some components can be considered as black boxes, such as sensitive hardware components like Hardware Security Modules, or proprietary firmware. The skills required to understand a component in depth may also be lacking within the company. Such components should therefore not be decomposed any further during analysis, and the study of their failure modes can be kept to a bare minimum.

Packaged components that cannot be decomposed, such as soldered electronic components or closed-source software, often cannot be corrected if one of their sub-components fails. For example, if a DRAM chip malfunctions on a memory strip, or if there is a bug in a closed-source software function, the component as a whole should be considered faulty. Therefore, it does not make much sense to study them in depth.

Finally, if fixing a problem on a component is too complex, requires too much money, or even costs more money than replacing the component itself, there is little point in decomposing it further. This may be the case for systems that have reached the end of their life cycle, or whose manufacturer's support contract has expired.

For these reasons, we think that the average customers of software and hardware technologies should not carry out their own analyses on the components they purchase, not to mention that n customers would need to perform n analyses. Packaging components with their risk analyses ensures that the expertise of each stakeholder is respected, as the ones designing systems are among those who know them best. Furthermore, providing an update channel ensures that different customers are alerted when problems are identified.

IV.4.2 Analyzing systems

The various parts of the infrastructure form systems, which in turn are further assembled into systems, ultimately forming the infrastructure as a whole. Needless to say, an assembly of components can exhibit behaviors that individual components do not; risk analysis is no exception to this rule. Even if all components behave as specified, the assembly of two components may present incompatibilities. If we think in terms of feature sets, components must provide enough features to components requiring them. For software, this would correspond to modules implementing the interfaces required by other modules. If we think in terms of standards, components must "fit" (physical standards) and use the right protocols (communication standards). Figure IV.19 shows an overview of the five possible cases of feature matchings, for each of which we give an example:

1 ($=$). All the required features are supplied; for example, a piece of software requires a specific version of a library, which is provided;
2 ($\subset$). More features are supplied than what is required; for example, a client uses a subset of the features offered by a remote server;
3 ($\supset$). Fewer features are supplied than what is required; for example, an algebra library requires advanced CPU instructions that are not available;
4 ($\cap$). Some of the required features are supplied, as well as other features; for example, two applications talking different dialects of a protocol, say Active Directory and OpenLDAP;
5 ($\varnothing$). Required features are not supplied; for example, a package not implementing any of the required functionalities.

Requirements must be expressed correctly at the component interface. Abstracting them too much may lead to cases 3 and 4. For example, if our application requires the specific "MariaDB Server 10.11.5", but we say we require any "SQL server", critical features may be lacking. On the contrary, too much

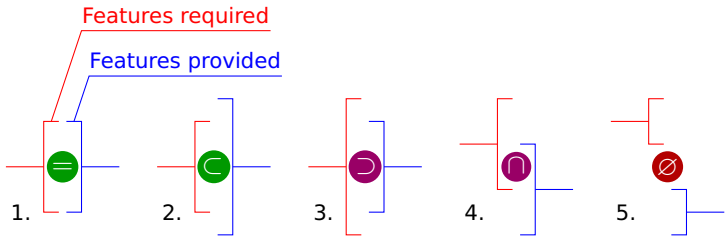


Figure IV.19: Five types of feature matching when assembling components
Legend. ● Compatibility, ● Possible incompatibility, ● Incompatibility.

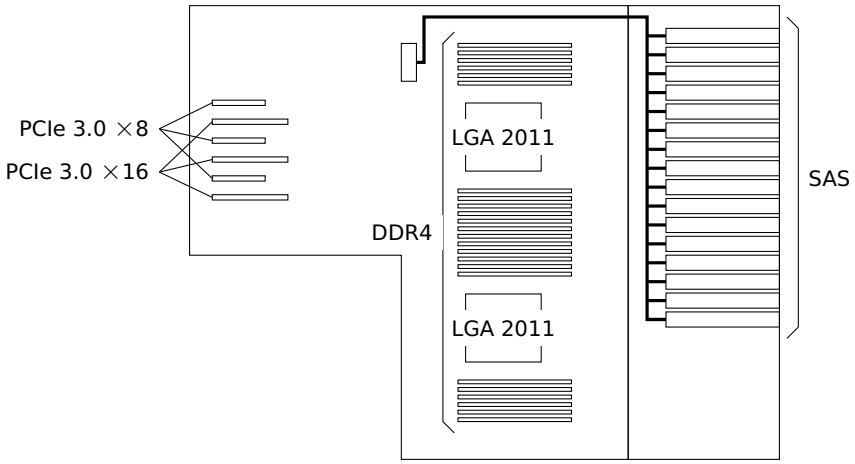


Figure IV.20: Typical server motherboard, with memory (DDR4) and expansion card (PCIe 3.0) slots, CPU sockets (LGA 2011) and drive trays (SAS)

specialization may make a system less configurable or make it dependent on unsafe components. For example, if our application requires a specific version of an external module, and this external module is later found to be critically vulnerable, our application can propagate the risk.

Without impacting nominal system operation, these cases can invalidate risk analyses, and in the worst case hide unsafe scenarios. A pathological case appears in hardware components such as motherboards (figure IV.20): the presence of standardized sockets creates a set of explicit compatibility requirements (if a sub-component does not fit, it cannot be used). However, these necessary conditions are often not sufficient, and manufacturers have to specify additional logical constraints in set-up guides. For example, old PCI Express expansion cards may not function properly on modern hardware (and *vice versa*), because of deprecation or a partial implementation of standards by the hardware controller. Another example is the layout and characteristics of RAM modules, often constrained by motherboard manufacturers and CPU vendors.

IV.4.3 Closing the loop

As we show in figure IV.2 (page 27), risk management is an iterative process: risks lead to requirements that are translated into infrastructure constraints, which can have a direct impact on the initial risk. This impact may either be positive (reducing the initial risk criticality), negative (introducing new risks), or both. Let us take the requirements from the Payment Card Industry Data Security Standard (PCI DSS) 4.0 as an example. PCI DSS was created by the Payment Card Industry Security Standards Council as a

PCI Security Standards Council

Requirements and Testing Procedures		Guidance
9.4 Media with cardholder data is securely stored, accessed, distributed, and destroyed.		
Defined Approach Requirements **9.4.1** All media with cardholder data is physically secured. **Customized Approach Objective** Media with cardholder data cannot be accessed by unauthorized personnel.	**Defined Approach Testing Procedures** **9.4.1.** Examine documentation to verify that the procedures defined for protecting cardholder data include controls for physically securing all media.	**Purpose** Controls for physically securing media are intended to prevent unauthorized persons from gaining access to cardholder data on any media. Cardholder data is susceptible to unauthorized viewing, copying, or scanning if it is unprotected while it is on removable or portable media, printed out, or left on someone's desk.
9.4.1.1 Offline media backups with cardholder data are stored in a secure location. **Customized Approach Objective** Offline backups cannot be accessed by unauthorized personnel.	**9.4.1.1.a** Examine documentation to verify that procedures are defined for physically securing offline media backups with cardholder data in a secure location. **9.4.1.1.b** Examine logs or other documentation and interview responsible personnel at the storage location to verify that offline media backups are stored in a secure location.	**Purpose** If stored in a non-secured facility, backups containing cardholder data may easily be lost, stolen, or copied for malicious intent. **Good Practice** For secure storage of backup media, a good practice is to store media in an off-site facility, such as an alternate or backup site or commercial storage facility.
Defined Approach Requirements **9.4.1.2** The security of the offline media backup location(s) with cardholder data is reviewed at least once every 12 months. **Customized Approach Objective** The security controls protecting offline backups are verified periodically by inspection.	**Defined Approach Testing Procedures** **9.4.1.2.a** Examine documentation to verify that procedures are defined for reviewing the security of the offline media backup location(s) with cardholder data at least once every 12 months. **9.4.1.2.b** Examine documented procedures, logs, or other documentation, and interview responsible personnel at the storage location(s) to verify that the storage location's security is reviewed at least once every 12 months.	**Purpose** Conducting regular reviews of the storage facility enables the organization to address identified security issues promptly, minimizing the potential risk. It is important for the entity to be aware of the security of the area where media is being stored.

Figure IV.21: PCI DSS 4.0, page 200, describing sub-requirement 9.4

structured way to address the risk of credit card fraud. Figure IV.21 shows how the requirement 9 (Restrict Physical Access to Cardholder Data) is declined into a sub-requirement mentioning copying cardholder data onto removable media. To prevent the situations described in the requirement's guidance, a company may decide to encrypt its staff's hard drives and deploy a corporate policy prohibiting removable media [Sharwood18]. While such an implementation of the requirement technically removes the risks associated with removable media and data extraction from disks, this can hinder legitimate activities such as sharing critical data to an authorized colleague. This in turn can lead to data being shared on less secure media, such as public remote drives, or worse, to attempt to circumvent corporate security policies [Blythe15], increasing the initial risk.

Care should thus be taken to properly analyze residual risks at each iteration of the risk cycle, and determine whether they are acceptable according to tolerance criteria. We have identified four families of acceptability criteria:

- Confidence:
  - Invulnerability, if a risk cannot impact an infrastructure (for example, an unsafe function that the infrastructure does not use in a library),
  - Unlikelihood, if a risk is so improbable that it does not seem relevant to consider it (for example, bit shifts due to cosmic rays on earth);

- Inability:
  - Impossibility, if there is no way to fix a risk (for example, a Hardware Security Modules with anti-tamper features disabling itself during an earthquake),
  - Lack of means, if a company does not have the resources to mitigate a risk (for example, a start-up company that cannot afford expensive security appliance),
  - Illegality, if a company is not allowed to fix a risk (for example, many jurisdictions regulate companies' ability to monitor employee behavior, even if leaks are suspected);
- Cost-benefit, if the cost of implementing risk mitigations exceeds the potential benefit of the protected asset (for example, if re-engineering an application to distribute its load is not justified by the few downtimes it might suffer);
- Transfer, if a risk is transferred to a third party (for example, through insurance, contracts or outsourcing).

Of course, insufficient knowledge or awareness can lead to overconfidence, inaccurate assement of (in)abilities, miscalculations of costs and benefits, and inappropriate risk transfer. We have unfortunately no compelling solution to cover this "meta-risk". We should note that these acceptability criteria can change over time; a company may decide, for example, that it is no longer acceptable to transfer its risk to a third party.

IV.5 Sharing analyses

Whereas the sharing of vulnerabilities (notably via CVE) and the means of analyzing their presence on a system (via SCAP for example) is commonplace in IT communities, the same cannot be said of risk analyses themselves. In the automotive industry, the recent [openXSAM] initiative (for *open format for eXchanging Security Analysis Models*) is a valuable tool for sharing security data between teams or with third-party suppliers. It can help to improve the efficiency and accuracy of security analyses, and make it easier to comply with regulations and security standards, such as the new [ISO/SAE 21434] (Road vehicles – Cybersecurity engineering). Because of its strong focus on cybersecurity in the automotive industry, it seems difficult to build a direct bridge with our work. We think however that this approach is crucial for IT infrastructures, which also involve hundreds of manufacturers in the development of end systems.

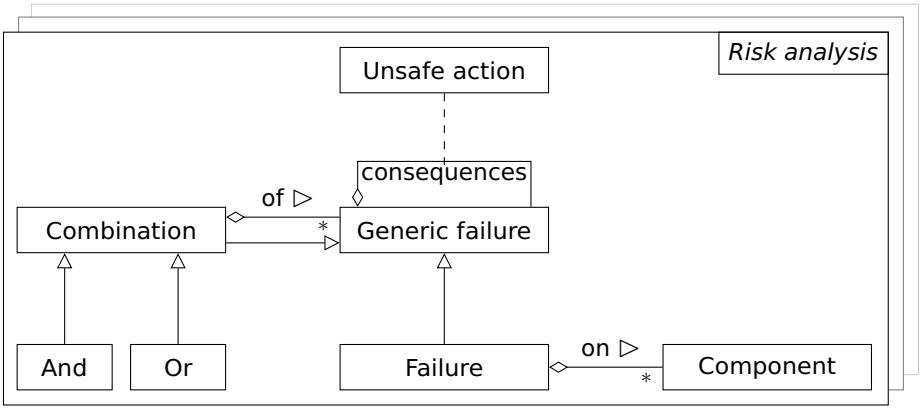


Figure IV.22: Generic model for risk analysis

We propose in figure IV.22 a generic model for risk analysis, which we claim captures the essential elements of FTA, FMEA, HAZOP and STPA methods. Basically, a *combination* of *failures* on *components* can have as *consequences*, potentially when triggering an *unsafe action*, other *combinations* of *failures* on *components*. For example, a mechanical problem on a rack fan can, when the fan is turned on, vibrate the entire rack and degrade mechanical disk performance. From our model, we can build a common exchange format to share risk analyses between cooperating parties. Sharing analyses can help clients to configure systems properly (assembling them in a way that avoids failures) and troubleshoot common problems (identify causes of incidents).

IV.5.1 Building open analyses

There are things companies do not want to share with the entire world. From design secrets to the finer details of how specific components work, a great deal of information is filtered out before a product reaches the end customer. However, to reduce the burden on customer service and respect consumer rights, many fragments of risk analysis are actually shared for small repairs and troubleshooting. Figure IV.23 shows how manufacturers such as Dell share them in a textual, human-readable way. As page 191 of the figure mentions, some repairs and troubleshooting "may only be done by a certified service technician", which has access to more detailed instructions, and thus a more thorough view of the risk analysis.

It may therefore be advisable to add an access level mechanism to risk analyses, enabling only certain information to be shared with customers, service technicians and internal teams for example. The fact that the structure of risk analyses is closely linked to that of the components in a system makes it easy to implement such a mechanism, as shown in figure IV.24. In this figure, we represent an assembly of components along with a risk analysis built on three access levels. Components which can be diagnosed by the end customer are in `Access level 1`, those reserved for certified technicians are in `Access level 2`, and those reserved for the company's internal teams are in `Access level 3`.

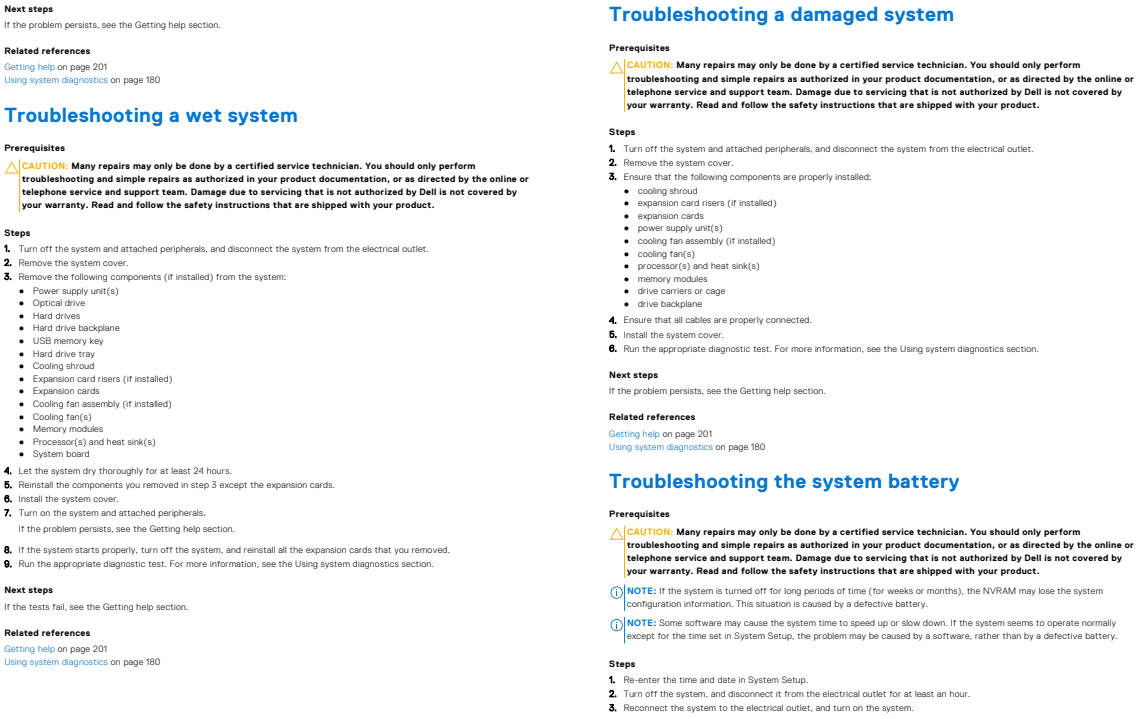
Once risk analyses and their respective permissions are properly encoded, companies can generate troubleshooting guides semi-automatically by defining main troubleshooting categories and extracting publicly-disclosable data in a systematic way.

IV.5.2 Composing analyses

In the same way that components can be assembled to form systems, their analyses can be combined to study these systems. Combining the analyses is a three-stage process:

1. Gathering analyses of the different components of the system;
2. Producing an analysis of the assembled system;
3. Defining access levels for sharing the resulting analysis.

Because risk analysis is a constantly evolving discipline, we consider that the various shared analyses should not remain static. By providing a dynamic catalog of components and their risk analyses, companies can continuously update their knowledge bases, and the assemblies of these components can benefit from these changes.



Next steps

If the problem persists, see the Getting help section.

Related references

Getting help on page 201
Using system diagnostics on page 180

Troubleshooting a wet system

Prerequisites

CAUTION: Many repairs may only be done by a certified service technician. You should only perform troubleshooting and simple repairs as authorized in your product documentation, or as directed by the online or telephone service and support team. Damage due to servicing that is not authorized by Dell is not covered by your warranty. Read and follow the safety instructions that are shipped with your product.

Steps

1. Turn off the system and attached peripherals, and disconnect the system from the electrical outlet.
2. Remove the system cover.
3. Remove the following components (if installed) from the system:
   - Power supply unit(s)
   - Optical drive
   - Hard drives
   - Hard drive backplane
   - USB memory key
   - Hard drive tray
   - Cooling shroud
   - Expansion card risers (if installed)
   - Expansion cards
   - Cooling fan assembly (if installed)
   - Cooling fan(s)
   - Memory modules
   - Processor(s) and heat sink(s)
   - System board
4. Let the system dry thoroughly for at least 24 hours.
5. Reinstall the components you removed in step 3 except the expansion cards.
6. Install the system cover.
7. Turn on the system and attached peripherals.

   If the problem persists, see the Getting help section.
8. If the system starts properly, turn off the system, and reinstall all the expansion cards that you removed.
9. Run the appropriate diagnostic test. For more information, see the Using system diagnostics section.

Next steps

If the tests fail, see the Getting help section.

Related references

Getting help on page 201
Using system diagnostics on page 180

190 Troubleshooting your system

Troubleshooting a damaged system

Prerequisites

CAUTION: Many repairs may only be done by a certified service technician. You should only perform troubleshooting and simple repairs as authorized in your product documentation, or as directed by the online or telephone service and support team. Damage due to servicing that is not authorized by Dell is not covered by your warranty. Read and follow the safety instructions that are shipped with your product.

Steps

1. Turn off the system and attached peripherals, and disconnect the system from the electrical outlet.
2. Remove the system cover.
3. Ensure that the following components are properly installed:
   - cooling shroud
   - expansion card risers (if installed)
   - expansion cards
   - power supply unit(s)
   - cooling fan assembly (if installed)
   - cooling fan(s)
   - processor(s) and heat sink(s)
   - memory modules
   - drive carriers or cage
   - drive backplane
4. Ensure that all cables are properly connected.
5. Install the system cover.
6. Run the appropriate diagnostic test. For more information, see the Using system diagnostics section.

Next steps

If the problem persists, see the Getting help section.

Related references

Getting help on page 201
Using system diagnostics on page 180

Troubleshooting the system battery

Prerequisites

CAUTION: Many repairs may only be done by a certified service technician. You should only perform troubleshooting and simple repairs as authorized in your product documentation, or as directed by the online or telephone service and support team. Damage due to servicing that is not authorized by Dell is not covered by your warranty. Read and follow the safety instructions that are shipped with your product.

NOTE: If the system is turned off for long periods of time (for weeks or months), the NVRAM may lose the system configuration information. This situation is caused by a defective battery.

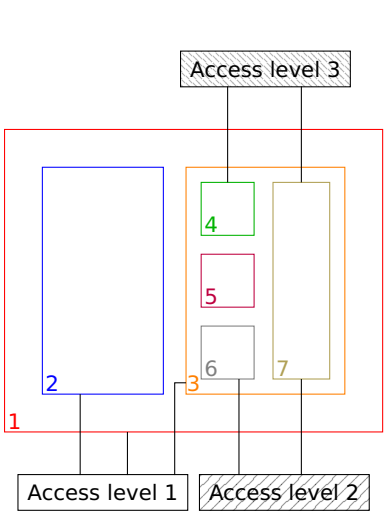
NOTE: Some software may cause the system time to speed up or slow down. If the system seems to operate normally except for the time set in System Setup, the problem may be caused by a software, rather than by a defective battery.

Steps

1. Re-enter the time and date in System Setup.
2. Turn off the system, and disconnect it from the electrical outlet for at least an hour.
3. Reconnect the system to the electrical outlet, and turn on the system.

Troubleshooting your system 191

Figure IV.23: Excerpt from the "Troubleshooting your system" section of Dell PowerEdge R730 Owner's Manual



ID	Component	Failure	Action	Consequence	
a	1	...	...	...	Category 1
b	1	...	...	...	
c	1	...	...	...	
d	2	...	...	b	Category 2
e	3	...	...	a	
f	3	...	...	b	
g	3	...	...	...	
h	4	...	...	...	Category 3 (Internal)
i	4	...	...	...	
j	5	...	...	e	
k	6	...	...	c	Category 4 (Certified tech.)
l	6	...	...	f	
m	7	...	...	...	
n	7	...	...	...	

Figure IV.24: Multi-level open risk analysis

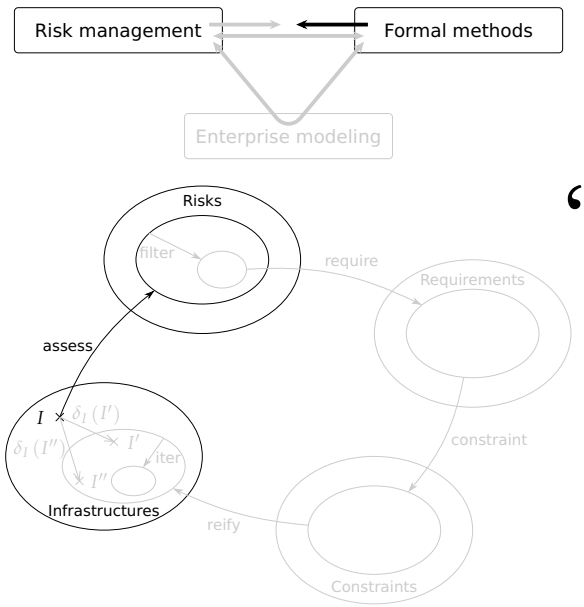
IV.6 Conclusion

Risk management is a very diverse discipline, which has been the subject of a great number of contributions in the literature. In this chapter, we have outlined the entire process, giving various recommendations on how to perform analyses, as well as suggestions for future developments. This chapter shows that the process is still very manual, and that it can be cumbersome to navigate between the various sources of truth. When it comes to security, a topic covered extensively, it is a perpetual cat-and-mouse game that can quickly become asymmetrical: companies react to the various CVEs, but it is impossible to guard against everything (many attacks and software actually do not exist yet). When it comes to safety, it is much easier to predict every eventuality, but remediation can be very costly when it comes to the physical world; furthermore, the field is much less covered by generic IT frameworks, at the risk of relegating it to second tier status.

But risk management does not stop with what we have presented in this chapter, and we can learn a lot about the discipline from the perspective of other scientific communities. This is what we explore in the rest of this dissertation.

Chapter V

Checking IT Infrastructures



> “ Galen Tyrol — Verify PC-2 pressure zero, throttles closed, oxygen generator and master switches off.
>
> Check. Shutdown complete. — Kara Thrace
>
> Galen Tyrol — Nothing you could do, captain. Too far away. ”
>
> *Battlestar Galactica – Scar*

Contents

V.1 Rethinking risk assessment . . . 56
V.1.1 Formalism . . . 56
V.1.2 Risk and properties . . . 58
V.2 Modeling IT Infrastructures . . . 58
V.2.1 From the technical world... . . . 60
V.2.2 ... to the formal one . . . 62
V.2.3 Case study . . . 64
V.3 Model checking . . . 69
V.3.1 Properties and checkers . . . 70
V.3.2 The need for proper abstractions . . . 72
V.3.3 Going back to our case study . . . 72
V.4 Automating risk assessment . . . 75
V.4.1 Expressing formal properties... . . . 75
V.4.2 ... and combining models together . . . 76
V.5 Conclusion . . . 77

Now that we have set out the theoretical framework for our study and introduced risk management for IT infrastructures, we want to focus on the risk assessment phase, which we think has to face major semantic misalignments. The infrastructure, whether technical, human or organizational, is assessed according to criteria subject to auditors' interpretation, using a common taxonomy. This assessment is influenced by the way infrastructure facts are presented, and the subsequent mental representation of these facts by auditors. We are trying to reduce this human evaluation in the assessment of safety and security properties. Moreover, verifying such properties on large IT infrastructures can be a daunting and costly task. To simplify the process and alleviate various concerns related to human judgement, we consider automation as an advisable step forward in risk assessment, which involves three key steps:

1. The development of infrastructure models;
2. The verification of properties on these models;
3. The interpretation of these properties in the risk taxonomy.

In this chapter, we adopt a trial-and-error approach, with the help of a case study, to produce an infrastructure's risk assessment, drawing relevant conclusions along the way. As we are effectively concerned with refining the `assess` function described in the previous chapter, we start this chapter by adapting our formalism in section V.1. We then present different ways of modeling IT infrastructures and introduce our case study in section V.2. Next, we show how to verify properties on models and apply these methods to our case study in section V.3. We then show how we can add automation to the process through formal interpretation of properties and model combination in section V.4. Finally, we conclude this chapter in section V.5.

V.1 Rethinking risk assessment

Regardless of the risk analysis framework adopted, the human factor plays an important role in the process, from establishing the scope of the study to assessing the criticality of risks. The various parties involved in the analysis represent the different areas of the company in which risk assessment is, at the very least, desirable, and at the most, mandatory. With these different areas come different expertises, different mental and technical *models* of what the infrastructure is and does, on which risk assessment is based. Here, we formalize this notion of models and introduce the concept of model checkers, to verify properties on these models.

V.1.1 Formalism

A model is a simplified representation of a component or system, created to analyze, understand, or predict its behavior under various conditions. Here, we consider that models have inputs, which we call *parameters*, and outputs, which can be *results* directly obtained from the model, or *properties* to be checked. Parameters can be used to make models generic and instantiate them under different conditions (for example, a model studying the safety of an n-element system, with n as a parameter). Results are data produced by or extracted from a model (for example, an artifact produced by a neural network). Finally, properties represent facts that can be verified by so-called *model checkers*. For example, if a real function $f : x \mapsto 1 - x^2$ models the behavior of a system, a model checker could verify the property $p : \forall x, f(x) < 2$. This formalism is represented on figure V.1, which additionally shows that certain model checkers can check certain kinds of properties (and not others).

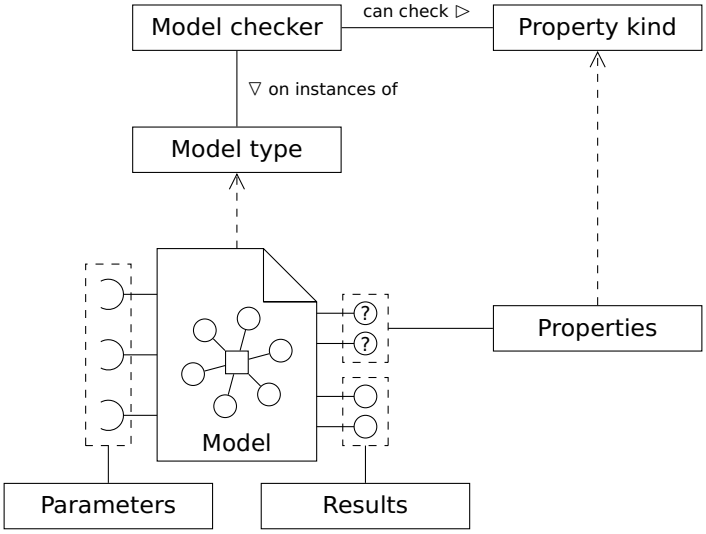


Figure V.1: Model of a model and its properties
Legend. □—□ Relationship, □-≻□ Instantiation

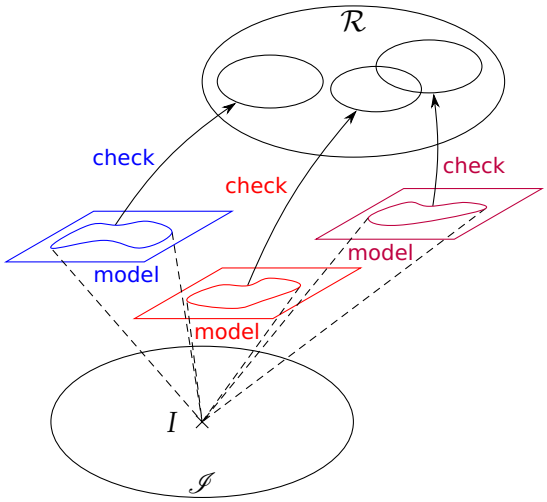


Figure V.2: Refinement of the assess function shown on figure IV.5 (page 32) into sub-model and check functions

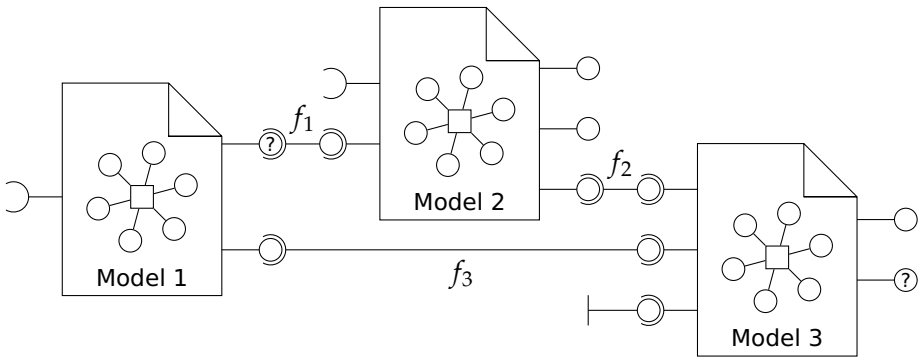


Figure V.3: Composition of models. f_1, f_2 and f_3 transform outputs into suitable inputs for the next model.

In this chapter, we consider the assess function of the previous chapter to be the composition of a model function, in which the systems under study are modeled, and a check function, in which the models are verified. This decomposition is represented in figure V.2.

We define a model as a 4-tuple $M = (I, m, \mathscr{P}, R)$, where I is the set of input parameters, m is the model definition (that we leave opaque in this formalization), $\mathscr{P}$ is a set of properties to prove, and R is the set of results. A model checker C is an application that takes a model and its parameters to produce a verdict on its properties. Formally, if $(v_i)_{i\in I}$ is a family of input values indexed by I, $C : (I, m, \mathscr{P}, R), (v_i)_{i\in I} \mapsto \Pi$, where $\Pi \in \{\top, \bot, ?\}^{\mathscr{P}}$ is the set of verdicts of the form:

$$p \mapsto \begin{cases} \top & \text{if } p \text{ is proved to be true by the model checker,} \\ \bot & \text{if } p \text{ is proved to be false by the model checker,} \\ ? & \text{if no verdict can be reached for } p \text{ by the model checker.} \end{cases}$$

A "?" verdict for a property p may indicate that:

- p has a property kind that the model checker is unable to verify;
- Spatial or temporal limits have been exceeded, stopping the checking process;
- The property is undecidable.

To illustrate this definition, if we reuse the function f previously mentioned and consider a model $M = (\varnothing, f, \{p_i : \forall x, f(x) < i \mid i \in \mathbb{N}\}, \varnothing)$ and a checker C able to solve this mathematical problem, we can get $C(M, \varnothing) = \{p_i \mapsto [i > 1] \mid i \in \mathbb{N}\}$. The global verdict ($\Pi$ in the definition) is important, since it forms the basis on which we can draw conclusions about the safety and security of the system under study, *i.e.* the actual risk assessment.

V.1.2 Risk and properties

As we discussed in section IV.5, component failure modes can be combined in fault trees for risk analyses. Similarly, risks on portions of an infrastructure can be expressed as a logical combination of properties verified on their models. For example, if we consider the risk r : "Unsafe overheating of equipment", we can break it down as $r = \neg\mu \vee (\neg\varphi \wedge \neg\sigma)$, where μ is "Correct measurement of the temperature T", φ is "Fans activate when $T > T_1$" and σ is "Equipment shuts down when $T > T_2$". These properties can be further broken down as logical combinations of sub-properties. Determining whether a system is subject to a risk involves finding a verification path (*i.e.* a means of verifying inductively the properties) in a ternary verdict tree (using the standard Kleene's K_3 three-valued logic — *true*, *false*, and *unknown*) to show the presence or absence of the risk.

Breaking down the risk into verifiable sub-goals can be used to feed an interactive verification assistant: if the valuation of a set of properties is required to establish a risk verdict, the assistant can suggest a set of model checkers able to verify such properties along with types of models fulfilling the need. To know which model checkers can verify them, properties must first be assigned property kinds; we provide a classification of these in section V.3. In our example r, if μ and φ provably hold, we could refine σ into lower-level properties that the verification assistant could link to suitable models checkers.

Finally, we can chain the models together to produce richer models, as shown on figure V.3, which is what this chapter shows in order to build a complete infrastructure model.

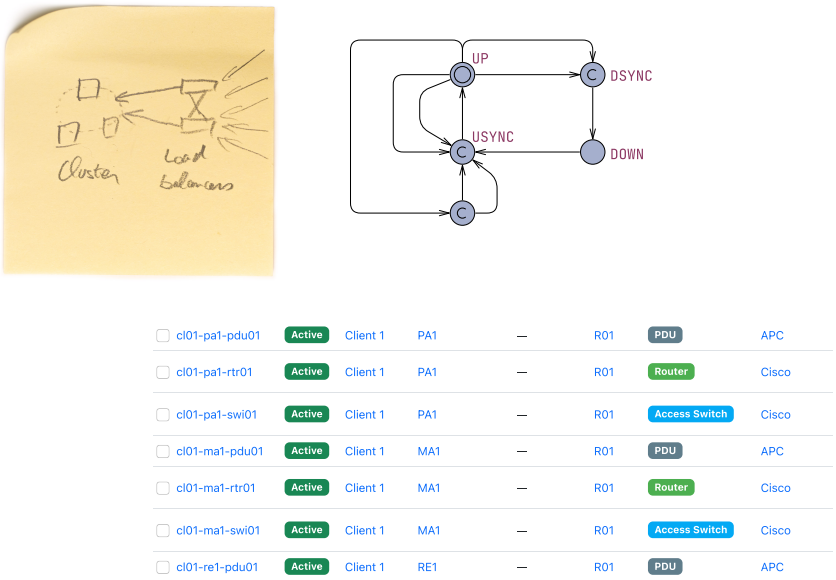


Figure V.4: A scribble, an automaton, a hardware inventory: three models

V.2 Modeling IT Infrastructures

Modeling part of an infrastructure allows to extract, on a specific "view", characteristic elements deemed relevant to study or represent. In reality, everything can be seen as a model [Sandkuhl18], from a scribble on a piece of paper describing an architectural idea, to an inventory of computers and their interconnections, to formal automata capturing behavioral properties of a system (figure V.4).

We have identified a number of criteria on which models can vary:

- Nature, what the model is:
  - descriptive, to represent what *is*,
  - prescriptive, to represent what *should be*,
  - predictive, to represent what *could be*;
- Purpose, why the model exists:
  - informational, as a way to convey insights,
  - functional, as a working tool,
  - theoretical, as a basis for demonstration and reasoning;
- Type, how the model is presented:
  - graphical, a visual representation of a system,
  - mathematical, a formal representation,
  - simulational, an executable model;
- Scope, what the model covers:
  - behavioral, to study system dynamics,
  - structural, to study system architecture,
  - operational, to study organizations and processes.

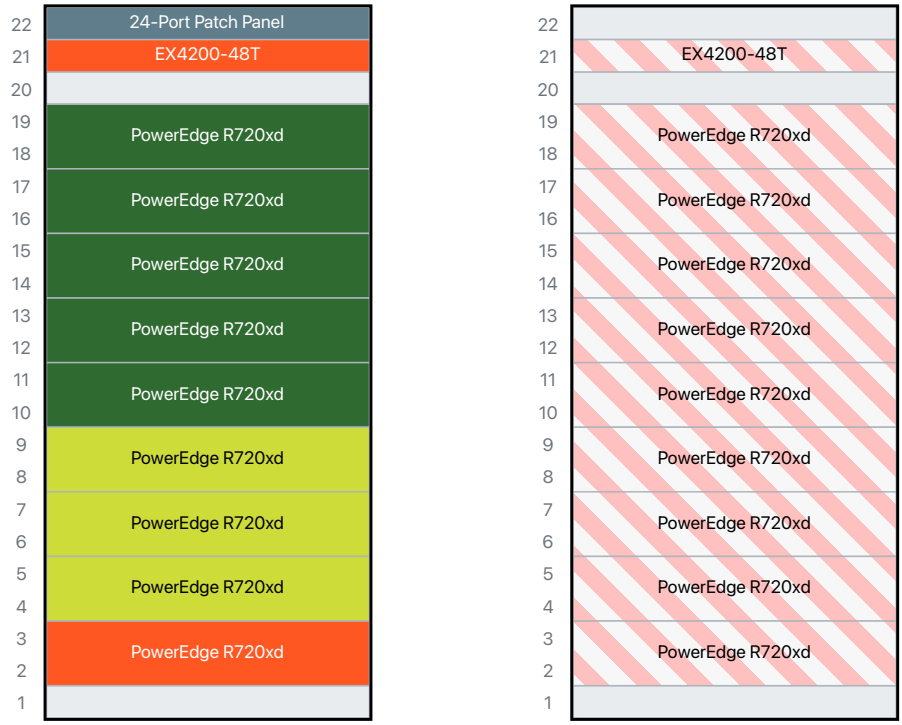


Figure V.5: NetBox model of a 22 U datacenter rack (front and rear of the rack)

Models that do not cover the same criteria may represent different complementary aspects of the infrastructure. For example, a descriptive, informational, graphical, structural model may be used to train staff about a company's IT architecture, while a predictive, theoretical, mathematical, behavioral model may be used to estimate future network traffic. To carry out an in-depth risk analysis, it is therefore important to carefully select models appropriate to the particular aspects being studied. In this section, we give a brief overview of modeling, from the technical to the formal world, before presenting our case study.

V.2.1 From the technical world...

In datacenters, the vast amount of components, both hardware and software, has led to the emergence of specialized tools for projecting the infrastructure onto various "views" that can be exploited by different business areas. This approach led to the creation of models, serving as aids for professionals to visualize and grasp the intricacies of complex infrastructures. Data Center Infrastructure Management (DCIM) software like [NetBox], which we use in our case study, are an example of such tools, providing visual representations simplifying resource planning and management. A NetBox model of a datacenter rack is given in figure V.5. We like to label such models as "incidental", since they result from a business process and not from a pure modeling endeavor.

To add structure to infrastructure models, initiatives such as the Simple Network Management Protocol ([SNMP], by the IETF) and the Common Information Model ([CIM], by the DMTF) were introduced respectively in 1988 and around 1997. The former behaves as a tree-like datastore linking standard and proprietary object identifiers (OID) to their values and is mainly used for network monitoring (and to a lesser extent, system monitoring). An example of an "SNMP walk" on a network switch is given in figure V.6. The latter, along with Web-Based Enterprise Management (WBEM), aim to provide a

```
SNMPv2-MIB::sysDescr.0 = STRING: Juniper EX4600-40F-AFO
SNMPv2-MIB::sysObjectID.0 = OID: SNMPv2-SMI::enterprises.2636.1.1.1.2.109
DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (739441167) 85 days, 14:00:11.67
IF-MIB::ifNumber.0 = INTEGER: 852
IF-MIB::ifIndex.4 = INTEGER: 4
IF-MIB::ifIndex.5 = INTEGER: 5
IF-MIB::ifIndex.6 = INTEGER: 6
IF-MIB::ifIndex.7 = INTEGER: 7
IF-MIB::ifIndex.8 = INTEGER: 8
IF-MIB::ifIndex.9 = INTEGER: 9
```

Figure V.6: Excerpt from an SNMP walk over a live Juniper EX4600-40F-AFO switch

Caption	CommandLine	Name	CreationClass...	CreationDate	Description	ExecutablePath
AggregatorHos...		AggregatorHo...	Win32_Process	14/09/2023 04:14:44	AggregatorHo...	
AppVShNotify....		AppVShNotify....	Win32_Process	17/09/2023 11:51:03	AppVShNotify....	
csrss.exe		csrss.exe	Win32_Process	14/09/2023 04:14:43	csrss.exe	
csrss.exe		csrss.exe	Win32_Process	14/09/2023 04:14:43	csrss.exe	
csrss.exe		csrss.exe	Win32_Process	03/10/2023 08:35:07	csrss.exe	
csrss.exe		csrss.exe	Win32_Process	03/10/2023 08:47:21	csrss.exe	
csrss.exe		csrss.exe	Win32_Process	03/10/2023 09:22:22	csrss.exe	
ctfmon.exe		ctfmon.exe	Win32_Process	03/10/2023 08:35:09	ctfmon.exe	
ctfmon.exe		ctfmon.exe	Win32_Process	03/10/2023 08:47:22	ctfmon.exe	
ctfmon.exe		ctfmon.exe	Win32_Process	03/10/2023 09:22:24	ctfmon.exe	
dllhost.exe		dllhost.exe	Win32_Process	14/09/2023 04:14:44	dllhost.exe	
dllhost.exe		dllhost.exe	Win32_Process	14/09/2023 04:14:54	dllhost.exe	
dllhost.exe		dllhost.exe	Win32_Process	03/10/2023 08:47:51	dllhost.exe	
dllhost.exe		dllhost.exe	Win32_Process	03/10/2023 10:53:42	dllhost.exe	
dllhost.exe	C:\Windows\system32\DllHost.exe /P...	dllhost.exe	Win32_Process	03/10/2023 10:58:00	dllhost.exe	C:\Windows\system32\DllHost.exe
dwm.exe		dwm.exe	Win32_Process	14/09/2023 04:14:43	dwm.exe	
dwm.exe		dwm.exe	Win32_Process	03/10/2023 08:35:08	dwm.exe	
dwm.exe		dwm.exe	Win32_Process	03/10/2023 08:47:21	dwm.exe	
dwm.exe		dwm.exe	Win32_Process	03/10/2023 09:22:23	dwm.exe	

Figure V.7: Excerpt from a list of `Win32_Process` instances in the namespace `root\CIMV2` for a Windows 7 system

more expressive representation of IT systems and their relationships, and are mainly used by major infrastructure players such as Microsoft, IBM and Oracle. An example of a process inventory on a Windows server is given in figure V.7.

As datacenters transitioned from traditional to virtualized infrastructures, models have begun to abstract the physical reality of infrastructures, so that the expression of needs becomes less dependent on the underlying hardware. Among the trends that have emerged as a consequence of this shift, we can mention Software-Defined Networking (SDN) [Masoudi16] and Infrastructure as Code (IaC) [Morris20]. Both approaches tend towards a modeling process, with various stakeholders describing infrastructure elements in a prescriptive way. In the case of SDN, network operators model desired virtual network topologies and behavior; in the case of IaC, IT architects describe how to provision IT resources to run applications. With these, modeling takes a more “intentional” form, as the model drives the initiative.

Finally, more systematic and structured technical approaches exist, notably in the field of software development, with frameworks relying on the Unified Modeling Language ([UML]). Rigorous use of UML helps build rich infrastructure models that can be further enhanced by the use of formal methods, which we present now.

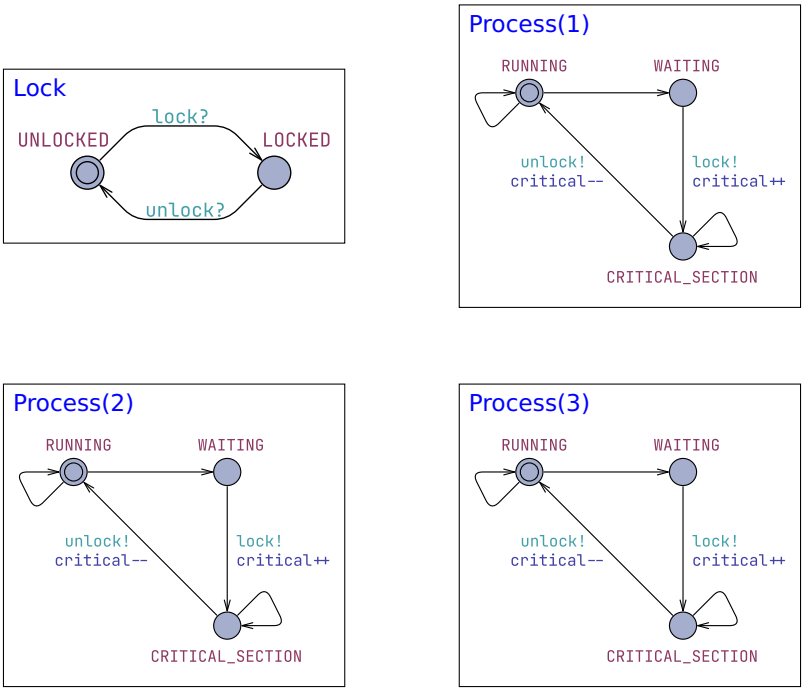


Figure V.8: UPPAAL model of a lock and processes able to acquire it

V.2.2 ... to the formal one

While modeling can provide a clearer vision of infrastructures, the lack of mathematical rigor in most tools and representations makes it difficult to use infrastructure models in a systematic way. Formal methods provide a sound theoretical framework for the study of systems, and we present here some of those which we consider of particular importance in our work. We are intentionally leaving aside methods that are not related to modeling, such as code verification, mentioned in section IV.2.3, and general-purpose solvers such as Z3, presented in section VI.3.

Building systems safe and secure by design is one of the use cases where formal methods excel, and B method [Abrial96] is a prime example of this. B method guides the design of provably reliable systems in a top-down fashion by leveraging logic and set theory for unambiguous specification of properties and behaviors. It adds the concept of proof obligation to models, requiring mathematical proof that each refinement of a model preserves the requirements of the previous modeling steps. Our approach to risk analysis in this chapter is very close to that of B-method for system design. As another formal specification language, we can also mention Alloy [Jackson11], which allows for the description of complex system structures, behaviors, and invariants in a high-level, abstract manner.

Whether for descriptive or prescriptive modeling, automata are used to represent the possible states of a system and transitions between them. These abstract representations make it possible to rigorously explore and analyze all possible behaviors of the system. Tools such as UPPAAL, in which we have modeled a lock system shown in figure V.8, allow to design and compose complex systems using automata. It constitutes an important tool for our case study.

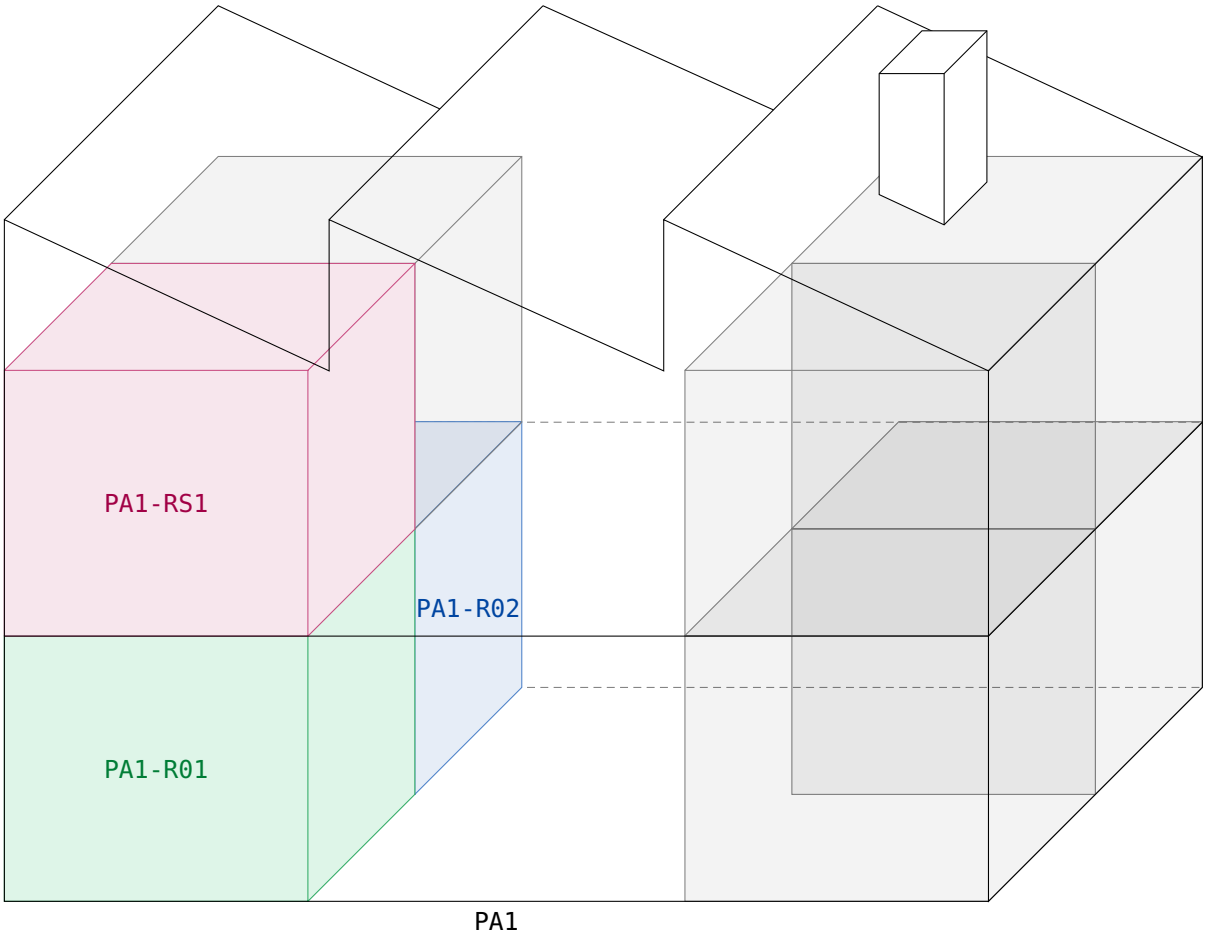


Figure V.9: Overview of the `PA1` datacenter, with the three rooms `PA1-R01`, `PA1-R02` and `PA1-RS1`

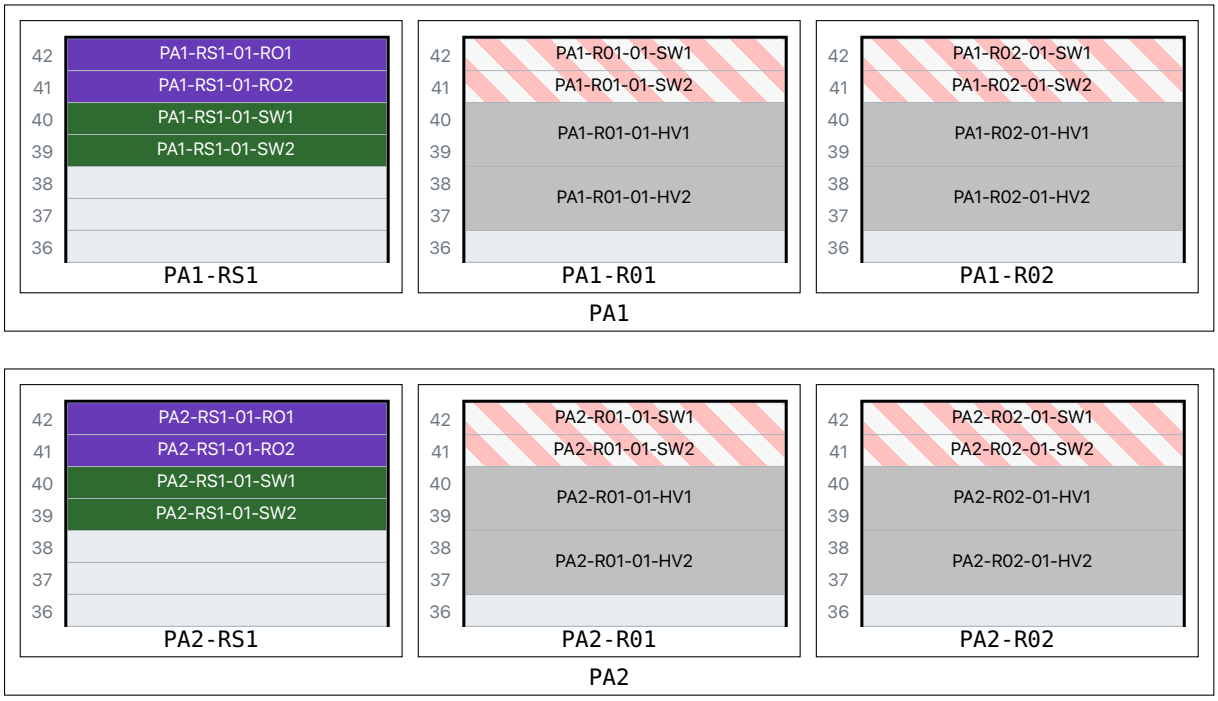


Figure V.10: Physical infrastructure modeled with NetBox

V.2.3 Case study

In this subsection, we propose a case study that serves as a common thread for this chapter. We consider a small fictitious virtualization infrastructure composed of eight servers distributed over two datacenters. All eight servers run the Proxmox Virtual Environment ([Proxmox VE]) hypervisor, hosting virtual machines supporting critical business activities for a company. For this case study, we consider three views of the infrastructure which we model with NetBox and UPPAAL: a physical view, a network view, and a functional view.

Physical infrastructure

The infrastructure we study is made of two datacenters located in Paris, named `PA1` and `PA2`. Figure V.9 gives an overview of `PA1`. We consider in each of these datacenters three rooms:

- $\texttt{PA}_2^1\texttt{-R01}$ and $\texttt{PA}_2^1\texttt{-R02}$, containing each two servers and two so-called *Top-of-Rack* (ToR) switches,
- $\texttt{PA}_2^1\texttt{-RS1}$ containing so-called *aggregation* switches and routers to interconnect the datacenters.

The rooms of each datacenter have been described on NetBox by the company's datacenter technicians. We give in figure V.10 an excerpt from NetBox' representation of the datacenter, highlighting the elements considered in this case study. Each datacenter rack is represented by a rack diagram (which is the *de facto* standard representation in the field), in which we have colored the routers purple, the aggregation switches green, the ToR switches hatched pink and the servers gray.

For the purposes of this study, we consider each room electrically independent from one another, although a general electrical incident on a whole datacenter can impact all its rooms.

Network infrastructure

These various pieces of equipment are linked together to form a network. Both datacenters can communicate with each other thanks to an L2 VPN[1]. Each server is redundantly connected to two stacked ToR switches and share the same broadcast domains[2]. In our example, the `PA1-R01-01-HV1` server is linked to both the `PA1-R01-01-SW1` and `PA1-R01-01-SW2` switches. Each ToR switch is linked to an aggregation switch (for example, `PA1-R01-01-SW1` is linked to `PA1-RS1-01-SW1`), which in turn is redundantly linked to two routers (for example, `PA1-RS1-01-SW1` is linked to both `PA1-RS1-01-RO1` and `PA1-RS1-01-RO2`). The network topology presented in figure V.11 is directly extracted from NetBox (device names correspond to figure V.10).

Functional infrastructure

Proxmox Virtual Environment is a hypervisor, which is a platform enabling the creation of a virtualized execution environment, notably capable of running so-called virtual machines. Many hypervision solutions (such as Proxmox VE) enable servers (referred to as *nodes*) to be grouped together in so-called *clusters*, making infrastructures more resilient. In particular, such clusters enable virtual machines to

[1] A VPN at the layer 2 of the OSI model

[2] They can "see" one another at the layer 2, thanks to the L2 VPN

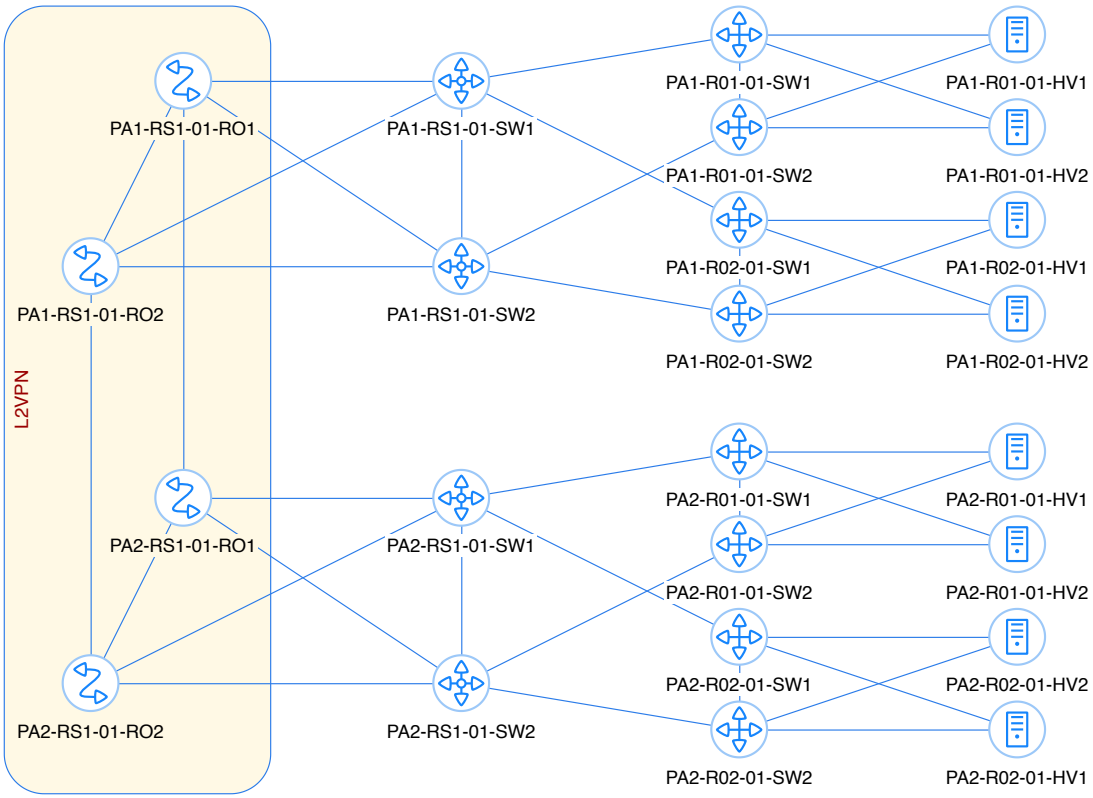


Figure V.11: Network infrastructure extracted from NetBox

be restarted, in the event of a failure of some nodes, onto other nodes. To this end, Proxmox VE uses [Corosync], an open-source cluster communication library, as its inter-node communication system, whose operation can be summarized as follows.

At regular intervals, the nodes try to communicate with one another, and if the number of nodes that "see" one another exceeds a certain score (called minimum votes), they form a so-called *quorum* and heuristically elect a *master*. As long as the quorum exists, the cluster can continue to operate. Figure V.12 shows an example of the loss of a quorum for a 5-node cluster with 3 required votes. Initially, all nodes see one another, but following a succession of events (losses and recoveries of nodes), only two nodes can communicate and the quorum is broken. In some cases, a cluster can have more than one quorum, for example if we reduce the required votes to 2 and two pairs of servers that see each other are isolated. This behavior is usually undesirable and called a "split-brain".

We have implemented in UPPAAL several behavioral models for Corosync, one of which is shown in figure V.13. The source files for the complete models are available on Github[3]. A cluster (figure V.13a) is modeled as a system which can either be `QUORATE` or `INQUORATE`, depending on whether its members (here, nodes) reach a quorum (`quorums() > 0`) or not (`quorums() == 0`). A simplified implementation, limited to one quorum, is given in appendix B.1.2. To detect several quorums, we have adapted the Tarjan's strongly connected components algorithm[4] [Tarjan72] to UPPAAL, by reimplementing it in a non-recursive way. The algorithm allows us to identify isolated groups of nodes that can communicate with one another, thus able to form small quorums. Its implementation is given in appendix B.1.3.

[3] `https://github.com/CAPRICA-Project/UPPAAL-models`

[4] Algorithm used for finding the strongly connected components of a directed graph in linear time. Strongly connected components are subgraphs that are strongly connected, that is, where each vertex is reachable from the others.

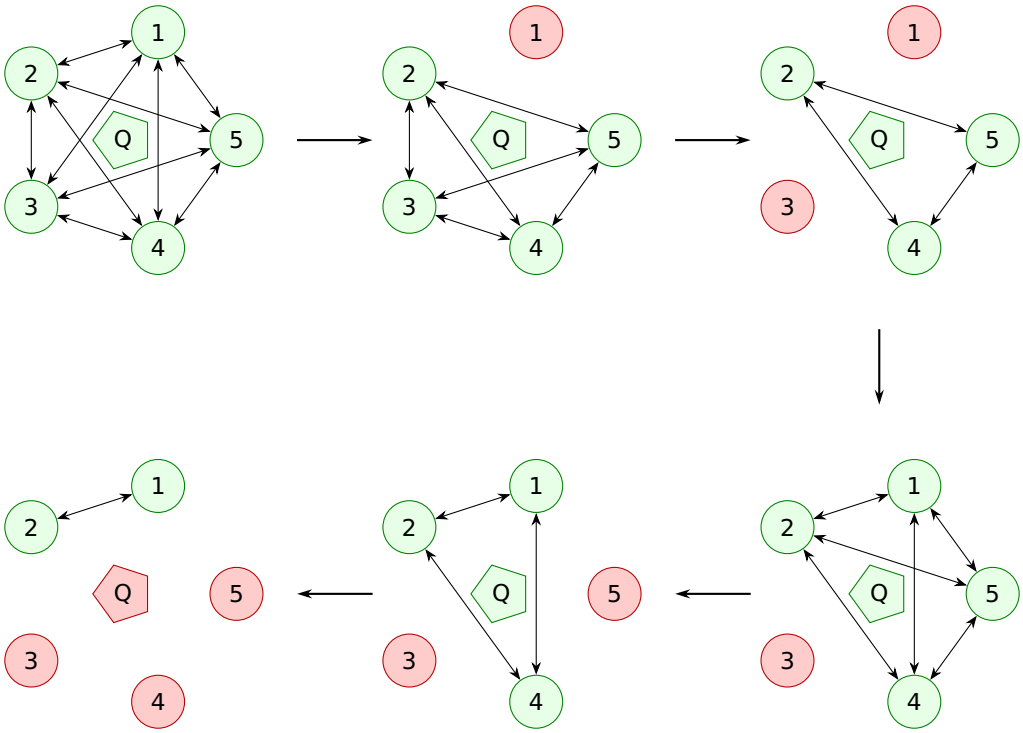
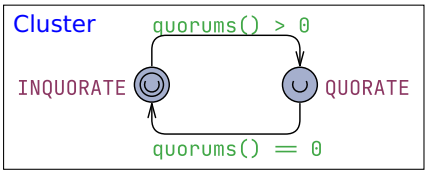


Figure V.12: Loss of a 5-node quorum with 3 votes
Legend. ○ Up, ○ Down, ⬡ Quorum

(a) Corosync cluster model



(b) Corosync node model

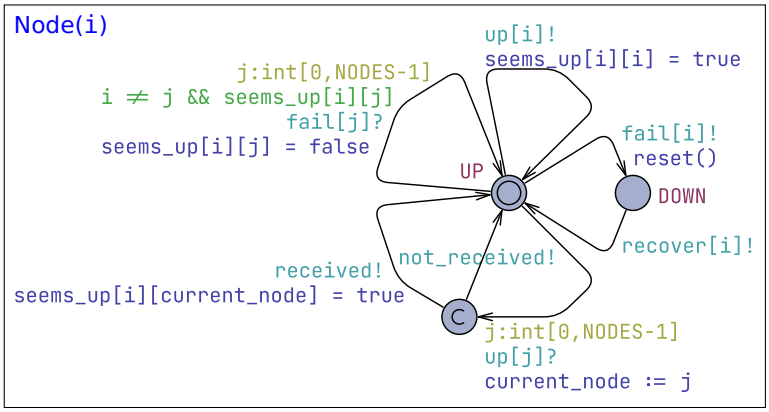


Figure V.13: UPPAAL model of Corosync

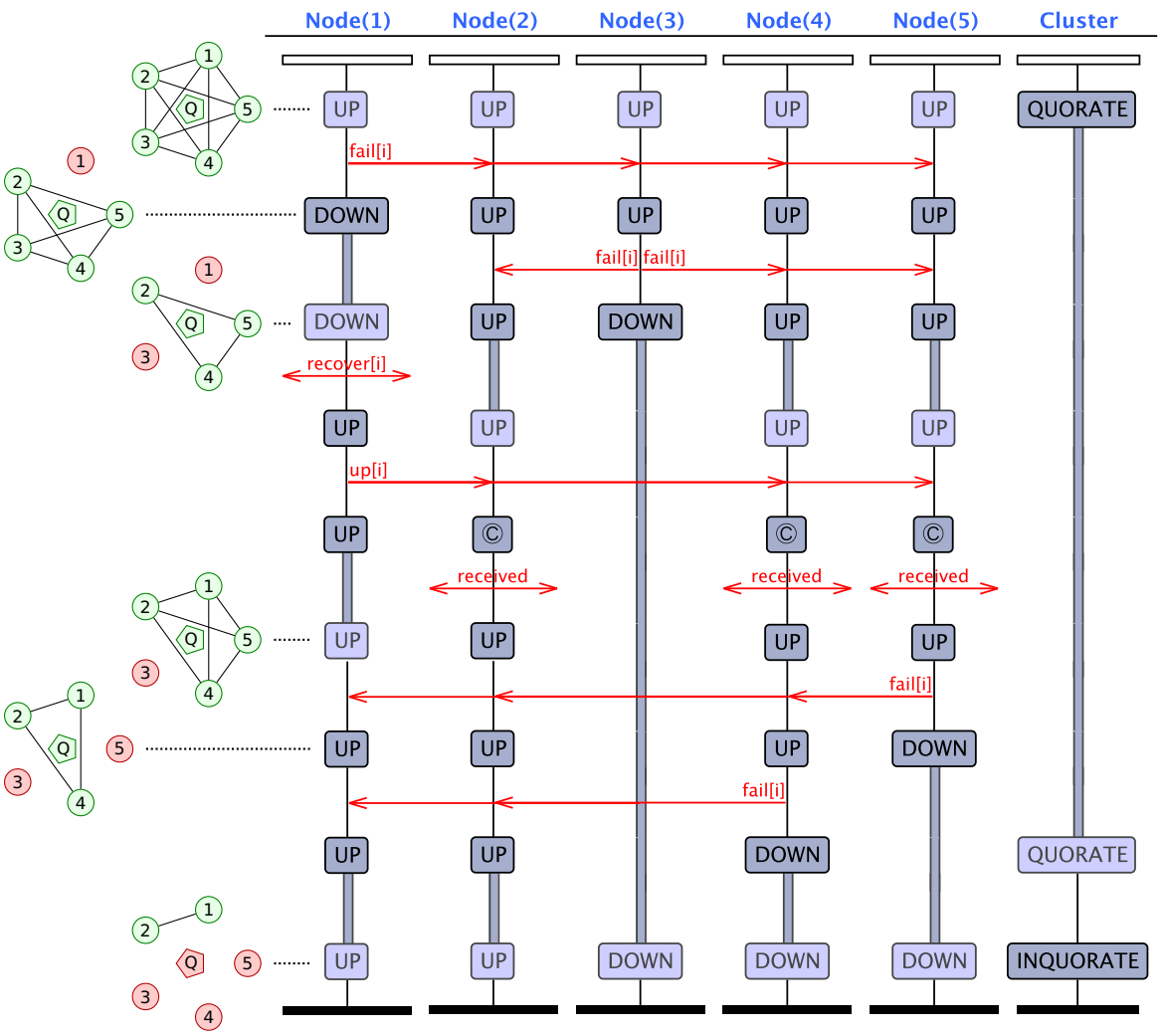


Figure V.14: Trace of the loss of a 5-node quorum with 3 votes

Nodes (figure V.13b) can advertise their presence (`up[i]!`[5]) to the others (`up[j]?`[6]), which may or may not receive it (`received!` or `not_received!`). They can fail (`fail[i]!`), which is detected by the other nodes (`fail[j]?`), and recover from a failure (`recover[i]!`). `seems_up[i][j]` represents whether or not node i thinks node j is up. Finally, `reset()` is an internal function taking care of resetting some variables when a node fails. The complete UPPAAL code is given in appendix B.1.4.

We show a UPPAAL simulation trace corresponding to the loss of quorum scenario shown in figure V.12 in figure V.14.

Composition

We now have two technical models (the physical infrastructure and the network infrastructure) and a formal model (the functional infrastructure). We want to get a complete view of the infrastructure from these models. On the one hand, technical models are obtained from specialized tools used on a daily basis within the company; it is important to avoid interfering with the employees' habits by modifying

[5]The exclamation mark means here "send a message to everyone"
[6]The question mark means here "receive a message"

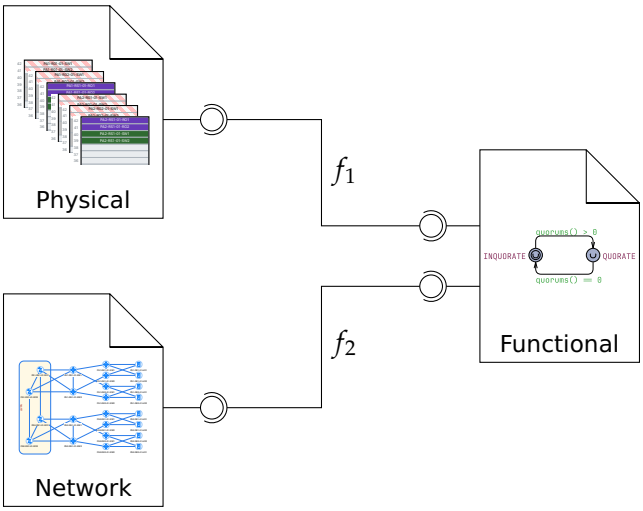
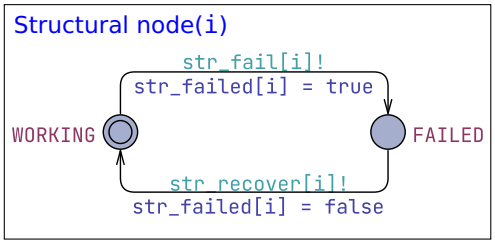


Figure V.15: Linking technical models to a formal model

(a) Structural node model



(b) Network node model

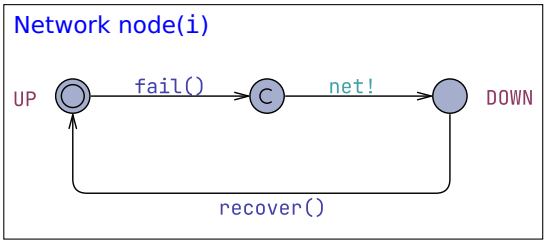


Figure V.16: UPPAAL transitional models

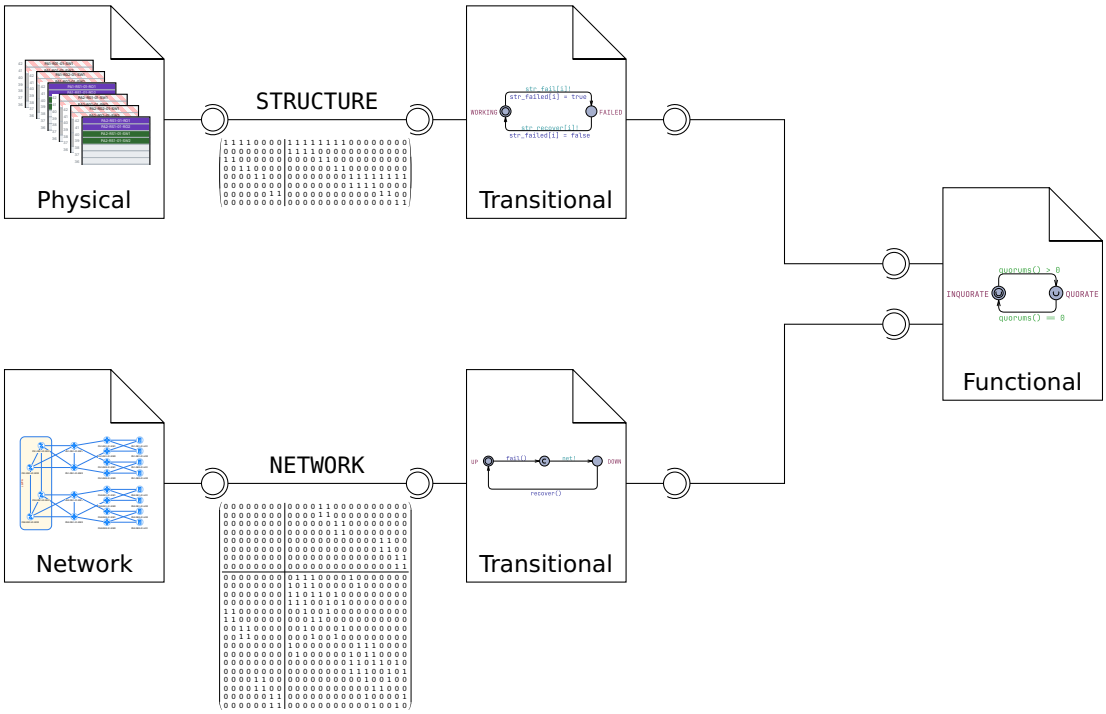


Figure V.17: Linking technical models to a formal model thanks to transitional models

the tools or the models they produce. On the other hand, it is impossible to feed these models directly to our formal model. We want to link the models as shown in figure V.15, and therefore need to transform the technical ones by defining the f_1 and f_2 transformations.

First, we want to integrate the physical topology to take into account failures caused by external events related to the physical location of nodes (power loss, earthquakes...). We model a physical room or datacenter as a simple state machine that can either be `WORKING` or `FAILED`, and call it a "structural" node. The model is shown on figure V.16a. Due to any reason, such nodes can fail (`str_fail[i]!`) and work again (`str_recover[i]!`). We keep track of the failed structural nodes in the `str_failed` array. These nodes can be contained by other nodes (for example, a room inside a datacenter), so we keep track of these relationships in an adjacency matrix, `STRUCTURE`.

Then, we want to deal with the network aspects of the infrastructure. We model network devices as so-called "network" nodes, shown on figure V.16b. These nodes can be `UP` or `DOWN`, and `fail` and `recover` from a failure. We use the `net` channel to inform other nodes that the network state has changed. We keep track of an adjacency matrix, `can_communicate`, which is updated each time a network node's state changes. To update this matrix, we run the Floyd–Warshall algorithm[7] [Floyd62] to identify which nodes are able to communicate with which. The network topology is stored as a matrix `NETWORK`. The code for the model is given in appendix B.1.5.

The final model assembly is shown on figure V.17, in which we have integrated the models shown in figure V.16 (called "transitional" models). For the sake of clarity, we have skipped over a number of details concerning the synchronization of `str_fail`, which the curious reader can find in the model's source code online. The code for the assembled model is given in appendix B.1.6.

Now that we have all the models that we need, it is time to take a look at the various verifications we can perform on them.

V.3 Model checking

But first, a little context. From the technical to the more formal models presented in the previous section, it is possible to draw meaningful conclusions for risk analysis. Models built on robust semantic foundations can offer analytical capabilities for verification (evaluating system compliance to requirements) and validation (ensuring that the system meets initial needs). Checking models with a verification tool consists in establishing a verdict on the soundness of statements (properties) with respect to the verifier's own set of theories. We traditionally identify the following two families of properties:

- Safety properties, intuitively expressing that "bad things do not happen", with
  - invariants, expressing conditions that always hold true,
  - deadlock freedom properties, ensuring that the system never enters a state where progress[8] is not possible,
  - reachability properties, investigating whether it is possible to reach a particular state from the initial state;

[7] Algorithm used for finding shortest paths in a directed weighted graph in cubic time.
[8] Moving from one state to another

- Liveness properties, intuitively expressing that "good things happen", with
  - eventualities, referring to properties that eventually become true,
  - fairness properties, ensuring that resources are fairly shared,
  - termination properties, ensuring that a process eventually terminates.

As with the modeling criteria described in section V.2, model checkers are often able to check models with a combination of these property kinds, with varying degrees of expressivity. In this section, we first focus on properties and how to check them, then we discuss some of the limitations inherent to model checking, before returning to our case study.

V.3.1 Properties and checkers

The use of formal methods allows precise verification of system properties and behavior. For instance, formal methods can be used to verify the correctness of algorithms and assist in verifying safety and security properties of critical components. The main idea is analogous to risk analysis: we prescribe good things and proscribe bad things.

Unfortunately, technical methods do not offer the same range of verification as their formal counterparts [Qadir15]. Models can be studied, executed, verified, validated and, to a certain extent, proved (to verify existentials and refute universals). However, proofs on technical models requiring the exploration of a full state space are often doomed to failure, since the spaces in which these models live are often not formalized.

In formal ecosystems such as Alloy, this state space is properly defined, and tools such as the Alloy Analyzer allow to perform automated checks to verify that a model satisfies its specified constraints and properties. The analyzer uses SAT solvers to explore the model's state space exhaustively on a finite number of objects (to ensure decidability). It makes it possible to find errors in specifications and to generate instances that satisfy given conditions.

In the case of automata models (and more generally, state-transition models), it is possible to animate them (execute them step-by-step); an execution of our lock example (figure V.8, on page 62) with the UPPAAL simulator is shown on figure V.18. In this execution, we validate the intended behavior of our lock, that is, two processes cannot perform actions in the critical section at the same time, although the validation is performed on a small subset of the state space. To check properties exhaustively, the UPPAAL verifier allows to express properties in Computation Tree Logic (CTL) and check safety criteria (absence of deadlocks, progress, *etc.*). In our lock example, we can prove that only one process enters the critical region with the property p : `A[] critical` $\leq$ `1` (in UPPAAL's CTL flavor).

Expressing these properties is not always straightforward: the logical formulation of intuitive concepts such as "S precedes P between Q and R" can be daunting to write. In regular CTL, it would be `AG(Q & !R -> A[(!P | AG(!R)) W (S | R)])` with `W` the "weak until"[9] operator. In UPPAAL, the process is much more complex, as mentioned in section V.4.

In principle, this formal verification is very appealing, but as the systems studied grow, the size of the state space explodes [Clarke12], possibly hindering verification within a reasonable timeframe.

[9] `A W B` means that `A` has to hold at least until B; if B never becomes true, A must remain true forever

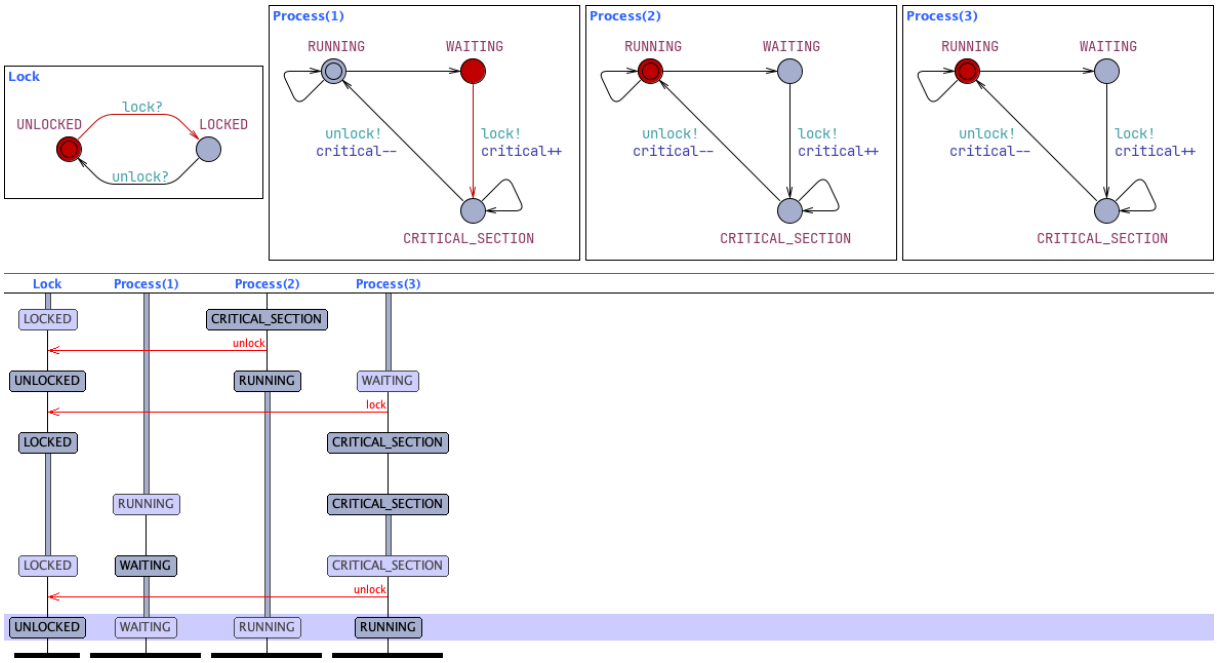


Figure V.18: Execution of our lock model in the UPPAAL simulator

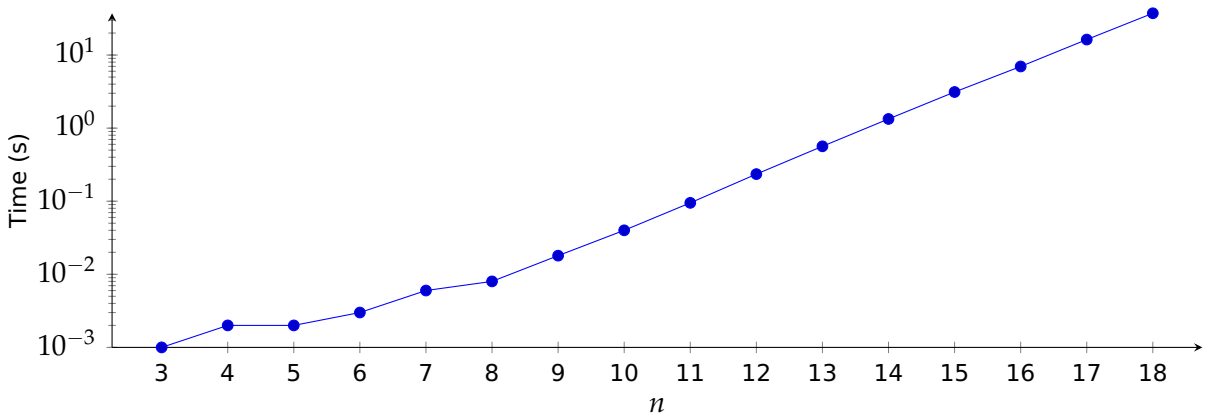


Figure V.19: Verification time for property p in our lock example as a function of the number n of composed processes. The graph represents a single run of experiments, and therefore does not feature error bars.

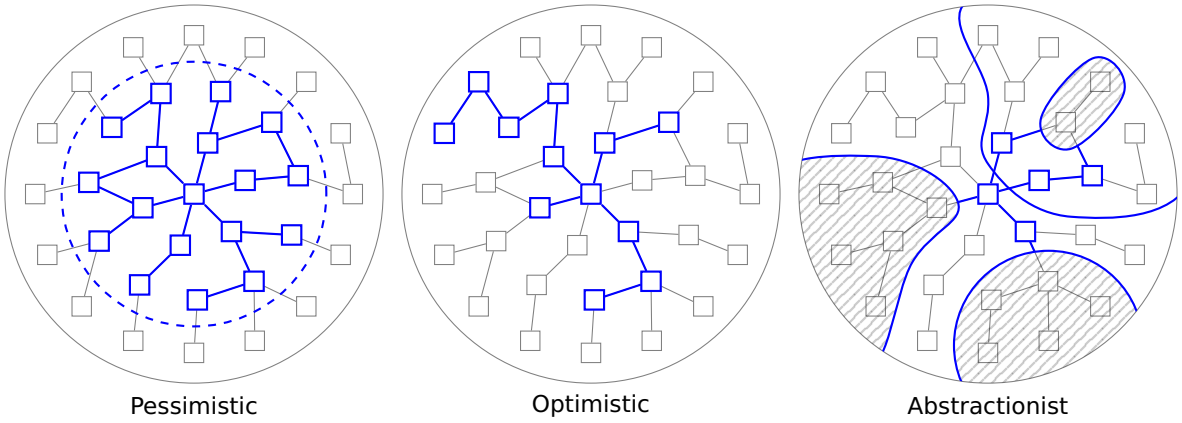


Figure V.20: State space exploration strategies

V.3.2 The need for proper abstractions

Model checking calls for a keen focus on proper abstraction and granularity, as mentioned in section IV.4.1. As models become more complex, their state spaces grow exponentially, and verification can lead to combinatorial explosions. The verification of the property p on our lock model with $n = 3$ processes takes at most a few milliseconds on a modern machine. However, increasing the number of processes leads to an exponential increase of verification time, as depicted in figure V.19.

It is not uncommon for model checkers to fail because of spatial or temporal constraints [Metzler19]. That does not mean, however, that no conclusion can be drawn from a failed verification [Pavese16]: the nature of the state space and the techniques used to explore it play an important role. From a purely probabilistic point of view, the more undesirable states there are, the greater the chance of encountering them; "the longer our checker takes to find errors, the greater the chance that there are none". The literature shows three responses to this, represented in figure V.20, which we provocatively call "pessimistic", "optimistic" and "abstractionist":

- The pessimistic approach is to say "if we cannot reduce the state space, let us reduce the scope of our analysis". This is what is done in bounded model checking engines such as Alloy, and often consists in adopting an iterative deepening depth-first search of state spaces;
- The optimistic approach is to say "if we cannot reduce the state space, let us hope to find a clever exploration strategy". This is what is done in modular and probabilistic model checkers such as Storm, presented in [Dehnert17];
- The abstractionist approach is to say "if we cannot reduce the state space, let us simplify the system and refine only the parts that do not satisfy our properties". This is what is done in counterexample-guided abstraction refinement [Clarke00].

Although these methods can help navigate state spaces, it is sometimes more pragmatic to opt for multiple smaller models as opposed to a monolithic global model. First, the composition of models can benefit from the advantages of several model checkers, potentially widening the range of property kinds available. Second, checking small models reduces the overall number of states, making them easier to check. This strategy necessitates however a careful consideration to ensure the holistic dynamics of the system are not compromised, thus insisting on the semantics of system composition. Let us look at those aspects in our case study.

V.3.3 Going back to our case study

To ensure the safety of our cluster, we want to prove the following properties:

P_1: The cluster can be quorate and at any given time, there is at most one quorum, to avoid "split-brain" conditions;
P_2: There is no single point of failure in the infrastructure;
P_3: (Ideally,) There is no dual point of failure in the infrastructure.

We revise our diagram from figure V.17 (page 68) in figure V.21 to add these properties. More generally, we can ask ourselves the following question: "what is the smallest number of failures such that our cluster fails?".

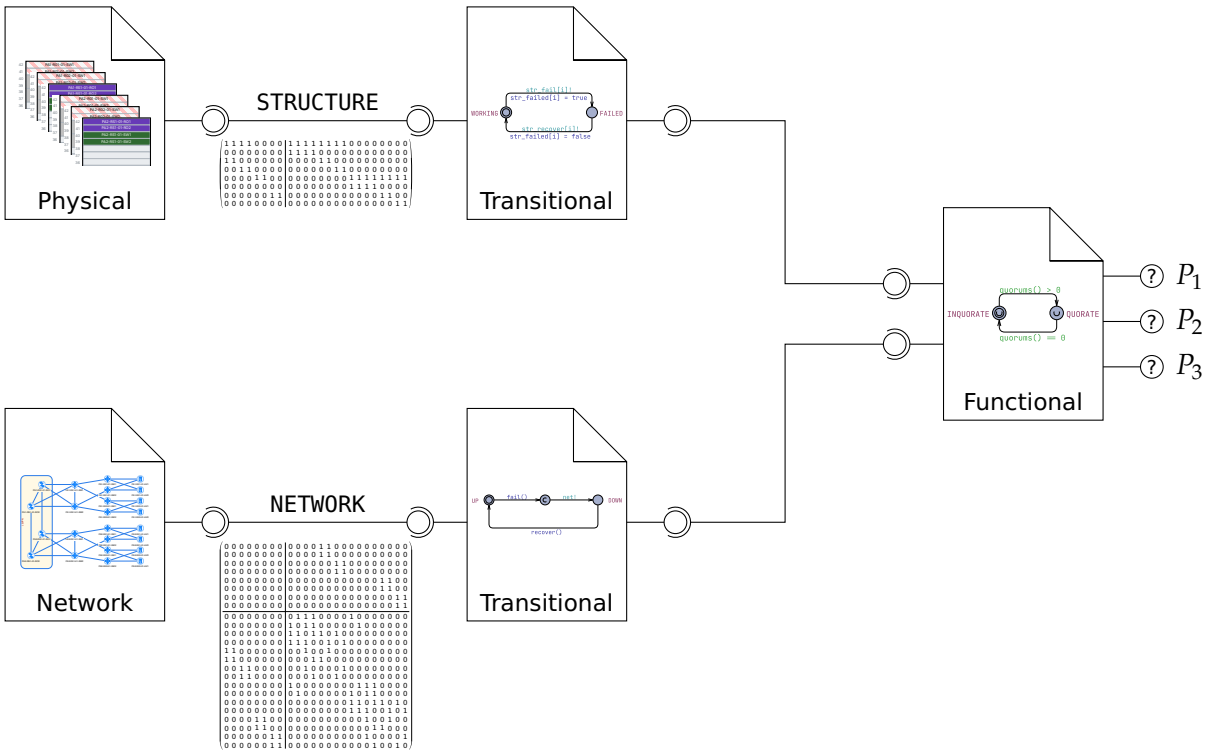


Figure V.21: Composed model with its properties

First and foremost, we can navigate UPPAAL's simulator to explore our system and devise verification scenarios:

S_1. Consider a 5-votes quorum, make every node communicate with one another, then cut the electricity in `PA1-RS1`;

S_2. Consider a 4-votes quorum[10], make every node communicate with one another, then cut the network links between `PA1` and `PA2`;

S_3. Consider a 5-votes quorum, make every node communicate with one another, then cut the electricity in `PA1-R01`, then `PA1-R02`.

Upon execution, we get the traces shown in the appendices in figures B.1 to B.3 (pages 156 to 158). The first scenario shows a violation of P_2, the second one shows a violation of P_1, and the third one shows a violation of P_3.

Formally speaking, we have proved by counterexample that the properties do not hold. However, the process is lengthy and does not adapt to, say, a reconfiguration of the cluster. Reinterpreting P_1 as a combination of CTL statements is rather easy with our model: $P_1 : (\texttt{E<>\ Cluster.QUORATE}) \wedge (\texttt{A[]\ Cluster.quorums()} \leq 1)$. We have however no way to express P_2 and P_3; our model needs to change. Additionally, the use of algorithms such as those of Tarjan or Floyd-Warshall both increase the state space and slow down the computation of each state by UPPAAL's solver (due to its $\Theta\left(n^3\right)$ complexity)[11].

[10] Tarjan-enabled version of the model only

[11] Although even without both algorithms, P_1 does not check within one hour on a high-end computer

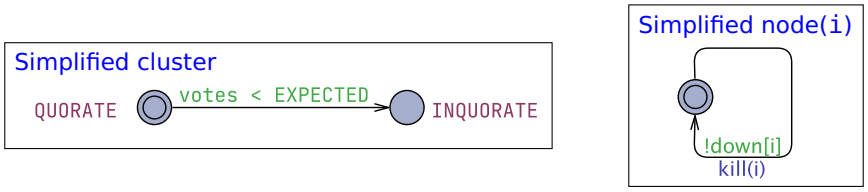


Figure V.22: Simplified UPPAAL model for our system

Because our properties only deal with quorums and failures, modeling more behaviors only leads to combinatorial explosion. More precisely, a large part of our state space is useless for proving our properties:

- The fact that the different nodes can come back up has no influence on our properties and therefore generate superfluous transitions and states;
- Obtaining a quorum requires at least $v \times n$ transitions from the initial state, and n^2 transitions for an optimal situation in which every node is seen by one another, where v is the required number of votes and n the number of nodes. P_2 and P_3 only make sense when analyzing the *loss* of a quorum, so it is safe to assume an optimal initial situation, simplifying the state exploration[12];
- Because we assume an optimal initial state, P_1 can be overapproximated with $P_1' : v \geq \lceil \frac{n}{2} \rceil$, making the property trivially provable by an arithmetic checker.

We have therefore adopted a much simpler approach: a cluster is quorate by default, and becomes inquorate as soon as the amount of votes from nodes is lower than v. Additionally, nodes (whether regular, structural or network) have now a single action: become inoperative (`kill(i)`) if they are not already down (`!down[i]`). Everytime a node becomes down, a counter, `actions`, is increased; when the node is a regular node, an additional counter, `node_actions`, is increased. The new model is shown on figure V.22. The code logic is similar to that described in the previous section. We have encoded two families of properties in this model:

- $p(j)$: `E<> actions` $\leq j$ `&& Cluster.INQUORATE`, helping us identify how many node failures (`actions`) lead to a loss of quorum (`Cluster.INQUORATE`);
- $q(j)$: `E<> actions` $\leq j$ `&& actions == node_actions && Cluster.INQUORATE`, given here purely as an example, helping us identify how many regular nodes have to go down (`node_actions` to lose the quorum (`Cluster.INQUORATE`).

Also, P_1' is verified by design, $P_2 = p(1)$ and $P_3 = p(2)$. Checking the model with UPPAAL gives us:

$$p(j) \mapsto \begin{cases} \top & \text{if } j \geq 1 \\ ? & \text{otherwise} \end{cases} \qquad q(j) \mapsto \begin{cases} \top & \text{if } j \geq 4 \\ ? & \text{otherwise} \end{cases}$$

This is enough to (dis)prove our properties, but exhaustive exploration fails once again. For this study, the situation is not particularly important, as the "?" verdicts are beyond scope, but other properties may require that we split the $\{\text{Corosync} + \text{structure} + \text{network}\}$ model into $\{\text{Corosync} + \text{structure}\}$ and $\{\text{Corosync} + \text{network}\}$ models.

[12] In our example, this reduces the depth of state exploration by $n^2 = 64$, which is considerable.

We can draw three important conclusions from the practitioner's point of view from this case study:

- If a mathematical projection is identified, technical models can easily be brought into the formal world; luckily, many elements of IT infrastructures can be modeled as graphs, or equivalently as matrices;
- Models and properties are tightly coupled: certain model design choices may make it impossible to verify certain properties; both should be chosen at the same time;
- Holistic models are doomed to fail in automatic verification: they may be a great choice for manual execution to check traces or demonstrate how a system works, but often generate too many states, which can be very costly to reduce.

With that out of the way, we can now discuss the question of automating the process, and how to avoid repeating the same mistakes.

V.4 Automating risk assessment

First of all, fully automating IT infrastructure checking does not seem realistic. As we have seen throughout this chapter, formal infrastructure models can be generated automatically from technical models, but in the end, these truth sources are created and maintained by human beings. In addition, formal models that are not produced from technical models are designed, optimized, and sometimes abstracted and generalized by domain experts. In this section, we first look at how to define formal properties and then discuss how an automated assistant can help with risk assessment.

V.4.1 Expressing formal properties...

The various risk taxonomies that we presented in section IV.2.1 offer a high-level textual representation of what characterizes risk. They can break down risks into criteria that we can translate into formal properties. This translation can be done manually or using natural language processing tools. Let us consider for example the section *AC-3 – Access Enforcement* of NIST SP 800-53. Control enhancement 5 (*Security-relevant information*) reads "Prevent access to [*Assignment: organization-defined security-relevant information*] except during secure, non-operable system states.". The element between brackets is defined as customizable by the standard; here we can consider it to mean "secure resources".

We can transform this sentence into a parse tree, as shown on figure V.23, to help us formulate a property. The words "during" and "states" suggest that we can use a temporal logic to express this property. Noun phrases (NP) can be expressed as states; let us define S_1:"access to critical resources" and S_2:"secure, non-operable system state". Then, key words such as "prevent <X>" and "except during <Y>" can respectively be transformed as $\neg$X and $\vee$Y, giving us the property $p : \neg S_1 \vee S_2$, which we can temporalize in LTL as $p' : \square\neg S_1 \vee S_2$ (where $\square$ means "always").

Sometimes, the keywords are not easy to express in, say, LTL or CTL. For example, formulae for "before", "after" and "until" are dependent on the context and can be complex to combine. To ease the specification of such properties, specification patterns for qualitative [Dwyer99], real-time [Konrad05] and probabilistic [Grunske08] properties have been proposed and eventually aligned [Autili15] in the literature. These patterns can be used as a basis for expressing complex properties such a "if we push the power button on a powered-off server and it does not power on, then the server malfunctions".

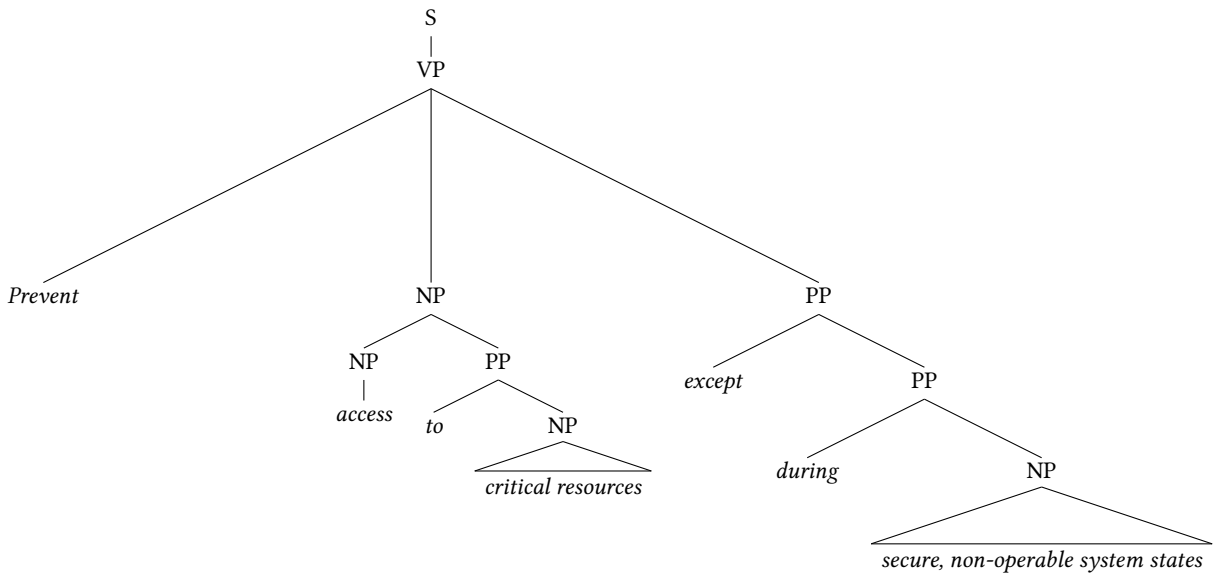


Figure V.23: Parse tree for "Prevent access to critical resources except during secure, non-operable system states"

Tools such as UPPAAL, however, only support a reduced subset of CTL, and expressing such properties requires making changes to the model, underlining once again this strong coupling between models and their properties.

V.4.2 ... and combining models together

Once we have our formal models and have expressed the risk in formal properties, we need to verify them. Risk relates to specific infrastructure subsystems: the seismic risk does not directly affect software components, just as the risk of an attack on a database does not directly affect a physical rack in a datacenter. We must therefore:

1. Identify the sections of the infrastructure concerned by the risk;
2. Identify a set of models for these portions of the infrastructure that can be used to prove or disprove safety and security properties;
3. Prove the properties or return to the operator the properties and models that are missing to establish a risk verdict.

The first step involves assigning a set of labels to each risk, in order to characterize it. These labels may have a hierarchical structure, indicating, for example, that one risk applies to all database engines, while another applies specifically to SQL engines. Unique identifiers such as those from the Common Platform Enumeration ([CPE]) help identify specific software and hardware, but lack this hierarchy of concepts. For example, Proxmox VE 6.3 is identified as `cpe:/a:proxmox:virtual_environment:6.3`, but there is no way to refer to a generic hypervisor. These labels should then be assigned in the actual infrastructure to help identify systems concerned by a given risk.

The second step suggests that companies maintain a repository of models, associated with the actual components. As we have seen, a single complete model is not always a good idea, so these models should be reasonably sized and easily composable. A verification assistant can then decompose the risk into its sub-risks expressed as logical properties, and choose the appropriate models to verify them.

Finally, the various model checkers queried by the assistant lead to a verdict that proves or disproves the properties. If a verdict of "?" is reached, the proof assistant should give the control back to the human operator to give them pointers for further model development.

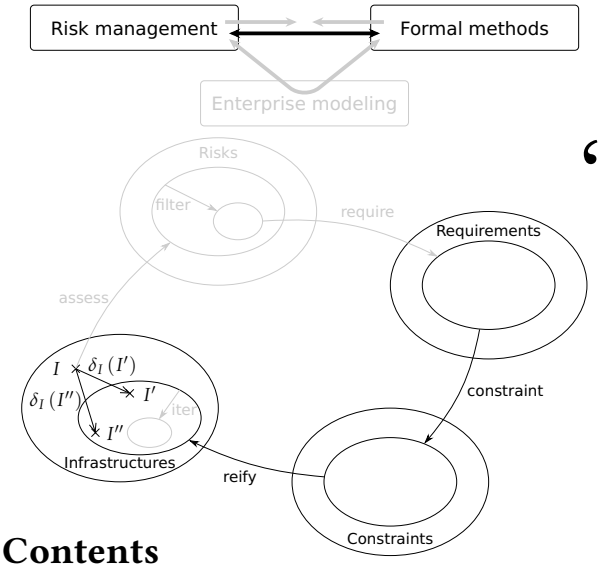
V.5 Conclusion

Model checking helps to guarantee safety and security properties, increasing overall confidence in systems and simplifying auditing procedures. We have built this chapter around a case study illustrating various issues that can arise in the modeling process and in the subsequent composition and verification of the models produced. We have proposed an approach to alleviate the problems associated with human judgement in the overly-manual discipline that is risk analysis. Future work on the subject may involve looking at the interactions between diverse model checkers and the composition of their verdicts. But do not be decieved: we do not make the process magically automatic (we actually think that it is not possible); reducing the subjectivity of the practice is however a step towards building infrastructures that are more safe and secure.

Having explored the `assess` part of the risk cycle, there are still functions that we have not yet concretized, and this is the aim of the next chapter.

Chapter VI

Deploying and Maintaining IT Infrastructures



> Karl Agathon — Snowbirds, Galactica. Regroup into deployment formation and proceed to position one.
>
> Margaret Edmondson — Snowbird One to Snowbirds, drop point in eight seconds.
>
> Louanne Katraine — Okay, Snowbirds, let's get this deployment bang on.
>
> *Battlestar Galactica – Occupation*

Contents

VI.1 Requirements–configuration–execution triad . . . 80
VI.1.1 Inconsistencies . . . 81
VI.1.2 Change . . . 82
VI.1.3 Formalization . . . 83
VI.1.4 Approach . . . 84
VI.2 The CL/I language . . . 84
VI.2.1 Another language? . . . 84
VI.2.2 Modeling in CL/I . . . 85
VI.2.3 Syntactic processing . . . 86
VI.2.4 Semantic processing . . . 88
VI.2.5 Extensions . . . 90
VI.3 Mapping into Z3 . . . 91
VI.3.1 Translation rules . . . 92
VI.3.2 Conformance checking . . . 95
VI.4 Case studies . . . 95
VI.4.1 Virtual environment model . . . 95
VI.4.2 Proxmox VE configuration and execution . . . 97
VI.4.3 Model checking . . . 98
VI.4.4 Scaling . . . 99
VI.4.5 A more complete case study . . . 100
VI.5 Conclusion . . . 101

If we look back at the risk cycle we presented earlier, the risk assessment process allows to define a set of requirements, which in turn lead to the creation of constraints, materializing into an infrastructure. Let us take a step back: this process is not specific to risk management and can clearly be applied to the design of infrastructures more generally. This is what we explore in this chapter.

IT infrastructures are inherently dynamic, constantly evolving to accomodate the changing needs of modern businesses. As organizations strive to stay at the forefront, the complexity of these infrastructures grow [Wehling17], and the trend is further accelerated by the rise of cloud technologies. Such a complexity introduces challenges in ensuring their proper functioning and adherence to specified requirements [Ozkaya23]. From the initial definition of requirements to the configuration, deployment and execution of applications, non-conformities can arise due to many factors and their presence in critical infrastructures can be catastrophic [Graham19].

In addition to safety and security risks, these infrastructures are subject to regulations and must comply with audits for companies to be allowed to continue their activities. For example, in Europe, banking infrastructures must be compliant with the Revised Payment Services Directive ([PSD2]). Addressing these problems while ensuring the desired safety and security properties on IT infrastructures requires comprehensive modeling and verification techniques which can be costly to implement. In this chapter, we choose to focus on the *requirements–configuration–execution* triad, by studying non-conformities between, and within, each of these three aspects of the infrastructure lifecycle. More specifically, we study how *requirements* are translated into infrastructure *configuration*, how this *configuration* is reflected in the *execution* of the infrastructure, and how this *execution* may or may not conform to the initial *requirements*.

In line with *infrastructure as code* approaches [Hüttermann12; Artac17], we present in this chapter a component-oriented infrastructure description language, CL/I, proposed to express these three aspects in a common framework. First, we adapt our formalism in section VI.1 to address the infrastructure design process that we cover in this chapter. Then, we introduce through examples in section VI.2 CL/I, an infrastructure description language that we have developed in the course of our work. Next, we show how to link our language to the Z3 theorem prover to perform model checking in section VI.3. We then present two case studies validating our approach in section VI.4. Finally, we conclude this chapter in section VI.5.

VI.1 Requirements–configuration–execution triad

To outline the integration of a project into an IT infrastructure, we can ask ourselves three questions:

1. What is *wanted* for our project?
2. What is *asked* to the infrastructure?
3. What is *done* by the infrastructure?

At the origin of a new project, a need is expressed and translated into requirements, what we actually *want*. To fulfill this need, we *ask* infrastructures and their components to operate as required through a special configuration[1]. Lastly, the infrastructure exhibits a particular behavior during its operation, the work that the elements constituting it actually *do*.

[1]We mean "configuration" in a broad sense: software configuration, systems architecture, and more generally anything that controls structure and behavior.

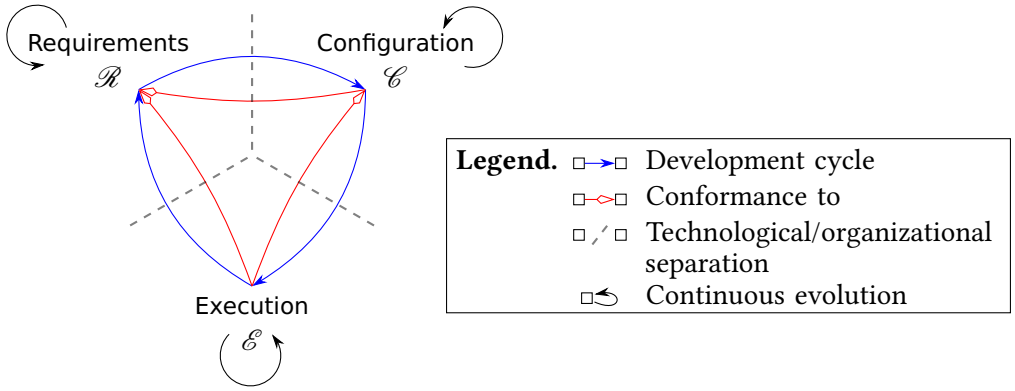


Figure VI.1: *Requirements–configuration–execution* triad: technical point of view

If we take the example of a banking application, it could correspond to the following:

1. We *want* a performant banking infrastructure, that i) is fully PCI DSS compliant and ii) respects its service-level agreements (SLA) with its clients;
2. We *ask* for i) proper access rights to cardholder data, through a careful definition of roles; ii) proper SLAs, by setting up redundancy and using efficient systems;
3. The infrastructure enables to *do* high performance electronic transactions and customer support.

In this section, we first have a look at the different inconsistencies that can arise between the *wanted*, the *asked* and the *done*. Then, we provide further details on our notion of change, mentioned in section IV.1.4. We finally formalize the scope of this chapter, before describing our approach.

VI.1.1 Inconsistencies

Sometimes in infrastructures, the *asked* does not match the *wanted* if the requirements are not (or no longer) properly translated into an application's configuration [Bleikertz15]. For example, if we define access rights that do not match PCI DSS criteria or if such requirements evolve over time without being updated in the configuration. Sometimes, the *done* does not (or no longer) correspond to the *asked* if bugs arise or conflicting orders are given during runtime. For example, if memory leaks lead to unstable behavior or if we push new configuration files that are not taken into account by the application. Finally, the *done* sometimes does not (or no longer) answer the *wanted* if the application has to (temporarily) break invariants to face exceptional situations. For example, if banking transactions are automatically accepted in case of an outage, or if the service cannot cope with an excessive influx of requests (legitimate or not). This requirements–configuration–execution triad is shown on figure VI.1, where we represent each element as a continuously evolving process along the development cycle. We also show how the conformance is evaluated: configuration shall conform to requirements, and execution shall in turn conform to both. The boundaries between the *asked*, the *wanted* and the *done* make it difficult to ensure consistency between the three, notably due to semantic gaps.

Although they are often causally linked (if an application is misconfigured, it probably does not work the way we intend it to), the three classes of inconsistencies described previously may arise for different reasons. For example, let us consider a virtualized infrastructure with k virtual machines $(v_1, ..., v_k)$ and l physical hosts $(n_1, ..., n_l)$ with the safety requirement R: "v_1 and v_2 run together on the same physical host". The virtualization environment may be misconfigured if a human operator fails to implement the

safety requirement in the configuration language, but this does not necessarily lead to the requirement being violated. Indeed, if v_1 and v_2 are run on the same host n_i and are never relocated on other hosts, the requirement holds. If a user forces one of the virtual machines to run on another host, the execution would behave outside of the domain set by the configuration, breaking the requirement. Finally, if there is a failure on n_i, v_1 may be moved onto another node and be running, while v_2 has not yet been relocated, also breaking the requirement. Further examples of properties on virtualized infrastructures are given in section VI.4.

VI.1.2 Change

In addition to these considerations, infrastructures and needs evolve, to respond to urgent events (such as emergency patches [Liu09]), when regulations change, or as companies and products mature (vertical and horizontal scaling). This new dimension implies the need to maintain conformance over time, leading to infrastructure and code erosion issues [deSilva12]. Such changes can occur frequently in cloud infrastructures, making automation an interesting option [Weinreich14].

As we mentioned in section IV.1.4, these changes have a cost, which can vary from one company to another. For example, we may want to minimise the financial impact of purchasing new hardware or the time impact of configuring new applications. To help with change management, we can represent the differences between two infrastructures (the δ in our formalism in chapter IV) as a directed graph of changes, in which each edge represents a change with a measurable cost for the company, as shown in figure VI.2. In this figure, we represent *checkpoints* $(i_1, i_2, \ldots, i_{n+1})$, between which changes are represented as colored edges. Each edge color represents a different "kind" of change (defined on the left graph), to which we can assign weights to represent the priorities of a company (with regard to finance, time, *etc.*).

For example, suppose that between checkpoints i_{n-1} and i_n, we have developed two components of an application and deployed them on an existing physical server (steps represented between the dashed lines on the center and right graphs). We have measured for i_n that the application is slow at peak times, so we need to update our infrastructure. i_{n+1} (center graph) represents a horizontal scaling, that is, we deploy part of the application on another physical server (in addition to the existing server) to distribute the load. The cost of the change is the combined cost of purchasing a new server, setting it up and deploying the application on it. Conversely, i'_{n+1} (right graph) represents a vertical scaling, that is, we redeploy the whole application on a server with better resources (replacing the existing server). Here, the cost is also the combined cost of purchasing a server, setting it up and deploying the application, along with the cost of canceling the changes from i_{n-1} to i_n. Depending on the importance of each of these costs for the company, solution i_{n+1} or i'_{n+1} may be adopted.

We argue that adopting a systematic method decomposing changes as small steps that we can categorize can simplify the decision-making process, by allowing to estimate costs more precisely. The cost of undoing changes should however not be underestimated: for example, decommissioning a physical server involves a cost of storage or recycling, and even a cost of erasing or destroying disks for sensitive activities.

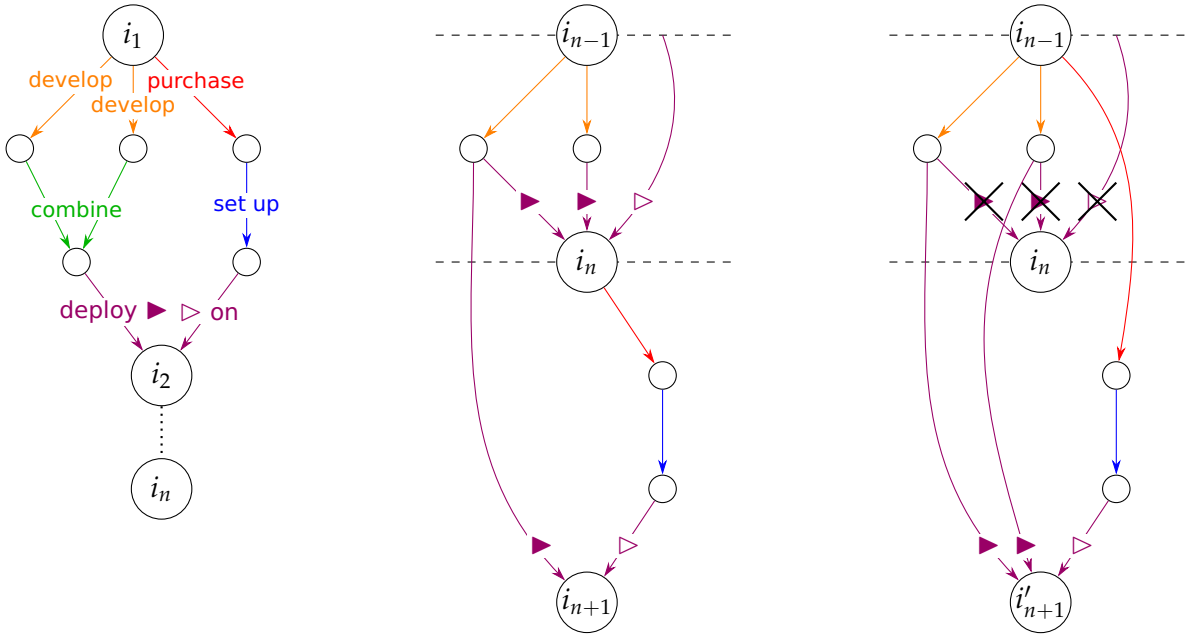


Figure VI.2: Deployment graphs

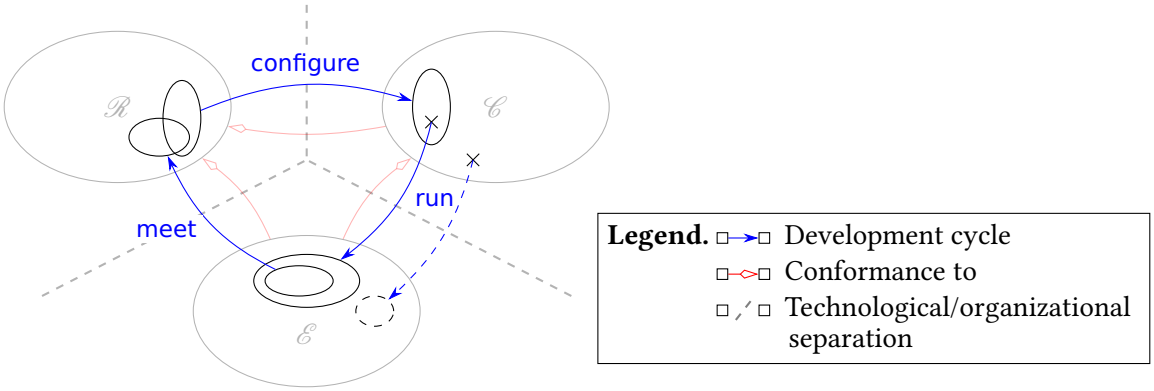


Figure VI.3: *Requirements–configuration–execution* triad: mathematical point of view

VI.1.3 Formalization

For the sake of completeness with respect to our formalism, we define, for a given application with a given configuration language, $\mathscr{R}$ the set of all requirements, $\mathscr{C}$ the set of all possible configurations and $\mathscr{E}$ the set of all possible executions. The possible implementations of a set of requirements $r \in \mathbb{P}(\mathscr{R})$ is a function configure from $\mathbb{P}(\mathscr{R})$ to $\mathbb{P}(\mathscr{C})$ that translates r into a set of possible configurations. The configuration of the application runtime is a function run from $\mathscr{C}$ to $\mathbb{P}(\mathscr{E})$ that maps a chosen configuration (not necessarily meeting the requirements) into the possible executions that it leads to. Finally, the satisfied requirements by a set of executions is a function meet from $\mathbb{P}(\mathscr{E})$ to $\mathbb{P}(\mathscr{R})$ that "observes" a sample of the application executions and tells which requirements it satisfies (not necessarily the initial ones)[2]. We can refine the triad of figure VI.1 as shown in figure VI.3, in which we represent a cycle where observed executions of an application (run with a configuration translated from some requirements) meet requirements that are not aligned to the initial ones.

[2]Formal methods can be used to model executions and prove the conformance to requirements, but the actual execution on physical components can exhibit behaviors outside the ideal domain of an execution model.

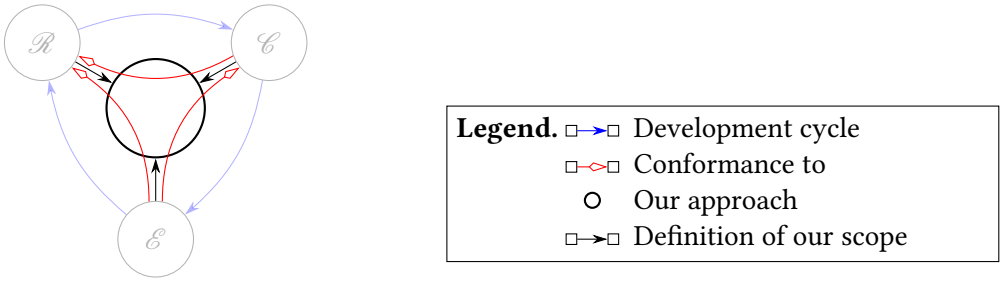


Figure VI.4: *Requirements–configuration–execution* triad: our approach

VI.1.4 Approach

From a technical point of view, it is difficult to generate a set of valid configurations (evaluating the function configure), to fully characterize the behavior of an application (evaluating the function run) and to determine which requirements a set of executions meets (evaluating the function meet). Instead, our approach attempts to verify that a configuration satisfies the requirements, that the execution of an application falls within what is permitted by its configuration, and that this execution also conforms to the requirements. It is also essential to check the coherence of requirements and configurations to ensure that the system is correctly defined. This study scope is presented in figure VI.4, where the requirements, configurations and executions we study define a framework in which we check for conformance.

VI.2 The CL/I language

Complex technical infrastructures rely on thousands of components that interact with one another [Sommerville12], each of them having their own requirements, configuration and execution. To carry out our approach on those, we have developed a component-oriented infrastructure modeling language, CL/I, as a bridge between infrastructure modeling and formal methods. Much like developers reusing a catalog of existing libraries, the main idea behind CL/I is to enable infrastructure designers to instantiate models designed by system designers, augmented by models designed by technical experts and people from formal methods communities, through a pivot language.

In this section, we defend the need for a new language and present the use of CL/I by example, then outline various semantic processing rules that take place when expressing configurations and requirements. It should be noted that at the current state of the project, the observation of executions is delegated to external tools.

VI.2.1 Another language?

We are conducting our research in a corporate context. Within our company, many of the employees work in technical positions and have a certain IT culture. For developers in particular, the work environment and standard workflows revolve around the collaborative and dynamic culture of code production. Despite this strong background, many technical staff have no particular modeling skills. Yet these employees are the ones we aim to integrate into the risk management process.

As a result, the use of modeling languages such as UML is not common practice. When we focus on infrastructures, it can be interesting to study their evolution and the impact of certain changes on risk. The software development community is well equipped for such objectives, and versioning

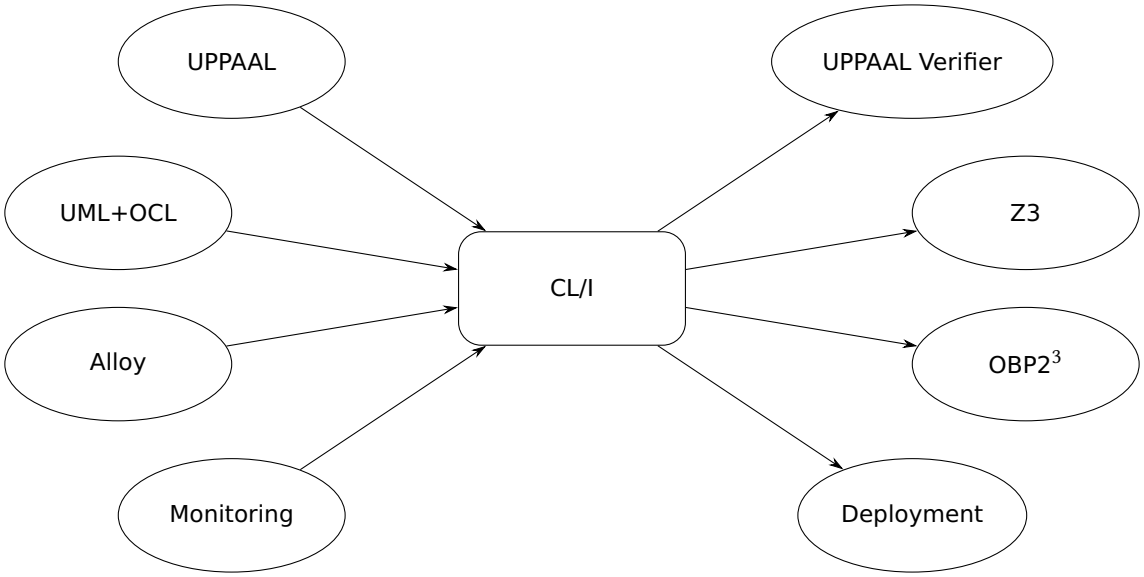


Figure VI.5: Our language as a pivot between formal and informal tools

tools such as Git exist for this purpose. These tools often include features enabling precise tracking of who is responsible for which changes, thus providing a precise control over the responsibilities of each stakeholder. We discuss this responsibility aspect in greater detail in chapter VII. Given the lack of a versioning system for graphical models, we decided in this work to use code, closer to the actual employee's habits.

Our language is not intended to replace existing modeling and verification tools, but rather to federate them. The motivation behind the creation of our language is, as presented in the previous chapter, to be able to compose and verify systems made of other systems from various origins. We want to handle cases with several input models, to check infrastructure states in real-time, to verify properties on models with external provers, to deploy verified infrastructure components on runtime platforms... Our language then acts as a pivot in this ecosystem, represented in figure VI.5.

Within the company, many different professions coexist, each with its own tools, skills and responsibilities. A pivot language allows to focus on the interactions between the components rather than the methods used to design and represent them, and allows employees to keep their existing habits and tools. Our aim in developing CL/I is to be able to federate diverse and heterogeneous models and, in the long term, to verify properties on formal models, integrate these properties with higher-level or less formal models, and draw conclusions about the overall risk on infrastructures. For example, we can imagine a verified B component that we instantiate in our language, integrated to other components verified by other tools, then deployed on a concrete infrastructure following a study of the safety and security of the assembled system.

VI.2.2 Modeling in CL/I

CL/I is a component-oriented, statically typed modeling language designed to have a syntax close to common object-oriented programming languages such as Java. It is component-oriented, as every *thing* worth of modeling is represented as a component in the same way as it would be represented as a class

[3]OBP [Teodorov23] is a requirement verification environment developed in our research team

```
component User {
  let id : Integer;
  let homedir : String;
  let name : String;
}
```

```
component Permission {
  let u : Integer;
  let g : Integer;
  let o : Integer;
}
```

```
component File {
  let owner : User;
  let path : String;
  let perm : Permission;
}
```

Figure VI.6: CL/I components for a user, a permission triple and a file

```
let root = User { id ← 0; homedir ← "/root"; name ← "root"; };

let 600_p = Permission { u ← 6; g ← 0; o ← 0; };
let 644_p = Permission { u ← 6; g ← 4; o ← 4; };
let 700_p = Permission { u ← 7; g ← 0; o ← 0; };

let file_1 = File { owner ← root; };
let file_2 = File { path ← "/root/test"; perm ← 755_p; };
let file_3 = File { path ← "/root/.ssh/test"; perm ← 755_p; };
```

Figure VI.7: CL/I instances of the components in figure VI.6. Note that not all instance attributes need to be defined.

in Java. Types and components start with an uppercase letter and values and instances start with a lowercase letter or a digit. As an example, we model a file on a Linux system with a path, an owner and a permission triple in figure VI.6. Here, three concepts are represented as components with attributes:

- A user, with a unique identifier, a name and a home directory;
- A permission triple, which stores the authorization for the owner of a file (`u`), the users in the file's group (`g`) and other users (`o`);
- A file, with an owner, a path and a permission triple.

In CL/I, components can be instantiated by calling them with parameters between parentheses (or a single pair of parentheses if no parameters are required) and their attributes can be assigned between curly braces:

```
let instance = Component(param_1,param_2);
let instance_2 = Component();
let instance_3 = instance { attr_1 ← val_1; attr_2 ← val_2; };
```

For convenience, `Component{...}` $\stackrel{\text{def}}{=}$ `Component(){...}`. CL/I is designed to support partial specifications; unlike with traditional languages, components can be instantiated with some values remaining undefined. Figure VI.7 shows several incomplete instances of the components defined in figure VI.6. For the curious reader, the complete grammar of CL/I is presented in appendix C.1.

VI.2.3 Syntactic processing

`clc` is the reference CL/I compiler, developed in OCaml. Together with lexing (with `ocamllex`) and parsing (with Menhir), the first processing step is to transform CL/I source code into an Abstract Syntax Tree (AST). We have designed our ASTs in such a way that a well-typed AST is a valid language construct, thus simplifying and systematizing semantic processing, at the cost of a slightly more complex AST structure. Appendix C.2 presents this structure exhaustively, along with examples of sentences in CL/I

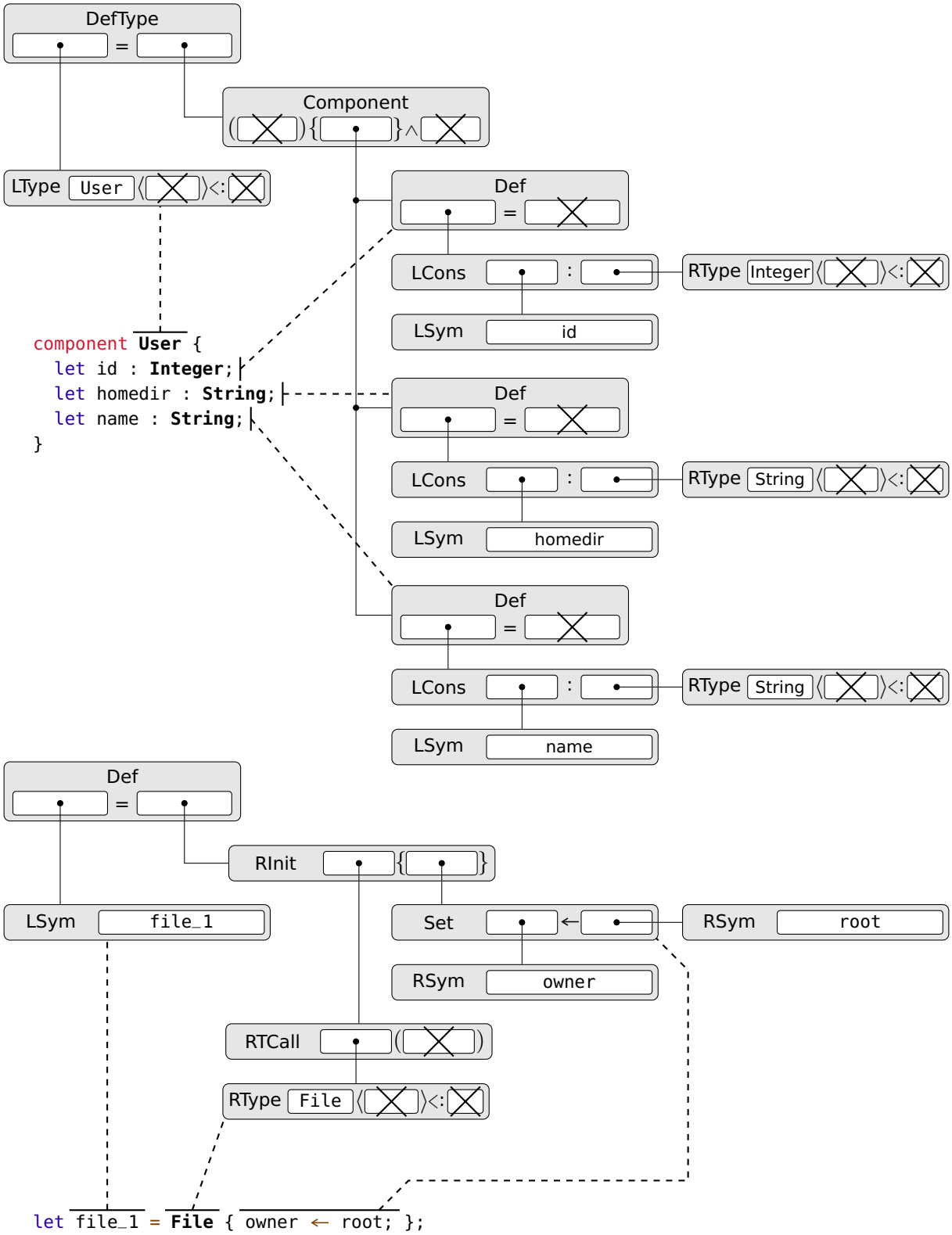


Figure VI.8: Syntactic transformation of the `User` component and the `file_1` instance

building each AST node. For illustrative purposes, we give in figure VI.8 the ASTs we get for the `User` component and the `file_1` instance. The main point, as with many languages, is that we separate terms between those that go before the equal sign (*lhs*, for left hand side[4]) and those that go after (*rhs*, for right hand side[5]).

A model (or a file) in CL/I is a succession of value and type definitions (`let x = y;` for values and `type T = U;` for types), value initializations (`x ← y;`)[6], and expressions (`print(x);`). Finally, component definitions (`component C { ... }`) are represented as type definitions in ASTs.

VI.2.4 Semantic processing

We have designed CL/I with the purpose of interacting with various model checkers, so we decided to adopt an approach based on an intermediate representation, as an exchange format. This intermediate representation, named CLIR, flattens our data structures into integer-indexed symbol tables, which makes it easier to encode our structures into tools that do not support our naming conventions or nested data structures.

The intermediate representation consists of two symbol tables: one for the types defined in our models, and one for their values. In the rest of this chapter `T#`i (respectively `#`i) for some $i \in \mathbb{N}$ refers to the *type symbol* (respectively *value symbol*) numbered i. The set of *type symbols* is denoted $\#_T$, and the set of *value symbols*, $\#_V$. Their union, the set of *symbols*, is denoted # ($\# \stackrel{\text{def}}{=} \#_V \cup \#_T$).

Both CL/I types and values can have *attributes* (that we can access with `value.attribute` and `Type.attribute`). The value of each attribute is encoded as a *value symbol*, and the set of attributes for a type or a value is encoded as a map from strings (the name of the attributes) to *value symbols*. The type of attribute maps is $S \to \#_V$ (where S is the type of strings). In our `File` example of figure VI.6 (page 86), the attribute map could be[7] $[\text{“owner”} \mapsto \#1, \text{“path”} \mapsto \#2, \text{“perm”} \mapsto \#3]$. We present the encoding in more detail later in this subsection, in figure VI.9.

The symbol table for types records both the type of the *type symbol* and its set of attributes. It has the type $\#_T \to T \times (S \to \#_V)$, with T defined by the following grammar, adapted from a subset of our AST[8] (where i is an integer):

$$T \quad ::= \quad \mathsf{Abstract} \mid \mathsf{Unit} \mid \mathsf{Boolean} \mid \mathsf{String} \mid \mathsf{Integer} \mid \mathsf{List}\ \langle \mathsf{T\#}i \rangle$$

`Abstract` denotes abstract types, and is used to encode CL/I components. The next four elements describe the primitive types (`Unit` is used for values that return nothing). The last element represents lists (of a given type `T#`i).

The symbol table for values records, for each *value symbol*, its value, its attributes and its *type symbol*. It has the type $\#_V \to V \times (S \to \#_V) \times \#_T$ with V defined by the following grammar (where s is a string, i is an integer and vectors denote a possibly empty list).

$$\begin{aligned} V \quad &::= \quad ? \mid \mathsf{Instance} \mid \mathsf{Boolean\ true} \mid \mathsf{Boolean\ false} \mid \mathsf{String}\ \text{“}s\text{”} \mid \mathsf{Integer}\ i \mid r \mid \mathsf{List}\ \vec{r} \mid \mathsf{App}(f, \vec{V}) \\ r \quad &::= \quad \mathsf{Ref}\ i \\ f \quad &::= \quad = \mid \mathsf{and} \mid \Rightarrow \mid\ ! \mid \mathsf{has} \mid \mathsf{Ext}\ s \mid \ldots \end{aligned}$$

[4]What OCaml calls patterns (type *pattern* in OCaml's AST)
[5]What OCaml calls expressions (type *expression* in OCaml's AST)
[6]Note here that both `x` and `y` are *rhs*es, as we are not defining a new value, but setting an existing one.
[7]Modulo numbering
[8]Function types have not been properly implemented at the time of writing. Additionally, tuples are erased in the CLIR.

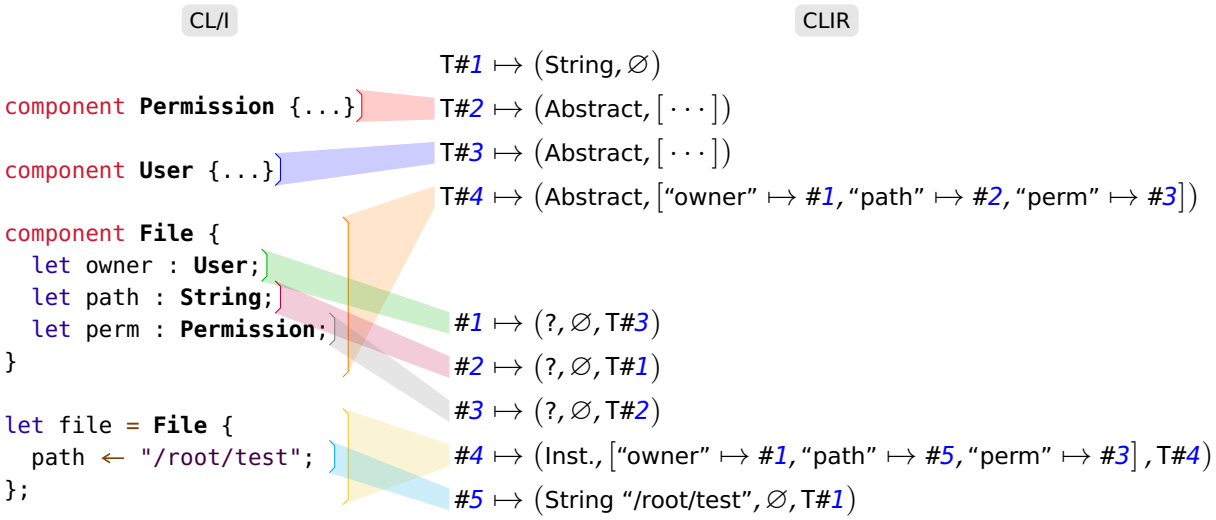


Figure VI.9: Translation from CL/I to the CLIR

CL/I is a modeling language addressing both completely defined models and partial models where some values may be unknown (denoted "?"). The values of component instances are also abstract and denoted Instance. The next four elements describe primitive values. Lists are encoded as lists of value references (in the symbol table), and function applications allow to call predefined functions on values. External operators and functions not defined in the core language are denoted Ext s.

We define the following functions to fetch elements from both tables:

$$v : \#_V \to V \qquad a : \# \to (S \to \#_V) \qquad t : \#_V \to \#_T \qquad \tau : \#_T \to T$$

We illustrate in figure VI.9 a possible translation from part of our examples of figures VI.6 and VI.7 (page 86) to their corresponding CLIR. We invite the curious reader to read the semantic rules for the translation in appendix C.3. We give in figure VI.10 three simplified rules (LET, LETCMP and LETEQ) to help understand the example. We denote $\mathcal{E}$ the association of names to symbols, and $\mathcal{S}$ and the symbol table for types and values. fresh T#i (respectively #i) means that T#i (respectively #i) is a new, globally unique, type (respectively value) symbol. Each of these rules produces ($\Rightarrow$) a pair of new name–symbol associations and the updated symbol table.

LET transforms a typed value declaration in CL/I (`let x : Y;`) into a new symbol #i in the CLIR linked to an unknown value of the corresponding type (?, ∅, T#j), along with an association between the value name and #i. LETCMP transforms a component definition in CL/I (`component Y` $\{t_1; \ldots; t_n\}$) into a new type symbol T#i in the CLIR linked to an abstract type and the component attributes (Abstract, $\mathcal{E}'$), along with an association between the component name and T#i. Finally, LETEQ transforms a component initialization in CL/I (`let x = Y {` $x_1 \leftarrow v_1; \ldots; x_n \leftarrow v_n$ `};`) into a new symbol #i in the CLIR linked to the component attributes with the new associations $x_i \leftarrow v_i$ taken into account, along with an association between the value name and #i.

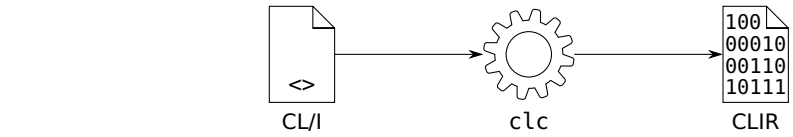
In our previous example, **File** is registered as a type symbol (T#4, rule LETCMP), with its attributes `owner`, `path` and `perm` mapping to empty values #1, #2 and #3 (rule LET). `file` is an instance of **File**, initializing its attribute `path` to a specific value, stored in #5. `file` itself is stored in #4, with the same symbol table as **File**, except for `path` pointing to the new symbol #5 (rule LETEQ).

$$\textsc{Let}\ \frac{\text{fresh } \#i \qquad \mathcal{E}(\mathbf{Y}) = \mathsf{T}\#j}{\vdash \mathcal{E}, \mathcal{S}, \texttt{let x : Y;} \Rightarrow \{\mathsf{x} \mapsto \#i\}, \mathcal{S} \cup \{\#i \mapsto (?, \varnothing, \mathsf{T}\#j)\}}$$

$$\textsc{LetCmp}\ \frac{\text{fresh } \mathsf{T}\#i \qquad \vdash \mathcal{E}, \mathcal{S}, t_1; \ldots; t_n \Rightarrow \mathcal{E}', \mathcal{S}'}{\vdash \mathcal{E}, \mathcal{S}, \texttt{component Y \{ } t_1; \ldots; t_n \texttt{ \}} \Rightarrow \{\mathbf{Y} \mapsto \mathsf{T}\#i\}, \mathcal{S}' \cup \{\mathsf{T}\#i \mapsto (\mathsf{Abstract}, \mathcal{E}')\}}$$

$$\textsc{LetEq}\ \frac{\text{fresh } \#i \qquad \mathcal{E}(\mathbf{Y}) = \mathsf{T}\#j \qquad \vdash v_i \Rightarrow \#k_i}{\begin{array}{l}\vdash \texttt{let x = Y \{ } x_1 \texttt{←} v_1; \ldots; x_n \texttt{←} v_n \texttt{ \}; } \Rightarrow \\ \qquad \{\mathsf{x} \mapsto \#i\}, \mathcal{S} \cup \left\{\#i \mapsto \left(\mathsf{Instance}, \bigcup_i \{x_i \mapsto \#k_i\} \cup \boldsymbol{a}(\mathsf{T}\#j), \mathsf{T}\#j\right)\right\}\end{array}}$$

Figure VI.10: Simplified semantic rules from CL/I to the CLIR



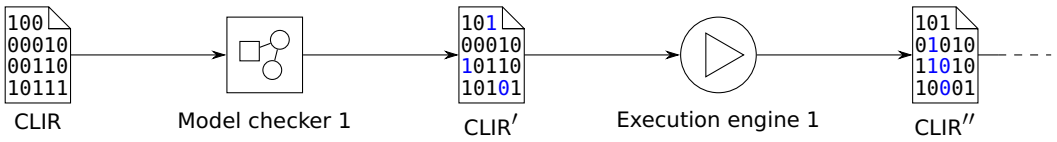


Figure VI.11: Two-stage processing of CL/I models

VI.2.5 Extensions

The CL/I language is designed to be compiled in two stages. The first one produces the intermediate representation we have shown. This intermediate representation enables tools in the second compilation stage to process symbols independently of the initial naming and AST structure. The second stage schedules processing passes on the symbol table. These passes can for example be execution engines, to run models, or model checkers and theorem provers, to prove properties on them. The interactions are shown on figure VI.11, where tools directly manipulate the CLIR and can modify its values. We have developed a simple runtime engine for computing model values, but we leave it to the reader to explore the compiler's source code[9] for details (which we feel would make this dissertation unwieldy). In the rest of this chapter, we focus on linking our language to Z3.

Before moving on to the next section, let us extend our type grammar to include binary relation types, useful to express logical properties:

$$\begin{array}{lll} T & \mathrel{+}= & \mathsf{Rel}\langle R \rangle \\ R & ::= & \mathsf{Equivalence} \mid \mathsf{Symmetric} \mid \mathsf{Asymmetric} \mid \ldots \end{array}$$

Let us also extend our value grammar to add universals, existentials and assertions:

$$V \mathrel{+}= \mathsf{ForAll}(\vec{r}, \vec{v}) \mid \mathsf{Exists}(\vec{r}, \vec{v}) \mid \mathsf{Assert}(v)$$

[9] Available on Github (`https://github.com/CAPRICA-Project/CL-I`).

It goes without saying that new AST structures are also defined, but we do not present them here for the sake of brevity. These new constructs can be used like so:

```
component C {
  let <<contains>> : Z3.Relation.Asymmetric;
}

for all (c1 : C, c2 : C, c3 : C) {
  (c1 <<contains>> c2 and c2 <<contains>> c3) => c1 <<contains>> c3;
};
```

VI.3 Mapping into Z3

Now that we have extended our grammars, let us dive into the mapping of our language with Z3.

> **SMT-LIB primer.**
>
> SMT-LIB [Barrett17] is an initiative to ease the development of SMT-based systems by providing established standards. Among the elements of SMT-LIB, we use in this chapter its so-called input langage for SMT solvers (which is the input language of Z3). Before explaining the translation of our modeling language into Z3, we need to describe the basic elements of SMT-LIB on which we rely.
>
> First, the syntax of SMT-LIB uses the Lisp notion of S-expressions. They are either made of a single token or a sequence of S-expressions surrounded by parentheses. The most important tokens for us are numerals, strings, symbols and keywords. A symbol is either a sequence of "unquotted" letters, digits and characters like = or >, or "quoted" strings (*i.e.* strings between vertical bars |) which can contain spaces or special characters such as #.
>
> Second, a SMT-LIB program is called a *script*. It is composed of a sequence of *commands*. The complete language contains a large number of commands but our translation uses only four of them: `assert` to add a new assertion, `check-sat` to check the satisfiability of the previously defined assertions, `declare-sort` and `declare-fun` to respectively declare a new symbol for a *sort* and a *function*.
>
> Third, assertions are *terms* expressing formulae in a multi-sorted first-order logic where each term is associated to a sort (its type) and the composition of terms must be well-sorted. The solver comes with predefined sorts such as `Bool`, `Int` or `String` and the user may declare new ones. Function symbols have a rank specifying the sorts of their inputs and results. We rely on the `forall` and `exists` variable binders, which quantify over a set of variables, each from a specified sort. Terms are built from binders, the standard logical connectives, and symbols either predefined or defined by the user.

In the rest of this chapter, we denote ε the empty SMT-LIB script. Composing several scripts may be denoted $\sqcap$ or expressed by putting the scripts one below the other. We use conjunction $\wedge$ and disjunction $\vee$ operators to compose terms to produce respectively `(and` $term_1$ `...` $term_n$`)` and `(or` $term_1$ `...` $term_n$`)` in Z3. Finally, we denote $\bullet$ the concatenation operator. We write in blue the meta-variables and in orange the meta-function applications that appear inside pairs of vertical bars, to distinguish them

$$\mathcal{S}(\texttt{T\#}i) = \|\texttt{Bool}\| \quad \text{if } \tau(\texttt{T\#}i) = \text{Boolean} \qquad \mathcal{S}(\texttt{T\#}i) = \|\texttt{Int}\| \quad \text{if } \tau(\texttt{T\#}i) = \text{Integer}$$
$$\mathcal{S}(\texttt{T\#}i) = \|\texttt{String}\| \quad \text{if } \tau(\texttt{T\#}i) = \text{String} \qquad \mathcal{S}(\texttt{T\#}i) = \|\mathcal{S}(\texttt{T\#}j)*\| \quad \text{if } \tau(\texttt{T\#}i) = \text{List}\langle\texttt{T\#}j\rangle$$
$$\mathcal{S}(\texttt{T\#}i) \text{ undefined} \quad \text{if } \tau(\texttt{T\#}i) \in \{\text{Unit}, \text{Rel}\,\langle\cdot\rangle\}\ ^a \qquad \mathcal{S}(\texttt{T\#}i) = \|\texttt{|T\#}i\texttt{|}\| \quad \text{otherwise}$$

$$\mathcal{S}(\#i) = \mathcal{S}(\boldsymbol{t}(\#i))$$

$$\mathcal{A}(\#i) = \bigwedge_{a \mapsto \#j \in \boldsymbol{a}(\#i)} \|(= (\texttt{|attr!}t(\#i)\texttt{!}a\texttt{|}\ \texttt{|\#}i\texttt{|})\ \texttt{|\#}j\texttt{|})\|$$

$$\mathcal{Q}(\#i) = \bigcdot_{\cdot \mapsto \#j \in \boldsymbol{a}(\#i)} \|(\texttt{|\#}j\texttt{|}\ \mathcal{S}(\#j))\|$$

$$\begin{aligned}
\mathcal{V}(\text{Boolean true}) &= \|\texttt{true}\| & \mathcal{V}(\text{Boolean false}) &= \|\texttt{false}\| \\
\mathcal{V}(\text{Integer}\,i) &= \|i\| & \mathcal{V}(\text{String "}s\text{"}) &= \|\texttt{"}s\texttt{"}\| \\
\mathcal{V}(\text{Ref}\,i) &= \|\texttt{|\#}i\texttt{|}\| & \mathcal{V}(\text{Assert}(v)) &= \|\mathcal{V}(v)\| \\
\mathcal{V}(\text{App}(f, \vec{v})) &= \Big\| (Z3(f)\ \cdot \overrightarrow{\mathcal{V}(v)}) \Big\| \\
\mathcal{V}(\text{ForAll}\,(\vec{r}, \vec{v})) &= \Big\| \begin{array}{l} \texttt{(forall (}\ \bigcdot_{\#i \in \vec{r}} (\|(\texttt{|\#}i\texttt{|}\ \mathcal{S}(\#i))\| \cdot \mathcal{Q}(\#i))\texttt{)} \\ \quad \texttt{(=>}\ \bigwedge_{\#i \in \vec{r}} \mathcal{A}(\#i)\ \wedge \overrightarrow{\mathcal{V}(v)}\texttt{))} \end{array} \Big\| \\
\mathcal{V}(\text{Exists}\,(\vec{r}, \vec{v})) &= \Big\| \begin{array}{l} \texttt{(exists (}\ \bigcdot_{\#i \in \vec{r}} \|(\texttt{|\#}i\texttt{|}\ \mathcal{S}(\#i))\|\texttt{)} \\ \quad \texttt{(forall (}\ \bigcdot_{\#i \in \vec{r}} \mathcal{Q}(\#i))\ \texttt{(=>}\ \bigwedge_{\#i \in \vec{r}} \mathcal{A}(\#i)\ \wedge \overrightarrow{\mathcal{V}(v)}\texttt{)))} \end{array} \Big\|
\end{aligned}$$

[a] This case is unreachable.

Figure VI.12: Helper functions for our translation rules

from Z3 elements. We highlight in `|gray|` the various name manglings[10] we use in Z3. In this section, we first present the translation rules between the CLIR and Z3, before discussing how to check models using Z3.

VI.3.1 Translation rules

We present in figure VI.13 the translation rules that we use. These rules use helper functions defined in figure VI.12. We have not detailed the $Z3$ function for brevity; its goal is to reinterpret CL/I built-in functions (Integer comparisons, String operations...) into Z3 built-in functions.

The rest of this section explains how the translation rules work, and we invite the reader to go back and forth between the explanations and the rules, using their numbers (in the leftmost column of figure VI.13)[11].

Our translation maps each CL/I type to a Z3 sort, either by declaring a new one in the case of components (Abstract types) and lists (List$\langle\cdot\rangle$ types) [rule 1] or an existing one [rule 2]. CL/I values are mapped to Z3 as nullary function symbols declared as returning the correct sort [rule 3][12]. Components

[10] The transformations from CLIR elements into valid Z3 names.

[11] Rules and their explanations are clickable in the digital version of this document to simplify the navigation.

[12] In Z3, constants are nullary functions. For example, an integer can be defined as a function from `()` to `Int` returning the value of said integer.

$$
\begin{array}{lll}
1: & [\![\texttt{T\#}i]\!]_{\mathcal{S}} = \|\texttt{(declare-sort } \mathcal{S}(\texttt{T\#}i)\texttt{)}\| & \text{if } \tau(\texttt{T\#}i) \in \{\mathsf{Abstract}, \mathsf{List}\langle\cdot\rangle\} \\
2: & [\![\texttt{T\#}i]\!]_{\mathcal{S}} = \varepsilon & \text{otherwise} \\
\\
3: & [\![\texttt{\#}i]\!]_{\mathcal{T}} = \|\texttt{(declare-fun |\#}i\texttt{|} \texttt{ () } \mathcal{S}(\texttt{\#}i)\texttt{)}\| & \\
\\
4: & [\![\texttt{T\#}i]\!]_{\mathcal{A}} = \prod_{a \mapsto \texttt{\#}j \in a(\texttt{T\#}i)} \left\| \begin{array}{l} \texttt{(declare-fun |attr!T\#}i\texttt{!}a\texttt{|} \texttt{ (|T\#}i\texttt{|) } \mathcal{S}(\texttt{\#}j)\texttt{)} \\ \texttt{(assert (forall ((c |T\#}i\texttt{|))} \\ \quad \texttt{(exists ((v } \mathcal{S}(\texttt{\#}j)\texttt{))} \\ \quad\quad \texttt{(= (|attr!T\#}i\texttt{!}a\texttt{|} \texttt{ c) v))))} \end{array} \right\| & \text{if } \tau(\texttt{T\#}i) = \mathsf{Abstract} \\
5: & [\![\texttt{T\#}i]\!]_{\mathcal{A}} = \varepsilon & \text{otherwise} \\
6: & [\![\texttt{\#}i]\!]_{\mathcal{A}} = \|\texttt{(assert } \mathcal{A}(\texttt{\#}i)\texttt{)}\| & \text{if } \tau(t(\texttt{\#}i)) = \mathsf{Abstract} \\
7: & [\![\texttt{\#}i]\!]_{\mathcal{A}} = \varepsilon & \text{otherwise} \\
\\
8: & [\![\texttt{T\#}i]\!]_{\mathcal{L}} = \left\| \begin{array}{l} \texttt{(declare-fun |<has:}\mathcal{S}(\mathsf{List}\langle\texttt{T\#}j\rangle)\texttt{,}\mathcal{S}(\texttt{T\#}j)\texttt{|} \\ \quad (\mathcal{S}(\mathsf{List}\langle\texttt{T\#}j\rangle) \ \mathcal{S}(\texttt{T\#}j)) \ \texttt{Bool),} \\ \texttt{(declare-fun |<can have:}\mathcal{S}(\mathsf{List}\langle\texttt{T\#}j\rangle)\texttt{,}\mathcal{S}(\texttt{T\#}j)\texttt{|} \\ \quad (\mathcal{S}(\mathsf{List}\langle\texttt{T\#}j\rangle) \ \mathcal{S}(\texttt{T\#}j)) \ \texttt{Bool),} \\ \texttt{(assert (forall ((l } \mathcal{S}(\mathsf{List}\langle\texttt{T\#}j\rangle)\texttt{)) (v } \mathcal{S}(\texttt{T\#}j)\texttt{))} \\ \quad \texttt{(=> (|<has:}\mathcal{S}(\mathsf{List}\langle\texttt{T\#}j\rangle)\texttt{,}\mathcal{S}(\texttt{T\#}j)\texttt{|} \texttt{ l v)} \\ \quad\quad \texttt{(|<can have:}\mathcal{S}(\mathsf{List}\langle\texttt{T\#}j\rangle)\texttt{,}\mathcal{S}(\texttt{T\#}j)\texttt{|} \texttt{ l v))))} \end{array} \right\| & \text{if } \tau(\texttt{T\#}i) = \mathsf{List}\langle\texttt{T\#}j\rangle \\
9: & [\![\texttt{T\#}i]\!]_{\mathcal{L}} = \varepsilon & \text{otherwise} \\
\\
10: & [\![\texttt{\#}i]\!]_{\mathcal{L}} = \left\| \begin{array}{l} \prod_{\texttt{\#}k \in \vec{v}} \left\| \begin{array}{l} \texttt{(assert (|<has:}\mathcal{S}(\mathsf{List}\langle\texttt{T\#}j\rangle)\texttt{,}\mathcal{S}(\texttt{T\#}j)\texttt{|} \\ \qquad\qquad \texttt{|\#}i\texttt{|} \ \texttt{|\#}k\texttt{|))} \end{array} \right\| \\ \texttt{(assert (forall ((v } \mathcal{S}(\texttt{T\#}j)\texttt{))} \\ \quad \texttt{(=> (|<has:}\mathcal{S}(\mathsf{List}\langle\texttt{T\#}j\rangle)\texttt{,}\mathcal{S}(\texttt{T\#}j)\texttt{|} \texttt{ |\#}i\texttt{|} \texttt{ v)} \\ \quad\quad (\vee_{\texttt{\#}k \in \vec{v}} \|\texttt{(= v |\#}k\texttt{|)}\|)))) \end{array} \right\| & \begin{array}{l} \text{if } \ v(\texttt{\#}i) = \mathsf{List}\ \vec{v} \text{ and} \\ \tau(t(\texttt{\#}i)) = \mathsf{List}\langle\texttt{T\#}j\rangle \end{array} \\
11: & [\![\texttt{\#}i]\!]_{\mathcal{L}} = \varepsilon & \text{otherwise} \\
\\
12: & [\![\texttt{\#}i]\!]_{\mathcal{V}} = \varepsilon & \text{if } v(\texttt{\#}i) \in \{\mathsf{List}\ \cdot, \mathsf{Instance}\} \\
13: & [\![\texttt{\#}i]\!]_{\mathcal{V}} = \|\texttt{(assert } \mathcal{V}(v(\texttt{\#}i))\texttt{)}\| & \text{if } v(\texttt{\#}i) \in \{\mathsf{ForAll}\ \cdot, \mathsf{Exists}\ \cdot, \mathsf{Assert}\ \cdot\} \\
14: & [\![\texttt{\#}i]\!]_{\mathcal{V}} = \|\texttt{(assert (= |\#}i\texttt{|} \ \mathcal{V}(v(\texttt{\#}i))\texttt{))}\| & \text{otherwise}
\end{array}
$$

Figure VI.13: Translation rules

may have attributes; being an attribute is expressed by a specific function symbol whose name mangles the type identifier and the attribute's name [rule 4][13]. This function is defined to take the (component) instance and return the value associated with the attribute. It also comes with an assertion that any instance of this type has this attribute. In our implementation, attributes only make sense for components [rule 5]. Furthermore, for each instance, an assertion ensures that the function symbol for each of its attributes returns the corresponding value [rule 6]. Similarly, the behavior is only present for component instances [rule 7]. In our encoding, list types require the definition of two membership testing operators: one to test whether an element belongs to a list and one to test whether it is possible that an element belongs to a list [rule 8][14]. List processing only make sense when we have lists [rule 9].

[13] For example, we encode the attribute `a` of the type symbol `T#i` as `|attr!T#i!a|`.

[14] It allows us to write modal rules directly in our CL/I code.

The fact that a value is in a list makes use of membership operators; additionally, we add an assertion to ensure that no other element can belong to the list [rule 10][15]. Similarly, the behavior is only present for lists [rule 11]. After this processing, lists do not need more consideration [rule 12]. Quantifiers and assertions are directly translated into Z3 assertions [rule 13]. Finally, the fact that a symbol has a value is expressed by an equality assertion [rule 14]. We show in figure VI.14 a concrete example of a translation between our language, our intermediate language and Z3.

CL/I

```
component File {
  let path: String;
  let perm: Integer;
}

for all (file : File) {
  String.ends_with(file.path, "/.ssh")
    => (file.perm = 700);
};

let file = File {
  path ← "/root/.ssh";
  perm ← 700;
}
```

CLIR

$$
\begin{aligned}
&T\#1 \mapsto (\text{String}, \varnothing)\\
&T\#2 \mapsto (\text{Integer}, \varnothing)\\
&T\#3 \mapsto (\text{Unit}, \varnothing)\\
&T\#4 \mapsto (\text{Abstract}, [\text{“path”} \mapsto \#1, \text{“perm”} \mapsto \#2])\\
&\#1 \mapsto (?, \varnothing, T\#1)\\
&\#2 \mapsto (?, \varnothing, T\#2)\\
&\#3 \mapsto (\text{ForAll } (\#4)\ (\text{App}(\\
&\qquad \text{Ext “Math\=>”}, [\\
&\qquad\quad \text{App}(\\
&\qquad\qquad \text{Ext “String\starts\_with”},\\
&\qquad\qquad [\text{Ref } 5, \text{String “/.ssh”}]\\
&\qquad\quad), \text{App}(\\
&\qquad\qquad \text{Ext “Math\=”},\\
&\qquad\qquad [\text{Ref } 6, \text{Integer } 700]\\
&\qquad\quad)\\
&\qquad]\\
&\quad)), \varnothing, T\#3)\\
&\#4 \mapsto (\text{Instance}, [\text{“path”} \mapsto \#5, \text{“perm”} \mapsto \#6], T\#4)\\
&\#5 \mapsto (?, \varnothing, T\#1)\\
&\#6 \mapsto (?, \varnothing, T\#2)\\
&\#7 \mapsto (\text{Instance}, [\text{“path”} \mapsto \#8, \text{“perm”} \mapsto \#9], T\#4)\\
&\#8 \mapsto (\text{String “/root/.ssh”}, \varnothing, T\#1)\\
&\#9 \mapsto (\text{Integer } 700, \varnothing, T\#2)
\end{aligned}
$$

Z3

```
(declare-sort |T#4|)
(declare-fun |attr!T#4!path| (|T#4|) String)
(declare-fun |attr!T#4!perm| (|T#4|) Int)
(assert (forall ((c |T#4|))
  (exists ((v String))
    (= (|attr!T#4!path| c) v))))
(assert (forall ((c |T#4|))
  (exists ((v Int))
    (= (|attr!T#4!perm| c) v))))
(declare-fun |#1| () String)
(declare-fun |#2| () Int)
(assert (forall ((|#4| |T#4|)
                 (|#5| String)
                 (|#6| Int))
  (=> (and (= (|attr!T#4!path| |#4|) |#5|)
           (= (|attr!T#4!perm| |#4|) |#6|))
      (=> (str.suffixof "/.ssh" |#5|)
          (= |#6| 700)))))
(declare-fun |#7| () |T#4|)
(declare-fun |#8| () String)
(declare-fun |#9| () Int)
(assert (= (|attr!T#4!path| |#7|) |#8|))
(assert (= (|attr!T#4!perm| |#7|) |#9|))
(assert (= |#8| "/root/.ssh"))
(assert (= |#9| 700))
```

Figure VI.14: Translation from CL/I to the CLIR to Z3

[15] When we define a list `[1;2;3]`, we tell Z3 that `1`, `2` and `3` are in the list, and no other element (else, Z3 could consider that other elements belong to the list).

VI.3.2 Conformance checking

Model checking with Z3 can be leveraged in two development stages: during the design of systems, to check the consistency and non-contradiction of requirements, and during the exploitation and instantiation of such models, to check the conformance of the instances to the requirements. At the design phase, Z3 can help identify problems in models, proving whether there can be instances of them or not. For the instantiation of models, Z3 can identify violations of model properties by the instances. We illustrate in figure VI.14 the translation from CL/I to the CLIR to Z3 of a simplified version of the model in figure VI.6 (page 86) with an added Z3 assertion, where permissions are just Integers and file owners are omitted for brevity.

If Z3 proves the set of formulae to be unsatisfiable, we could extract a subset of formulae (called an *unsat core*) used to derive unsatisfiability and link them back to the symbols that led to them, then to the model source code. If the set of formulae is proved to be satisfiable, we could extract an *instance* (called a *model* in Z3) and express it into our language. Both features are not yet implemented.

VI.4 Case studies

Modern cloud-based infrastructures offer resilience and scalability capabilities that are attractive to businesses. However, these qualities come at the cost of an abstraction that is not always fully understood, and many layers of indirection. Resource virtualization doesn't eliminate the need to correctly provision infrastructures, and this abstraction can give rise to new problems, such as network and storage contention [Kotsovinos10].

Some security and safety requirements may conflict with the abstraction principles offered by virtualization. For example, in the case of security, these may include i) rules prohibiting two competing customers' virtual machines from running on the same physical hosts. In the case of safety, it could be ensuring that ii) two virtual machines exchanging tens of gigabits of data per second always run on the same physical node, to prevent network congestion. Such rules are called i) *antiaffinity* and ii) *affinity* rules, but are not implemented by all virtualization solutions on the market, Proxmox VE being one of those where the feature is lacking. We focus in this case study on ensuring that even though Proxmox VE has no knowledge of affinity and antiaffinity, it can still be configured to enforce them.

VI.4.1 Virtual environment model

Typical virtualization environments have four core concepts: *nodes* (machines offering virtualization capabilities), *clusters* (of such nodes), *groups* (of nodes within a cluster) and *virtualized resources* (which we call *virts*, such as virtual machines and containers). Virts can be assigned to a specific group or run freely on any node. From this simple model, we add two structural rules:

R1. If a virt is assigned to a group, it can be assigned to any node within this group;
R2. If a virt is assigned to a group within a cluster, it must be assigned to a node within this cluster.

Some hypervisors such as Proxmox VE have a concept of (non-)restricted groups:

R3. If a virt is assigned to a non-restricted group, it can run on a node outside the group (*e.g.* if they are all offline);
R4. If a virt is assigned to a restricted group, it can never run on a node outside the group.

```
component Node {
  let virts : Virt*;
}

component Group {
  let nodes : Node*;
  let virts : Virt*;
  let is_restricted : Boolean;

  for all (node : Node, virt : Virt) {
    (nodes <<has>> node and virts <<has>> virt)
      => (node.virts <<can have>> virt); ** (R1)

    (nodes !<<has>> node and virts <<has>> virt and is_restricted)
      => (node.virts !<<can have>> virt); ** (R4)
  };
}

component Cluster {
  let nodes : Node*;
  let groups : Group*;

  for all (group : Group, virt : Virt) {
    (groups <<has>> group and group.virts <<has>> virt)
      => exists (node : Node) {
           nodes <<has>> node;
           node.virts <<has>> virt;
         }; ** (R2)

    for all (node : Node) {
      (groups <<has>> group and nodes <<has>> node and group.virts <<has>> virt
       and !group.is_restricted) => (node.virts <<can have>> virt);
    }; ** (R3)
  };
}
```

Figure VI.15: Model for `Node`, `Group` and `Cluster`. The definition of `Virt` is not shown.

This model is written in CL/I in figure VI.15. The rules R1 to R4 are translated to logical propositions as follows:

R1. If a virt is assigned to a group (`virts <<has>> virt`), it can be assigned to any node (`node.virts <<can have>> virt`) within this group (`nodes <<has>> node`);

R2. If a virt is assigned to a group (`group.virts <<has>> virt`) within a cluster (`groups <<has>> group`), it must be assigned to a node (`node.virts <<has>> virt`) within this cluster (`nodes <<has>> node`);

R3. If a virt is assigned to a non-restricted (`!group.is_restricted`) group (`group.virts <<has>> virt`), it can run on a node (`node.virts <<can have>> virt`) outside the group (*e.g.* if they are all offline);

R4. If a virt is assigned to a restricted (`is_restricted`) group (`virts <<has>> virt`), it can never run (`node.virts !<<can have>> virt`) on a node outside the group `nodes !<<has>> node`.

```
component Virt {
  let <<has affinity with>> : Relation.Equivalence;
  let <<has antiaffinity with>> : Relation.Symmetric;
  let <<can have affinity with>> : Relation.Equivalence;
  let <<can have antiaffinity with>> : Relation.Symmetric;
}

** Node affinity
for all (node_1 : Node, node_2 : Node, virt_1 : Virt, virt_2 : Virt) {
  (virt_1 != virt_2 and node_1.virts <<can have>> virt_1
   and node_2.virts <<can have>> virt_2 and virt_1 <<has affinity with>> virt_2)
    => (node_1 = node_2);

  (node_1.virts <<has>> virt_1 and node_2.virts <<has>> virt_2
   and virt_1 <<can have affinity with>> virt_2)
    => (node_1 = node_2);
};

** Node antiaffinity
for all (node_1 : Node, node_2 : Node, virt_1 : Virt, virt_2 : Virt) {
  (node_1.virts <<can have>> virt_1 and node_2.virts <<can have>> virt_2
   and virt_1 <<has antiaffinity with>> virt_2) => (node_1 != node_2);

  (node_1.virts <<has>> virt_1 and node_2.virts <<has>> virt_2
   and virt_1 <<can have antiaffinity with>> virt_2) => (node_1 != node_2);
};
```

Figure VI.16: Model for `Virt` and affinity and antiaffinity rules

Finally, the affinity relation is an equivalence (reflexive, symmetric and transitive) and the antiaffinity relation is (at least) symmetric; we model them as such in figure VI.16, along with their logical properties and modal counterparts.

VI.4.2 Proxmox VE configuration and execution

Configuring a Proxmox VE cluster can be done using a dedicated web interface, its web API or by directly editing files in the `/etc/pve` directory of a physical node. We focus in our study on three files: `/etc/pve/corosync.conf`, to configure the cluster and its nodes, `/etc/pve/ha/groups.cfg`, to configure high-availability groups (groups resilient to the loss of nodes), and `/etc/pve/ha/resources.cfg`, to configure virts. With a simple transformation script, we can convert such files into a set of CL/I instances, as shown in figure VI.17[16].

To check in real time the location of virts within a cluster and whether it invalidates the model, we can interrogate the Proxmox VE API at regular intervals and inject membership assertions such as `node0.virts <<has>> vm100` into our model. To ensure the reproducibility of our experiments, we decided to deploy our test cluster on Emulab [White02], a network testbed for distributed systems. To this end, we have developed an Emulab deployment profile, publicly available on Github[17].

[16] We have added three assertions on nodes to ensure that Z3 knows they are distinct.
[17] `https://github.com/CAPRICA-Project/emulab-proxmox`

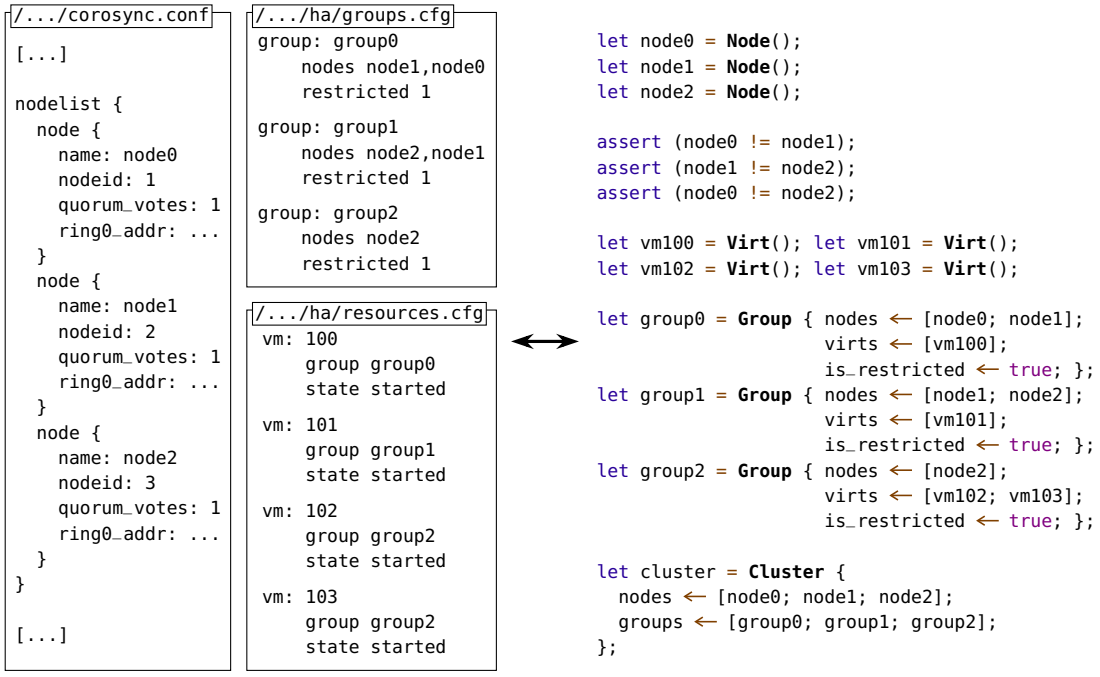


Figure VI.17: Proxmox VE configuration translation (parts of the files not shown)

	vm100	vm101	vm102	vm103
vm100	affinity	affinity? antiaffinity?	antiaffinity	antiaffinity
vm101	affinity? antiaffinity?	affinity	affinity? antiaffinity?	affinity? antiaffinity?
vm102	antiaffinity	affinity? antiaffinity?	affinity	affinity
vm103	antiaffinity	affinity? antiaffinity?	affinity	affinity

Table VI.1: Summary of relations. affinity, affinity?, antiaffinity and antiaffinity? respectively denote that `<<has affinity with>>`, `<<can have affinity with>>`, `<<has antiaffinity with>>` and `<<can have antiaffinity with>>` (and only them) are satisfied.

VI.4.3 Model checking

From our model, we want to check whether affinity and antiaffinity rules can and do hold. For example, asserting `vm100 <<can have affinity with>> vm101` checks whether there are configurations which do not violate an affinity between `vm100` and `vm101` in the model (there are, for example when both `vm100` and `vm101` are on `node1`). Conversely, asserting `vm100 <<has affinity with>> vm101` checks whether our model entails the affinity (it does not). Table VI.1 summarizes all affinity and antiaffinity rules that hold in the model, in a symmetric (because all our relations are) 4×4 matrix. For instance, `vm100 <<can have affinity with>> vm101`, because nothing forbids it, but $\neg$ (`vm100 <<has affinity with>> vm101`), because there are possible situations where it does not hold. Similarly, `vm102 <<has affinity with>> vm103`, because they are both in the same group (`group2`) and this group has a single node (`node2`).

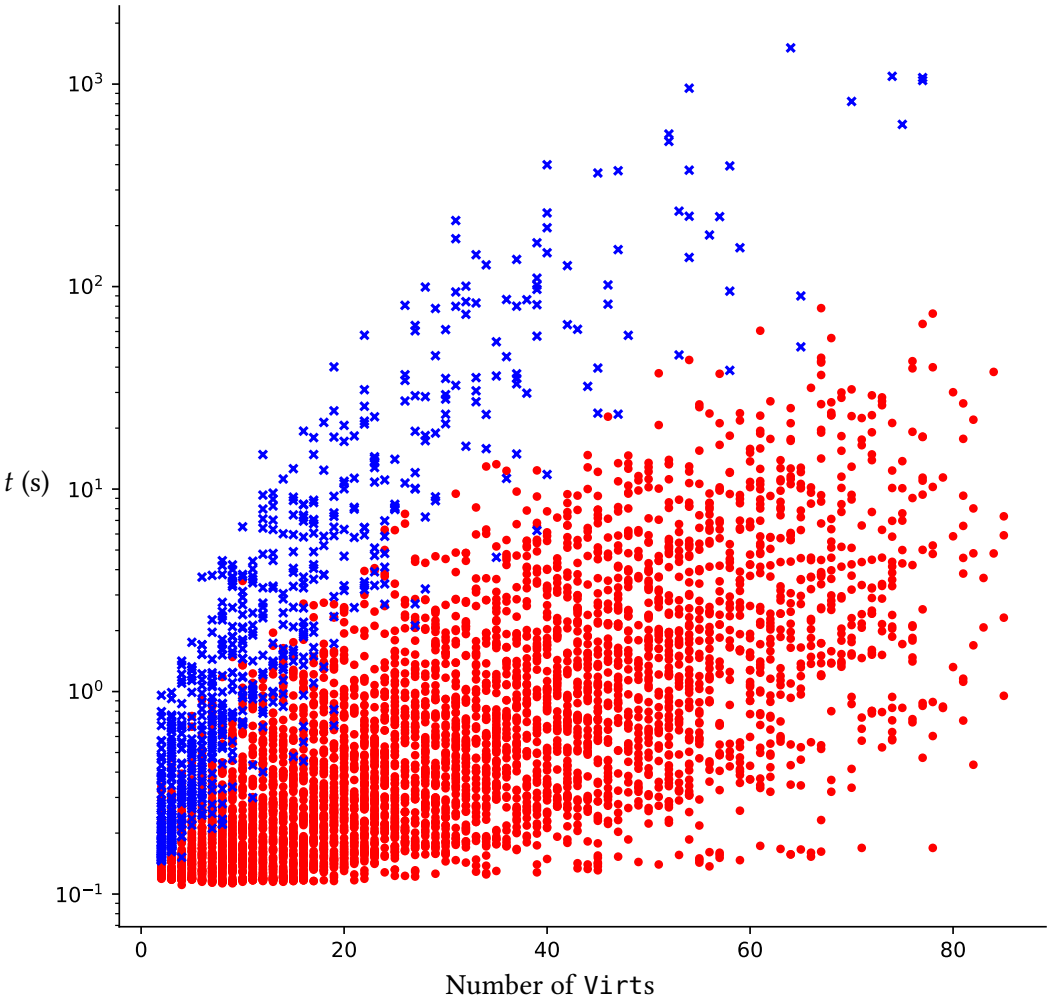


Figure VI.18: Verification time (log scale) as a function of the number of `Virts`
Legend. × `SAT`, ● `UNSAT`

Although our analysis of the model allows us to formulate the requirement "`vm102` and `vm103` have affinity", real-time observation of the cluster enabled us to identify a violation of this requirement. When an operator contradicts the cluster configuration and requests an illegal migration (*e.g.* moving `vm102` to `node0`), the hypervisor performs the migration, starts the machine on the other node, then migrates it back to `node2` to resolve the inconsistency. During the re-migration, the requirement is not met.

VI.4.4 Scaling

To validate the robustness of our approach, we generated a set of 5000 infrastructure models[18] by randomly varying the number of `Nodes`, `Groups`, `Virts` and affinity assertions, and by randomizing the distribution of `Nodes` and `Virts` within `Groups`. It should be noted that this distribution is made in such a way that we have a better chance of obtaining a satisfiable model (here, approximately 13 %) than with a uniform distribution. We have measured our compiler's execution times (almost exclusively Z3 verification times) for each of the models on a single core; we have not explored Z3's parallelization capabilities in our work. We plot on figure VI.18 the execution time for each model.

[18]Available on Github (`https://github.com/CAPRICA-Project/CL-I-Proxmox-case-study`).

Several points emerge from this study:

- Infrastructures where constraints are satisfied are much slower (up to 3 orders of magnitude) to verify than infrastructures where the verdict is `UNSAT`;
- The strongest (exponential) contributors to verification time seem to be the number of `Nodes` and `Virts`, the other parameters do not seem to have any particular influence;
- The verification time seems reasonable to us for projects where affinity between virtual machines is critical (all our examples were checked in less than an hour).

VI.4.5 A more complete case study

Now that we have validated our approach on a technical infrastructure (within a single business domain), we validate it by using CL/I to verify models spanning across several domains within a company. We use the UML diagram shown in figure VI.19 as the basis for our case study. First, employees have a set of skills that increase through training. Rights can be given to employees to allow them to access an application, provided they have a set of required skills. Applications are able to host other applications, for example a VM hosting business software. Finally, structures with employees can be given access to applications, for example, a SaaS solution (the application) sold to a corporate customer (the structure) for its employees (which become users of the application).

All the elements of this model can be extracted from a variety of data sources. Skills and training can be managed by a dedicated e-learning tool, employees can be managed in a human resources database, rights can be extracted from an access right management solution, applications can be listed in an inventory, and structures can be managed by a customer relationship management system. However, these tools lack interoperability and verifying properties across multiple domains can be a daunting task. Such properties could be:

- A user with access rights to an application must not have priviledged access rights to platforms hosting it. For example, such a user could modify an application's critical files on its VM to gain priviledged rights, which would lead to security concerns;
- Rights can only be assigned to users with the appropriate skills;
- Employees who have completed training courses acquire the skills associated with them;
- Competing companies must use applications hosted on different platforms, as studied previously.

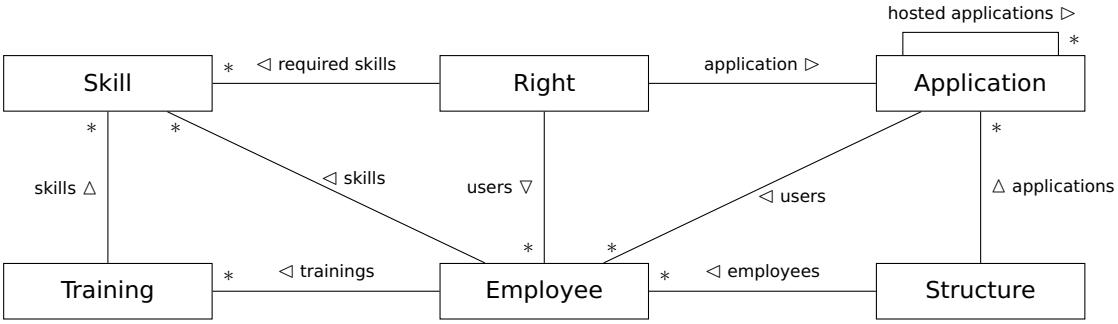


Figure VI.19: Multi-domain model

We have conducted this study, whose CL/I code can be found on Github[19], on a small fictitious company. We have modeled an enterprise with several dozen users and client companies (some in competition with others), as well as the applications they use. We have checked on this model the four previous properties (whose encoding is not detailed here). It appears that the verdict of satisfiability is slower than the unsatisfiability, again by a factor of 1000. However, we have been able to highlight, in a few seconds, problems with right management that can be tedious to spot by human operators. For example, we could identify a user who has attended courses that do not provide the required skills, or another user who has access to both an application and to the hypervisor that hosts the VM that hosts this application. The curious reader can consult the model online to appreciate the various properties that can be verified on it.

VI.5 Conclusion

The design of our language is evolving rapidly and its constructs are not completely fixed. Moreover, our compiler is still a prototype in stabilization which needs further testing and some edge cases are not yet implemented. However, current results seem promising and confirm our approach.

By using SMT solvers such as Z3 and allowing to write any number of nested quantifiers, we are exposed to combinatorial explosions, particularly during Z3's quantifier instantiation phase. Checking large and complex models may not terminate in a reasonable time, but the issues we have faced in our studies were either solved by modifying our semantic translations or by fine-tuning the Z3 engine. For all purposes, we have found the following parameters suited to our studies:

```
(set-option :tactic.default_tactic smt)
(set-option :sat.core.minimize true)
(set-option :smt.mbqi.max_iterations 5000)
(set-option :smt.mbqi.max_cexs 6)
```

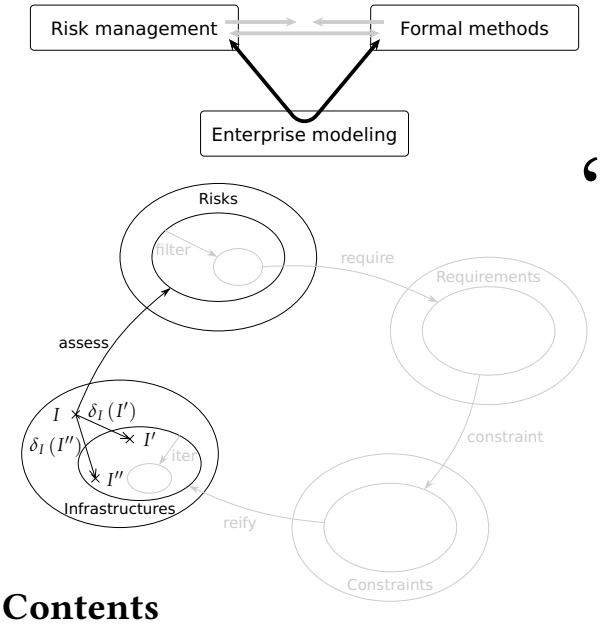
We currently rely on Z3's sorting as our (very simple) typing engine and intend to develop one supporting subtyping, subsorting being impossible in Z3. Subtyping will lead to sorting issues in Z3, as functions and relations sorted for a type T must also be sorted for types $T' <: T$. We are currently working on a special encoding to solve this issue.

The problems identified by Z3 and the instances it is able to produce are currently returned to the user in the SMT-LIB language, requiring the intervention of experts to debug models. We are working to add tracing from our language to its intermediate representation to its Z3 translation, to better identify which lines of code led to which errors. As shown in the previous chapter, models can be produced from many tools, and a natural extension of our work is to express problems and instances into such tools.

[19] https://github.com/CAPRICA-Project/CL-I-multi-domain-case-study.git

Chapter VII

Integrating our Approach



> “ Gaius Baltar — Who cares about this stupid planet anyway? All this song and dance over nothing. Less than 20% of that place actually supports human life, so...
>
> You’re not seeing the big picture. This is your new home, the place where you will lead a new life.
>
> — Number Six ”
>
> *Battlestar Galactica – Lay Down Your Burdens*

Contents

VII.1 Theoretical framework . . . 104
VII.1.1 Actors and responsibilities . . . 104
VII.1.2 Components and instances . . . 106
VII.1.3 Metamodel links . . . 106
VII.2 Collaborative enterprise modeling . . . 106
VII.2.1 Enterprise modeling . . . 107
VII.2.2 Collaborative modeling . . . 108
VII.3 Federating models . . . 109
VII.3.1 Modeling guidelines . . . 110
VII.3.2 Scaling infrastructures . . . 112
VII.4 Integration guidelines . . . 113
VII.4.1 Component catalogs . . . 113
VII.4.2 *A posteriori* modeling . . . 114
VII.4.3 *A priori* modeling . . . 116
VII.5 Case study . . . 116
VII.5.1 Heterogeneous models... . . . 116
VII.5.2 ... linked together . . . 118
VII.5.3 Exploiting the model . . . 118
VII.6 Conclusion . . . 120

Now that we have explored the theoretical and technical aspects of our work, we propose to focus on scaling the approach to large-scale enterprises. The framework of study we have set up allows to group together several expert models, to formalize their interactions, and to establish a multi-criteria risk analysis on various infrastructure elements. However, this approach assumes the absence of modeling errors (for both systems and links between them) and overlooks the way in which teams cooperate.

As a matter of fact, the proper operation of an IT infrastructure depends on more than just its hardware and software components. The human dimension and the performance of business processes are also important factors when we consider infrastructures in their corporate context. A parallel branch to our study, namely Enterprise modeling [Vernadat20], embraces such concepts, with two major challenges.

First, while human organizations are typically divided into various business domains (finance, support, cloud operations...), technical infrastructures tend to be structured in technical layers (hardware, software, network...). This leads to a misalignment between IT and business models, since there is no one-to-one mapping between these different views. Second, people who practice modeling (whether for IT or business infrastructures), are often well informed about good modeling practices, which is not necessarily the case for other employees. These employees, however, manipulate the infrastructure and have specific visions and technical knowledge that can benefit risk analyses. They should then be included in the modeling process [Voinov10], to properly capture the interactions within the infrastructure.

The importance of federating this knowledge is the focus of the final chapter of this dissertation. We first present the theoretical framework of our approach in section VII.1 through a component- and responsibility-oriented metamodel. Then, we discuss how to model enterprises in a collaborative way in section VII.2. Next, we look at several criteria for properly federating enterprise models in section VII.3. We then give guidelines on how to integrate our approach in a company in section VII.4. We propose a case study illustrating our work in section VII.5. Finally, we conclude this chapter in section VII.6.

This chapter was partly published in [Somers23.2].

VII.1 Theoretical framework

As part of our study, we have focused on the various components within an infrastructure, the actors who interact with them, and to a lesser extent their responsibilities, to draw a broad picture of the risk. We propose here to bring these elements together in a common model, combining a responsibility-oriented metamodel and a component-oriented metamodel. Our metamodel is divided in three parts: the actors and their responsibilities (on figure VII.1a), the components and their instances (on figure VII.1c), and the additional links in the metamodel (on figure VII.1b).

VII.1.1 Actors and responsibilities

We have decided to model *responsibilities* in our metamodel in a generic way. They can either be low-level (“read/write access over ▷”, “install updates on ▷”...) or high-level (“audit server ▷”, “manage project ▷”...). *Responsible entities* have *responsibilities* over *entities* and can assume *roles*. *Roles* represent generic sets of *responsibilities*. They can be used to encode access rights on an information system or positions in a company’s organizational chart for example. *Actors* represent the actual *entities* which can assume *roles* and have *responsibilities*. They can be used for example to encode users, allowed to

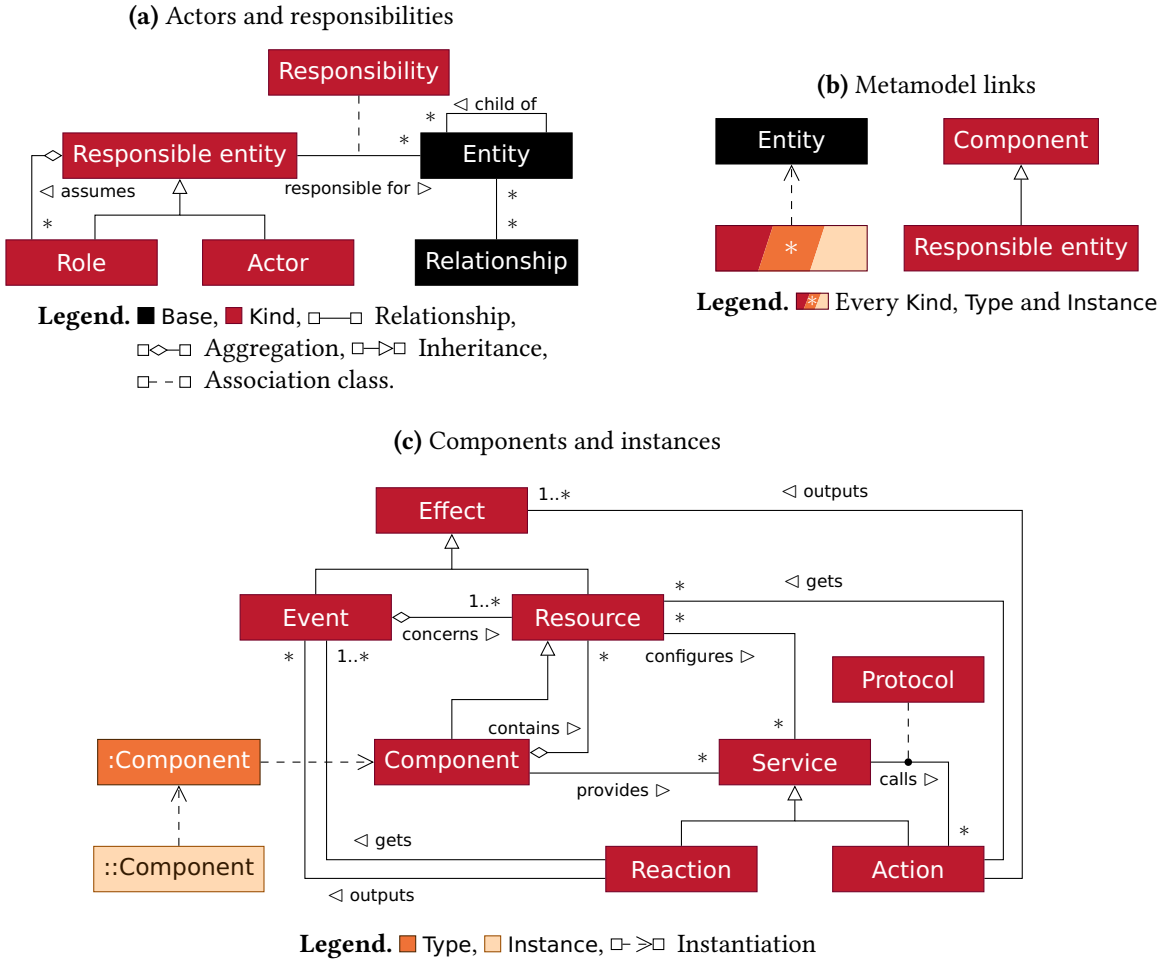


Figure VII.1: Component- and responsibility-oriented metamodel

access specific servers because of their positions, or even a whole company, responsible for the proper functioning of the products it sells. Finally, *relationships* represent all lines and arrows in models (and in the metamodel itself).

We have chosen an approach simpler to what can be found in metamodels such as ReMoLa [Feltus11] to keep our model generic, and because we did not need to elaborate much on the notion of *responsibility* in our work. If further refinement is required in modeling a system, we recommend specifying *responsibilities* as shown on figure VII.2 to benefit from ReMoLa's expressiveness by adding the concepts of Rights, Accountability and Obligations.

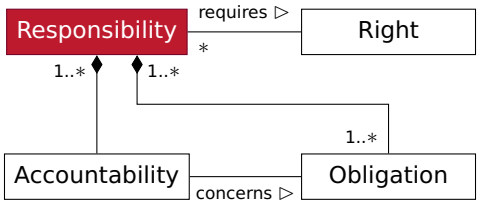


Figure VII.2: Specialization of the metamodel for ReMoLa compatibility

VII.1.2 Components and instances

The main focus of our metamodel is *components*. *Components* are *entities* that provide *services* (which we split between *actions*, and *reactions* to *events*). For example, a web service allowing employees to change their corporate passwords can be modeled as a *component* providing a "Change password" *service*. *Services* can in turn use other *components*' *actions* through *protocols*. For example, our "Change password" *service* can use a remote SQL database's "`UPDATE`" *action* to update passwords (using for example PostgreSQL's *protocol* 3.0). *Components* can also contain *resources*, that may be *entities* providing *services* (other *components*) or not (such as configuration files, web data...). *Events* are anything that can happen to *resources* (creation, change, deletion...). Both *resources* and *events* can be the result of an *action*, so we decided to unify them under the *effect* kind. Three layers appear in our metamodel:

- The Kind layer (), representing the core concepts of the metamodel;
- The Type layer (), representing "types of" these concepts. For example, "physical server" is a type of *component*. To follow the UML notation, a *component* type is represented here as :Component.
- The Instance layer (), representing "instances of" these concepts and types. For example, *a* physical server is an instance of "physical server" and "check web page availability" is a *service* instance. In our work, we have not encountered the need for *service* types. An instance of :Component is represented here as ::Component.

VII.1.3 Metamodel links

For consistency, every Kind, Type and Instance of the metamodel is an instance of *entity*; it means that *responsible entities* can have *responsibilities* over them. For example, an *actor* can be responsible for the development of a software *component* (on the Type layer) and another can be responsible for its configuration and deployment (on the Instance layer).

We have designed our metamodel to be reflexive, which allows *responsible entities* to be responsible for *responsibilities* themselves. It makes sense in a context of access management where a person may be responsible for the access rights given to another person. It also allows models to be seen themselves as *entities* in the metamodel, which makes it easy to represent situations where an employee is responsible for modeling a particular *component*. To the best of our knowledge, both characteristics are distinctive features of our approach.

Finally, *responsible entities* are also *components*, and their behaviors can be encoded as *services*. For example, a system operator updating a server can be represented as both an *actor* with *roles* inherited from their job position ("IT employee", "system administrator"...), and a *component* with *services* to perform server maintenance.

VII.2 Collaborative enterprise modeling

Enterprise modeling is instrumental in better understanding and designing organizations. The metamodel we have presented follows on from, but does not replace, a long tradition of modeling frameworks used in companies throughout the world. And enterprise modeling is no easy task: even though small systems can be designed and modeled by a few experts, creating a robust enterprise model is a collaborative endeavor. It demands the collective expertise of many stakeholders across a range of departments and domains, to produce a comprehensive overview of infrastructures.

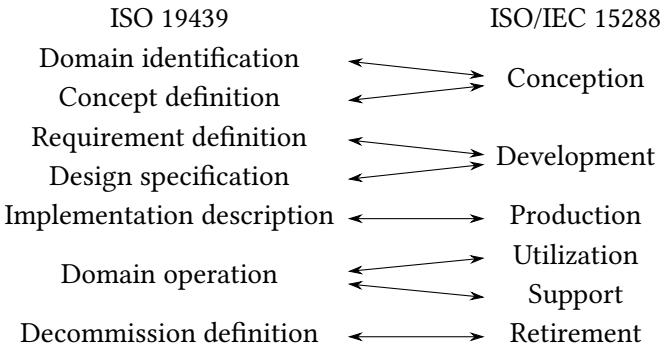
VII.2.1 Enterprise modeling

The field of enterprise modeling has evolved significantly since its inception [Vernadat20]. Incorporating formal methods, high-fidelity models can provide significant value to companies that adopt them [Cohn04]. However, producing such models is complex due to the wide range of professions and tools used within these companies, leading to discrepancies [vdAalst13]. As we described in section V.2, we are potentially talking about dozens of different approaches and representations, from software development to infrastructure design. To these approaches, we can also add everything that makes a company tick, from project management to finance, to human relations.

Standards like [ISO 19439] (*Enterprise integration — Framework for enterprise modelling*) and its precursors CIMOSA (*Computer Integrated Manufacturing Open Systems Architecture*) and GERAM (*Generalized Enterprise Reference Architecture and Methodology*) recommend modeling enterprises in (at least) four views:

- Function view, focusing on operations, processes and behaviors;
- Information view, dealing with data and information flows;
- Resource view, looking at human and technical resources;
- Organization view, addressing the organizational structure, roles and responsibilities within the enterprise.

Since we have mentioned software development, we think it is worthwhile to underline an important connection between ISO 19439 and [ISO/IEC/IEEE 15288] (*Systems and software engineering — System life cycle processes*). ISO 19439 identifies several modeling phases that can be mapped to ISO/IEC 15288's lifecycle stages, stressing similarities between enterprise and software modeling:



A representation of the standard is given in figure VII.3. Such a layered approach is found in many modeling frameworks, among which we can mention [RM-ODP] and [Archimate]. These frameworks provide a high-level way to model enterprises and allow to represent the infrastructures on which businesses lie. However, the large amount of concepts and the lack of precise semantics regarding IT infrastructures make the frameworks difficult to use in IT domains for people whose main job is not enterprise modeling [Lantow14]. Moreover, the matricial aspects of their approaches and their division into layers are not always suitable [Jørgensen09], as we very often need to cross boundaries.

One of the most important aspects of organizational and IT infrastructures we have captured in chapter VI is dynamism: systems are frequently modified, and companies keep evolving. These changes should therefore be reflected in enterprise models as well.

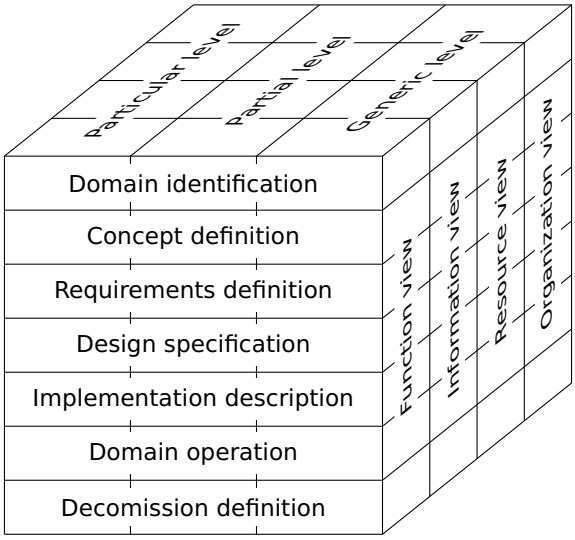


Figure VII.3: Representation of the ISO 19439 standard

[Greenwood95] identifies three types of model dynamics:

- Passive models, which represent systems as they are perceived at some point in time, without any change;
- Dynamic passive models, which represent dynamic systems without synchronization or links to the real system;
- Active models, which maintain consistency between models and actual systems by reacting to change.

The idea of active models, which dates back to the 90s, is a topic that drives the enterprise modeling community and is in line with the efforts to maintain our infrastructure models valid over time.

VII.2.2 Collaborative modeling

In the connected and open world of 2024, it would seem absurd for most companies to embark on large software development projects in small teams, with everything developed from the ground up. And yet, this is what we often observe in enterprise modeling [vdLinden20]. In a corporate environment covering several fields, we see many domain-specific languages that are adapted to their respective domains [Deursen00] and can represent in detail things that holistic frameworks cannot. But these languages are often not understandable by other parties. It leads to many metamodels [Kaczmarek15], often sharing the same core concepts [Breton00], being used to model enterprises.

The modeling process must be driven by a need and be undertaken by including all the professional disciplines concerned in a company. However, due to a lack of modeling skills, some stakeholders are excluded from the process [Renger08]. Work in the model federation community [Golra16] (where we maintain links between models expressed in different metamodels) is a step towards including the expertise of such stakeholders. This is the approach we have sought to defend in this dissertation. Other approaches, such as composition [Fleurey08] (where we build a common metamodel to align models) and unification [Vernadat02] (where we build a single model) are described and discussed in the literature [ISO 14258]. Figure VII.4 illustrates all three methods.

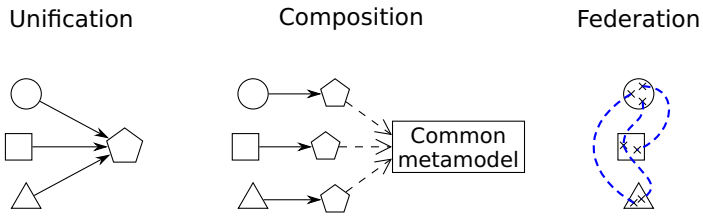


Figure VII.4: Unification, composition, federation: three approaches to collaborative models

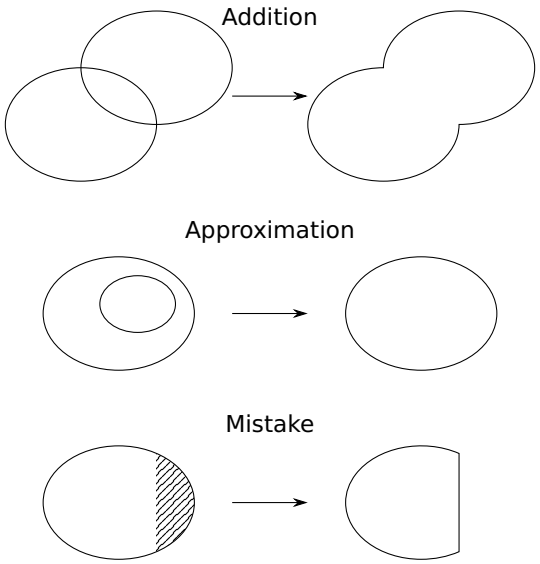


Figure VII.5: Inconsistencies by addition, approximation and mistake and their resolution

But if collaboration is unanimously supported, how it is implemented is a matter of debate. We do not claim to have a definitive solution to the problem, and the absence of a generic solution is a well-known fact in the Enterprise Modeling community. We would like to draw some thoughts from our experience of IT infrastructures in this chapter.

VII.3 Federating models

When trying to bring together several infrastructure models, we usually come up against a major obstacle: inconsistencies. We identify three categories of inconsistencies, represented on figure VII.5.

- Inconsistencies by addition, when several people talk about the same thing, but with different aspects (for example, a piece of software where different functions are used and modeled by two teams). Such inconsistencies are solved as a simple union of the aspects;
- Inconsistencies by approximation, when several people talk about the same thing, but with different levels of abstraction (for example, a user modeling their operating system as a black box able to execute applications and a systems engineer modeling it in greater detail). Such inconsistencies are solved by only keeping the most suitable granularity;
- Inconsistencies by mistake, when people make errors in their models (for example, if a piece of software is thought to perform an action that it does not). Such inconstistencies are solved by removing the mistakes.

Resolving these inconsistencies, through a process called model reconciliation, is an important step when building bigger models from smaller ones. Although reconciling approximations is a simple process, it can be difficult to tell the difference between complementary (which can be reconciled additively) and contradictory (where reconciliation involves solving conflicts) models.

VII.3.1 Modeling guidelines

We argue that three main principles should be respected to enable the federation of infrastructure models. We present them here, using examples expressed in our metamodel.

Non-intrusiveness ("start small, don't bother the neighbor").

The majority of IT domains use their own sets of tools and representations to communicate and to model systems [Amaral10]. People within the same domain can understand each other thanks to this shared jargon, but may find it difficult to communicate with experts in domains with which they are unfamiliar. This semantic boundary initially provides a scope within which each team can develop its own models. This is what [Sandkuhl18] calls "grass-roots modeling".

Experts can collaborate effectively while retaining the integrity and coherence of each model by working on clearly defined small models. Furthermore, they can use the most appropriate tools and techniques for their particular domains. Our metamodel provides a framework adapted to linking these tools together, rather than replacing them. "Locally-collaborative" work among employees mitigates the concerns raised by [Renger08] about employee exclusion and model reconciliation in the first phases of modeling.

When, and only when, the small local models are validated, it is advisable to begin modeling workshops between the various teams.

Refinability ("model what's necessary in your domain, elaborate later if needed").

To accurately mirror the actual design process of its components, people involved in infrastructure modeling should be able to incrementally refine their models when needed. As with risk analysis, the process can be carried out either in a top-down (where one adds details) or bottom-up (where one abstracts them) approach, or a combination of both, as shown in figure VII.6. In this example, a top-down approach would describe what a "service checker" is, by dividing it into sub-components (`Service checker`, made of `Checker API` and `Logger`) and then refining their services (`check`, `history`...). A bottom-up approach would be to describe the services wanted for a "service checker" and combining them into components and super-components providing them.

Iterative conception goes through a succession of incomplete models. When developing a new piece of software, a commonly used approach is to have end-users describe their needs and iteratively produce code that meets these needs [Ruparelia10]. Initial needs may be very imprecise and high-level and refinement steps may be necessary throughout the project's lifespan.

"Holes" in models can also arise when *blackbox* software (for example proprietary applications) or hardware (for example Hardware Security Modules) are deployed in an infrastructure. It can also happen with legacy components whose knowledge has been lost, for example due to employee turnover. Even though the knowledge of an IT infrastructure is partial, properties can still be deduced from its models. By allowing imprecision, we argue that the benefit is threefold:

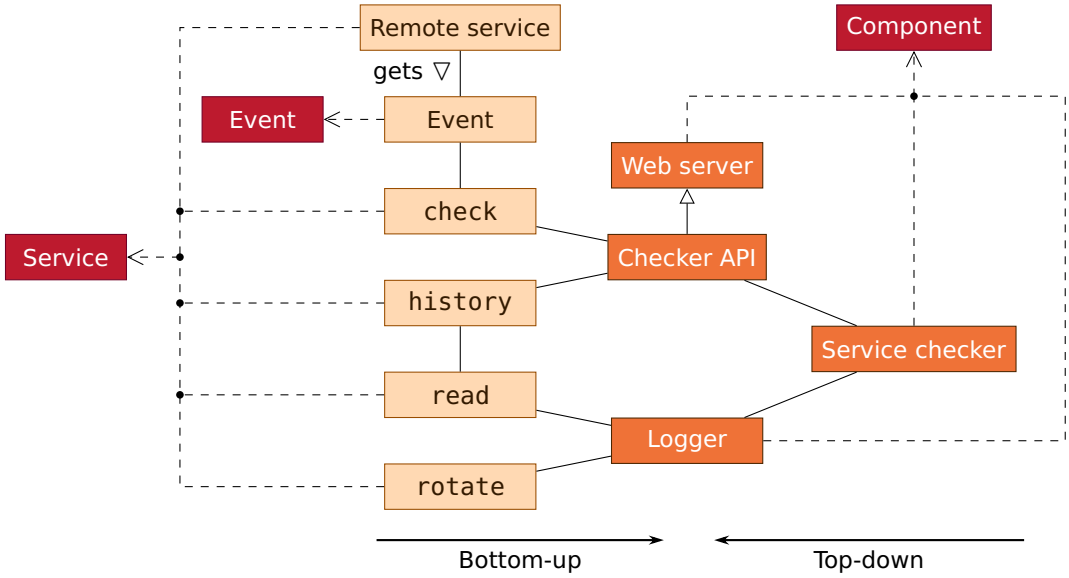


Figure VII.6: Model for a generic service checker
Legend. □⊢▷□ Inheritance, □-⩾□ Instantiation

- Coherence: instead of making wrong assumptions, modelers can express their lack of knowledge using black boxes and unknowns, not adding new inconsistencies between models;
- Reconciliability: employees should not attempt to refine a component they are not responsible for, simplifying the reconciliation phase and ensuring that responsibilities are respected. Only the responsible actors participate and changes can then be properly tracked;
- "Fail-early"-ness: some safety and security issues can be identified early when modeling projects iteratively; addressing them as soon as possible is critical to their success.

Correctness ("gather in small teams and align your models").

To obtain a comprehensive understanding of an IT infrastructure, it is crucial to engage all of its stakeholders in the modeling process. The sum of individual, local viewpoints is not enough to produce an overall model. Model reconciliation establishes links between these viewpoints which can carry additional semantics. As the literature shows [Nuseibeh03], model reconciliation is a complex task, this is why we strongly advocate to start the collaborative modeling process as soon as possible in projects.

If the modeling adheres to these principles, model reconciliation becomes a matter of refining black boxes in other models and linking them together, as illustrated in figure VII.7. Here, we depict three points of view, from three different teams:

- On the left, the organizational structure of a human resources department is represented;
- In the middle, we have the design of a web application using a remote database (not modeled) allowing some HR people (unknown at modeling time) to manage employees;
- On the right, there is a simple model of said database.

During the process of model reconciliation, teams align their vocabularies (`Human resources` and `HR`), may specify black boxes ("`?`" becomes `Personnel administration`) and combine knowledge (`10.2.42.1:3306` refers to `mysql-02-01`). Although there is no universal method to solve modeling conflicts, we think that modeling in incremental steps allows to solve them on smaller models. Rather

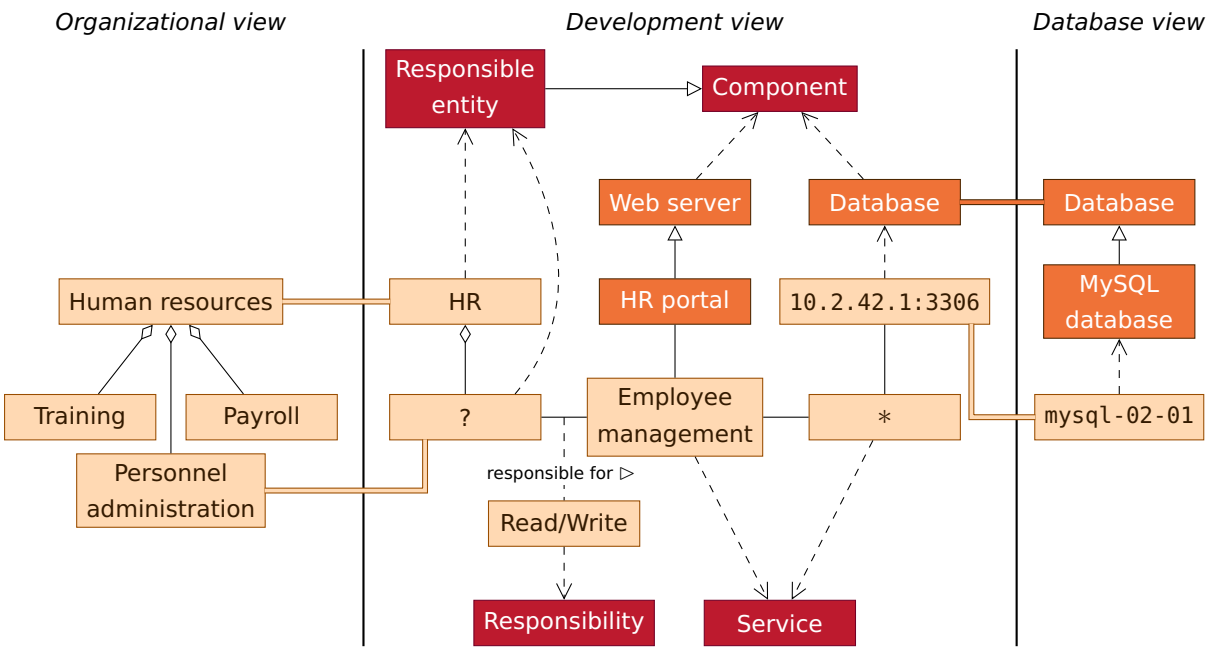


Figure VII.7: Model reconciliation with three points of view
Legend. ▭═▭ Reconciliation, ∗ Any service

than trying to fix the problem of reconciling models, we actually try to avoid it. The reconciliation process itself may be modeled using our metamodel, by assigning responsibilities to the employees performing the reconciliation. When a model is updated, it becomes easy to know who performed the reconciliation and notify them to review whether the change invalidates their work or not. In the case of a modeling process done with CL/I, a commit to a Git repository of a new version of a model could automatically trigger a validation workflow, for example. This idea, which to our knowledge has not been explored, simplifies keeping the overall model correct in the long term.

VII.3.2 Scaling infrastructures

Designing all infrastructure models by hand seems both unrealistic and counterproductive. Indeed, performing a complete modeling of a fleet of several thousands of machines or a workforce of hundreds of employees would take a considerable amount of time. As with risk analyses, models would also present validity issues over time. There are two main axes of automation for scaling up infrastructure models: automatic model construction and automatic model validation.

A number of inventory systems and Configuration Management DataBases (CMDB) exist, that provide a global view of the hardware and logical assets of an IT infrastructure. These databases can be leveraged to automatically perform a digital cartography of a company with minimal human intervention. At the same time, the many software applications used by companies contain vast quantities of data that can be extracted to gain valuable knowledge and produce rich models. However, the ability to extract meaningful exploitable data relies on the implementation of formally-defined semantics, which is often not the case in “grass-roots models”.

Finally, model validation heavily depends on these semantics. We think the complexity of the examples we have presented in chapters V and VI clearly illustrates this difficulty. In practice, we have identified two main obstacles to automatic validation of infrastructure models:

- The need for an oracle to differentiate, when combining two models representing a similar reality, between inconsistencies by addition (where both models represent complementary aspects) and inconsistencies by mistake (where some aspects are wrong);
- The need for an oracle to compare existing infrastructures with their models, to validate that the aspects represented are consistent with the reality.

We have not identified any generic, automated oracle that meets the first criterion. For the second one, the common infrastructure monitoring approaches used in the IT sector [Giamattei24] can meet the need, but they are defined on an *ad hoc* basis and are not suited to non-technical systems. For these however, the decentralisation of models we defend in the following section may open up new avenues.

VII.4 Integration guidelines

Modern IT companies typically have a combination of business processes that are more administrative in nature and very detailed technical workflows. It seems unrealistic to choose a holistic modeling framework that can cover all these aspects in detail. Instead, a federated approach like [ISO 14258] (Industrial automation systems — Concepts and rules for enterprise models) can leverage everyone's skills while achieving a more comprehensive modeling process. This approach preserves each stakeholder's metamodels and models, creating semantic connections between them. However, interdependencies and inconsistencies between models can hinder collaboration, particularly when changes to one model affect others. As noted in section VII.3.1, our framework enables identification and response to these changes.

In this section, we propose a methodological framework adaptable to concrete business processes, to build thorough yet accurate models using our metamodel. First, we insist on the importance of sharing models in so-called component catalogs. Then, we present two modeling scenarios: *a posteriori* modeling, where we model existing infrastructures descriptively, and *a priori* modeling, where we model future infrastructures prescriptively.

VII.4.1 Component catalogs

Companies typically design their systems by using external components. In the hardware world, systems frequently contain COTS components sold by other companies. In the software domain, third-party libraries and packages are essential components of modern systems.

This decentralized system design can apply to infrastructure modeling as well, by allowing manufacturers to produce their system models, which users can integrate into their infrastructure models. Such models can be made available in catalogs available internally within companies and externally to clients or for public use. There are two primary benefits to this approach. First, the responsibilities and knowledge are better distributed as the models are produced by the system designers rather than the users. Second, it speeds up the modeling process as model reuse, similar to code reuse in software

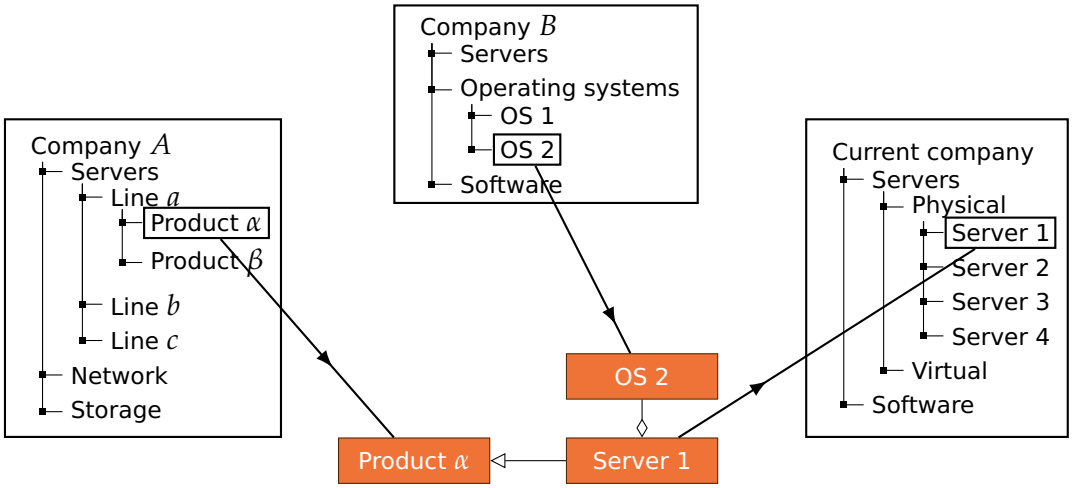


Figure VII.8: An example of model reuse

development, enables designers to create composite systems more quickly. This approach goes hand in hand with the idea of open risk analyses we have proposed in section IV.5, and shares many of its aspects (making analyses/models public and decentralized, developing a common exchange format...).

Both modeling steps, design and use, are illustrated in figure VII.8, where a company's `Server 1` is built using other companies' `Product` α and `OS 2`. An employee can then instantiate this `Server 1` architecture in their models without redesigning it. By allowing to model infrastructure components, our metamodel and CL/I can act as an exchange format for serializing models, which can then be reused by other employees or third parties.

VII.4.2 *A posteriori* modeling

Understanding the orchestration of a company's business processes can help optimize the existing and build the new in a more controlled way. In the banking industry for example, the use of legacy systems imposes technical choices that can only be made with a good knowledge of the existing architectures. For example, the development of software tools interfacing with mainframes presents challenges specific to the domain. Within the company's departments, this knowledge exists in the form of diagrams, source code, configuration files, *etc.*, which must first be identified (step 1 of figure VII.9). This step may be accompanied by hardware and software inventories if needed.

Next, the identified elements are mapped onto our metamodel. Care must be taken in the early stages of modeling to assign responsibilities properly to the model entities (who owns which product, who develops which service, who is responsible for modeling which component...). Step 2 of figure VII.9 illustrates this process.

In an iterative manner, business processes that interact with the modeled elements must be identified. As this is a collaborative process, we can learn from approaches such as Agile methods and apply them to the domain. Advocating for quick and iterative changes, we believe they are well-suited for our method; despite this, we still lack the technical tools to achieve this goal, and we think that future developments should focus on this. The interaction is shown in step 3 of figure VII.9. The resulting global model is an important artifact for the company: it can be used, as mentioned in the previous chapters, to carry out formal verifications, but also to train employees to better understand the company, to present the infrastructure to auditors... Modeling is not just a stylistic exercise, it serves concrete purposes.

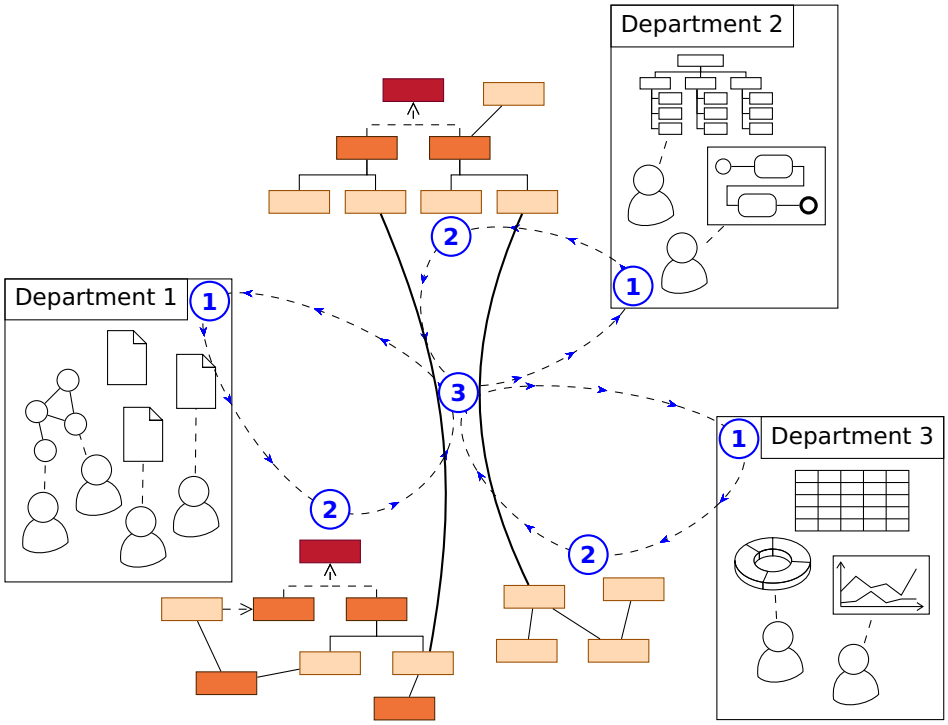


Figure VII.9: *A posteriori* modeling by federation. Step 1 corresponds to the inventory, step 2 is the modeling process and step 3 is the reconciliation.

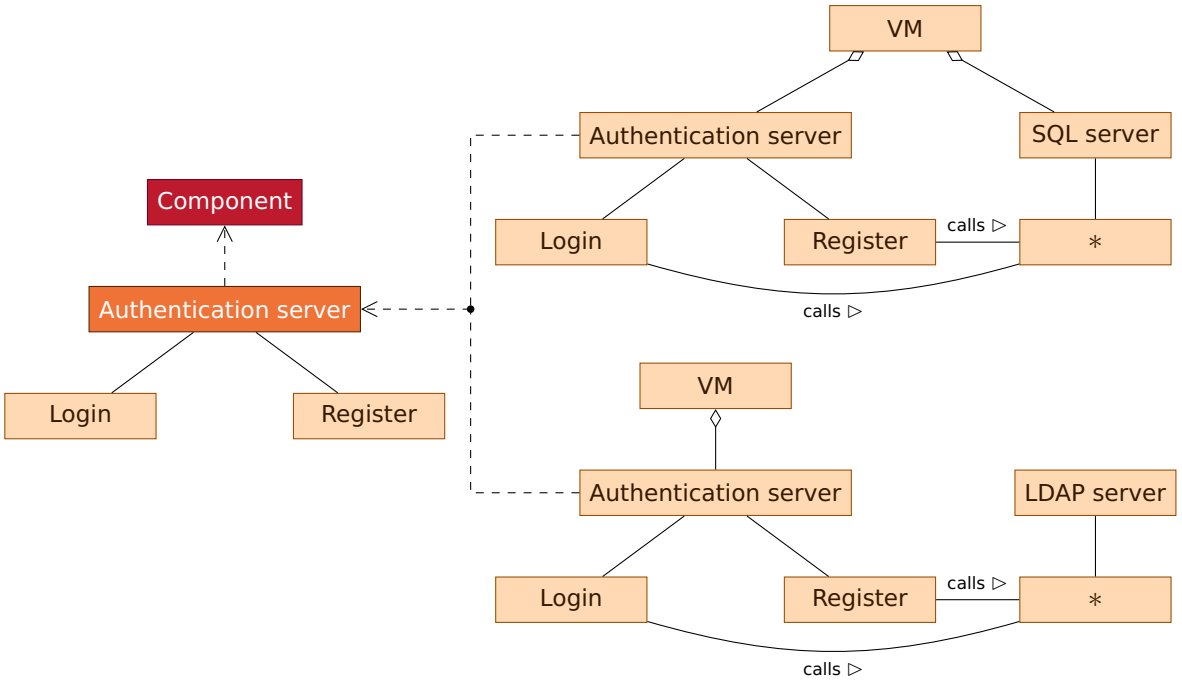


Figure VII.10: *A priori* modeling. Two instances of Authentication server are proposed to implement its specification. Here, we simplify the services provided by SQL server and LDAP server with ∗. Note that we are leaving the location of the LDAP server unspecified in this model.

VII.4.3 *A priori* modeling

Accurate and comprehensive models enable improved assessment of financial and technical costs of projects, better dimensioning of infrastructures, and the creation of *safe and secure by design* systems. Throughout a project's life cycle, it is crucial to ensure that systems do not deviate from their specifications due to lack of communication, misunderstandings, or the urge to move too quickly. Verification of expert-defined properties on these models, which we have discussed in section V.3, ensures system conformity prior to implementation and can aid in selecting an optimal technical solution.

We can draw a link between infrastructure models and software development: specifications can be seen as interfaces that the candidate models (classes) have to implement, extending the concepts from Object-Oriented Programming to infrastructure modeling. The idea of model typing is explored in [Steel07], and we have illustrated it with an example of two implementations of an authentication server in figure VII.10. In this example, we describe on the left the desired structure for our application, its structural specification, and on the right, instances of the specification model. These instances both conform to the desired structure, with `Login` and `Register` services, but with different implementations. It should be noted that while we can use our metamodel to ensure the syntactic conformance of models to their specification (whether the structure of models correspond to that of the specification), the semantic conformance (whether their behaviors are actually correct) must be verified by domain experts.

VII.5 Case study_

To illustrate our approach, let us now consider a fictitious banking company, called eBank. eBank provides banking and payment services to consumers and businesses. One of its flagship product, ePay, acts as a payment processor for companies and as an instant payment and expense sharing tool for consumers. The employees of the company want to have a better understanding of its overall processes and decided to use our approach to this end. In this section, we first make an inventory of the company's models. Then, we link these models together into our metamodel. Finally, we use the resulting big picture for a cross-model case study.

VII.5.1 Heterogeneous models...

The company decided to start its modeling by focusing on ePay's environment, namely the company's organizational structure, the business processes around the product and the technical architecture. Each department uses domain-specific modeling tools, leading to different views of the overall infrastructure.

Organizational structure.

eBank is structured in two directions: Technical and Administrative, each subdivided into structures, divided themselves into departments. An organizational chart of the company is given in figure VII.11.

Business processes.

The company's activities are guided by various business processes. For the sake of brevity, we consider here only the equipment purchase process, represented in figure VII.12.

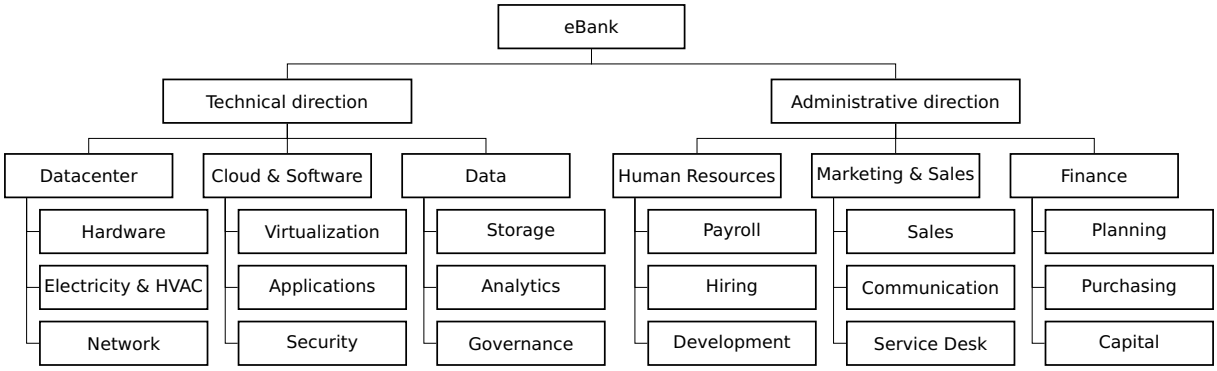


Figure VII.11: eBank's organizational structure

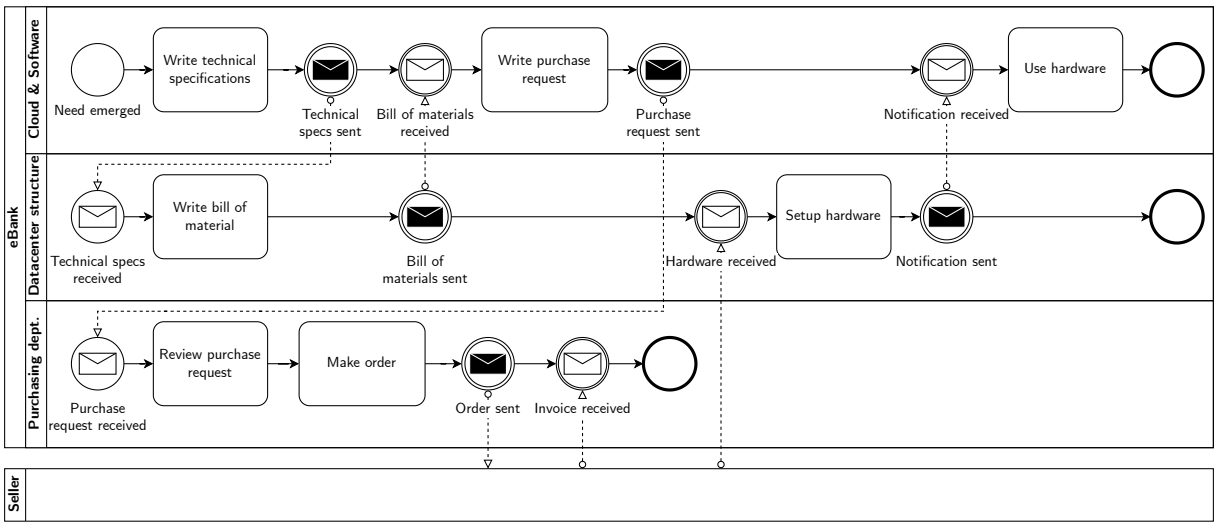


Figure VII.12: BPMN diagram for purchasing new hardware. For clarity, we do not show the exclusive gateways and assume requests to be automatically accepted.

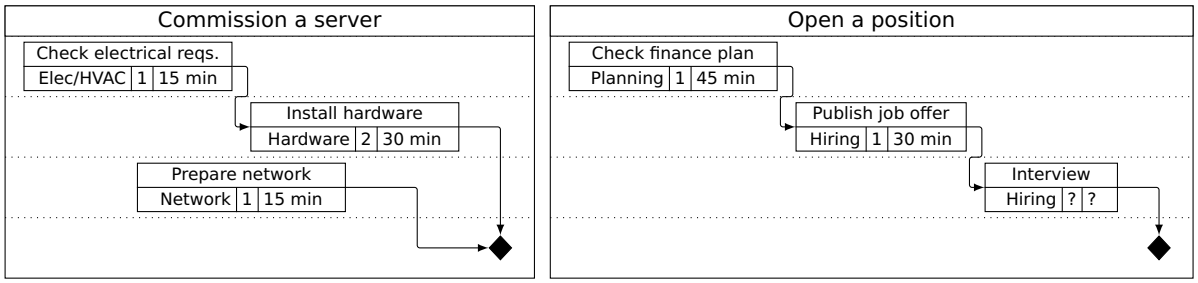


Figure VII.13: Task catalog
Legend. $\boxed{\begin{smallmatrix} a \\ b\,c\,d \end{smallmatrix}}$ Sub-task (a: title, b: department, c: people, d: duration), ◆ End

eBank has been using a task management solution for many years to track how many person-hours are needed for which projects. The solution is also used to know who is working on what at a given time. By reusing this software's database, employees created a catalog of common company tasks to predict their durations and help project planning. This catalog ranges from technical tasks, for example "commission a server", to administrative ones, for example "open a position". These two tasks are represented in figure VII.13.

Lastly, all financial transactions are managed by the finance department.

Technical architecture.

eBank manages a datacenter hosting the hardware necessary for its activities. Some services are hosted on dedicated machines and others are on an internal cloud infrastructure. Due to time constraints, ePay has not been migrated to a modern cloud infrastructure yet. The service follows an active/passive architecture, where only one node operates at a time.

To check the proper functioning of its services in real time, the company has a monitoring infrastructure that measures availability and several key performance indicators. For business clients, ePay must process its requests within three seconds 99.9% of the time and must pay penalties in case of non-compliance.

VII.5.2 ... linked together

After several rounds of modeling, eBank's employees came up with the representation shown in figure VII.14. First, the organizational structure (figure VII.11) is partially mapped to the eBank component and its four sub-components representing structures and departments. The BPMN diagram (figure VII.12) adds the Hardware component type, along with the Maintenance and Usage responsibilities that the Cloud & Software structure and Hardware department have on this Hardware. The task catalog (figure VII.13) adds knowledge about the Hiring department and its Hiring responsibility. The task management solution (Task manager) keeps track of the time spent on the Usage and Maintenance of the Hardware and on the Hiring process, highlighting the particular nature of responsibilities in our metamodel. Finally, the Finance department is responsible for employees' Wage payment, for the Invoice payment of Hardware and the company's Financial obligation regarding its Payment processing's SLAs.

VII.5.3 Exploiting the model

The company's real time monitoring has recently identified slowdowns in ePay on peak hours. Following our model, we can see that there is a potential impact on the Finance department because of its Financial obligations. Some employees have suggested scaling the infrastructure before such slowdowns violate SLAs. One way to do so is to setup new Physical servers and change the overall architecture of the services. This new architecture is expected to mobilize part of the Cloud & Software structure for several months. The Human Resources structure proposes to either ignore the potential problem or to assign its teams on the scaling project (by hiring new staff, outsourcing some of the work or reassigning staff without additional hiring or outsourcing).

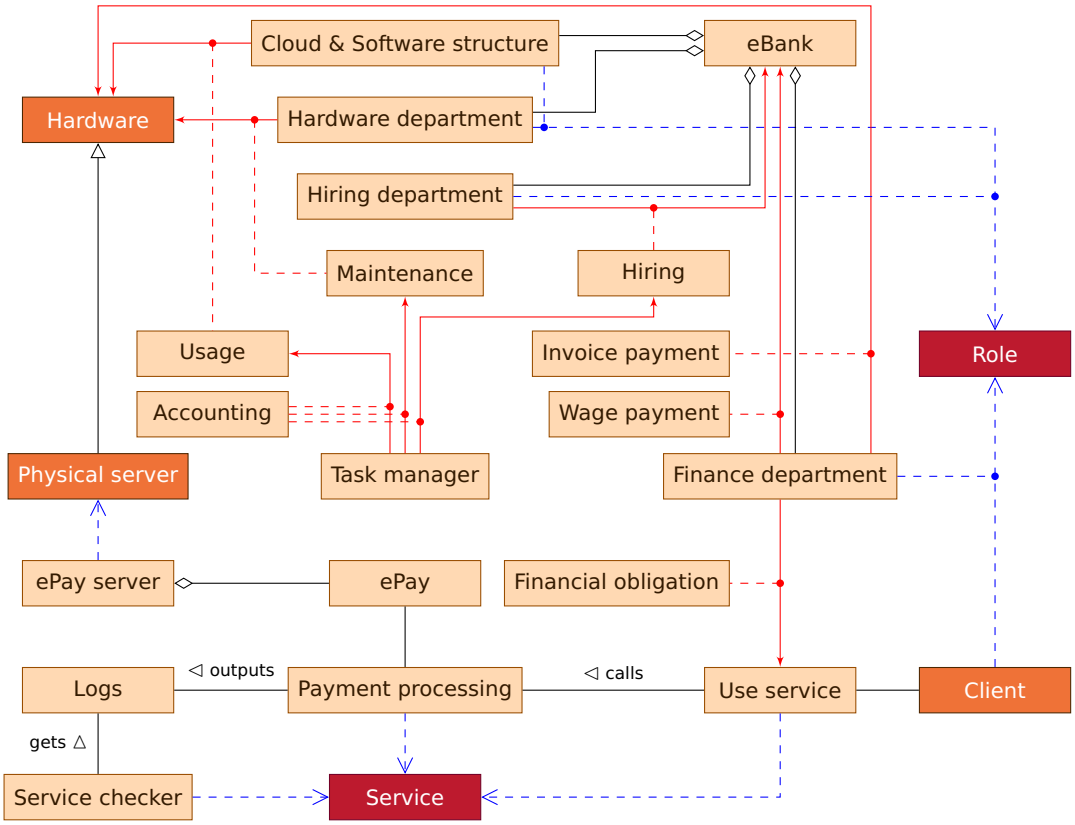


Figure VII.14: ePay's big picture. To help the reader, instantiations are blue (□-≫□) and responsibility links and their association classes are red (□→□ and □- -□).

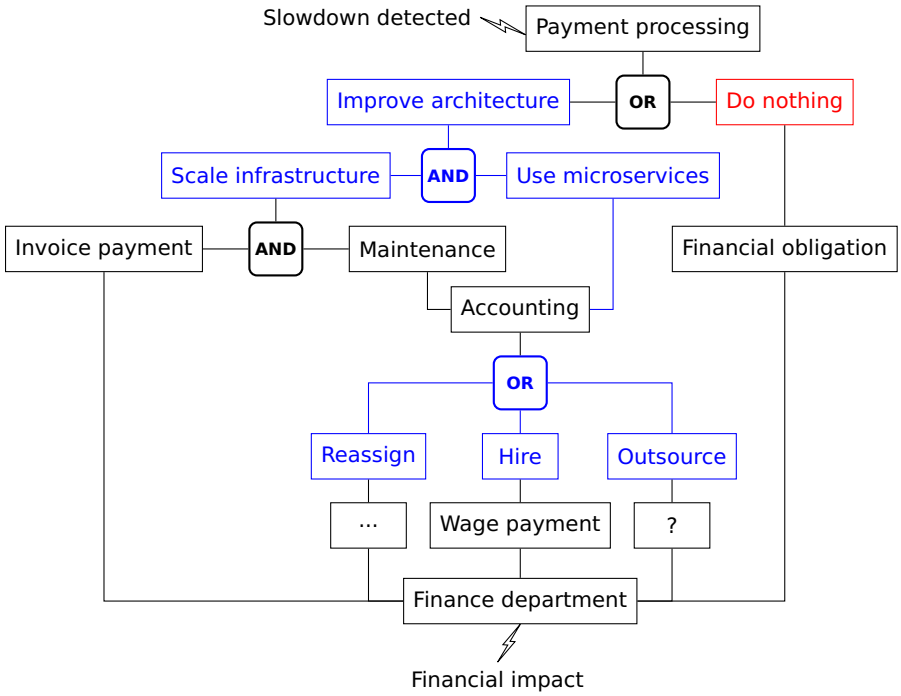


Figure VII.15: Impact tree from application slowness to financial impact
Legend. ☐ Default branch, ☐ Employee suggestions, ☐ Model elements

In figure VII.15, we explore our model to trace the paths between the slowdown and the Finance department, to identify potential financial impacts. To make its decision, the company has to compare (a) the financial impact if nothing is done (resulting from the Financial obligation) to (b) the financial impact of the improved architecture (resulting from Invoice payments and the personnel cost). The company does not have an outsourcing process, so the analysis of the Outsource branch, represented by "?" in the figure, cannot be performed. We have not detailed the Reassign branch for the sake of simplicity; however, its analysis is valuable to the company since the reassignment of staff would change the time allocated in the Task manager. This would consequently have indirect impacts on the Finance department (for example, failure to deliver new features to clients due to a lack of time, leading to increased customer churn).

By calculating the cost of each decision branch, as discussed in section VI.1.2, the company can make a decision regarding this particular problem.

VII.6 Conclusion

In this chapter, we have presented an approach to IT infrastructure modeling which capitalizes on the lessons we have learned throughout our work. This approach consists of a generic metamodel aimed at linking models together and better controlling the responsibilities of each stakeholder, through principles inspired by software engineering to guide the modeling process. Our approach is not intended to substitute established methods within organizations, but rather to enable complex analyses spanning across multiple models. Through model federation, we think that business modeling can include more stakeholders, while preserving the tools and models they are used to working with.

In the course of this thesis, we did not have the opportunity to validate our approach on a large scale. Nevertheless, we hold the view that our method offers a well-organized framework for study, and its systematic application to smaller infrastructures has enabled us to develop simple and accurate cross-domain models. These models were however made by a single person, so future work needs to focus on the collaboration between a variety of teams, which we did not have the opportunity to explore further.

Chapter VIII

Conclusion

Contents

VIII.1 Synthesis of contributions 121
VIII.2 Limitations and perspectives 123
VIII.2.1 Risk management 123
VIII.2.2 Modeling 124
VIII.2.3 CL/I 124
VIII.2.4 Enterprise integration 124

IT infrastructures are subject to a wide range of safety and security risks. A number of methods are available to help understand these risks and establish strategies to avoid them, or reduce their effects. But these methods are still largely manual, and therefore subject to human error. This human error is often caused by a lack of knowledge of infrastructures, or a misunderstanding of the links between their many components. Infrastructure modeling helps people build a better understanding and can, with the help of formal methods, be used to deduce safety and security properties that serve as guidance for risk analysis. It seems to us however that the techniques developed by the scientific and technical communities dealing with these subjects (risk analysis, formal methods, enterprise modeling) lack interoperability, and that the links between these communities need to be strenghened.

VIII.1 Synthesis of contributions

The work presented in this dissertation revolves around the concept of "risk cycle". We have opted for a wide approach, in order to explore the various gaps in the literature throughout this process, which is synthesized in figure VIII.1 where we show the contributions from each chapter to the cycle. The figure reads as follows: infrastructures are modeled, and checking them gives insights about the risk. The risk that is deemed acceptable is filtered, and the remaining risk leads to requirements for the infrastructure. These requirements are implemented as constraints, some of which are defined through the configuration of infrastructure components, which can lead to executions meeting the requirements or not on the reified infrastructures. Then, the cycle can start again from these reified infrastructures.

In chapter IV, we have introduced our formalism around the risk cycle. We explored the risk management field and focused on two main aspects: risk classification and risk analysis frameworks. For risk classification, we have outlined a number of taxonomy initiatives, highlighting the poor

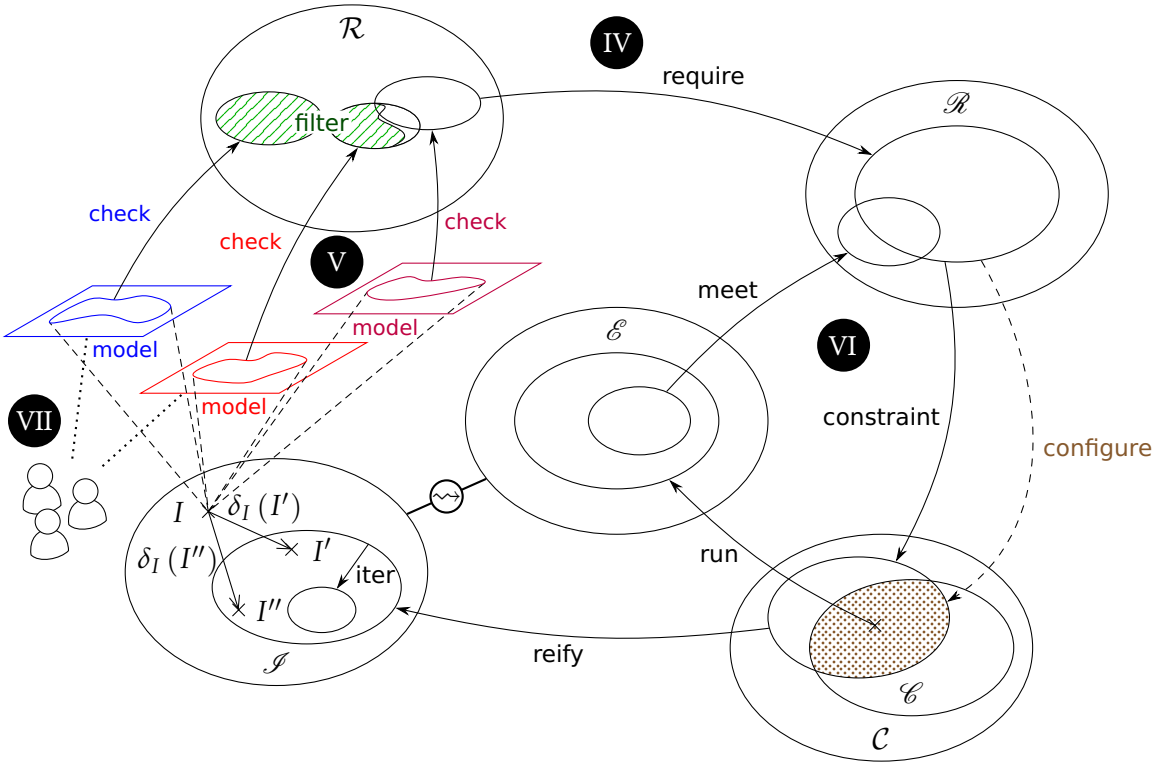


Figure VIII.1: Refinement of the risk cycle throughout this dissertation
Legend. (x) Presented in chapter x, (~) Execution of infrastructures.

representation of safety risk. After taking a particular interest in the MITRE reference datasets, we developed a tool, `mitre2owl`, to translate the CAPEC, CVE and CWE projects into rich, interoperable ontologies. We have shown how to exploit and combine our ontologies and produced a *Vulnerability–Weakness rosette* to visually show the prevalence of CWEs in CVEs. For risk analysis frameworks, we have drawn lessons from our experience of using traditional analysis methods on IT infrastructures and compiled them on a set of guidelines on how to adapt existing methods to such infrastructures. Since business activities based on IT infrastructures inherently present their own share of risk, we have discussed criteria for risk acceptability. We closed the chapter with the hope that different manufacturers and developers would eventually share their risk analyses, and provided a simple framework for composing and serializing said analyses.

In chapter V, we have presented infrastructure modeling through technical tools and formal methods and how to translate models from the technical to the formal one. We have produced concrete, realistic models for a small IT infrastructure through physical, network, and behavioral aspects, and have shown how to link them together. We have then proceeded with the verification of these models using UPPAAL. Using a trial-and-error approach, we drew lessons on how to develop more complete, usable and verifiable infrastructure models. We concluded the chapter by presenting some ideas for automating the process, by translating high-level requirements into logical formulae and linking these formulae to shared model catalogs.

In chapter VI, we have introduced what represents the largest part of our work: the CL/I modeling language. The goal of CL/I is to provide a pivot language for IT infrastructure modeling, allowing automatic verification and execution of models by various tools. We have introduced the grammar, AST, and various semantic processing rules of CL/I alongside examples showing how our language

works. We have then presented two case studies, the former exploring the structure and dynamics of the Proxmox VE hypervisor and the latter building and verifying a model covering several business domains. To this end, we introduced one of the language's extensions enabling it to be linked to the Z3 prover, and showed the presence of unsafe behavior in Proxmox VE. Through our first case study, we also contributed to the Emulab community by developing a profile to deploy Proxmox clusters on Emulab servers.

Finally, in chapter VII, we met with the Enterprise Modeling community to explore the integration of our work in a corporate environment. We have first proposed a component- and responsibility-oriented metamodel to link enterprise models together through a common framework. Its main focus is the federation of the different viewpoints that coexist between stakeholders from various corporate domains. We have then presented several guidelines for a more collaborative practice of infrastructure modeling before discussing important points to consider when scaling up the approach, echoing the ideas developed earlier in the dissertation.

The period during which we have developed our line of research has enabled us to contribute to different fields with the contributions we have presented here. However, our project aims for the long term.

VIII.2 Limitations and perspectives

The avenues for further improvement in our work are numerous, and over the course of our doctoral program, we had to narrow the focus of our research. In this section, we present the limitations of our work and some short-, intermediate- and long-term perspectives.

VIII.2.1 Risk management

We have presented a number of risk analysis frameworks in this dissertation, each of which can be used to feed a risk taxonomy. However, we have focused on a single family of reference documents, those of the MITRE, to develop our ontologies. This choice is mainly due to the technical limitations of other standards, which are often presented in an unstructured (in the data processing sense) textual form. In the short term, standards such as that of ENISA could be ported to ontologies, and we could develop semantic rules to federate them. In the long term, purely textual standards could also be transformed into ontologies, but this would be a task for scientific communities interested in topics such as Natural Language Processing. Because frameworks often refer to one another (for example, many CWEs refer to NIST and OWASP entries), such a federation has already been partly done, although informally.

When we discussed risk analysis frameworks, we put forward the idea of an exchange format for IT infrastructure analyses. We carried out in our work a short analysis on physical servers, using the FMEA, FTA and STPA frameworks, combining *in situ* testing and translations of manufacturers' user manuals, and have produced a unique representation of the risk on several Dell and HPE servers. We plan to conduct in the short term further studies in this direction and publish our results to promote a more open approach to the discipline.

VIII.2.2 Modeling

We originally come from a software engineering community. One of the foundations of the field is what we like to call the "lazy principle":

1. Do not redo what already exists;
2. Do not overdo;
3. Do not overcomplicate.

We have suggested in this dissertation to apply these points to modeling, by 1) encouraging the development of generic, reusable models, 2) modeling only the necessary structure and behavior, and 3) modeling it in a sufficiently abstract way. Actually, the last two points are more of a constraint of model checking than a deliberate choice, so we think that our work would benefit most from more development in the first point. As with risk analyses, we plan to work, in the intermediate term, on producing a public catalog of instantiable models. Such models could be combined with our open risk analyses to help automatic certification of infrastructures.

Models are made to represent aspects of concrete infrastructure elements, but they are too often static, and for a good reason: updating models can sometimes be costly. An acceptable in-between is the automatic generation of infrastructure tests from their models, to ensure that safety and security invariants are not violated over time. In the intermediate term, we plan to develop and integrate in our CL/I ecosystem backends for software such as Nagios or Ansible, enabling real-time verification and automatic reconfiguration of systems.

VIII.2.3 CL/I

Our language and its infrastructure are still under development, and there are a number of issues that need to be addressed. Firstly, although our language is typed, its type engine is still too primitive (no unification, subtyping not supported by our Z3 extension). We consider it important to work on this point to make the language more reliable.

Secondly, some of our AST constructs do not yet have a semantic rule so they can be transformed into the language's intermediate representation, CLIR. We need to work on completing our semantic rules, and then ensure that the new CLIR constructs are properly translated into Z3.

Finally, Z3 is able to generate a set of complete models when a (possibly partial) model satisfies properties, and counterexamples in the opposite case or if the model is not consistent. While we are able to switch from CL/I to CLIR, and from the CLIR to Z3, inverse transformations are not implemented yet, and informations such as the original source code locations are not currently recorded. This is an important consideration for improving the explicability of Z3 diagnostics in our language.

VIII.2.4 Enterprise integration

Lastly, our approach lacks validation in an industrial context, and the various guidelines we have proposed in this dissertation need to applied to various teams within a company. As with any large-scale experiment, we do not expect this goal to be achieved in the short term. This approach goes hand in hand with the other future developments we have proposed, as experimentation in a corporate environment will put into practice the model sharing principles we have advocated. Finally, the systematic integration of our approach into business processes needs to be explored, as an intermediate-term objective.

Bibliography

[Abbass15] Wissam Abbass, Amine Baina, and Mostafa Bellafkih. "Using EBIOS for risk management in critical information infrastructure". In: *5th World Congress on Information and Communication Technologies*. 2015, pp. 107–112. DOI: 10.1109/WICT.2015.7489654 (cit. on p. 13).

[Abrial96] Jean-Raymond Abrial. *The B-book — Assigning Programs to Meanings*. Cambridge university press, 1996 (cit. on p. 62).

[Agha18] Gul Agha and Karl Palmskog. "A Survey of Statistical Model Checking". In: *ACM Transactions on Modeling and Computer Simulation* 28.1 (Jan. 2018). ISSN: 1049-3301. DOI: 10.1145/3158668 (cit. on p. 16).

[Alturkistani14] Fatimah Alturkistani and Ahmed Emam. "A Review of Security Risk Assessment Methods in Cloud Computing". In: *New Perspectives in Information Systems and Technologies*. Vol. 1. Springer International Publishing, 2014. ISBN: 9783319059518. DOI: 10.1007/978-3-319-05951-8_42 (cit. on p. 47).

[Amaral10] Vasco Amaral, Cécile Hardebolle, Gabor Karsai, László Lengyel, and Tihamér Levendovszky. "Recent Advances in Multi-paradigm Modeling". In: *Models in Software Engineering*. 2010, pp. 220–224. ISBN: 978-3-642-12261-3 (cit. on p. 110).

[Amundrud17] Øystein Amundrud, Terje Aven, and Roger Flage. "How the definition of security risk can be made compatible with safety definitions". In: *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* 231.3 (2017), pp. 286–294. DOI: 10.1177/1748006X17699145 (cit. on p. 10).

[Anderson14] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. "NetKAT: Semantic Foundations for Networks". In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 113–126. ISSN: 0362-1340. DOI: 10.1145/2578855.2535862 (cit. on p. 14).

[ANSSI DAT-NT-028] Agence Nationale de la Sécurité des Systèmes d'Information. *Recommandations de configuration d'un système GNU/Linux*. 2022. URL: https://www.ssi.gouv.fr/reco-securite-systeme-linux (cit. on p. 36).

[Archimate] The Open Group. *ArchiMate® 3.1 Specification*. URL: https://publications.opengroup.org/c197 (cit. on pp. 16, 107).

[ARP4761] SAE International. *ARP4761 — Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, 1996. URL: https://www.sae.org/standards/content/arp4761/ (cit. on pp. 13, 44).

[Artac17] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. "DevOps: Introducing Infrastructure-as-Code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. May 2017, pp. 497–498. DOI: 10.1109/ICSE-C.2017.162 (cit. on p. 80).

[AS9100D] SAE International. *Quality Management Systems – Requirements for Aviation, Space, and Defense Organizations*. SAE International, 2016. URL: https://www.sae.org/standards/content/as9100d/ (cit. on p. 13).

[Autili15] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. "Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar". In: *IEEE Transactions on Software Engineering* 41.7 (2015), pp. 620–638. DOI: 10.1109/TSE.2015.2398877 (cit. on p. 75).

[Baier08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. Apr. 2008. ISBN: 978-0-262-02649-9 (cit. on pp. 16, 17).

[Barrett17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017 (cit. on p. 91).

[Baybutt15] Paul Baybutt. "A critique of the Hazard and Operability (HAZOP) study". In: *Journal of Loss Prevention in the Process Industries* 33 (2015). ISSN: 0950-4230. DOI: 10.1016/j.jlp.2014.11.010 (cit. on p. 45).

[Bleikertz15] Sören Bleikertz, Carsten Vogel, Thomas Groß, and Sebastian Mödersheim. "Proactive Security Analysis of Changes in Virtualized Infrastructures". In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACSAC '15. Association for Computing Machinery, 2015, pp. 51–60. ISBN: 9781450336826. DOI: 10.1145/2818000.2818034 (cit. on pp. 20, 81).

[Blythe15] John Matthew Blythe, Lynne Coventry, and Linda Little. "Unpacking security policy compliance: The motivators and barriers of employees' security behaviors". In: *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*. 2015, pp. 103–122 (cit. on p. 50).

[Boustras20] Georgios Boustras and Alan Waring. "Towards a reconceptualization of safety and security, their interactions, and policy requirements in a 21st century context". In: *Safety Science* 132 (2020), p. 104942. ISSN: 0925-7535. DOI: https://doi.org/10.1016/j.ssci.2020.104942 (cit. on p. 10).

[Breton00] Erwan Breton and Jean Bézivin. "An Overview of Industrial Process Meta-Models". In: *International Conference on Software & Systems Engineering and their Applications*. 2000 (cit. on p. 108).

[Brikman19] Yevgeniy Brikman. *Terraform: Up & Running*. 2nd. O'Reilly Media, Inc., 2019. ISBN: 9781492046905 (cit. on p. 19).

[Buldyrev10] Sergey Buldyrev, Roni Parshani, Gerald Paul, Harry Eugene Stanley, and Shlomo Havlin. "Catastrophic cascade of failures in interdependent networks". In: *Nature* 464.7291 (Apr. 2010), pp. 1025–1028. ISSN: 1476-4687. DOI: 10.1038/nature08932 (cit. on p. 12).

[Burns16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade". In: *Queue* 14.1 (Jan. 2016), pp. 70–93. ISSN: 1542-7730. DOI: 10.1145/2898442.2898444 (cit. on p. 20).

[Burns92] Alan Burns, John McDermid, and John Dobson. "On the Meaning of Safety and Security". In: *The Computer Journal* 35.1 (Feb. 1992), pp. 3–15. ISSN: 0010-4620. DOI: 10.1093/comjnl/35.1.3 (cit. on p. 9).

[Campbell20] Bradley Campbell. "The AWS CDK and Pulumi". In: *The Definitive Guide to AWS Infrastructure Automation : Craft Infrastructure-as-Code Solutions*. Berkeley, CA: Apress, 2020, pp. 237–272. ISBN: 978-1-4842-5398-4. DOI: 10.1007/978-1-4842-5398-4_6 (cit. on p. 19).

[CAPEC] The MITRE Corporation. *Common Attack Pattern Enumeration and Classification*. URL: https://capec.mitre.org (cit. on p. 37).

[Caracciolo15] Andrea Caracciolo. "A Unified Approach to Automatic Testing of Architectural Constraints". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. 2015, pp. 871–874. DOI: 10.1109/ICSE.2015.281 (cit. on p. 17).

[Chardet18] Maverick Chardet, Helene Coullon, Dimitri Pertin, and Christian Perez. "Madeus: A Formal Deployment Model". In: *2018 International Conference on High Performance Computing & Simulation (HPCS)*. 2018, pp. 724–731. DOI: 10.1109/HPCS.2018.00118 (cit. on p. 20).

[CIM] Distributed Management Task Force. *Common Information Model*. URL: https://www.dmtf.org/standards/cim (cit. on p. 60).

[Clarke00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-45047-4. DOI: 10.1007/10722167_15 (cit. on p. 72).

[Clarke09] Edmund Clarke, Ernest Allen Emerson, and Joseph Sifakis. "Model Checking: Algorithmic Verification and Debugging". In: *Communications of the ACM* 52.11 (Nov. 2009), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/1592761.1592781 (cit. on p. 16).

[Clarke12] Edmund Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. "Model Checking and the State Explosion Problem". In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1 (cit. on p. 70).

[Cohn04] David Cohn and Markus Stolze. "The rise of the model-driven enterprise". In: *IEEE International Conference on E-Commerce Technology for Dynamic E-Business*. 2004, pp. 324–327. DOI: 10.1109/CEC-EAST.2004.65 (cit. on p. 107).

[CORAS] Mass Soldal Lund, Bjrnar Solhaug, and Ketil Stlen. *Model-Driven Risk Analysis: The CORAS Approach*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642123228 (cit. on p. 13).

[Corosync] Corosync developer community. *The Corosync Cluster Engine*. URL: http://corosync.github.io/corosync/ (cit. on p. 65).

[Cousot05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTREÉ Analyzer". In: *Programming Languages and Systems*. Ed. by Mooly Sagiv. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 21–30. ISBN: 978-3-540-31987-0. DOI: 10.1007/978-3-540-31987-0_3 (cit. on p. 41).

[CPE] National Institute of Standards and Technology. *Official Common Platform Enumeration (CPE) Dictionary*. URL: https://nvd.nist.gov/products/cpe (cit. on p. 76).

[CRAMM] Zeki Yazar. "A qualitative risk analysis and management tool – CRAMM". In: *SANS InfoSec Reading Room White Paper* 11.1 (2002), pp. 12–32 (cit. on p. 13).

[Cristea17] Gabriel Cristea and D. Constantinescu. "A comparative critical study between FMEA and FTA risk analysis methods". In: *IOP Conference Series: Materials Science and Engineering* 252 (2017). DOI: 10.1088/1757-899x/252/1/012046 (cit. on p. 45).

[CVE] The MITRE Corporation. *Common Vulnerability Enumeration*. URL: https://cve.org (cit. on p. 37).

[CWE] The MITRE Corporation. *Common Weakness Enumeration*. URL: https://cwe.mitre.org (cit. on p. 37).

[daCunhaR16] Guilherme da Cunha Rodrigues, Rodrigo Neves Calheiros, Vinicius Tavares Guimaraes, Glederson Lessa dos Santos, Márcio Barbosa de Carvalho, Lisandro Zambenedetti Granville, Liane Margarida Rockenbach Tarouco, and Rajkumar Buyya. "Monitoring of Cloud Computing Environments: Concepts, Solutions, Trends, and Future Directions". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: Association for Computing Machinery, 2016, pp. 378–383. ISBN: 9781450337397. DOI: 10.1145/2851613.2851619 (cit. on p. 20).

[DeDomenico13] Manlio De Domenico, Albert Solé-Ribalta, Emanuele Cozzo, Mikko Kivelä, Yamir Moreno, Mason A. Porter, Sergio Gómez, and Alex Arenas. "Mathematical Formulation of Multilayer Networks". In: *Physical Review X* 3 (4 Dec. 2013). DOI: 10.1103/PhysRevX.3.041022 (cit. on p. 12).

[Dehnert17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. "A Storm is Coming: A Modern Probabilistic Model Checker". In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 592–600. ISBN: 978-3-319-63390-9. DOI: 10.1007/978-3-319-63390-9_31 (cit. on p. 72).

[deMoura08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2008, pp. 337–340. ISBN: 978-3-540-78800-3 (cit. on p. 16).

[deSilva12] Lakshitha de Silva and Dharini Balasubramaniam. "Controlling software architecture erosion: A survey". In: *Journal of Systems and Software* 85.1 (2012). Dynamic Analysis and Testing of Embedded Software, pp. 132–151. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2011.07.036 (cit. on p. 82).

[Deursen00] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages: An Annotated Bibliography". In: *SIGPLAN Notices* (2000). ISSN: 0362-1340. DOI: 10.1145/352029.352035 (cit. on p. 108).

[Dhaussy12] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. "Improving Model Checking with Context Modelling". In: *Advances in Software Engineering* 2012 (Jan. 2012). ISSN: 1687-8655. DOI: 10.1155/2012/547157. URL: https://doi.org/10.1155/2012/547157 (cit. on p. 16).

[DiCosmo14] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. "Aeolus: A component model for the cloud". In: *Information and Computation* 239 (2014), pp. 100–121. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2014.11.002 (cit. on p. 20).

[Dimitrov23] Vladimir Dimitrov. *CWE Ontology*. Feb. 2023. DOI: 10.55630/sjc.2022.16.39-56 (cit. on p. 41).

[Dwyer99] Matthew Dwyer, George Avrunin, and James Corbett. "Patterns in property specifications for finite-state verification". In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 1999, pp. 411–420. DOI: 10.1145/302405.302672 (cit. on p. 75).

[EBIOS-RM] Agence Nationale de la Sécurité des Systèmes d'Information. *EBIOS Risk Manager*. 2019. URL: https://www.ssi.gouv.fr/uploads/2019/11/anssi-guide-ebios_risk_manager-en-v1.0.pdf (cit. on pp. 13, 27, 46).

[Esen22] Muhammed Fevzi Esen. "Business Continuity in Data Centers and Seismic Isolation Applications". In: *J. Inf. Technol. Res.* 15 (2022), pp. 1–23. DOI: 10.4018/JITR.299928 (cit. on p. 12).

[Eurocode 8] European Commission. *Eurocode 8: Design of structures for earthquake resistance*. 1998. URL: https://eurocodes.jrc.ec.europa.eu/EN-Eurocodes/eurocode-8-design-structures-earthquake-resistance?id=138 (cit. on p. 28).

[Feltus11] Christophe Feltus, Michaël Petit, and Eric Dubois. "ReMoLa: Responsibility model language to align access rights with business process requirements". In: *International Conference on Research Challenges in Information Science*. 2011, pp. 1–6. DOI: 10.1109/RCIS.2011.6006828 (cit. on p. 105).

[Fleurey08] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. "A Generic Approach for Automatic Model Composition". In: *Models in Software Engineering*. 2008, pp. 7–15. ISBN: 978-3-540-69073-3 (cit. on p. 108).

[Floyd62] Robert W Floyd. "Algorithm 97: Shortest path". In: *Commun. ACM* 5.6 (June 1962), p. 345. ISSN: 0001-0782. DOI: 10.1145/367766.368168 (cit. on p. 69).

[GDPR] European Parliament and Council of the European Union. *General Data Protection Regulation*. 2016. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679 (cit. on pp. viii, 11).

[Giamattei24] Luca Giamattei, Antonio Guerriero, Roberto Pietrantuono, Stefano Russo, Stefano Malavolta, Tanjina Islam, Madalina Dînga, Anne Koziolek, Snigdha Singh, Martin Armbruster, José-María Gutierrez-Martinez, Sergio Caro-Alvaro, Daniel Rodriguez, Sebastian Weber, Jörg Henss, Estrella Fernandez Vogelin, and Fernando Simon Panojo. "Monitoring tools for DevOps and microservices: A systematic grey literature review". In: *Journal of Systems and Software* 208 (2024). ISSN: 0164-1212. DOI: 10.1016/j.jss.2023.111906 (cit. on p. 113).

[Golra16] Fahad Rafique Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, and Christophe Guychard. "Addressing Modularity for Heterogeneous Multi-model Systems using Model Federation". In: *Companion Proceedings of the International Conference on Modularity (MoMo)*. ACM, 2016. ISBN: 978-1-4503-4033-5. DOI: 10.1145/2892664.2892701 (cit. on pp. 16, 108).

[GR-63-CORE] Ericsson. *NEBS Requirements: Physical Protection*. Ericsson, 2017. URL: https://telecom-info.njdepot.ericsson.net/site-cgi/ido/docs.cgi?ID=SEARCH&DOCUMENT=GR-63 (cit. on pp. viii, 28).

[Graham19] John Graham-Cumming. *Details of the Cloudflare outage on July 2, 2019*. 2019. URL: https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/ (cit. on p. 80).

[Grati15] Rima Grati, Khouloud Boukadi, and Hanêne Ben-Abdallah. "Overview of IaaS monitoring tools". In: *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*. 2015, pp. 1–7. DOI: 10.1109/AICCSA.2015.7507146 (cit. on p. 20).

[Greenwood95] R. Mark Greenwood, Ian Robertson, Robert Archer Snowdon, and Brian Warboys. "Active Models in Business". In: *Annual Conference on Business Information Technology (BIT)*. 1995 (cit. on p. 108).

[Grottke08] Michael Grottke, Rivalino Matias, and Kishor S. Trivedi. "The fundamentals of software aging". In: *2008 IEEE International Conference on Software Reliability Engineering Workshops (ISSRE Workshop)*. 2008, pp. 1–6. DOI: 10.1109/ISSREW.2008.5355512 (cit. on p. 20).

[Grunske08] Lars Grunske. "Specification Patterns for Probabilistic Quality Properties". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 31–40. ISBN: 9781605580791. DOI: 10.1145/1368088.1368094. URL: https://doi.org/10.1145/1368088.1368094 (cit. on p. 75).

[Hannousse21] Abdelhakim Hannousse and Salima Yahiouche. "Securing microservices and microservice architectures: A systematic mapping study". In: *Computer Science Review* 41 (2021), p. 100415. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2021.100415 (cit. on p. 12).

[Hasan15] Samiul Hasan and Greg Foliente. "Modeling infrastructure system interdependencies and socioeconomic impacts of failure in extreme events: emerging R&D challenges". In: *Natural Hazards* 78.3 (Sept. 2015), pp. 2143–2168. ISSN: 1573-0840. DOI: 10.1007/s11069-015-1814-7 (cit. on p. 13).

[He18] Jinyuan He and Le Sun. "A Review on SLA-Related Applications in Cloud Computing". In: *2018 1st International Cognitive Cities Conference (IC3)*. 2018. DOI: 10.1109/IC3.2018.00027 (cit. on p. 11).

[Hochstein17] Lorin Hochstein and Rene Moser. *Ansible: Up & Running*. 2nd. O'Reilly Media, Inc., 2017. ISBN: 1491979801 (cit. on p. 19).

[Holzmann97] Gerard Holzmann. "The model checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521 (cit. on p. 16).

[Hughes10] John Hughes. "Software Testing with QuickCheck". In: *Central European Functional Programming School*. Ed. by Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 183–223. ISBN: 978-3-642-17685-2. DOI: 10.1007/978-3-642-17685-2_6 (cit. on p. 17).

[Hüttermann12] Michael Hüttermann. "Infrastructure as Code". In: *DevOps for Developers*. Berkeley, CA: Apress, 2012, pp. 135–156. ISBN: 978-1-4302-4570-4. DOI: 10.1007/978-1-4302-4570-4_9 (cit. on p. 80).

[IEC 60050] International Electrotechnical Commission. *International Electrotechnical Vocabulary*. 2023. URL: https://www.electropedia.org (cit. on p. 10).

[IEC 61882] International Electrotechnical Commission. *Hazard and operability studies (HAZOP studies) — Application guide* (cit. on pp. 13, 45).

[ISO/IEC/IEEE 15288] International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers. *Systems and software engineering — System life cycle processes*. 2023. DOI: 10.1109/IEEESTD.2023.10123367 (cit. on p. 107).

[ISO/IEC 27005] International Organization for Standardization. *Information security, cybersecurity and privacy protection — Guidance on managing information security risks*. 2022. URL: https://www.iso.org/standard/80585.html (cit. on pp. 13, 46).

[ISO/SAE 21434] International Organization for Standardization. *Road vehicles — Cybersecurity engineering*. 2021. URL: https://www.iso.org/standard/70918.html (cit. on p. 51).

[ISO 14258] International Organization for Standardization. *Industrial automation systems and integration — Concepts and rules for enterprise models*. 1998. URL: https://www.iso.org/standard/24020.html (cit. on pp. 108, 113).

[ISO 19439] International Organization for Standardization. *Enterprise integration — Framework for enterprise modelling*. 2006. URL: https://www.iso.org/standard/33833.html (cit. on p. 107).

[ISO 26262] International Organization for Standardization. *Road vehicles – Functional safety*. 2018. URL: https://www.iso.org/standard/68383.html (cit. on p. 13).

[ISO 31000] International Organization for Standardization. *Risk management — Guidelines*. 2018. URL: https://www.iso.org/standard/65694.html (cit. on p. 26).

[ITIL19] ITIL Foundation. *Glossary terms and definitions*. Jan. 2019 (cit. on p. 2).

[Jackson11] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. Second. The MIT Press, 2011. ISBN: 9780262300254 (cit. on p. 62).

[Jones17] Nora Jones. "Performing Chaos at Netflix Scale". AWS re:Invent 2017. 2017. URL: https://www.youtube.com/watch?v=LaKGx0dAUlo (cit. on p. 21).

[Jørgensen09] Håvard D. Jørgensen. "Enterprise Modeling – What We Have Learned, and What We Have Not". In: *The Practice of Enterprise Modeling*. 2009, pp. 3–7. ISBN: 978-3-642-05352-8 (cit. on p. 107).

[Kaczmarek15] Monika Kaczmarek. "Ontologies in the Realm of Enterprise Modeling – A Reality Check". In: *Formal Ontologies Meet Industry*. 2015, pp. 39–50 (cit. on p. 108).

[Khurshid13] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. "VeriFlow: Verifying Network-Wide Invariants in Real Time". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 15–27. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid (cit. on p. 20).

[Kirchner15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Aspects of Computing* 27.3 (May 2015), pp. 573–609. ISSN: 0934-5043. DOI: 10.1007/s00165-014-0326-7 (cit. on p. 17).

[Konrad05] Sascha Konrad and Betty Cheng. "Real-time specification patterns". In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 2005, pp. 372–381. DOI: 10.1109/ICSE.2005.1553580 (cit. on p. 75).

[Kotsovinos10] Evangelos Kotsovinos. "Virtualization: Blessing or Curse? Managing Virtualization at a Large Scale is Fraught with Hidden Challenges." In: *Queue* 8.11 (Nov. 2010), pp. 40–46. ISSN: 1542-7730. DOI: 10.1145/1874534.1889916 (cit. on p. 95).

[Kumara21] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. "The do's and don'ts of infrastructure code: A systematic gray literature review". In: *Information and Software Technology* 137 (2021), p. 106593. ISSN: 0950-5849. DOI: j.infsof.2021.106593 (cit. on p. 19).

[Lantow14] Birger Lantow. "On the Heterogeneity of Enterprise Models: ArchiMate and Troux Semantics". In: *IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*. 2014, pp. 67–71. DOI: 10.1109/EDOCW.2014.18 (cit. on p. 107).

[Larsen97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. "Uppaal in a nutshell". In: *International Journal on Software Tools for Technology Transfer* 1.1 (Dec. 1997), pp. 134–152. ISSN: 1433-2779. DOI: 10.1007/s100090050010 (cit. on p. 16).

[Lecomte17] Thierry Lecomte, David Deharbe, Etienne Prun, and Erwan Mottin. "Applying a Formal Method in Industry: A 25-Year Trajectory". In: *Formal Methods: Foundations and Applications*. Ed. by Simone Cavalheiro and José Fiadeiro. Cham: Springer International Publishing, 2017, pp. 70–87. ISBN: 978-3-319-70848-5. DOI: 10.1007/978-3-319-70848-5_6 (cit. on p. 17).

[Leroy09] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814 (cit. on p. 17).

[Leveson12] Nancy Leveson, Cody Harrison Fleming, Melissa Spencer, John Thomas, and Chris Wilkinson. "Safety Assessment of Complex, Software-Intensive Systems". In: *SAE International Journal of Aerospace* 5.1 (2012). ISSN: 1946-3855. DOI: 10.4271/2012-01-2134 (cit. on pp. 13, 46).

[Leveson19] Nancy Leveson. "A Systems Approach to Safety and Cybersecurity". In: Dublin: USENIX Association, Oct. 2019 (cit. on p. 14).

[Li02] Lei Li, K. Vaidyanathan, and K.S. Trivedi. "An approach for estimation of software aging in a Web server". In: *Proceedings International Symposium on Empirical Software Engineering*. 2002, pp. 91–100. DOI: 10.1109/ISESE.2002.1166929 (cit. on p. 20).

[Liu09] Simon Liu, Rick Kuhn, and Hart Rossman. "Surviving Insecure IT: Effective Patch Management". In: *IT Professional* 11.2 (2009), pp. 49–51. DOI: 10.1109/MITP.2009.38 (cit. on p. 82).

[Lv18] Junjie Lv and Juling Rong. "Virtualisation security risk assessment for enterprise cloud services based on stochastic game nets model". In: *IET Information Security* 12.1 (2018). DOI: 10.1049/iet-ifs.2017.0038 (cit. on p. 18).

[Maniah22] Maniah, Benfano Soewito, Ford Lumban Gaol, and Edi Abdurachman. "A systematic literature Review: Risk analysis in cloud migration". In: *Journal of King Saud University - Computer and Information Sciences* 34.6, Part B (2022), pp. 3111–3120. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2021.01.008 (cit. on p. 11).

[Masoudi16] Rahim Masoudi and Ali Ghaffari. "Software defined networks: A survey". In: *Journal of Network and Computer Applications* (2016). ISSN: 1084-8045. DOI: 10.1016/j.jnca.2016.03.016 (cit. on pp. 14, 61).

[Mastercard21] Mastercard. *Transaction Processing Rules*. 2021. URL: https://mastercard.us/content/dam/public/mastercardcom/na/global-site/documents/transaction-processing-rules.pdf (cit. on p. 11).

[MEHARI] Club de la sécurité de l'information français. *MEHARI 2010 — Risk analysis and treatment Guide*. 2019. URL: https://clusif.fr/wp-content/uploads/2015/10/mehari-2010-risk-analysis-and-treatment-guide.pdf (cit. on p. 13).

[Merabti11] Madjid Merabti, Michael Kennedy, and William Hurst. "Critical infrastructure protection: A 21st century challenge". In: *2011 International Conference on Communications and Information Technology (ICCIT)*. 2011. DOI: 10.1109/ICCITECHNOL.2011.5762681 (cit. on p. 18).

[Metzler19] Patrick Metzler, Neeraj Suri, and Georg Weissenbacher. "Extracting Safe Thread Schedules from Incomplete Model Checking Results". In: *Model Checking Software*. Ed. by Fabrizio Biondi, Thomas Given-Wilson, and Axel Legay. Cham: Springer International Publishing, 2019, pp. 153–171. ISBN: 978-3-030-30923-7. DOI: 10.1007/978-3-030-30923-7_9 (cit. on p. 72).

[Monniaux16] David Monniaux. "A Survey of Satisfiability Modulo Theory". In: *Computer Algebra in Scientific Computing*. Ed. by Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov. Cham: Springer International Publishing, 2016, pp. 401–425. ISBN: 978-3-319-45641-6. DOI: 10.1007/978-3-319-45641-6_26 (cit. on p. 16).

[Morris20] Kief Morris. *Infrastructure as code*. O'Reilly Media, 2020 (cit. on p. 61).

[Myrbakken17] Håvard Myrbakken and Ricardo Colomo-Palacios. "DevSecOps: A Multivocal Literature Review". In: *Software Process Improvement and Capability Determination*. Ed. by Antonia Mas, Antoni Mesquida, Rory V. O'Connor, Terry Rout, and Alec Dorling. Cham: Springer International Publishing, 2017, pp. 17–29. ISBN: 978-3-319-67383-7. DOI: 10.1007/978-3-319-67383-7_2 (cit. on p. 14).

[NetBox] NetBox Labs. *NetBox*. URL: https://github.com/netbox-community/netbox (cit. on p. 60).

[Netflix10] Gregory Orzell and Yury Izrailevsky. "Validating the Resiliency of Networked Applications". US 2012/0072571 A1. 2010 (cit. on p. 20).

[Neumann19] Peter Gabriel Neumann. "How Might We Increase System Trustworthiness?" In: *Communications of the ACM* 62.10 (Sept. 2019), pp. 23–25. ISSN: 0001-0782. DOI: 10.1145/3357225 (cit. on p. 14).

[Neville22] George Neville-Neil. "I Unplugged What?" In: *Communications of the ACM* 65.2 (2022). ISSN: 0001-0782. DOI: 10.1145/3506579 (cit. on pp. 14, 18).

[Newcombe14] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. "Use of Formal Methods at Amazon Web Services". In: (Sept. 2014). URL: https://research.microsoft.com/en-us/um/people/lamport/tla/formal-methods-amazon.pdf (cit. on p. 17).

[Nigmatullin23] Ildar Nigmatullin, Andrey Sadovykh, Sophie Ebersold, Nan Messe. "RQCODE: Security Requirements Formalization with Testing". In: *Testing Software and Systems*. Ed. by Silvia Bonfanti, Angelo Gargantini, and Paolo Salvaneschi. Cham: Springer Nature Switzerland, 2023, pp. 126–142. ISBN: 978-3-031-43240-8. DOI: 10.1007/978-3-031-43240-8_9 (cit. on p. 11).

[NIST SP 800-53] National Institute of Standards and Technology. *Security and Privacy Controls for Information Systems and Organizations*. URL: https://csrc.nist.gov/pubs/sp/800/53/r5/upd1/final (cit. on p. 33).

[Nuseibeh03] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. "ViewPoints: meaningful relationships are difficult!" In: *International Conference on Software Engineering*. 2003, pp. 676–681. DOI: 10.1109/ICSE.2003.1201254 (cit. on p. 111).

[OCTAVE] Richard Caralli, James Stevens, Lisa Young, and William Wilson. *Introducing OCTAVE Allegro: Improving the Information Security Risk Assessment Process*. Tech. rep. CMU/SEI-2007-TR-012. May 2007. DOI: 10.1184/R1/6574790.v1 (cit. on p. 13).

[Ohm20] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves. Cham: Springer International Publishing, 2020, pp. 23–43. ISBN: 978-3-030-52683-2. DOI: 10.1007/978-3-030-52683-2_2 (cit. on p. 12).

[Opdebeeck20] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. "Does Infrastructure as Code Adhere to Semantic Versioning? An Analysis of Ansible Role Evolution". In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2020, pp. 238–248. DOI: 10.1109/SCAM51674.2020.00032 (cit. on p. 20).

[openXSAM] openXSAM. *Towards a common Security Analysis Exchange Format for ISO/SAE 21434 and UN Reg. 155*. URL: https://www.openxsam.io/wp-content/uploads/2023/06/openXSAM-Towards-a-common-Security-Analysis-Exchange-Format.pdf (cit. on p. 51).

[OWASP ASVS] The OWASP Foundation. *OWASP Application Security Verification Standard*. 2021. URL: https://owasp.org/www-project-application-security-verification-standard/ (cit. on p. 33).

[Ozkaya23] Ipek Ozkaya. "Infrastructure as Code and Software Architecture Conformance Checking". In: *IEEE Software* 40.1 (2023), pp. 4–8. DOI: 10.1109/MS.2022.3213880 (cit. on p. 80).

[Pandya22] Sneh Pandya and Riya Guha Thakurta. "Introduction to Infrastructure as Code with Chef". In: *Introduction to Infrastructure as Code: A Brief Guide to the Future of DevOps*. Berkeley, CA: Apress, 2022, pp. 165–176. ISBN: 978-1-4842-8689-0. DOI: 10.1007/978-1-4842-8689-0_8 (cit. on p. 19).

[Park00] Kihong Park and Walter Willinger. "Self-similar network traffic: An overview". In: *Self-Similar Network Traffic and Performance Evaluation*. John Wiley & Sons, Ltd, 2000. Chap. 1, pp. 1–38. ISBN: 9780471206446. DOI: 10.1002/047120644X.ch1 (cit. on p. 14).

[Pavese16] Esteban Pavese, Víctor Braberman, and Sebastian Uchitel. "Less is More: Estimating Probabilistic Rewards over Partial System Explorations". In: *ACM Transactions on Software Engineering and Methodology* 25.2 (Apr. 2016). ISSN: 1049-331X. DOI: 10.1145/2890494 (cit. on p. 72).

[PCI DSS] Payment Card Industry Security Standards Council. *Payment Card Industry Data Security Standard*. Payment Card Industry Security Standards Council, 2022. URL: https://www.pcisecuritystandards.org/documents/PCI-DSS-v4_0.pdf (cit. on pp. viii, 11).

[Pietrantuono20] Roberto Pietrantuono and Stefano Russo. "A survey on software aging and rejuvenation in the cloud". In: *Software Quality Journal* 28.1 (Mar. 2020), pp. 7–38. ISSN: 1573-1367. DOI: 10.1007/s11219-019-09448-3 (cit. on p. 20).

[Proxmox VE] Proxmox Server Solutions GmbH. *Proxmox Virtual Environment*. URL: https://www.proxmox.com (cit. on pp. ix, 64).

[PSD2] European Commission. *Revised Payment Services Directive (PSD2, Directive (EU) 2015/2366)*. 2015. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A02015L2366-20151223 (cit. on p. 80).

[Qadir15] Junaid Qadir and Osman Hasan. "Applying Formal Methods to Networking: Theory, Techniques, and Applications". In: *IEEE Communications Surveys & Tutorials* 17.1 (2015), pp. 256–291. DOI: 10.1109/COMST.2014.2345792 (cit. on p. 70).

[Rahman21] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. "Security Smells in Ansible and Chef Scripts: A Replication Study". In: 30.1 (2021). ISSN: 1049-331X. DOI: 10.1145/3408897. URL: https://doi.org/10.1145/3408897 (cit. on p. 19).

[Renger08] Michiel Renger, Gwendolyn L. Kolfschoten, and Gert-Jan de Vreede. "Challenges in Collaborative Modeling: A Literature Review". In: *Advances in Enterprise Engineering I*. 2008, pp. 61–77. ISBN: 978-3-540-68644-6 (cit. on pp. 108, 110).

[Rinaldi01] Steven Rinaldi, James Peerenboom, and Terrence Kelly. "Identifying, understanding, and analyzing critical infrastructure interdependencies". In: *IEEE Control Systems Magazine* 21.6 (2001), pp. 11–25. DOI: 10.1109/37.969131 (cit. on p. 12).

[RM-ODP] *Reference Model of Open Distributed Processing (RM-ODP)*. International Organization for Standardization, International Electrotechnical Commission, International Telecommunication Union Telecommunication Standardization Sector. URL: http://rm-odp.net/ (cit. on pp. 16, 107).

[Rose03] Louis Rose. "Risk Management of COTS Based Systems Development". In: *Component-Based Software Quality: Methods and Techniques*. Ed. by Alejandra Cechich, Mario Piattini, and Antonio Vallecillo. Springer Berlin Heidelberg, 2003. ISBN: 978-3-540-45064-1. DOI: 10.1007/978-3-540-45064-1_16 (cit. on p. 47).

[Ruparelia10] Nayan Ruparelia. "Software Development Lifecycle Models". In: *SIGSOFT Software Engineering Notes* 35.3 (2010), pp. 8–13. ISSN: 0163-5948. DOI: 10.1145/1764810.1764814 (cit. on pp. 18, 110).

[Sandkuhl18] Kurt Sandkuhl, Hans-Georg Fill, Stijn Hoppenbrouwers, John Krogstie, Florian Matthes, Andreas Opdahl, Gerhard Schwabe, Ömer Uludag, and Robert Winter. "From Expert Discipline to Common Practice: A Vision and Research Agenda for Extending the Reach of Enterprise Modeling". In: *Business & Information Systems Engineering* 60.1 (Feb. 2018), pp. 69–80. ISSN: 1867-0202. DOI: 10.1007/s12599-017-0516-y (cit. on pp. 16, 59, 110).

[Schröder18] Sandra Schröder and Matthias Riebisch. "An Ontology-Based Approach for Documenting and Validating Architecture Rules". In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. ECSA '18. New York, NY, USA: Association for Computing Machinery, 2018. DOI: 10.1145/3241403.3241457 (cit. on p. 11).

[Schwarz18] Julian Schwarz, Andreas Steffens, and Horst Lichter. "Code Smells in Infrastructure as Code". In: *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. 2018, pp. 220–228. DOI: 10.1109/QUATIC.2018.00040 (cit. on p. 19).

[Sharma16] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. "Does Your Configuration Code Smell?" In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 189–200. ISBN: 9781450341868. DOI: 10.1145/2901739.2901761 (cit. on p. 19).

[Sharwood18] Simon Sharwood. "IBM bans all removable storage, for all staff, everywhere". In: *The Register* (May 10, 2018). URL: https://www.theregister.com/2018/05/10/ibm_bans_all_removable_storage_for_all_staff_everywhere/ (cit. on p. 50).

[SNMP] Jeffrey Case, Mark Fedor, Martin Schoffstall, and James Davin. *Simple network management protocol (SNMP)*. Tech. rep. 1989 (cit. on p. 60).

[Somers23.1] Benjamin Somers, Fabien Dagnat, and Jean-Christophe Bach. "How IT Infrastructures Break: Better Modeling for Better Risk Management". In: *Risks and Security of Internet and Systems*. Ed. by Slim Kallel, Mohamed Jmaiel, Mohammad Zulkernine, Ahmed Hadj Kacem, Frédéric Cuppens, and Nora Cuppens. Cham: Springer Nature Switzerland, 2023, pp. 169–184. ISBN: 978-3-031-31108-6. DOI: 10.1007/978-3-031-31108-6_13 (cit. on p. 11).

[Somers23.2] Benjamin Somers, Fabien Dagnat, and Jean-Christophe Bach. "Modeling Heterogeneous IT Infrastructures: A Collaborative Component-Oriented Approach". In: *Enterprise, Business-Process and Information Systems Modeling*. Ed. by Han van der Aa, Dominik Bork, Henderik A. Proper, and Rainer Schmidt. Springer Nature Switzerland, 2023, pp. 227–242. ISBN: 978-3-031-34241-7 (cit. on p. 104).

[Sommerville12] Ian Sommerville, Dave Cliff, Radu Calinescu, Justin Keen, Tim Kelly, Marta Kwiatkowska, John Mcdermid, and Richard Paige. "Large-Scale Complex IT Systems". In: *Communications of the ACM* 55.7 (July 2012), pp. 71–77. ISSN: 0001-0782. DOI: 10.1145/2209249.2209268 (cit. on p. 84).

[SonarQube] SonarSource. *SonarQube*. URL: https://www.sonarqube.org/ (cit. on p. 41).

[Steel07] Jim Steel and Jean-Marc Jézéquel. "On model typing". In: *Software & Systems Modeling* 6.4 (Dec. 2007), pp. 401–413. ISSN: 1619-1374. DOI: 10.1007/s10270-006-0036-6 (cit. on p. 116).

[Sulaman19] Sardar Muhammad Sulaman, Armin Beer, Michael Felderer, and Martin Höst. "Comparison of the FMEA and STPA safety analysis methods–a case study". In: *Software Quality Journal* 27.1 (2019). ISSN: 1573-1367. DOI: 10.1007/s11219-017-9396-0 (cit. on p. 45).

[Tarjan72] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010 (cit. on p. 65).

[Ten10] Chee-Wooi Ten, Govindarasu Manimaran, and Chen-Ching Liu. "Cybersecurity for Critical Infrastructures: Attack and Defense Modeling". In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 40.4 (2010), pp. 853–865. DOI: 10.1109/TSMCA.2010.2048028 (cit. on p. 12).

[Tenable Nessus] Tenable. *Tenable Nessus*. URL: https://www.tenable.com/products/nessus (cit. on p. 36).

[Teodorov23] Ciprian Teodorov. "G∀min∃: Exploring the Boundary Between Executable Specification Languages and Behavior Analysis Tools". Habilitation à diriger des recherches. Université de Bretagne Occidentale (UBO), Brest, Apr. 2023. URL: https://hal.science/tel-04066483 (cit. on p. 85).

[Threat Taxonomy] European Union Agency for Cybersecurity. *Threat Taxonomy*. URL: https://www.enisa.europa.eu/topics/cyber-threats/threats-and-trends/enisa-threat-landscape/threat-taxonomy/view (cit. on p. 34).

[TOGAF] The Open Group. *The TOGAF® Standard*. URL: https://publications.opengroup.org/c182 (cit. on p. 16).

[Trellix ePO] Trellix. *Trellix ePolicy Orchestrator*. URL: https://trellix.com/products/epo/ (cit. on p. 36).

[Turner10] Julian Turner. "Effects of Data Center Vibration on Compute System Performance". In: *First USENIX Workshop on Sustainable Information Technology (SustainIT 10)*. San Jose, CA: USENIX Association, Feb. 2010. URL: https://www.usenix.org/conference/sustainit-10/effects-data-center-vibration-compute-system-performance (cit. on p. 13).

[Uchenna17] Uchenna Daniel Ani, Hongmei He, and Ashutosh Tiwari. "Review of cybersecurity issues in industrial critical infrastructure: manufacturing in perspective". In: *Journal of Cyber Security Technology* 1.1 (2017), pp. 32–74. DOI: 10.1080/23742917.2016.1252211 (cit. on p. 11).

[UML] Object Management Group. *Unified Modeling Language (UML), Version 2.5.1*. Dec. 2017. URL: https://www.omg.org/spec/UML/2.5.1 (cit. on pp. 14, 61).

[Vaidya19] Ruturaj Kiran Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. *Security Issues in Language-based Software Ecosystems*. 2019. URL: https://arxiv.org/abs/1903.02613 (cit. on p. 12).

[Vaillancourt20] Peter Vaillancourt, Bennett Wineholt, Brandon Barker, Plato Deliyannis, Jackie Zheng, Akshay Suresh, Adam Brazier, Rich Knepper, and Rich Wolski. "Reproducible and Portable Workflows for Scientific Computing and HPC in the Cloud". In: *Practice and Experience in Advanced Research Computing*. PEARC '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 311–320. ISBN: 9781450366892. DOI: 10.1145/3311790.3396659 (cit. on p. 20).

[Vallespir18] Bruno Vallespir and Yves Ducq. "Enterprise modelling: from early languages to models transformation". In: *International Journal of Production Research* 56.8 (2018), pp. 2878–2896. DOI: 10.1080/00207543.2017.1418985 (cit. on p. 16).

[vdAalst13] Wil van der Aalst. "Business Process Management: A Comprehensive Survey". In: *ISRN Software Engineering* (2013). ISSN: 2356-7872. DOI: 10.1155/2013/507984 (cit. on p. 107).

[vdLinden20] Dirk van der Linden, Jolita Ralyté, Kurt Sandkuhl, and Jelena Zdravkovic. "Panel Discussion: Enterprise Modeling in the Digital Age". In: *The Practice of Enterprise Modeling*. 2020. URL: https://api.semanticscholar.org/CorpusID:231879000 (cit. on p. 108).

[Vernadat02] François Vernadat. "UEML: Towards a unified enterprise modelling language". In: *International Journal of Production Research* 40.17 (2002), pp. 4309–4321. DOI: 10.1080/00207540210159626 (cit. on pp. 16, 108).

[Vernadat20] François Vernadat. "Enterprise modelling: Research review and outlook". In: *Computers in Industry* 122 (2020), p. 103265. ISSN: 0166-3615. DOI: 10.1016/j.compind.2020.103265 (cit. on pp. 16, 17, 104, 107).

[Visa22] Visa. *Visa Core Rules and Visa Product and Service Rules*. 2022. URL: https://bb.visa.com/content/dam/VCOM/download/about-visa/visa-rules-public.pdf (cit. on p. 11).

[Voas21] Jeffrey Voas, Nir Kshetri, and Joanna F. DeFranco. "Scarcity and Global Insecurity: The Semiconductor Shortage". In: *IT Professional* 23.5 (2021), pp. 78–82. DOI: 10.1109/MITP.2021.3105248 (cit. on p. 12).

[Voinov10] Alexey Voinov and François Bousquet. "Modelling with stakeholders". In: *Environmental Modelling & Software* 25 (2010), pp. 1268–1281. ISSN: 1364-8152. DOI: 10.1016/j.envsoft.2010.03.007 (cit. on p. 104).

[Wehling17] Kenny Wehling and Ina Schaefer. "Towards an Expert System for Identifying and Reducing Unnecessary Complexity of IT Architectures". In: *INFORMATIK 2017* (2017). DOI: 10.18420/in2017_152 (cit. on p. 80).

[Weinreich14] Rainer Weinreich and Georg Buchgeher. "Automatic Reference Architecture Conformance Checking for SOA-Based Software Systems". In: *2014 IEEE/IFIP Conference on Software Architecture*. 2014, pp. 95–104. DOI: 10.1109/WICSA.2014.22 (cit. on p. 82).

[White02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. "An integrated experimental environment for distributed systems and networks". In: *SIGOPS Operating Systems Review* 36 (Dec. 2003), pp. 255–270. ISSN: 0163-5980. DOI: 10.1145/844128.844152 (cit. on p. 97).

[Xu15] Tianyin Xu and Yuanyuan Zhou. "Systems Approaches to Tackling Configuration Errors: A Survey". In: *ACM Computing Surveys* 47.4 (July 2015). ISSN: 0360-0300. DOI: 10.1145/2791577 (cit. on p. 19).

[Yates14] Athol Yates. "A framework for studying mortality arising from critical infrastructure loss". In: *International Journal of Critical Infrastructure Protection* 7.2 (2014). ISSN: 1874-5482. DOI: 10.1016/j.ijcip.2014.04.002 (cit. on p. 13).

Appendix A

`mitre2owl` Algorithm

Contents

A.1 Detail of the algorithm 143
A.1.1 Parsers$_S$ 143
A.1.2 Parsers$_D$ 145
A.2 Semantic transformation 147

A.1 Detail of the algorithm

In this appendix, we present the algorithms behind `mitre2owl` along with the semantic transformation rules from its output to a well-formed Turtle representation. The code for `mitre2owl` can be found on Github (`https://github.com/CAPRICA-Project/mitre2owl`).

A.1.1 Parsers$_S$

First of all, `mitre2owl` parses schemas to understand the structure of the data it needs to parse in a second phase.

Algorithm: Parser builder (`Schema`)

Data: `xs:schema` element
`types` $\leftarrow \{\}$;
`elements` $\leftarrow \{\}$;
foreach `n` $\in$ `/*/xs:complexType` **do**
  `types[n]` $\leftarrow$ `ComplexType(n)`
end
foreach `n` $\in$ `/*/xs:simpleType` **do**
  `types[n]` $\leftarrow$ `SimpleType(n)`
end
foreach `/*/xs:element` **do**
  `elements[@name]` $\leftarrow$ `Element(.)`
end

Algorithm: Element parser builder (`Element`)
Data: `xs:element` element
`type` ← `@type`

Algorithm: Attribute parser builder (`Attribute`)
Data: `xs:attribute` element
`type` ← `@type`

Algorithm: Any parser builder (`Any`)
Data: `xs:any` element
`/* The algorithm does no processing */`

Algorithm: SimpleType parser builder (`SimpleType`)
Data: `xs:simpleType` element
`/* The algorithm does no processing */`

Algorithm: ComplexType parser builder (`ComplexType`)
Data: `xs:complexType` element
`attrs` ← {};
`type` ← ?;
foreach `xs:attribute` **do**
  `attrs[@name] = Attribute(.)`
end
if `xs:sequence[1]` **then**
  `type` ← `Sequence(.)`
else
  if `xs:extension[1]` **then**
    `type` ← `Extension(.)`
  else
    `type` ← `Choice(xs:choice[1])`
  end
end

Algorithm: Sequence parser builder (`Sequence`)
Data: `xs:sequence` element
`children` ← {};
`any` ← `xs:any[1]`;
foreach `xs:element` **do**
  `children[@name] += Element(.)`
end
foreach `xs:choice` **do**
  `children += Choice(.).children`
end

Algorithm: Choice parser builder (`Choice`)

Data: `xs:choice` element

`children` ← {};

foreach `xs:element` **do**

  `children` = `Element(.)`

end

foreach `xs:sequence` **do**

  `children` += `Sequence(.).children`

end

Algorithm: Extension parser builder (`Extension`)

Data: `xs:extension` element

`base` ← `@base`;

`attrs` ← {};

foreach `xs:attribute` **do**

  `attrs[@name]` = `Attribute(.)`

end

A.1.2 Parsers$_D$

After the schemas are parsed, `mitre2owl` can parse the actual data.

Algorithm: XML parser (`Schema.parse`)

Data: XML data

`elements[name()].parse(.)`

Algorithm: Element parser (`Element.parse`)

Data: XML element

Result: `<?> <has name() > <types[type].parse(.)>.`

Algorithm: Attribute parser (`Attribute.parse`)

Data: XML element

Result: `<?> <has name() > types[type].parse(.) .`

Algorithm: Any parser (`Any.parse`)

Data: XML element

Result: `type[name()].parse(.)`

Algorithm: SimpleType parser (`SimpleType.parse`)

Data: XML element

Result: `< text() > a name() .`

Algorithm: ComplexType parser (`ComplexType.parse`)

Data: XML element

```
assertions ← [];
foreach @* do
    assertions += attrs[name()].parse(.)
end
assertions += type.parse(.);
```

Result:

```
<makeName(.)> a <name()>; assertions.
```

`makeName` creates a unique name for the parsed element based on its attributes and its type. The implementation is not given here.

Algorithm: Sequence parser (`Sequence.parse`)

Data: XML element

```
assertions ← [];
if any then
    assertions += any.parse(div(*))
else
    foreach * do
        assertions += children[name()].parse(.)
    end
end
```

Result: `assertions`

Algorithm: Choice parser (`Choice.parse`)

Data: XML element

```
assertions ← [];
foreach * do
    assertions += children[name()].parse(.)
end
```

Result: `assertions`

Algorithm: Extension parser (`Extension.parse`)

Data: XML element

```
assertions = [];
foreach @* do
    assertions += attrs[name()].parse(.)
end
assertions += types[base].parse(.);
```

Result: `assertions`

A.2 Semantic transformation

The results of the algorithms are not exactly Turtle expressions, but rather a recursive flavor of Turtle that needs postprocessing. Three rules have to be applied:

1. Recursive definitions must be flattened:
   `<subject> <predicate> < subject' <...> <...>. >` expressions are recursively expanded to `subject predicate subject'. subject' <...> <...>`;
2. Wildcards are removed:
   `<subject> <predicate> <object>;` `<?> <predicate'> <object>.` `.` expressions are expanded to `<subject> <predicate> <object>. <subject> <predicate'> <object'>.`;
3. Valid Turtle expressions are not transformed further:
   `<subject> <predicate> <object>` expressions stay the same.

Appendix B

UPPAAL Model

Contents

B.1 Source code . . . 149
B.1.1 Common functions . . . 149
B.1.2 Corosync cluster . . . 149
B.1.3 Multi-quorum Corosync cluster . . . 150
B.1.4 Corosync node . . . 152
B.1.5 Network node above Corosync . . . 153
B.1.6 Declarations for Corosync . . . 153
B.2 Traces . . . 156
B.2.1 Scenario 1 . . . 156
B.2.2 Scenario 2 . . . 157
B.2.3 Scenario 3 . . . 158

In this appendix, we present the source code for our UPPAAL models, along with the traces of the scenarios described in section V.3.3. This code can also be found on Github (`https://github.com/CAPRICA-Project/UPPAAL-models`).

B.1 Source code

B.1.1 Common functions

```
int max(int i, int j)
{
    return i > j ? i : j;
}
```

B.1.2 Corosync cluster

```
int quorums()
{
    meta int votes = 0;
    for (i : int[0, NODES-1])
```

```
    {
        meta int v = 0;
        for (j : int[0, NODES-1])
            if (seems_up[i][j])
                v += VOTES[j];
        votes = max(votes, v);
    }
    return votes >= expected_votes;
}
```

B.1.3 Multi-quorum Corosync cluster

```
int nth_adj(int v, int n)
{
    meta int adj = -1, i = 0;
    while (n-- >= 0 && i < NODES) // The model checker needs the second condition
    {
        while (!seems_up[v][i])
            if (++i == NODES) // The model checker needs this test
                return adj;
        adj = i;
        i++;
    }
    return adj;
}

int n_adj(int v)
{
    meta int n = 0;
    for (i : int[0, NODES-1])
        if (seems_up[v][i])
            n++;
    return n;
}

int quorums()
{
    meta int comp[NODES];
    meta int low[NODES];
    meta int idx[NODES];
    meta bool in_stack[NODES];
    meta int i = 0;
    meta int stack[NODES];
```

```
meta int stack_i = 0;
meta int call_stack[NODES][2];
meta int call_stack_i = 0;
meta int comp_i = 0;
meta int vv, pi, n, w;
meta int votes[NODES];
meta int n_quorums = 0;

for (k : int[0, NODES-1])
{
    comp[k] = 0;
    low[k] = 0;
    idx[k] = -1;
}

for (v : int[0, NODES-1])
{
    if (idx[v] == -1)
    {
        call_stack[call_stack_i][0] = v;
        call_stack[call_stack_i++][1] = 0;
        while (call_stack_i)
        {
            call_stack_i--;
            vv = call_stack[call_stack_i][0];
            pi = call_stack[call_stack_i][1];
            if (pi == 0)
            {
                idx[vv] = i;
                low[vv] = i++;
                stack[stack_i++] = vv;
                in_stack[vv] = true;
            }
            else if (pi > 0)
            {
                low[vv] = min(low[vv], low[nth_adj(vv, pi-1)]);
            }
            n = n_adj(vv);
            while (pi < n && idx[nth_adj(vv, pi)] != -1)
            {
                w = nth_adj(v, pi++);
                if (in_stack[w])
```

```
                    low[vv] = min(low[vv], idx[w]);
                }
                if (pi < n)
                {
                    w = nth_adj(vv, pi);
                    call_stack[call_stack_i][0] = vv;
                    call_stack[call_stack_i++][1] = pi+1;
                    call_stack[call_stack_i][0] = w;
                    call_stack[call_stack_i++][1] = 0;
                }
                else if (low[vv] == idx[vv])
                {
                    w = -1;
                    do
                    {
                        w = stack[--stack_i];
                        in_stack[w] = false;
                        comp[w] = comp_i;
                    }
                    while (vv != w);
                    comp_i++;
                }
            }
        }
    }

    for (k : int[0, NODES-1])
        votes[comp[k]] += VOTES[k];
    for (k : int[0, NODES-1])
        if (votes[k] >= expected_votes)
            n_quorums++;
    return n_quorums;
}
```

B.1.4 Corosync node

```
broadcast chan received, not_received;

int current_node;

void reset()
{
    for (j : int[0, NODES-1])
```

```
        seems_up[i][j] = false;
}

void update_net()
{
    for (j : int[0,NODES-1])
        seems_up[i][j] = seems_up[i][j] && can_communicate[j][i];
}
```

B.1.5 Network node above Corosync

```
void update() {
    meta int distances[NODES+NETWORK_NODES][NODES+NETWORK_NODES];
    for (i : int[0,NODES+NETWORK_NODES-1])
    {
        meta bool link_i = i < NODES ? 1 : network_up[i-NODES];
        for (j : int[0,NODES+NETWORK_NODES-1])
        {
            meta bool link_j = j < NODES ? 1 : network_up[j-NODES];
            distances[i][j] = link_i && link_j ? i == j ? 0 : NETWORK[i][j] ? 1
    : 1000 : 1000;
        }
    }
    for (k : int[0,NODES+NETWORK_NODES-1])
        for (i : int[0,NODES+NETWORK_NODES-1])
            for (j : int[0,NODES+NETWORK_NODES-1])
                if (distances[i][j] > distances[i][k] + distances[k][j])
                    distances[i][j] = distances[i][k] + distances[k][j];
    for (i : int[0,NODES-1])
        for (j : int[0,NODES-1])
            can_communicate[i][j] = distances[i][j] < 1000;
}

void fail() {
    network_up[i] = false;
    update();
}

void recover() {
    network_up[i] = true;
    update();
}
```

B.1.6 Declarations for Corosync

```
const int NODES = 8;
const int NETWORK_NODES = 16;
const int expected_votes = 5;
const int VOTES[NODES] = {1,1,1,1,1,1,1,1};

const int STRUCTURAL = 8;
const bool STRUCTURE[NODES+NETWORK_NODES][STRUCTURAL] = {
    /* ^ */ {1,0,1,0,0,0,0,0},
    /* N */ {1,0,1,0,0,0,0,0},
    /* O */ {1,0,0,1,0,0,0,0},
    /* D */ {1,0,0,1,0,0,0,0},
    /* E */ {0,0,0,0,1,0,1,0},
    /* S */ {0,0,0,0,1,0,1,0},
    /*   */ {0,0,0,0,1,0,0,1},
    /* v */ {0,0,0,0,1,0,0,1},

    /* ^ */ {1,1,0,0,0,0,0,0},
    /* N */ {1,1,0,0,0,0,0,0},
    /* E */ {1,1,0,0,0,0,0,0},
    /* T */ {1,1,0,0,0,0,0,0},
    /* W */ {1,0,1,0,0,0,0,0},
    /* O */ {1,0,1,0,0,0,0,0},
    /* R */ {1,0,0,1,0,0,0,0},
    /* K */ {1,0,0,1,0,0,0,0},
    /*   */ {0,0,0,0,1,1,0,0},
    /* N */ {0,0,0,0,1,1,0,0},
    /* O */ {0,0,0,0,1,1,0,0},
    /* D */ {0,0,0,0,1,1,0,0},
    /* E */ {0,0,0,0,1,0,1,0},
    /* S */ {0,0,0,0,1,0,1,0},
    /*   */ {0,0,0,0,1,0,0,1},
    /* v */ {0,0,0,0,1,0,0,1}
};

bool can_communicate[NODES][NODES] = {
    {1,1,1,1,1,1,1,1},
    {1,1,1,1,1,1,1,1},
    {1,1,1,1,1,1,1,1},
    {1,1,1,1,1,1,1,1},
    {1,1,1,1,1,1,1,1},
    {1,1,1,1,1,1,1,1},
    {1,1,1,1,1,1,1,1},
```

```
    {1,1,1,1,1,1,1,1}
};

const bool NETWORK[NODES+NETWORK_NODES][NODES+NETWORK_NODES] = {
//       <----NODES---->  <--------NETWORK NODES-------->
/* ^ */ {0,0,0,0,0,0,0,0, 0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0},
/* N */ {0,0,0,0,0,0,0,0, 0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0},
/* O */ {0,0,0,0,0,0,0,0, 0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0},
/* D */ {0,0,0,0,0,0,0,0, 0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0},
/* E */ {0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0},
/* S */ {0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0},
/*   */ {0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1},
/* v */ {0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1},

/* ^ */ {0,0,0,0,0,0,0,0, 0,1,1,1,0,0,0,0,1,0,0,0,0,0,0,0},
/* N */ {0,0,0,0,0,0,0,0, 1,0,1,1,0,0,0,0,0,1,0,0,0,0,0,0},
/* E */ {0,0,0,0,0,0,0,0, 1,1,0,1,1,0,1,0,0,0,0,0,0,0,0,0},
/* T */ {0,0,0,0,0,0,0,0, 1,1,1,0,0,1,0,1,0,0,0,0,0,0,0,0},
/* W */ {1,1,0,0,0,0,0,0, 0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0},
/* O */ {1,1,0,0,0,0,0,0, 0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0},
/* R */ {0,0,1,1,0,0,0,0, 0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0},
/* K */ {0,0,1,1,0,0,0,0, 0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0},
/*   */ {0,0,0,0,0,0,0,0, 1,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0},
/* N */ {0,0,0,0,0,0,0,0, 0,1,0,0,0,0,0,0,1,0,1,1,0,0,0,0},
/* O */ {0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,1,1,0,1,1,0,1,0},
/* D */ {0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,1,1,1,0,0,1,0,1},
/* E */ {0,0,0,0,1,1,0,0, 0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0},
/* S */ {0,0,0,0,1,1,0,0, 0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0},
/*   */ {0,0,0,0,0,0,1,1, 0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},
/* v */ {0,0,0,0,0,0,1,1, 0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0}
};

bool network_up[NETWORK_NODES] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};

broadcast chan up[NODES];
bool seems_up[NODES][NODES];
broadcast chan fail[NODES];
broadcast chan recover[NODES];
broadcast chan str_fail[STRUCTURAL];
broadcast chan str_recover[STRUCTURAL];
bool str_failed[STRUCTURAL];
broadcast chan net;
```

B.2 Traces

B.2.1 Scenario 1

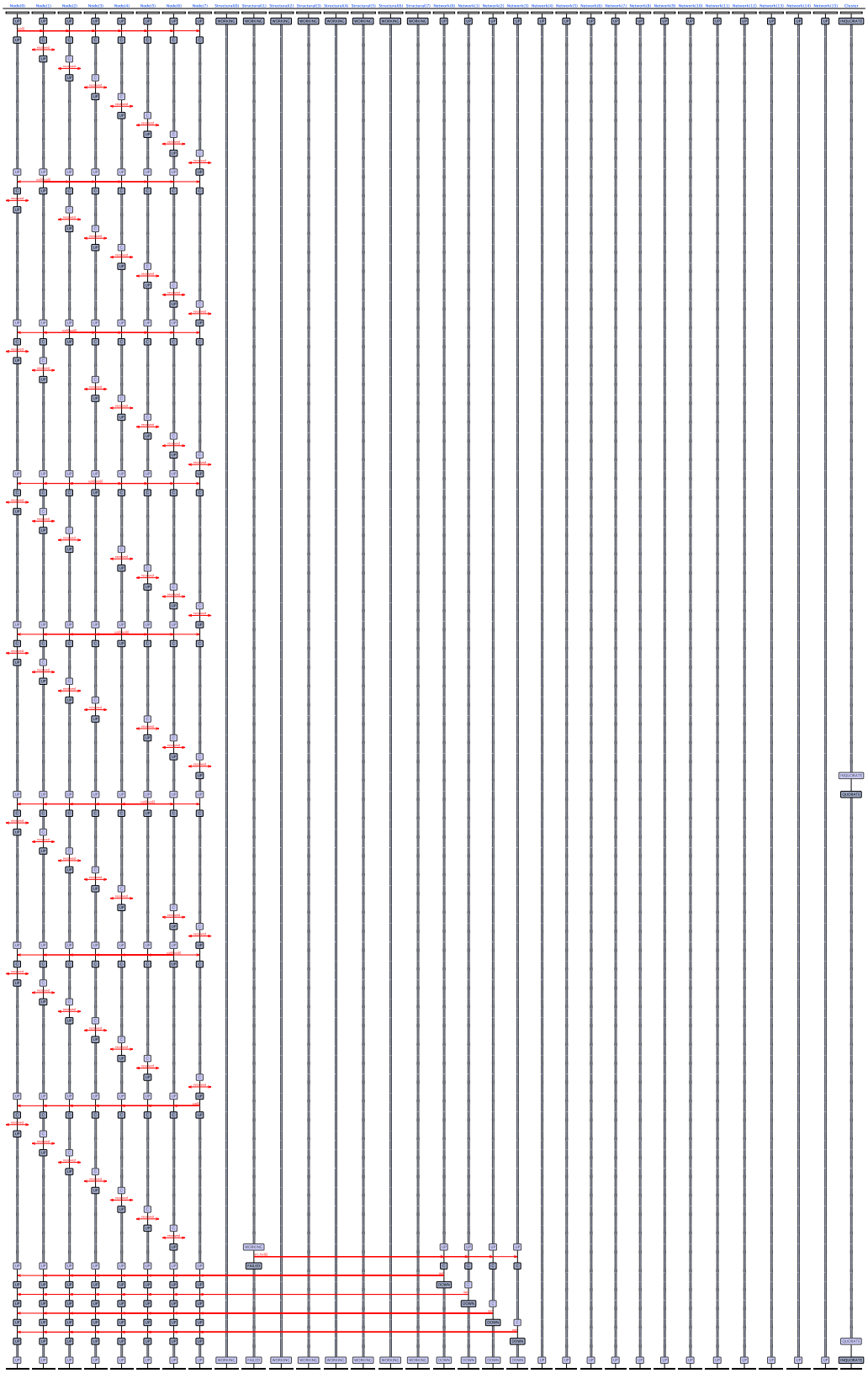


Figure B.1: Trace for the scenario 1

B.2.2 Scenario 2

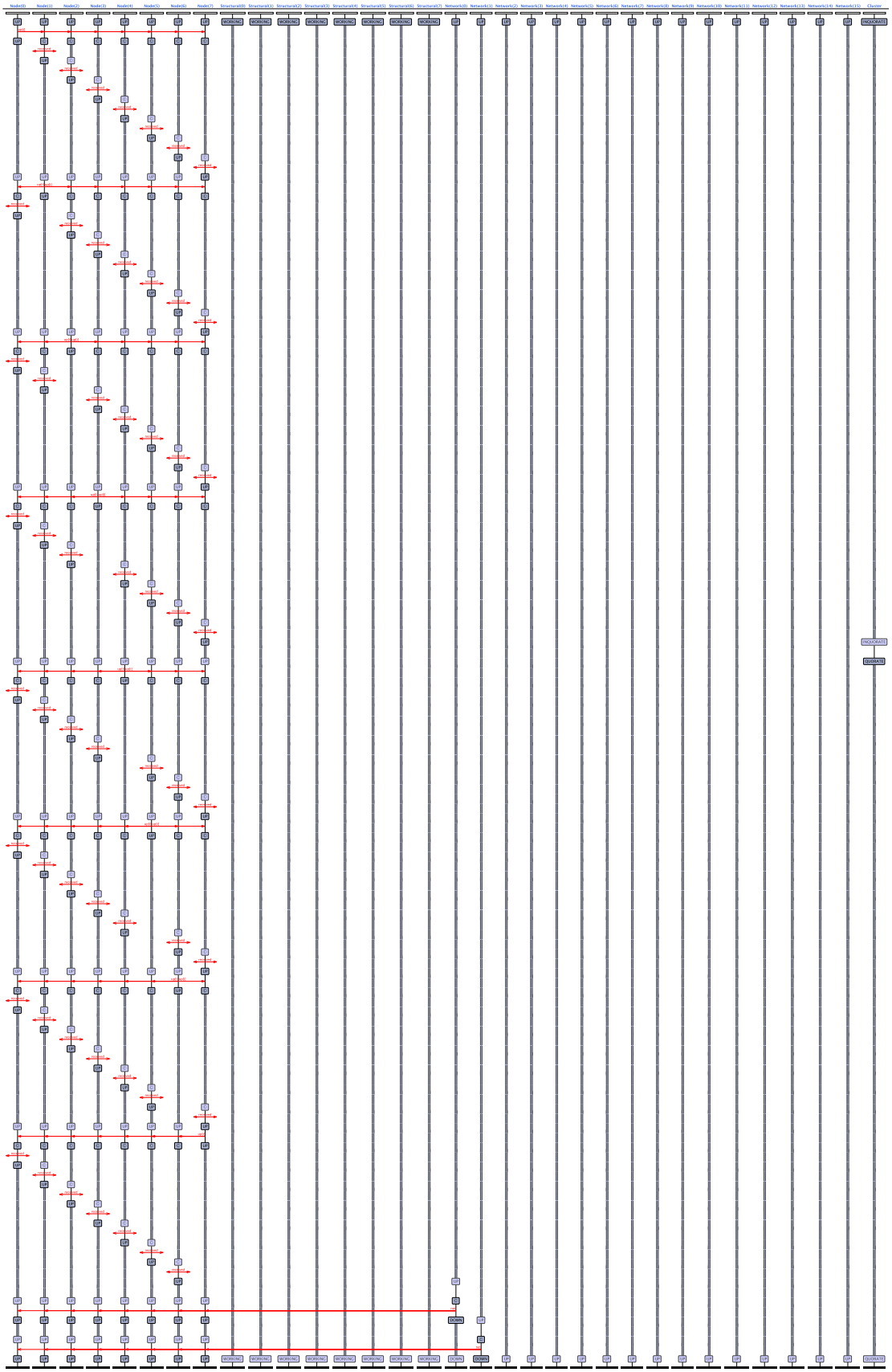


Figure B.2: Trace for the scenario 2

B.2.3 Scenario 3

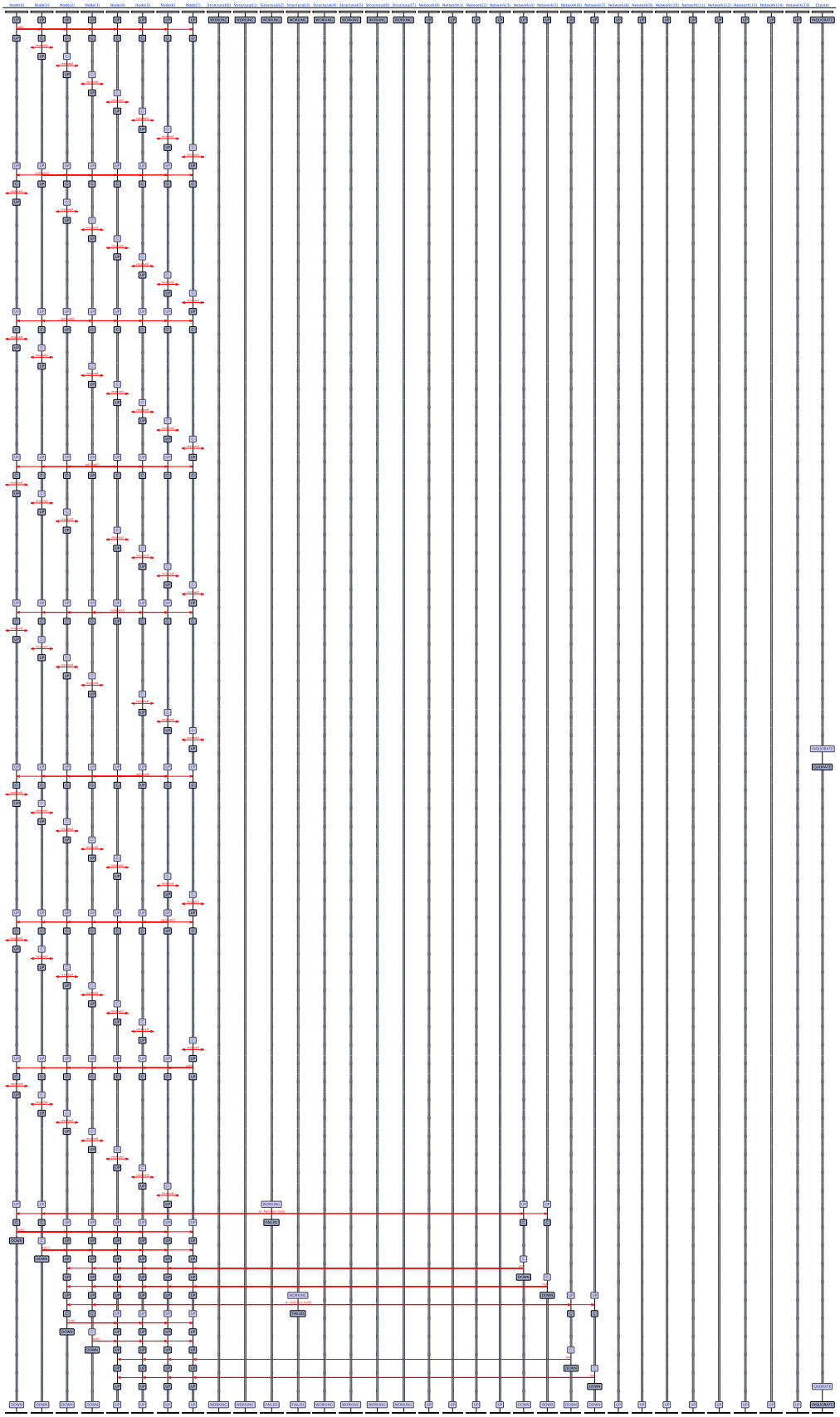


Figure B.3: Trace for the scenario 3

Appendix C

CL/I

Contents

C.1 Language grammar 160
C.2 AST construction rules 168
C.3 Transformation from CL/I's AST to the CLIR 170
C.3.1 AST 170
C.3.2 Structure 171
C.3.3 Right values 174
C.3.4 RTRDot 175
C.3.5 RInit 176
C.3.6 Right types 177

In this appendix, we present the grammar of CL/I (appendix C.1), how its AST is built (appendix C.2) and how this AST is transformed into the CLIR intermediate representation (appendix C.3). The code for CL/I is available on Github (`https://github.com/CAPRICA-Project/CL-I`).

C.1 Language grammar

The grammar presented here follows a parameterized EBNF syntax: $\langle \texttt{rule}(\textit{arguments}) \rangle$ ::= $\textit{choice}_1 \mid \textit{choice}_2$. Sequences of characters recognized as tokens by CL/I's lexer are written `as such`. Parameterized tokens and the "end of file" (`EOF`) token are written `AS SUCH`. We define conditional rules thanks to a special encoding for booleans given by the $\top$ (true) and $\bot$ (false) rules, and write booleans `as such`. Rule arguments that are themselves rules are written `as such`. X^*_Y (respectively X^+_Y) denotes possibly empty (respectively non-empty) sequences of Xes each separated with Ys. Finally, ϵ denotes the empty token sequence.

$$
\begin{aligned}
\langle \texttt{start} \rangle &::= \langle \texttt{structures\_or\_defs} \rangle\ \texttt{EOF} \\
\\
\langle \texttt{arg}(X) \rangle &::= X \\
&\mid\ \texttt{...} \\
\\
\langle \texttt{lax}(X) \rangle &::= X \\
&\mid\ \texttt{?} \\
\\
\langle \texttt{arg\_list}(X) \rangle &::= \left[\texttt{...}\ \texttt{,}\right] X \left[\texttt{,}\ \langle \texttt{arg\_list\_continue}(X) \rangle\right] \\
\\
\langle \texttt{arg\_list\_continue}(X) \rangle &::= \texttt{...} \left[\texttt{,}\ X \left[\texttt{,}\ \langle \texttt{arg\_list\_continue}(X) \rangle\right]\right] \\
&\mid\ X \left[\texttt{,}\ \langle \texttt{arg\_list\_continue}(X) \rangle\right] \\
\\
\langle \texttt{parameter\_list}(X) \rangle &::= \texttt{(}\ \texttt{)} \\
&\mid\ \langle \texttt{inpar}(\langle \texttt{arg\_list}(X) \rangle) \rangle \\
\\
\langle \top(X, \cdot) \rangle &::= X \\
\\
\langle \bot(\cdot, Y) \rangle &::= Y \\
\\
\langle \texttt{ifte}(\texttt{b}, \mathit{THEN}, \mathit{ELSE}) \rangle &::= \texttt{b}(\mathit{THEN}, \mathit{ELSE})
\end{aligned}
$$

$$
\begin{aligned}
\langle \texttt{ift}(\texttt{b}, \mathit{THEN}) \rangle &::= \texttt{b}(\mathit{THEN}, \epsilon) \\
\\
\langle \texttt{unless}(\texttt{b}, \mathit{THEN}) \rangle &::= \langle \texttt{ifte}(\texttt{b}, \epsilon, \mathit{THEN}) \rangle \\
\\
\langle \texttt{structure\_or\_def} \rangle &::= \langle \texttt{structure} \rangle \\
&\mid \langle \texttt{def} \rangle \\
&\mid \langle \texttt{set} \rangle \\
&\mid \langle \texttt{rhs}(\top) \rangle \\
\\
\langle \texttt{structures\_or\_defs} \rangle &::= \langle \texttt{structure\_or\_def} \rangle^* \\
\\
\langle \texttt{def} \rangle &::= \langle \texttt{decl} \rangle \\
&\mid \langle \texttt{assignment}(\bot) \rangle \\
\\
\langle \texttt{assignment}(\texttt{is\_litt}) \rangle &::= \texttt{let}\ \texttt{[]}\ \langle \texttt{tpar}(\langle \texttt{lhs\_id} \rangle, \top) \rangle\ \texttt{=}\ \langle \texttt{rhs}(\top) \rangle \\
&\mid \texttt{let}\ \langle \texttt{lhs} \rangle\ \texttt{=}\ \langle \texttt{rhs}(\top) \rangle \\
&\mid \texttt{type}\ \langle \texttt{lhs\_type} \rangle\ \texttt{=}\ \langle \texttt{typespec} \rangle \left\langle \texttt{unless}\left(\texttt{is\_litt}, \texttt{;}\right) \right\rangle \\
\\
\langle \texttt{decl} \rangle &::= \texttt{let}\ \texttt{[]}\ \langle \texttt{tpar}(\langle \texttt{lhs\_id} \rangle, \top) \rangle\ \texttt{;} \\
&\mid \texttt{let}\ \langle \texttt{lhs\_sig} \rangle\ \texttt{;} \\
&\mid \texttt{let}\ \left\langle \texttt{more\_list}\left(\texttt{,}, \langle \texttt{lhs\_sig} \rangle\right) \right\rangle\ \texttt{;} \\
&\mid \texttt{type}\ \langle \texttt{lhs\_type} \rangle\ \texttt{;} \\
&\mid \texttt{type}\ \left\langle \texttt{more\_list}\left(\texttt{,}, \langle \texttt{lhs\_type} \rangle\right) \right\rangle\ \texttt{;} \\
\\
\langle \texttt{set} \rangle &::= \langle \texttt{rhs\_value} \rangle\ \texttt{<-}\ \langle \texttt{rhs}(\top) \rangle \\
\\
\langle \texttt{structure} \rangle &::= \langle \texttt{component\_structure} \rangle
\end{aligned}
$$

$$
\begin{array}{rcl}
 & | & \langle \texttt{pragma} \rangle \\
\langle \texttt{pragma} \rangle & ::= & \texttt{::}\ \langle \texttt{dot\_rhs} \rangle\ \texttt{;} \\
\langle \texttt{component\_structure} \rangle & ::= & \langle \texttt{annotation} \rangle^{*}\ \texttt{component}\ \left[\texttt{interface}\right]\ \left\langle \texttt{more\_list}\left(\texttt{,},\langle \texttt{csig} \rangle\right) \right\rangle\ \texttt{;} \\
 & | & \langle \texttt{annotation} \rangle^{*}\ \texttt{component}\ \left[\texttt{interface}\right]\ \langle \texttt{csig} \rangle\ \langle \texttt{component\_body} \rangle \\
\langle \texttt{csig} \rangle & ::= & \texttt{UID}\ [\langle \texttt{lhs\_type\_params} \rangle]\ [\langle \texttt{component\_parameters} \rangle] \\
 & | & \texttt{(}\ \texttt{UID}\ [\langle \texttt{lhs\_type\_params} \rangle]\ \texttt{<:}\ \langle \texttt{dot\_rhs\_ty} \rangle^{+}_{\texttt{,}}\ \texttt{)}\ [\langle \texttt{component\_parameters} \rangle] \\
\langle \texttt{component\_parameters} \rangle & ::= & \left\langle \texttt{inpar}\left(\langle \texttt{arg}(\langle \texttt{tpar}(\langle \texttt{lhs\_id} \rangle, \top) \rangle) \rangle^{*}_{\texttt{,}} \right) \right\rangle \\
\langle \texttt{annotation} \rangle & ::= & \texttt{@}\ \langle \texttt{rhs\_type} \rangle \\
\langle \texttt{hook} \rangle & ::= & \texttt{\^{}}\ \langle \texttt{dot\_rhs} \rangle \\
\langle \texttt{component\_body} \rangle & ::= & \langle \texttt{hook} \rangle^{*}\ \texttt{;} \\
 & | & \texttt{=}\ \langle \texttt{rhs}(\top) \rangle\ \left[\langle \texttt{hook} \rangle^{+}\ \texttt{;}\right] \\
 & | & \texttt{\{}\ \langle \texttt{structures\_or\_defs} \rangle\ \texttt{\}}\ \left[\langle \texttt{hook} \rangle^{+}\ \texttt{;}\right] \\
\langle \texttt{tpar}(X,\texttt{allow\_paren}) \rangle & ::= & \langle \texttt{fun\_in\_par}(\langle \texttt{t} \rangle, X, \texttt{allow\_paren}) \rangle \\
\langle \texttt{t}(X) \rangle & ::= & X\ [\langle \texttt{of\_type} \rangle] \\
\langle \texttt{lhs\_fun\_sig} \rangle & ::= & \langle \texttt{lhs\_id} \rangle\ \langle \texttt{parameter\_list}(\langle \texttt{lax}(\langle \texttt{typespec} \rangle) \rangle) \rangle\ \left[\texttt{->}\ \langle \texttt{lax}(\langle \texttt{typespec} \rangle) \rangle\right] \\
\langle \texttt{lhs\_sig} \rangle & ::= & \langle \texttt{tpar}(\langle \texttt{simple\_lhs} \rangle, \top) \rangle
\end{array}
$$

⟨lhs_type_params⟩ ::= ⟨type_params(⟨uid_or_word⟩)⟩
| WORD

⟨uid_or_word⟩ ::= UID
| WORD

⟨lhs_type⟩ ::= UID [⟨lhs_type_params⟩]
| ⟨tuple(⟨arg(⟨lax(⟨lhs_type⟩)⟩)⟩)⟩

⟨type_params(X)⟩ ::= < ⟨arg(X)⟩*, >

⟨tuple(X)⟩ ::= ⟨inpar(⟨more_list(,,X)⟩)⟩

⟨fun_in_par(f,X,recurse)⟩ ::= ⟨ifte(recurse,⟨_fun_in_par(f,X)⟩,f(X))⟩

⟨_fun_in_par(f,X)⟩ ::= f(X)
| f(⟨inpar(⟨_fun_in_par(f,X)⟩)⟩)

⟨ttpar(X,allow_paren)⟩ ::= ⟨fun_in_par(⟨tt⟩,X,allow_paren)⟩

⟨tt(X)⟩ ::= X [⟨of_type⟩]

⟨rhs_type⟩ ::= UID [⟨rhs_type_params⟩] ⟨cardinality⟩*

⟨dot_rhs_ty⟩ ::= ⟨dot_rhs⟩ . ⟨rhs_type⟩
| ⟨dot_rhs_ty⟩ . ⟨rhs_type⟩
| ⟨rhs_type⟩

⟨cardinality⟩ ::= +

	|	`*`
	|	`[]`
	|	`[` ⟨rhs(⊥)⟩ $\left[$ `...` [⟨rhs(⊥)⟩]$\right]$ `]`
	|	`[` `...` ⟨rhs(⊥)⟩ `]`
⟨rhs_type_params⟩	::=	⟨type_params(⟨lax(⟨typespec_or_word⟩)⟩)⟩
	|	WORD
⟨typespec_or_word⟩	::=	⟨typespec⟩
	|	WORD
⟨of_type⟩	::=	`:` ⟨typespec⟩
⟨typespec⟩	::=	⟨typename⟩
	|	⟨arrow_list⟩
⟨typename⟩	::=	⟨dot_rhs_ty⟩
	|	⟨tuple(⟨typespec⟩)⟩
	|	`(` `)`
⟨arrow_list⟩	::=	⟨arg(⟨lax(⟨typename⟩)⟩)⟩$^{+}_{\texttt{->}}$ `->` ⟨lax(⟨typename⟩)⟩
⟨more_list(SEP,X)⟩	::=	X SEP X^{+}_{SEP}
⟨inpar(X)⟩	::=	`(` X `)`
⟨either(X,Y)⟩	::=	X
	|	$[X]$ Y

```
⟨simple_lhs⟩  ::=  ⟨lhs_id⟩
                |  ⟨tuple(⟨lhs⟩)⟩
                |  LITEXP

   ⟨lhs_id⟩  ::=  LID

   ⟨lhs_fun⟩  ::=  ⟨lhs_id⟩ ⟨parameter_list(⟨tpar(⟨lhs_id⟩,⊤)⟩)⟩ [ -> ⟨lax(⟨typespec⟩)⟩ ]

       ⟨lhs⟩  ::=  ⟨tpar(⟨simple_lhs⟩,⊤)⟩
                |  ⟨tpar(⟨lhs_fun⟩,⊤)⟩
                |  [ ⟨rhs(⊥)⟩ ]
                |  [ + ]
                |  []

 ⟨rhs_value⟩  ::=  ⟨tuple(⟨rhs(⊥)⟩)⟩
                |  ⟨dot_rhs⟩

  ⟨rhs_list⟩  ::=  [⟨rhs(⊥)⟩ [ ; ⟨rhs_list⟩]]

  ⟨rhs_atom⟩  ::=  LID
                |  ⟨str⟩
                |  []

   ⟨dot_rhs⟩  ::=  ⟨dot_rhs⟩ . ⟨rhs_atom⟩
                |  ⟨dot_rhs_ty⟩ . ⟨rhs_atom⟩
                |  ⟨rhs_atom⟩
                |  [ ⟨rhs_list⟩ ]
                |  ⟨call(⟨dot_rhs⟩,⟨rhs(⊥)⟩)⟩
```

```
                |  ⟨tcall(⟨dot_rhs_ty⟩,⟨rhs(⊥)⟩)⟩
                |  ⟨dot_rhs⟩ { ⟨set⟩* }
                |  ⟨dot_rhs_ty⟩ { ⟨set⟩* }

        ⟨preop⟩ ::= PREOP
                |  !
                |  +
                |  -

       ⟨postop⟩ ::= POSTOP0
                |  POSTOP1
                |  POSTOP2
                |  =
                |  <
                |  >
                |  POSTOP3
                |  +
                |  -
                |  LITEXP
                |  POSTOP4
                |  *

  ⟨bang_postop⟩ ::= ! LITEXP

⟨rhs(has_colon)⟩ ::= ⟨rhs_value⟩ ⟨ift(has_colon,;)⟩
                |  ⟨preop⟩ ⟨rhs(has_colon)⟩
                |  ⟨rhs(⊥)⟩ ⟨postop⟩ ⟨ift(has_colon,;)⟩
                |  ⟨rhs(⊥)⟩ ⟨postop⟩ ⟨rhs(has_colon)⟩
```

$$
\begin{array}{rcl}
 & | & \langle \texttt{rhs}(\bot) \rangle \ \langle \texttt{bang\_postop} \rangle \left\langle \texttt{ift}\left(\texttt{has\_colon}, \texttt{;} \right) \right\rangle \\
 & | & \langle \texttt{rhs}(\bot) \rangle \ \langle \texttt{bang\_postop} \rangle \ \langle \texttt{rhs}(\texttt{has\_colon}) \rangle \\
 & | & \texttt{(} \ \langle \texttt{rhs}(\bot) \rangle \ \texttt{)} \left\langle \texttt{ift}\left(\texttt{has\_colon}, \texttt{;} \right) \right\rangle \\
 & | & \langle \texttt{z3\_property} \rangle \left\langle \texttt{ift}\left(\texttt{has\_colon}, \texttt{;} \right) \right\rangle \\
 & | & \texttt{if} \ \langle \texttt{rhs\_value} \rangle \ \texttt{then} \ \langle \texttt{rhs}(\texttt{has\_colon}) \rangle \\
 & | & \texttt{if} \ \langle \texttt{rhs\_value} \rangle \ \texttt{then} \ \langle \texttt{rhs}(\bot) \rangle \ \texttt{else} \ \langle \texttt{rhs}(\texttt{has\_colon}) \rangle
\end{array}
$$

$$
\langle \texttt{str} \rangle \ ::= \ \texttt{BSTRING} \ \langle \texttt{\_str} \rangle \ \texttt{ESTRING}
$$

$$
\begin{array}{rcl}
\langle \texttt{\_str} \rangle & ::= & [\texttt{STRING} \ \langle \texttt{\_str} \rangle] \\
 & | & \langle \texttt{rhs}(\bot) \rangle \ \texttt{EEXPR} \ \langle \texttt{\_str} \rangle
\end{array}
$$

$$
\langle \texttt{call}(X, Y) \rangle \ ::= \ X \ \langle \texttt{parameter\_list}(\langle \texttt{lax}(Y) \rangle) \rangle
$$

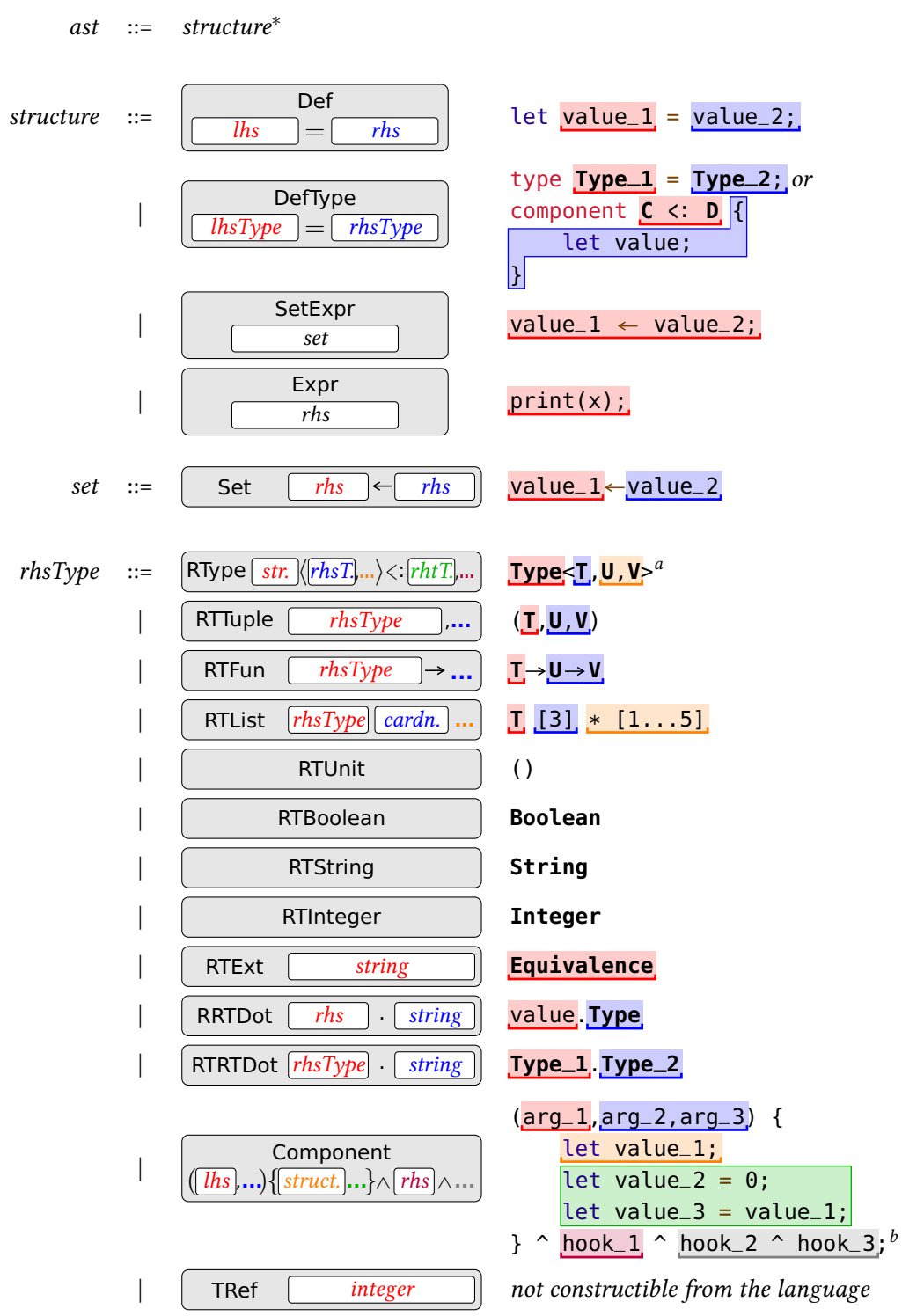
$$
\langle \texttt{tcall}(X, Y) \rangle \ ::= \ X \ \langle \texttt{parameter\_list}(\langle \texttt{lax}(Y) \rangle) \rangle
$$

$$
\langle \texttt{z3\_parameters} \rangle \ ::= \ \left\langle \texttt{inpar}\left(\langle \texttt{tpar}(\langle \texttt{lhs\_id} \rangle, \top) \rangle^{+}_{\texttt{,}} \right) \right\rangle
$$

$$
\begin{array}{rcl}
\langle \texttt{z3\_property} \rangle & ::= & \texttt{for} \ \texttt{all} \ \langle \texttt{z3\_parameters} \rangle \ \texttt{\{} \ \langle \texttt{rhs}(\top) \rangle^{*} \ \texttt{\}} \\
 & | & \texttt{exists} \ \langle \texttt{z3\_parameters} \rangle \ \texttt{\{} \ \langle \texttt{rhs}(\top) \rangle^{*} \ \texttt{\}}
\end{array}
$$

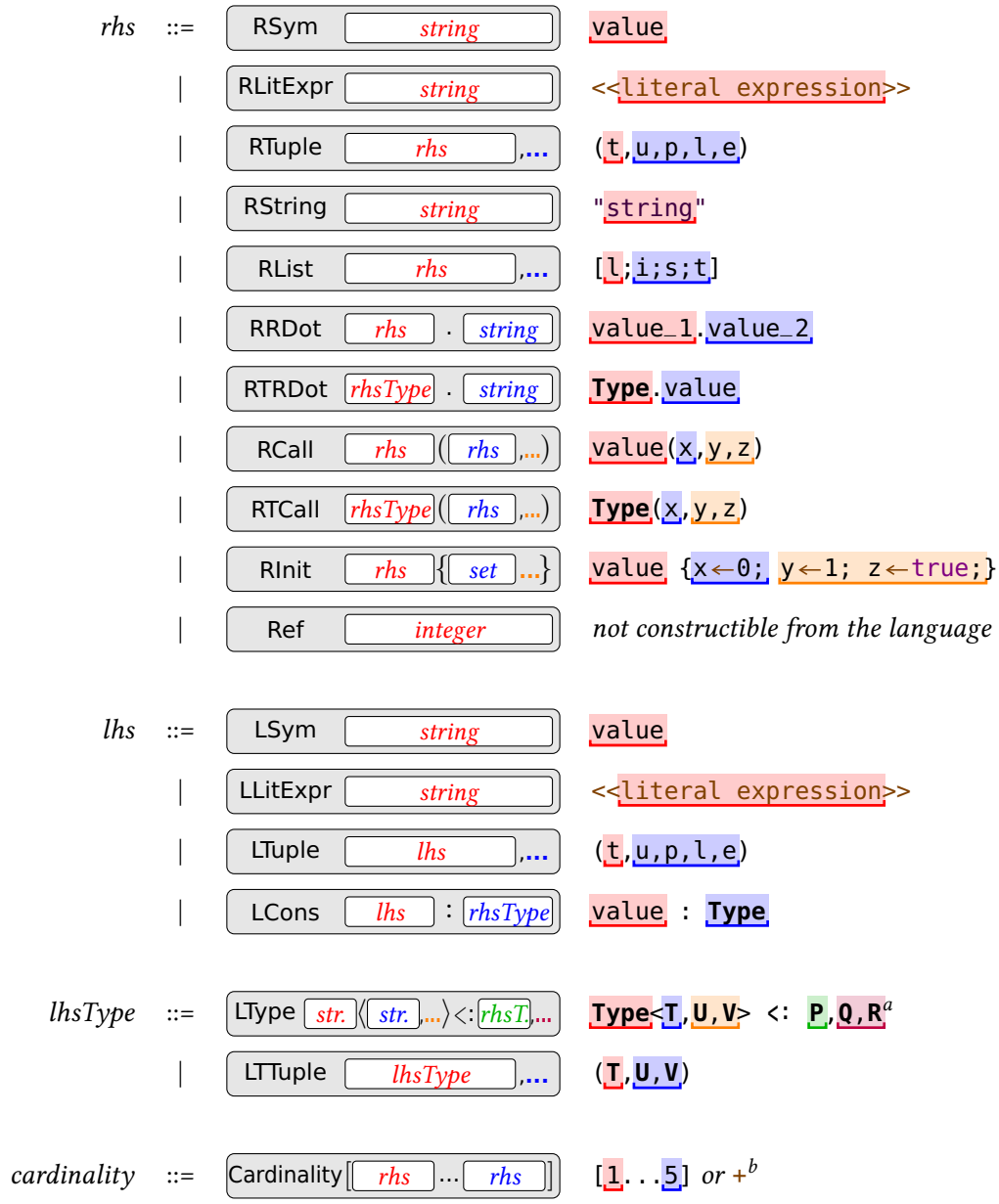
C.2 AST construction rules

Here, we show the different elements constituting CL/I's AST, along with examples to produce them in CL/I. The entry point of the AST is *ast*.



[a] Templating is optional and the subtyping part is not constructible from the language
[b] Arguments and hooks are optional

Table C.1: Presentation of CL/I's AST, along with a few examples to build each node



[a] Templating and subtyping are optional
[b] Transformed to Cardinality [RSym ("1") ... ∞]

Table C.1 *(continued)*: Presentation of CL/I's AST, along with a few examples to build each node

C.3 Transformation from CL/I's AST to the CLIR

In this last section, we present the semantics of the transformation from the CL/I language to its intermediate representation (CLIR), briefly presented with an example in section VI.2.4. First, we define four environments:

- $\mathcal{E}_V : S \to \#_V$, the value name environment, which maps the names of defined values to their symbols in $\mathcal{S}_V$;
- $\mathcal{E}_T : S \to \#_T$, the type name environment, which maps the names of defined types to their symbols in $\mathcal{S}_T$;
- $\mathcal{S}_V : \#_V \to V \times (S \to \#_V) \times \#_T$, the value symbol environment, which maps value symbols to their value, attributes and type triples;
- $\mathcal{S}_T : \#_T \to T \times (S \to \#_V)$, the type symbol environment, which maps type symbols to their type and attributes pairs;

We define the operator $\lhd$ to merge two environments, keeping the values from the right operand if they are defined in both environments:

$$\mathcal{X} \lhd \mathcal{X}' = \left\{ x \mapsto \begin{bmatrix} \mathcal{X}'(x) & \text{if } x \in \operatorname{dom}(\mathcal{X}') \\ \mathcal{X}(x) & \text{otherwise} \end{bmatrix} \middle| \; x \in \operatorname{dom}(\mathcal{X}) \cup \operatorname{dom}(\mathcal{X}') \right\}$$

We define the following zero elements:

$$\begin{aligned} 0_v &= (?, \varnothing, ?) \\ 0_t &= (?, \varnothing) \end{aligned}$$

We define the following reductions:

- $\Rightarrow_a$, the CLIR reduction, which reduces CL/I's AST to the CLIR;
- $\Rightarrow_s$, the environment reduction, which encodes changes in the environments;
- $\Rightarrow_e$, the value reduction, which reduces CL/I's expressions to CLIR symbols.

Finally, we define the function fresh, such that fresh $(\mathcal{S}_V)$ (respectively fresh $(\mathcal{S}_T)$) gives a fresh value (respectively type) symbol, that is, a new symbol not defined in $\mathcal{S}_V$ (respectively $\mathcal{S}_T$).

C.3.1 AST

A CL/I model is either the empty model ε or a sequence of *structures*, for which environments are successively passed from one to the next:

$$\textsc{Empty}\ \frac{}{\begin{aligned} \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \varepsilon \quad &\Rightarrow_a \quad \mathcal{S}_V, \mathcal{S}_T \\ &\Rightarrow_s \quad \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T \end{aligned}}$$

$$\textsc{Seq}\ \frac{\mathcal{E}^0_{V,T} = \mathcal{E}_{V,T} \qquad \mathcal{S}^0_{V,T} = \mathcal{S}_{V,T} \qquad \vdash \bigtriangleup_{k=0}^{i-1} \mathcal{E}^k_V, \bigtriangleup_{k=0}^{i-1} \mathcal{E}^k_T, \mathcal{S}^{i-1}_V, \mathcal{S}^{i-1}_T, t_i \Rightarrow_s \mathcal{E}^i_V, \mathcal{E}^i_T, \mathcal{S}^i_V, \mathcal{S}^i_T}{\begin{aligned} \vdash \mathcal{E}^0_V, \mathcal{E}^0_T, \mathcal{S}^0_V, \mathcal{S}^0_T, t_1; \ldots; t_n \quad &\Rightarrow_a \quad \mathcal{S}^n_V, \mathcal{S}^n_T \\ &\Rightarrow_s \quad \bigtriangleup_{i=1}^{n} \mathcal{E}^i_V, \bigtriangleup_{i=1}^{n} \mathcal{E}^i_T, \mathcal{S}^n_V, \mathcal{S}^n_T \end{aligned}}$$

It should be noted that name environments are synthetized, because of the local nature of names. The following rules only produce name environments with new definitions and declarations. On the other hand, symbol environments are both inherited and synthetized, because of the global nature of symbols. The following rules produce full symbol environments (these environments are actually mutable in our implementation).

C.3.2 Structure

Structures are the main element of CL/I models. They can be value definitions, type definitions, value assignments and expressions.

Def

Lhs values can be declared without giving them an explicit value. A fresh value symbol is created:

$$\textsc{Decl}\ \frac{s \notin \mathrm{dom}\,(\mathcal{E}_V) \qquad \#i = \mathrm{fresh}\,(\mathcal{S}_V)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Def}\ \boxed{\text{LSym}\ s} = \boxed{\times}} \Rightarrow_s \{s \mapsto \#i\}, \varnothing, \mathcal{S}_V \lhd \{\#i \mapsto 0_v\}, \mathcal{S}_T}$$

Lhs values can be defined equal to *rhs* values. A fresh value symbol is created:

$$\textsc{Def}\ \frac{s \notin \mathrm{dom}\,(\mathcal{E}_V) \qquad \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_v \Rightarrow_e (v, a, t) \qquad \#i = \mathrm{fresh}\,(\mathcal{S}_V)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Def}\ \boxed{\text{LSym}\ s} = \boxed{e_v}} \Rightarrow_s \{s \mapsto \#i\}, \varnothing, \mathcal{S}_V \lhd \{\#i \mapsto (v, a, t)\}, \mathcal{S}_T}$$

Tuples of *lhs* values can be defined equal to tuples of *rhs* values, provided their sizes match (note that this rule is recursive):

$$\textsc{Def(Tuple/Tuple)}\ \frac{N = [\![1, n]\!] \qquad \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Def}\ \boxed{e_{v_1}} = \boxed{e'_{v_1}}}; \ldots; \boxed{\text{Def}\ \boxed{e_{v_n}} = \boxed{e'_{v_n}}} \Rightarrow_s \mathcal{E}'_V, \mathcal{E}'_T, \mathcal{S}'_V, \mathcal{S}'_T}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Def}\ \boxed{\text{LTuple}(e_{v_i})_{i \in N}} = \boxed{\text{RTuple}(e'_{v_i})_{i \in N}}} \Rightarrow_s \mathcal{E}'_V, \mathcal{E}'_T, \mathcal{S}'_V, \mathcal{S}'_T}$$

$$\textsc{Def(Tuple/Tuple)Err}\ \frac{\#N \neq \#N'}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Def}\ \boxed{\text{LTuple}(e_{v_i})_{i \in N}} = \boxed{\text{RTuple}(e'_{v_i})_{i \in N'}}} \Rightarrow_s \textsc{SizeMismatch}}$$

Tuples of *rhs* values can only be assigned to tuples of *lhs* values:

$$\textsc{Def(}\neg\textsc{Tuple/Tuple)Err}\ \frac{e_v \neq \boxed{\text{LTuple}\ \boxed{\cdot}}}{\vdash \mathcal{E}^0_V, \mathcal{E}^0_T, \mathcal{S}^0_V, \mathcal{S}^0_T, \boxed{\text{Def}\ \boxed{e_v} = \boxed{\text{RTuple}(e_{v_i})_{i \in N}}} \Rightarrow_s \textsc{UnassignableTuple}}$$

In parallel, tuples of *lhs* values can be defined equal to lists of *rhs* values, provided their sizes match (this rule is also recursive):

$$\textsc{Def(Tuple/List)}\ \frac{N = [\![1, n]\!] \qquad \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Def}\ \boxed{e_{v_1}} = \boxed{e'_{v_1}}}; \ldots; \boxed{\text{Def}\ \boxed{e_{v_n}} = \boxed{e'_{v_n}}} \Rightarrow_s \mathcal{E}'_V, \mathcal{E}'_T, \mathcal{S}'_V, \mathcal{S}'_T}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Def}\ \boxed{\text{LTuple}(e_{v_i})_{i \in N}} = \boxed{\text{RList}(e'_{v_i})_{i \in N}}} \Rightarrow_s \mathcal{E}'_V, \mathcal{E}'_T, \mathcal{S}'_V, \mathcal{S}'_T}$$

$$\textsc{Def(Tuple/List)Err}\ \frac{\#N \neq \#N'}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Def}\ \boxed{\text{LTuple}(e_{v_i})_{i \in N}} = \boxed{\text{RList}(e'_{v_i})_{i \in N'}}} \Rightarrow_s \textsc{SizeMismatch}}$$

Redefinitions are not allowed:

$$\textsc{RedefErr}\ \frac{s \in \mathrm{dom}\,(\mathcal{E}_V)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Def}\ \boxed{\text{LSym}\ s} = \boxed{e_v}} \Rightarrow_s \textsc{Redefinition}}$$

DefType

Lhs types can be declared without giving them an explicit type. For each type template, a fresh empty type symbol is created:

$$\textsc{DeclType}\ \frac{s \notin \mathrm{dom}\,(\mathcal{E}_T) \qquad \mathtt{T\#}i = \mathrm{fresh}\,(\mathcal{S}_T) \qquad \mathtt{T\#}k_i = \mathrm{fresh}\,(\mathcal{S}_T) \qquad \mathcal{S}'_T = \mathcal{S}_T \lhd \{\mathtt{T\#}k_1 \mapsto 0_t, \ldots, \mathtt{T\#}k_n \mapsto 0_t\}}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{DefType}\ \boxed{\text{LType}\ s\ \langle s_1, \ldots, s_n \rangle} = \boxed{\times}} \Rightarrow_s \varnothing, \{s \mapsto \mathtt{T\#}i\}, \mathcal{S}_V, \mathcal{S}'_T \lhd \{\mathtt{T\#}i \mapsto 0_t\}}$$

Lhs types can be defined equal to *rhs* types. Type templates declared in the *lhs* type can be used by the *rhs* type:

$$\textsc{DefType}\ \frac{\begin{array}{rl} \mathtt{T\#}i = \mathrm{fresh}\,(\mathcal{S}_T) & \mathcal{E}'_T = \{s_1 \mapsto \mathtt{T\#}k_1, \ldots, s_n \mapsto \mathtt{T\#}k_n\} \\ \mathtt{T\#}k_i = \mathrm{fresh}\,(\mathcal{S}_T) & \mathcal{S}'_T = \mathcal{S}_T \lhd \{\mathtt{T\#}k_1 \mapsto 0_t, \ldots, \mathtt{T\#}k_n \mapsto 0_t\} \\ s \notin \mathrm{dom}\,(\mathcal{E}_T) & \vdash \mathcal{E}_V, \mathcal{E}_T \lhd \mathcal{E}'_T, \mathcal{S}_V, \mathcal{S}'_T, e_t \Rightarrow_e (\tau, a) \end{array}}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{DefType}\ \boxed{\text{LType}\ s\ \langle s_1, \ldots, s_n \rangle} = \boxed{e_t}} \Rightarrow_s \varnothing, \{s \mapsto \mathtt{T\#}i\}, \mathcal{S}_V, \mathcal{S}'_T \lhd \{\mathtt{T\#}i \mapsto (\tau, a)\}}$$

As with values, tuples of *lhs* types can be defined equal to tuples of *rhs* types, provided their sizes match (note that this rule is recursive):

$$\textsc{DefType(Tuple/Tuple)}\ \frac{N = [\![1, n]\!] \qquad \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{DefType}\ \boxed{e_{t_1}} = \boxed{e'_{t_1}}}; \ldots; \boxed{\textsf{DefType}\ \boxed{e_{t_n}} = \boxed{e'_{t_n}}} \Rightarrow_s \mathcal{E}'_V, \mathcal{E}'_T, \mathcal{S}'_V, \mathcal{S}'_T}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{DefType}\ \boxed{\textsf{LTTuple}(e_{v_i})_{i \in N}} = \boxed{\textsf{RTTuple}(e'_{v_i})_{i \in N}}} \Rightarrow_s \mathcal{E}'_V, \mathcal{E}'_T, \mathcal{S}'_V, \mathcal{S}'_T}$$

$$\textsc{DefType(Tuple/Tuple)Err}\ \frac{\#N \neq \#N'}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{DefType}\ \boxed{\textsf{LTTuple}(e_{t_i})_{i \in N}} = \boxed{\textsf{RTTuple}(e'_{t_i})_{i \in N'}}} \Rightarrow_s \textsc{SizeMismatch}}$$

Tuples of *rhs* types can only be assigned to tuples of *lhs* types:

$$\textsc{DefType}(\neg\textsc{Tuple/Tuple})\textsc{Err}\ \frac{e_t \neq \boxed{\textsf{LTTuple}\ \boxed{\cdot}}}{\vdash \mathcal{E}^0_V, \mathcal{E}^0_T, \mathcal{S}^0_V, \mathcal{S}^0_T, \boxed{\textsf{DefType}\ \boxed{e_t} = \boxed{\textsf{RTTuple}(e_{t_i})_{i \in N}}} \Rightarrow_s \textsc{UnassignableTypeTuple}}$$

Redefinitions are not allowed:

$$\textsc{RedefTypeErr}\ \frac{s \in \mathrm{dom}\,(\mathcal{E}_T)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{DefType}\ \boxed{\textsf{LType}\ s} = \boxed{e_t}} \Rightarrow_s \textsc{Redefinition}}$$

SetExpr

Rhs values can be assigned to previously declared *Rhs* values:

$$\textsc{SetExpr}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_v \Rightarrow_e (\textsf{Ref}\ \texttt{i}, \cdot, \cdot) \qquad \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e'_v \Rightarrow_e (v, a, t)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{SetExpr}\ \boxed{e_v} \leftarrow \boxed{e'_v}} \Rightarrow_s \varnothing, \varnothing, \mathcal{S}_V \lhd \{\texttt{\#i} \mapsto (v, a, t)\}, \mathcal{S}_T}$$

$$\textsc{SetExprNonDeclErr}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_v \Rightarrow_e (v, \cdot, \cdot) \qquad v \neq \textsf{Ref}\ \cdot}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{SetExpr}\ \boxed{e_v} \leftarrow \boxed{e'_v}} \Rightarrow_s \textsc{NotPreviouslyDeclared}}$$

Expr

No semantic transformation happens for expressions at the top level.

C.3.3 Right values

We only support a subset of the language constructs for values in the following semantic rules.

RSym

Rhs values can be previously declared symbols (we consider integers and booleans to be predefined built-in symbols in CL/I):

$$\textsc{RSym}\ \frac{s \in \mathrm{dom}\,(\mathcal{E}_V) \qquad \mathcal{E}_V\,(s) = \#i}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RSym}\ \boxed{s}} \Rightarrow_e (\mathsf{Ref}\ i, \varnothing, ?)}$$

$$\textsc{RBoolean}\ \frac{s \text{ represents a boolean } b}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RSym}\ \boxed{s}} \Rightarrow_e (\mathsf{Boolean}\ b, \varnothing, \mathsf{Boolean})}$$

$$\textsc{RInteger}\ \frac{s \text{ represents an integer } i}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RSym}\ \boxed{s}} \Rightarrow_e (\mathsf{Integer}\ i, \varnothing, \mathsf{Integer})}$$

$$\textsc{RSymErr}\ \frac{s \notin \mathrm{dom}\,(\mathcal{E}_V) \qquad s \text{ does not represent a boolean or an integer}}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RSym}\ \boxed{s}} \Rightarrow_e \textsc{NotFound}}$$

RString

Rhs values can be strings:

$$\textsc{RString}\ \frac{}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RString}\ \boxed{s}} \Rightarrow_e (\mathsf{String}\ s, \varnothing, \mathsf{String})}$$

RList

Rhs values can be homogeneous lists of declared values:

$$\textsc{RList}\ \frac{\begin{array}{c}\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RSym}\ \boxed{s_1}} \Rightarrow_e (\mathsf{Ref}\ i_1, \cdot, \cdot) \\ \vdots \\ \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RSym}\ \boxed{s_n}} \Rightarrow_e (\mathsf{Ref}\ i_n, \cdot, \cdot) \\ \boldsymbol{t}\,(\mathcal{S}_V\,(i_1)) = \ldots = \boldsymbol{t}\,(\mathcal{S}_V\,(i_n)) = \mathsf{T\#}j\end{array}}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RList}\ \boxed{\mathsf{RSym}\ s_1, \ldots, \mathsf{RSym}\ s_n}} \Rightarrow_e (\mathsf{List}\,(\mathsf{Ref}\ i_1, \ldots, \mathsf{Ref}\ i_n), \varnothing, \mathsf{List}\,\langle \mathsf{T\#}j \rangle)}$$

$$\text{RListTypeErr}\ \frac{\begin{array}{c}\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RSym}\ \boxed{s_1}} \Rightarrow_e (\text{Ref}\ i_1, \cdot, \cdot) \\ \vdots \\ \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RSym}\ \boxed{s_n}} \Rightarrow_e (\text{Ref}\ i_n, \cdot, \cdot) \\ \neg\left(\boldsymbol{t}\left(\mathcal{S}_V\left(i_1\right)\right) = \ldots = \boldsymbol{t}\left(\mathcal{S}_V\left(i_n\right)\right) = \text{T\#}j\right)\end{array}}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RList}\ \boxed{\text{RSym}\ s_1, \ldots, \text{RSym}\ s_n}} \Rightarrow_e \textsc{InhomogeneousList}}$$

$$\text{RListNotDeclErr}\ \frac{\begin{array}{c}\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RSym}\ \boxed{s_1}} \Rightarrow_e (v_1, \cdot, \cdot) \\ \vdots \\ \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RSym}\ \boxed{s_n}} \Rightarrow_e (v_n, \cdot, \cdot) \\ \neg\left(v_1 = \text{Ref}\ \cdot \wedge \ldots \wedge v_n = \text{Ref}\ \cdot\right)\end{array}}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RList}\ \boxed{\text{RSym}\ s_1, \ldots, \text{RSym}\ s_n}} \Rightarrow_e \textsc{NotSupported}}$$

RRDot

Rhs values can be *rhs* values' attributes. The value whose attribute is retrieved must be correctly defined and have the corresponding attribute.

$$\text{RRDot}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_v \Rightarrow_e (\text{Ref}\ i, \cdot, \cdot) \qquad s \in \text{dom}\left(\boldsymbol{a}\left(\mathcal{S}_V\left(i\right)\right)\right)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RRDot}\ \boxed{e_v}\ \cdot\ \boxed{s}} \Rightarrow_e \boldsymbol{a}\left(S_V\left(i\right)\right)\left(s\right)}$$

$$\text{RRDotErr}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_v \Rightarrow_e (\text{Ref}\ i, \cdot, \cdot) \qquad s \notin \text{dom}\left(\boldsymbol{a}\left(\mathcal{S}_V\left(i\right)\right)\right)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RRDot}\ \boxed{e_v}\ \cdot\ \boxed{s}} \Rightarrow_e \textsc{NotFound}}$$

$$\text{RRDotNotDefErr}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_v \Rightarrow_e (v, \cdot, \cdot) \qquad v \neq \text{Ref}\ \cdot}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RRDot}\ \boxed{e_v}\ \cdot\ \boxed{s}} \Rightarrow_e \textsc{NotSupported}}$$

C.3.4 RTRDot

Rhs values can be *rhs* types' attributes. The type whose attribute is retrieved must be correctly defined and have the corresponding attribute.

$$\text{RTRDot}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_t \Rightarrow_e (\text{TRef}\ i, \cdot) \qquad s \in \text{dom}\left(\boldsymbol{a}\left(\mathcal{S}_T\left(i\right)\right)\right)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RTRDot}\ \boxed{e_t}\ \cdot\ \boxed{s}} \Rightarrow_e \boldsymbol{a}\left(S_T\left(i\right)\right)\left(s\right)}$$

$$\text{RTRDotErr}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_t \Rightarrow_e (\text{TRef}\ i, \cdot) \qquad s \notin \text{dom}\left(\boldsymbol{a}\left(\mathcal{S}_T\left(i\right)\right)\right)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RTRDot}\ \boxed{e_t}\ \cdot\ \boxed{s}} \Rightarrow_e \textsc{NotFound}}$$

$$\textsc{RTRDotNotDefErr}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_t \Rightarrow_e (t, \cdot) \qquad t \neq \mathsf{TRef}\ \cdot}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RTRDot}\ \boxed{e_t}\ \cdot\ \boxed{s}} \Rightarrow_e \textsc{NotSupported}}$$

RCall

Rhs values can be calls to *rhs* values (we only support a few predefined operators):

$$\textsc{RCall}\ \frac{s \in \{=, \mathsf{and}, \Rightarrow, !, \mathsf{has...}\} \qquad \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_{v_i} \Rightarrow_e (v_i, \cdot, \cdot)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RCall}\ \boxed{\mathsf{RSym}\ s}\ (\boxed{e_{v_1}, ..., e_{v_n}})} \Rightarrow_e (\mathsf{App}\ \ (s, (v_1, ..., v_n)), \varnothing, ?)}$$

$$\textsc{RCallErr}\ \frac{s \notin \{=, \mathsf{and}, \Rightarrow, !, \mathsf{has...}\}}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RCall}\ \boxed{\mathsf{RSym}\ s}\ (\boxed{e_{v_1}, ..., e_{v_n}})} \Rightarrow_e \textsc{NotSupported}}$$

$$\textsc{RCallNotCallableErr}\ \frac{e_v \neq \boxed{\mathsf{RSym}\ \boxed{\cdot}}}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RCall}\ \boxed{e_v}\ (\boxed{e_{v_1}, ..., e_{v_n}})} \Rightarrow_e \textsc{NotCallable}}$$

RTCall

Rhs values can be calls to *rhs* types (we do not support call arguments yet):

$$\textsc{RTCall}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_t \Rightarrow_e (\mathsf{TRef}\ \mathtt{i}, \cdot)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RTCall}\ \boxed{e_t}\ (\boxed{\varnothing})} \Rightarrow_e (\mathsf{Instance}, a\ (\mathsf{T\#}\mathtt{i}), \mathsf{T\#}\mathtt{i})}$$

$$\textsc{RTCallErr}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_t \Rightarrow_e (t, \cdot) \qquad t \neq \mathsf{TRef}\ \cdot}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RTCall}\ \boxed{e_t}\ (\boxed{\varnothing})} \Rightarrow_e \textsc{NotSupported}}$$

C.3.5 RInit

Finally, *rhs* values can be component initializations. When initializing components, two reductions are used, as new value symbols are defined for initialized attributes:

$$\textsc{RInit}\ \frac{\begin{array}{c}\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_v \Rightarrow_e (v, a, t) \qquad \#\mathtt{j}_i = \mathrm{fresh}\,(\mathcal{S}_V) \qquad \vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_{v_i} \Rightarrow_e (v_i, a_i, t_i) \\ s_i \in \mathrm{dom}\,(a) \qquad a' = a \lhd \{s_i \mapsto \#\mathtt{j}_i \mid i \in [\![1, n]\!]\} \qquad \mathcal{S}'_V = \mathcal{S}_V \lhd \{\#\mathtt{j}_i \mapsto (v_i, a_i, t_i) \mid i \in [\![1, n]\!]\}\end{array}}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RInit}\ \boxed{e_v}\ \{\boxed{\mathsf{Rsym}\ s_1 \leftarrow e_{v_1}, ..., \mathsf{Rsym}\ s_n \leftarrow e_{v_n}}\}} \begin{array}{ll}\Rightarrow_s & \varnothing, \varnothing, \mathcal{S}'_V, \mathcal{S}_T \\ \Rightarrow_e & (v, a', t)\end{array}}$$

$$\textsc{RInitErr}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_v \Rightarrow_e (v, a, t) \qquad \neg\,(s_1 \in \mathrm{dom}\,(a) \wedge ... \wedge s_n \in \mathrm{dom}\,(a))}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\mathsf{RInit}\ \boxed{e_v}\ \{\boxed{\mathsf{Rsym}\ s_1 \leftarrow e_{v_1}, ..., \mathsf{Rsym}\ s_n \leftarrow e_{v_n}}\}} \Rightarrow_e \textsc{NotFound}}$$

$$\textsc{RInitNotDefErr}\ \frac{\neg\left(e_{v_1} = \boxed{\textsf{RSym}\ \boxed{\cdot}} \wedge \ldots \wedge e_{v_n} = \boxed{\textsf{RSym}\ \boxed{\cdot}}\right)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{RInit}\ \boxed{e_v}\ \left\{\boxed{e_{v_1}, \ldots, e_{v_n}}\right\}} \Rightarrow_e \textsc{NotSupported}}$$

C.3.6 Right types

RType

Rhs types can be previously defined type symbols (we do not support templating yet):

$$\textsc{RType}\ \frac{s \in \mathrm{dom}\,(\mathcal{E}_T) \qquad \mathcal{E}_T\,(s) = \textsf{T\#}i}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{RType}\ \boxed{s}} \Rightarrow_e (\textsf{TRef}\ i, \varnothing)}$$

$$\textsc{RTypeErr}\ \frac{s \notin \mathrm{dom}\,(\mathcal{E}_T)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{RType}\ \boxed{s}} \Rightarrow_e \textsc{NotFound}}$$

RTList

Rhs types can be list types of previously defined type symbols (we do not support cardinalities yet):

$$\textsc{RTList}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{RType}\ \boxed{s}} \Rightarrow_e (\textsf{TRef}\ i, \cdot)}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{RTList}\ \boxed{\textsf{RType}\ s}} \Rightarrow_e (\textsf{List}\ \langle \textsf{T\#}i \rangle, \varnothing)}$$

$$\textsc{RTListErr}\ \frac{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, e_t \Rightarrow_e (\tau, \cdot) \qquad \tau \neq \textsf{TRef}\ \cdot}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{RTList}\ \boxed{e_t}} \Rightarrow_e \textsc{NotSupported}}$$

RTUnit

Rhs types can be the Unit type:

$$\textsc{RTUnit}\ \frac{}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{RTUnit}} \Rightarrow_e (\textsf{Unit}, \varnothing)}$$

RTBoolean

Rhs types can be the Boolean type:

$$\textsc{RTBoolean}\ \frac{}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{RTBoolean}} \Rightarrow_e (\textsf{Boolean}, \varnothing)}$$

RTString

Rhs types can be the String type:

$$\textsc{RTString}\ \frac{}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\textsf{RTString}} \Rightarrow_e (\textsf{String}, \varnothing)}$$

RTInteger

Rhs types can be the Integer type:

$$\textsc{RTInteger}\ \frac{}{\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{RTInteger}} \Rightarrow_e (\text{Integer}, \varnothing)}$$

Component

Components are made abstract (we do not support hooks yet):

$$\textsc{Component}\ \frac{\begin{array}{c}\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Def}\ \boxed{e_{v_1}} = \boxed{\times}}; \ldots; \boxed{\text{Def}\ \boxed{e_{v_n}} = \boxed{\times}} \Rightarrow_s \mathcal{E}'_V, \mathcal{E}'_T, \mathcal{S}'_V, \mathcal{S}'_T \\ \vdash \mathcal{E}'_V + \mathcal{E}_V, \mathcal{E}_T \lhd \mathcal{E}'_T, \mathcal{S}'_V, \mathcal{S}'_T, t_1; \ldots; t_n \Rightarrow_s \mathcal{E}''_V, \mathcal{E}''_T, \mathcal{S}''_V, \mathcal{S}''_T\end{array}}{\begin{array}{lll}\vdash \mathcal{E}_V, \mathcal{E}_T, \mathcal{S}_V, \mathcal{S}_T, \boxed{\text{Component}\ (\boxed{e_{v_1}, \ldots, e_{v_n}})\{\boxed{t_1; \ldots; t_n}\}} & \Rightarrow_s & \varnothing, \varnothing, \mathcal{S}''_V, \mathcal{S}''_T \\ & \Rightarrow_e & (\text{Abstract}, \mathcal{E}''_V)\end{array}}$$

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Modélisation d'infrastructure informatique pour l'identification et la prévention des risques

Mot clés : Modélisation d'infrastructure, gestion des risques, fédération de modèles, vérification formelle, sûreté et sécurité

Résumé : Les infrastructures informatiques font partie de notre vie quotidienne et ont gagné ces dernières décennies une importance capitale. Au cœur de notre système bancaire, de nos transports et de nos hôpitaux, leur omniprésence et les implications liées à leur défaillance en font une cible privilégiée pour les attaquants. Au-delà du risque de sécurité, ces infrastructures sont soumises à un ensemble de risques de sûreté, allant de l'aléa climatique à l'incendie industriel, en passant par la simple usure des composants mécaniques. Ces risques, bien que prévisibles, ne sont pas toujours pris en compte par les entreprises et peuvent mener à des conséquences parfois catastrophiques.

Nos travaux se positionnent tout au long du processus de gestion des risques. Nous défendons la thèse qu'une modélisation correcte et exhaustive des infrastructures informatiques permet de déduire un ensemble de propriétés de sûreté et de sécurité constituant une analyse des risques satisfaisante du système d'information étudié. Nous proposons un ensemble de recommandations et de méthodes pour procéder collaborativement à la modélisation des infrastructures et à l'étude du risque. Enfin, nous présentons un langage de description d'infrastructures formellement spécifié, CL/I, faisant le lien entre modèles d'infrastructures, outils de vérification formelle et supervision fonctionnelle.

Title: IT infrastructure modeling for risk identification and prevention

Keywords: Infrastructure Modeling, Risk Management, Model federation, Formal Verification, Safety & Security

Abstract: IT infrastructures are part and parcel of our daily lives, and have become of vital importance over the last few decades. At the heart of our banking system, transport facilities and hospitals, their omnipresence and the implications of their failure make them a prime target for attackers. Beyond the security risk, these infrastructures are subject to a whole range of safety risks, from climatic events to industrial fires, to the sheer wear and tear of mechanical components. These risks, although foreseeable, are not always taken into account by companies, and can lead to potentially catastrophic consequences.

Our work encompasses the entire risk management process. We defend the thesis that correct and exhaustive modeling of infrastructures allows the deduction of a set of safety and security properties constituting a satisfying risk analysis of the information system under study. We propose a set of recommendations and methods to collaboratively conduct infrastructure modeling and risk analysis. Finally, we present a formally specified infrastructure description language, CL/I, linking infrastructure models, formal verification and functional supervision.