UNIVERSITE BRETAGNE LOIRE / MATHSTIC

ENSTA Bretagne

# THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
DE TECHNIQUES AVANCÉES BRETAGNE
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
*Mathématiques et Sciences et Technologies de l'Information et de la Communication*
Spécialité : informatique

Par

**Vincent LEILDÉ**

## Aide au diagnostic de vérification formelle de systèmes

**Thèse présentée et soutenue à l'ENSTA-BRETAGNE (BREST), le 14 Novembre 2019**
**Unité de recherche : Lab-STICC UMR CNRS6285**

**Rapporteurs avant soutenance :**
Reda BENDRAOU, Professeur des Universités, Sorbonne Université, Paris
Frédéric BONIOL, Professeur des Universités, ONERA, Toulouse

**Composition du Jury :**

| | |
|---|---|
| Président : | Isabelle BORNE, Professeur des Universités, Université Bretagne Sud, Vannes |
| Examinateurs : | Reda BENDRAOU, Professeur des Universités, Sorbonne Université, Paris |
| | Frédéric BONIOL, Professeur des Universités, ONERA, Toulouse |
| | Eric LE PORS, Docteur, Thales Systèmes Aéroportés, Brest |
| Dir. de thèse : | Philippe DHAUSSY, Professeur, ENSTA Bretagne, Brest |
| Co-encadrant : | Vincent RIBAUD, Maître de Conférences, Université Bretagne Occidentale, Brest |

# Remerciements

Je tiens d'abord à remercier mon directeur de thèse, Philippe Dhaussy pour m'avoir permis de réaliser cette thèse et pour sa confiance offerte ces trois années et les précédentes. J'ai pu apprécier sa vision du model checking, ainsi que sa générosité à partager ses connaissances.

Je remercie chaleureusement Vincent Ribaud pour m'avoir épaulé et encouragé. Sa disponibilité et sa générosité ont été capitales pour l'accomplissement de la thèse, je lui dois beaucoup.

Je remercie Frédéric Boniol et Reda Bendraou, pour avoir accepté de rapporter cette thèse. Merci pour le soutien, les conseils et l'intérêt porté à mon travail. Merci à Isabelle Borne, présidente du jury, ainsi qu'à Eric Le Pors qui m'a fait l'honneur de participer au jury. Je remercie Antoine Beugnard pour m'avoir suivi lors des comités de thèses.

Je n'aurais pas eu autant de plaisir à réaliser cette thèse, sans l'accueil de l'ensemble des membres de STIC SLS, en particulier Joël Champeau, à qui j'adresse mes sincères remerciements pour l'ensemble des années passées dans son équipe. Merci à toute l'équipe, pour les conseils avisés, le soutien moral et l'ambiance conviviale, essentielle à la motivation et au travail. Je remercie particulièrement Jean-Charles Roger, Bastien Drouot, Ciprian Teodorov, Luka Leroux, Faad Golrah, Pascale Gautron, Cyrielle Féron, Nicolas Sun, Hanna Boenning, Fadi Obeid. Sans oublier Michèle Hofmann et Annick Billon-Coat, pour leur disponibilité et leur aide quotidienne. Merci aussi à toute l'équipe d'Openflexo, en particulier Christophe Guychard, Sylvain Guérin, Antoine Beugnard, Fabien Dagnat, Gilles Godet et Philippe Bonnet, pour les moments et les idées partagées, et qui ont pu se diffuser dans la thèse.

Je souhaite exprimer toute ma reconnaissance à mes proches. Mon épouse, Anne-claire, pour sa patience, sa compréhension, et ses encouragements constants. Elle a assuré, d'une main de maître, la logistique familiale, me permettant de libérer du temps pour la thèse. Merci à mes parents, qui m'ont toujours encouragé à faire ce que je souhaitais et à donner le meilleur de moi-même.

Enfin, je souhaite remercier mes trois enfants, Alice, Jean et Mathilde, pour votre patience. Votre enthousiasme permanent m'a porté jusqu'au bout.

Merci à toutes les autres personnes que je n'ai pas cité et qui se reconnaîtront dans ces lignes.

# Table des Matières

Chapitre 1

# Introduction

## 1.1 Contexte

### 1.1.1 Améliorer la qualité des systèmes logiciels complexes

#### Introduction

Les systèmes logiciels critiques doivent relever des enjeux financiers, sécuritaires et vitaux. La découverte de dysfonctionnements liés à des défauts de conception ou de sécurité, a d'autant plus d'impact qu'elle est détectée tardivement. Il est donc primordial d'évaluer la sûreté et la sécurité d'un système bien avant son déploiement, comme au cours de la phase de conception. Pour y parvenir, les ingénieurs ont recours à différentes techniques du génie logiciel, en particulier la simulation et la vérification. Mais la correction de ces systèmes reste un problème complexe. Ceux-ci se composent généralement d'une multitude de processus concurrents qui interagissent. Leurs échanges sont caractérisés par leur non-déterminisme, c'est-à-dire l'incertitude quant à leur ordre d'exécution. Cette caractéristique conduit parfois à des situations imprévues et difficiles à diagnostiquer pour les concepteurs.

La notion de "génie logiciel" fût proposée dès 1968 [NR69], et définie comme "une discipline d'ingénierie qui concerne tous les aspects de la production de logiciels depuis les premières étapes de la spécification du système jusqu'à sa maintenance" [Som11]. Il est destiné à soutenir le développement de logiciels conséquents, plutôt que la programmation individuelle. Il comprend des techniques pour aider à la spécification, conception, validation et évolution d'un programme, dont aucune n'est normalement pertinente pour développement de logiciels par un seul individu.

En dehors de la programmation, les briques de base sont des activités abstraites produisant un logiciel [Som11] :

**La spécification** durant laquelle les clients et les ingénieurs définissent le système logiciel à produire et les contraintes sur son fonctionnement. Au cours de cette phase des exigences sont produites, dont certaines peuvent être exprimées par des propriétés formelles.

**La conception** durant laquelle le système logiciel est réalisé. Cette phase inclut la dé-

finition de modèles d'ordre structurel (comme des diagrammes de blocs SysML [1]), ou comportemental (tels que des machines à états UML [2]), décrivant comment le système modélisé se comporte.

**La vérification** durant laquelle le système logiciel est vérifié pour s'assurer qu'il est conforme aux spécifications et aux attentes du client. On distingue la vérification de la validation [Boe79]. La validation répond à la question "construisons-nous le bon produit ?", et donc de s'assurer que le logiciel répond aux attentes du client. La vérification elle, répond à la question "construisons-nous le produit correctement ?", et donc que les exigences fonctionnelles et non fonctionnelles sont correctement conçues et mises en œuvre.

### La vérification

La vérification est une activité fondamentale au sein du génie logiciel qui permet de renforcer la qualité du système à concevoir. Comme en atteste une étude de B.Cheng [CA07], il n'existe pas de réel consensus quant à ce processus. Malgré cela, de nombreuses approches et techniques existent. Il est par exemple possible de respecter des règles de conception, telles que des conventions de nommage (Camel case par exemple), ou bien de formatage (indentation par exemple), facilement automatisables grâce à des outils tels que Checkstyle [3] ou Findbugs [4]. Un autre exemple de bonne pratique est la relecture de code par des pairs (peer review).

Lorsque les systèmes sont complexes, ces mesures sont bien souvent insuffisantes. Le test est une méthode d'analyse complémentaire. Qu'il soit unitaire, d'intégration, du système complet, ou bien encore d'acceptance, il repose sur un même principe. On exécute le système dans des conditions bien définies et reproductibles, puis on compare les résultats obtenus aux résultats attendus, aussi appelés oracles. Les tests sont sujets à divers problèmes. D'une part, les écrire requiert une compréhension détaillée du système, et d'autre part, il n'y a pas de certitude quant à leur couverture. En effet, il n'est pas toujours possible de vérifier certaines propriétés, comme par exemple la terminaison. Et dans le cas où l'on cherche à modéliser un système non déterministe, deux exécutions avec le même jeu de données d'entrée peuvent produire des résultats différents. Dijkstra soulignait ainsi, “Testing can only show the presence of bugs, not their absence” [DDH72]. Comme en atteste une étude, les tests logiciels atteignent déjà 50 % du coût total de production [WL11], décrire des tests de manière exhaustive s'avère inatteignable. Alors que le test se limite à vérifier une portion des comportements possibles du système, d'autres moyens

1. https ://www.omg.org/spec/SysML/1.3/
2. https ://www.omg.org/spec/UML/2.5.1/
3. http ://checkstyle.sourceforge.net/
4. http ://findbugs.sourceforge.net/

sont nécessaires pour s'assurer de l'absence totale de bugs.

Les vérifications dites formelles permettent de résoudre une partie des problèmes restants.

**Définition 1.1.** *Les méthodes formelles* ont pour but d'établir l'exactitude d'un système suivant une rigueur mathématique. Elles intègrent la vérification très tôt dans le processus de conception, via une technique de vérification plus efficace et réduisant le temps de vérification [BK08].

Selon I.Sommerville [Som11], le point de départ de ce processus est la définition de spécifications formelles. Celles-ci sont habituellement développées au cours d'un processus planifié, de type cascade, au sein duquel le système est complètement spécifié avant son développement. La spécification formelle du système est développée après que les exigences aient été spécifiées, mais avant que le système détaillé soit conçu. Dans cette approche, un modèle mathématique d'une spécification du système peut être créé, puis raffiné au travers de transformations mathématiques conservant sa cohérence, jusqu'à l'obtention d'un code exécutable. Si les transformations mathématiques sont correctes, alors le programme généré est conforme à ses spécifications. Le modèle final est donc vérifié par construction. Un exemple d'une telle approche est la méthode B [Sch01], particulièrement adaptée au développement de systèmes critiques.

Si ces spécifications formelles permettent de vérifier la conception, elles permettent également de préciser et lever les ambiguïtés des caractéristiques du système. Par exemple quand des exigences sont décrites dans un langage naturel, certaines expressions non formelles peuvent être interprétées de façon différente [Som11].

### Le model checking

Les modèles formels décrivent le comportement possible du système de façon non ambiguë et mathématique. Ils constituent les fondements d'un ensemble de techniques de vérifications. Le model checking, initialement proposé par Clarke, Emerson et Sifakis [CES09], garantit formellement qu'une spécification (souvent exprimée dans de la logique temporelle), est effectivement respectée par le système conçu.

**Définition 1.2.** *Le model checking* est une technique de vérification formelle qui vise à explorer tous les états possibles du système, dans le but de vérifier que le modèle du système satisfait des propriétés exprimées formellement [BK08].

Alors que les spécifications de propriétés prescrivent ce que le système doit faire ou ne doit pas faire, le modèle de comportement décrit comment le système se comporte en réalité. La vérification par model checking consiste à s'assurer que les spécifications de propriétés sont bien satisfaites pour un modèle formel de comportement.

Il existe plusieurs types de propriétés. Elles sont qualifiées de propriété de sûreté (safety), quand elles énoncent qu'une situation ne pourra jamais se produire ("il n'existe pas de deadlock", ou bien "l'exclusion mutuelle est garantie"), ou de vivacité (liveness) quand elles énoncent qu'une situation finira par avoir lieu ("chaque processus accède à la section critique", ou bien "chaque requête est traitée").

Le model checker est l'outil qui réalise le model checking. Il examine l'ensemble des états du système et vérifie s'ils satisfont ou non les propriétés. Dans le cas où un état viole une propriété, le model checker fournit un contre-exemple, c'est-à-dire une trace partant d'un état initial et menant jusqu'à l'état d'erreur. Cette trace concrétise une exécution où le modèle a atteint un état indésirable.

La technique de model checking est mise en œuvre dans de nombreux outils, le plus connu étant probablement SPIN [Hol03]. Pendant longtemps, elle a été limitée par le problème dit de l'explosion de l'espace d'état, due à une combinatoire importante entre les différents processus qui interagissent, la rendant inefficace pour des gros systèmes. Au fil du temps, les outils se sont grandement améliorés, permettant leur usage à de plus larges échelles, et l'admission du model checking au sein de nombreuses équipes, comme chez Amazon Web Services (AWS), où les ingénieurs définissent, depuis 2011, des spécifications formelles de leurs systèmes critiques, afin d'y appliquer du model checking pour résoudre des problèmes de conception difficiles [New+14]. Parmi les optimisations qui permettent ces avancées, un exemple est l'approche proposée par l'environnement de model checking OBP-CDL [TLD14], qui grâce à la séparation du système et de l'environnement, permet de vérifier de larges systèmes.

### 1.1.2 Cadre de la thèse

#### Origines

L'approche traditionnellement admise par la communauté du model checking est celle définie dans [BK08]. À partir de spécifications de propriétés, un modèle est conçu, vérifié, puis le cas échéant analysé et corrigé.

**Phase de formalisation et modélisation** La formalisation et la modélisation du système sont réalisées en parallèle. Le système est modélisé en utilisant le langage accepté en entrée du model checker. Les modèles produits décrivent le comportement du système de manière précise et non ambiguë, généralement exprimés sous forme d'automates à états finis. À titre d'exemple, le model checker SPIN s'appuie sur un langage nommé PROMELA, tandis qu'OBP effectue ses analyses sur des modèles FIACRE. Des simulations permettent éventuellement une évaluation rapide des modèles produits, permettant ainsi d'éliminer des erreurs évidentes. Parallèlement, les propriétés à vérifier sont formalisées à l'aide d'un langage de spécification de pro-

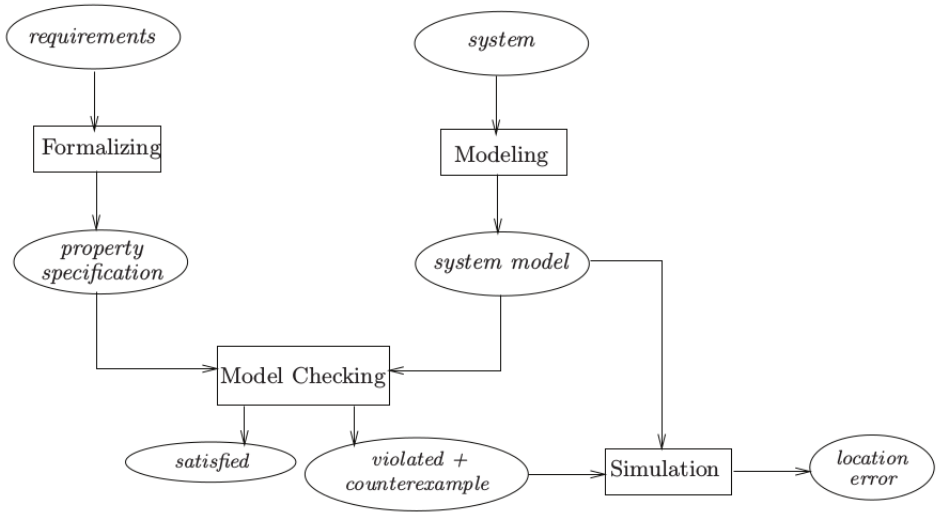


FIGURE 1.1 – Le processus traditionnel de model checking d'après C.Baier [BK08]

priétés. Elles doivent également être décrites de manière non ambiguë et précise. Le langage de spécification de propriétés est souvent de la logique temporelle [Pnu77], soit linéaire comme LTL ou arborescente comme CTL.

**Phase d'exécution** Le model checker est utilisé pour vérifier les propriétés sur le modèle du système.

**Phase d'analyse** Si les propriétés sont toutes satisfaites, le modèle est correct. Si une propriété est violée, il faut analyser le contre-exemple généré afin d'en déterminer les causes. Le modèle, la conception ou les propriétés seront ensuite corrigés [Avi+04], et l'ensemble de la procédure répétée. Il arrive que dans certains cas, quand la combinatoire est trop élevée, la capacité mémoire soit dépassée, on parle alors du problème de l'explosion de l'espace d'état. Dans ce cas, il est préconisé de revoir le modèle afin de le rendre exploitable.

### Objectifs

Dès lors que la vérification a détectée une propriété violée, les ingénieurs procèdent à la phase d'analyse, aussi appelée *diagnostic*. Notre objectif est de faciliter le diagnostic. Dans le cadre du model checking, le diagnostic est un processus qui, à partir de symptômes matérialisés par des traces menant à des propriétés violées, cherche à identifier les causes dans les spécifications (exigences, design, modèle ...). Une fois ces causes identifiées, des corrections seront appliquées, et le processus de vérification réexecuté afin de s'assurer que la correction de la faute est effective et n'a pas engendré d'autres problèmes (vérification de non-régression).

**Hypothèses**

Compte tenu de la richesse des formalismes et des outils utilisés tout au long de la conception d'un système logiciel, nous partons des hypothèses suivantes :

— Le système doit être décrit sous la forme d'un système de transition d'états étiquetés (Labelled Transition System ou LTS), c'est-à-dire un graphe orienté composé d'états liés entre eux grâce à des transitions étiquetées. Si cette hypothèse exclut les systèmes dont le comportement n'est pas descriptible par un LTS, par exemple des systèmes algorithmiques basés sur des équations différentielles, elle couvre une partie importante des systèmes industriels.

— La vérification est réalisée par model checking. Celle-ci permet la production et l'exploration de l'espace d'état formé par le LTS. Nous réduisons notre approche à l'utilisation de model checkers parfaitement fonctionnels, c'est-à-dire n'ayant aucun bug, de sorte qu'il n'y ait pas de faux positifs possibles. Nous supposons aussi que l'ensemble des outils ou transformations tierces manipulés durant la thèse sont exempts de défauts.

— Les propriétés à vérifier sont formalisées dans une logique temporelle, condition nécessaire à la vérification par model checking. De plus, leur validité n'est pas remise en cause.

— Les activités que nous cherchons à faciliter sont des activités de conception et de diagnostic, qui sont elles-mêmes des activités de résolution de problèmes.

— La réutilisation de produits et de processus favorise la qualité des systèmes. Une assistance à la conception et au diagnostic doit s'appuyer sur l'expérience.

## 1.2 Enjeux

Nous avons identifié plusieurs enjeux qui doivent être relevés pour faciliter le diagnostic.

### 1.2.1 Mettre au point des applications vérifiées

Pour vérifier une application le diagnosticien doit mener différentes actions. Une pratique courante est l'acquisition de connaissance sur le modèle par stimulation ou simulation. La stimulation concerne la modification de certains paramètres du système dans le but d'en analyser le fonctionnement grâce à ses réactions. Par exemple, le diagnosticien peut modifier progressivement la taille d'une *Fifo* pour en connaître la taille maximale atteinte. Parallèlement, la simulation du modèle consiste à comprendre des traces d'exécutions soigneusement sélectionnées en les rejouant. Il existe de nombreux outils

de diagnostic qui facilitent cette démarche, très souvent visuels, comme des diagrammes de séquences ou bien des chronogrammes. Ces deux techniques sont principalement des techniques d'interaction avec les traces.

Quand les traces contiennent des milliers de configurations, les outils de visualisations deviennent inutilisables. De plus, chaque état dans un LTS représente un pas d'une exécution entrelacée de multiples objets concurrents. Le pas est donc très fin et il est très fastidieux de donner un sens à une trace où l'évolution des objets d'intérêt est dissimulée par les changements d'état des autres objets du système. Malgré l'emploi des diverses techniques comme le *slicing* [Wei84], le diagnosticien ne peut traiter une trace trop grande, et donner un sens à l'entrelacement des processus reste encore fastidieux. *L'enjeu porte alors sur la manière de structurer et d'outiller l'analyse de la trace.*

Si les techniques de simplification permettent de faciliter le diagnostic, sans apport d'information de plus haut niveau, elles restent très proches du langage d'implantation du modèle. Rares sont les concepteurs qui choisissent d'utiliser directement le langage formel du model checker (Fiacre ou Promela par exemple), car la plupart utilisent des langages de modélisation plus communs (UML, SysML, Simulink [5]). Un outil de transformation génère le modèle formel correspondant à partir d'un modèle de plus haut niveau. En conséquence, les traces obtenues reflètent le modèle formel du model checker, et sont donc d'un niveau sémantique différent des modèles utilisés pendant la conception. Qui plus est, les ingénieurs préfèrent utiliser des concepts propres à leur domaine, et plus proches de leur logique métier. Il s'agit donc de corréler d'une part les traces et les concepts, et d'autre part les modèles et le domaine. *L'enjeu est de réduire le semantic gap [GV03].*

Quand le diagnosticien est le concepteur du modèle, il ajoute parfois sans le savoir des patrons de solution. Dans la trace obtenue, il sait retrouver le fonctionnement de son patron, mais celui-ci est construit par rapport à sa propre expertise, et n'est pas extrait d'une connaissance commune à un domaine. Si les techniques précédentes permettent à l'ingénieur d'acquérir de la connaissance, elles ne permettent pas de la capitaliser ni de la partager. À cet effet, certains travaux préconisent la définition et l'utilisation du domaine, dont le rôle dans la conception du système est aussi important que les exigences ou le design [Bjø06]. Mais si dans l'approche classique de vérification par model checking le design et les exigences sont réalisés en parallèle [BK08], le domaine n'intervient pas.

Qui plus est, le processus de conception et vérification crée une profusion d'informations hétérogènes (spécifications, modèles structurels ou comportementaux, propriétés, traces etc.), à différents niveaux d'abstractions et de détails, il est par conséquent difficile d'avoir un consensus sur un formalisme de données. Les ingénieurs ont procédé à plusieurs cycles de vérification incluant des informations comme des modèles ou des résultats de propriétés etc... Mais il est impossible de les partager s'il n'y a pas d'accord sur une ma-

5. https ://fr.mathworks.com/products/simulink.html

nière de les capturer. Il est donc difficile de capitaliser l'expérience qui serait souhaitable pour le diagnostic. *L'enjeu est de faciliter le partage de cette connaissance.*

### 1.2.2 Organiser la conception et la vérification

Parfois des connaissances sont partagées, soit au travers d'un patron abstrait pour lequel il n'y a pas de solution toute faite, soit au travers d'un patron possédant une solution complètement réutilisable. La capitalisation de l'expérience peut donc se faire à différents niveaux sur un continuum allant du design (solution) aux spécifications (problème). Si la solution doit être construite à la main, les propriétés implicites liées à cette solution ne sont pas réinjectées, et bien souvent, cela entraîne un diagnostic plus complexe car le problème résultant est mal exprimé. Il faut donc non seulement exprimer la solution choisie, mais aussi l'expression des problèmes que la solution adresse. On s'accorde à dire que la conception est guidée par un ensemble de problèmes mal structurés [SN71] que l'on cherche à résoudre [Vis+03]. L'activité de résolution consiste donc à structurer le problème pour le résoudre. *L'enjeu est d'aider à la résolution, c'est-à-dire à la construction et à la vérification de la solution.*

Le model checking provoque une prolifération de modèles reliés et de sessions de vérifications qui doivent être manipulés avec précaution si l'on veut contrôler le processus de vérification et l'outiller [RB03]. Mais d'une part, les artefacts de vérification sont difficilement gérables compte tenu de la complexité, et donc indisponibles. D'autre part, les activités manipulant ces artefacts sont mal contrôlées, ni enregistrées. Ce problème est peu traité par les travaux de recherche actuels, se concentrant surtout sur les techniques. *L'enjeu est d'organiser les connaissances requises pour le diagnostic et les interactions qu'elles supportent.*

## 1.3 Plan de la thèse

Dans cette thèse nous aborderons ces enjeux avec différents niveaux de détails. Nous proposons pour cela un cadre d'aide au diagnostic, reposant sur une infrastructure organisant l'information suivant plusieurs niveaux de connaissances, et les interactions qu'elles supportent suivant la taxonomie de Bloom. À partir de ces informations nous pouvons définir des outils, tels qu'un outil de simplification de traces. Ces informations sont capitalisées et préparées au partage d'expérience grâce à ce que nous appelons des *problem cases* et des *solution cases*. Nous proposons également de formaliser une démarche de résolution de problèmes s'appuyant sur ce cadre.

La thèse est structurée de la façon suivante. Guidé par différentes illustrations, le chapitre 2 expose les contributions réalisées durant la thèse. Avant de détailler chacune de

ces contributions, le chapitre 3 fait l'état des lieux des activités concourant au diagnostic et des techniques connexes. Ensuite, le chapitre 4 aborde le processus de diagnostic du point de vue des connaissances liées au model checking. Puis, en changeant ce point de vue pour celui des connaissances de domaine, le chapitre 5 présente une méthode de résolution de problème. Des exemples d'application de cette méthode sont présentés dans le chapitre 6. Le chapitre 7 suivant définit un système destiné à organiser les connaissances et les interactions des processus de vérification et de diagnostic. Des exemples d'interactions sont proposés dans le chapitre 8. Enfin, le chapitre 9 conclu sur l'ensemble du travail réalisé, et expose des perspectives possibles pour compléter ce travail.

Chapitre 2

# Illustration du problème et contributions

## 2.1 Approche illustrée

### 2.1.1 Système à l'étude (SAE)

En considérant nos hypothèses de départ, nous évaluons notre approche sur un système à l'étude (SAE) présenté dans la thèse de F. Obeid [OD17]. F.Obeid cherche à sécuriser, conformément à une politique de sécurité donnée, une architecture SCADA en s'appuyant sur la réutilisation de patrons de sécurité. Pour ce faire, il cherche à prouver que les modèles d'architecture SCADA générés dans un langage formel (Fiacre) et dotés de mécanismes basés sur les patrons de sécurité, respectent les exigences décrites dans la politique de sécurité choisie au regard d'attaques. Si les exigences de sécurité ne sont pas respectées par le modèle, alors des traces sont produites, indiquant soit que l'application du patron est incorrecte, soit que les choix des patrons ou leurs combinaisons sont incorrects vis-à-vis de l'architecture SCADA et de la politique de sécurité attendue.

Les systèmes d'acquisition et de contrôle de données (SCADA) nous intéressent particulièrement, car ils sont fortement impactés par les enjeux de complexité et de sécurité. Ces systèmes sont mis en œuvre dans les installations industrielles modernes (centrales électriques, gestions de l'eau, raffineries de pétrole, installations nucléaires, etc), et sont composés de moyens informatiques, hautement concurrents, qui contrôlent et pilotent les procédés industriels.

Un SCADA regroupe un ensemble de composants, des capteurs ou actionneurs, des contrôleurs logiques programmables (PLC) ou des machines de contrôle et coordination (GC), connectés ensemble par différents réseaux (Network). Comme le montre la figure 2.1, notre SAE est un système SCADA composé de 4 entités, deux contrôleurs locaux ($Plc_1$ et $Plc_2$), un contrôleur global ($Gc$) connectés par un réseau ($Network$). Ce système est intégré dans un environnement ($Env$), via une interface au niveau du contrôleur global. Les liens entre les entités sont des canaux (channels) du type FIFO unidirectionnels permettant l'échange de messages de manière asynchrone.

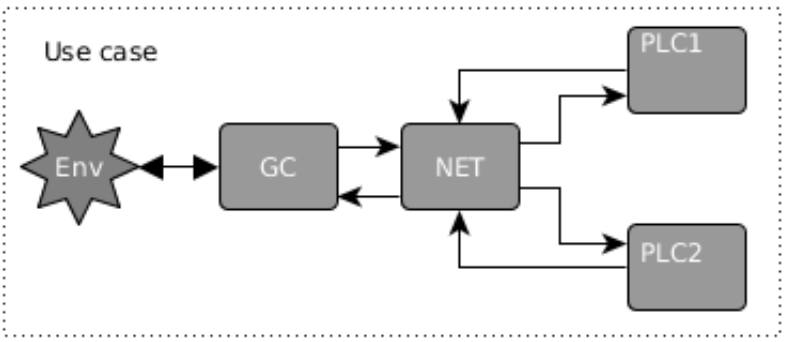


FIGURE 2.1 – Architecture SCADA abstraite

### 2.1.2 Cas d'utilisations

Considérons les trois exemples de diagnostic sur deux architectures différentes, illustrés sur la figure 2.2.

Dans le premier cas (1), un ingénieur vérifie le comportement d'un système conçu par un tiers, illustré sur la figure 2.3. L'ingénieur a une pratique de la vérification par model checking, mais peu de connaissances quant au domaine du système. Dans le second cas, un concepteur reprend ce même système, et y ajoute un mécanisme d'exclusion mutuelle, comme illustré sur la figure 2.4, puis le vérifie. Il bénéficie d'une expérience dans le domaine de l'application qu'il conçoit. Dans le dernier cas, un architecte sécurise le système, comme illustré sur la figure 2.5, puis le vérifie. Il bénéficie d'une base de connaissances externe. Rappelons que la question générale qui nous concerne est de faciliter la phase de diagnostic qui survient lorsqu'une vérification a échouée.

#### Premier cas : Vérification du modèle construit par un tiers

Dans le premier cas, un ingénieur que nous appellerons Luka, doit vérifier un modèle conçu par un tiers. Luka n'est pas expert dans le domaine des SCADA, mais il a une bonne connaissance en model checking. Il est familier avec les processus communicants décrits dans des langages formels, familier avec les propriétés formelles décrites en logique temporelle, et familier avec la technique d'exploration des LTS. Le modèle qu'il doit vérifier est un modèle formel exprimé en langage Fiacre généré à partir d'un modèle UML (figure 2.3). En parallèle, il lui a été fourni un ensemble de propriétés à vérifier, décrites en LTL. Il dispose également d'un ensemble de scenarii d'utilisations du modèle formalisé sous la forme de contextes CDL. Luka suit l'approche décrite par C.Baier [BK08]. À l'aide d'un model checker (OBP), il génère le LTS correspondant au modèle, et l'explore.

Dans un des scenarii, le client envoie plusieurs messages au $PLC_1$ et attend un message

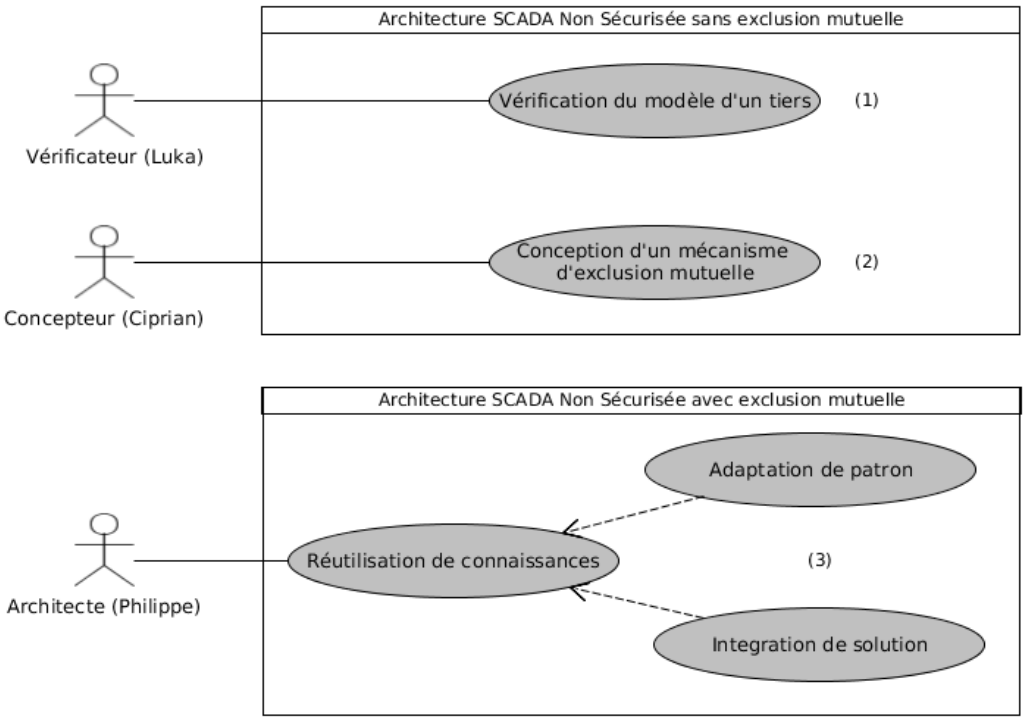


FIGURE 2.2 – Cas d'utilisations

d'acquittement (*ACK*) de la part du système. L'exploration montre que ce scénario ne fonctionne pas car une propriété vérifiant ce comportement est violée (*CLT1SendAck*), déclenchant ainsi une phase de diagnostic. Cette propriété énonce que *si un Client envoie une donnée au* $PLC_1$*, alors celui-ci recevra fatalement un ACK venant de* $PLC_1$. OBP génère un contre-exemple exposant à Luka les configurations amenant à cette conclusion. Grâce à sa connaissance en model checking, Luka peut rejouer le contre-exemple pour en comprendre la cause.

Mais la trace produite par le model checker est trop détaillée, constituant un frein à l'analyse manuelle. Il existe un ensemble de techniques d'aide au diagnostic, comme comparer des traces positives et négatives afin d'isoler des parties suspectes de la trace, ou bien réduire la trace par slicing. Luka opte pour l'utilisation d'un outil de visualisation de trace sous forme d'un diagramme de séquence. Le diagramme de séquence permet d'afficher uniquement les interactions entre les différents processus. Cet outil aide Luka à identifier qu'un message n'est pas transmis entre deux processus, $PLC_1$ et *Network*. Le message ne parvient pas au *Network* car celui-ci n'est pas transmis par le bon canal. Luka propose un diagnostic, le processus $PLC_1$ est mal connecté.

Une fois la correction apportée au modèle, Luka s'assure que celui-ci est correct en effectuant une autre vérification. Mais la propriété est une nouvelle fois violée. À travers

le diagramme de séquence, Luka s'aperçoit pourtant que sa correction a fonctionné, le message est correctement transmis. Ici, le diagramme de séquence ne permet pas d'aller plus loin dans l'analyse, il faut changer de stratégie. Luka énonce de nouvelles propriétés lui permettant d'observer des comportements particuliers du modèle, en y ajoutant au besoin quelques annotations. Par exemple, une propriété énonce que *quand un channel est lu il doit être dépilé*. Ces propriétés de contrôle sont inhérentes aux processus communicants. Le model checker indique maintenant qu'une propriété est violée. Un channel est lu sans être dépilé. Conclusion, le channel doit être dépilé après avoir été lu.

Le modèle est une nouvelle fois corrigé, mais la propriété est toujours violée. Bien que Luka ait connaissance des spécifications du problème et de connaissances toujours applicables en matière de model checking, celles-ci sont parfois insuffisantes pour effectuer le diagnostic, le rendant long et fastidieux. De nouvelles connaissances sont alors nécessaires pour comprendre la solution.

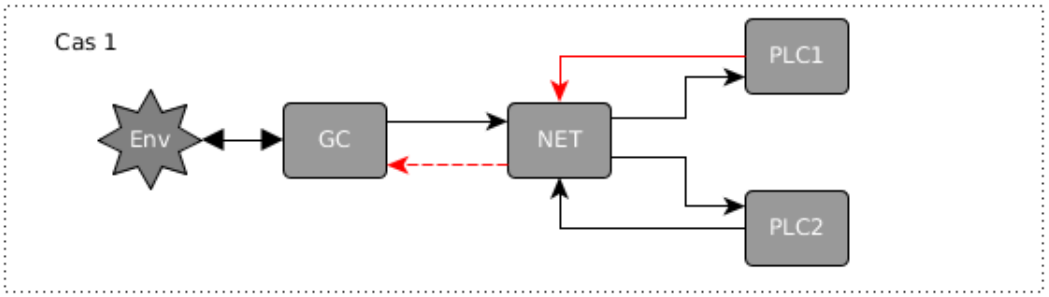


FIGURE 2.3 – Cas d'utilisation 1, en rouge les erreurs identifiées par Luka

### Second cas : Conception d'un mécanisme d'exclusion mutuelle

Ciprian est le concepteur du modèle, et il le maîtrise. Il souhaite ajouter une nouvelle fonctionnalité. $PLC_1$ et $PLC_2$ partagent tous deux une ressource, lue ou écrite lorsque $PLC_1$ ou $PLC_2$ sont dans leur état *Access*, soit *PLC1@Access ou PLC2@Access*[1]. Le modèle doit vérifier l'exigence suivante : *il n'est pas possible que* $PLC_1$ *et* $PLC_2$ *soient en même temps dans leur état* Access, exigence traduite sous la forme d'une propriété *PSingleAccess*. Pour répondre à ce nouveau problème il opte pour un mécanisme de drapeaux. Le principe est le suivant, $PLC_1$ et $PLC_2$ possèdent tous deux un drapeau. $PLC_1$ ou $PLC_2$ lève son drapeau, pour informer qu'il accède à la ressource. Avant d'accéder à cette ressource, un processus doit vérifier que le drapeau de l'autre processus n'est pas levé, sinon, il doit attendre qu'il soit baissé. Lorsqu'un processus a fini d'utiliser la ressource, il baisse son drapeau et libère la ressource.

1. Nous exprimons qu'un processus *Processus* est dans son état *Etat* par la notation *Processus@Etat*

Après vérification, il s'avère que la propriété *PSingleAccess* est violée, signalant l'existence d'un contre exemple où $PLC_1$ et $PLC_2$ utilisent en même temps la ressource. Comme précédemment, Ciprian commence par vérifier des propriétés liées à la modélisation des processus communicants (comme dans le cas précédent). Au bout d'un certain temps, sans succès, il en conclut que le problème vient de la manière dont il a conçu le mécanisme. Pour identifier parmi les milliers de configurations de la trace le comportement du mécanisme implémenté, il doit exprimer des propriétés liées à ce mécanisme. Ciprian s'aperçoit alors que le drapeau est levé après le test d'accès à la ressource. Il doit donc corriger le modèle et repartir sur un cycle de vérification.

Si Ciprian connaît la spécification du problème et est capable de le résoudre, il n'a pas de connaissances de solutions existantes. Il va devoir concevoir lui-même un mécanisme d'exclusion mutuelle et suivra un cycle itératif de conception-vérification, pouvant être bloqué s'il ne sait pas le résoudre entièrement. Celui-ci pourrait être évité par la réutilisation de solutions du domaine.

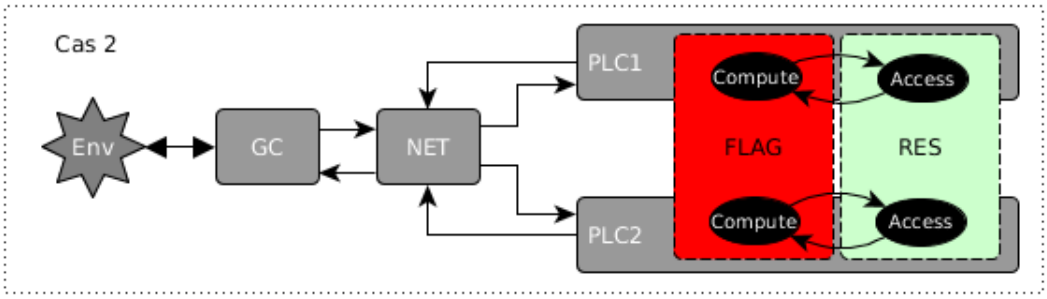


FIGURE 2.4 – Cas d'utilisation 2, Ciprian conçoit une exclusion mutuelle

### Troisième cas : Réutilisation de connaissances

Après avoir assuré un mécanisme d'exclusion mutuelle pour les deux *Plc* envers la ressource, un autre ingénieur, Philippe, doit sécuriser l'architecture face à des cyberattaques. Pour ce faire, il prévoit d'implémenter un mécanisme de Single Access Point (*SAP*), réduisant l'accès à la ressource en un point unique. Philippe choisit de réutiliser une solution existante, mettant en œuvre le *SAP*, qui est accompagnée de propriétés et d'une architecture abstraite. Pour l'utiliser, il doit réaliser un ensemble de modifications dans son modèle, et pour le vérifier, il doit faire correspondre les propriétés exprimées par le patron avec les éléments de son modèle. Après vérification, il s'avère qu'une des propriétés est violée. Grâce au *SAP*, Philippe bénéficie d'un ensemble de propriétés qui peuvent l'aider à localiser le problème. La connaissance du domaine, c'est-à-dire la connaissance de solutions, permet alors de le guider dans le diagnostic. Finalement, il s'avère que la

solution construite n'est pas conforme. Bien que des propriétés aient pu aider au diagnostic, la solution conçue de manière ad hoc, reste difficile à diagnostiquer. Philippe doit donc comprendre comment il a mis en œuvre la solution pour comprendre l'origine du problème.

Une nouvelle exigence, traduite sous la forme d'une propriété formelle, exige que si le client a les droits suffisants, il doit être autorisé à accéder à la ressource. Le mécanisme de *SAP* n'est plus suffisant pour protéger l'architecture *SCADA*. Un mécanisme d'autorisation (*AUTH*) doit être ajouté. Ainsi, seuls les clients autorisés pourront accéder à la ressource. Cette fois-ci, il existe un patron d'autorisation incluant une solution adaptable. Philippe applique ce patron à son modèle. La propriété est tout de même violée, préfigurant deux problèmes. Soit l'adaptation du patron a été mal réalisée, c'est-à-dire par exemple que le patron n'est pas adapté au modèle de la bonne manière, soit c'est un mauvais choix de patron. Une fois corrigée, la solution est capturée sous la forme d'un composant réutilisable. Ici c'est la méthode employée qui est au cœur du problème. La solution choisie amène avec elle un lot de propriétés liées aux composants intégrés. Le modèle est ainsi constitué d'un ensemble de composants, chacun associé à un ensemble de propriétés axiomatiques dont certaines permettent de contrôler l'intégration. Cette couverture permet d'isoler plus précisément le composant qui porte le problème, réduisant ainsi l'effort de diagnostic.

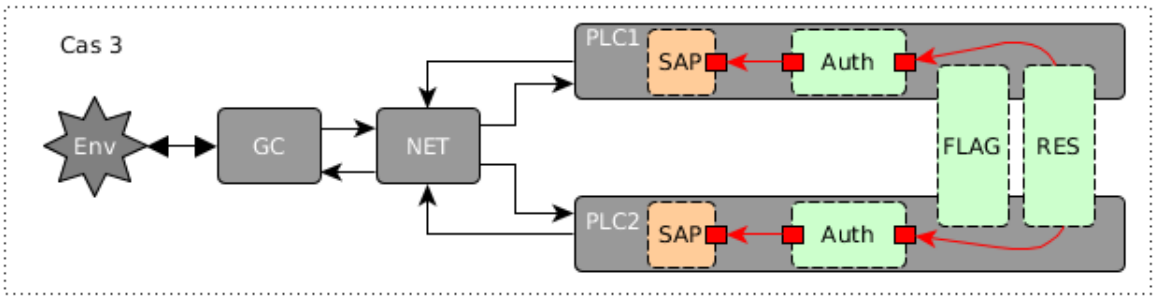


FIGURE 2.5 – Cas d'utilisation 3, Philippe sécurise l'architecture

## 2.2 Contributions

### 2.2.1 Cadre

Comme nous cherchons à fournir des aides au diagnostic, nous devons définir ce qu'est un diagnostic. Il est défini par [Mer04] comme une *investigation ou une analyse des causes ou natures d'une condition, situation ou problème.* Dans le cas du model checking, il est

déclenché par la découverte d'un contre-exemple. Il est traditionnellement facilité par différentes techniques, comme par exemple :

— *Simuler* en rejouant la trace, *stimuler* par l'altération légère du modèle ou une modification de son contexte d'exécution, ou *générer* de nouvelles propriétés afin d'acquérir de nouvelles connaissances.

— *Séparer*, *isoler* ou *réduire* la trace, afin d'éliminer ou simplifier les connaissances.

— *Comparer* des traces valides ou violées, ou *corréler* les traces et modèles avec de nouvelles informations (pretty-printing, annotations, propriétés) pour changer de point de vue ou permettre de nouvelles analyses.

Dans la définition de [Mer04], le diagnostic représente un processus cognitif (investigation ou analyse). De même, *simuler*, *stimuler*, *générer*, *isoler*, *réduire*, *séparer*, *corréler* ou *comparer* sont des opérations cognitives que nous utilisons dans le contexte de la vérification par model checking.

### Processus cognitifs et diagnostic

La taxonomie de Bloom [Blo+64] est une classification éprouvée et mature qui permet de classer les activités cognitives. Destinée à l'évaluation des connaissances des étudiants, elle est suffisamment générique pour être extrapolée à la connaissance sur tout type de problème, dont celui de la conception de systèmes logiciels [SR04][BE03a]. Dans [XR04] les auteurs associent les activités cognitives requises lors du debug un programme logiciel avec les activités cognitives de Bloom, montrant qu'il était possible d'exprimer une partie de ce processus à l'aide d'activités de la taxonomie.

La taxonomie regroupe des comportements intellectuels, initialement les résultats attendus par le processus éducatif, en six catégories. *Se souvenir* représente le fait de mémoriser des informations, et de pouvoir se les rappeler, sans pour autant les comprendre. Les opérations cognitives associées sont entre autres *rappeler* ou *reconnaître*. *Comprendre* est la capacité à expliquer le sens de l'information, c'est-à-dire être capable de traduire un contenu dans un autre langage, ou bien l'extrapoler pour en faire des prédictions. Les opérations cognitives associées sont par exemple *interpréter*, *exemplifier*, *classifier*, *résumer*, *inférer*, *comparer* ou *expliquer*. *Appliquer* est l'usage d'abstractions dans des situations concrètes, autrement dit, le fait de sélectionner des données pour répondre à un problème nouveau. Parmi les opérations cognitives associées on retrouve *exécuter* ou *développer*. *Analyser* est la décomposition d'un tout en composants, autrement dit, rendre explicite des liens entre ces éléments pour déterminer des interactions, ou bien reconnaître des arrangements ou des structures qui maintiennent ces éléments entre eux. Cette catégorie inclut les opérations cognitives *différencier*, *organiser* ou *attribuer*. *Créer* est la mise ensemble de parties afin de réaliser un nouveau tout, les opérations cognitives sont *générer*,

***1.0 Remember*** – Retrieving relevant knowledge from long-term memory.
- ***1.1 Recognizing***
- ***1.2 Recalling***

***2.0 Understand*** – Determining the meaning of instructional messages, including oral, written, and graphic communication.
- ***2.1 Interpreting***
- ***2.2 Exemplifying***
- ***2.3 Classifying***
- ***2.4 Summarizing***
- ***2.5 Inferring***
- ***2.6 Comparing***
- ***2.7 Explaining***

***3.0 Apply*** – Carrying out or using a procedure in a given situation.
- ***3.1 Executing***
- ***3.2 Implementing***

***4.0 Analyze*** – Breaking material into its constituent parts and detecting how the parts relate to one another and to an overall structure or purpose.
- ***4.1 Differentiating***
- ***4.2 Organizing***
- ***4.3 Attributing***

***5.0 Evaluate*** – Making judgments based on criteria and standards.
- ***5.1 Checking***
- ***5.2 Critiquing***

***6.0 Create*** – Putting elements together to form a novel, coherent whole or make an original product.
- ***6.1 Generating***
- ***6.2 Planning***
- ***6.3 Producing***

FIGURE 2.6 – Taxonomie de Bloom, processus cognitifs d'après D.Krathwohl [Kra02]

***A. Factual Knowledge*** – The basic elements that students must know to be acquainted with a discipline or solve problems in it.
- **Aa. Knowledge of terminology**
- **Ab. Knowledge of specific details and elements**

***B. Conceptual Knowledge*** – The interrelationships among the basic elements within a larger structure that enable them to function together.
- **Ba. Knowledge of classifications and categories**
- **Bb. Knowledge of principles and generalizations**
- **Bc. Knowledge of theories, models, and structures**

***C. Procedural Knowledge*** – How to do something; methods of inquiry, and criteria for using skills, algorithms, techniques, and methods.
- **Ca. Knowledge of subject-specific skills and algorithms**
- **Cb. Knowledge of subject-specific techniques and methods**
- **Cc. Knowledge of criteria for determining when to use appropriate procedures**

***D. Metacognitive Knowledge*** – Knowledge of cognition in general as well as awareness and knowledge of one's own cognition.
- **Da. Strategic knowledge**
- **Db. Knowledge about cognitive tasks, including appropriate contextual and conditional knowledge**
- **Dc. Self-knowledge**

FIGURE 2.7 – Taxonomie de Bloom révisée, connaissances d'après D.Krathwohl [Kra02]

*produire* ou *planifier*. Enfin *évaluer* est le fait de produire des jugements à propos des idées ou des phénomènes. On peut considérer que le diagnostic est une opération cognitive d'évaluation, pour autant celle-ci a recours à l'ensemble des opérations cognitives de la taxonomie, par exemple le diagnostic *le processus GC ne transmet pas le message ACK* inclut d'*isoler* les communications entre les processus.

La figure 2.6 présente la taxonomie de Bloom du point de vue de la dimension des *processus cognitifs*.

### Connaissances et diagnostic

On décrit généralement les objectifs d'apprentissage en termes de contenu (exprimé sous la forme d'un nom) et de descriptions de ce qui doit être fait avec ce contenu (exprimé

sous la forme d'un verbe). Cette distinction s'applique aussi au diagnostic car d'après la définition, le processus de diagnostic (verbe, ex : isoler) s'établit à partir de conditions, situations ou problèmes (nom, ex :Plc1). D.Krathwohl [Kra02] a proposé de faire apparaître dans la taxonomie de Bloom une nouvelle dimension, celle des connaissances. La figure 2.7 présente cette nouvelle dimension. En reprenant cette idée d'une dimension de connaissance indépendante des processus cognitifs, nous structurons les connaissances en trois catégories, les connaissances liées au model checking, les connaissances du domaine et les connaissances de gestion.

Les connaissances liées au model checking regroupent un ensemble d'informations qui sont inhérentes au contexte de vérification et de diagnostic. De manière non exhaustive, il s'agit des connaissances relatives aux processus concurrents (processus, canaux...), aux LTS (états, transitions, configurations, variables...), aux propriétés formelles (LTL, vivacité, sûreté ...) ainsi que des connaissances générales en matière de vérification par model checking (scénarios, model checkers, exploration exhaustive, BFS, DFS, cycles, deadlocks, contre-exemples...).

Les connaissances du domaine ne sont pas toujours des connaissances applicables dans la vérification par model checking, elles sont liées au domaine de l'application. Un domaine d'application peut être décrit par tous les phénomènes observables du domaine (entités, fonctions, événements, comportements...) et les relations qui les unissent. Il exprime le périmètre de l'application, sans aucune référence à ses exigences, ni à ses implémentations [Bjø06]. Une représentation d'un domaine n'est pas simplement de la connaissance contenue dans le cerveau d'un expert du domaine ; c'est une abstraction rigoureusement organisée et sélective de cette connaissance. Les connaissances du domaine sont par exemple une architecture SCADA (GC, PLC, Network), un mécanisme d'exclusion mutuelle (façonné en trois états : Entrée, Section Critique, Sortie) un mécanisme d'autorisation, ou encore le mécanisme de drapeau répondant à l'exclusion mutuelle. Ces différentes connaissances sont exprimables dans un patron, c'est-à-dire une solution ou un ensemble de bonnes pratiques éprouvées pour une classe de problèmes récurrents. Il existe différentes formes de patrons, de conception, d'architecture, ou bien encore donc, de sécurité. Ainsi, dans notre troisième cas d'utilisation, chaque patron de sécurité est décrit par une description formelle de sa structure et de son comportement, ainsi qu'une description formelle des propriétés de sécurité associées à ce patron.

Les connaissances de gestion, par exemple celles relatives à la résolution de problème ou au résultat de diagnostic, sont transversales aux autres connaissances. Tout d'abord, nous avons vu dans la définition du diagnostic que les processus cognitifs manipulaient des connaissances afin d'atteindre un objectif, le diagnostic. Ce diagnostic ici, est le résultat du processus de diagnostic, et représente un type de connaissance que l'on peut vouloir conserver. Il existe aussi des connaissances transversales aux domaines liées à la méthode

de résolution, souvent implicites et qui sont généralement assimilées à de l'expérience. Un exemple est la méthode d'application d'un patron. D'un côté, intégrer un patron sous la forme d'un nouveau processus dans son design est un gain du point de vue de la lisibilité de design, mais peut être un frein du point de vue de d'explosion combinatoire. D'un autre côté, mélanger le comportement du patron dans le modèle, par exemple en ajoutant des nouveaux états dans un processus, peut-être un choix judicieux pour diminuer l'impact de l'explosion combinatoire, mais rendra aussi le diagnostic plus complexe.

La décomposition de la dimension processus de Bloom est utilisée dans cette thèse. Sans avoir repris la décomposition de la dimension connaissance de Bloom, nous utilisons son pouvoir structurant pour organiser la thèse grâce aux trois catégories de connaissances définies ci-dessus.

### 2.2.2 Comprendre et analyser le Système A l'Étude

Parmi les enjeux identifiés au premier chapitre, les deux premiers sont inhérents aux connaissances de model checking.

*Comment structurer et outiller l'analyse de la trace ?* Pour analyser le contre-exemple, le diagnosticien doit réaliser un ensemble d'activités cognitives qu'il est difficile d'appréhender. En plaçant ces activités dans la taxonomie de Bloom, nous permettons de comprendre ces activités cognitives menées lors d'un diagnostic, chacune pouvant être facilitée par un outil ou une technique.

Si l'on porte l'effort sur les traces, l'ingénieur a à sa disposition au moins deux approches, soit simplifier (réduire ou isoler les parties suspectes amenant à la trace, séparer pour appréhender le système par parties), soit offrir un nouveau point de vue sur la trace à travers un outil de visualisation par exemple.

*Comment réduire le fossé sémantique ?* Lors du diagnostic, les connaissances liées au model checking ne permettant pas toujours de résoudre le problème, il faut souvent raisonner sur une représentation abstraite tel qu'un algorithme. Ce changement de point de vue contraint le diagnosticien à résoudre un fossé sémantique entre les spécifications abstraites et le contre-exemple exprimé par des connaissances liées au model checking. Il faut pour cela définir des corrélations entre les éléments de trace qui doivent être annotés, et le problème formulé. Nous proposons un cadre pour corréler les éléments entre les différents niveaux de connaissances, et ce à travers un mécanisme de fédération de modèles.

### 2.2.3 Formaliser, partager et réutiliser la connaissance

*Comment peut-on faciliter le partage et la réutilisation de la connaissance ?* Pour partager des connaissances, il faut qu'il y ait un consensus ontologique et une structure.

Nous proposons une formalisation des différentes connaissances, de la vérification par model checking, à la méthode et au diagnostic en passant par le domaine. Cette formalisation permet de capitaliser la connaissance et de la préparer au partage d'expérience grâce à des *problem cases*.

*Comment aider à la résolution, c'est-à-dire à la construction et à la vérification de la solution ?*. Dans une approche par réutilisation, comme dans le cas 3, le diagnostic est grandement simplifié. Premièrement, les solutions sont déjà éprouvées, par conséquent les erreurs sont liées aux choix de ces patrons, ou à leur composition. Deuxièmement, ces patrons ramènent un ensemble de propriétés qui, une fois assemblées, vont pouvoir être exploitées dans le modèle à vérifier. Troisièmement, les patrons constituent une documentation quant au fonctionnement du système, aidant à la compréhension. Nous proposons de formaliser cette démarche sous la forme d'une méthode de résolution de problèmes.

### 2.2.4 Organiser la méthode, le domaine

*Comment organiser toutes ces connaissances et les interactions qu'elles supportent ? Quelle infrastructure peut supporter la méthode ?* C.Baier [BK08] souligne que, de manière transversale aux autres phases, le processus de vérification doit être planifié, administré et organisé, à travers une activité appelée *l'organisation de la vérification*. Une plateforme logicielle supportant l'organisation de la vérification serait aussi en mesure de capitaliser les expériences passées, et d'en permettre la réutilisation. Tout comme le pense [Bou09], la mise en œuvre d'un support ou d'une méthodologie outillée est primordiale pour faciliter à la fois la détection de problèmes, la localisation des fautes, et l'analyse des causes de ces dysfonctionnements. Ainsi nous proposons d'outiller l'ensemble des points précédents en s'appuyant sur une infrastructure organisant leurs artefacts et les interactions qu'ils supportent.

Chapitre 3

# Etat de l'Art

## 3.1 Contexte

### 3.1.1 Généralités

#### Sûreté de fonctionnement

Pour certains systèmes, les défaillances peuvent engendrer de sévères conséquences, matérielles, financières ou humaines. La sûreté de fonctionnement est la première exigence en matière de qualité et est définie comme l'aptitude d'une entité à fonctionner quand et tel que requis [Avi+04]. Le comité technique IEC TC 56 développe et maintient une série de standards internationaux autour de la sûreté de fonctionnement (NF EN 60300) encouragée par l'utilisation d'une terminologie commune (IEC 60050-192) aux domaines concernés par la sûreté de fonctionnement. La sûreté de fonctionnement logiciel fait l'objet d'une norme complémentaire à la norme IEC 60300 (NF EN 62628) qui offre des lignes directrices concernant la sûreté de fonctionnement du logiciel. Elle redéfinit la sûreté de fonctionnement d'un logiciel comme étant la capacité de l'élément logiciel à fonctionner au moment voulu et tel que requis lorsqu'il est intégré dans l'exploitation du système.

#### Faute, erreur et défaillance

On distingue trois types de menaces pouvant réduire la sûreté de fonctionnement d'un système, les défauts, les erreurs et les défaillances, défini pour le logiciel par la norme NF EN 62628.

**Faute**. Action, volontaire ou non, pouvant générer un défaut.

**Défaut ou Bug**. Écart entre une caractéristique du logiciel et la caractéristique voulue, aussi appelé bug. C'est un état d'un élément logiciel qui peut empêcher sa bonne exécution.

**Erreur**. Une erreur (appelée infection dans [Zel06]), est un état d'un modèle qui diffère de l'intention du concepteur. Elle est latente tant que la partie erronée du système n'est pas sollicitée. Elle devient effective au moment de la sollicitation de la partie erronée.

**Défaillance** Une défaillance est une cessation de l'aptitude d'une entité à accomplir une fonction requise. Elle survient lorsque le service délivré dévie du service spécifié, la

spécification étant une description agréée du service attendu.

Pour améliorer la qualité d'un logiciel l'objectif est de réduire les défaillances, et pour ça, le diagnostic cherche à en identifier les causes. Une cause peut être définie comme la raison d'une action ou d'une situation [Mer04], elle est la prémisse à une conséquence (ou effet), qui elle est l'effet résultat d'une cause. On peut croire que la cause d'une défaillance vient systématiquement de la conception du logiciel, mais ce n'est pas toujours le cas. Une exigence originelle peut ne pas prendre en considération les changements futurs, ou bien le modèle peut ne pas posséder d'interfaces compatibles, ou encore il se peut que le modèle interagisse de manière imprédictible car distribué [Zel06]. Parfois les causes sont bien plus profondes, et la causalité est complexe. J.L Mackie [Mac90] définit la causalité comme étant une séquence d'événements liés. Par exemple, les ingénieurs qui développaient le F-16 avaient réutilisé un logiciel initialement conçu pour des fusées, une faute car le logiciel est inadapté aux F-16 (les fusées n'ont pas d'orientation). Cette faute a produit des bugs latents dans le logiciel implémenté, qui furent activés lorsque l'avion atteignit les conditions nécessaires (i.e traverser l'équateur) causant une erreur dans le système. Cette réaction en chaîne produisit une défaillance de l'avion qui se retourna.

En model checking les causes sont variées [MR11], [Chr+16]. Elles peuvent provenir d'erreurs dans la spécification formelle du comportement, ce qui arrive quand le modèle formel est incorrect et ne satisfait pas les propriétés, alors que les propriétés expriment bien le souhait du concepteur. Elles peuvent provenir d'erreurs dans la spécification formelle des exigences (propriétés), qui peuvent être incorrectes ou incomplètes, c'est-à-dire que la spécification formelle du comportement satisfait les propriétés, mais les propriétés n'expriment pas ce que le concepteur veut. Elles peuvent provenir de faux positifs, par exemple des manquements de l'outil de vérification (incomplet pour les conditions de validation), d'un mauvais paramétrage de l'outil de vérification, des spécifications incomplètes, ou d'une mauvaise interprétation des observations ou des métriques obtenus.

Dans cette thèse nous prendrons l'hypothèse que les fautes proviennent principalement de la spécification formelle du comportement.

### Techniques

Retracer la chaîne d'infection à partir des traces demande à la fois une recherche dans l'espace (trouver la ou les variables infectées parmi des milliers) et dans le temps (parmi des millions de configurations, trouver le moment où l'infection a commencé) [CZ05]. Cette activité doit donc être au maximum automatisée, mais par quelle technique ? A notre connaissance il n'existe pas de consensus sur une classification de techniques de diagnostic pour le model checking. Dans la tentative de classification de V. Ribaud et al. [Rib+14], trois grandes catégories de techniques sont proposées. Celles permettant de déterminer les causes d'erreurs, celles permettant de mieux comprendre le système,

et celles relatives à la simulation, l'observation et la manipulation du modèle. Pour ces auteurs, aujourd'hui la plupart des techniques étudiées se focalisent sur l'analyse de traces, mais comme le montre leur étude, il existe de nombreuses autres techniques qui pourraient être appliquées.

En effet, depuis les années 70 et dans le domaine du diagnostic, deux communautés scientifiques convergent l'une vers l'autre, la communauté de l'automatique et la communauté de intelligence artificielle, couvrant divers domaines comme la surveillance des procédés industriels, la médecine ou le logiciel, et produisant techniques et outils de diagnostic. Dans le domaine du logiciel, les approches de localisation des bugs sont classées par W. E. Wong et al. [Won+16] en deux catégories : les techniques classiques laissant un effort humain important (logs, assertions, points d'arrêt, profilage) et les techniques avancées laissant un effort humain moindre (slicing, analyse spectrale, statistiques, apprentissage, data-mining). Dans le domaine des procédés industriels, les approches de diagnostic sont classées aussi en deux catégories (figure 3.1) par V. Venkatasubramanian et al. [VRK03] : les approches basées sur des modèles qui s'appuient sur un modèle décrivant le fonctionnement du système (l'élaboration des modèles y est central) et les approches basées sur un historique de données reposant sur des connaissances produites lors d'expériences passées (l'acquisition des connaissances y est central). Il y a donc effectivement un large spectre de techniques mêlant informatique et mathématiques, de l'apprentissage automatique, aux statistiques, en passant par les probabilités, ou la théorie de l'information. La richesse et l'hétérogénéité de ces techniques rend difficile l'organisation du diagnostic par ce biais.

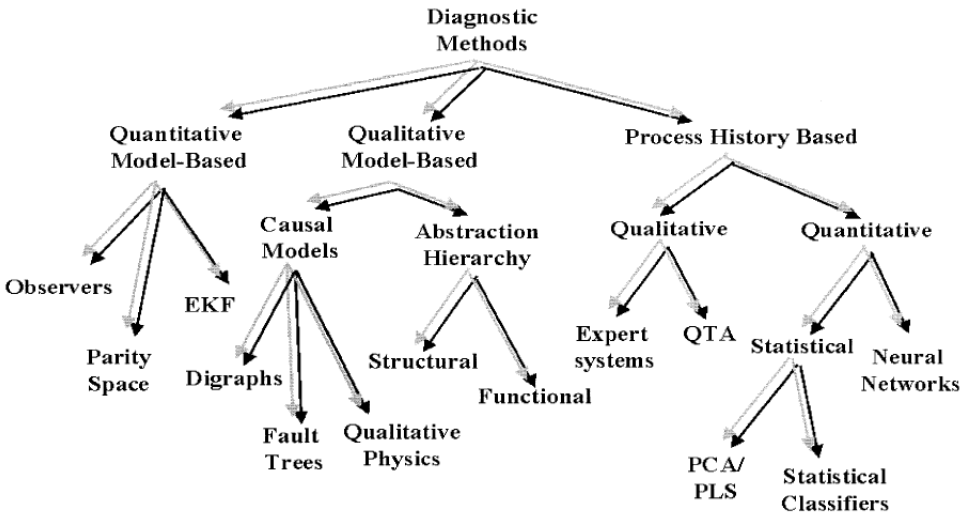


FIGURE 3.1 – Classification

### 3.1.2 Le choix d'un cadre

#### Définition du diagnostic

Reprenons la définition du diagnostic comme une investigation ou une analyse des causes ou natures d'une condition, situation ou problème [Mer04]. Cela le désigne comme un processus dont le produit est un diagnostic. Dans le domaine médical, le diagnostic infirmier est le processus qui permet d'énoncer un jugement clinique sur les réactions aux problèmes de santé d'une personne, tandis que le diagnostic médical est le processus d'identification de la maladie expliquant les signes ou symptômes présentés chez le patient. Dans le domaine des procédés industriels, un diagnostic est défini par J.C. Laprie comme un processus d'identification du défaut ayant conduit à une défaillance observée du système ou de ses composants [Lap95]. Le diagnostic est donc un processus cognitif (investigation, analyse), qui agit sur des connaissances (condition, situation ou problème) pour proposer le diagnostic (causes), déclenché par la présence de symptômes (observations sur le système).

Dans notre propre définition, le diagnostic est le processus cognitif qui, à partir de contre-exemples (symptômes) produits par un model checker explorant un modèle de comportement formel sur la base de propriétés formelles (condition, situation, problème), cherche à identifier les erreurs dans le modèle de comportement formel (causes, problèmes).

#### Processus de diagnostic et activités cognitives

Un processus de diagnostic est une succession d'activités cognitives plus ou moins complexes.

Nous admettons trois manières de penser, l'abduction, l'induction et la déduction. Considérons trois éléments, les *faits*, les *causes* de l'apparition de ces *faits*, et les *règles* qui associent les *faits* aux *causes*. Le raisonnement déductif consiste à déduire de la connaissance à partir de connaissances préexistantes. Étant donné une *cause* (ex : le programme a une erreur de segmentation) et une *règle* (ex : l'erreur de segmentation cause l'arrêt du programme) alors on déduit le *fait* (ex : le programme s'arrête). Le raisonnement inductif consiste à partir d'observations pour aller vers une loi générale. Étant donné un *fait* (ex : le programme s'arrête) associé à une *cause* (ex : le programme a une erreur de segmentation), on infère la *règle* (ex : l'erreur de segmentation cause l'arrêt du programme). Le raisonnement abductif cherche les *causes* à partir de *faits* et de *règles*. Étant donné un *fait* (ex : le programme s'arrête) et une *règle* (ex : la segmentation fault cause l'arrêt du programme) alors on émet une *hypothèse* sur la *cause* (ex : le programme a (peut être) une segmentation fault). Ainsi, l'abduction produit des idées et des concepts à expliquer, puis l'induction participe à la construction de l'hypothèse abductive en lui donnant de la consistance, enfin la déduction formule une explication prédictive à partir de cette

construction [Buc+99].

Dans un contexte de systèmes de contrôle modernes, J. Chen et R. J. Patton [CP99] définissent un processus de diagnostic large fait de trois activités : - l'activité de détection de défaut, qui décide si quelque chose va ou ne va pas ; l'isolation du défaut, qui détermine l'emplacement du défaut ; et l'identification du défaut, qui estime la taille, le type ou la nature du défaut.

Le debugging est le processus de diagnostic dont l'objectif est de retirer les bugs dans un programme qui cause une défaillance lors des exécutions, il s'agit d'une situation applicable dans notre contexte. Selon A. Zeller [Zel06] il est composé d'une succession d'activités cognitives : - récupérer le problème à partir d'une base ; - reproduire le problème dans son environnement ; - simplifier le cas de test ;- émettre des hypothèses sur la cause du problème ;- se focaliser sur la cause la plus probable ; - isoler le problème et corriger le défaut. Dans ce processus, on distingue certaines activités présentes dans [CP99] comme par exemple *isoler*, et des façons de penser comme l'abduction (émettre une hypothèse). Mais il existe d'autres activités comme *simplifier*, ou *corriger*. Nous devons donc trouver un cadre plus général et mature pour classer les activités cognitives.

### Taxonomie de Bloom

La taxonomie de Bloom est un cadre qui permet de quantifier différents niveaux de connaissances dans un contexte éducatif. Pour ce faire, elle regroupe des comportements intellectuels, résultats attendus par le processus éducatif, en catégories plus ou moins complexes. Elle compte six niveaux de comportements intellectuels, *se souvenir*, *comprendre*, *appliquer*, *analyser*, *créer* et *évaluer*. Chaque catégorie contient des activités cognitives.

Cette taxonomie bénéficie d'une grande maturité et généricité. Elle peut être appliquée au processus de diagnostic logiciel, comme le montrent X. Shaochun et V. Rajlich [SR04], qui l'appliquent pour le processus de debug, ou encore J. Buckley et C. Exton [BE03b] qui l'extrapolent pour la compréhension de programmes logiciels. Dans cette dernière étude, les auteurs questionnent la compréhension des apprenants face à une IHM java d'une calculatrice. Chaque catégorie de Bloom est évaluée par des questions. Par exemple *se souvenir* est évaluée par la question "quel est le style de convention utilisée dans le programme ?", une réponse possible étant "chaque variable d'une méthode commence par la lettre m". Cette question est liée au souvenir car la réponse est répétée telle qu'elle a été apprise (pas d'interprétation). Pour *comprendre* une question est "quelle phrase décrit le rôle de la variable "spacer" dans la classe Calculator ?". Pour *appliquer*, une des questions est "quelle classe dans le système pourrait être réutilisée pour calculer des intérêts d'un compte bancaire ?". Cette question induit de sélectionner une abstraction pour l'utiliser dans un contexte différent. Pour *analyser*, une question est "expliquer comment les calculs et l'enregistrement de l'historique interagissent ?". Cette question demande une connais-

sance des relations entre deux éléments délocalisés. Pour *créer*, ce n'est pas une question qui est posée, mais une tâche qui consiste à implémenter une capacité de mémorisation à la calculatrice. Enfin pour *évaluer*, une question posée est "ce code est-il un programme orienté objet bien conçu ?".

S'il est difficile d'organiser le diagnostic suivant l'axe des techniques, nous choisissons de classer ces techniques selon l'axe des activités cognitives, à travers la taxonomie de Bloom. Nous prenons ici les définitions des catégories données par R. E Mayer [May02].

## 3.2 Se souvenir

La première catégorie de Bloom est la catégorie "*se souvenir*". Elle vise à retransmettre des informations sous une forme sensiblement identique à celle dans laquelle elles ont été capturées. Ici on récupère des connaissances pertinentes de la mémoire à long terme. Par exemple, pour pouvoir rédiger un rapport, il est nécessaire de connaître l'orthographe correcte des mots. Dans cette catégorie, les connaissances sont mémorisées indépendamment du contexte, se souvenir n'est qu'un moyen pour atteindre un but, plutôt que le but lui-même.

Il englobe deux processus cognitifs, *reconnaître* et *rappeler*. *Reconnaître*(ou identifier) implique de localiser une connaissance parmi d'autres dans une mémoire à long terme compatible avec une certaine demande. Cela correspond par exemple à l'activité permettant de répondre à "Identifier parmi les traces trace1, trace2, trace3 les contre-exemples". *Le rappel* (ou *récupération*) consiste à extraire de la connaissance pertinente de la mémoire à long terme. Cela correspond par exemple à l'objectif suivant "Rappeler les traces qui sont des contre-exemples".

### 3.2.1 Reconnaître

*Reconnaître* est la capacité à identifier des concepts à partir de données brutes. Par exemple "le fichier .trace d'OBP représente une trace", "de l'instruction 10 à 50 c'est le comportement d'un processus", "ceci est une configuration initiale", ou bien "la trace se lit du haut en bas". Ce n'est pas l'ensemble des informations brutes, ce sont des parties bien spécifiques, les concepts clés de la vérification par model checking. Reconnaître c'est donc faire le lien entre les informations produites ou requises par le model checker et les concepts clés de la vérification par model checking. *Reconnaître* c'est constituer une grille d'analyse qui permet de se retrouver dans la structure des informations produites/requises par le model checker choisi. Ces informations varient ainsi d'un model checker à un autre (Promela et LTL pour SPIN, Fiacre et observateurs pour OBP).

### 3.2.2 Rappeler

*Rappeler* succède à la reconnaissance, elle consiste à trouver l'emplacement d'une information reconnaissable et l'extraire. Il s'agit par exemple d'identifier un concept dans un environnement technique comme "ce fichier représente la trace associée à ce modèle", ou "le fichier .fiacre contient le modèle formel".

#### Interroger des traces

En model checking il est fréquent d'extraire des informations à partir des traces produites. Celles-ci peuvent être rappelées au moyen de requêtes. Le langage PQL (Program Query Language) décrit par M. Martin et al. [MLL05] permet de faire des requêtes sur des séquences d'événements associés à des objets liés. S. F. Goldsmith [GOA05] proposent le langage Program Trace Query Language (PTQL), un langage basé sur des requêtes relationnelles sur les traces d'un programme. Les traces produites résultent d'une instrumentation automatique par un outil (Particle) qui surveille certaines propriétés. Étant donné une requête PTQL et un programme Java, Particle instrumente le programme afin d'exécuter des requêtes durant l'exécution du programme.

Dans [EsS+14], K. Es-Salhi et al. proposent de définir un langage de requête, KriQL, spécialisé pour le diagnostic à partir de traces issues d'un LTS. KriQL permet la manipulation uniforme à la fois de l'espace d'état et des contre-exemples. Cette approche offre le moyen d'explorer les résultats du model checking pour en faciliter la compréhension, la localisation et l'isolation des causes des contre-exemples. Ces requêtes incluent par exemple la recherche de certains chemins entre deux configurations, comme plus court ou le plus long, selon des critères particuliers (changements de valeurs de variables).

## 3.3 Comprendre

Lorsque l'objectif est de promouvoir le transfert de connaissances, l'accent est mis sur d'autres processus cognitifs que *se souvenir*. La catégorie *comprendre* regroupe les processus capables de construire un sens à partir d'informations brutes. Cette catégorie comprend des processus comme *interpréter*, *illustrer*, *classer* ou *classifier*, *résumer*, *déduire*, *comparer* et *expliquer*.

### 3.3.1 Classifier

La *classification* (également appelée *catégorisation*) se produit lorsqu'on détermine qu'un élément (par exemple, une instance ou un exemple particulier) appartient à une certaine catégorie (par exemple, un concept ou un principe). Par exemple, classer les

éléments du système par versions. Pour pouvoir classifier il faut avant tout définir des catégories ou des classes. Cette classification dépend du domaine qui est une abstraction rigoureusement organisée et sélective de la connaissance contenue dans le cerveau d'un expert du domaine. Par exemple, le domaine du transport inclut des concepts comme les lignes de chemins de fers, les voitures et camions, les voyageurs etc....

### DSL

Le domaine peut être défini de différentes manières, par exemple au moyen d'un langage spécifique appelé DSL *Domain Specific Language*, pouvant servir à générer du code ou effectuer des analyses. Dans [Ben+], R. Bendraou et al. proposent une extension de la norme SPEM pour la modélisation de processus logiciels avec un ensemble de concepts et de sémantique comportementale permettant aux modèles de processus d'être vérifiés.

### Ontologies

Une autre façon de structurer et de représenter les données est de définir une ontologie, une spécification explicite et formelle d'une conceptualisation partagée. Elle permet de représenter le sens des concepts et des relations qui lient ces concepts. Une ontologie peut se décrire dans un langage comme OWL [1], enrichissant les triplets RDF (Resource Description Framework) (et RDFS) de relations ontologiques. Un exemple est la nomenclature systématique de la médecine - Termes cliniques (SNOMED CT), une ontologie conçue pour prendre en charge les applications électroniques en santé et en médecine [2]. SNOMED encode, stocke, et permet de retrouver de l'information liée à la santé et aux maladies utile aux systèmes informatisés.

### Problem frames

Une autre approche consiste à décrire le domaine dans des *problem frames* [Jac01]. Dans les années 90, M. Jackson a introduit ce concept pour présenter, classer et comprendre les problèmes de développement logiciels. L'approche vise à décomposer un problème général en sous-problèmes élémentaires récurrents en ingénierie du logiciel. Un *problem frame* se compose d'une machine (le logiciel produit), du domaine du problème dans laquelle la machine existe, et des exigences, c'est-à-dire des besoins identifiés. Ces éléments sont décrits au moyens de phénomènes contrôlables ou symboliques (entités, événements, états, valeurs etc...). Plus de détails sur les concepts qui permettent de décrire un *problem frame* sont donnés dans le métamodèle proposé par D. Hatebur et al. [HHS08].

1. https ://www.w3.org/OWL/
2. http ://www.ihtsdo.org/snomed-ct

L'illustration 3.2 présente un exemple de *problem frame*. Le rectangle à deux bandes représente la machine, le rectangle simple représente le domaine du problème, et l'ovale en pointillés représente l'exigence. L'exigence impose un comportement au domaine du problème, et est exprimée en termes de phénomènes contrôlables C3. L'interface entre la machine (Control Machine) et le domaine consiste en des phénomènes contrôlables partagés C1 et C2. Les points d'exclamation indiquent que les phénomènes C1 sont contrôlés par la machine et C2 par le domaine. Le C dans le coin de la boîte indique qu'il s'agit d'un domaine causal, il est autonome et réactif au phénomène C1.

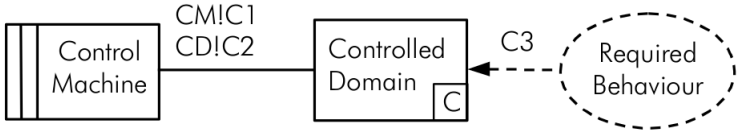


FIGURE 3.2 – Problem frame

### Patrons

Les *problem frames* encodent les principaux problèmes d'un domaine, mais pas les solutions. Un patron représente une solution à un problème dans un contexte particulier. Introduit par C. Alexander dans le domaine de l'architecture et de la conception urbaine, il est repris plus tard par E. Gamma, R. Helm, R. Johnson et J. Vlissides (surnommé le GoF) [Gam95] pour le domaine du logiciel et plus particulièrement sur la conception orientée objet. Les avantages sont nombreux, entre autres, ils constituent une solution éprouvée et optimale pour un problème récurrent, ils sont génériques et indépendants vis-à-vis d'une technologie d'implémentation. Les patrons s'appliquent à beaucoup de domaines, comme par exemple la sécurité. Au fil du temps les connaissances dans ce domaine ont pu être capturées au sein de patrons de sécurité. Tout comme les patrons d'Alexander ou du GoF, ceux-ci sont des solutions réutilisables pour un problème récurrent, la sécurité. Ils peuvent être utilisés pour analyser, construire et évaluer des systèmes sécurisés [Sch+13], et fournissent des guides détaillés quant à une solution architecturale pour un problème particulier de sécurité [FL10]. Dans [OD17], F. Obeid propose de formaliser les exigences de sécurité à travers divers patrons de sécurité. À l'aide d'un model checker, il vérifie ensuite la combinaison entre ces patrons et une architecture conformément à une politique de sécurité. Il exprime ses patrons sous la forme suivante, **nom** du patron, **fonctionnement** attendu et les contraintes associées, **structure** architecturale (diagramme de classes (UML)), **comportement** (diagrammes de séquences, automates), **propriétés** de sécurité attendues (formalisées en LTL), et **exemples** illustrant l'utilisation du patron, avec des scénarios nominaux et des scénarios d'attaque.

### 3.3.2 Expliquer

L'*explication* est par exemple la construction mentale d'un modèle de cause à effet d'un système, comme l'explication du déroulement d'une trace. Le processus de debugging qui consiste à dérouler mentalement la chaîne de cause à effet rentre dans cette catégorie.

#### Debugger interactif

Le débogage interactif permet d'observer et de contrôler une exécution afin d'en mieux comprendre le comportement donné et de déterminer la cause d'un défaut. De façon générale, le débogage fournit des fonctionnalités permettant de suspendre et d'avancer pendant une exécution, notamment par le biais de points d'arrêt, conditions dans lesquelles l'exécution peut être interrompue. Suivant cette méthode, l'exécution est toujours explorée de manière prospective. Lorsqu'elle est en pause, l'observation est généralement complétée par des vues, telles que la pile d'appel de méthode ou les valeurs de toutes les variables existantes. Certains debuggers sont spécifiques à un domaine, c'est-à-dire spécifiques à un DSL exécutable unique [VVV17].

#### Debugger omniscients

Contraignant à l'usage, le débogage interactif a été étendu au débogage dit omniscient [PT09], qui inclut la possibilité de définir des points d'arrêt dans le passé, d'exécuter le modèle en arrière ou de revenir en arrière sur une instruction précédemment exécutée. Le débogage omniscient évite de devoir redémarrer complètement une exécution pour revenir à un état précédemment atteint. Pour pouvoir remonter dans le temps, ces approches nécessitent de stocker les états ou événements passés dans une trace d'exécution.

Si le débogage omniscient est monnaie courante pour des langages généraux, il reste encore peu utilisé pour des DSLs, car son développement a un coût. Mais certaines approches tentent de le rendre accessible aux DSL, dans [Bou+18] par exemple les auteurs ont défini un débogueur générique omniscient, prenant en charge un large éventail de DSL exécutables grâce à un ensemble commun de fonctions de débogage. Cette tentative est partie prenante du projet GEMOC et de son atelier, GEMOC Studio [3].

### 3.3.3 Visualiser

L'analyse de l'exécution de grands modèles à travers les traces est presque impossible sans l'aide d'outils de visualisations. Ceux-ci exploitent la capacité des humains à rapidement comprendre des schémas visuels complexes. Il existe un grand nombre d'outils de

3. http ://gemoc.org/studio.html

visualisation permettant d'étudier les traces d'exécution, allant des diagrammes structurels se focalisant sur les composants du système et leurs relations, aux chronogrammes, en passant par des visualisations de graphes ou des diagrammes de séquences permettant de suivre des états de différents composants, variables ou encore des machines à états finis décrivant le comportement du système.

### Visualisation des traces et des automates

La présentation visuelle des contre-exemples générés est explorée par C. R. Ramakrishnan et S. A. Smolka [RS03]. Les auteurs ont mis au point un outil qui simplifie l'exploration des contre-exemples en présentant des preuves de mu-calcul modal à travers diverses vues graphiques. La mise en évidence de correspondance entre le contre-exemple généré et la propriété analysée est traitée dans leur processus de visualisation. Plus généralement les IDEs de model checking comme SPIN [Hol03] ou OBP fournissent différents types de diagrammes. Des diagrammes de séquence permettent de visualiser les traces, des automates aident à comprendre le comportement des processus, et des graphes permettent de visualiser le résultat de l'exploration (LTS, traces).

Les visualisations peuvent être complétées par des animations. Les diagrammes de séquence de SPIN peuvent être animés, soit en laissant le choix des transitions exécutées lors de la simulation à l'utilisateur (mode guidé), soit aléatoirement. Dans ce même IDE, les instructions Promela de la spécification peuvent être mises en évidence au fur et à mesure de l'exécution. Dans le model checker UPPAAL [LPY97], les simulations mettent en évidence les transitions et les états des automates qui sont impliqués dans le pas d'exécution. Dans [Mao08] S. Maoz et D. Harel proposent de combiner des modèles de conception avec des traces, afin de suivre le comportement de ces modèles tout au long de l'exécution. Ces modèles représentent des scénarios inter-objets (diagrammes de séquence) et ou des comportements internes (statecharts). L'approche permet de suivre leur progression au cours de l'exécution du système à travers différentes abstractions du niveau des modèles comportementaux définis durant la conception. Les auteurs ont développé un outil de visualisation et d'interaction permettant à un utilisateur humain d'explorer et d'analyser des traces basées sur des modèles qui sont longues et complexes, et qui seraient très difficiles à gérer manuellement dans leur forme textuelle.

### Visualisation spécifique au domaine

Une difficulté est l'interprétation du sens des traces par rapport au domaine du problème d'origine. La visualisation fait alors parfois intervenir des connaissances spécifiques au domaine. Ce nouveau point de vue permet de valider un modèle de conception par rapport à des exigences spécifiées de manière informelle et d'interpréter la signification des résultats d'analyse par rapport au domaine du problème [Mag+00].

Par exemple, StateMate [Har+88] prend en charge l'animation de modèles via un ensemble de widgets graphiques liés à un domaine. Dans [Mag+00], les auteurs décrivent une approche s'appuyant sur la base sémantique de l'entreprise, sur laquelle sont construites les animations associées au modèle LTS qui les pilote. Les modèles de comportement spécifiés sous la forme de LTS peuvent générer des animations graphiques. Les animations sont décrites par un document XML utilisé pour générer un ensemble de JavaBeans. Les JavaBeans élaborés exécutent les actions d'animation comme définit par le LTS. Dans [MTZ16], F. Muram et al. cherchent à faire un lien entre les modèles de processus BPMN décrivant le comportement du système à différents niveaux d'abstraction, et des contre-exemples produits par leur vérification par model checking. Ils présentent visuellement les éléments de modèles impliqués de sorte que les développeurs puissent facilement comprendre et résoudre les problèmes, car ils s'adressent à des utilisateurs non initiés aux méthodes formelles.

### 3.3.4 Simplifier

*Simplifier* (ou abstraire) est le fait de résumer d'une information. Par exemple, résumer une trace suivant une certaine facette. Un exemple de simplification est le model checking conditionnel. Cette technique génère une condition décrivant l'espace d'état vérifiant une propriété avec succès, et permettant d'indiquer au model checker les parties restantes du système à vérifier. Cette technique évite de vérifier à nouveau des parties d'ores et déjà vérifiées pour accélérer les prochaines vérifications [BW13].

En dehors de la simplification de l'espace d'état, les propriétés peuvent elles aussi être simplifiées, car une des principales difficultés dans l'adoption des méthodes formelles est que les ingénieurs ne sont pas suffisamment familiers avec les formalismes et stratégies de spécifications. M. B. Dwyer et al. [DAC99] ont proposé un ensemble de patrons de propriétés, représentant les pratiques les plus courantes pour la spécification de systèmes concurrents et réactifs. Ces patrons sont applicables à un certain nombre de formalismes. Par exemple, le patron *precedence* noté *S precedes P*, décrit la relation entre une paire d'événements/états où l'occurrence du premier est une précondition nécessaire à l'occurrence du second.

La méthode CEGAR [Cla+00] s'intéresse à une version abstraite du modèle du système. Supposons une abstraction $M_0$ d'un modèle $M$. $M_0$ autorise plus de comportements que $M$, donc si $M_0$ vérifie la propriété de sûreté $P$, $M$ aussi. A l'inverse si $M_0$ ne la vérifie pas, il n'est pas possible de conclure sur $M$. CEGAR consiste à produire une abstraction $M_0$ à partir de $M$, puis vérifie sur $M_0$ la propriété. Si la propriété n'est pas vérifiée, le contre-exemple obtenu par $M_0$ est également un contre-exemple de $M$. Si le contre-exemple n'existe pas dans $M$ (il est produit par l'abstraction), $M_0$ est raffinée en écartant le contre-exemple. Ce processus peut être ensuite réitéré.

### 3.3.5 Interpréter

*Interpréter* (également appelée *corréler*) se produit lorsqu'un ingénieur est capable de convertir des informations d'une forme de représentation à une autre. Par exemple, "traduire des exigences sous forme de phrases en formules logiques". Dans la plupart des projets de développement, chaque problème est analysé par les experts en s'appuyant sur des formalismes bien adaptés pour produire leur point de vue sur une solution. Les modèles produits sont de niveaux sémantiques différents et appartiennent à plusieurs espaces techniques. Pour garantir une cohérence globale entre les modèles ou bien de créer des vues transversales à partir de ces mêmes modèles, l'approche par fédération de modèles [Guy+13] permet de réaliser des corrélations entre modèles hétérogènes issus de divers espaces techniques et sémantiques.

### 3.3.6 Inférer

*Inférer* (aussi appelé conclure, extrapoler, interpoler ou prédire) implique de tirer une conclusion logique à partir d'informations présentes. Par exemple, inférer que des propriétés violées dépendent d'événements identifiables de l'environnement.

Les ontologies structurent l'information de telle sorte que de l'inférence soit possible pour un domaine. Elle est décrite par un ensemble de propriétés (axiomes) offrant des capacités d'inférence logique. De nombreux systèmes de diagnostic profitent de cette capacité, comme ODDIN [Gar+10] dédié au diagnostic de médecine interne.

### 3.3.7 Exemplifier

*Exemplifier* (également appelé *illustrer* ou *instancier*) se produit lorsqu'on trouve un exemple ou une instance spécifique d'un concept ou d'un principe général. Par exemple, "identifier différents contre exemples".

Choisir les exemples de traces peut se faire à différents niveaux. Dans OBP par exemple, il est possible de sélectionner un ensemble de configurations intermédiaires parmi lesquelles la trace (potentiellement un contre-exemple) doit impérativement passer. Dans SPIN, il existe de multiples options pour sélectionner différents contre-exemples, en paramétrant une profondeur de recherche dans le but de trouver la séquence d'exécution la plus courte possible qui viole une assertion donnée. Une telle recherche tronquée, cependant, n'est pas garantie de trouver toutes les violations possibles, même dans la profondeur de recherche.

Décrit dans [GKL04], l'outil *explain* utilise des métriques de distance sur les exécutions de programme pour fournir une assistance automatisée dans la compréhension et la localisation des erreurs dans les programmes ANSI-C. *Explain* peut par exemple produire automatiquement une trace aussi semblable que possible à l'exécution défaillante, mais ne

violant pas la spécification, produire automatiquement un nouveau contre-exemple aussi différent que possible du contre-exemple initial, ou bien encore déterminer les dépendances de cause à effet entre les prédicats d'une exécution. L'outil *explain* est intégré à CBMC, un model checker pour le langage C.

### 3.3.8 Comparer

*Comparer* implique la détection de similitudes et de différences entre deux éléments. Par exemple, déterminer les différences entre deux traces.

De nombreux auteurs ont cherché à distinguer les zones de proximités entre contre-exemples et traces saines. Dans [BNR03], T. Ball et al. ont introduit une approche implémentée pour l'outil SLAM pour comparer des contres-exemples avec des exécutions saines pour en isoler les états/transitions susceptibles d'être à l'origine du contre-exemple. Une fois qu'une cause est découverte, ils vérifient par model checking un modèle restreint dans lequel le système n'est pas autorisé à exécuter les transitions amenant aux causes, et déterminer ainsi d'autres causes possibles. Dans la même veine, N. Sharygina et D. Peled [SP01] proposent une exploration du voisinage d'un contre-exemple.

Dans [GV03], A. Groce et R. Joshi dissocient les parties d'information d'un contre-exemple produit par un model checker pouvant être à l'origine de l'erreur. L'originalité de l'approche tient dans une analyse approfondie des différences entre plusieurs versions d'une erreur (et des exécutions similaires ne générant pas d'erreur). Ainsi, ils distinguent les traces positives, c'est-à-dire proches du contre-exemple mais n'amenant pas à un état d'erreur, des traces dites négatives, c'est-à-dire des traces proches du contre-exemple et aboutissant à une erreur. Ces exécutions sont ensuite analysées pour produire une description succincte des éléments-clés de l'erreur, incluant par exemple l'identification de portions du code source essentielles pour distinguer les exécutions défaillantes des exécutions non défaillantes.

## 3.4 Appliquer

*Appliquer* implique l'utilisation de procédures pour résoudre des problèmes. Cette catégorie comprend deux processus cognitifs : *exécuter* lorsque la tâche est familière à l'ingénieur ; et *mettre en œuvre* lorsque la tâche est non familière à l'ingénieur.

### 3.4.1 Exécuter

*Exécuter* (ou *contrôler*) se produit lorsqu'un ingénieur applique une procédure pour effectuer une tâche familière. Le model cheker SPIN par exemple peut exécuter l'exploration exhaustive à partir d'un ensemble d'arguments comme la profondeur de recherche

ou bien encore le temps d'exécution. Certaines propriétés peuvent être automatiquement vérifiées, par exemple la recherche des boucles sans progression, ou bien la présence de deadlocks.

### 3.4.2 Générer

*Générer* (ou *implémenter*) se produit lorsqu'on applique une ou plusieurs procédures à une tâche, mais cette fois inconnue. Par exemple, générer un lot de propriétés à vérifier. Ici, il faut non seulement appliquer une procédure (générer), mais aussi s'appuyer sur une compréhension conceptuelle du problème (catégorie *compréhension*).

La définition des propriétés directement exploitables par un model checker n'est pas une tâche triviale. Elle dépend des compétences des ingénieurs en vérification car il faut à la fois comprendre les spécifications et la logique temporelle. Ces définitions sont vulnérables aux erreurs humaines et la quantité de propriétés à fournir est très importante. Dans [Sil+13], W. Silva et al. proposent de définir les propriétés à l'aide d'un processus automatisé. Les propriétés sont directement générées sous forme d'assertions via les machines à états finis et les diagrammes de séquence définis dans la spécification, puis vérifiés par l'outil ABV.

Dans [Bol+14], M. L. Bolton apportent une aide à la vérification formelle de systèmes d'interaction homme-machine, qui incluent souvent des interactions système que les concepteurs ne peuvent pas anticiper. Pour remédier à cela, ils présentent une méthode permettant de générer automatiquement des propriétés de spécification à partir de modèles de tâches, puis de les vérifier formellement.

Dans [OD17], F. Obeid et P. Dhaussy cherchent à sécuriser, conformément à une politique de sécurité donnée, une architecture SCADA en s'appuyant sur la réutilisation de patron de sécurité et de leurs propriétés abstraites. Cette approche est illustrée par la figure 3.3. Après transformations, le model checker apporte la preuve que les modèles d'architecture générés au format FIACRE et dotés des mécanismes basés sur les patrons de sécurité, respectent les exigences décrites dans la politique de sécurité choisie au regard d'attaques.

## 3.5 Analyser

L'*analyse* est la décomposition des informations en sous informations et l'identification de relations entre ces sous informations et leur structure globale. Cette catégorie comprend les processus cognitifs de *différenciation*, d'*organisation* et d'*attribution*. Elle différencie les éléments pertinents ou importants d'une information et examine leur organisation. L'analyse est un prolongement de *compréhension* ou un prélude à l'*évaluation* ou à la

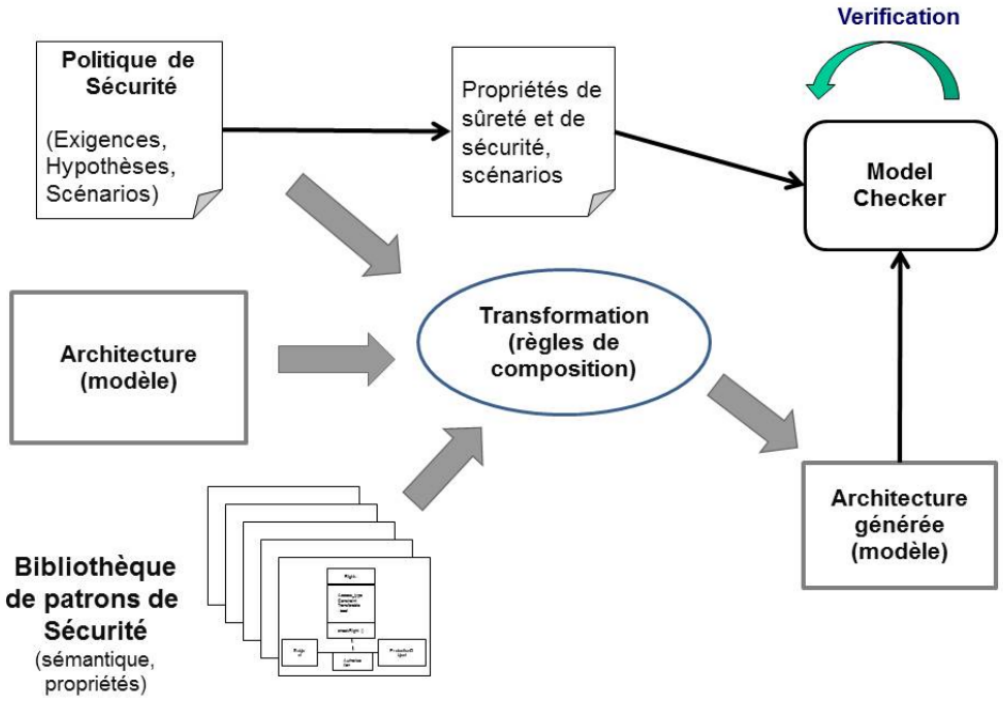


FIGURE 3.3 – Application de patrons de sécurité et génération de propriétés

*création*. Dans le cadre de la vérification par model checking, de nombreuses analyses se basent sur la différenciation ou l'organisation des traces (*trace-based analysis*).

### 3.5.1 Différencier

*Différencier* est une activité cognitive basée sur le principe de séparer les parties intéressantes des parties moins intéressantes d'une information, ce terme étant équivalent à discriminer, distinguer ou se concentrer. Par exemple, on peut exposer les différences entre des traces en succès et des traces en rejet. Différencier inclut généralement le processus de *comparer*.

#### Réduire

Étant donné que le nombre d'états à explorer à l'aide d'un model checker peut croître de manière exponentielle, un des aspects les plus importants du diagnostic est de *réduire* l'espace d'état. Une première solution consiste à exploiter la commutativité des transitions concurrentes pour réduire la taille de l'espace d'état [Hol94]. Le model checking symbolique est une autre approche [Bur+92], dans laquelle les états sont représentés à l'aide de fonctions booléennes. Par exemple supposons un système décrit par deux variables v1 et v2, et que (11,01,10) soient les trois combinaisons de valeurs possibles pour celles-ci. Plutôt

que de manipuler cette liste d'états il est plus efficace de gérer une fonction booléenne qui la représente (comme $v1 \vee v2$). Mais ces techniques sont limitées à des contraintes mémoire pour stocker et manipuler ces diagrammes de décision binaire (BDD), contraintes adressées par le model checking borné (BMC) [Bie+03]. Le principe est de rechercher un contre-exemple dans les exécutions dont la longueur est limitée par un entier k. Si aucun bug n'est découvert, on augmente la valeur de l'entier k jusqu'à ce qu'un bug soit trouvé.

### Isoler

*Isoler* est l'activité qui consiste à déterminer quels sous ensembles d'éléments (parties de programmes ou modèles) doivent être modifiés pour corriger l'erreur [CZ05], [JM11], [RR03], [Wei84]. Dans cette étape, le sous système fonctionnel à l'origine de l'anomalie est identifié. Ceci est réalisé au travers de différentes techniques, comme les techniques de découpage (*slicing*) [Wei84] ou sa variante le *dynamic slicing* [KL90]. A partir d'une variable donnée qui contient une valeur incorrecte à un moment dans le programme, la technique de slicing analyse les dépendances et détermine les instructions dans le programme pouvant affecter cette variable. Le résultat est un programme réduit qui contient la faute. Cette technique est applicable dans un contexte de model checking, comme le montrent les travaux de W. Visser et al. [Vis+03]. Ceux-ci ont abouti au développement d'un environnement de vérification et de test pour Java, appelé *Java PathFinder* (JPF). L. Millett et T. Teitelbaum [MT98] réalisent également un *slicing* du langage de programmation Promela pour le diagnostic de protocoles de communications.

Le *delta debugging* est une technique imaginée par A. Zeller [Zel02] permettant d'isoler les variables et les valeurs pertinentes d'une exécution en réduisant systématiquement la différence entre une exécution réussie et une exécution en échec. Les exécutions en erreur sont alors minimisées. Ce principe est étendu dans des approches de débogage automatique afin de révéler les chaînes de cause à effet menant à l'erreur. Dans le contexte de model checking, il s'avère que cette technique a inspiré A. Groce et W. Visser [GV03] pour la réduction de trace.

*BugAssist* [JM11] est un outil qui utilise des algorithmes de model checking basés sur la satisfabilité booléenne (SAT). La SAT analyse ainsi des contrats fournis par les utilisateurs afin de déterminer la partie du code responsable de leurs violations. Les techniques de model checking [CK04] qui produisent un contre-exemple ou une séquence d'état transitions amenant à la défaillance ont également été utilisées pour localiser les états en erreur. T. Ball et al. [BNR03] ont introduit une approche qui compare les contre-exemples renvoyés par le model checker avec les traces réussies pour identifier les séquences d'état transition erronés.

### 3.5.2 Organiser

*Organiser* implique de déterminer comment les éléments fonctionnent ensemble dans une structure. Par exemple, on veut être capable de reconnaître les dépendances entre les processus.

#### Séparer

Dans [JRS02], H. Jin et al. augmentent le contre-exemple avec des informations relatives à une alternance de segments, ceux-ci montrant soit des progrès inévitables du contre-exemple vers l'erreur, soit des choix qui, s'ils étaient évités, auraient pu empêcher l'erreur. Il faut alors déterminer si le segment en question est inévitable ou essentiel pour provoquer l'erreur. Dans cette approche, non seulement les parties de la trace sont distinguées, mais elles sont aussi organisées en différents types de segments.

Les approches reposant sur l'idée du *Divide-And-Conquer* consistent à réduire l'ensemble des calculs ou des variables d'état pouvant être à l'origine de l'échec. E. Y. Shapiro [Sha83] a travaillé sur l'idée de diviser récursivement l'arbre de calcul d'un programme en sous-arbres. Chaque sous-arbre correspond à un sous-calcul dont la sortie doit être validée. Les sous-arbres dont la sortie est incorrecte sont les emplacements des défauts.

La technique cause-transitions [CZ05], appartenant à la suite de techniques Delta-debugging de Zeller, utilise l'idée du *Divide-And-Conquer* pour lier les échecs à des sous-ensembles de variables d'état. Une recherche binaire sur les états de la mémoire entre un test réussi et un test infructueux est effectuée pour déterminer un ensemble minimal à l'origine de l'échec. Un débogueur symbolique est utilisé pour comparer et échanger les états de la mémoire entre les deux exécutions aux points de programme souhaités pour observer l'effet sur la sortie.

#### Localiser

Les approches basées sur les couvertures de tests (*test-spectrum*) [Abr+09] se basent sur la découverte de coïncidences statistiques entre les défaillances du système et l'activité des différentes parties d'un système. Elles se servent d'abstractions des traces du programme pour trouver des relations statistiques entre des endroits dans le code source et des défaillances observées. La couverture de code indique les parties du programme testées qui ont été couvertes durant l'exécution. Il est possible d'identifier les composants impliqués dans la défaillance, réduisant ainsi la recherche aux composants en faute [Abr+08]. Ces techniques peuvent appliquer différents coefficients de mesures de similarités [JHS], [AZV06].

La localisation de défauts dans les conceptions VHDL, explorant le contre-exemple produit par le model checker à l'aide de la technique de diagnostic basée sur un modèle,

est présentée dans [PW04]. Dans cette approche, un modèle de diagnostic est produit à partir d'une trace d'exécution en utilisant un contre-exemple spécifique.

Comme les erreurs peuvent se produire tout au long du programme, il faut regarder à la fois dans l'espace des variables (valeurs pertinentes), et dans l'espace temporel (moments où les transitions amenant à la cause se produisent). Dans [CZ05], H. Cleve et A. Zeller se concentrent sur ces aspects. Dans l'espace, ils identifient la différence entre les états d'un modèle lors d'une exécution en échec avec les états d'une exécution sans échec (variables, valeurs pertinentes). Cette différence est automatiquement réduite en utilisant le Delta Debugging [Zel02], et révèle ainsi la chaîne de cause à effet de l'échec (variables et les valeurs qui ont provoqué l'échec). Sur le plan temporel, ils se concentrent sur les transitions amenant à la cause, c'est-à-dire à des moments où une variable cesse d'être une cause de défaillance alors qu'une autre variable le devient. Ces transitions localisent les défauts qui causent la panne.

#### Raisonner sur des règles

Les systèmes experts [Ign91], [GR94] reposent sur une approche heuristique basée sur l'expérience passée des experts d'un système, et se présentent sous forme de règles générales. Ils sont généralement réputés rapides et robustes car les raisonnements sont peu complexes. MYCIN est un exemple de système expert dans le domaine du diagnostic médical [Sho76]. La difficulté de l'usage des systèmes experts réside dans l'acquisition de l'expertise. Dans des conditions de systèmes évolutifs et complexes, il est fastidieux de couvrir toutes les règles, et de s'adapter à l'évolution du problème ou du système traité sans une remise en cause de l'intégralité de la base de règles. De ce fait, n'incluant que partiellement les règles, ces systèmes peuvent parfois amener à des solutions erronées.

#### Raisonner sur les modèles

Les méthodes basées sur les modèles reposent sur l'existence d'un modèle explicite du système. Parfois quantitatifs, ils s'expriment au travers d'une représentation formelle et mathématique, comme une équation différentielle ; parfois qualitatifs, ils constituent une vue schématique du système. Un système est généralement modélisé par des descriptions (*System Descriptions* ou *SD*), c'est-à-dire le comportement normal des composants ainsi que leurs connexions, l'ensemble des composants du système (*COMPS*) et l'ensemble des observations disponibles sur le système (*OBS*), c'est-à-dire les valeurs connues de certaines variables.

Quand le diagnostic repose sur le comportement normal du système, on parle d'approches basées sur la cohérence (consistency based) [Rei87]. L'objectif du diagnostic est de trouver, à partir d'un comportement anormal du système, les parties (sous-ensembles de

*COMP*) du système qui en sont responsables. Cela sous-entend que le système puisse être décrit à travers un ensemble de composants, ayant chacun la responsabilité d'une partie du comportement global. Les causes des dysfonctionnements sont simplement identifiées en comparant le comportement courant du système (*OBS*) avec celui prédit par le modèle (*SD*). À partir d'un ensemble de symptômes observables, exprimant des divergences, le raisonnement consiste à rejeter un ensemble d'hypothèses pour rétablir la cohérence entre le comportement attendu et les observations.

Quand le diagnostic repose sur les relations entre les causes et les symptômes, on parle alors d'approches par abduction (*abductive-based*) [PW03], [HCK92]. De tels modèles peuvent être représentés au moyen de modèles de propagation de fautes. Ceux-ci décrivent les effets des fautes de façon causale, où chaque effet peut propager une suite d'effets supplémentaires. Des prédictions peuvent ainsi être faites sur les effets d'une faute de cause première. Inversement, en regardant les symptômes, il est possible de déterminer les causes possibles. C'est l'objectif des digraphs signés ou les arbres de fautes [Iri+79] qui sont utilisés pour l'analyse de la fiabilité et la sécurité des systèmes. Ce sont des arbres logiques qui propagent des événements primaires ou des fautes jusqu'à l'événement principal ou danger. Ils sont composés de couches et de nœuds ou chaque nœud supporte un opérateur logique, ET, OU. D'autres représentations peuvent être utilisées, comme les réseaux de Petri [Pet81], ou bien des automates de diagnostic [Pen02].

Si dans les approches basées sur la cohérence, on cherche à expliquer les incohérences après les avoir détectées, les approches par abduction cherchent à expliquer les observations et non plus nécessairement les incohérences.

M. Bourahla [Bou09] combine le *Model based diagnosis* avec le model checking pour le diagnostic de systèmes physiques et le débogage de programmes logiciels. Pour le diagnostic, le système physique est modélisé par un ensemble de composants connectés, qui sont utilisés par le model checker conjointement avec des observations (obtenues par simulation) exprimées sous forme de propriétés à satisfaire. Ainsi, l'approche permet de vérifier des combinaisons de composants défectueux et non défectueux. Pour le débogage, le programme est transformé dans un modèle de diagnostic utilisé par le model checker afin de vérifier la satisfaction des propriétés de conception. Si au moins une propriété est violée, les sous-ensembles de composants défaillants et les instructions associées à ceux-ci sont sélectionnés.

### Raisonner sur les connaissances passées

Les méthodes basées sur l'historique des données s'appuient sur la disponibilité d'une large quantité de données, glanées au fil des exécutions du système. Ces données brutes sont transformées en connaissances afin d'être traitées par le système de diagnostic.

Acquérir de la donnée au fil du temps permet d'appliquer des analyses statistiques

ou probabilistes, ou de rechercher des corrélations. S'appuyant sur les méthodes d'analyse statistique à variables multiples, les *Statistical Process Monitoring*(SPM) sont des ensembles de méthodes statistiques permettant de caractériser et suivre l'état normal ou anormal d'un processus [Joe03]. Les réseaux bayésiens [Cha91] représentent les dépendances causales entre des variables sous forme de graphes orientés où les arcs symbolisent des probabilités conditionnelles [Bay70]. Les arbres de décision [Qui86], [Qui93] reposent sur la représentation des possibilités via un arbre dont les feuilles sont les conclusions au problème. Chaque nœud illustre un test sur les valeurs d'un ensemble de variables. Les fouilles de données ou data-mining par exemple, sont des techniques qui se focalisent sur l'extraction de connaissances. Les méthodes d'apprentissage par règles d'associations [SA97] sont principalement utilisées pour la découverte de corrélations fréquentes entre deux ou plusieurs variables stockées dans de très importantes bases de données.

Les méthodes basées sur les données ne nécessitent pas de connaissances approfondies, mais exigent un long apprentissage. Le *machine learning* se définit comme un ensemble d'outils statistiques ou géométriques et d'algorithmes informatiques permettant d'automatiser la construction d'une fonction de prédiction f à partir d'un ensemble d'observations que l'on appel l'ensemble d'apprentissage [Lem+16].

La programmation logique inductive (ILP) [Sha83], est une approche spécifique d'apprentissage basée sur la logique. D. Alrajeh et al. [Alr+13] combinent cette approche avec le model checking pour générer un ensemble d'exigences opérationnelles complet par rapport à des objectifs donnés. L'apprentissage permet de calculer les exigences opérationnelles à partir des contre-exemples et des traces positives. Les exigences opérationnelles apprises sont garanties pour éliminer les contre-exemples et être cohérentes avec les objectifs. Ce processus est exécuté de manière itérative jusqu'à obtention d'un modèle correct.

## 3.6 Créer

*Créer* implique de rassembler des éléments pour former un nouveau tout cohérent ou fonctionnel. Le processus de création peut être divisé en trois phases : (a) la représentation du problème, où l'on tente de comprendre la tâche et de proposer des solutions ; (b) la planification de solutions, dans laquelle on examine les possibilités et élabore un plan viable ; et (c) l'exécution de la solution, dans laquelle on exécute le plan avec succès. Ainsi, le processus de création peut être considéré comme commençant par une phase d'exploration (i), au cours de laquelle diverses solutions possibles sont envisagées, une phase de convergence (ii) au cours de laquelle une solution est conçue puis transformée en un plan d'action qui est exécuté à mesure que la solution est produite. Créer comprend trois processus cognitifs : *corriger*, *planifier* et *produire*.

### 3.6.1 Corriger

*Corriger* consiste à élaborer de nouvelles hypothèses permettant de résoudre un problème. Par exemple, implémenter une correction permettant de valider une propriété qui est jusqu'alors violée.

F. Buccafurri et al. [Buc+99] proposent de suggérer des réparations pour des types d'erreurs très spécifiques (telles que l'affectation de variable ou des instructions consécutives inversées) dans les programmes concurrents. Les auteurs s'appuient pour ça sur un raisonnement abductif.

Dans [Alr+15], la réparation s'appuie sur de l'apprentissage basé sur la logique. D. Alrajeh et al. proposent un cadre produisant à partir d'une description formelle donnée et d'une propriété, une description modifiée garante de la satisfaction de la propriété. Le model checking détecte automatiquement les erreurs dans la description formelle. L'apprentissage effectue alors les tâches de diagnostic et de réparation des erreurs identifiées, afin de fournir une description correctement révisée.

### 3.6.2 Planifier

*Planifier* implique de concevoir une méthode pour accomplir certaines tâches, comme diviser une tâche en sous-tâches à exécuter lors de la résolution du problème. Par exemple, pour corriger le modèle, il faut isoler l'erreur, générer une solution qui corrige l'erreur et revérifier jusqu'à ce qu'il n'y ait plus d'erreurs.

#### Raisonnement par cas

Initialement fondé sur les travaux de R. Schank [Sch86], le raisonnement à base cas ou *Case Based Reasoning* (CBR) est une méthode d'apprentissage et de résolution de problèmes. Cette méthode s'inspire du raisonnement humain qui fonctionne souvent par analogie. Un nouveau problème est alors comparé à des problèmes passés, appelés cas, qui ont pu être résolus. La solution passée est ensuite copiée ou adaptée pour produire une nouvelle solution. Quand celle-ci résout le problème, ce nouveau cas alimente une base d'expériences. Le raisonnement par cas est présenté comme un processus cognitif *analyser* mais il est plus approprié de le voir comme une extension du processus *comprendre* et un prélude aux processus *évaluer* et *créer*.

Le cycle de fonctionnement est défini comme illustré sur la figure 3.4 extrait de [AE94].

**Retrieve**. Après qu'une situation problématique ait été identifiée, la première étape consiste à récupérer dans la base de cas celui qui est le plus pertinent vis-à-vis du problème rencontré. Au cours de cette étape, un algorithme (algorithme des plus proches voisins, recherche hiérarchique...) est utilisé. Il sélectionne les cas potentiels et les classe selon leur degré de similarité, afin de sélectionner le plus proche du problème courant.

**Reuse**. Une fois un cas récupéré, la seconde étape est la réutilisation de ce cas dans la nouvelle situation. Une possibilité de réutilisation est la copie : la solution passée est directement transférée dans la solution présente. Mais généralement quand un cas est retrouvé il ne correspond pas exactement à la solution présente, et doit donc être adapté. Dans ce cas, deux méthodes se distinguent : réadapter la solution passée (*transformational reuse*), ou bien réutiliser la méthode passée qui a permis de construire la solution (*derivational reuse*).

**Revise** Une nouvelle solution est réalisée, inspirée des solutions passées qu'il faut maintenant évaluer. L'évaluation peut se faire soit avant l'application de la solution au problème, soit après. Dans tous les cas, si la solution n'est pas satisfaisante, cette solution doit être à nouveau adaptée, ou bien un nouveau cas doit être récupéré.

**Retain** Pour finir, quand la solution est jugée satisfaisante, la dernière étape consiste à mettre à jour la base de cas avec le nouveau cas.

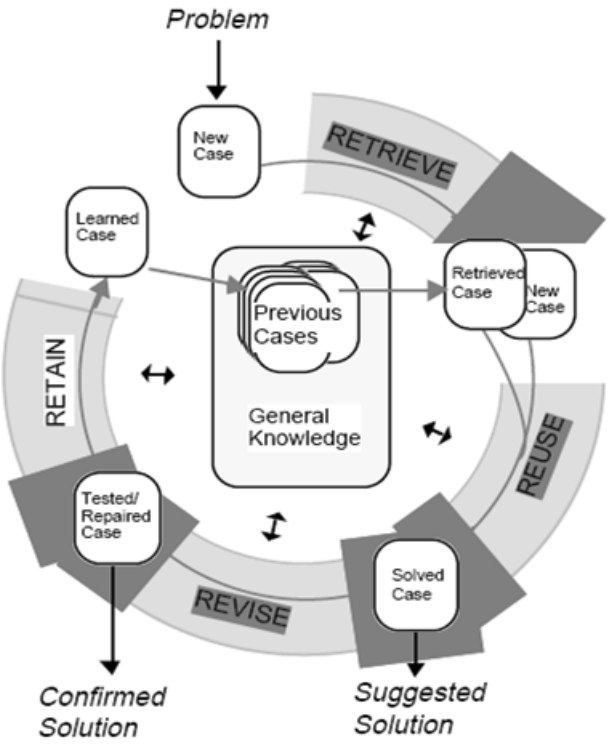


FIGURE 3.4 – Case based reasoning

L'un des avantages des systèmes CBR est qu'il n'ont nullement besoin d'un modèle de domaine explicite. La présence d'une quantité de cas suffit à la résolution du problème. L'implémentation est réduite à identifier des caractères signifiants qui décrivent le cas, ce qui est souvent plus simple que d'identifier un modèle explicite. De plus, leur capacité d'apprentissage leur confère une grande pérennité. En outre, leur efficacité dépend du

nombre de cas capturés [AE94]. Cette méthode est donc destinée aux applications qui disposent d'une certaine expérience [Spa01]. Le CBR est par exemple utilisé pour des tâches allant de la conception à la planification, et évidemment au diagnostic [PMT04].

Une difficulté réside dans la représentation des cas, car celle-ci impacte fortement la récupération des cas et l'évaluation de leur similarité. La plupart du temps, un cas est décrit par un problème et une solution, formulé par $case = (pb, sol(pb))$ [Mil06]. Les cas peuvent être représentés de différentes manières, des prédicats, des objets, des graphs ou simplement des vecteurs de caractéristiques.

La méthode est supporté par des outils, parmi lesquels jColibri, un framework développé en Java qui permet notamment aux utilisateurs de sélectionner parmi différents composants et configurations CBR. myCBR, un outil open source, permet de réaliser et de tester des mesures de similarités.

### 3.6.3 Produire

Produire ou construire consiste à inventer une solution. L'ingénieur reçoit la spécification d'un objectif et doit créer un produit qui réponde à cette spécification.

#### Une activité de résolution de problèmes

H. Simon [Sim73] énonce que la caractéristique fondamentale de la conception est d'être une activité de résolution de problèmes.

Ces problèmes sont bien souvent plus ou moins mal structurés (mal définis) car les spécifications d'un projet de conception le spécifient habituellement à un niveau abstrait ou par un ensemble réduit de contraintes. Simon distingue alors deux étapes dans la résolution : d'abord la structuration puis la résolution du problème bien structuré qui en résulte. La structuration est au moins aussi importante que la résolution en matière d'effort. À la construction de représentation de problème puis la génération de solutions peut s'ajouter l'étape de l'évaluation de solution [Vis09].

Dans la réalité, la conception ne sépare pas l'analyse du problème et l'élaboration de solutions en deux activités distinctes, consécutives, car celles-ci progressent en parallèle. De plus, les concepteurs génèrent sans cesse de nouvelles tâches et redéfinissent les contraintes de celles-ci, et restructurent donc continuellement le problème [Vis09].

#### Stratégies de conception

La conception de logiciels fait face à de nombreux problèmes résumés par F. Darses et al. dans : [DDV01] : (i) Les problèmes à résoudre sont complexes, en cas de sous-systèmes indépendants à gérer par exemple ; (ii) La résolution de ces problèmes nécessite des compétences variées ; (iii) Les solutions choisies sont toujours plus ou moins acceptables (il y

a toujours plusieurs solutions à un problème) ; (iv) Les phases de résolution interagissent constamment et leur causalité n'est pas évidente. (v) Il n'y a pas de procédé de conception prédéterminé (s'il existe d'anciennes solutions, il faut les réadapter au nouveau contexte, s'il n'y en a pas il faut les créer).

Pour progresser plus rapidement vers une solution une stratégie fondamentale et typique des concepteurs est la réutilisation de connaissances [Jac01], [OD17], en particulier par un raisonnement analogique [Vis09], [AE94]. Pour réutiliser ces connaissances, il faut permettre d'organiser l'activité de conception.

Selon W. Visser [Vis09], la conception est opportuniste car les concepteurs "procèdent d'une façon non systématique, multidirectionnelle en formulant des plans locaux, aussi bien à des niveaux abstraits qu'à des niveaux concrets", notamment car "une telle organisation de l'activité est motivée par un souci d'*économie cognitive*"". Les concepteurs procèdent de façon entremêlée souvent de haut-en-bas, aussi bien que de bas-en-haut, comme on peut le voir dans l'approche Twin Peaks [Hal+02].

#### Twin Peaks

Les modèles de processus de développement ont évolué, des modèles en cascade aux modèles en spirale. Les modèles en spirale à grain fin sont utilisés par les méthodes agiles. La pierre angulaire de ces processus réside dans le fait que les développeurs élaborent simultanément les exigences et l'architecture d'un système, et entrelacent leur développement [SB82]. J. G. Hall, M. Jackson, R.C Laney, B. Nuseibeh et L. Rapanotti de l'Open University ont proposé une adaptation du modèle de cycle de vie en spirale, appelé le modèle Twin peaks illustré dans la figure 3.5, pour mettre l'accent sur le statut égalitaire qu'ont les exigences et l'architecture [Hal+02]. Le modèle de développement logiciel proposé est un processus itératif au cours duquel les structures de problème et les structures de solution sont détaillées et enrichies. Dans ce contexte, les chercheurs de l'Open University considère que l'utilisation du support architectural aide à se concentrer sur les exigences de conception essentielles du problème en permettant un traitement plus abstrait des problèmes de conception et des exigences comportementales [Hal+02]. Les auteurs ont étendu les *problems frames* à ces fins.

## 3.7 Évaluer

*Évaluer* est défini comme un jugement basé sur des critères et des normes (qualité, performance...) qualitatifs ou quantitatifs. Il regroupe différentes activités comme *juger* ou *vérifier* .

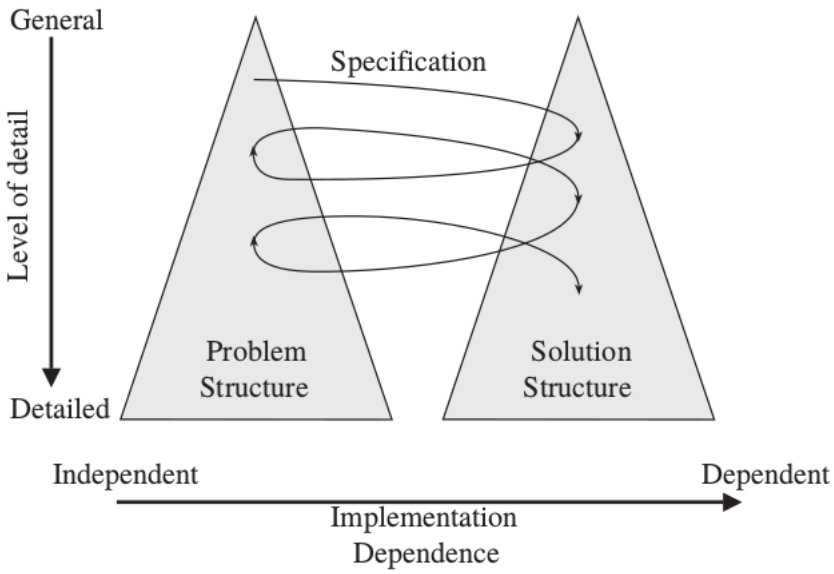


FIGURE 3.5 – Twin Peaks

### 3.7.1 Juger

Le *jugement* des diagnosticiens peut être utile pour affiner le diagnostic. Dans [Lin+17], Y. Lin et al. proposent une approche de débogage basée sur les retours des ingénieurs, qui proposent alors des recommandations quant aux étapes suspectes de la trace. Whyline [KM08] est un type d'outil de débogage permettant aux développeurs de donner un retour en sélectionnant une question quant aux résultats du programme parmi un ensemble de questions. L'outil trouve ensuite une ou plusieurs explications possibles à ce résultat en combinant divers algorithmes.

On retrouve aussi cette idée dans [Gon+12], où une technique de localisation de faute interactive intègre le retour des utilisateurs au sein d'une technique d'analyse spectrale. Chaque fois qu'un utilisateur inspecte un élément de programme suspicieux, il juge s'il est correct ou non. Il fournit alors un retour permettant de réorganiser le classement. La liste des éléments du programme est alors classée en fonction du caractère suspect des éléments.

Les techniques de debugging statistiques [JS07], [Lib+05] exploitent l'analyse statistique pour produire un classement des instructions les plus suspectes, en analysant les données collectées sur les exécutions réussies et manquées.

### 3.7.2 Vérifier

*Vérifier* permet de démontrer que le système satisfait ou ne satisfait pas des propriétés. On distingue généralement deux sortes de vérifications, dynamique et statique.

#### Vérification dynamique

La vérification dynamique est réalisée sur l'exécution du modèle du logiciel. Elle est supportée par la mise en œuvre de tests, de simulations ou d'analyses de modèles dynamiques. Pour être efficace, le programme cible doit être exécuté avec des entrées suffisamment variées. La couverture de code aide à s'assurer qu'un ensemble adéquat des comportements possibles du programme a été observé. Il existe de nombreux outils, comme Valgrind par exemple qui permet de détecter des fuites de mémoire ou des dépassements de buffer.

#### Vérification statique

L'analyse statique examine les exigences, les spécifications, les modèles ou le code source, sans pour autant en exécuter le code. Parmi les avantages, elle permet très tôt d'analyser le modèle (avant implémentation), et elle permet de répondre aux problèmes causés par la terminaison d'un programme. Les analyses sont nombreuses, entre autre l'analyse de la qualité du code (ESLint), des bugs (Findbugs), de la sécurité (Sonarqube), du respect de règles de codage (Checkstyle) etc... Le model checking est aussi une technique de vérification statique. Le model checker le plus connu est probablement SPIN [Hol03]. Il existe d'autres model checkers comme Kronos [Boz+98], NuSMV [Cimatti 1999], Uppaal [Larsen 1997], DiVIne, TLA+, ou encore OBP conçu à l'ENSTA Bretagne.

#### Gestion du processus de vérification par model checking

S'il n'existe pas de processus standard, certains travaux présentent néanmoins les grandes étapes du processus de vérification par model-checking. Dans ceux de C. Baier [BK08], des exigences sont formalisées et le comportement du système modélisé. Commence alors la phase de vérification, au cours de laquelle le modèle du système est traduit dans le langage du modèle checker. Une première analyse par simulation est réalisée jusqu'à l'obtention d'un modèle estimé fonctionnel. Puis les propriétés sont formalisées et l'exploration exhaustive réalisée par le model checker. Celui-ci renvoie un contre-exemple si la propriété n'est pas vérifiée. Dans ce cas le contre-exemple est analysé à travers des simulations. Des corrections sont apportées au système, et la procédure de vérification est répétée. Dans certains cas, le model checker manque de mémoire, il faut alors modifier le modèle en conséquence (abstraire certaines parties par exemple).

Sur la base de ces grandes étapes, différentes stratégies sont possibles. Dans [RB98], T. C. Ruys et E. Brinksma présentent deux approches de vérification par model checking. L'approche par *vérification* vise à proposer un modèle correct à un certain niveau d'abstraction. Pendant la phase de validation, tous les aspects d'un modèle abstrait du système sont validés, en abstrayant les autres parties. L'approche par *falsification* vise à détecter les erreurs et les faiblesses de la conception (initiale) d'un système. Elle se concentre sur la vérification des parties du système où les défauts sont les plus susceptibles de se produire. Dans cette approche, la phase de validation est démarrée avec un modèle à haut niveau d'abstraction. Ensuite des zooms sont effectués sur certains aspects du modèle en utilisant des techniques de raffinement locaux. Seule une partie limitée du système est alors validée.

T. C. Ruys évoque dans plusieurs de ses travaux [RB98], [RB03] la problématique de la gestion du processus de vérification, qu'il nomme la trajectoire de vérification. Le constat est que la vérification par model checking entraîne une prolifération de modèles interdépendants, à différents niveaux d'abstractions et détails. Il fait aussi apparaître la présence de cycles de vérification, faisant intervenir diverses versions de modèles. Ainsi, la vérification par model checking présente les mêmes problèmes de gestion d'informations que ceux rencontrés plus largement dans le génie logiciel. Les problèmes sont liés au suivi de divers types d'informations et de données, incluant les documents décrivant des parties du système sous plusieurs versions, les spécifications sous différentes versions, ainsi que les résultats des tentatives de vérification.

Or il est important de porter une attention particulière à la gestion du processus de model checking afin d'améliorer son application et tirer ces techniques à leur meilleur avantage. T. C. Ruys identifie alors le besoin d'un support outillé, intégrant entre-autres des outils de gestion de configuration logicielle pour gérer et contrôler la trajectoire de vérification. Le projet Xspin/Project est une extension de Xspin qui contrôle et gère automatiquement la trajectoire de validation lors de l'utilisation du vérificateur de modèle Spin.

## 3.8 Conclusion

La détection d'une propriété violée et son contre-exemple déclenche généralement le processus de diagnostic. Celui-ci doit amener l'ingénieur à comprendre la cause de ce symptôme. Dans un contexte où les informations à disposition sont complexes (grandes traces, fossé sémantique...), l'ingénieur doit être accompagné par des techniques automatisant certaines tâches. La plupart des techniques utilisées en model checking se focalisent sur l'analyse de contre-exemples. Nous avons vu qu'il existait d'autres techniques, dont certaines pouvaient accompagner le diagnostic post model checking (machine learning,

automatic repair...). Si l'étude de ces dernières semble une voie prometteuse, dans cette thèse nous faisons le choix de ne pas nous investir dans une technique particulière. Cela permet d'envisager le diagnostic comme une somme d'activités cognitives différentes qui emploient un éventail de techniques très large.

Ne pas opter pour une technique nous confronte au problème du manque de cadre. Il existe des cadres pour des domaines bien spécifiques (procédés industriels, médecine), mais pas appliqués au model checking. Comme un diagnostic est un ensemble d'activités cognitives qui peuvent être automatisées par des techniques, nous choisissons d'organiser ces techniques au sein de catégories d'activités cognitives. La taxonomie de Bloom fait référence en la matière, et a déjà été utilisée pour catégoriser des activités d'ingénierie du logiciel [SR04], [BE03b]. L'état de l'art présenté dans ce chapitre a utilisé la taxonomie pour présenter des techniques utiles pour le diagnostic.

La proposition de révision de la taxonomie de Bloom [Kra02] a fait apparaître une deuxième dimension, celle des connaissances. Il convient alors d'organiser les différentes approches autour de ces deux dimensions, les activités cognitives et les connaissances. Nous distinguerons trois catégories, les connaissances liées au model checking, les connaissances du domaine et les connaissances de gestion.

Dans le chapitre suivant, nous présentons les activités cognitives mobilisées pour vérifier le SAE, qui s'appuient uniquement sur des connaissances liées au model checking, et qui emploient les techniques associées à cette catégorie.

Chapitre 4

# Comprendre et analyser le SAE

## 4.1 Connaissances liées au model checking

Dans cette section nous définissons les connaissances liées au model checking, que nous abrégerons en *connaissances techniques*, comme l'ensemble des informations requises et produites par le processus de vérification par model checking, telles que les propriétés, le modèle formel, le LTS ou encore les traces. Ces connaissances sont toujours présentes lors de l'activité de diagnostic.

### 4.1.1 Concepts

#### SAE

Le système à l'étude (SAE) est décrit par une spécification comprenant un lot de propriétés formelles, et un modèle formel de son comportement.

#### Composant et processus

Le comportement du système est décrit par un *composant*. Il contient des variables globales (fifos par exemple) et des instances de *processus* reliés entre eux par des *canaux*. Un *processus* est une machine à état qui évolue de manière concurrente aux autres processus. Il est décrit par un ensemble de variables et d'un automate, lui-même composé d'*états* et de *transitions*. Durant l'exécution d'une transition, différentes actions sont réalisées comme l'affectation de variable, la lecture ou l'écriture dans un canal. Un des états est initial.

#### Message et canal

Un processus communique avec d'autres processus, soit de façon synchrone, soit de façon asynchrone. Dans la thèse, nous faisons le choix que les échanges sont uniquement asynchrones. Dans ce cas précis, des *messages* sont échangés via des *canaux* de type *FIFO* où l'ordre d'envoi des messages est préservé lors de la réception.

### Environnement

L'environnement communique également par envoi de messages asynchrones avec le système. Il guide ainsi l'exécution du système suivant un scénario, réduisant de ce fait son espace d'état. Au moment de l'exécution, il est instancié automatiquement sous la forme d'un processus unique.

### Graphe d'exploration

Un graphe d'exploration produit par un model checker (OBP dans notre cas) représente l'ensemble des états atteignables du modèle nommés *configurations*, composé du système (modélisé en langage FIACRE) et de l'environnement (décrit avec CDL pour OBP). Un graphe d'exploration est produit par le model checker, celui-ci possède toujours une configuration initiale et un ensemble de configurations finales.

### Configuration

Une configuration est une structure de données regroupant des informations relatives à un état du système et son environnement. Une configuration regroupe les informations sur des instances de processus et de leurs informations (variables, états), des canaux de communications, des prédicats ou des observateurs.

### Transition

Une transition est un arc entre deux configurations. Chaque transition est associée à un label. Dans le cas d'OBP il représente soit une émission de message, soit une réception de messages, soit un rendez-vous, soit un "internal" (changement interne à un processus).

### Trace

Une trace est la description d'un chemin dans le graphe d'exploration. Une trace possède une configuration initiale, et une configuration finale, correspondant généralement au rejet d'un observateur. Il peut exister plusieurs traces ayant la même configuration initiale et finale.

### Prédicats

Un prédicat permet d'évaluer des expressions basées sur les données d'exécution du modèle (variables, états...). Par exemple, déterminer si le processus *NET* est dans l'état *idle*, se traduit en CDL par $predicate\ P_1\ is\ Net1@Idle$. Déterminer si la variable *length* de la fifo $F_1$ du composant $Run_1$ est inférieure à une constante donnée *SIZE*, se traduit en CDL

par *predicate* $P_2$ *is* $Run1 : F_1.length <= SIZE$. Autre cas de figure, un événement associé à un changement de valeur d'un prédicat. Exemple, l'événement $E_1$ survient lorsque le prédicat $P_1$ devient vrai, se traduit en CDL par l'expression $E_1$ *is* $P_1$ *becomes true*.

#### Observation des propriétés

Un observateur, au sens OBP, est un automate. Selon la séquence des transitions, consécutive à l'observation des activités du système et de l'environnement, l'observateur atteint soit un état que l'on souhaite obtenir (état succès), soit un état qui ne doit pas apparaître (état rejet). L'observateur peut observer des occurrences d'événements (émission ou réception de message), ou des changements de valeur booléenne d'un prédicat. Les transitions, déclenchées par l'observation des événements (*event*), peuvent être gardées par des prédicats (*predicate*). Une version simplifiée d'une transition (non-temporisée) entre un état source (*sourceState*) et un état cible (*targetState*) est définie grâce à la syntaxe *sourceState – / predicate / event / -> targetState*. Un observateur permet d'exprimer une propriété de type sûreté (il contient un état reject).

#### Logique temporelle

La logique temporelle, comme la LTL, permet d'exprimer des propriétés et des prédicats logiques à l'aide d'opérateurs logiques (nous utiliserons ici *not* ($\neg$), *and* ($\wedge$), *or* ($\vee$) et *implies* ($\rightarrow$)) et temporels (nous utiliserons ici *globally* ($\Box$), *eventually* ($\Diamond$) et *weakly until* ($W$)). Pour exprimer de telles propriétés, il est possible d'utiliser Plug, une extension à OBP. Si les observateurs permettent d'exprimer pleinement les propriétés de sûreté, ils ne permettent pas d'exprimer la vivacité. Il faut donc utiliser la LTL pour exprimer des propriétés de vivacité.

### 4.1.2 Modèle

Reprenons le premier scénario. Le modèle à vérifier est exprimé dans le langage formel Fiacre. L'architecture est composée de 4 entités, un contrôleur global appelé *GC*, deux contrôleurs locaux $Plc_1$ et $Plc_2$ et un réseau de communication *Network* reliant le *GC* aux $Plc_1$ et $Plc_2$. Les contrôleurs locaux permettent d'accéder à une ressource partagée *Res*, chacun par un accès, respectivement $Res_1$ et $Res_2$.

L'architecture fonctionne de la manière décrite par les automates 4.1, 4.2 et 4.3. Une entité de l'environnement souhaite accéder à une ressource de l'architecture. Elle envoie un message au *GC* indiquant dans le message la demande d'accès à la ressource ($Res_1$ ou $Res_2$) et l'opération à réaliser sur cette ressource (*Read* ou *Write*). *GC* reçoit cette requête (état *Idle*), l'ignore si elle lui est directement adressée (état *ReceivedForward*, puis transition *GcIgnoring*), ou dans le cas contraire, la retransmet à un *PLC* via le

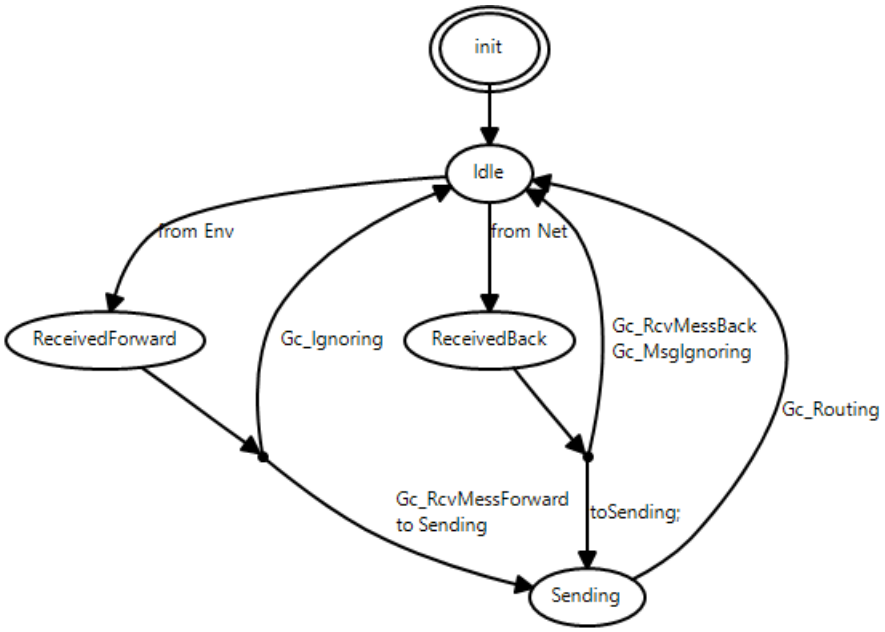


FIGURE 4.1 – Automate de GC

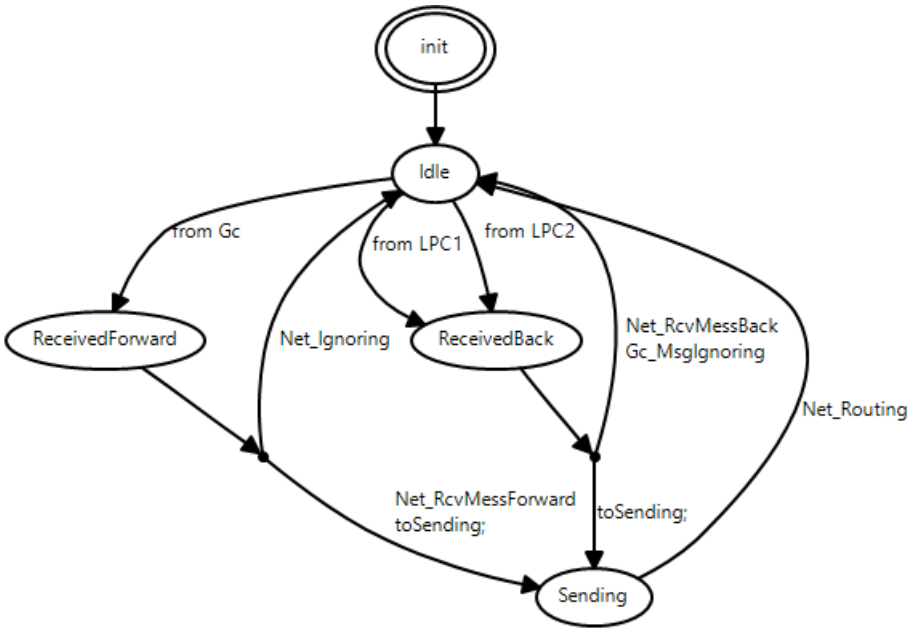


FIGURE 4.2 – Automate de NET

*Network* (état *Sending*). Le *Network* traite ensuite la requête (état *ReceivedForward*). Comme *GC*, si la requête lui est destinée, il l'ignore (retour dans l'état *Idle*), sinon il la retransmet au *PLC* concerné. Le *PLC* reçoit la requête (état *Received*), accède à la ressource (état *Access*) et envoie un message contenant une réponse (état *Sending*). La réponse est transmise à l'environnement en passant d'abord par le *Network*, puis par le *GC*, via leurs états respectifs (*ReceivedBack*).

Les entités considérées jouent des rôles différents. Le rôle $PLC_1\_OWNER$ a les droits en lecture et écriture sur $RES$ via $Res_1$ mais pas via $Res_2$. Le rôle $PLC_2\_OWNER$ a les droits en lecture et écriture via $Res_2$ mais pas via $Res_1$. Nous considérons deux clients, le $Client_1$ joue le rôle de $PLC_1\_OWNER$, et le $Client_2$ joue le rôle de $PLC_2\_OWNER$.

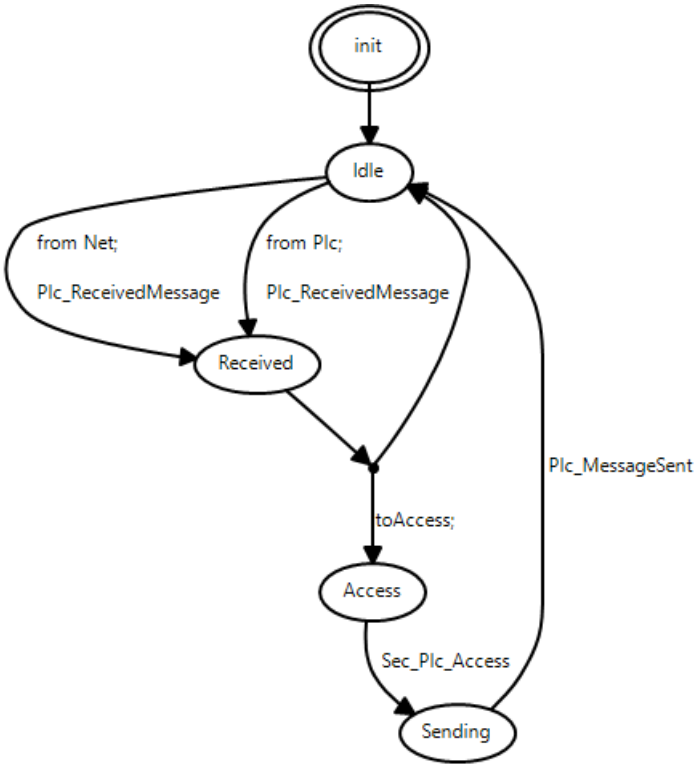


FIGURE 4.3 – Automate de PLC

Les processus sont reliés par des FIFOs de taille finie. Ici la taille initiale est 4, c'est à dire que la fifo peut contenir jusqu'à 4 messages, pas au delà. Les FIFOs supportent des messages de deux types, *SEND* (requête) ou *RECV* (réponse). Un message contient différents paramètres, l'entité source à l'origine du message (*source*), l'entité destinataire (*target*), et une donnée (*data*). Une donnée (*data*) est composée de trois informations, le type d'opération, soit lecture (*RED*) ou écriture (*WRT*), l'accès à la ressource, soit $Res_1$ ou $Res_2$, et un type d'acquittement, soit *ACK* ou *NACK*. Le listing 4.1 présente un exemple de message exprimé en Fiacre.

```
// Exemple, un message de lecture par le Client1 de la Res1 geree par le
   PLC1
CLT1_READ_PLC1 is {source = ID_CLT_1, target = ID_PLC_1, data = RED_RES1}
```

Listing 4.1 – Message

Pour simplifier, les messages sont associés à une constante qui respecte des conventions de nommage dans l'ordre suivant *type de message [envoi (SEND) / réception (RECV)], entité source, entité cible, type d'accès [lecture (RED) / écriture (WRT)], ressource*. Par exemple, une demande de lecture de la $Res_1$ détenue par le $Plc_1$ par le $Client_1$ est exprimée par la constante $SEND\_CLT_1\_PLC_1\_RED\_RES_1$.

### 4.1.3 Propriétés

Prenons par exemple deux propriétés à vérifier pour ce modèle, une de vivacité et une de sûreté. La première propriété, $pty\_CLT_1\_PLC_1\_RED\_RES_1\_Scenario_1a$, exprime que l'envoi du message $CLT_1\_PLC_1\_RED\_RES_1$ par l'entité $Clt_1$ est suivi d'une réception de $Clt_1$ du message $CLT_1\_PLC_1\_RED\_RES_1\_ACK$. Il est possible d'exprimer une telle propriété en LTL :

$$pty\_CLT_1\_PLC_1\_RED\_RES_1\_Scenario_1a : \\ \Box\{SEND\_CLT_1\_PLC_1\_RED\_RES_1 \rightarrow \Diamond RECV\_PLC_1\_CLT_1\_RED\_RES_1\_ACK\} \tag{4.1}$$

Avec la syntaxe CDL, dans l'outil OBP Plug sous la forme suivante :

```
pty_CLT1_PLC1_RED_RES1_Scenario1a is {
      [] (| SEND_CLT1_PLC1_RED_RES1 |=> <>| RECV_PLC1_CLT1_RED_RES1_ACK | )
}
```

Une seconde propriété affine la première propriété, en exigeant que si le client demande une seule fois la ressource il ne peut recevoir qu'un seul acquittement. Cette propriété, de sûreté cette fois, est exprimée en CDL par l'observateur suivant. Celui-ci rentre dans un état de rejet si deux demandes de ressources sont réalisées, ou si deux acquittements sont reçus.

```
property pty_CLT1_PLC1_RED_RES1_Scenario1b is
{
  start -- / / send_CLT_1_PLC_1_RED_RES1     / -> wait;
  wait  -- / / recv_CLT_1_PLC_1_RED_RES1_ACK  / -> onlyonce;
  wait  -- / / send_CLT_1_PLC_1_RED_RES1  / -> reject;
  onlyonce -- / / recv_CLT_1_PLC_1_RED_RES1_ACK / -> reject
}
```

Listing 4.2 – Propriété pty_CLT1_PLC1_RED_RES1_Scenario1b

### 4.1.4 Exploration

Reprenons le premier scénario évoqué dans la section 2.1.2. Luka reçoit ce modèle réalisé par un tiers. Il est en charge de sa vérification, et s'il n'est pas expert dans le domaine des SCADA, il possède une expérience significative en vérification par model checking. Ceci sous entend qu'il s'appuie sur les connaissances liées au model checking définies dans ce chapitre.

Luka suit l'approche décrite par [BK08]. Ainsi, à l'aide du model checker OBP, il génère le LTS correspondant au modèle, et l'explore. Afin de limiter l'explosion combinatoire, Luka effectue des vérifications cadrées par des scénarios différents, formalisés sous la forme de contextes CDL. Après vérification, il obtient un graphe d'exploration qu'il analyse. Il y découvre que la propriété $pty\_CLT_1\_PLC_1\_RED\_RES_1\_Scenario_1a$ est violée, indiquant que la demande d'accès à une ressource par le client ne lui renvoie jamais d'acquittement ($ACK$).

Luka démarre une phase de diagnostic en étudiant un contre-exemple. Dans OBP, une trace est divisée en deux parties, la liste des configurations et la liste des transitions qui composent le chemin passant par ces configurations, un extrait est présenté sur le listing 4.3.

```
Config: 0{
 component: '{Run}1' [FIFOs]
 proc: '{Gc}1' [@Idle, targetAvail=false, canPass=false, mess={} ]
 proc: '{Net}1' [@Idle, targetAvail=false, canPass=false, mess={} ]
 proc: '{Plc}1' [@Idle, resAvail=false, access=false, opRes={}, mess={} ]
 proc: '{Plc}2' [@Idle, resAvail=false, access=false, opRes={}, mess={} ]
 context: '{env}1' [@start]
 observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1b}1' [@start]
}

transitions: {
( 0 , "internal" , 1)
( 1 , "internal" , 2)
  ...
( 5 , "internal" , 6)
}
```

Listing 4.3 – Trace du scénario 1

Luka est confronté à deux enjeux principaux, la taille de la trace et son niveau de détail. En conséquence, pour établir un diagnostic Luka va devoir réaliser différentes activités cognitives successives.

## 4.2 Activités cognitives de diagnostic des traces

Cette section présente des activités cognitives mobilisées dans le diagnostic.

### 4.2.1 Première tentative

#### Isolation et visualisation

Dans un premier temps il cherche à analyser la progression des messages en isolant les communications entre les processus. Pour comprendre plus aisément les communications, il opte pour l'utilisation d'un outil de visualisation de trace fourni par le model checker OBP. La représentation choisie est un diagramme de séquence présenté sur la figure 4.4, mettant en avant les interactions entre les processus. Dans le diagramme chaque ligne de vie correspond à un processus. Entre deux lignes de vies, représentés par des traits verticaux, figurent les messages échangés. Les traits pointillés horizontaux représentent des changements d'états internes aux processus.

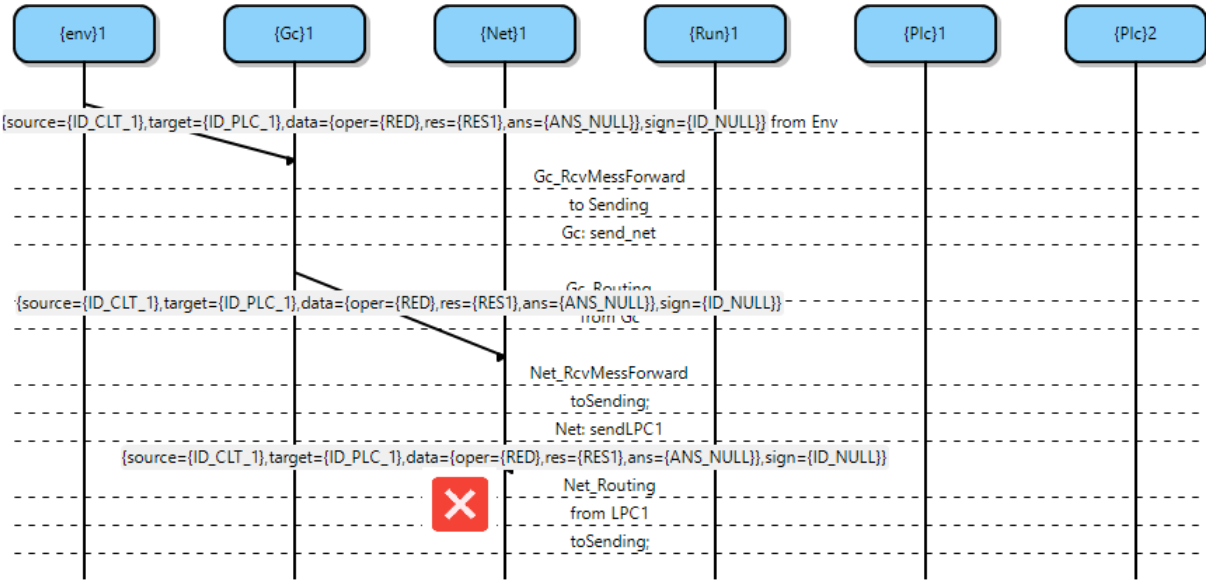


FIGURE 4.4 – Diagramme de séquence de la trace

Cette visualisation permet d'identifier aisément qu'un message n'est pas transmis entre deux processus, *Net* et $Plc_1$. Deux hypothèses sont envisageables, le message n'est jamais transmis, ou bien il n'est pas transmis au bon destinataire.

#### Localisation

Il faut maintenant analyser les instructions suspectes. Luka cherche alors à localiser dans le programme Fiacre l'ensemble des instructions jouant un rôle dans la communication (envois et réceptions de messages, déclarations des canaux). Il remarque qu'il n'y a pas de connexion entre $Plc_1$ et $Net$, l'instanciation du processus $Net$ étant mal réalisée comme le montre le listing 4.4.

```
par
Gc (ID_Gc, ...) || Plc (ID_PLC_1, ...) || Plc (ID_PLC_2, ...) ||
//  Il n'y a pas de FIFO entre NET et PLC1, uniquement PLC1 vers NET
|Net  (ID_NET, &fifo_Gc_NET, &fifo_PLC_1_NET, &fifo_PLC_2_NET, &fifo_NET_Gc
    , &fifo_PLC_1_NET, &fifo_NET_PLC_2)
end
```

Listing 4.4 – Premier problème

Ici les activités cognitives de Luka ont consisté à isoler les communications entre processus à travers des visualisations de traces, puis localiser les instructions suspectes.

### 4.2.2 Seconde tentative

#### Visualisation

Une fois la correction apportée, Luka évalue sa qualité en effectuant une autre vérification. Le graph d'état atteint désormais 1 871 états et 5 470 transitions, et devient difficilement compréhensible alors qu'OBP signale une nouvelle fois qu'une propriété est violée, comme le montre le résultat extrait d'OBP 4.5.

Il s'agit de $pty\_CLT_1\_PLC_1\_RED\_RES_1\_Scenario_1b$ laissant supposer que deux messages d'acquittements sont transmis au *Client* alors qu'une seule une demande est effectuée.

```
pty_CLT1_PLC1_RED_RES1_Scenario1a: reached success in conf 25.
pty_CLT1_PLC1_RED_RES1_Scenario1b: reached reject in conf 95.
```

Listing 4.5 – Vérification du scénario 1b

De la même manière que précédemment, Luka observe une trace à travers un diagramme de séquence contenant la configuration 95. Le message est transmis entre le $Client_1$ et $Plc_1$, puis un acquittement est renvoyé par $Plc_1$ au $Client_1$. Mais l'acquittement est renvoyé une seconde fois, comme le montre le diagramme 4.5.

#### Stimulation

Ici, le diagramme de séquence ne permet pas d'aller plus loin dans l'analyse, il faut changer de stratégie. Expert en model checking, Luka tente d'analyser le modèle en le

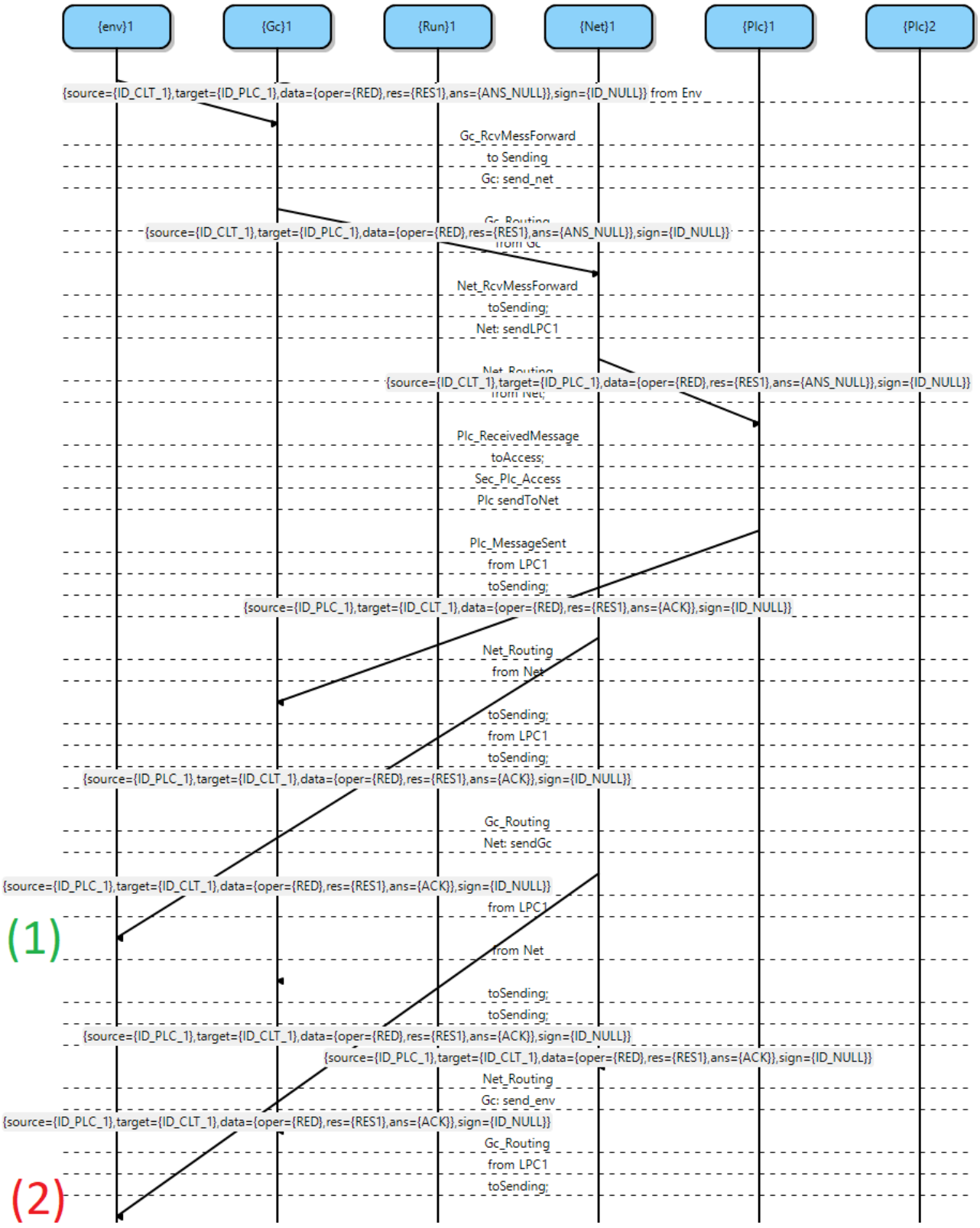


FIGURE 4.5 – Diagramme de séquence de la trace, deux acquittements sont renvoyés

stimulant. Il modifie par exemple la taille des FIFOs, qui passent de 4 à 2. Une nouvelle vérification indique une réduction de l'espace d'état, 677 états et 1 794 transitions, et que la propriété est toujours violée. Les FIFOs ont une certaine tendance à se remplir, elles représentent un point critique à observer. Stimuler le modèle permet d'analyser le fonctionnement du système et ses limites.

### Génération

Luka applique de nouvelles propriétés, que nous appelons *propriétés de contrôle*, lui permettant d'observer des comportements particuliers des FIFOs suspectées. Il les énonce en CDL (listing ci-dessous). Le prédicat *pre_Fifo_push* est vrai si la FIFO $PLC_1\_NET$ contient un message, et le prédicat *pre_Fifo_pop* est vrai si la FIFO n'en contient pas. Deux événements dépendent de ces prédicats, *evt_Fifo_push* est déclenché quand le prédicat *pre_Fifo_push* est vrai, et *evt_Fifo_pop* est déclenché quand le prédicat *pre_Fifo_pop* est vrai.

Un nouvel observateur, $pty\_CLT_1\_PLC_1\_RED\_RES_1\_Scenario_1bFifo$, est en succès s'il observe successivement les événements *evt_Fifo_push* puis *evt_Fifo_pop*. Autrement dit si cet observateur n'est jamais en succès, la FIFO n'est jamais vidée.

```
predicate pre_Fifo_push is {{Run}1:fifo_PLC_1_NET.length = 1} }
predicate pre_Fifo_pop is {{Run}1:fifo_PLC_1_NET.length = 0} }

event evt_Fifo_push is {pre_Fifo_push becomes true}
event evt_Fifo_pop is {pre_Fifo_pop becomes true}

property pty_CLT1_PLC1_RED_RES1_Scenario1bFifo is{
  start   -- / / evt_Fifo_push /  -> wait;
  wait  -- / / evt_Fifo_pop / -> success
}
```

Listing 4.6 – Propriété pty_CLT1_PLC1_RED_RES1_Scenario1bFifo et prédicats

Luka affecte la taille maximale des FIFOs à 1, par conséquent, il ne peut pas y avoir plus d'une fois l'événement *evt_Fifo_push*, ou autrement dit, une FIFO remplie n'a pas d'autre choix que d'être vidée. Le résultat de la nouvelle exploration donne :

```
pty_CLT1_PLC1_RED_RES1_Scenario1a:  reached success the first time in conf
    25.
pty_CLT1_PLC1_RED_RES1_Scenario1b:  reached reject the first time in conf
    89.
Violated asserts: 'pre_Fifo_push the first time in configuration 21.
```

Listing 4.7 – Vérification du scénario 1b

L'observateur de la FIFO atteint l'état *wait* (après observation de *evt_Fifo_push*) mais ne parvient pas à l'état *success* (n'observe pas d'événement *evt_Fifo_pop*). Premier

diagnostic, la FIFO a bien été remplie par un message, mais n'est pas dépilé par la suite. Le prédicat $pre\_Fifo\_push$ à *false* confirme qu'elle contient bien un message à partir de la configuration 21. Luka fait l'hypothèse que la FIFO n'est pas dépilée. Il modifie le modèle pour dépiler la FIFO, et une nouvelle tentative de vérification vérifiant la propriété confirme l'hypothèse, le diagnostic est que la FIFO n'était jamais dépilée.

## 4.3 Conception d'une nouvelle solution

Après avoir compris que la FIFO n'était pas dépilée, Luka a corrigé le modèle et la propriété est désormais satisfaite. Une nouvelle propriété est souhaitée, $P\_NotRes_1AndRes_2$, qui indique que la ressource *Res* ne peut pas être accédée en même temps par les deux *Plcs*, autrement dit il n'est pas possible que $Plc_1$ et $Plc_2$ soient simultanément dans leur état *Access*. Cette propriété est exprimée en LTL par :

$$\boxed{P\_NotRes_1AndRes_2 : \square\neg\{PLC_1@Access \wedge PLC_2@Access\}} \tag{4.2}$$

Après une première vérification, la propriété $P\_NotRes_1Res_2$ est violée. Luka ne fait pas d'autres analyses, les connaissances liées au model checking ne lui permettent pas d'aller plus loin, il doit apporter des modifications à la solution, et changer de point de vue pour passer du correcteur au concepteur.

### 4.3.1 Solution par mécanisme de drapeaux

#### Abstraction

Il imagine alors un mécanisme de drapeaux que chaque processus devra lever pour signaler son désir d'accéder à *Res*. L'algorithme imaginé est le suivant :

```
1  Plc1:
2    while true {
3      while flagPlc2 == up {};
4      flagPlc1 := up;
5      access;
6      flagPlc1 := down;
7    }
8
9  Plc2:
10   while true {
11     while flagPlc1 == up {};
12     flagPlc2 := up;
13     access;
14     flagPlc2 := down;
15   }
```

Les drapeaux sont deux variables booléennes partagées entre les deux $Plcs$ concurrents, nommées $flagPlc_1$ et $flagPlc_2$. Quand ils sont levés (leur valeur est $UP$), ils indiquent que le $Plc$ souhaite accéder à la ressource, à l'inverse quand ils sont baissés (leur valeur est $DOWN$), ils indiquent que le $Plc$ a libéré la ressource. Par exemple, $flagPlc_1 := UP$ signifie que le $Plc_1$ lève son drapeau, et donc qu'il souhaite accéder à la ressource. Un test est effectué avant de lever le drapeau afin de vérifier que le drapeau adverse n'est pas levé, auquel cas il faut attendre qu'il se baisse.

#### Implémentation

Pour intégrer cet algorithme au système, Luka doit implicitement le lier au modèle (Fiacre en l'occurrence). Par exemple l'opération $flagPlc_1 == UP$ est mise en œuvre dans la transition entre les états de l'automate *Received* (symbolisé par deux sous états, *Received_Test* et *Received_Affect*), et *Access*. Après intégration il obtient le comportement décrit par la machine à états de la figure 4.6.

Les transitions entre états respectent la notation $Si \xrightarrow{\{Event\}[Condition]} Sj$ où $Si$ et $Sj$ sont des *états*, la flèche représente une *transition* déclenchée sur l'évaluation d'une *condition* booléenne et/ou la réception d'événements (*events*, précédés de ?). L'*action* d'un état est une instruction (affectation de variable ou un événement émis (précédé de !)). Dans la sémantique du langage Fiacre, l'ensemble des actions exécutables d'un état est considéré atomique.

A l'initialisation du système, les deux drapeaux sont baissés, et les $Plcs$ sont dans leur état $Idle$. Ils attendent un message provenant du processus $Net$ via la fifo $fromNetPlc$. Si un $Plc$ reçoit un message du $Net$, il progresse dans l'état *Received* (comprenant l'état *Received_Test* et l'état $Received\_Affect$) pour le traiter. Si le message correspond à une demande d'accès à la ressource, le $Plc$ doit s'assurer que son $Plc$ concurrent ne demande pas lui aussi la ressource (par exemple $Plc_2$ demande la ressource quand $flagPlc_2$ est à $UP$), auquel cas il devra attendre dans l'état $Received\_Test$. Si le processus concurrent ne demande pas la ressource, alors le processus lève son drapeau (par exemple $flagPlc_1 := UP$ pour $Plc_1$) et accède à la ressource (état *Access*). Lorsqu'il a terminé il sort de la section critique (état *Sending*) en baissant son drapeau (par exemple $flagPlc_1 := DOWN$ pour $Plc_1$) pour informer de la disponibilité de la ressource. Enfin il transmet une information acquittant le processus $Net$ via la fifo $fifoPlcNet$.

### 4.3.2 Vérification et diagnostic

Après vérification, il s'avère que la propriété $P\_NotRes_1AndRes_2$ est violée, signalant l'existence d'un contre exemple où $Plc_1$ et $Plc_2$ utilisent en même temps la ressource, ce qui déclenche une phase de diagnostic.

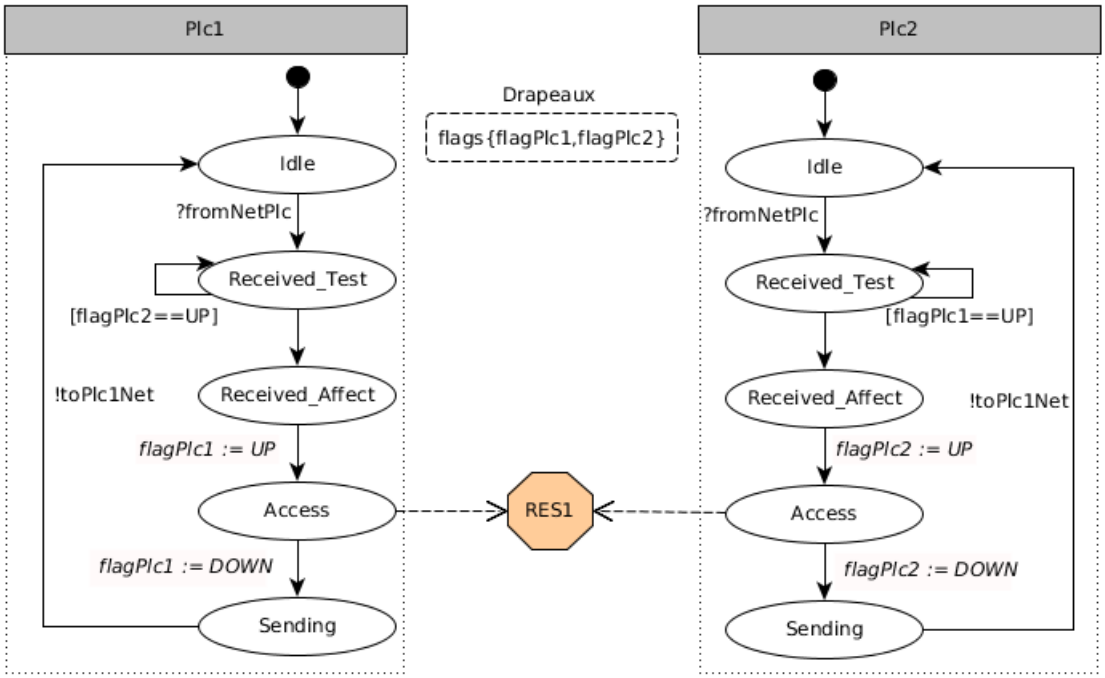


FIGURE 4.6 – Comportement des Plcs avec un mécanisme de drapeaux.

### Réduction

Un contre-exemple brut (a), filtré (b) et vu de l'algorithme (c) est présenté dans la figure 5.1. Après une étude infructueuse du contre-exemple brut (a) de façon analogue au scénario précèdent (visualisations, expression de propriétés de contrôle), il s'avère que l'algorithme est bien implémenté mais mal conçu. Le contre-exemple brut (a) contient de nombreuses configurations dont quelques unes uniquement sont liées à l'algorithme mis en cause, or si le problème vient de l'algorithme, il faut permettre de raisonner dessus.

Luka réduit alors le contre-exemple aux seules configurations produites par l'action de l'algorithme (b). Ces configurations regroupent des informations techniques (canaux, processus, états...), et il est difficile de les percevoir sous l'angle de l'algorithme (*while, flag :=up*). C'est le problème du fossé sémantique. Pour parvenir à le réduire, Luka doit extraire du contre-exemple les informations liées à cet algorithme en explicitant les liens implicites qui ont permis de passer de l'algorithme au modèle.

### Interprétation

Pour résoudre le fossé sémantique entre l'algorithme et le contre-exemple, Luka doit disposer d'une nouvelle vue (c) qui lui permettrait de raisonner sur l'algorithme. Cette vue n'existe pas dans OBP, mais la possibilité d'annoter les transitions peut résoudre en partie

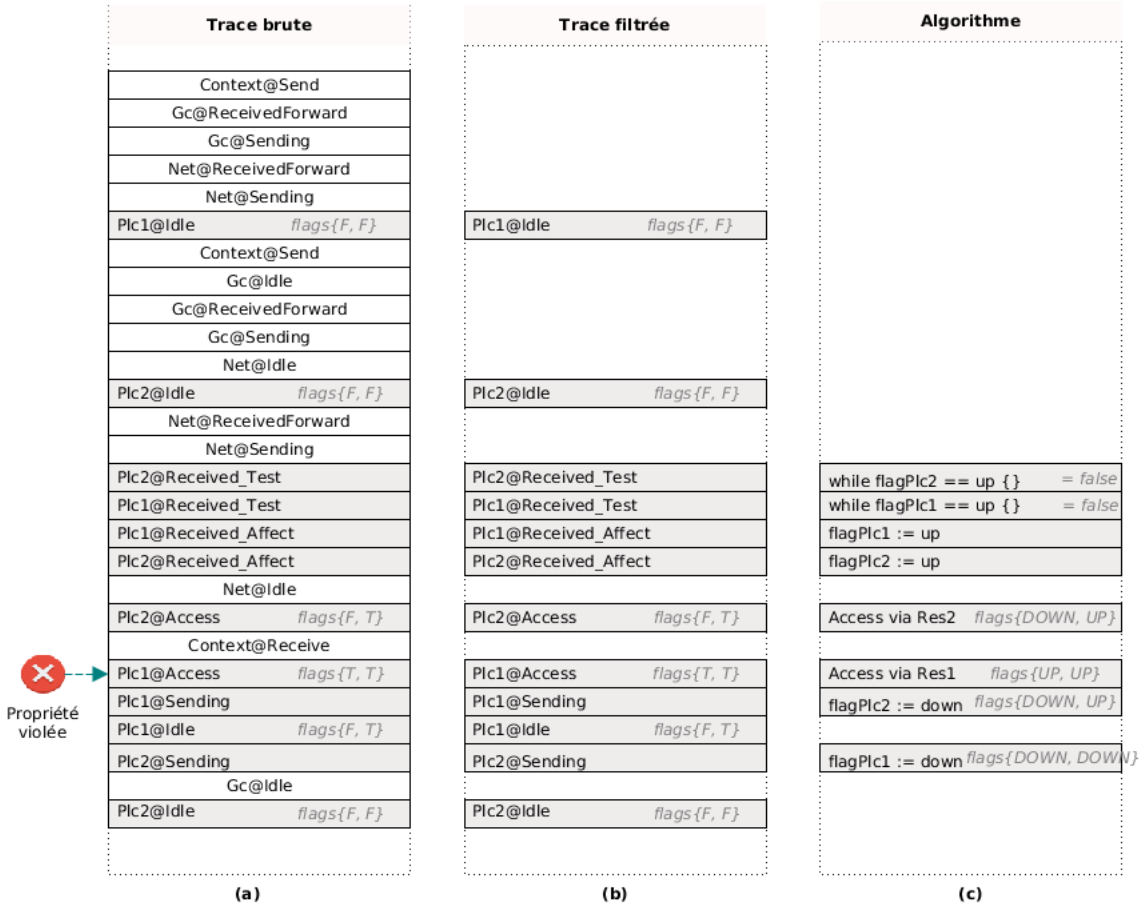


FIGURE 4.7 – Un contre-exemple brut (a), filtré (b) et vue de l'algorithme (c)

le problème. En complément, nous proposons d'utiliser un mécanisme de fédération de modèles, décrit à la section 7.2.2, pour établir des liens sémantiques entre traces, modèles et algorithmes.

#### Évaluation et correction

L'analyse de la trace décrit la situation suivante : les deux drapeaux sont initialement baissés. Le premier processus vérifie que le drapeau adverse est baissé ; donc lève son drapeau et entre dans l'état *Access*. Ceci est entrelacé avec un comportement similaire du deuxième processus. Ce qui explique pourquoi la propriété est violée, Luka doit revoir la solution et s'intéresser au domaine de l'exclusion mutuelle.

## 4.4 Conclusion

### 4.4.1 Cadre conceptuel

Luka doit donc mettre en œuvre un ensemble d'activités cognitives, chacune apparentée à un outil ou technique. Ces activités peuvent être placées dans la taxonomie de Bloom comme le montre le tableau 4.1. *Se souvenir* consiste à retrouver l'emplacement dans un environnement technique d'une information identifiée. Par exemple Luka est capable de retrouver le modèle, les propriétés ou les contre-exemples. *Comprendre* est la superposition de représentations. Pour Luka, cela consiste par exemple à visualiser les contre-exemples à travers des diagrammes de séquence, ou bien faire le lien entre les éléments du modèle et le contre-exemple, ou les éléments du modèle et l'algorithme. *Appliquer* consiste notamment à générer un ensemble de propriétés de contrôle dans le but de comprendre l'état des FIFOs. Luka réalise différentes *analyses*. Il isole les communications entre les processus, il localise dans le modèles les éléments suspicieux, il stimule le modèle en jouant sur différentes variables ou il réduit la trace selon un point de vue particulier. Les *évaluations* sont d'une part, la vérification par model checking qui donne le statu des propriétés par rapport au modèle, et d'autre part les diagnostics (finaux ou intermédiaires) énonçant la cause de la violation de propriété, qu'elle soit issue du modèle ou du design. Pour finir Luka réalise une activité de *création* lorsqu'il conçoit le mécanisme de drapeau.

### 4.4.2 Changement de point de vue

A travers le prisme des connaissances liées au model checking, le diagnostic est difficile à établir pour deux raisons principales, le grand nombre d'objets exposés par les traces analysées (nombreuses configurations, transitions, taille des graphes d'exploration), et

| | Connaissances liées au model checking (Concepts, Modèle et Propriétés formelles...) |
|---|---|
| Se souvenir | OBP, Fiacre, Ltl, CDL |
| Comprendre | Visualiser (MSC), Corréler (Semantic gap) |
| Appliquer | Générer des propriétés de contrôles et les vérifier |
| Analyser | Isoler les communications, localiser le code, stimuler, réduire la trace |
| Évaluer | Vérifications (model checking), Diagnostics (erreurs de modèle ou design) |
| Créer | Mécanisme de drapeau |

TABLE 4.1 – Les activités cognitives de Luka classées selon Bloom

leur niveau de détail, parfois décorrélés des concepts "métier" (artefacts obtenus dans le cadre des transformations de la chaîne outillée sans rapport direct et évident avec la couche conceptuelle de haut niveau). Il est difficile pour l'utilisateur de pouvoir manipuler l'intégralité de ces données pour analyser son problème.

Rester au niveau des traces ne permet pas toujours de résoudre le problème, et il faut souvent raisonner sur l'algorithme. Ce changement de point de vue sur le contre-exemple contraint le diagnosticien à résoudre un fossé sémantique entre les spécifications et le contre-exemple exprimé par des connaissances liées au model checking. Il doit ainsi expliciter les corrélations entre les spécifications et le modèle qui les implémentent. Ces corrélations peuvent être réalisées techniquement par la mise en œuvre de prédicats, de propriétés, ou parfois d'annotations relatives aux spécifications mais décrites sur la base de connaissances liées au model checking.

L'ingénierie de domaine facilite la transition de la fabrication artisanale d'une solution unique en son genre vers la production automatisée d'une famille de solutions [CE00]. Placer l'application à vérifier au sein d'un domaine éprouvé permet d'effectuer un nouveau changement de point de vue. L'ingénierie de domaine nous aide à concevoir des composants réutilisables et une architecture commune pour une famille de solutions. Pour bénéficier de l'expérience d'un domaine, il faut avoir accès à des problèmes résolus et des problèmes en échec. Il faut pouvoir modifier une solution passée pour l'adapter à une nouvelle situation-problème. Ce sont ces enjeux que nous allons traiter dans le chapitre suivant, où Ciprian mène des activités de conception et de vérification autour du SAE, en bénéficiant de connaissances de domaine.

Chapitre 5

# Formaliser, partager et réutiliser les connaissances

## 5.1 Introduction

On distingue généralement la donnée de l'information, l'information étant une donnée à laquelle on associe du sens. Jusqu'à présent, les activités cognitives de diagnostic manipulaient des informations de nature uniquement technique. Cependant celles-ci restent parfois insuffisantes pour établir un diagnostic, et doivent être complétées par des informations provenant du domaine. Ces informations sont souvent hétérogènes et il n'y a pas de consensus ni sur leur formalisation, ni sur un système permettant leur organisation et leurs interactions. Dans cette section, nous présentons comment les activités de diagnostic bénéficient des informations du domaine, et nous présentons une formalisation permettant leur réutilisation.

## 5.2 Prise en compte du domaine

Prenons le second scénario où Ciprian joue le rôle de concepteur du système. Ciprian reprend le modèle de Luka où deux processus concurrents $Plc_1$ et $Plc_2$ partagent une même ressource $Res_1$. Cette ressource ne peut pas être accédée au même instant par $Plc_1$ et $Plc_2$ (aussi bien en lecture qu'en écriture).

### 5.2.1 Description du domaine

En explorant la littérature, il y découvre le problème de l'exclusion mutuelle formulée par Lamport dans [Lam86]. Le problème est décrit comme suit : Considérons un processus $Pr_i$, dont la structure est composée d'une opération non-critique ($NCS$) et d'une opération critique ($CS$). Ces opérations sont considérées élémentaires, aussi, ni la structure interne de chaque opération ni même leur durée n'est considérée. Ces opérations sont exécutés alternativement, comme dans l'exemple suivant : $NCS \rightarrow CS \rightarrow NCS \rightarrow CS$....

Supposons deux processus distincts, $Pr_i$ et $Pr_j$ respectant cette structure, l'exclusion mutuelle (entre ces deux processus) revient à dire que les opérations $Pr_iCS \wedge Pr_jCS$ ne

doivent jamais être exécutées simultanément. Cette propriété est nommée ici $PMutexPr_iPr_j$, et est équivalente à l'expression LTL :

$$\boxed{PMutexPriPrj : \Box\neg(Pr_iCS \wedge Pr_jCS)} \tag{5.1}$$

Pour que ce principe soit respecté, il faut ajouter un mécanisme qui empêche cette propriété d'être violée. Celui-ci consiste en un ensemble d'opérations de synchronisation inter-processus. Ces opérations ne doivent pas être concurrentes des sections critiques et non critiques de ce même processus et doivent être atomiques. Ainsi, l'opération *Trying* réalise une demande d'accès à la ressource, et doit être exécutée entre la section non critique et la section critique. L'opération *Exit* réalise la libération de la ressource, et doit être exécutée entre la section critique et la section non critique suivante. La structure d'un processus se décrit alors comme :

```
initial declaration;
  repeat forever
    noncritical section ;
    trying ;
    critical section ;
    exit ;
  end repeat
```

La séquence d'opérations est la suivante : $NCS \rightarrow Trying \rightarrow CS \rightarrow Exit \rightarrow NCS$....

### 5.2.2 Application du domaine

Ciprian connaît désormais l'énoncé du problème, l'exclusion mutuelle. Son problème se compose d'une structure faite de concepts abstraits $(NCS, Trying, CS, Exit)$, et de propriétés abstraites qui devront être redéfinies au regard de la solution réalisée.

En appliquant le domaine de l'exclusion mutuelle, Ciprian rajoute de nouvelles propriétés à son système, par exemple les deux *Plcs* ne peuvent pas accéder à la ressource critique en même temps. Cette propriété est une *propriété de problème de domaine (PPD)* .

Les *PPDs* pour l'exclusion mutuelle sont abstraites par rapport à une solution choisie. Pour les appliquer, il faut identifier parmi les éléments de la solution, ceux qui correspondent aux éléments des *PPDs*. Ciprian fait la correspondance suivante : - l'opération *Trying* est réalisée dans la solution à deux instants dans l'état *Received*, lors de l'évaluation du drapeau du processus concurrent (transition entre état *Received_Test* et *Received_Affect* avec l'opération $[flag == UP]$), et la levée du drapeau (transition entre l'état *Received_Affect* et *Access* avec l'opération $[flag := UP]$) ; - l'opération d'accès à la section critique est réalisée entièrement dans l'état *Access* ; - enfin l'opération

*exit* est réalisée entièrement dans l'état *Sending* (succédant à la transition avec l'opération $flag := DOWN$). Ciprian peut maintenant énoncer la propriété d'exclusion mutuelle au regard de sa solution :

$$\boxed{PMutexPlc_1Plc_2 : (\Box\neg(Plc_1@Access \wedge Plc_2@Access))} \tag{5.2}$$

### 5.2.3 Vérification et diagnostic

Comme précédemment, la propriété $PMutexPlc_1Plc_2$ est violée, signalant l'existence d'un contre exemple où $Plc_1$ et $Plc_2$ utilisent en même temps la ressource, commence donc une phase de diagnostic.

Un contre-exemple brut (a), filtré (b), et vu du domaine (c) est présenté dans la figure 5.1. La figure (a) montre qu'il existe de nombreuses configurations dans le contre-exemple, dont quelques unes uniquement, sont liées au problème de l'exclusion mutuelle comme le montre le contre-exemple filtré (b). Le contre-exemple est produit sur la base d'informations techniques (canaux, processus, états...),et il est difficile de percevoir les informations venant du domaine (section critique, exit...). Or, si le problème vient du domaine, et donc permettre de raisonner dessus, il faut réussir à extraire du contre-exemple les informations liées au domaine. Il va falloir par conséquent corréler le domaine et le contre-exemple.

Ciprian isole au sein de la trace les configurations concernées par les opérations de l'exclusion mutuelle (c). Techniquement, les opérations d'accès à la ressource peuvent être isolées en définissant des prédicats détectant l'état *Access* (prédicats $plc_1CS$, $plc_2CS$). En effet, cette opération est uniquement réalisée dans cet état. De façon analogue, l'opération *exit* est isolable en définissant des prédicats détectant l'état *Sending* (prédicats $plc_1Exit$, et $plc_2Exit$). Pour retrouver l'opération *trying* diffuse entre les états $Received\_Test$ et $Received\_Affect$, Ciprian choisit d'annoter les instructions concernées par cette opération (le test des drapeaux et leur levées), puis de définir des prédicats les observants (nommés $plc_1Try$ et $plc_2Try$).

Il peut maintenant filtrer le contre-exemple avec ces prédicats. La trace produite est visible sur la figure (c) de la figure 5.1. Ciprian s'aperçoit que la propriété est violée à cause de l'opération *trying*. Le problème se situe donc dans cette opération constituée des états $Received\_Test$ et $Received\_Affect$. Or la présence de ces deux états indique que l'opération $Trying$ est non-atomique ce qui contredit l'énoncé du problème l'interdisant (section 5.2.1). Il y a donc une possibilité d'entrelacement entre l'opération $Trying$ de $Plc_1$ et l'opération $Trying$ de $Plc_2$.

L'algorithme ne marche pas à cause de l'entrelacement produit par la non atomicité de l'opération $Trying$. Grâce au domaine, Ciprian peut préciser le diagnostic sur l'opération $Trying$. Dans ce scénario, les connaissances de domaine ramènent des spécifications liées à

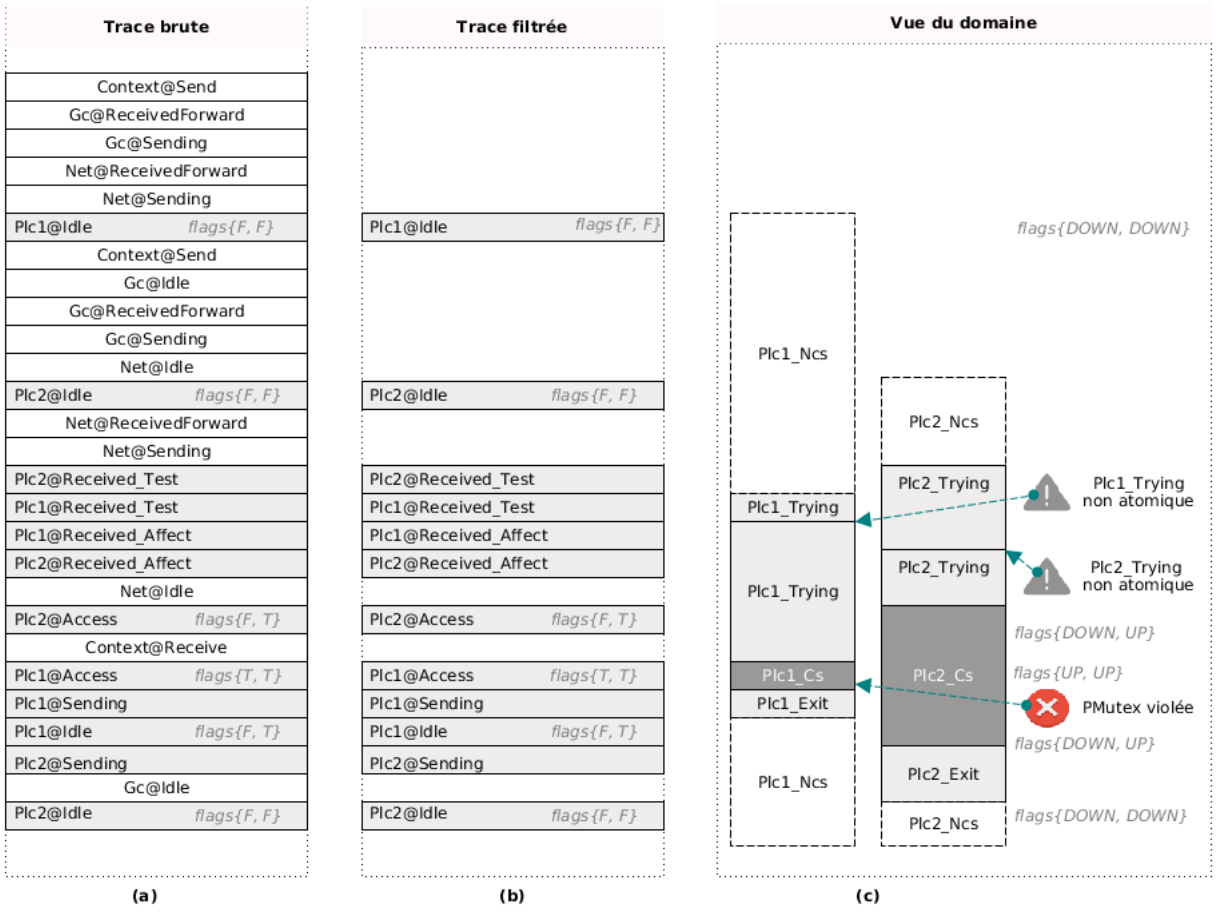


FIGURE 5.1 – Un contre-exemple brut (a), filtré (b), et vu du domaine (c)

un problème. A partir de là, une solution est développée, devant respecter les spécifications du problème (propriétés, structure) mais aussi les propriétés de l'application. Si celles-ci sont violées, il faut améliorer la solution en passant par la phase de diagnostic. Lors de ce diagnostic, les connaissances liées au model checking ne permettent pas toujours de résoudre le problème, notamment quand celui-ci est inhérent au domaine. Dans ce cas, le diagnosticien doit raisonner sur les connaissances de domaine. Ce changement de point de vue sur le contre-exemple contraint le diagnosticien à résoudre un fossé sémantique de même nature que celui évoqué lors du chapitre précèdent.

### 5.2.4 Différentes natures de propriétés

Il faut concevoir une nouvelle solution qui respectera un certain nombre de propriétés. On peut faire une distinction entre les propriétés exigées par l'application et les propriétés originaires du domaine.

Avant même d'implémenter l'algorithme d'exclusion mutuelle, le système doit répondre aux exigences décrites dans la section 4.1.3.

Par exemple $pty\_CLT_1\_PLC_1\_RED\_RES_1\_Scenario_1a$ qui indique que $Plc_1$ doit acquitter toute demande extérieure. Cette propriété est une *propriété du problème d'application*, en abrégé $PPA$. De plus, Ciprian va devoir appliquer les propriétés du domaine qui permettront d'évaluer plus précisément sa nouvelle solution, c'est à dire les $PPDs$ . Conformément au problème énoncé par Lamport, en plus de la propriété $PMutexPriPrj$ Ciprian applique la propriété $PDeadlockFreedom$, qui sera directement gérée par le model checker, et $PNoLockout_i$ qui sera énoncée en LTL par :

$$\boxed{PNolockoutPri : \Box(PriTrying \rightarrow \Diamond PriCs)} \tag{5.3}$$

$PNoLockout_i$ est traduite pour la solution en :

$$\boxed{\begin{array}{l} PNolockoutPlc_1 : \Box(Plc_1@Received \rightarrow \Diamond Plc_1@Access) \\ PNolockoutPlc_2 : \Box(Plc_2@Received \rightarrow \Diamond Plc_2@Access) \end{array}} \tag{5.4}$$

#### Propriétés de solutions de l'application

Si le langage de modélisation utilisé par Ciprian ne possède pas la capacité d'exprimer des opérations atomiques, il doit trouver une solution alternative. Par exemple, l'implémentation de la levée du drapeau peut être faite avant le test du drapeau. Ciprian corrige alors l'ancien mécanisme de drapeaux en inversant $Received\_Test$ avec $Received\_Affect$ (levée du drapeau puis test du drapeau du processus concurrent). De plus, la nouvelle solution va s'assurer que la levée du drapeau est bien réalisée entièrement avant le test à travers une nouvelle propriété propre à la solution. Nous appelons les propriétés amenées

par la solution des *propriétés de solution de l'application*, en abrégé $PSA$. L'illustration 5.2 montre le comportement mis à jour.

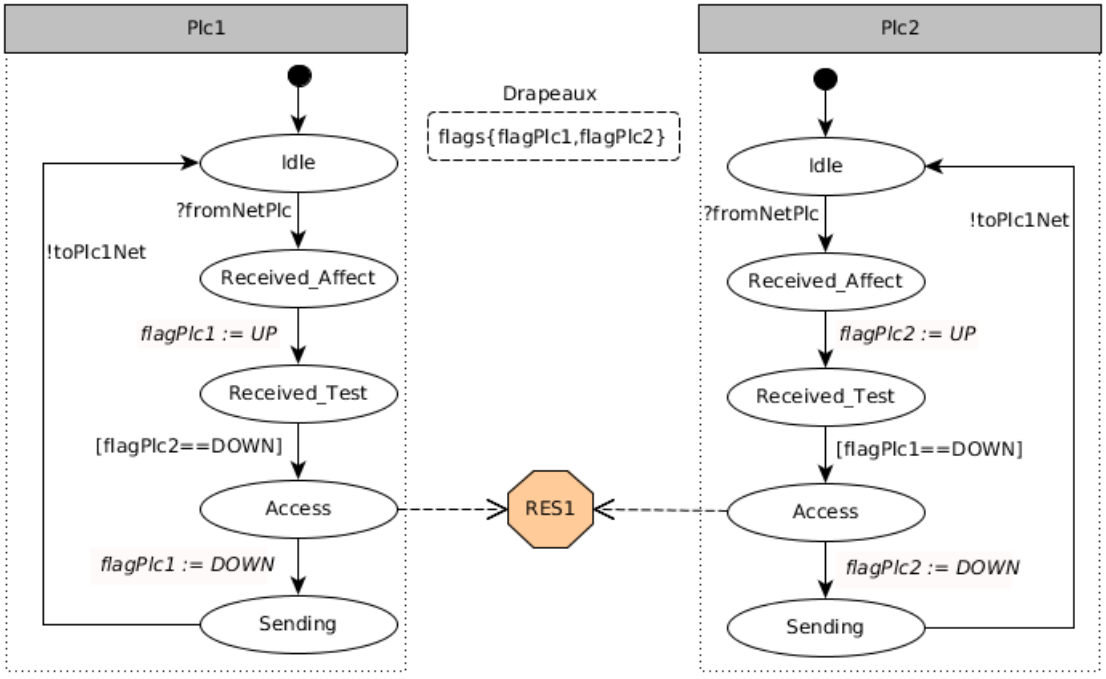


FIGURE 5.2 – Comportement des Plcs, deuxième version du mécanisme de drapeaux

Les $PSA$ créées sont $Plc_1FlagUpBeforeAccess$ et $Plc_2FlagUpBeforeAccess$, exprimées en LTL par :

$$\boxed{\begin{array}{l} Plc_1FlagUpBeforeAccess : \neg(Plc_1@Access\ W\ (Plc_1.flag)) \\ Plc_2FlagUpBeforeAccess : \neg(Plc_2@Access\ W\ (Plc_2.flag)) \end{array}} \quad (5.5)$$

### Vérification et diagnostic

La nouvelle solution est évaluée par un model checker. Cette fois-ci, les propriétés $PMutexPlc_1Plc_2$, $Plc_1FlagUpBeforeAccess$ et $Plc_2FlagUpBeforeAccess$ sont vérifiées, mais le model checker détecte tout de même la présence d'un *deadlock*. Le contre-exemple (a) est illustré sur la figure 5.3.

Comme précédemment, Ciprian effectue dans un premier temps des analyses sur des connaissances liées au model checking (visualisation, évaluation de propriétés de contrôle...), mais une nouvelle fois le problème vient du domaine. Comme précédemment, il isole le domaine dans le contre-exemple à l'aide de prédicats et d'annotations, et obtient

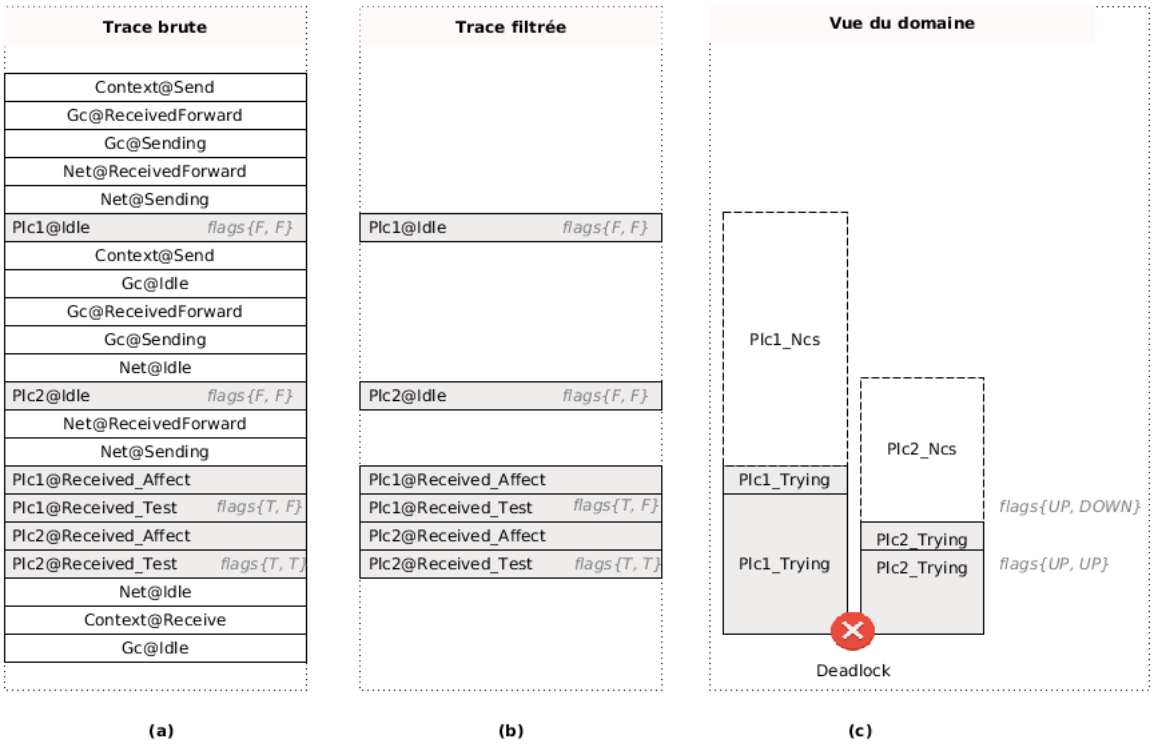


FIGURE 5.3 – Un contre-exemple brut (a), filtré (b) et vu du domaine (c)

la trace (c) de la figure 5.3. Cette trace vue du domaine l'informe que les deux processus restent continuellement dans l'opération *trying*, et n'en sortent jamais.

Supposons que les deux *Plcs* soient dans l'état *Received_Test*. A cet instant, tout deux désirent accéder à la ressource et ont déjà levé leur drapeau. Or pour accéder à la ressource ils doivent s'assurer que le drapeau de l'autre est baissé, ce qui n'est pas le cas, aucun *Plc* ne peut alors accéder à la ressource. Puis comme les drapeaux ne peuvent plus être baissés car il faudrait qu'un *Plc* atteigne l'opération *exit*, aucun processus ne peut avancer, il y a *deadlock*.

Au cours de ce cycle de conception-vérification, différentes propriétés sont énoncées, certaines proviennent du problème que l'on veut résoudre, soit du domaine (*propriétés de problème du domaine*), soit de l'application ( *propriétés de problème de l'application*), et certaines sont ramenées par la solution choisie (*propriétés de solution de l'application*).

### 5.2.5 Conservation de propriétés

#### Propriétés

Ciprian estime que le mécanisme de drapeau pourrait fonctionner en le complétant. En repartant de cette dernière solution, l'ensemble des propriétés précédentes ($PPA$, $PPD$, $PSA$) est conservé. Pour corriger le problème de *deadlock*, une solution serait de forcer l'un des deux drapeaux à se baisser en cas de conflit, c'est à dire quand les deux drapeaux sont levés simultanément. La correction choisie est que $Plc_1$ baisse son drapeau lorsque le drapeau du $Plc_2$ est levé, ce qui est formulé avec la $PSA$ :

$$\boxed{PClearFlagPlc_1 : \Box(Plc_1FlagUp \wedge Plc_2FlagUp) \rightarrow \Diamond Plc_1FlagDown} \quad (5.6)$$

Le comportement du modèle est décrit sur la figure 5.4, l'action $flagPlc_1$ *:=DOWN* est désormais exécutée dans l'état $Plc_1@Received$ si l'expression $flagPlc_2 == UP$ est évaluée à vrai.

#### Vérification et diagnostic

Ce complément de solution est évalué au regard des différentes propriétés :

- Des ***PPDs* et *PPAs*** ($PMutexPlc_1Plc_2$, $PDeadlockFreedom$, $PNoLockoutPlc_1$, $PNoLockoutPlc_2$)
- Des ***PSAs*** précédentes ($Plc_1FlagUpBeforeAccess$, $Plc_2FlagUpBeforeAccess$)
- Des ***PSAs*** de la nouvelle solution ($PClearFlagPlc_1$).

Après vérification, le model checker détecte la présence d'une propriété ($PNoLockoutPlc_1$),

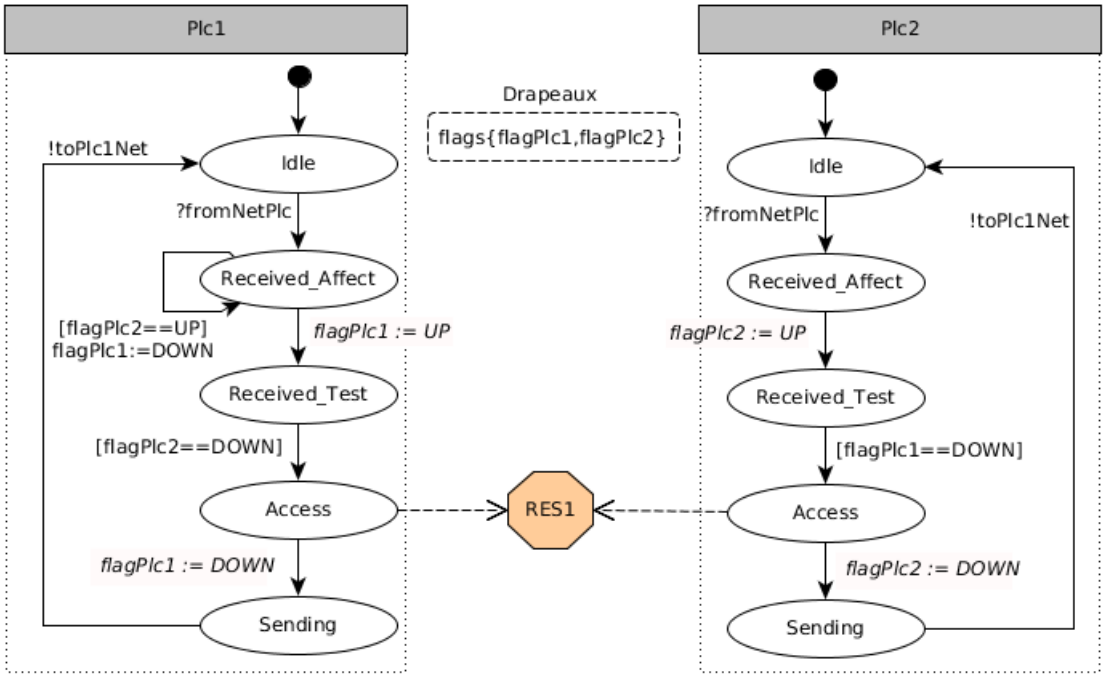


FIGURE 5.4 – Le Plc1 baisse son drapeau en cas de conflit

indiquant que la solution n'est pas équitable car $Plc_1$ peut ne jamais accéder à la section critique.

Dans ce cycle de conception-vérification, la solution précédente est conservée pour être complétée. Par conséquent, l'ensemble de ses propriétés sont conservées puis complétées par de nouvelles *propriétés de solution de l'application*.

### 5.2.6 Renouvellement de la solution

#### Propriétés

La solution précédente ne convient pas puisqu'elle ne satisfait toujours pas la propriété d'équité. À ce stade, Ciprian décide d'abandonner le mécanisme de drapeau, et opte pour un mécanisme de tour. La ressource sera désormais utilisée équitablement, à tour de rôles. Le modèle ne possédant plus de drapeau, il devra compter sur une variable ($turn$), ainsi que sur deux nouvelles *PSAs*, $PChange$ : après qu'un processus ait fini son exécution la variable $turn$ doit être changée ; $PTurn$ : un processus ne peut pas être exécuté si ce n'est pas son tour. Ce changement de solution rend obsolète les *PSAs* du drapeau, $Plc_1FlagUpBeforeAccess$, $Plc_2FlagUpBeforeAccess$ et $PClearFlagPlc_1$.

### Modèle

La solution se compose des machines à états illustrées dans la figure 5.5. Le processus qui souhaite accéder à la ressource doit d'abord vérifier si c'est bien son tour à travers la variable *turn* évaluée dans l'opération *trying* (état *Received*). Une fois la ressource utilisée, le processus qui en a bénéficié cède l'accès au processus qui lui est concurrent, toujours grâce à la variable *turn*, et ce pendant l'opération *exit* (état *Sending*).

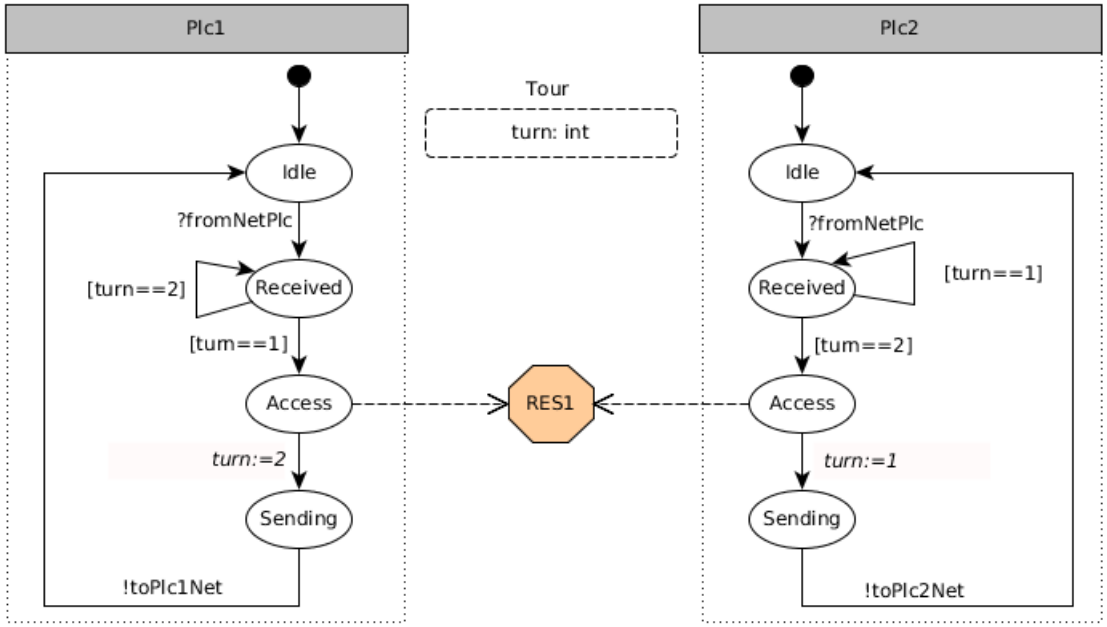


FIGURE 5.5 – Comportement des Plcs, mécanisme de tour

### Vérification et diagnostic

Lors d'une nouvelle tentative de vérification, il s'avère que $PNoLockoutPlc_1$ est violée. $Plc_1$ peut continuellement rester dans l'opération *trying* sans jamais accéder à la ressource si $Plc_2$ a le *turn* mais décide de ne jamais y accéder.

Dans ce cycle de conception-vérification, la solution précédente n'est pas conservée. L'impact sur la vérification de la nouvelle solution est que les *PSAs* précédentes ne sont pas toutes conservées, mais les *PPDs* et *PPAs* le sont.

## 5.2.7 Conclusion

Ciprian sait spécifier son problème, l'exclusion mutuelle. Il en connaît sa structure abstraite $(NCS, Trying, CS, Exit)$ et ses propriétés abstraites $(PMutexPr_iPr_j,\ PNoLockoutPr_i)$,

mais n'a pas connaissance de solutions existantes. Il conçoit alors des solutions qu'il doit évaluer au regard des propriétés énoncées du domaine et appliquées à sa solution. On peut distinguer différents éléments, l'énoncé du problème (concepts du domaine, et de propriétés du problème de domaine), des propriétés du problème de l'application et des propriétés inhérentes à la solution de l'application.

Si la solution mise en œuvre ne convient pas, il faut itérer entre la conception de la solution et l'évaluation de cette solution jusqu'à obtenir un modèle satisfaisant, mais sans être sûr d'y parvenir. A chaque solution conservée, de nouvelles propriétés émergent, et font alors partie prenante du problème et des futures vérifications. A contrario, si une solution n'est pas conservée on ne conserve pas ses propriétés. Un problème se pose alors, l'organisation dans le temps de ces différentes propriétés.

La figure 5.6 illustre les différents cycles de conceptions-vérification précédents. On distingue deux espaces, l'espace du problème et l'espace de la solution. D'un côté l'espace du problème regroupe les problèmes que l'on cherche à résoudre. Le problème de l'exclusion mutuelle est par exemple décrit par un ensemble de propriétés du domaine telles que $PMutex$ ou $PNoLockout$. D'un autre côté l'espace de la solution regroupe les solutions à ces problèmes. Par exemple, une solution aux problèmes d'exclusion mutuelle est réalisée à partir d'un mécanisme de drapeaux. Pour vérifier que cette solution est valable, les propriétés du problème doivent être appliquées à cette solution. Ainsi $P\_Mutex$ est appliquées sous la forme $PS\_MutexPlcs$. De plus, de nouvelles propriétés émergent de la solution comme $P\_Plc_1FlagUp$. Si la solution est bonne, ou si l'on estime qu'elle est incomplète (c'est à dire partiellement bonne), celle-ci est gardée, ou complétée, et les propriétés sont conservées. Si celle-ci est invalide, la solution est redéfinie. Dans ce cas de nouvelles propriétés de solution émergent par exemple $P\_Turn$, quand aux propriétés de l'ancienne solution, elle ne sont pas conservées à l'image de $P\_Plc_1FlagUp$.

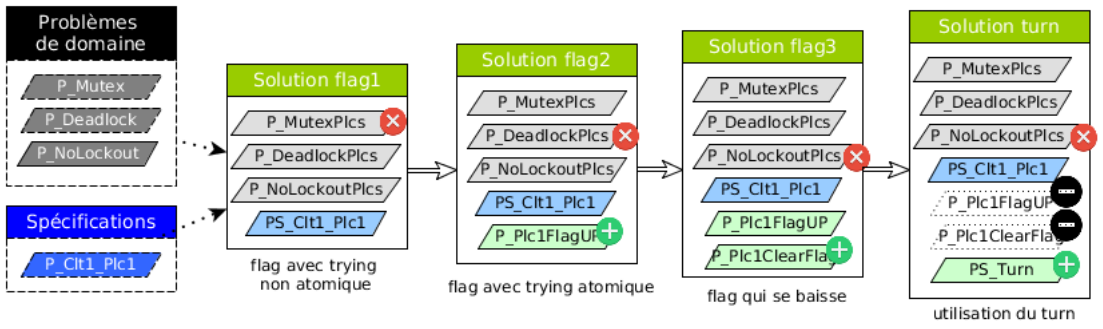


FIGURE 5.6 – Cycles de conception

## 5.3 Vers l'organisation d'un domaine

Au cours du scénario précèdent, aucune solution satisfaisante n'a été trouvée. Malgré plusieurs cycles de conception-vérification, Ciprian ne peut trouver de réponse au problème probablement car il n'a pas de connaissances de solutions existantes. Cela pourrait être évité par la réutilisation de solutions du domaine. Nous pouvons constater que le domaine est utilisé comme un guide de conception et de vérification, en fournissant une description du problème, une structure et des propriétés. Plusieurs questions se posent alors, comment décrire ce domaine, comment l'utiliser, et enfin comment capitaliser des solutions.

### 5.3.1 Contexte

Ces cycles de conception-vérification s'appuient sur un ensemble de problèmes et solutions du domaine, qu'il faut capturer, partager et réutiliser.

D'un côté, nous devons exprimer les caractéristiques du domaine. Une approche connue est celle des *problem frames* (voir section 3.3.1) qui s'appuient sur la formalisation de problèmes de développement de logiciels. Ils structurent l'analyse du monde dans lequel se trouve le problème, appelé le domaine du problème, et décrivent les effets qu'un système, situé dans celui-ci, produit. Un *problem frame* caractérise une classe de problèmes élémentaires simplifiés qui se présentent généralement comme les sous-problèmes d'un problème plus vaste. Ils permettent ainsi d'analyser des problèmes réalistes en les décomposant en sous-problèmes connus. Cette analyse fournit un contexte dans lequel l'expérience précédemment capturée peut être efficacement exploitée.

D'un autre côté, nous devons exprimer les solutions du domaine. Les architectures logicielles [Bus96], ainsi que les frameworks de développement et patrons de conceptions ont les mêmes objectifs de construction, de partage et de réutilisation des connaissances. Ceux-ci sont généralement attribués à l'espace de solution. Un patron est une solution éprouvée et optimale à un problème récurrent dans un contexte particulier. Les patrons sont exprimés de façon générique, permettant ainsi leur implémentation dans des contextes variés avec des technologies différentes. Mais cette généricité a des limites car elle nécessite un effort conséquent d'implémentation. D'autres approches réutilisent des solutions partagées sous la forme de briques réutilisables. Les composants logiciels (plus généralement les COTS) constituent une partie physique et remplaçable d'un système ; les COTS packagent une implémentation ainsi que la réalisation d'un ensemble d'interfaces [1]. Un composant regroupe un certain nombre de fonctionnalités (dont l'implémentation peut être masquée), utilisable à travers les interfaces fournies par le composant. Cette solution est réutilisable dans des contextes différents dans lesquels le composant a été développé, et possède de ce fait un comportement général, mais pas toujours optimal.

1. OMG, "Unified Modeling Language Specification" version 1.3, June 1999

Pour capturer, partager et réutiliser efficacement ces problèmes et solutions, il faut s'appuyer sur une méthode. Le *Case Based Reasoning* ou *CBR* (voir section 3.6.2) est une méthode inspirée du fonctionnement humain pour la réutilisation de cas passés. Lorsqu'un nouveau problème survient, celui-ci est comparé aux cas précédents contenus dans la base, où le plus pertinent est sélectionné. La solution associée au cas sélectionné est ensuite adaptée au contexte du nouveau problème pour permettre de le résoudre. Lorsque la nouvelle solution produite est jugée satisfaisante, c'est à dire qu'elle résout le problème, celle-ci est capturée pour un usage ultérieur.

### 5.3.2 Propositions

Pour organiser le domaine il faut mettre en place différents mécanismes, d'un côté des structures pour capturer les problèmes et les différents types de solutions, et de l'autre une méthode pour orchestrer leur capture, leur partage et leur réutilisation lors des cycles de conception-vérification.

Notre proposition est de capturer le domaine à travers des cas appelés *problem cases*, accompagnés de leurs solutions, les *sample cases*, *pattern cases* ou *component cases*. Ces trois types de solutions correspondent à trois modes de réutilisations. Un *sample case* est une solution fonctionnelle ad-hoc qui a valeur d'exemple ; un *pattern case* est une solution ré-applicable structurée sous forme d'un patron ; un *component case* est une solution intégrable avec le minimum d'adaptations. Ces cas capturent des informations pouvant être réutilisées de façon optimale dans le cadre de la vérification par model checking, en particulier car elles amènent des propriétés énoncées formellement à l'aide de LTL.

Lors de la vérification de la solution de l'application, l'existence de ces propriétés complète le diagnostic en permettant d'évaluer d'une part la pertinence des éléments du domaine sélectionnés, mais aussi leur bonne application. Les *PPDs* affinent la connaissance globale du système à vérifier, et précisent ainsi le diagnostic. Par exemple, si le système comprend un sous système d'exclusion mutuelle, il est possible d'identifier ce sous système et de l'évaluer. Les solutions du domaine amènent également des propriétés, les *propriétés de solutions de domaine* (PSD) qui permettent de focaliser le diagnostic. Par exemple, lorsqu'une solution basée sur un algorithme d'exclusion mutuelle est utilisée, les propriétés permettent de contrôler si elle est bien implémentée.

D'un certain point de vue, les *problem frames* jouent le même rôle que les *problem cases*, car tous deux permettent de décrire un domaine. L'approche par problèmes permet aux praticiens d'acquérir de l'expérience et de développer une expertise mais se concentre sur l'espace du problème et sa décomposition en sous problèmes, et peu sur l'espace des solutions [Hal+02]. Or si il existe des arguments convaincants qui justifient une compréhension précoce des besoins des parties prenantes (accent mis sur le problème), d'autres tout aussi convaincants justifient la construction précoce d'une architecture système-logiciel adaptée

(concentration sur la solution). Nous avons pu constater dans notre environnement de travail, que les solutions réutilisées étaient de différentes natures, patrons ou composants, et nous devons toutes les considérer. Nos *problem cases* regroupent toutes ces formes de solutions. Ce partitionnement est d'autant plus important que la vérification et le diagnostic dépendent du type de solution, comme nous le verrons par la suite.

Le *CBR* propose d'encapsuler problèmes et solutions ensemble au sein d'un *cas*. La difficulté de cette approche réside dans la définition de ce cas, car la pertinence des cas retrouvés dépend entièrement de cette définition et de la possibilité de l'évaluer. C'est pourquoi, cette approche ne s'adapte pas bien quand les informations sont trop hétérogènes, comme celles produites par la vérification par model checking. Par exemple l'approche par model checking impose que tout problème puisse être vérifié, et donc qu'il amène avec lui un ensemble de propriétés formelles, définies au regard d'une architecture traduite par exemple sous la forme d'un automate.

## 5.4 La méthode

Des mouvements se déroulent parfois inter-espaces. Les approches strictement orientées solutions sont efficaces quand le domaine est bien connu, et pour lequel les solutions sont associées à des problèmes déjà rencontrés. Or dans le logiciel, les problèmes posés aux ingénieurs sont changeants et pervasifs. A titre d'exemple, la mise en place de pratiques de sécurité pour faire face aux attaques dans les systèmes logiciels est aujourd'hui primordiale dans toutes les entreprises. L'ingénierie du domaine évolue et saisir l'ensemble des problèmes posés aux applications logicielles est devenu une utopie. Des allers-retours sont constamment effectués entre de nouveaux problèmes et leurs solutions. Les *problem cases* présentés dans le chapitre précèdent sont une proposition pour articuler l'espace du problème et l'espace de la solution au sein du domaine. Dans l'ingénierie de l'application, les propriétés du *problème de l'application* doivent être respectées par la solution de l'application, puis lorsque cette solution est vérifiée et conservée, certaines de ses propriétés sont intégrées à celles du problème de l'application.

Ces mouvements se déroulent parfois inter-ingénierie. D'un côté l'ingénierie de l'application réutilise les problèmes ou les solutions de l'ingénierie du domaine, et d'un autre côté, quand une solution produite par l'ingénierie de l'application est viable, elle peut être capturée dans le domaine pour une réutilisation future.

Ces mouvements témoignent de la dynamique de l'activité de conception. Une partie de cette dynamique doit être enregistrée car elle favorise la gestion de l'expérience. La conception sélectionne les problèmes du domaine, leurs solutions les plus pertinentes, les applique, les vérifie. Elle capture les solutions construites et vérifiées pour les réutiliser plus tard. Dans ce chapitre, nous présentons le besoin de formaliser cette partie dynamique,

et son impact en terme de vérification et de diagnostic. Nous proposons également une méthode à cet effet qui s'appuie sur la constitution et la réutilisation d'un domaine.

### 5.4.1 Des processus de développements différents

Généralement, le développement de logiciels s'accompagne d'exigences de départ et/ou de choix architecturaux, pouvant amener à un processus de développement en cascade. Ce processus produit des documents d'exigences gelés artificiellement pour une utilisation dans la prochaine étape du cycle de développement. Il peut aboutir à des systèmes aux architectures contraignantes, qui limitent les utilisateurs et les développeurs en empêchant les modifications inévitables et souhaitables des exigences. Le modèle de cycle de vie en spirale résout de nombreux inconvénients d'un modèle en cascade. Il fournit un processus de développement incrémental, dans lequel les développeurs évaluent successivement les risques liés à l'évolution des projets afin de gérer les exigences instables et les coûts. Un cycle de vie en spirale, de grain encore plus fin, reflète à la fois les réalités et les nécessités du développement logiciel moderne. Un tel cycle de vie amène à développer des architectures logicielles à la fois stables et adaptables face à l'évolution des besoins. La pierre angulaire de ce processus est que les développeurs définissent simultanément les exigences du système et l'architecture.

### 5.4.2 Processus de développement intégrant problèmes et solutions

Un tel partitionnement de la connaissance relève également du domaine des architectures logicielles. Les architectures logicielles faisant partie intégrante du domaine de la solution. Les structures de problèmes et les architectures logicielles peuvent être considérées comme complémentaires. Même des problèmes, modestement complexes, obligent à examiner en détail l'architecture de la solution, ce qui va à l'encontre des *problem frames* dont le but est de retarder la prise en compte de l'espace de la solution jusqu'à ce que le problème soit bien compris. Dans [Nus01] les auteurs soutiennent que le développement de logiciels modernes doit exploiter la synergie existant entre les domaines de problèmes et de solutions, afin de permettre aux développeurs de logiciels d'explorer les exigences et les possibilités de conception. Cette idée a été étudiée par l'approche *Twin Peaks* [Hal+02], un processus de développement logiciel itératif qui se concentre sur la combinaison de structures de problèmes et de solutions et qui étend les *problem frames* pour inclure des domaines avec un support architectural existant. Elle permet ainsi aux structures architecturales, aux services et aux artefacts d'être considérés comme faisant partie du domaine du problème.

### 5.4.3 Notre proposition

Nous proposons une méthode de résolution de problème découpée en deux espaces, l'espace du problème dans lequel on construit la représentation du problème, et l'espace de la solution où l'on génère et évalue la solution choisie.

Dès lors qu'une solution est réalisée, elle doit être évaluée. Si elle est considérée valide, cette solution amène à reconsidérer le problème. Par exemple, si le choix s'est porté sur une architecture client-serveur, tout nouveau problème prendra en considération le choix client-serveur et ses contraintes. Si la solution n'est pas valide, le processus de diagnostic est déclenché. La méthode effectue par conséquent des allers-retours entre espaces de problèmes et espaces de solutions tout en constituant progressivement une base de connaissance regroupant problèmes et solutions réutilisables.

S'appuyant sur la base de connaissance pour retrouver des problèmes et solutions du domaine, il est possible de décomposer les problèmes en sous problèmes plus simples, plus facile à résoudre ou pour lesquelles il y a déjà une solution que l'on peut réutiliser. Cette approche par décomposition est généralement considérée comme une stratégie importante de résolution de problèmes.

Enfin notre méthode est axée sur la vérification et le diagnostic. Elle aide l'ingénieur à rapprocher les informations de haut niveau avec les observations anormales. Les solutions regroupent des conceptions formelles et des tentatives de vérification, tandis que les *problem cases* sont formalisés par un ensemble de propriétés et une structure de différentes solutions. Des solutions choisies dépend le type d'erreur à chercher lors du diagnostic.

### 5.4.4 Étapes de la méthode

Cette méthode par étapes est présentée par le diagramme d'activité de la figure 5.7. Elle est réitérée jusqu'à ce qu'une solution satisfaisante soit obtenue.

(1) Le problème est formulé comme un ensemble de propriétés et de contraintes (choix architecturaux ou techniques) et capturé dans une base de connaissances. Par exemple, "tout accès à une ressource doit respecter les droits d'accès". (2) Le problème est décomposé en sous-problèmes, soit des problèmes connus - appelés *problem cases* - sélectionnés depuis la base de connaissances, soit des situations inconnues. Par exemple, "un mécanisme d'autorisation est utilisé". (3) Lorsque le besoin d'une vue concrète se fait sentir, nous rentrons dans l'espace de la solution. Les éléments de la solution sont organisés. Par exemple, "l'Auth *problem case* est introduit dans la solution actuelle". (4) Nous considérons comment combiner le *problem case* sélectionné avec la solution [2]. Le *problem case* peut être soit (5) composé avec d'autres parties de la solution, (6) appliqué en tant que patron, ou agir uniquement en tant que spécification auquel cas (7) une conception ad

2. Chaque type de combinaison est représenté avec une forme de flèche particulière.

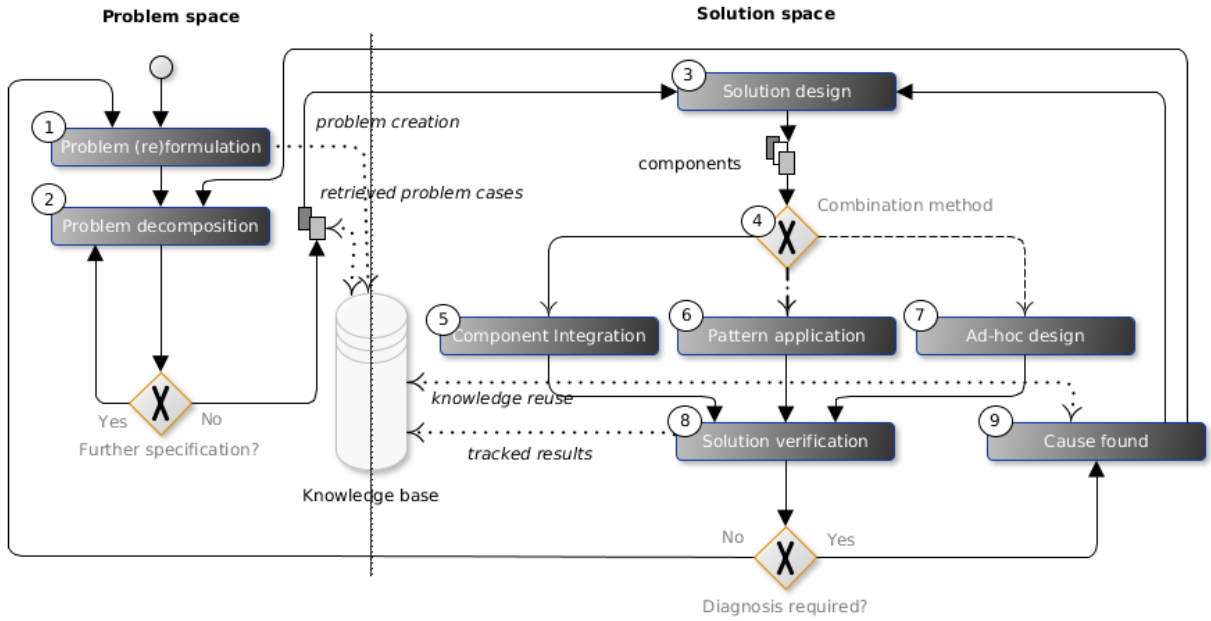


FIGURE 5.7 – Etapes de la méthode

hoc est laissée à l'ingénieur. (8) A ce stade, nous avons construit une partie de la solution attendue ; nous pouvons donc commencer un cycle de vérification. Lorsque des anomalies sont observées, cela déclenche un processus de diagnostic. Les résultats de la vérification sont stockés dans la base de connaissances. (9) Lorsque le processus de diagnostic est effectué, la connaissance des *problem case* peut être utilisée pour faciliter le processus. La conception est corrigée et la tentative de vérification répétée. Dans certains cas, le *problem case* sélectionné ne convient pas, nous devons donc revenir en arrière et retravailler la combinaison des *problem cases*, dans ce cas il peut être utile de garder une trace de cette tentative infructueuse.

Cette méthode par étapes peut être répétée plusieurs fois tant que des composants peuvent être combinés. L'ingénieur se retrouve avec un problème à résoudre pour lequel aucune solution connue n'existe, et pour laquelle une activité classique de conception et de vérification doit être effectuée.

## 5.5 Partager le domaine

### 5.5.1 Capitaliser une solution

#### Attendre son tour et lever son drapeau

Dans l'exemple traité précédemment, deux mécanismes ont été proposés, un mécanisme de drapeau, qui se lève pour indiquer l'intention d'un processus d'entrer en section critique, et un mécanisme de tour, qui impose aux processus d'attendre leur tour avant d'entrer en section critique.

Ciprian tente alors de combiner ensemble ces deux mécanismes ; $Plc_1$ et $Plc_2$ doivent lever leur drapeau avant d'accéder à la ressource, mais l'entrée est accordée uniquement tour à tour. Lorsque qu'un processus, par exemple $Plc_1$, lève son drapeau, il commence par céder le tour au processus concurrent, et attend si le processus concurrent a son drapeau levé. Si le $Plc_2$ n'a pas de drapeau levé, ou que $Plc_1$ a levé son drapeau et c'est en plus son tour, $Plc_1$ peut accéder à la section critique. Une fois la ressource utilisée, $Plc_1$ baisse son drapeau. Le comportement est décrit par la machine à état de la figure 5.8.

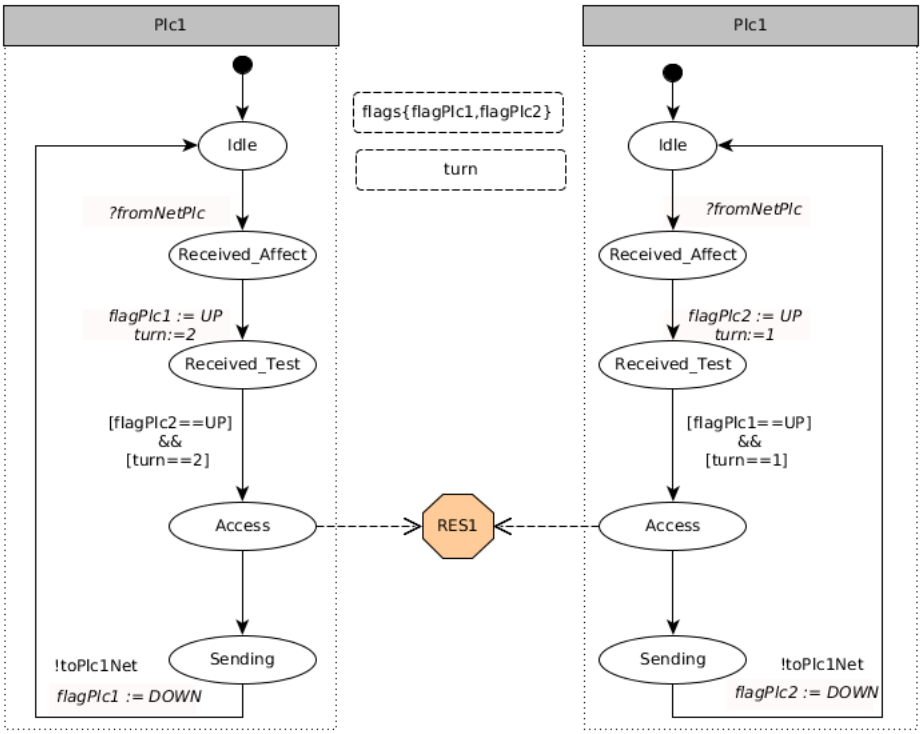


FIGURE 5.8 – Comportement des Plcs, mécanisme de tour et drapeau

La solution est mise en œuvre et évaluée au regard des différentes propriétés :

- Des ***PPDs et PPAs*** appliquées à la solution ($PMutexPlc_1Plc_2$, $PDeadlockFreedom$, $PNoLockoutPlc_1$ et $PNoLockoutPlc_2$).

- Des ***PSAs*** ($PTurn$, $PChange$, $Plc1FlagUpBeforeAccess$, $Plc_2FlagUpBeforeAccess$).

L'ensemble des propriétés est bien vérifié, la solution du tour combinée à celle du drapeau résout donc l'énoncé du problème.

### Sample case

Comme cette solution est valide, il est intéressant de pouvoir la capitaliser pour une réutilisation future et cela impose alors d'en regrouper les informations les plus pertinentes. Nous nommons *sample case* la capitalisation d'une solution. Un *sample case* regroupe des éléments de vérification, de modèle et de spécification. Un élément de vérification (*verification element*) est un artefact de connaissance appartenant au domaine de la vérification par model checking. L'élément principal de la vérification est la tentative de vérification que nous appelons *run*. Un *run* s'effectue à partir de différentes informations, des éléments de modèles bien sûr, mais aussi des propriétés (*property verification*), des scénarios d'exécutions (*verification scenario*), ainsi qu'un model checker et sa configuration (*modelchecker configuration*) désignant par exemple le type d'exploration (BFS, DFS etc...). Un *run*, qu'il se termine ou pas, fournit différents résultats, des *logs*, des informations sur l'espace d'état exploré (*state space info*), différents métriques (*reach time* ...), et enfin des résultats sur les vérifications de propriétés (*property verification result*). Ces résultats peuvent être donnés sous la forme d'une réponse (*answer*) par exemple "la propriété est violée", ou bien sous la forme d'une *trace* (potentiellement un contre-exemple). Cette *trace* est composée de *configurations*, et de transitions entre ces configurations (*configuration transitions*) issues du *LTS*. Une *configuration* contient une liste d'éléments (*configuration elements*) relative aux éléments de modèle ou de spécification.

Un *run* s'effectue sur la base d'un modèle formel (*model*) d'un certain type (*model type*) comme par exemple un protocole ou un contrôleur. Il est composé d'un ensemble d'éléments de modèle (*model element*) comme des automates (*process*), des états (*state*), des *transitions*, des événements (*events*), des *actions* ou bien encore *variables*. Un modèle est exprimé à travers diverses implémentations en Fiacre, ou Promela par exemple (*implementation*).

Pour vérifier un modèle, une tentative de vérification se base sur un ensemble d'éléments de spécification (*specification element*). Ceux-ci sont des propriétés informelles (*informal property*), ou formelles (*formal property*), exprimées par une formule logique (*formula*) au moyen d'un langage (*LTL, CTL*). Une formule logique repose sur des propositions atomiques (*atomic proposition*) liées par des opérateurs logiques et temporels. Une propriété peut être liée au problème de l'application (*application problem property*), au

problème du domaine (*domain problem property*) ou encore à la solution de l'application (*solution application property*).

Des descripteurs complètent les éléments de vérification, de design et de spécification, comme la date ou heure de création de l'élément (*creationdate*), les rôles (*author*) ou des notes descriptives (*description*).

Un extrait d'un *sample case* est fourni par la figure 5.9. Cette figure montre différents éléments qualifiant le *sample case*, des *specification elements*, des *model elements* ainsi que des *verification elements*. Un *sample case* est, par exemple, caractérisé par une implantation du tour et du drapeau exprimée en langage Fiacre (*implementation*).

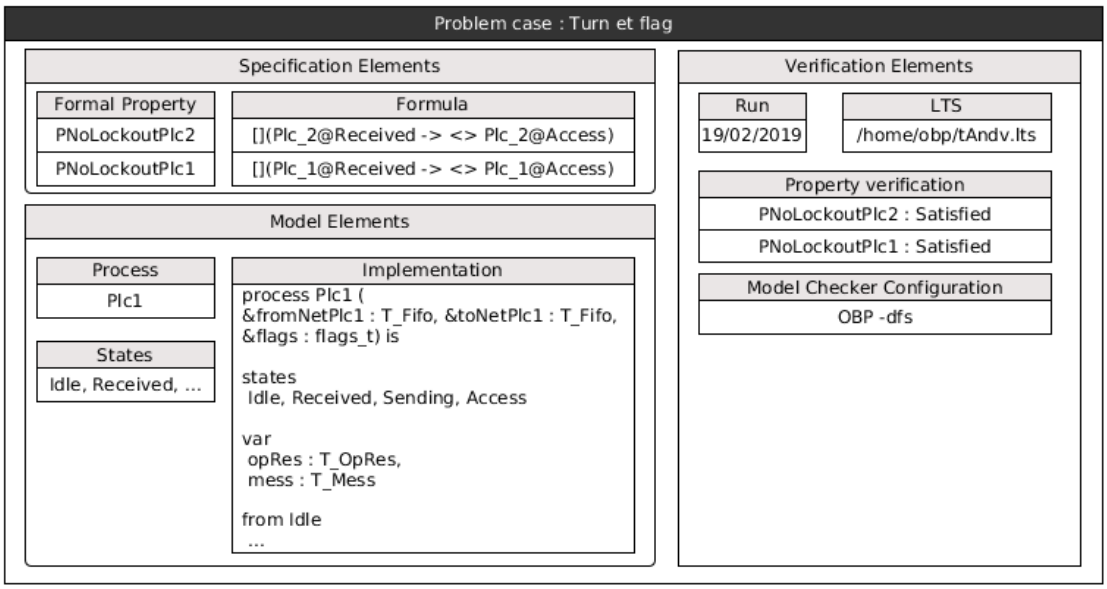


FIGURE 5.9 – Extrait du sample case "Tour et Drapeau"

Un *sample case* structure et conserve une solution à un problème. Il est associé à des descripteurs qui permettent une recherche à base de mots-clés. Il présente synthétiquement une solution qui peut être parcourue rapidement.

### 5.5.2 Les autres formes de solutions

Un *sample case* est une solution que l'on capitalise dans le but de pouvoir la réutiliser, mais conservée sous une forme brute, elle ne peut être réutilisée que dans un contexte similaire. Supposons par exemple que Ciprian réutilise le *sample case* d'exclusion mutuelle avec tour et drapeau, mais cette fois-ci entre deux nouveaux processus qui se comportent différemment de $Plc_1$ et $Plc_2$. Dans ce cas le *sample case* ne pourra pas être réutilisé tel qu'il est stocké, il faudra soit l'adapter soit s'en inspirer.

### Pattern case

Pour permettre une réutilisation plus générale, il faut dissocier les parties relatives à l'application des parties strictement liées au domaine. Le *sample case* de tour et drapeau est similaire à celui proposé par Gary Peterson, dont l'algorithme est fourni par le listing 5.1. Cet algorithme comprend deux processus $Pr_i$ et $Pr_j$, dont le comportement se découpe en quatre parties, la section non critique, l'opération d'entrée dans la section critique, la section critique et l'opération de sortie de la section critique. Les drapeaux sont représentés par un tableau de deux booléens (*drapeau[2]*), dont la levée (valeur à vrai) indique la volonté d'un processus d'atteindre la section critique. Celui-ci est levé au cours de l'opération d'entrée, puis baissé durant l'opération de sortie. Une variable *tour* permet de définir l'équité entre les processus pour accéder à la section critique. Lorsque qu'un processus réalise l'opération d'entrée, il passe le tour à l'autre processus. Ainsi, un processus ne peut pas accéder à la section critique si l'autre processus lève son drapeau et si c'est son tour. On retrouve dans cette solution les éléments du problème d'exclusion mutuelle.

```
1  bool drapeau[2]; drapeau[0] = false; drapeau[1] = false; int tour;
2  Pri:
3    drapeau[0] = true; tour = 1;                  // NCS
4      while (drapeau[1] && tour == 1){ ... }// Trying
5      ...                                         // CS
6      drapeau[0] = false;                         // Exit
7  Prj:
8    drapeau[1] = true; tour = 0;                  // NCS
9      while (drapeau[0] && tour == 0){ ... }// Trying
10     ...                                         // CS
11     drapeau[1] = false;                         // Exit
```

Listing 5.1 – Algorithme de Peterson

Un tel algorithme est assimilable à un patron de conception [Gam95], c'est à dire une solution générique à un problème de conception dans un contexte particulier. Nous appelons *pattern case* la capitalisation d'une solution sous la forme d'un patron qui répond à un problème de domaine. Cette solution est capturée sous la forme d'un comportement et/ou une structure abstraite, qui s'appuie sur des concepts et propriétés d'un problème de domaine. Un patron regroupe également des caractéristiques qui lui son propres et qui permettent sa mise en œuvre ou son choix, comme des implémentations, des exemples d'utilisation, des forces, des faiblesses ou des conséquences.

Un algorithme de Peterson est un patron qui offre une solution au problème du domaine de l'exclusion mutuelle. Il manipule les concepts de ce problème (NCS, Trying, CS et Exit), et met en œuvre ses propriétés. Par exemple la propriété $PMutexPr_iPr_j$ doit être respectée par le patron. En tant que solution, le patron fait émerger de nouvelles propriétés

que nous appelons *propriétés de solution du domaine* (PSD). Par exemple, la propriété $Pr_iFlagWhenTrying$ énonce qu'un drapeau doit être levé dans l'opération $Trying$. Elle est inhérente à cette solution au problème d'exclusion mutuelle. Elle est exprimée comme suit :

$$\boxed{Pr_iFlagWhenTrying : Pr_i@Trying \rightarrow \Box(Pr_i.flag)} \tag{5.7}$$

Lors de la vérification, les $PPD$ et les $PSD$ permettent d'assurer la bonne application du patron ainsi que son applicabilité. Imaginons un nouveau patron appelé *Atomicité*. Celui-ci stipule qu'aucune action ne peut s'intercaler entre la lecture et l'écriture d'une ressource partagée. Cette solution peut être vérifiée grâce à une $PSD$ que l'on nomme $PAtomicité$. Appliquons ce patron au modèle avec drapeaux de la section 5.2.4, dans lequel les drapeaux sont des ressources partagées entre les $Plcs$, et sont lus puis modifiés dans deux instructions différentes. Lors de la vérification, il faut s'assurer que les $PPD$ et les $PSD$ sont assurées. Mais dans cette solution la $PSD$ appelée $PAtomicité$ est violée. La lecture et l'écriture sont réalisés dans deux états différents, l'ensemble n'est donc pas atomique, et le patron est mal appliqué. Supposons maintenant que l'on modifie le modèle de sorte que les $PSD$ soient vérifiées. Le patron étant bien appliqué le problème d'exclusion mutuelle est réglé. Cependant, la vérification montre que la $PPD$ $PNolockout$ est violée, il y a famine (réglable par le mécanisme de tour). Une $PPD$ violée indique que le patron n'est pas toujours applicable. Un patron amène donc à réfléchir à sa mise en œuvre mais aussi à son applicabilité.

### Component case

Un *sample case* est une solution que l'on peut réutiliser telle quelle dans un même contexte (à la manière d'un copier-coller) et un *pattern case* est une solution générale que l'on peut appliquer à un nouveau contexte. Autre cas, peut-on recopier une solution dans un nouveau contexte ? Cette dernière possibilité est offerte par le *component case*. Il représente un élément de solution rendant un service et pouvant être incorporé à une application par le biais d'*interfaces*. Un composant spécifie des *interfaces fournies* et des *interfaces requises*, interfaces dont le composant a besoin pour fonctionner. Le service rendu par le *component case* est réalisé par :

- **Un processus concurrent** ayant une machine à état et dont les services sont invoqués par envois de messages. Dans ce cas il faut adapter les processus utilisateurs de ce composant pour qu'il puissent communiquer avec lui (potentiellement communiquer avec lui de façon synchrone).
- **Une fonction synchrone** appelée par les processus utilisateurs.
- **Une sous machine à d'états** intégrée à la machine à état du processus utilisateur

et connectée par des transitions.

Dans l'illustration 5.10, un *component case* rendant le service d'exclusion mutuelle est décrit par un processus concurrent. Les processus utilisateurs $Plc_1$ et $Plc_2$ sont reliés à celui-ci grâce à des canaux asynchrones. Pour invoquer les service du *component case*, les processus utilisateur doivent implémenter une interface.

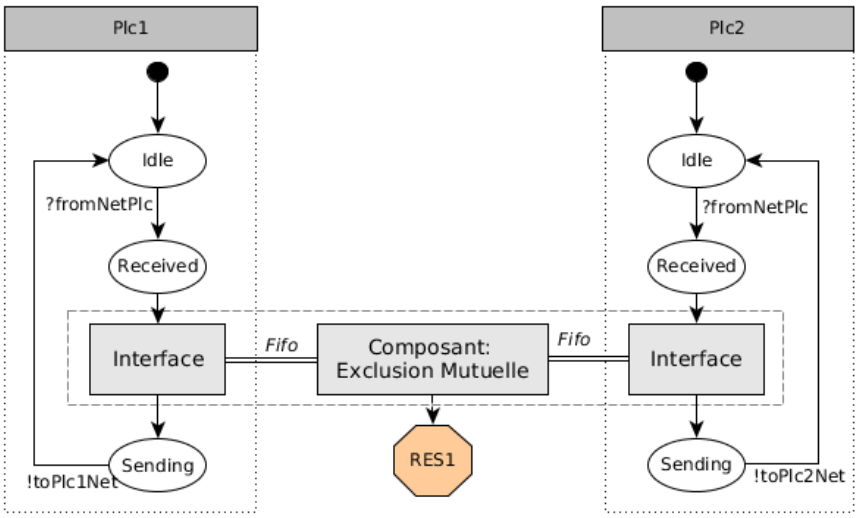


FIGURE 5.10 – Composant d'exclusion mutuelle

### Problem case

Que la solution de domaine soit un *sample case*, un *pattern case* ou un *component case*, tous peuvent être solutions d'un même problème appelé *problem case*. Le *problem case* est un problème de domaine regroupant des concepts et des relations (au sens ontologique), des comportements ainsi que des PPDs considérées comme abstraites au regard des solutions. Ainsi, les *problem cases* sont parfois résolus par des *solutions de domaine* (*sample case*, *pattern case* ou *component case*) qui en manipulent les concepts et en appliquent les propriétés.

Reprenons l'exemple de l'exclusion mutuelle. L'énoncé du problème présenté dans la section 5.2.1 constitue un *problem case*. Il défini les *concepts* NCS, CS, des *relations* entre ces concepts NCS -> CS, et un ensemble de propriétés tel que $PMutexPr_iPr_j$.

Différentes solutions à ce *problem case* ont été proposées, le *sample case* de tour et drapeau, le *pattern case* de Peterson, et le *component case* d'exclusion mutuelle. Toutes ces solutions se conforment aux concepts et comportements du *problem case*, et en appliquent les propriétés. $PMutexPr_iPr_j$, par exemple, est appliquée dans les solutions par la propriété $PMutexPlc_1Plc_2$. Si les solutions partagent les $PPD$ du problème qu'elle

solutionnent, elles amènent également leurs propres propriétés ($PSD$), par exemple la solution *pattern case* Peterson amène la propriété $Pr_iFlagWhenTrying$.

Du point de vue de la vérification et du diagnostic, le *problem case*, tant par ses propriétés que ses concepts, permet de remonter la nature des connaissances sur le système au niveau domaine, et ainsi réduire le fossé sémantique.

## 5.6 Discussion

### 5.6.1 Problèmes et solutions

Quand la connaissance technique n'est pas suffisante pour résoudre un problème, la connaissance du domaine apporte un nouveau point de vue sur le système et les traces. Avec les connaissances de domaine, la trace est perçue du point de vue du problème que l'on cherche à résoudre, simplifiant par conséquent l'activité de diagnostic. Reste un problème important, les connaissances liées au model checking et les connaissances de domaine sont sémantiquement éloignées (problème du fossé sémantique). Pour le réduire, il faut définir des corrélations entre les éléments techniques et les éléments du domaine, par le biais d'annotations, de prédicats ou d'observateurs.

Le processus de résolution de problème implique une multitude de cycles de conception-vérification. Lors de ces cycles, des solutions sont produites, conservées pour être améliorées ou bien abandonnées. A chaque vérification, la solution est évaluée au regard de propriétés du problème du domaine ($PPD$) appliquées à cette solution. Mais chaque solution est unique, et fait émerger des propriétés qui lui sont propres, les propriétés de solution de l'application ($PSA$).

Parfois il est difficile de trouver une solution et les cycles de conception-vérification s'accumulent. Réutiliser ou s'inspirer d'une solution est un moyen de réduire la quantité de ces cycles. On peut donc tirer avantage de conserver une solution satisfaisante pour un usage ultérieur. Il existe différentes natures de solution qui dépendent du besoin d'adaptation de la solution au système :

- **Les *sample cases*** sont des solutions capturées telles qu'elles ont été réalisées, et gardent un lien fort avec le contexte.

- **Les *pattern cases*** sont des solutions pouvant être appliquées en dehors du contexte initial.

- **Les *component cases*** sont des solutions réutilisables par intégration dans la solution.

Ces solutions répondent toutes à des *problem cases*, parfois même, au même *problem case*. Un *problem case* sert à la fois à comprendre et à capturer les problèmes passés, et ne définit pas des éléments de solution mais uniquement des éléments de problèmes. Grâce

à ces *problem cases*, les ingénieurs constituent une base d'expertise réutilisable, et liée à leur domaine d'ingénierie. *Les problem cases* fournissent des modèles pour raisonner sur la solution choisie. Ils facilitent l'activité de diagnostic car ils fournissent des concepts de domaine et des propriétés.

### 5.6.2 Deux ingénieries

Il est possible de classer les systèmes logiciels selon leur domaine métier et leurs tâches associées. Les systèmes d'un même domaine partagent alors un ensemble de caractéristiques qui leur sont communes. Si on considère le domaine de la sécurité, celui-ci possède un grand nombre de caractéristiques communes aux systèmes de ce domaine, comme les notions de "droits d'accès" ou de "vulnérabilité". En capturant ce domaine, il devient possible de concevoir de nouveaux produits en réutilisant ces éléments de domaine. Les bénéfices sont divers, réduction du temps de conception ou de vérification, amélioration de la qualité ou encore diminution du coût de conception. L'ingénierie du domaine est une approche systématique pour atteindre cet objectif [CE00].

**Définition 5.1.** *L'ingénierie de domaine* est la capacité de rassembler, d'organiser et de stocker l'expérience acquise dans la construction de systèmes ou de parties de systèmes dans un domaine particulier sous la forme d'artefacts réutilisables, ainsi que de fournir un moyen adéquat de réutiliser ces éléments lors de la construction de nouveaux systèmes [CE00].

De manière générale, l'ingénierie du domaine comprend différentes activités : l'analyse du domaine, la conception du domaine et l'implémentation du domaine. Ces activités produisent des artefacts réutilisables par l'ingénierie de l'application. L'ingénierie de l'application réalise concrètement le produit à travers différentes activités, l'analyse des exigences, le développement et la configuration du produit, l'intégration et tests. Elle bénéficie des artefacts produits par l'ingénierie du domaine pour progresser plus rapidement vers une application satisfaisante. Car les *problem cases* capturent des éléments d'un domaine à des fins de réutilisation, nous pouvons faire l'analogie avec l'ingénierie du domaine. L'illustration 5.11 présente les activités mises en œuvre du point de vue de l'ingénierie de l'application et du point de vue de l'ingénierie du domaine pour l'utilisation des *problem cases* dans un processus de conception-vérification.

L'ingénierie de l'application réalise l'analyse des exigences, et produit des *propriétés de problème de l'application* ($PPA$). L'application est alors conçue pour aboutir à une solution. Lors de la conception, cette solution peut être produite sans connaissances de domaine, dans ce cas la solution fait uniquement émerger des *propriétés de solution de l'application* ($PSA$). Quand cette solution est produite à partir de connaissances de

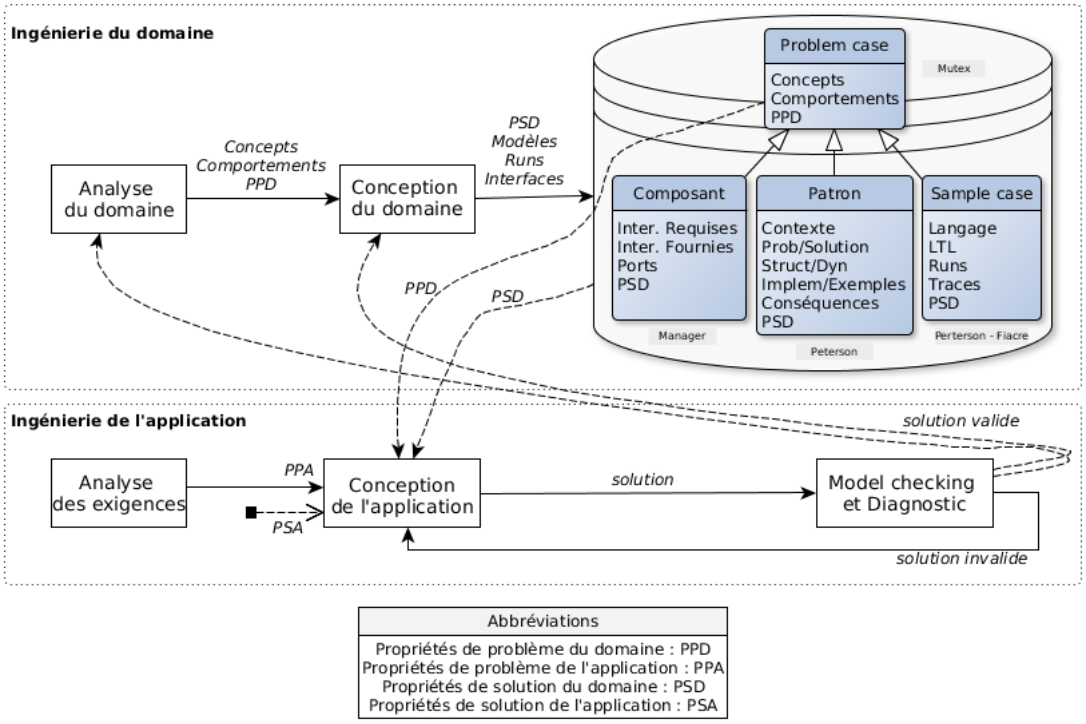


FIGURE 5.11 – Ingénierie du domaine

domaine, de nouvelles propriétés doivent être aussi prises en compte, les *propriétés de solution de domaine* et les *propriétés de problème du domaine*. Après vérification par model checking, le diagnostic bénéficie le cas échéant des éléments du domaine. Si la solution est invalide, une nouvelle solution est produite lors de la conception de l'application. Les propriétés *de solution du domaine ou/et de solution de l'application* peuvent alors être remises en cause si la solution change. Quand la solution est valide, celle-ci peut être capturée pour un usage ultérieur au sein du domaine.

L'ingénierie du domaine analyse le domaine pour produire des concepts, comportements et *propriétés de problème du domaine* qui sont considérés comme abstraits au regard des solutions. Ces informations sont capturées dans un *problem case*. Après l'analyse, la conception du domaine réalise des solutions de domaine qui résolvent le problème. Ces solutions de domaine sont capturées dans des *component cases*, *pattern cases*, ou *sample cases*. Pour qu'une solution de domaine réponde à un problème de domaine, les concepts et les propriétés du problème de domaine sont appliqués à la solution.

### 5.6.3 Les quadrants

Le diagnostic s'appuie sur l'expression exhaustive des spécifications et donc des propriétés formelles si la détection des dysfonctionnements utilise le model checking. Or certaines spécifications sont manquantes du fait du caractère mal défini de la conception. En réalité les spécifications d'un projet de conception sont habituellement d'un niveau abstrait [Vis09], il faut donc trouver ces spécifications manquantes qui peuvent être en relation avec les causes du dysfonctionnement.

Pour découvrir ces propriétés formelles, nous proposons d'effectuer une partie de l'analyse et de la conception au niveau du domaine. On admet généralement qu'il existe deux ingénieries qui cohabitent, l'ingénierie de l'application et l'ingénierie du domaine. L'ingénierie de l'application réutilise des éléments appartenant au domaine, entre autres des propriétés formelles et des éléments de réalisations, et produit de nouveaux éléments. Parallèlement, l'ingénierie du domaine produit des éléments de spécification et de réalisation pour l'ingénierie de l'application.

Ces ingénieries sont spatialement divisées, avec d'un côté l'espace du problème qui définit les problèmes du domaine et ceux de l'application, et d'un autre côté l'espace de la solution, définissant les solutions du domaine ainsi que celles de l'application. On a donc une répartition en quatre cadrants, comme illustré sur la figure 5.12.

Une des propositions de cette thèse est de mettre en avant le caractère complémentaire et indissociable des espaces du problème et de la solution. Les exigences énoncées au niveau problème doivent être exprimées dans la solution afin de vérifier si elles sont satisfaites. De plus certaines contraintes de conception d'architecture s'expriment sous forme d'exigences sur la solution. Certaines propriétés se situent donc plutôt dans l'espace de la solution.

Les propriétés formelles du cadrant 1 (PPA) formalisent des exigences qui sont propres à l'application et qui peuvent être être issues ou non du domaine. Les propriétés formelles du cadrant 2 (PPD) énoncent des invariants, des contraintes ou même des structures fréquemment employées dans le domaine. Les propriétés formelles du cadrant 3 (PSD) contraignent l'architecture et les réalisations. Enfin les propriétés formelles du cadrant 4 (PSA) expriment l'ensemble des exigences et des contraintes et les allouent aux éléments de la solution. Il existe une dynamique entre ces cadrants tout au long de la conception.

### 5.6.4 Application de la méthode

Dans ce chapitre, nous avons proposé une méthode de résolution de problèmes. Elle s'appuie sur le principe d'aller-retours entre espace du problème et espace de solution, structuré par les *problem cases* fournis par l'ingénierie du domaine.

Dans le chapitre suivant, nous allons illustrer l'emploi de la méthode sur le SAE présenté dans la thèse de F. Obeid [OD17]. F.Obeid cherche à sécuriser, conformément à une

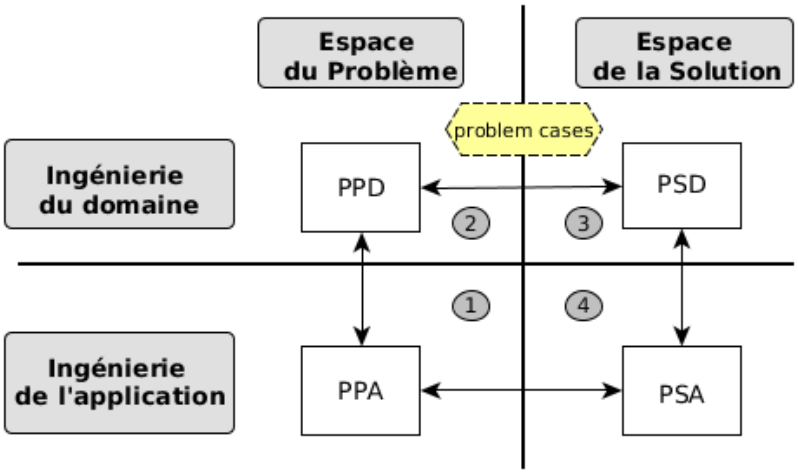


FIGURE 5.12 – Espaces du problème et de la solution versus ingénierie du domaine et de l'application

politique de sécurité donnée, une architecture SCADA en s'appuyant sur la réutilisation de patrons de sécurité. Du point de vue du problème il s'agit d'exprimer des propriétés de sécurité et de définir une architecture abstraite sur laquelle les propriétés sont énoncées. Différentes solutions sont produites et illustrent les différents types de *problem cases*.

Chapitre 6

# Application de la méthode à la sécurisation d'une architecture SCADA

## 6.1 Problème de l'application

Reprenons le cas illustratif. L'application est composée du client $Clt_1$ réalisant l'opération *Read* sur la ressource $Res_1$ détenue par le composant $Plc_1$. L'une des propriétés à garantir est $PRT_1$ qui énonce que tout accès aux ressources doit respecter les droits d'accès.

Après avoir réalisé un mécanisme d'exclusion mutuelle entre $Plc_1$ et $Plc_2$ pour la ressource $Res_1$, Ciprian doit sécuriser l'architecture car celle-ci est maintenant soumise à divers accès extérieurs qui pourraient s'avérer dangereux. Il ne connaît pas bien le domaine de la sécurité, alors prudent, il préfère laisser à Philippe, architecte et expert en sécurité, le soin de concevoir ce modèle.

Dans la section précédente nous avons décrit les *problem cases* et pourquoi nous les utilisons pour décrire le domaine. Différentes questions se posent alors, comment utiliser les *problem cases*, et comment ces *problem cases* alimentent la vérification et le diagnostic.

## 6.2 Domaine de la sécurité

On considère qu'il existe un domaine relatif à la sécurité, et qu'il est composé d'un ensemble de problèmes et de solutions exprimées par des *problem cases*. La présentation de ce domaine est en partie extraite du travail de [OD17] sur les patrons de sécurités.

### 6.2.1 Architecture abstraite

Pour exprimer les propriétés en LTL, et plus généralement pour exprimer les propriétés d'un problème, il est nécessaire d'imposer une architecture minimale sur les solutions. Comme le montre la figure 6.1, nous définissons dans l'espace du problème cette architec-

ture par le biais d'une structure abstraite et d'un comportement générique, semblable au concept de *machine* dans [Jac01].

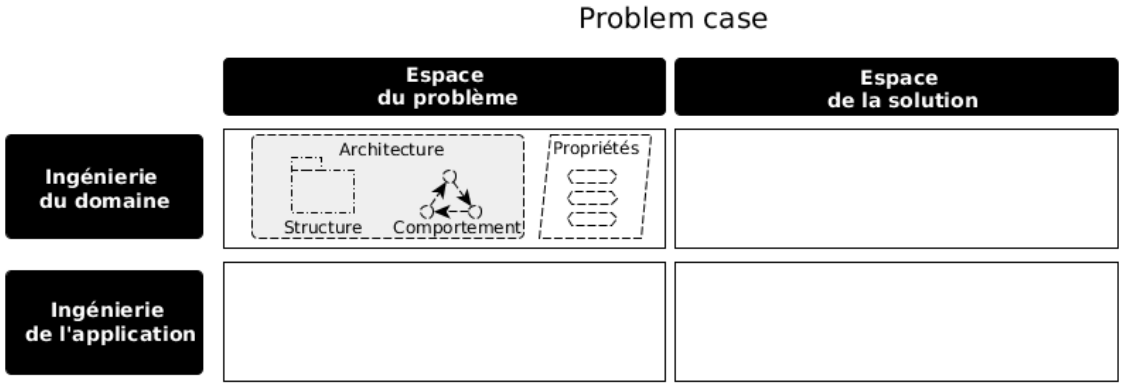


FIGURE 6.1 – Architecture abstraite des problem cases

La structure abstraite définit les constituants de base d'une architecture SCADA (clients, composants, ressources ...). Nous prenons pour hypothèse que tout mécanisme de sécurité est inclus dans un constituant de l'architecture de type composant. Pour accéder à un composant de l'architecture, un client *Clt* doit émettre une requête *req*. Pour accéder à une ressource *res* détenue par un composant, cette requête contient l'opération à effectuer sur la ressource *opRes*.

Un composant possède le comportement générique décrit dans la figure 6.2. De l'état *Idle*, le composant capte un message (*receive()*) provenant d'une fifo d'entrée, l'emmenant dans l'état de réception nommé *Received*. Si ce message lui est destiné (*mine()=true*), il le traite (*Compute*), sinon il le relaye (*Sending*). Après traitement, l'accès à la ressource est réalisé (*Access*), et une réponse est produite, potentiellement négative (*NAK*), puis transmise au client (*Sending*).

### 6.2.2 Point d'accès unique

Pour s'assurer de l'intégrité d'un système et lutter contre de mauvais usages, une possibilité est de s'assurer que toutes les interactions entre les clients externes au système et le système soient bien autorisées. Cette solution n'est pas viable car elle suppose de vérifier l'intégralité des interactions entre le système (incluant des sous systèmes) et son environnement, réduisant ainsi la performance du système et causant des désagréments pour les utilisateurs. Une solution plus appropriée est de mettre en place un point d'accès unique au système. Ce mécanisme est appelé *SAP* (*Single Access Point*). Grâce à ce point d'accès unique, il est possible de vérifier les interactions du client avec le système confor-

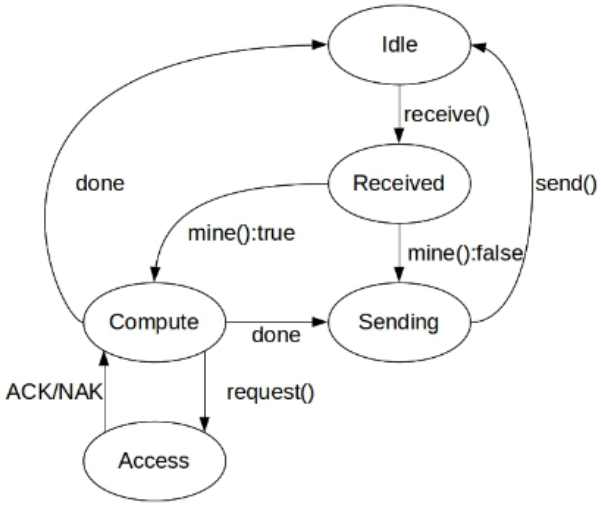


FIGURE 6.2 – Comportement générique

mément à une politique de sécurité donnée, en un seul endroit. Le mécanisme protège les frontières du système, facilite l'identification des clients, permet d'améliorer le contrôle et la surveillance des entrées et réduit la taille du code dédié à la sécurisation.

**Fonctionnement**.

Le mécanisme *SAP* unifie les points d'accès à la ressource *res* détenue par un composant *C_SAP*. Ainsi, toute requête *req* émise par un client *Clt* à destination d'un *C_SAP*, doit passer par le mécanisme *SAP*. Dans un premier temps, celui-ci appose une signature permettant de certifier aux autres composants son passage par *C_SAP*. Dans un second temps, si *res* est accessible, *C_SAP* réalise l'accès, et si *res* est inaccessible, *C_SAP* répond alors directement à la source de la requête avec une réponse négative (NAK) et le traitement se termine.

**Structure**. *SAP* comprend la fonction $check(req)$ qui vérifie si la ressource demandée *res* est accessible pour l'opération *opRes*.

**Propriétés**. Nous ne présentons ici qu'une propriété de disponibilité que nous appellerons *PRT_SAP*. Celle-ci énonce que toute requête demandant l'accès à une ressource détenue par un *C_SAP* (*request(C_SAP,req)*), finira par être contrôlée (*evt_check (C_SAP, req)*, ce qui s'exprime de manière triviale en LTL :

$$\boxed{PRT\_SAP : \Box[evt\_request(C\_SAP, req) \Rightarrow \Diamond evt\_check(C\_SAP, req)]} \quad (6.1)$$

**Solution**. Une solution concrète est fournie par la figure 6.3. Cette solution a été capturée dans un contexte où le composant *C_SAP* fonctionnait conjointement avec d'autres mécanismes de sécurité (*Checkpoint* par exemple).

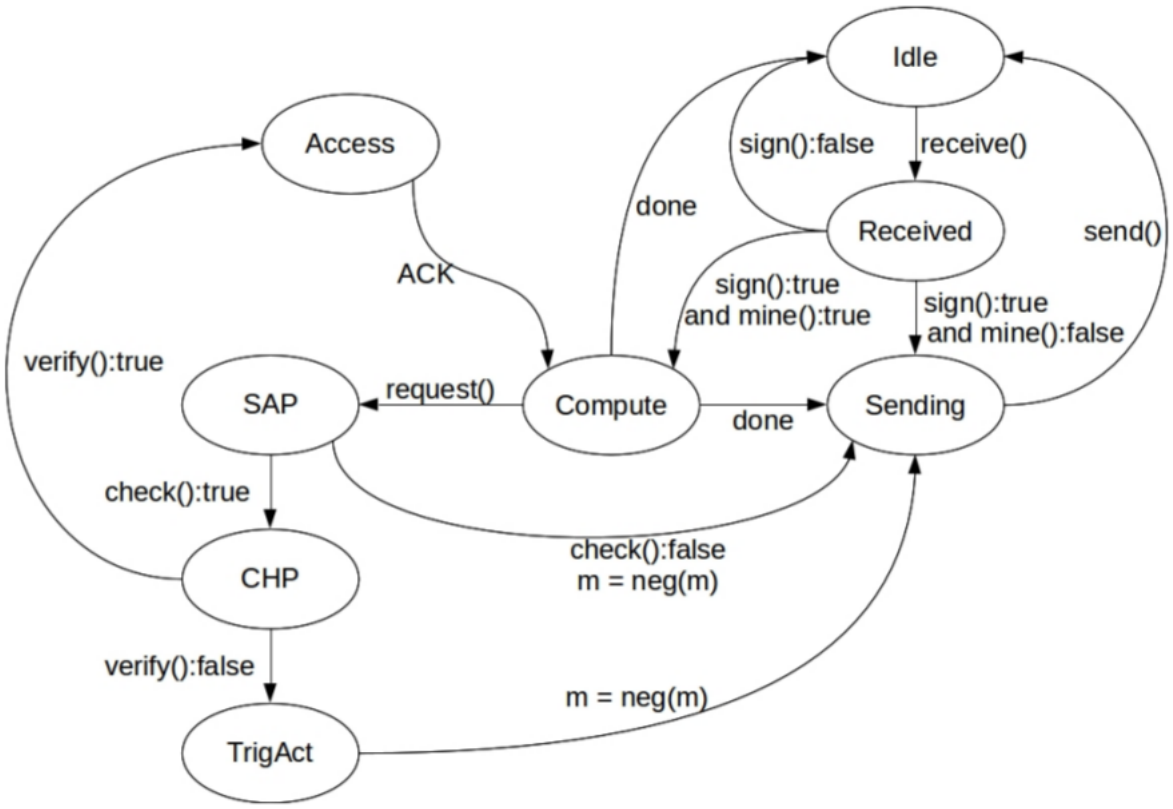


FIGURE 6.3 – Exemple d'une solution utilisant le SAP

### 6.2.3 Point de contrôle

Un système protégé doit être capable de répondre de façon appropriée lors des différents accès, c'est à dire laisser les clients autorisés à accéder au système, mais dans le même temps limiter l'accès aux clients interdits. Dans le cas où une requête viole les exigences de sécurité, une contre-mesure peut être déclenchée (pour ignorer un message ou envoyer une réponse négative). Il faut aussi prévoir à l'avance l'adaptation du système aux changements d'exigences en termes d'identification et d'autorisation. *CHP* (Checkpoint) est une solution à ce problème. Il peut être considéré comme l'application du pattern Strategy[Gam95] niveau du *SAP* [Sch+13]. Il définit l'interface qui devra être supportée par des implémentations concrètes afin de fournir le service d'identification et d'autorisation au *SAP*.

**Fonctionnement**. Dans l'approche de Obeid [OD17] un *CHP* complète un *SAP*. Soit *C_CHP*, un composant de type *CHP* et *cm*, une contre-mesure. La requête *req* est d'abord contrôlée par un *SAP*, qui, si elle est valide, demande au *CHP* de la vérifier au regard d'une politique de sécurité. Si *req* respecte la politique de sécurité, elle est exécutée (en accédant à la ressource ou en transférant le message). Sinon, le *CHP* déclenche une contre-mesure *cm* appropriée à cette requête. *Cm* peut être une réponse négative envoyée à l'entité source de la demande.

**Structure**. Le *CHP* comprend un composant décrivant des règles conformes à la politique de sécurité (par exemple, des autorisations d'accès), et un composant de contre-mesure. Il possède aussi un ensemble de fonctions. La fonction *policy_conform (C_CHP, req)* retourne vrai si *req* est conforme à la politique de sécurité de *C_CHP*. La fonction *evt_execute (C_CHP, req)* renvoie un événement quand *C_CHP* exécute *req*. La fonction *counterOf (C_CHP, req)* est la contre-mesure de *C_CHP* associée à la *req*. La fonction $verify(req)$ vérifie si *req* respecte la politique de sécurité, et enfin la fonction $trigAct(cm)$ déclenche une *cm*.

**Propriétés**. Nous ne présentons qu'une propriété, *PRT_CHP* énonce que toute requête prise en charge ($execute(c\_chp, req)$) a été vérifiée ($verify(c\_chp, req)$), et est conforme à la politique de sécurité ($policy\_conform(c\_chp, req)$).

$$\begin{array}{|l|}\hline PRT\_CHP : \\ \Box[evt\_execute(c\_chp, m) \Rightarrow evt\_policy\_conform(c\_chp, req) \wedge evt\_verify(c\_chp, req)] \\ \hline \end{array} \quad (6.2)$$

### 6.2.4 Autorisation

Pour éviter qu'une quelconque entité n'accède ou ne nuise à l'intégrité d'une ressource, une solution consiste à déclarer des autorisations d'accès. C'est le rôle joué par la solution d'autorisation *Auth*.

**Fonctionnement**. *Auth* protège l'accès des entités aux ressources. Elle affecte des droits d'accès à une ressource pour une entité et pour une opération (lecture, écriture, exécution). Toute opération *opRes* à une ressource *res* protégée par un composant *C_Auth* intégrant le mécanisme *Auth* nécessite une permission explicite pour l'entité demandeuse. Le mécanisme *Auth* est appelé après que *CHP* ait vérifié la requête.

**Structure**. *C_Auth* possède une liste d'*opRes* attribuant les permissions sur les opérations possibles sur les ressources (*protections_list* et *permissions_list*). Il comprend aussi des fonctions. La fonction *protect (SAP_C, opRes)* ajoute un *opRes* dans la liste *protections_list*. La fonction *allow (SAP_C, clt, opRes)* ajoute un *opRes* sur le client *Clt* dans la liste *permissions_list*. La fonction *isProtected (opRes)* retourne vrai si la ressource est protégée pour l'opération donnée par *opRes*. La fonction *isAllowed (clt, opRes)* retourne vrai si *clt* a une permission explicite de réaliser l'opération donnée par *opRes* sur la ressource. Enfin la fonction *hasRight (clt, opRes)* retourne vrai si *clt* a le droit de réaliser l'opération donnée par *opRes* sur la ressource.

**Propriétés**. La propriété *PRT_AUTH* énonce que tout accès à une ressource ($access(c\_auth, clt, opRes)$), doit respecter les droits d'accès ($right(c\_auth, clt, opRes)$).

$$PRT\_Auth : \Box[evt\_access(c\_auth, clt, opRes) \Rightarrow right(c\_auth, clt, opRes)]. \quad (6.3)$$

**Solution**. Ce patron est accompagné d'une solution sous la forme d'une transformation définie par des règles formelles, et permettant de passer d'un composant non sécurisé à un composant sécurisé avec le mécanisme *Auth*. Cette transformation *trf_bev_access* est exprimée de la manière suivante : pour tous les composants ($C$) de l'architecture dont le comportement est non sécurisé ($behavior(C)$), deviennent sécurisé ($bevahior(c\_sec)$) par la fonction de transformation $trf\_bev(behavior(c), PS)$, conformément à une politique de sécurité donnée ($PS$).

$$\forall c \in compList(arch) \Rightarrow behavior(c\_sec) = trf\_bev(behavior(c), PS) \quad (6.4)$$

Cette transformation n'est assurée que si la solution existante respecte les hypothèses suivantes :

- (1) La réception d'un message est effectuée en lisant le canal d'entrée. Cette réception

doit se produire dans une transition entre l'état *Idle* et l'état *Received*.

- (2) Le message est envoyé via le canal de sortie, lors de la transition entre l'état *Sending* et *Idle*.
- (3) Chaque transition vers l'état *Compute* passe par un l'état source de *Received*.
- (4) L'état d'accès a pour seul état source l'état de traitement.

## 6.3 Adaptation d'un Sample Case

### 6.3.1 Formalisation du problème

Très vite, Philippe décide de mettre en place un mécanisme permettant de protéger la ressource $Res_1$ de toutes attaques ou mauvaises manipulations externes, grâce à un Single Access Point ($SAP$).

Philippe a déjà implémenté un tel mécanisme dans un autre système, il l'avait alors capturé sous la forme du *sample case* défini précédemment. Si Philippe ne peut pas intégrer cette solution tel quelle, car elle inclut du fonctionnel dont il n'a pas besoin, elle lui permet quand même de disposer d'une source d'inspiration ainsi que d'une propriété de problème de domaine ($PRT\_SAP$).

### 6.3.2 Conception de la solution

Philippe modifie le comportement du $Plc_1$ en s'inspirant de l'exemple de $SAP$ extrait du domaine et obtient l'automate illustré sur la figure 6.4. De multiples changements ont été effectués par rapport à l'exemple. Après réception d'une requête provenant de la fifo d'entrée ($fromNetPlc$), le composant vérifie sa signature ($sign = true \wedge mine = true$) avant d'entrer dans le nouvel état *Compute* dont le rôle est de centraliser les traitements. De là, la requête d'accès doit être validée par le mécanisme $SAP$ ($check() = true$), pour que l'accès à la ressource puisse être réalisé. Les états $CHP$ et $AUTH$ initialement présents dans l'exemple ont été shuntés, et l'état *Access* s'est transformé dans l'état *Mutex*. Ce dernier inclut le mécanisme d'exclusion mutuelle avec l'algorithme de Peterson, et réalise l'accès à la ressource. Dans le cas où l'accès n'est pas validé par $SAP$ ($check() = false$), une réponse négative est préparée. Une fois l'accès terminé (*Mutex*), l'état *Compute* prend le relais avant d'envoyer une réponse (*Sending*).

### 6.3.3 Vérification et diagnostic

Pour vérifier le bon fonctionnement du modèle, Philippe doit vérifier la propriété du problème de l'application $PRT_1$. Si la solution proposée par le domaine a été une simple source d'inspiration, le problème de domaine lui demeure inchangé. La propriété abstraite

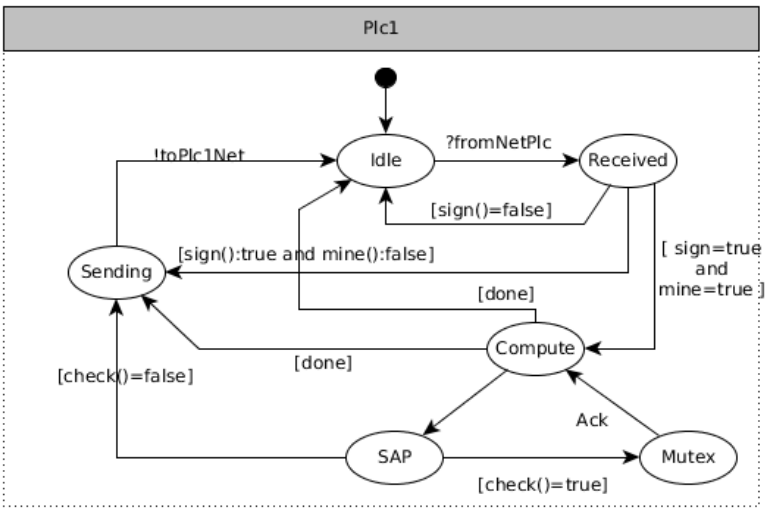


FIGURE 6.4 – Exemple du SAP intégré

*PRT_SAP* doit donc être vérifiée dans le contexte de la nouvelle solution proposée. Cette nouvelle propriété *PRT_SAP_1* est exprimée de la manière suivante :

$$PRT\_SAP\_1 : \Box[evt\_access(Plc_1, Clt_1\_Read\_Res_1) \Rightarrow check(Plc_1, Clt_1\_Read\_Res_1))] \tag{6.5}$$

Supposons que le model checker détecte un contre-exemple à cette propriété. La solution proposée par le *sample case* n'était pas réutilisable, amenant Philippe à construire la solution de manière ad-hoc. Mais elle n'est pas conforme à la propriété de problème de domaine *SAP*. Cette propriété permet à Philippe de préciser le problème car elle informe que c'est bien le *SAP* qui est en cause. La solution est donc mal implémentée, mais malheureusement, comme elle a été conçue de manière ad-hoc il faudra comprendre comment elle a été mise en œuvre pour retrouver l'origine du problème.

### 6.3.4 Conclusion

Un *sample case* a une facette problème et une facette solution. D'un côté, la facette solution du *sample case* ne peut pas être reprise telle quelle et doit être adaptée au contexte. De l'autre, la facette problème amène des éléments (propriétés, architecture abstraite...) qui restent présents dans la solution construite. Ces éléments permettent de préciser le diagnostic comme l'illustre la figure 6.5. Par exemple, les propriétés abstraites énoncées dans le problème de domaine sont appliquées sur l'architecture abstraite. Elles peuvent

être vérifiées sur la solution car celle-ci conserve l'architecture abstraite. La technique de vérification par model checking est orthogonale au domaine, et ce dernier l'aide lors du diagnostic grâce à de nouvelles propriétés qui permettent de préciser le problème, même si, dans le cas d'un *sample case*, la solution reste difficile à diagnostiquer.

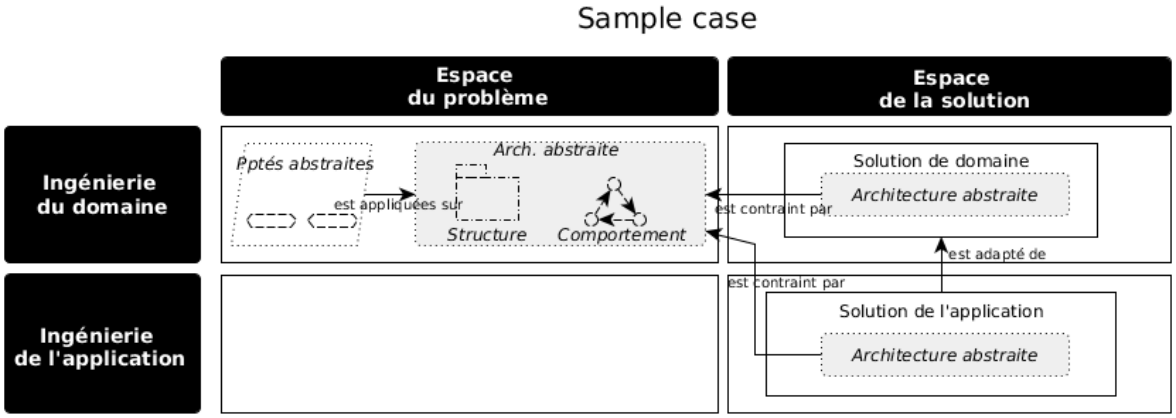


FIGURE 6.5 – Les cadrants et l'utilisation du sample case

## 6.4 Application d'un Pattern case

### 6.4.1 Formalisation du problème

Philippe reçoit de nouvelles exigences qui dictent que si le client $Clt_1$ a des droits d'accès sur une ressource détenue par le $Plc_1$, il doit être en mesure d'y accéder conformément à ses droits. Cette propriété est exprimée en LTL comme suit :

$$PRT\_2 : \Box[evt\_access(plc_1, clt_1, opRes) \Rightarrow right(plc_1, clt_1, opRes)]. \tag{6.6}$$

Le mécanisme *SAP* n'est plus suffisant pour protéger l'architecture SCADA, et des mécanismes de sécurité supplémentaires doivent être appliqués pour garantir les contraintes d'intégrité et de confidentialité du $Plc_1$.

### 6.4.2 Décomposition du problème

Pour résoudre plus simplement ce problème, il faut identifier les problèmes du domaine répondants à ces nouvelles exigences. Une possibilité est de chercher dans la base de *problem cases*, des *problem cases* qui ont des propriétés abstraites se rapprochant des propriétés exprimées à partir des exigences. Cette similarité peut être évaluée sur la structure de la formule logique. En l'occurrence, la structure de la formule logique exprimant la propriété $PRT_1$ est proche de la structure logique exprimant la propriété *PRT_Auth*.

L'ingénieur sélectionne le *pattern case Auth*. *Auth* est composé d'autres sous *problem cases*, *CHP* et *SAP*. Le choix de ce *problem case* impose également d'intégrer un mécanisme de *CHP*. Le choix d'une solution ou d'un problème contraint donc d'intégrer de nouveaux éléments non prévus initialement.

### 6.4.3 Récupération de la solution

*Auth* est capturé sous la forme d'un *pattern case*, le rendant indépendant d'un contexte. Il faut donc le transformer pour le plier au contexte de l'application. Cette transformation ne peut se faire qu'au regard d'un ensemble d'hypothèses que la solution doit satisfaire. Par exemple, il faut reconnaître l'état équivalent à l'état *Idle* dans l'application, emplacement où les transformations pourront être introduites. L'application se déroule donc en trois étapes, premièrement, l'évaluation des hypothèses, deuxièmement les transformations (automatiques ou manuelles), et troisièmement la vérification des transformations.

La première étape consiste à vérifier que l'application satisfait les hypothèses. Philippe fait l'analogie suivante entre l'application existante et les hypothèses :

- (1) La réception d'un message est effectuée en lisant la *fifoNetToPlc*. Cette réception se produit dans une transition entre l'état *Idle* et l'état *Received*.

- (2) L'envoi du message est effectué par une écriture dans la fifo *fifoPlcToNet*, lors d'une transition entre l'état *Sending* et l'état *Idle*.

- (3) Chaque transition vers l'état *Compute* passe par un l'état source *Received*

- (4) L'état *Mutex* a pour seul état source l'état *Compute*.

Si Philippe n'avait pas réussi à lier les hypothèses à l'application existante, il aurait du refactorer l'application pour y appliquer le pattern.

La seconde étape concerne la transformation du patron dans la solution. Celle-ci est décrite formellement dans le patron, et doit simplement être adaptée au contexte. Par exemple le composant $c$ dans la transformation sera ici remplacé par $Plc_1$. Il existe différentes options de transformation, soit sous la forme d'un processus à part entière, soit par l'ajout d'un ou plusieurs états spécifiques au sein de l'automate de comportement existant, soit par l'ajout d'un simple appel à une fonction synchrone. Ces choix dépendent du contexte et des besoins, comme par exemple la performance ou les capacités d'évolution

de l'application, et sont décidés au moment de l'application du patron.

### 6.4.4 Application de la solution

Ici le choix est d'introduire un ensemble d'états dans l'automate de comportement du processus gérant $Plc_1$. Ce choix est motivé par le problème de l'explosion combinatoire qui pourrait être amplifié dans le cas d'un ajout de processus. Le comportement obtenu après transformation est représenté par la figure 6.6. Celui-ci est similaire au comportement *SAP* présenté dans la figure 6.4, excepté que deux nouveaux états complètent le *SAP*, *CHP* et *TrigAct*. Après que le *SAP* ait contrôlé la requête ($check() = true$), le *CHP* vérifie si celle-ci respecte la politique de sécurité par l'appel à la fonction $verify()$. Si le client peut effectuer l'opération sur la ressource ($verify() = true$), l'accès est réalisé dans l'état *Mutex*, sinon l'état *TrigAct* déclenche la contre-mesure appropriée.

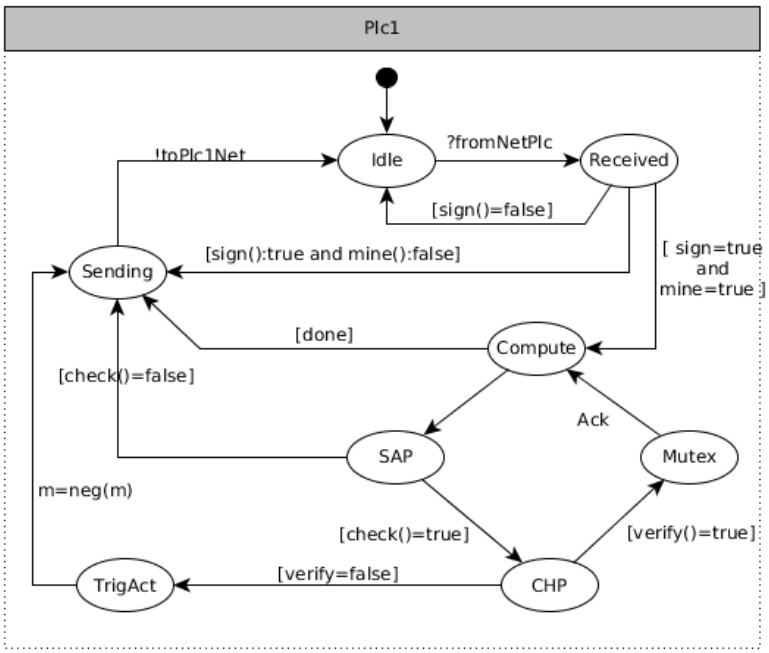


FIGURE 6.6 – Patron Auth intégré

### 6.4.5 Vérification et diagnostic

Pour vérifier que le système fonctionne, le scénario suivant est réalisé : le client $Clt_1$ initie une requête pour accéder à la ressource $Res_1$ du $Plc_1$ avec une opération de lecture (*Read*). Quand $Plc_1$ reçoit la demande, il la traite et répond à $Clt_1$. Comme le client n'a pas les droits en lecture sur cette ressource, il doit recevoir un $NACK$.

Les propriétés suivantes sont évaluées : - la propriété du problème de l'application $PRT_1$, représentant la politique de sécurité définie pour cette application ; - les propriétés du problème de domaine, incluant celle du problème *Auth* (*PRT_Auth*), plus l'ensemble des propriétés de ses sous problèmes, *PRT_Chp* et *PRT_Sap*. Supposons que $PRT_1$ soit violée. Les causes peuvent être les suivantes :

- (1) L'application du patron a été mal réalisée, par exemple, une des hypothèses préalables à la transformation a été mal fixée.

- (2) Ce n'est pas un bon choix de patron, et il ne permet pas de répondre à l'exigence exprimée par $PRT_1$. En supposant qu'une propriété de problème de domaine soit également violée (par exemple *PRT_Sap*), cette propriété va permettre d'isoler le diagnostic et de se focaliser sur le fonctionnement du *Sap*.

### 6.4.6 Conclusion

Dans le cas d'un *pattern case*, la solution du domaine peut être appliquée au nouveau contexte, il y a par conséquent moins d'efforts de conception que pour le *sample case*. Dans notre scénario, cette solution est exprimée sous la forme de règles de transformations. Ces règles prennent leur source dans l'espace du problème et constituent le lien entre le problème et la solution. Les contraintes sur l'architecture de la solution dans l'espace du problème sont prises en compte dans l'architecture de la solution dans l'espace de la solution. D'une architecture abstraite, on passe à une architecture concrète comme l'illustre la figure 6.7.

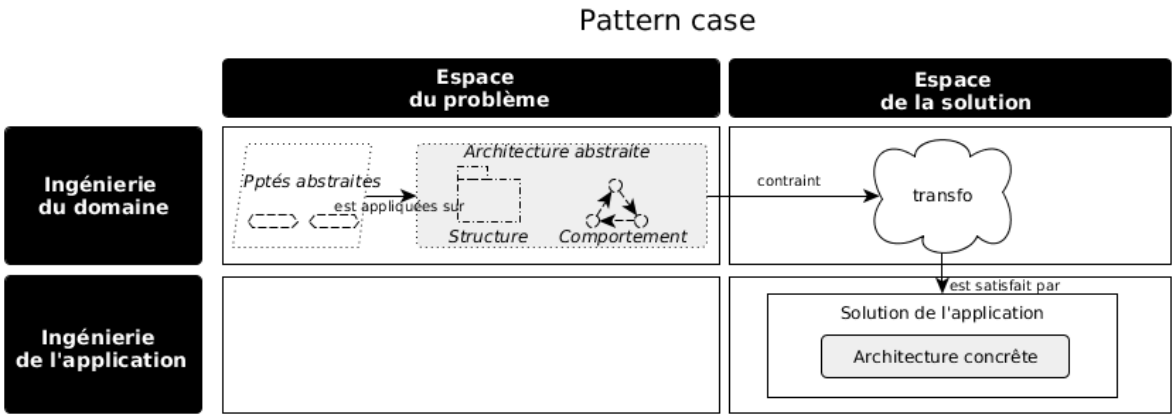


FIGURE 6.7 – Les cadrants et l'application du pattern case

Du point de vue du diagnostic, la solution de domaine déjà éprouvée réduit les erreurs possibles. De plus, le patron ramène un ensemble de propriétés abstraites qui vont pouvoir être exploitées dans le modèle à vérifier. Le modèle est ainsi constitué d'un ensemble

de solutions, chacune associée à un ensemble de propriétés axiomatiques dont certaines permettent de contrôler l'intégration. Cette couverture permet d'isoler plus précisément la solution qui porte le problème, et de focaliser l'effort de diagnostic. Enfin, nous ne l'avons pas mentionné, mais il ne faut pas oublier que les patrons constituent une documentation quant au fonctionnement du système, aidant ainsi à la compréhension du problème.

## 6.5 Intégration d'un Component case

### 6.5.1 Capture de la solution précédente

Supposons que l'application du *pattern case Auth* a produit une solution jugée satisfaisante et stable pour sécuriser le $Plc_1$. Celle-ci peut être stockée dans la base de connaissances sous la forme d'un *component case*. Pour capturer ce *component case*, Philippe doit définir les contrats du composant qui lui permettent d'échanger avec l'environnement dans lequel il est intégré. Le contrat $C_1$ fournit le service de détection d'un problème sur la signature du message (elle correspond au mécanisme *SAP*). Le contrat $C_2$ fournit le service d'évaluation de l'autorisation de la requête comprenant un client, une ressource et une opération. Les contrats $C_3$ et $C_4$ déterminent ou non l'autorisation d'accès du client sur une ressource pour l'opération souhaitée. Philippe nomme ce *component case SEC_ACCESS*. *SEC_ACCESS* constitue maintenant une sorte de COTS (Commercial off-the-shelf) facile à réutiliser.

### 6.5.2 Formalisation du problème

L'architecture SCADA évolue une nouvelle fois et un contrôleur local $Plc_3$ est ajouté. Celui-ci possède le même comportement que $Plc_1$, et doit être sécurisé de la même façon que $Plc_1$. Pour résoudre ce problème, Philippe peut réutiliser le *component case SEC_ACCESS*.

### 6.5.3 Conception de la solution

*SEC_ACCESS* est un composant défini par quatre *contrats* qui jouent le rôle de points de contacts avec la solution actuelle. Cette étape peut nécessiter du refactoring dans la solution existante. La figure 6.8 illustre la solution obtenue.

### 6.5.4 Vérification et diagnostic

La solution *composant case* est relative au *problem case Auth* et doit donc en satisfaire les propriétés. Qu'advient-t-il de celles-ci lors de la composition ? Si les propriétés de

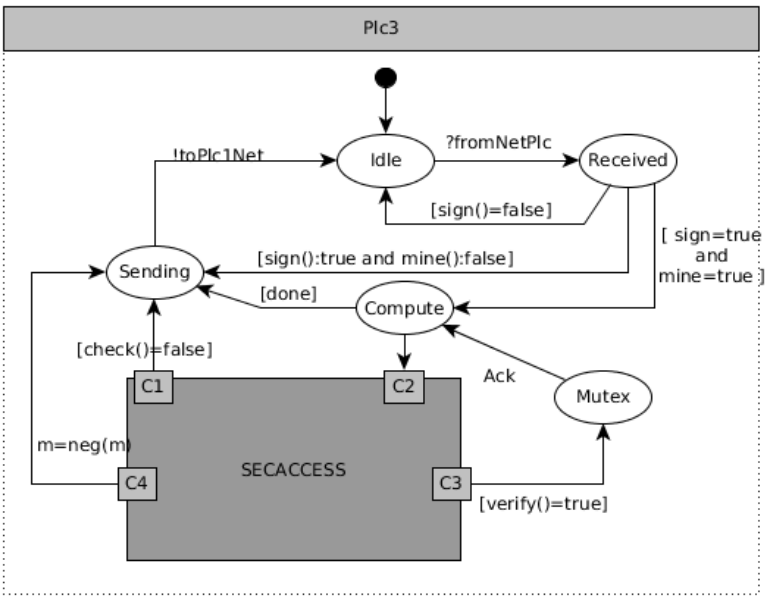


FIGURE 6.8 – Composant SecAccess intégré

sûreté sont vérifiées pour le *component case*, elle le sont aussi lorsque celui-ci est composé avec la solution actuelle. Mais ce n'est pas le cas des propriétés de vivacité qui devront être revérifiées. Supposons que la propriété $PRT_2$ soit violée. Car c'est une propriété de sûreté, le composant n'étant pas en faute, c'est l'intégration de celui-ci qui pose problème.

### 6.5.5 Conclusion

Dans ce scénario, le problème *SEC_ACCESS* offre une solution sous la forme d'un composant concret réutilisable, comme l'illustre la figure 6.9. Ce composant respecte les contraintes du problème et la structure abstraite donnée par celui-ci. Le *component case* est la forme de *problem case* qui demande le moins d'adaptation de la solution. Son intégration repose uniquement sur l'utilisation de *contrats* à remplir vis-à vis du composant. Les propriétés à vérifier sont réduites aux propriétés d'intégration et de vivacité, réduisant ainsi l'effort de diagnostic par rapport aux autres types de solutions.

## 6.6 Ingénierie du domaine

### 6.6.1 Diagnostic et domaine

Le type de solution influence la complexité de conception de la solution. Un *sample case* demande beaucoup d'efforts car il ne peut pas être réutilisé tel quel, mais son pro-

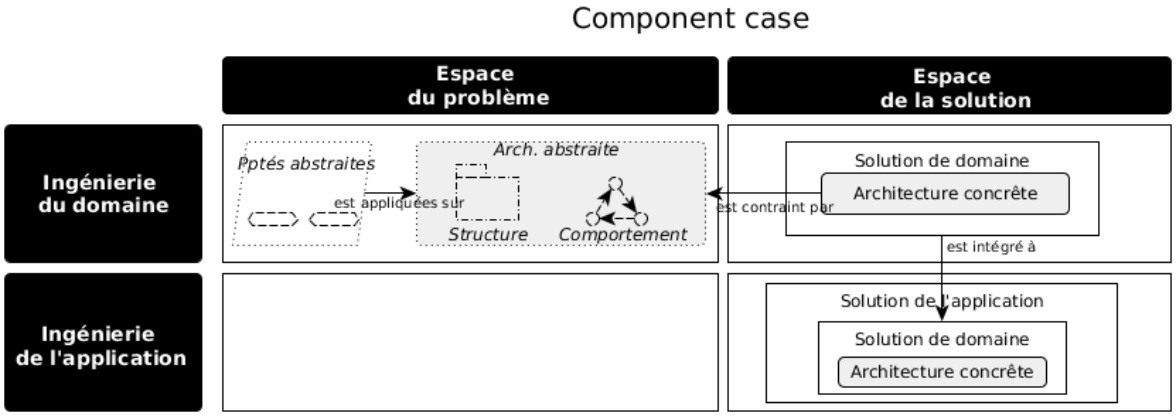


FIGURE 6.9 – Les cadrants et l'intégration du component case

blème associé permet de guider la conception de la solution de l'application grâce à une architecture abstraite. Dans le cas d'un *pattern case*, la solution sera moins coûteuse à produire puisqu'une solution est partiellement fournie. Il faut néanmoins l'adapter au contexte. Enfin le *composant case* peut être réutilisé (presque) tel quel et présente la solution la plus simple.

Le type de solution influence également la complexité du diagnostic de la solution. Sans solution connue, le diagnostic repose sur les connaissances liées au model checking. Utiliser une solution connue permet de préciser le diagnostic grâce à l'apport de nouvelles propriétés. Le *sample case* est celui-ci qui en ramène le moins, aucune propriété de solutions de domaine, simplement des propriétés de problème de domaine. Le *pattern case* ramène des propriétés qui permettent de focaliser le diagnostic, au même titre que le *component case*. Des deux, le *component case* est celui qui facilite le plus le diagnostic, car ses propriétés de sûreté internes ne sont pas à revérifier. La constitution d'un domaine (voir figure 6.10) permet d'accumuler des solutions offrant des propriétés permettant à la fois de préciser le diagnostic grâce aux problèmes, car ceux-ci fournissent la nature (au sens ontologique) de ce que l'on cherche, et de focaliser le diagnostic, grâce aux solutions qu'elles soient sous la forme de patrons ou bien de composants. Mais utiliser le domaine peut avoir un coût, car souvent il faut refactorer le système.

### 6.6.2 Dualité problème solution à résoudre

Pour faire collaborer les deux ingénieries, il existe deux cas de figures, soit l'application est déjà conçue, soit elle est en cours de conception.

Dans le cas où l'application est déjà conçue, il peut être utile de vouloir y retrouver les problèmes ou solutions de domaine, et réutiliser des propriétés pour préciser ou focaliser

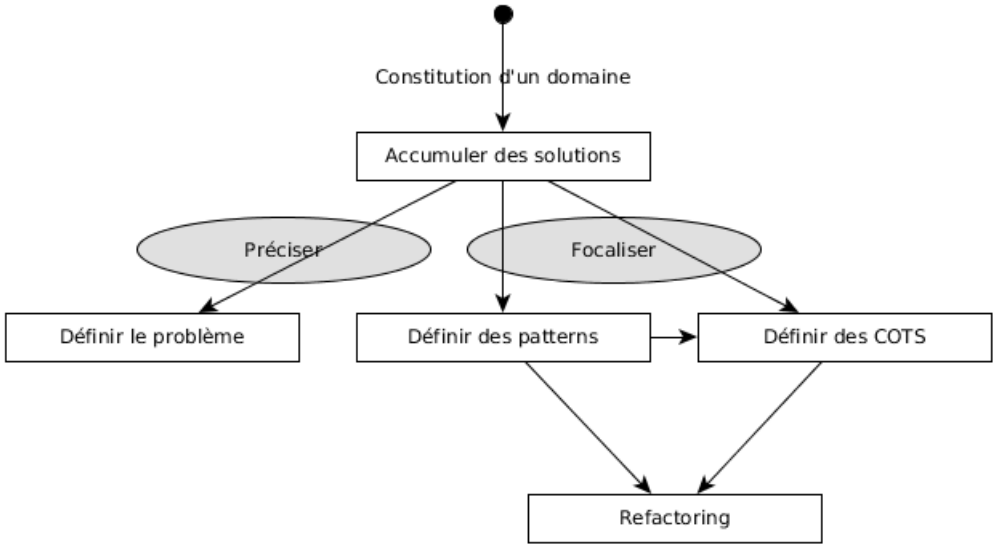


FIGURE 6.10 – Constitution d'un domaine

le diagnostic. Retrouver ces éléments dans la base de connaissance peut se faire suivant différentes approches, comme le CBR. En réalité, si l'application est déjà conçue, il est illusoire de penser que l'on va retrouver toutes ces informations. Même si de temps en temps il est possible de détecter la présence de problèmes de domaine, il faut réussir à refactorer l'application pour la faire correspondre aux problèmes, ou aux solutions existantes, lesquelles ramènent à leur tour des propriétés de problème à laquelle la solution répond. Ce cas de figure est illustré à gauche de la figure 6.11.

Dans le cas où l'on n'a pas conçu l'application, la conception est incrémentale. Il faut à chaque incrément réussir à capturer l'ensemble des solutions choisies dans une base de connaissance, les tracer au fil des améliorations ou abandons, afin de tracer l'état général des propriétés à un instant t qui permettent de prédire le comportement attendu de l'application. Ce cas de figure est illustré à droite de la figure 6.11.

### 6.6.3 Conclusion

L'application de la méthode, présentée dans ce chapitre, contraint à manipuler de nombreux éléments. Il faut les capturer, les retrouver ou encore les tracer. La question qui se pose est où et sous quelle forme, capturer, retrouver, garder ces informations ? De plus, ces informations sont capitales car elles permettent des interactions permettant, par exemple, de faciliter le diagnostic. Comment alors mettre en œuvre ces interactions ? Dans le chapitre suivant nous proposons un système permettant d'organiser ces informations, et les interactions qu'elles supportent, appelé le VOS (Verification Organizing System).

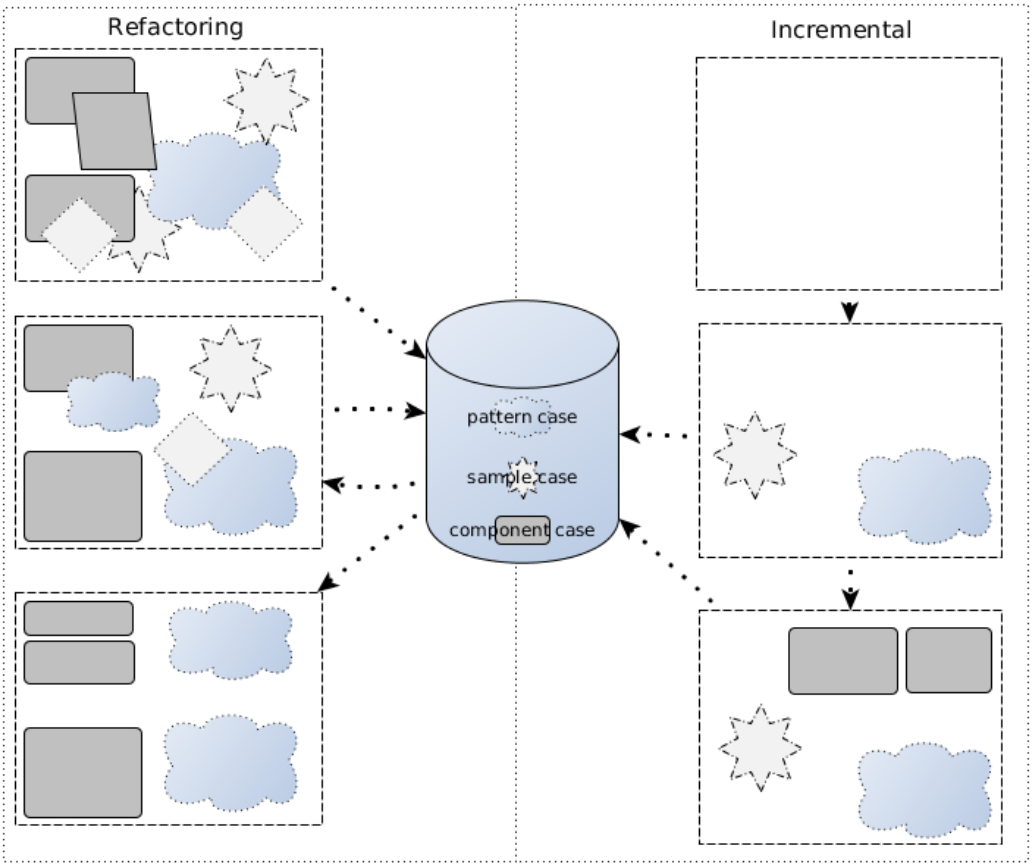


FIGURE 6.11 – Refactoring versus incremental

Chapitre 7

# Organiser les connaissances et les interactions

## 7.1 Introduction

### 7.1.1 Une profusion d'informations

#### Informations hétérogènes

Un processus de conception et vérification par model checking produit une profusion d'informations hétérogènes (spécifications, modèles structurels ou comportementaux, propriétés, traces ...), et ce à différents niveaux d'abstractions et de détails, comme argumenté par [RB03]. Dans le chapitre 4, nous avons vu par exemple qu'une des techniques de diagnostic consistait à écrire un ensemble de propriétés liées à des connaissances en model checking. Cette technique produisait 30 propriétés pour un système très simple. Aussi pour faciliter leur mise en œuvre et limiter les erreurs, ces propriétés doivent être générées (conformément à des normes de nommage) ou instanciées à partir d'un modèle ou d'un patron défini par un expert, et dont les concepts doivent être formalisés. Ces modèles peuvent être stockés dans une bibliothèque.

#### Relations

Des relations de tout ordre existent entre ces informations. Dans le chapitre 5, par exemple, nous avons vu que pour résoudre le fossé sémantique il fallait rapprocher deux types de connaissances, les connaissances du niveau domaine et les connaissances liées au model checking. Pour cela il s'agit de définir des corrélations entre ces éléments. Pour pouvoir outiller ces corrélations, celles-ci doivent être capturées et formalisées.

#### Temporalité

Ces informations et leurs relations évoluent au cours du temps. Elles sont modifiées, enrichies ou abandonnées tout au long du processus de conception et de vérification. Dans le chapitre 5, nous avons vu par exemple qu'un processus de résolution de problèmes

implique une multitude de cycles de conception-vérification. Lors de ces cycles, des solutions sont produites, conservées, améliorées ou bien abandonnées. Le système devient au fil du temps un enchevêtrement de solutions (associées à leurs problèmes) garantissant des propriétés. Si l'on ne conserve pas l'historique de ces solutions, ont perd alors les raisonnements qui nous ont conduits à ces solutions.

### 7.1.2 Besoin de gérer le processus de vérification

#### Constat actuel sur la gestion du processus de vérification

Actuellement on constate que les processus de vérification font face aux problèmes suivants :

- (1) Les connaissances ne sont pas bien gérées, et donc indisponibles.
- (2) Les activités manipulant ces connaissances ne sont pas bien contrôlées, ni enregistrées.
- (3) Il n'y a pas de consensus sur un formalisme de ces connaissances.
- (4) Il n'y a pas d'accord sur une manière de les capturer.

Pour capturer et utiliser les connaissances et les relations au fil du temps, il faut une gestion du processus de vérification qui n'est aujourd'hui pas pratiquée [RB03].

#### Vers un système organisationnel

Baier souligne que, de manière transversale aux autres phases, le processus de vérification doit être planifié, administré et organisé, à travers une activité appelée *l'organisation de la vérification* [BK08]. Un système supportant l'organisation de la vérification serait aussi en mesure de capitaliser les expériences passées, et d'en permettre la réutilisation, notamment pour contribuer au processus de diagnostic. L'enjeu de ce chapitre est de proposer une infrastructure pour organiser ces connaissances et les interactions qu'elles supportent. Pour cela, il faut répondre à différentes questions :

- (1) Où et sous quelle forme capturer ces éléments ?
- (2) Comment les retrouver et quelles sont les interactions possibles ?

#### Système organisationnel

Organiser, c'est créer des capacités en imposant intentionnellement un ordre et une structure [Glu12]. C'est une activité tellement commune que nous le faisons souvent inconsciemment (organiser les jouets dans des boîtes, organiser les vêtements dans les placards...). Ces tâches organisatrices ne sont souvent pas réfléchies ou exprimées. Nous prenons pour acquis les concepts et les méthodes utilisés dans le système d'organisation

avec lequel nous travaillons. Dans [Glu12], Glushko introduit le concept de système organisationnel qu'il définit comme "un ensemble de ressources organisé intentionnellement et les interactions qu'elles supportent". La figure 7.1 décrit un modèle conceptuel d'un tel système découpé en trois parties :

- (1) Les ressources disposées intentionnellement

- (2) Les différentes interactions (différents types de flèches)

- (3) Les entités (humaines ou machines) qui interagissent avec les ressources dans différents contextes.

Une ressource est une chose physique (un fichier de trace) ou non physique (le concept de configuration), ou une information sur cette chose (ce fichier de trace représente un contre-exemple). Une collection est une ressource qui regroupe des ressources sélectionnées dans un but (ce run de vérification vérifie la propriété $PRT_1$ sur le modèle $System_1$). Un arrangement intentionnel met l'accent sur l'organisation explicite ou implicite commis par des personnes (ou processus), se différenciant des disposition naturelles (créés par des processus physiques ou temporels). Une interaction est une action, une fonction, un service ou une capacité qui utilise les ressources d'une collection ou de la collection dans son ensemble.

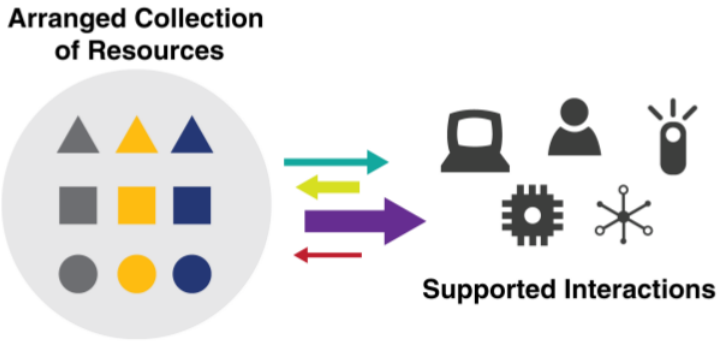


FIGURE 7.1 – Organizing System de Glusko

Il est généralement admis par les architectes et concepteurs de logiciels de séparer le stockage des données de la logique métier utilisant ces données et de l'interface utilisateur (ou les composants d'interactions). Cette architecture est modulaire, et permet à chacun des trois niveaux d'être modifié indépendamment pour répondre aux nouvelles exigences ou utiliser de nouvelles technologies.

Notre proposition est de définir un système organisationnel pour la conception et vérification par model checking axé sur les interactions de diagnostic.

## 7.2 Verification Organizing System

### 7.2.1 Description du système

Nous appelons *VOS* pour Verification Organizing System, le support organisationnel permettant à la fois de gérer les informations relatives aux activités de conception et de vérification, et de servir de base aux outils et méthodes de diagnostic. Le système est composé de 3 niveaux : - un niveau physique où sont stockées les données réelles induites par les processus de conception et de vérification ; - un niveau ontologique évolutif, au sein duquel les données physiques sont réinterprétées et mises à disposition d'interactions ; - et enfin un niveau d'accès où les interactions sont mises à disposition d'outils ou d'humain extérieurs.

Établir un tel système implique des décisions qui sont intrinsèquement liées, mais qu'il est plus facile d'introduire indépendamment. Glusko préconise cinq groupes de questions indépendantes pour décider de la conception : qu'est ce qui est organisé, pourquoi, sous quelle forme, quand et comment ?

#### Qu'est-ce qui est organisé ?

Il faut organiser les connaissances et les interactions. D'un côté nous disposons de trois types de connaissances :

- (1) Les connaissances liées au model checking incluent les modèles de système, les propriétés, et les vérifications

- (2) Les connaissances de domaine incluent les *problem cases*, les *sample*, *pattern* et *component cases*

- (3) Les connaissances de gestion incluent les éléments de la méthode, les cycles de vérification, les activités ou résultats de diagnostic.

D'un autre côté, les interactions sont des blocs élémentaires d'activité, par exemple "explorer la trace", "réduire la trace", "comparer deux traces". Le processus de diagnostic est un ensemble ordonné de ces activités (interactions) s'appuyant sur ces connaissances.

Toutes sont des ressources numériques, mais nous pouvons faire la distinction entre les ressources primaires et les descriptions de ressources à propos des ressources primaires. Tout utilisateur du VOS peut également organiser ses propres activités de vérification dans des collections ou des sous collections de ressources.

#### Pourquoi est-ce organisé ?

Les utilisateurs du VOS sont des ingénieurs en conception et vérification travaillant seuls ou en équipe qui doivent "gérer l'énorme quantité des données au fil du temps et qui est produite au cours de la phase de conception et de vérification" [RB03]. Le VOS

partage et organise des informations qualitatives et quantitatives permettant la création de connaissances et le raisonnement basé sur ces connaissances. Les utilisateurs du VOS partagent leurs connaissances sans être contraints d'épouser un formalisme donné. Les utilisateurs du VOS doivent naviguer efficacement dans cet espace de ressources.

#### Sous quelle forme est-ce organisé ?

Le VOS le plus simple consiste en un système de gestion de configuration (comme un GIT ou un SVN) contrôlant les changements de chaque ressource numérique. À l'opposé, le VOS peut être une ontologie complète où toute relation entre des éléments d'information est soigneusement définie et contrôlée. De notre point de vue, le VOS gère essentiellement des documents (modèles, résultats, traces). Chaque document organise sa structure de connaissances et son contenu, en fonction de son type, et l'ingénieur écrit et lit les informations en fonction de cette structure. La réification de la structure de connaissances sous-jacente est effectuée automatiquement par le VOS.

#### Quand est-il organisé ?

Le VOS est conçu pour assister l'ingénieur dans ses tâches quotidiennes de conception et de vérification. Les ressources sont donc organisées en permanence. Cependant, le VOS doit offrir une fonctionnalité d'ingestion permettant de saisir de nouvelles entrées. Cette fonctionnalité lui permet d'accepter des tentatives de vérification complexes ou des benchmarks et de préparer le contenu pour le stockage et la gestion au sein du VOS. À l'inverse, une fonctionnalité d'accès fournit les services et les fonctions qui aident les utilisateurs à interagir avec les informations stockées dans le VOS.

#### Comment ou par qui, ou par quels processus informatiques, est-il organisé ?

Même si un seul ingénieur en vérification bénéficie de l'utilisation du VOS, le VOS est conçu pour prendre en charge le travail en équipe et partager les connaissances sur les modèles et les tentatives de vérification. Des processus automatisés doivent extraire autant de connaissances que possible de la structure interne des documents et de l'organisation des collections. En tant que travail d'équipe collaboratif, l'organisation est effectuée de manière ascendante et distribuée.

### 7.2.2 Architecture

#### Vue conceptuelle

L'architecture conceptuelle du VOS est présentée sur la figure 7.2.

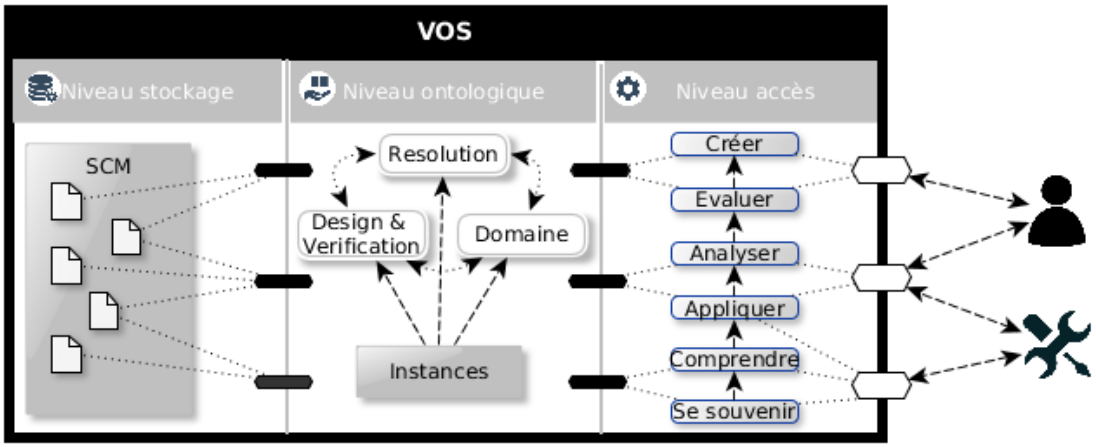


FIGURE 7.2 – Architecture conceptuelle du VOS

**Le niveau de stockage**. Une difficulté pratique liée à la conception et vérification basée sur du model checking réside dans la gestion de toutes les données (générées) au cours des opérations de conception et de vérification. Un enregistrement rigoureux des informations est nécessaire, et pour cela nous proposons d'utiliser un système de gestion de configuration logicielle (SCM) pour contrôler les artefacts versionnés produits lors des phases de conception et de vérification.

**Niveau ontologique**. D'un point de vue conceptuel, la couche ontologique est divisée en deux parties, le réseau sémantique de types et le réseau sémantique d'objets. Le réseau sémantique de types consiste en types sémantiques liés par des types de relations sémantiques, équivalents à un modèle relation-entité ou à un diagramme de classes UML. Il est lui-même constitué de trois parties, les connaissances de résolution, de domaine et de conception et vérification. Le réseau sémantique d'objets quant à lui, contient un nœud pour chaque objet à grain fin plus des nœuds pour les objets composites, chacun étant affecté à un ou plusieurs types sémantiques et lié à d'autres objets par des relations sémantiques. Le niveau ontologique évolue constamment au fil du temps, car de nouveaux types de ressources et de nouvelles ressources peuvent être ajoutés à tout moment. Il est irréaliste de s'attendre à ce que toutes les personnes et organisations développant des systèmes d'application de connaissances utilisent une ontologie commune partagée [HPS04]. Nous devons vivre avec différentes ontologies et il sera nécessaire de réconcilier ces ontologies avec une ontologie supérieure commune. Parce que nous sommes habitués à l'ontologie CIDOC CRM, une structure normalisée (ISO 21127 : 2014) permettant de décrire les concepts et relations implicites et explicites utilisés pour décrire le patrimoine culturel, nous utilisons CIDOC CRM comme une ontologie de haut niveau [1].

1. http ://www.cidoc-crm.or/official_release_cidoc.html

**Le niveau d'accès** L'un des principaux objectifs d'un système d'organisation est l'aide à la conception et à la mise en œuvre d'actions, fonctions ou services qui utilisent les ressources. Dans une architecture classique à 3 niveaux, le niveau de présentation est le niveau dans lequel les utilisateurs interagissent avec une application. Les interactions utilisateur les plus primitives sont les ingestions (importation de nouvelles ressources dans le système d'exploitation), les recherches, la navigation, les annotations ou l'extraction d'informations. Sur cette base d'interactions primitives se trouvent des interactions plus complexes, comme des comparaisons, des isolations, des générations. Ainsi, nous classons les interactions suivant les six catégories cognitives de Bloom [Blo+64], *se souvenir, comprendre, appliquer, analyser, évaluer et créer*. De façon générale, un système organisationnel est également destiné à interagir avec d'autres applications et doit fournir des fonctionnalités d'échange d'informations avec ou sans transformations sémantiques.

### Vue technique

L'architecture technique du VOS est présentée dans la figure 7.3. Il convient de réaliser un système facile d'utilisation, accessible et adaptable. Nos choix techniques se sont portés sur un environnement basé sur la technique de fédération de modèles pour l'adaptabilité, des technologies serveur/client web pour l'accessibilité, et du framework CSS Material Design pour l'ergonomie et la facilité d'utilisation.

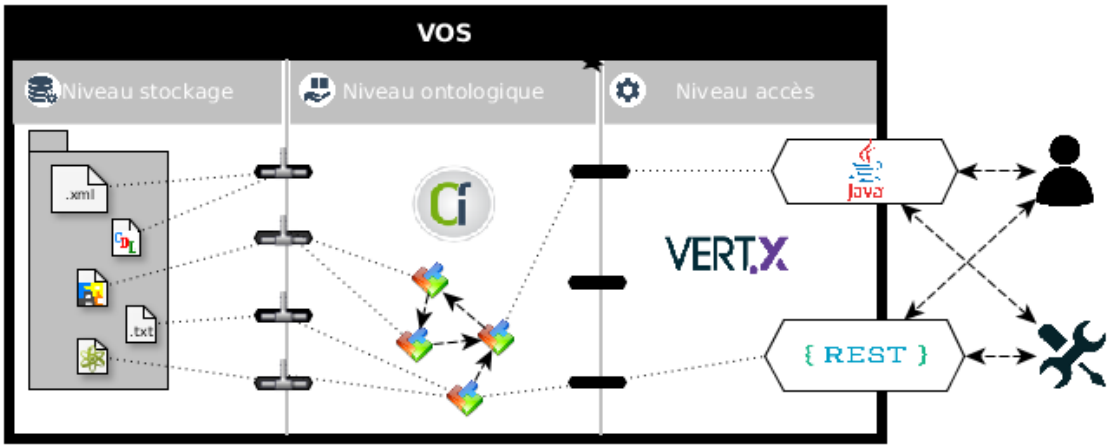


FIGURE 7.3 – Architecture technique du VOS

**Niveau stockage**. Car chaque système et chaque équipe est différente, nous optons pour une infrastructure flexible, supportant l'implémentation adaptateurs techniques. Ainsi, il n'y a pas de limitations sur le type de ressources qui peut être stocké dans le VOS. Nous n'imposons pas non plus d'arrangement de ressources, uniquement un répertoire regroupant les tentatives de vérification, avec une totale liberté pour y organiser

les informations internes. Les objets complexes tels que les ensembles de propriétés ou la décomposition de designs sont gérés de la même manière, avec un répertoire racine et une liberté d'organisation. Afin de faciliter l'intégration de chaque objet au niveau logique, un fichier de description XML stocke des informations sur les objets. La structure du XML (son schéma) est utilisée par les composants logiciels (fournissant des fonctionnalités d'acquisition et d'accès) pour maintenir un réseau ontologique à jour au niveau logique. Pour éviter la construction d'un silo d'information, c'est-à-dire un système de gestion insulaire qui ne peut fonctionner avec d'autres systèmes, fichiers, répertoires, descriptions XML, garantissent un accès indépendant de tout système de gestion. Comme la description XML est inspirée de la structure des modèles du BEEM [Pel07] (benchmark pour la vérification par model checking), la base de cas peut être remplie par ces éléments.

**Niveau ontologique**. Parmi les solutions possibles, nous avons choisi une approche pragmatique, appelée la fédération de modèles [Guy+13] supportée par un outil open source [2].

La fédération de modèles est une approche qui vise à créer un mapping bidirectionnel entre des modèles hétérogènes. Elle se distingue des autres approches d'interopérabilité comme l'intégration (grouper tous les concepts dans un métamodèle unique), ou l'unification (transformations vers un métamodèle pivot). La fédération offre un moyen pour franchir les frontières technologiques entre les modèles et favoriser les échanges dans les deux sens.

Openflexo est un outil permettant de réaliser la fédération. Il offre un moyen pour définir des modèles fédérés grâce à la notion de modèles virtuels (*virtual model*). Un *virtual model* est constitué d'un ensemble de concepts fédérés *flexoconcepts*. Tout *flexoconcept* comporte une partie structurelle qui définit ses caractéristiques propres ainsi que les relations (*roles*) vers d'autres concepts ou ressources provenant de divers espaces techniques, et comporte une partie comportementale supportant la création d'*actions* paramétrables. Il existe de nombreux espaces techniques actuellement supportés (fiacre, cdl, excel, word, powerpoint, owl, emf...). Ces *virtual models* peuvent être instanciés en *virtual model instances*, et les *flexo concept* en *flexo concept instances*.

Notre VOS repose sur cette base technique, comme l'illustre l'exemple de la figure 7.4. Un *virtual model* fédére les éléments appartenant au niveau de stockage aux connaissances exprimées dans l'infrastructure Openflexo grâces aux connecteurs techniques. Ces connaissances (model checking, domaine, résolution) sont exprimées par des *flexoconcepts*, par exemple, le concept de *SampleCase* est porté par un *flexoconcept* appartenant au *virtual model* de *domaine*.

Au niveau des instances, une instance de *samplecase* (aussi appelée *flexoconceptinstance*) est liée à plusieurs ressources (cdl, fiacre...) exprimant le *samplecase* dans le niveau

2. http ://research.openflexo.org

de stockage.

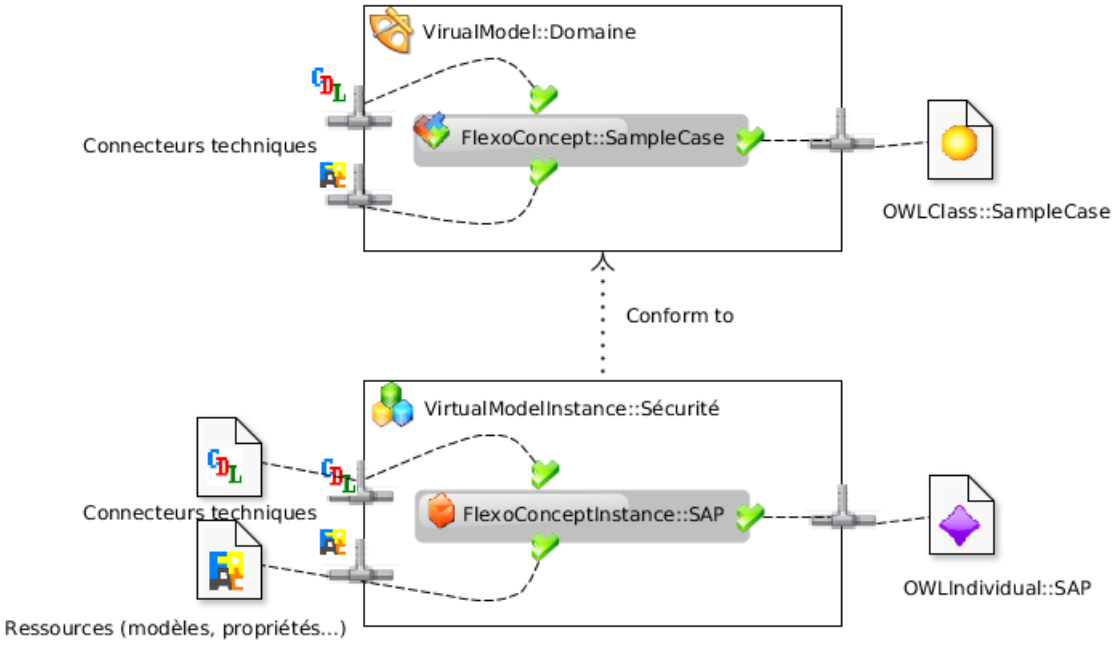


FIGURE 7.4 – Principe de la fédération pour le VOS

Un connecteur technique permet de fédérer l'ensemble à une ontologie, permettant ainsi de l'inférence et du partage de connaissances. Ainsi, chaque *flexoconcept* est lié à une classe ontologique et chaque *flexoconceptinstance* est associé à un *individual* via le connecteur technique OWL. Il existe donc un isomorphisme entre un regroupement d'objets physiques du niveau de stockage et les objets sémantiques de la couche ontologique.

**Niveau d'accès**. L'infrastructure Openflexo fournit une couche d'accès de différentes manières. D'abord un atelier de modélisation flexible et modulable peut être utilisé pour accéder aux diverses informations, mais aussi pour déclencher les comportements associés aux *flexoconcepts*. Pour permettre aux non initiés à Openflexo d'accéder aux informations, un serveur HTTP Vert.x[3] a été développé. Il propose un ensemble de services *REST*, utilisés par un front end de type single page application réalisé en Typescript/CSS(MDL)/HTML, ergonomique et plus adapté aux humains.

## 7.3 Illustrations

Le VOS organise donc les connaissances et les interactions qu'elles supportent. Dans cette section nous présentons plus précisément la structure des connaissances au moyen

3. https ://vertx.io/

d'exemples, ainsi que les interactions supportées et les moyens mis en œuvre pour la capture des connaissances.

La structure des connaissances est décrite par des classes et des propriétés (au sens ontologique). Cette structure est représentée en haut de chaque illustration par des rectangles gris (les classes) et des flèches (les propriétés). Pour simplifier les illustrations, nous représentons les sous-classes soit avec une flèche blanche pointillée, soit en positionnant une sous-classe à l'intérieur de sa superclasse. Conformément à la structure définie par ces classes, un ensemble de connaissances est instancié sous la forme d'un réseau d'individus. Chaque individu est représenté par un ovale au centre de la figure, et peut être lié à d'autres individus (flèche noire entre individus). Le lien d'instanciation entre un individu et sa classe est symbolisé par la flèche grise pointillée. Un ensemble d'interactions supportées par ces connaissances est présenté en bas de chaque figure, ces interactions sont extraites des chapitres précédents.

### 7.3.1 Connaissances liées au model checking

L'illustration 7.5 ci-dessous est un extrait de l'ontologie du point de vue des connaissances liées au model checking. En haut figurent les différentes classes qui ont été présentées au chapitre 5, au centre quelques individus et en bas des interactions propres à ces individus. Plus précisément il s'agit d'une vue sur l'application en bas à gauche, et une trace en bas à droite.

#### Structure

Les éléments techniques (*technical elements*) sont partagés en trois sous-ensembles, les éléments de spécification (*specification element*), les éléments de conception (*design element*) et les éléments de vérification (*verification elements*). Les éléments de spécification sont essentiellement des propriétés formelles (*formal property*), par exemple la propriété $PNotPlc_1Plc_2Access$. Chaque propriété est associée à une formule logique (*formula*), ici $PNotPlc_1Plc_2Access$ est associée à la formule $\Box\neg(plc_1@Access \wedge plc_2@Access)$. Une formule logique est un agrégat de propositions atomiques *(atomic proposition*), comme par exemple $plc_1@Access$, qui sont liées au moyen d'opérateurs (*operator*) par exemple $\wedge$. Une proposition atomique est liée à un ensemble d'éléments du modèle (*model element*), comme ici, la proposition $plc_1@Access$ est liée à l'état *Access* du processus $plc_1$. Le modèle (*model*), par exemple *Scada*, contient un ensemble de processus (*process*), ici $plc_1$ et $plc_2$, ou des variables (*variables*) partagées entre processus, comme $flag(boolean, boolean)$. Tout processus contient des variables locales, des états (*state*) et des transitions (*transition*), comme ici les états *Access*.

Une tentative de vérification (*run*) est réalisée à partir d'un modèle et de spécifications.

Chacune des spécifications, exprimée par une propriété, est vérifiée dans l'élément *propertyverfication*. Ici nous avons une vérification de propriété, celle de $PNotPlc_1Plc_2Access$. Une vérification de propriété nous fournit des résultats concernant l'évaluation de la propriété (sauf si l'exploration n'est pas possible). Un espace d'état ($LTS$) est produit ainsi qu'un contre-exemple, ici la trace $T12345A$. Une trace est un ensemble de configurations ordonné (*configuration*) possédant des informations (*configurationElement*) à propos d'éléments de modèles ou de spécifications. Par exemple nous avons ici deux configurations *C45Z76* et *C458A6*, chacune disposant d'informations sur l'état du *flag* et l'état courant des processus.

### Présentations

Ces connaissances et leurs relations sont complexes, mais nous sont utiles pour définir différentes interactions facilitant la construction de la solution. En bas de l'illustration, ces connaissances liées au model checking ont permis de produire différentes vues déjà présentées dans le mémoire au chapitre 5.5.1. La première vue est un affichage synthétique d'un sample case. Celle-ci permet de naviguer rapidement dans les éléments pertinents d'une application et des tentatives de vérifications. La seconde vue est la trace présentée au chapitre 4.3.2, elle met en évidence les éléments de configurations modifiés entre deux configurations d'une trace. D'autres vues peuvent être produites, comme des vues matricielles permettant de corréler les propriétés par rapport aux processus ou aux scénarios.

### Ingestions

Une des difficultés de ce type de système est la capture des informations. Il est bien sûr possible de les expliciter manuellement à travers un formulaire d'ingestion, qui est fourni par le VOS, mais il est moins laborieux d'automatiser leur acquisition. Le VOS facilite cette capacité d'ingestion de deux manières. D'une part il propose un ensemble de connecteurs techniques (incluant des parseurs) réifiant les ressources physiques (par exemple un fichier Fiacre) dans la structure du VOS de façon automatique. S'ils ne satisfont pas à l'utilisateur, le VOS offre la capacité d'ajouter de nouveaux connecteurs techniques. D'autre part, il joue le rôle d'une interface de commande en proposant des interactions. Par exemple, il est possible de déclencher l'exploration exhaustive avec OBP à partir d'un ensemble de propriétés (extraites directement d'un fichier cdl via le connecteur technique cdl), d'un modèle (connecteur technique fiacre), et d'une configuration du model checker. Après analyse, les résultats sont automatiquement rangés : un fichier de trace devient un individu trace ; le fichier sera parsé et des configurations crées pour cet individu trace.

Avec seulement les connaissances sur le SAE, ces vues ne permettent pas de construire

des vues de plus haut niveau. Il faut pour ça corréler ces connaissances avec d'autres connaissances.

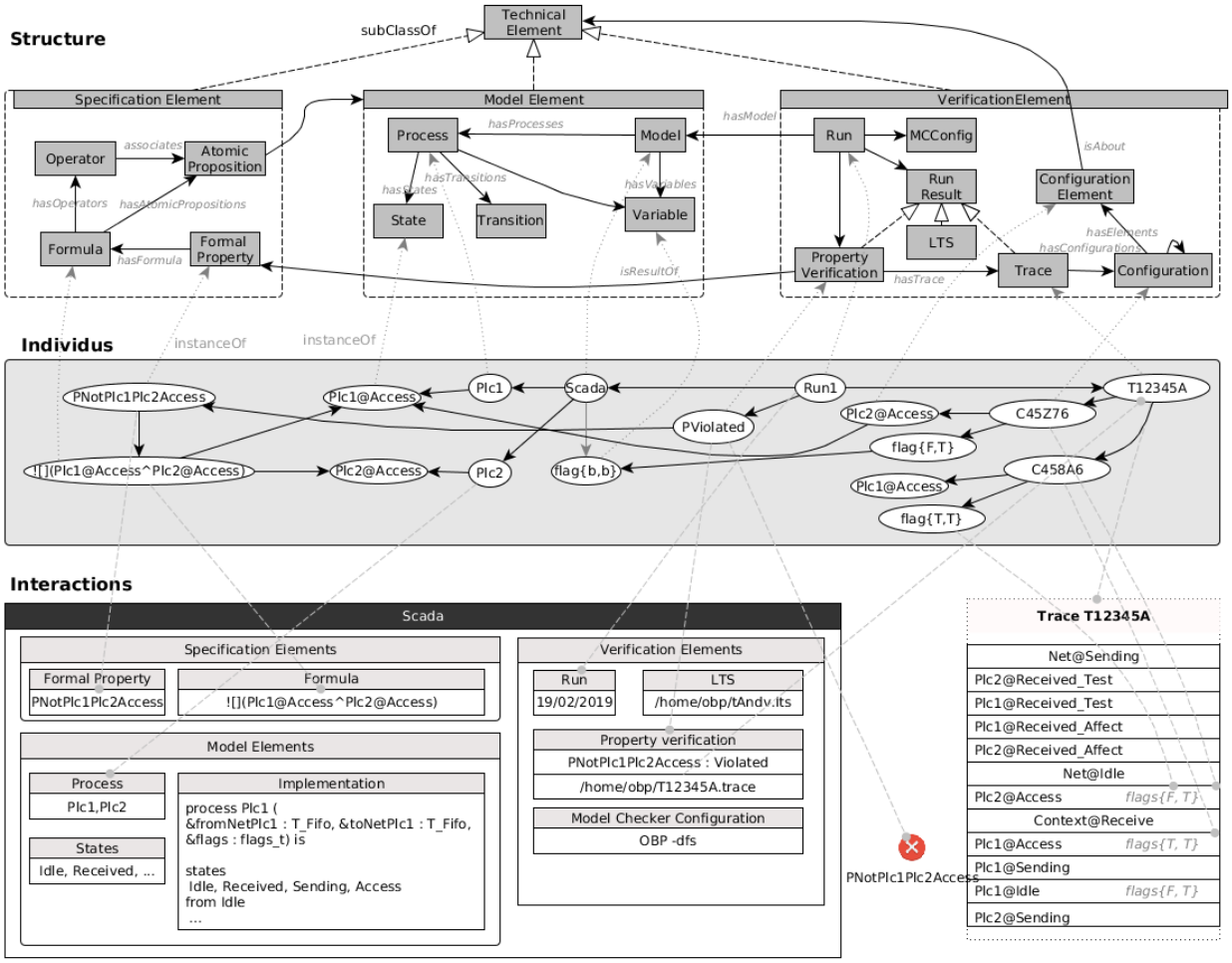


FIGURE 7.5 – Connaissances liées au model checking

### 7.3.2 Utilisation du domaine pour le diagnostic

#### Structure

Dans le cas précèdent le domaine n'est pas capturé dans le VOS. Dans l'illustration 7.6, on complète les connaissances précédentes avec des connaissances sur les problèmes du domaine. Dans la partie du haut, la structure des éléments du SAE a déjà été capturée (*design elements*, *specification elements* et *verification elements*). Un problème du domaine (*problem case*), par exemple le problème de l'exclusion mutuelle *PMutex*, est caractérisé

par un ensemble d'énoncés (*problem statement*) composés de propriétés (*abstract property*) ou d'architectures abstraites (*abstract architecture*) sur lesquelles vont s'appliquer les propriétés. Une propriété abstraite, comme ($PMutexPriPrj$), regroupe un ensemble de propositions atomiques abstraites (*abstract atomic proposition*) comme $Pr_iCS$, à implémenter pour la solution choisie, comme par exemple $Plc_1@Access$. Une architecture abstraite est composée d'éléments structurels (*abstract structure*) comme par exemple $Pr_i$, ou comportementaux (*abstract behavior*) comme $Pr_iCS$. De même que pour les propriétés abstraites, ils doivent correspondre à des éléments concrets dans le modèle. Ainsi, $Pr_i$ est concrétisé par $Plc_1$.

#### Présentations

A partir des connaissances de domaine corrélées aux connaissances du SAE, par exemple il existe un chemin entre la configuration $C45Z76$ et le concept de domaine $Pr_jCS$, directement lié à $Plc_2$, il est possible de générer le point de vue de la trace par rapport au domaine comme le montre la figure en bas de l'illustration, extraite du chapitre 5.2.3. D'autres interactions sont possibles comme des visualisations propres au domaine, ou bien générer des propriétés propres au domaine.

#### Ingestions

Les connaissances de domaine sont abstraites. Elles sont utilisées dans divers contextes, en associant les éléments de problème abstraits aux éléments concrets de la solution construite. Il devient alors possible de générer les vues de domaine. Comme dans le cas précédent, les liens peuvent être définis à la main via des formulaires, ou bien partiellement automatisés par des interactions du VOS (sélection du problème de domaine, association à des éléments de solution via un système de nommage par exemple).

### 7.3.3 Application d'un pattern case

Dans cette nouvelle illustration nous allons présenter la réutilisation d'une solution au *problem case* définit précédemment.

#### Structure

Une solution à ce problème est illustrée sur la figure 7.7. Nous appelons les solutions à un problème du domaine les *solution cases*, par exemple $Peterson$ est une solution au problème de l'exclusion mutuelle. Une solution vise à résoudre un *problem case* grâce à un ensemble d'éléments de solutions (*solution statements*). Un *solution statement* concrétise une partie de problème. Par exemple, le processus $Pr_1$ est une concrétisation de l'élément $Pr_i$

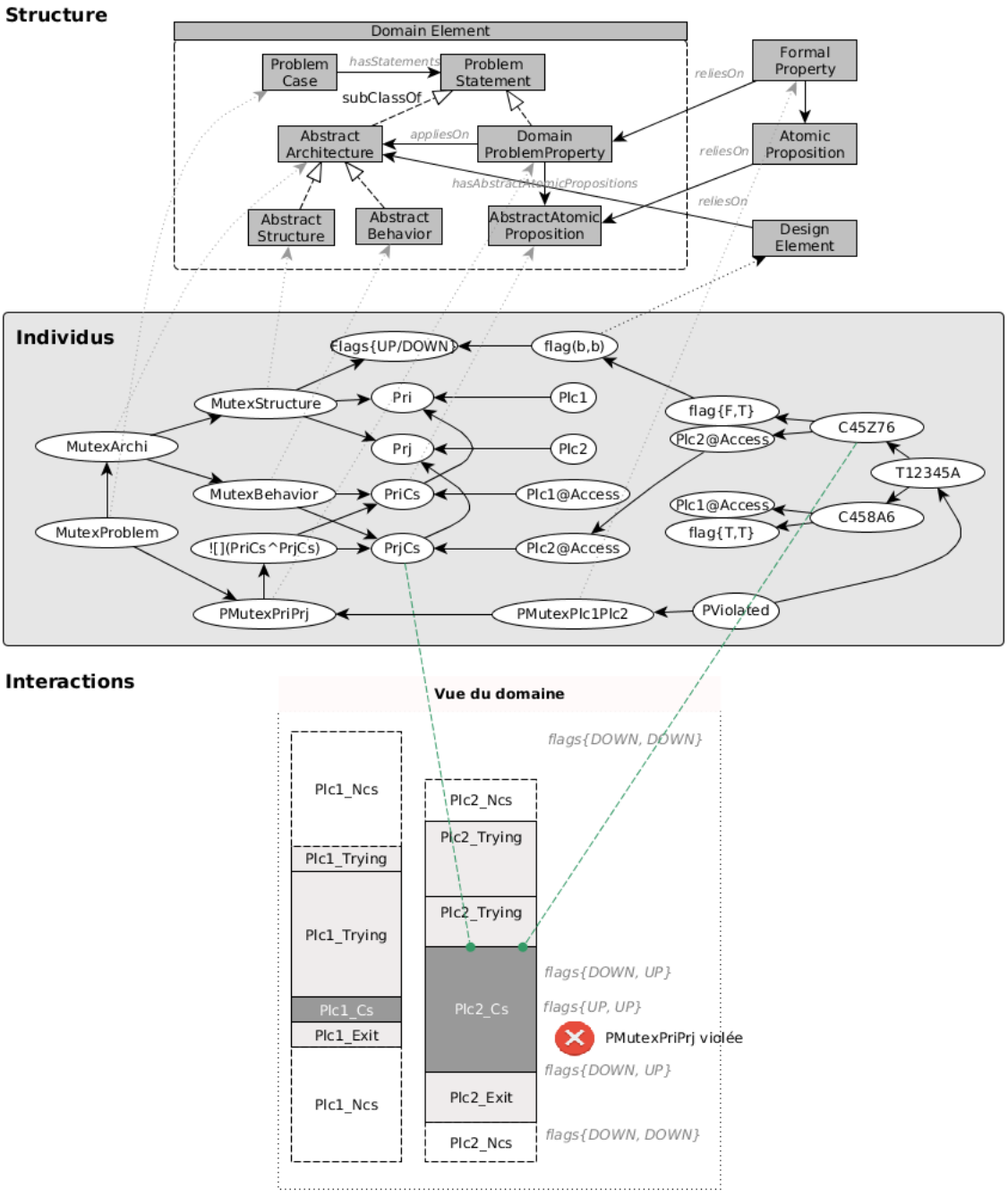


FIGURE 7.6 – Connaissances de problèmes domaine

de l'architecture abstraite du problème de l'exclusion mutuelle. De même, une propriété de solution de domaine (*domain solution property*) concrétise une propriété de problème de domaine. Par exemple $\Box\neg(Pr_1Cs \wedge Pr_2Cs)$ est la concrétisation de la propriété de problème du domaine $\Box\neg(Pr_iCs \wedge Pr_jCs)$. Il existe différentes solutions, le *sample case* est une solution capturée dans sa forme brute, le *component case* est réutilisable dans de nouveaux contextes par le biais de contrats à réaliser avec les éléments de l'application, et enfin le *pattern case* s'utilise par le biais d'un ensemble de règles de transformations (*transformation rule*). Une *transformation rule* prend en arguments un ensemble d'éléments de l'application, et un ensemble d'éléments de la solution du domaine, autrement dit $transfo(designElements, domainElements) \rightarrow solutionElements$. Ces transformations peuvent être des copies (*copy*), dans ce cas elles rajoutent du contenu dans l'application, ou bien des adaptations plus complexes (*adaptation*), lesquelles nécessiteront l'usage d'un langage de transformation (le VOS fournit un langage par l'intermédiaire d'Openflexo). Pour appliquer la solution Peterson à notre application il faut exprimer des règles. La règle $t(Pr_1, Plc)$ adapte un processus $Pr_1$ en un processus $Plc$, il s'agit en clair d'un changement de nom. La règle $t(Wait, Receive)$ adapte l'opération de *trying* à l'état *receive* en la sectionnant en deux nouveaux états $receive\_affect$ et $receive\_test$. La règle $t(flag)$ ajoute la déclaration d'une variable partagée (un drapeau) à l'application.

Pour pouvoir définir ces règles, il faut avoir en tête l'architecture de l'application. Supposons que l'on ait hérité d'une situation antérieure, schématisée par $C_i$ dans la figure 7.8. Celle-ci est une solution $S_i$ répondant aux propriétés $P_i$. À l'itération suivante, nous souhaitons appliquer un *pattern case*. Pour l'appliquer, toute la solution précédente ($C_i$) va jouer le rôle de problème ($P_i + 1$). Pour appliquer un patron il faut s'appuyer sur une architecture, celle-ci vient donc de la solution produite à l'itération précédente.

### Présentations

Dans le cas d'un *pattern case*, la solution de l'application peut être générée à partir de règles entre les éléments de l'application définis lors de l'itération précédente et les éléments du domaine. Cette génération peut se faire pour l'architecture, dans ce cas le principal bénéfice sera un gain de temps ainsi qu'une diminution des risques d'erreurs lors de la conception de la solution. Elle peut aussi se faire sur les propriétés, dans ce cas les propriétés générées peuvent aider à isoler le problème lors de la phase de diagnostic. Le bas de la figure 7.8 montre à gauche le code de l'application et à droite le code de la solution produite. En vert figurent des éléments de solutions copiés et en jaune des éléments de solutions générés grâce au patron. Cette vue permet d'isoler les informations liées au domaine, des informations spécifiques à l'application.

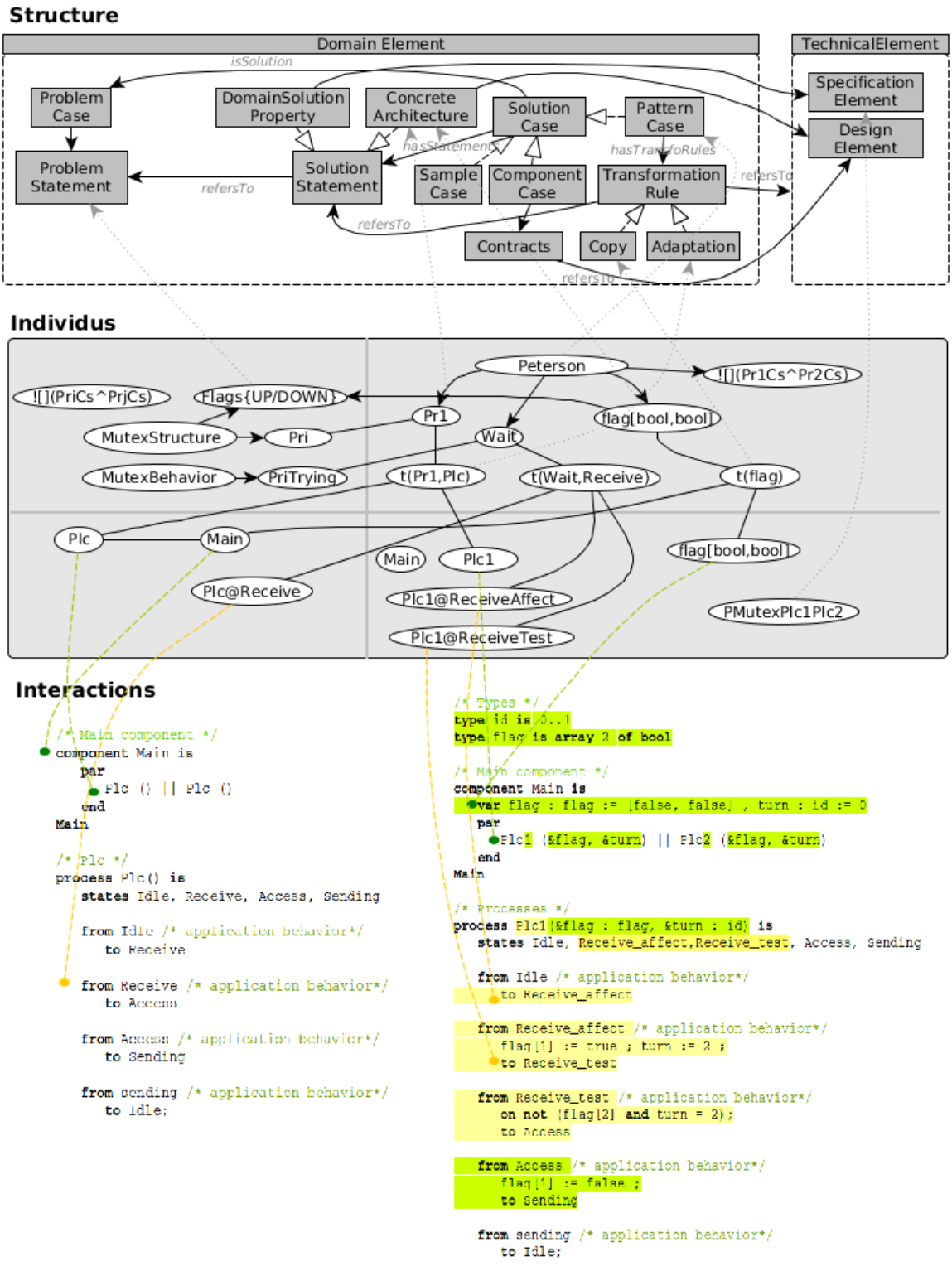


FIGURE 7.7 – Connaissances de solution domaine

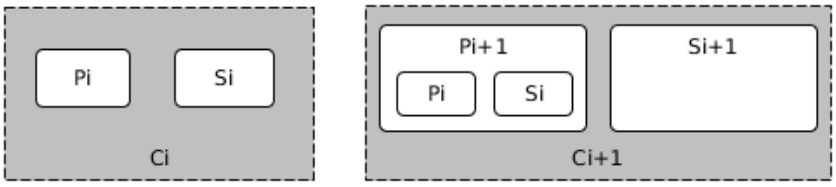


FIGURE 7.8 – Application du pattern case, contexte

#### Ingestion

Disposer de morceaux de solutions réutilisables ou bien de règles de génération permet de progresser plus rapidement dans la solution, ainsi que d'éviter de produire des liens inutiles dans le VOS. Prenons un système Scada avec une centaine de PLC à vérifier, tous redondants. Les instancier tous dans le VOS va nécessiter de créer beaucoup d'individus et de liens. S'ils sont construits sur la base d'un même modèle abstrait, il est possible de générer directement ces individus ou liens au moment où ils sont nécessaires.

### 7.3.4 Construction d'une solution et connaissances de gestion

#### Structure

L'illustration 7.9 présente les connaissances de gestion et un exemple d'interaction. Dans les connaissances de gestion nous avons regroupé les processus cognitifs (*reasoningProcess*) qui sont une succession d'activités cognitives plus simple (*cognitiveActivity*), comme la recherche et la récupération de problèmes ou solutions de domaine (*retrieve*), la conception de la solution de l'application (*design*), la vérification de la solution (*verify*) ou encore la capture d'une solution ou problème (*retain*). Une activité cognitive adresse une révision particulière de la solution (*revision*). Cette révision est composée d'éléments techniques et d'éléments de domaine. Par exemple la révision $revision_1$ s'applique sur la solution $solutionflag_3$, le problème $MutexProblem$ et au problème de l'application *Scada*. Une révision peut être construite à partir d'une révision précédente. Ce mécanisme est laissé volontairement simple, le rôle du VOS n'étant pas de suppléer au système de gestion de version gérant déjà les ressources physiques ingérées par le VOS.

#### Présentation

À partir de cette structure, il est possible de générer la vue du chapitre 5.2.7. Celle-ci met en avant les successions d'essais de construction de l'application. L'intérêt est double, faciliter la maintenance de la solution, imaginons par exemple que la solution soit reprise par un ingénieur n'ayant pas la connaissance de l'historique de sa construction, et

faciliter le diagnostic, car au cours de la construction et des multiples essais, l'ingénieur a sélectionné ou désélectionné des propriétés. Lors de la vérification l'ingénieur a besoin des propriétés qui caractérisent le mieux la solution.

La capture dans le temps des solutions construites et leurs évaluations permet des analyses basées sur l'historique (statistiques, probabilités, algorithmes de machine learning ...). Il est possible de proposer au reuse les solutions les plus pertinentes à l'ingénieur (en prenant en compte son domaine ou ses habitudes).

Enfin, en capturant la trace des activités cognitives réalisées par l'ingénieur, des outils pourraient permettre de mieux comprendre le fonctionnement du processus mental de résolution de problèmes. Se souvenir du processus passé permet de ne pas refaire les mêmes erreurs ou ne pas repartir de zéro.

#### Ingestion

Certaines connaissances de gestion peuvent être capturées automatiquement grâce à des interactions du VOS, comme lancer la vérification, sélectionner des problèmes ou des solutions pertinentes etc...

## 7.4 Conclusion

Le processus de conception et vérification par model checking produit une multitude d'informations corrélées et hétérogènes. Pour permettre leur organisation et fournir des interactions, il faut les organiser au sein d'une structure. Le VOS est une spécification simple et adaptable d'une structure répondant à ce besoin. Le VOS permet de réaliser les différentes interactions présentées dans la thèse (réutilisation de connaissances, vues...). Chaque organisation doit l'adapter à ses contraintes et usages métiers, et proposer ses propres interactions.

Pour qu'une telle structure soit efficace, elle doit contenir des connaissances, et fournir des interactions. Dans le chapitre suivant, nous remplissons la structure avec des connaissances, et spécifions différentes interactions.

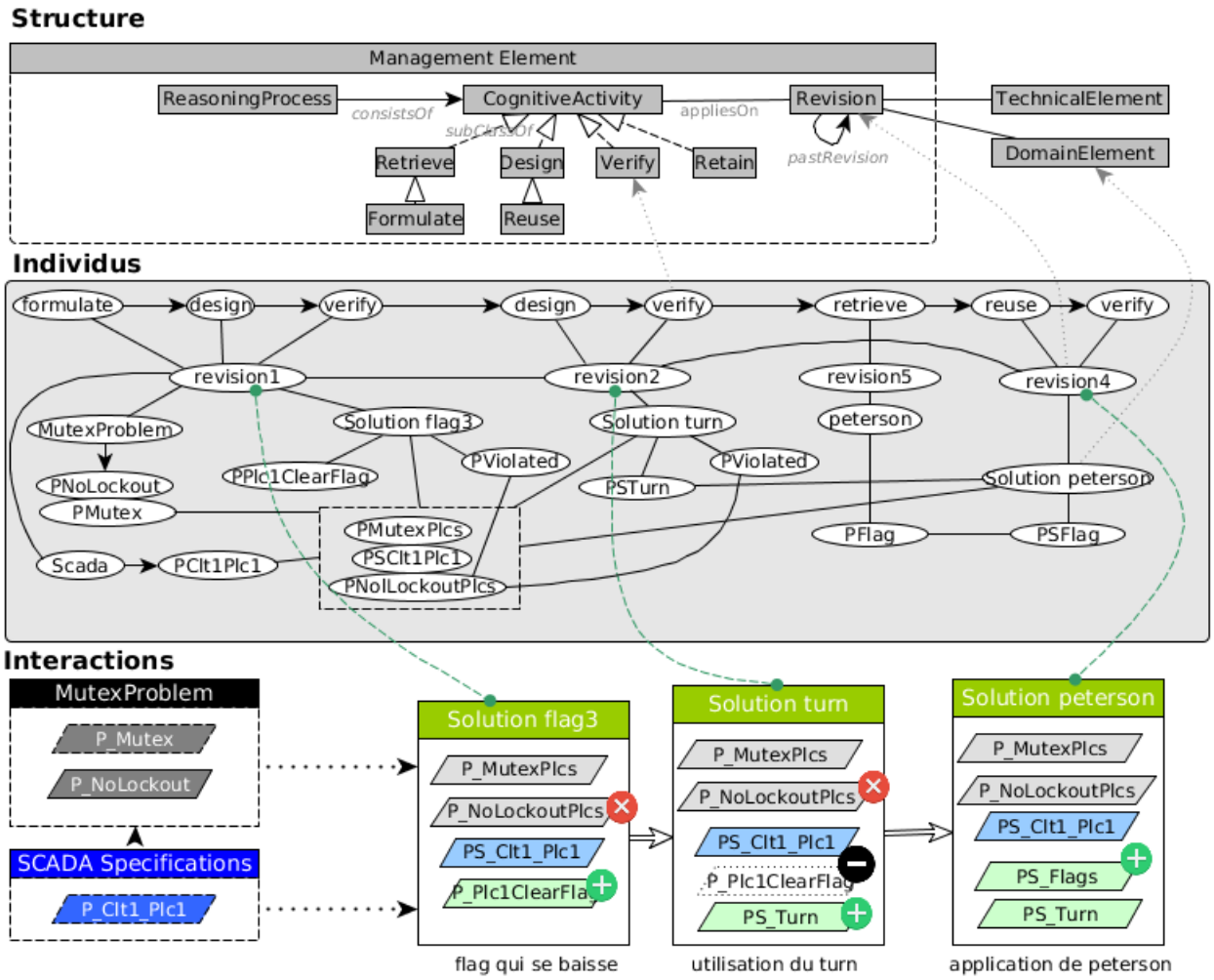


FIGURE 7.9 – Connaissances de gestion

Chapitre 8

# Exemple d'utilisation du VOS

## 8.1 Ingestion de connaissances

### 8.1.1 Beem

#### Présentation du benchmark

Pour constituer un ensemble de problèmes et de solutions réutilisables, il est nécessaire de s'appuyer sur l'ingénierie du domaine, une pratique non répandue dans le domaine du model checking. Heureusement, il existe dans cette communauté des benchmarks dont ont peut partir pour disposer du domaine. Le BEEM [Pel07] est un benchmark qui est utilisé pour évaluer les outils et les algorithmes des model checkers explicites. Cet ensemble de référence comprend plus de 50 modèles paramétrés (soit 300 exemples concrets), associés à leurs propriétés de sûreté ou de vivacité, des informations détaillées sur tous les modèles et sur les espaces d'états.

Les modèles du BEEM sont initialement implémentés dans un langage de modélisation de bas niveau basé sur des machines à états finis communicantes appelé DVE. Des transformations permettent de passer automatiquement de modèles DVE en modèles Fiacre ou Promela. Ces modèles sont pour la plupart des exemples bien connus couvrant divers domaines d'applications. Les domaines sont les suivants : - les algorithmes d'exclusion mutuelle ; - les protocoles de communication ; - les contrôleurs ; - les algorithmes d'élection de leader ; - les ordonnancements ; - les énigmes. Chaque modèle est classé selon sa complexité, allant d'un exemple très simple spécifié en quelques lignes de code (*toy*), à un étude de cas complexe dont la description comporte plus de 100 lignes de code (*complex*).

Ces modèles sont accompagnés de propriétés. Deux types de propriétés sont prises en charges, les propriétés d'atteignabilité et les propriétés de logique temporelle linéaire (LTL). Les propriétés sont exprimées par des propositions atomiques définies à travers d'expressions sur des variables de modèle. Les modèles ne sont pas tous corrects et certains contiennent des erreurs.

#### Les domaines d'applications

**Les algorithmes d'exclusion mutuelle** garantissent un accès exclusif à une ressource partagée. Les modèles de ces algorithmes se composent généralement de plusieurs processus presque identiques qui communiquent via des variables partagées. Les algorithmes appartenant à cette catégorie sont par exemple l'algorithme du boulanger ou de Peterson.

**Les protocoles de communication** ont pour objectif de garantir la communication sur un support peu fiable. Un modèle de protocole de communication comprend généralement un processus émetteur, un processus récepteur et un bus ou un support. Les processus communiquent par poignée de main et les variables partagées ne sont pas utilisées. Les protocoles sont par exemple le bounded retransmission ou le collision avoidance.

**Les algorithmes d'élection de leaders** ont pour objectif de choisir un leader unique parmi un ensemble de nœuds. Ces modèles consistent en un ensemble de processus connectés dans un anneau, un arbre ou un graphe ; la communication s'effectue via des canaux bufferisés. Parmi ces modèles ont trouve par exemple les algorithmes basés sur l'extinction et sur les filtres, le Firewire (IEEE 1394) tree identification protocol ou le Lann leader election algorithm for token ring.

**Les contrôleurs** sont généralement des modèles disposant d'une architecture centralisée : un processus de contrôle communique avec différents processus représentant des parties du système. La communication peut se faire à la fois par des variables partagées et par poignée de main. Les modèles sont par exemple le contrôleur de puissance audio / vidéo, le contrôleur d'un ascenseur, le contrôleur de vitesse ou le contrôleur de porte de train.

**Les énigmes, les ordonnanceurs, et autres** incluent divers modèles dont par exemple un modèle d'ordonnancement de machines de production, différentes énigmes (Bridge puzzle ou Peg solitaire puzzle), un protocole de service de télécommunication, des algorithmes d'authentification (Needham-Schroeder) etc....

### 8.1.2 Intégration au VOS

#### Structure du Beem

Le BEEM représente une base de cas de grande taille reconnue en model checking, il est donc naturel de s'inspirer du BEEM pour structurer les connaissances du VOS, aussi la structure des éléments du BEEM et du VOS est très proche. Chaque modèle du BEEM est décrit par un dossier dans lequel se trouve le fichier contenant le modèle (.mdve et .fiacre) pouvant être paramétrés. Pour Peterson par exemple, le nombre de processus participant à l'exclusion mutuelle est paramétrable (N). Le modèle est accompagné d'une description (fichier .xml) dans lequel se trouve son nom accompagné de descriptions, du

domaine et du type auquel il appartient, de paramètres (provoquant parfois des erreurs), de propriétés et d'informations sur le modèle. Les instances sont générées dans un sous-dossier, et les résultats de vérifications sont capturées (fichier result.xml). Celui-ci contient pour chaque instance les informations quant à la vérification (nombre d'états, propriétés violées, taille du contre-exemple etc...). Cette structure est partiellement représentée par le méta-modèle décrit par la figure 8.1. Pour ingérer les modèles du BEEM dans le VOS, nous considérons cette structure.

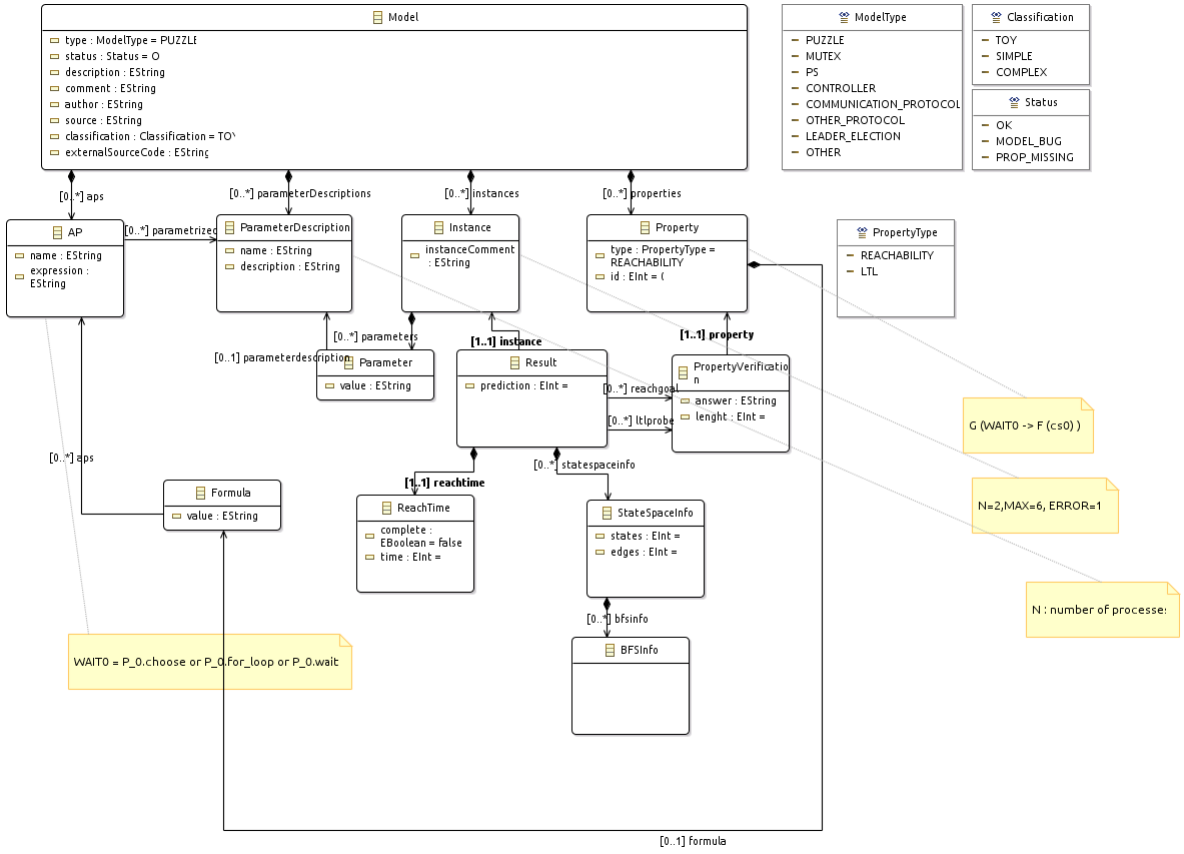


FIGURE 8.1 – La structure du Beem

### Ingestion du BEEM

Pour ingérer automatiquement ces modèles dans le VOS, nous proposons la spécification donnée par le tableau 8.1.

Après ingestion, le VOS contient 61 *Pattern cases*. Il faut bien comprendre que la finalité du BEEM n'est pas d'avoir une base de connaissances réutilisable, mais a pour objectif d'analyser les performances, c'est pourquoi il faut compléter manuellement l'ingestion. Nous avons défini manuellement un *Problem case* (celui de l'exclusion mutuelle)

| Concept du VOS | Concept Beem | Complément d'information |
|---|---|---|
| Model | Model | Les propriétés du Model du BEEM sont recréées en propriétés primitives dans le Model du VOS (type, status, auteurs...). |
| ModelElements | Process + code | Le code source correspondant au modèle est parsé (s'il s'agit d'un code fiacre), et des éléments de modèles (états, transitions...) sont instanciés dans le VOS. |
| Sample Case | Instance | Chaque instance d'un modèle du BEEM est traduite en un sample case dans le VOS |
| RunResult | Result | Les résultats de vérifications sont extraits du fichier de résultats. Les éléments du VOS ainsi instanciés sont : PropertyVerification, FormalProperty, RunResult, Formula |
| AtomicProposition | AP | Des références sont créées entre les propositions atomiques et les éléments de modèles correspondants. |
| Trace | – | Le BEEM ne contient pas de traces. Mais nous accompagnons chaque instance (au format fiacre) d'un fichier de trace du même nom (au format généré OBP). Ce fichier de trace est ensuite parsé et les éléments correspondants sont instanciés dans le VOS (Configurations, Configuration Transitions...). |
| ConfigurationElement | Component, Proc ... | Tous les éléments de configurations sont instanciés en ConfigurationElement dans le VOS, puis reliés à leur élément correspondant dans le modèle ou les spécifications, accompagné d'une valeur. |
| ProblemCase | – | Ne sont pas automatiquement ingérées |
| ComponentCase | – | Ne sont pas automatiquement ingérées |
| PatternCase | ParameterDescription et Parameter | Les modèles du BEEM sont des algorithmes paramétrables. Pour Peterson par exemple il est possible de modifier le nombre de processus (variable $N$). Pour ces modèles, la stratégie est de les assimiler à des pattern cases. |
| TransformationRule | – | Ne sont pas automatiquement ingérées |
| ReasoningProcess | – | Ne sont pas automatiquement ingérées |
| Revision | – | Ne sont pas automatiquement ingérées |

TABLE 8.1 – Règles d'ingestion du BEEM dans le VOS

et des liens entre ce *problem case* et les *pattern cases* qui en sont solution. Au final, le VOS contient les cas présentés dans le tableau 8.2.

| **Cas** | Problem cases | Component cases | Pattern cases | Sample cases |
|---|---|---|---|---|
| **Quantité** | 1 | 0 | 61 | 0 |

TABLE 8.2 – Métriques sur l'ingestion du BEEM dans le VOS

## 8.2 Interactions basées sur des connaissances liées au model checking

Les activités cognitives sont outillées par différentes interactions. Certaines visent à simplifier le diagnostic (visualisation des communication), d'autres à générer de nouvelles observations (observer des comportements clés). Si ces activités peinent à être réalisées manuellement (traces trop grandes et trop détaillées), c'est que la quantité d'information générée par l'activité de parcours du contre-exemple est grande, amenant le diagnosticien au-delà de ses capacités de mémoire de travail. Ce phénomène est connu sous le terme de surcharge cognitive [Swe+90]. Nous proposons d'outiller ces activités cognitives s'appuyant sur les connaissances acquises au sein du VOS. Nous comptons ainsi réduire l'effort et les risques d'erreurs du diagnosticien.

### 8.2.1 Inspecteur de trace

Une trace est composée de configurations et de transitions. Prenons comme exemple une configuration simplifiée extraite d'une vérification.

```
Config: 6{
 component: '{Run}1' [ FIFOs]
 proc: '{Gc}1' [@Idle, targetAvail=false, canPass=false, mess={}]
 proc: '{Net}1' [@Sending, targetAvail=false, canPass=false, mess={source={
   ID_CLT_1},target={ID_PLC_1},data={oper={RED},res={RES1},ans={}},sign={
   ID_Gc}}]
 proc: '{Plc}1' [@Idle, resAvail=false, access=false, opRes={}, mess={}]
 proc: '{Plc}2' [@Idle, resAvail=false, access=false, opRes={}, mess={}]
 context: '{env}1' [@s2]
 observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1}1' [@wait]
}
```

Listing 8.1 – Trace pour le scénario 1b

On y trouve l'état des FIFOs gérées par le processus *component*. Pour chaque instance de processus (par exemple : $Plc_1$), on trouve son état ($Idle$), l'état de chaque observa-

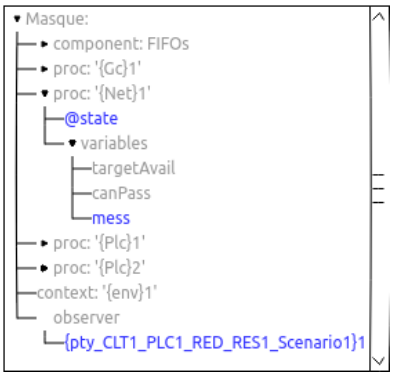


FIGURE 8.2 – Masque

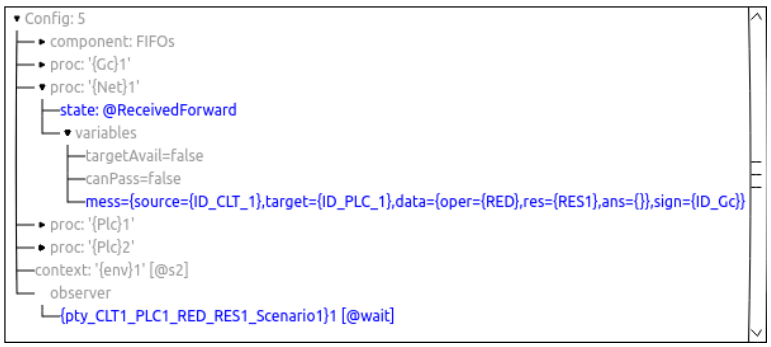


FIGURE 8.3 – Masque appliqué à une configuration

teur ($pty\_CLT_1\_PLC_1\_RED\_RES_1\_Scenario_1[@wait]$ et le contenu de ses variables ($resAvail = false$), toutes les informations n'étant pas utiles simultanément.

### Masque de configuration

Lors d'une vérification faite par OBP, toutes les configurations ont la même structure hiérarchique. Cette structure peut être représentée dans un composant graphique de type explorateur, qui reflète la décomposition de la configuration en éléments (terminal ou non), chaque élément non-terminal pouvant être déplié ou replié.

*Un masque de configuration* permet de définir les éléments qui sont pertinents pour l'inspection. Lorsqu'un niveau est replié, c'est généralement qu'on ne s'y intéresse pas, ni à ce qu'il contient. Lorsqu'un niveau est déplié, c'est généralement qu'on veut y inspecter quelque chose. L'illustration 8.2 présente un tel masque.

Dans ce masque, plusieurs niveaux sont dépliés, le processus $GC_1$ et les observateurs. Dans le processus $GC_1$, l'état (*state*) est sélectionné (couleur bleue), signifiant qu'il est inspecté. Les variables sont dépliées mais seule *mess* sera inspectée. Les observateurs sont également dépliés mais seul $pty\_CLT_1\_PLC_1\_RED\_RES_1\_Scenario_1a$ sera inspecté. Le masque de configuration agit ensuite comme un masque pour tous les inspecteurs qui seront instanciés conformément à celui-ci. Dans l'exemple 8.3, le masque est appliqué sur une configuration réelle .

Si la présentation de la configuration est définie par le masque utilisé, l'utilisateur a la possibilité à tout moment de déplier ou de replier des branches sur cette configuration, puis de sélectionner certains éléments pour affiner son inspection. Le masque utilisé est implicitement le masque modifié pour les inspections futures. Cependant, si on veut garder l'inspection (et son paramétrage) de la configuration courante, on peut détacher un inspecteur qui devient indépendant et sur lequel on peut faire des affinages qui ne seront pas répercutés sur les autres inspecteurs.

### Masque de transition

Ce masque peut également être appliqué au transitions. Une transition est composée de deux configurations et d'un label. Seule la différence entre les éléments des deux configurations est pertinente. La figure ci-dessous présente une transition et la différence entre les deux configurations 8.4.

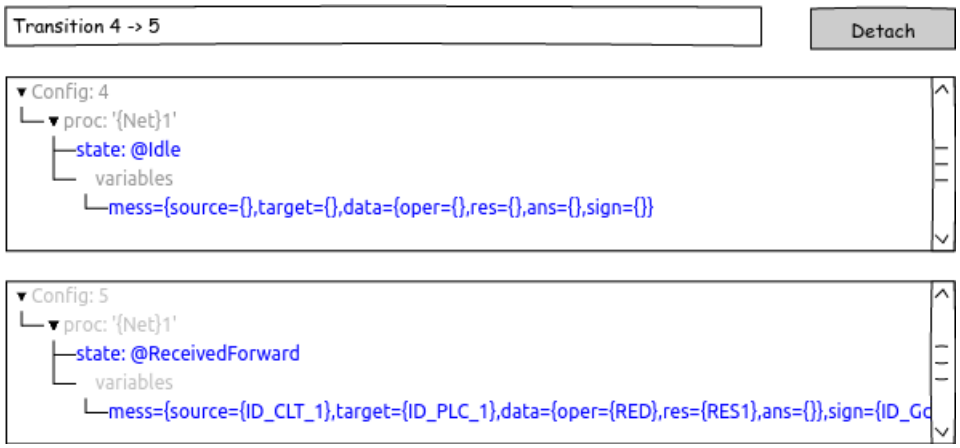


FIGURE 8.4 – Le masque appliqué sur une transition

### Masque de trace

La trace associée à une configuration (par ex : 5) est la succession de configurations et de transitions depuis le début de la simulation jusqu'à la configuration en question (5). Cependant une trace pouvant contenir des centaines de configurations, et il pouvoir la synthétiser. Une trace est un chemin dans le graphe d'exploration, mais selon le point d'intérêt de l'utilisateur, tous les nœuds (configurations) et arcs (transitions) ne sont pas pertinents. Le plus souvent la pertinence d'une configuration vient d'une différence avec les configurations précédentes, par exemple lorsqu'un processus ou un observateur a changé d'état ou bien qu'une variable ou un prédicat a changé de valeurs. Le point de vue (processus, observateur, variable, prédicat, ...) selon lequel on souhaite simplifier une trace est appelé une *facette*.

Généralement, lorsqu'on veut regarder la trace au travers d'une *facette*, les configurations pertinentes sont celles où la *facette* a changé de valeur. Ainsi l'opération de filtrage de la trace au travers d'une *facette* produit un sous-ensemble de configurations, qu'on appellera une *route*. La *route* associée à une *facette* est donc la succession des configurations où la *facette* change de valeur.

Prenons par exemple la trace du scénario b allant jusqu'à la configuration 95. Elle peut être filtrée selon la *facette* de l'observateur *pty_CLT1_PLC1_RED_RES1_Scenario1b*,

on obtient la route du listing 8.2 composée des 4 configurations pertinentes où l'observateur a changé d'état. Cette route est simplifiée ci-dessous :

```
1
2  config: 0 {
3   component: '{Run}1' [...]
4   proc: '{Gc}1'@Idle , '{Net}1'@Idle, '{Plc}1'@Idle, '{Plc}2'@Idle
5   context: '{env}1' [@start]
6   observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1a}1' [@start]
7   observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1b}1' [@start]
8  }
9
10 config: 1 {
11  component: '{Run}1' [...]
12  proc: '{Gc}1'@Idle , '{Net}1'@Idle, '{Plc}1'@Idle, '{Plc}2'@Idle
13  context: '{env}1' [@s2]
14  observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1a}1' [@wait]
15  observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1b}1' [@wait]
16 }
17
18 config: 25 {
19  component: '{Run}1' [
20  fifo_PLC_1_NET=[{source={ID_PLC_1},target={ID_CLT_1},data={oper={RED},res
       ={RES1},ans={ACK}},sign={ID_Gc}}]
21  proc: '{Gc}1'@Idle , '{Net}1'@Idle, '{Plc}1'@Idle, '{Plc}2'@Idle
22  context: '{env}1' [@s3]
23  observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1a}1' [@success]
24  observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1b}1' [@onlyonce]
25 }
26
27 config: 95 {
28  component: '{Run}1' [ fifo_PLC_1_NET=[{source={ID_PLC_1},target={ID_CLT_1
       },data={oper={RED},res={RES1},ans={ACK}},sign={ID_Gc}}]
29  proc: '{Gc}1'@Idle , '{Net}1'@Idle, '{Plc}1'@Idle, '{Plc}2'@Idle
30  context: '{env}1' [@s3]
31  observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1a}1' [@success]
32  observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1b}1' [@reject:error]
33 }
```

Listing 8.2 – Seconde trace pour le scénario 1b

### 8.2.2 Visualisation de routes

Une route est composée de deux ensembles : un ensemble de configurations et un ensemble de transitions abstraites. Une transition abstraite est un chemin entre deux configurations quelconques, ce qui correspond à une suite de transitions dans le graphe

d'exploration.

Une route résulte du filtrage de la trace via une facette. La figure ci-dessous 8.5 présente la route associée à la facette de l'observateur, dont la trace brute est composée des configurations 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 20, 25, 30 ,36, 44, 53, 64, 77 et 95.

Ici la route est composée de 4 configurations : 0, 1, 25, 95. Entre deux configurations de la route, on trouve une transition abstraite (suite de transitions concrètes) et le nombre de transitions concrètes formant la transition abstraite est indiqué sur un bouton situé sur la transition abstraite. Cliquer sur le bouton associé à une transition abstraite déplie entièrement la transition abstraite en la remplaçant par la suite de transitions concrètes dont elle est composée et les configurations extrémités des transitions concrètes. Le bouton *Suivant* permet de déplier la première transition concrète de la transition abstraite ; Le bouton *Précédent* permet de déplier la dernière transition concrète de la transition abstraite. Suite à un dépliement, une transition abstraite est remplacée par deux transitions connectées par la configuration suivant une des extrémités de la transition abstraite.

Dans la figure 8.6, les précédents des configurations 25 et 95 ont été dépliées, faisant apparaître les configurations 20 et 77. Les valeurs de la facette sont mises à jour pour refléter l'état de la facette dans les configurations qui sont apparues suite au dépliement. Il peut arriver que deux configurations de la route soient consécutives et séparées par une transition concrète.

### 8.2.3 Diagnostic graphique des traces

Jusqu'alors, on a présenté des visualisations où une seule facette était utilisée comme filtre de la trace, et la même seule facette était visualisée le long de la route résultant du filtre. Or ce sont deux dimensions orthogonales : on peut visualiser plusieurs facettes le long d'une route et/ou on peut utiliser plusieurs filtres pour générer la route.

#### Projection de la route sur plusieurs facettes

Une fois une route obtenue à partir d'une facette (un observateur, un prédicat, un processus, une variable, ...), il est intéressant de visualiser d'autres facettes, chacune dans un couloir, le long de la route.

Dans la figure ci-dessus 8.7, la route a été obtenue par filtrage de la facette $pty\_CLT_1\_PLC_1\_RED\_RES_1\_Scenario_1b$. Les deux couloirs représentent les valeurs de la facette NET1@state (l'état du processus *NET*) et de la facette toContext (une FIFO entre *GC* et le contexte) le long de la route. Dans cet exemple, on remarque que la FIFO *toContext* contient deux fois le message d'acquittement (configuration 20 et 77), puis le message est lu dans les configurations suivantes (configuration 25 et 95), signifiant que

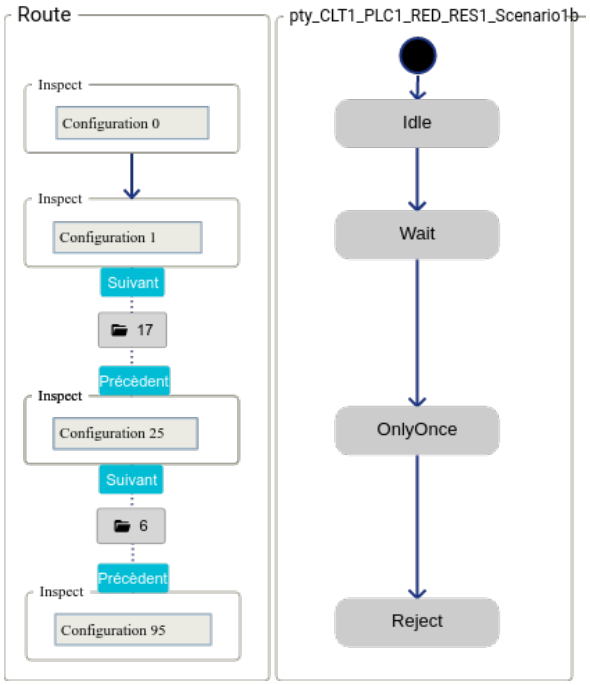


FIGURE 8.5 – Route

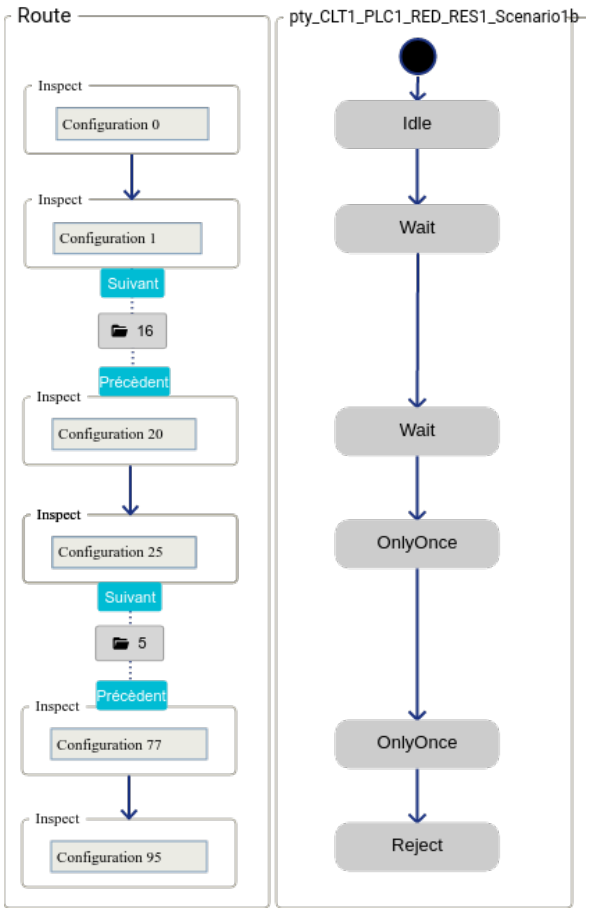


FIGURE 8.6 – Route dépliée

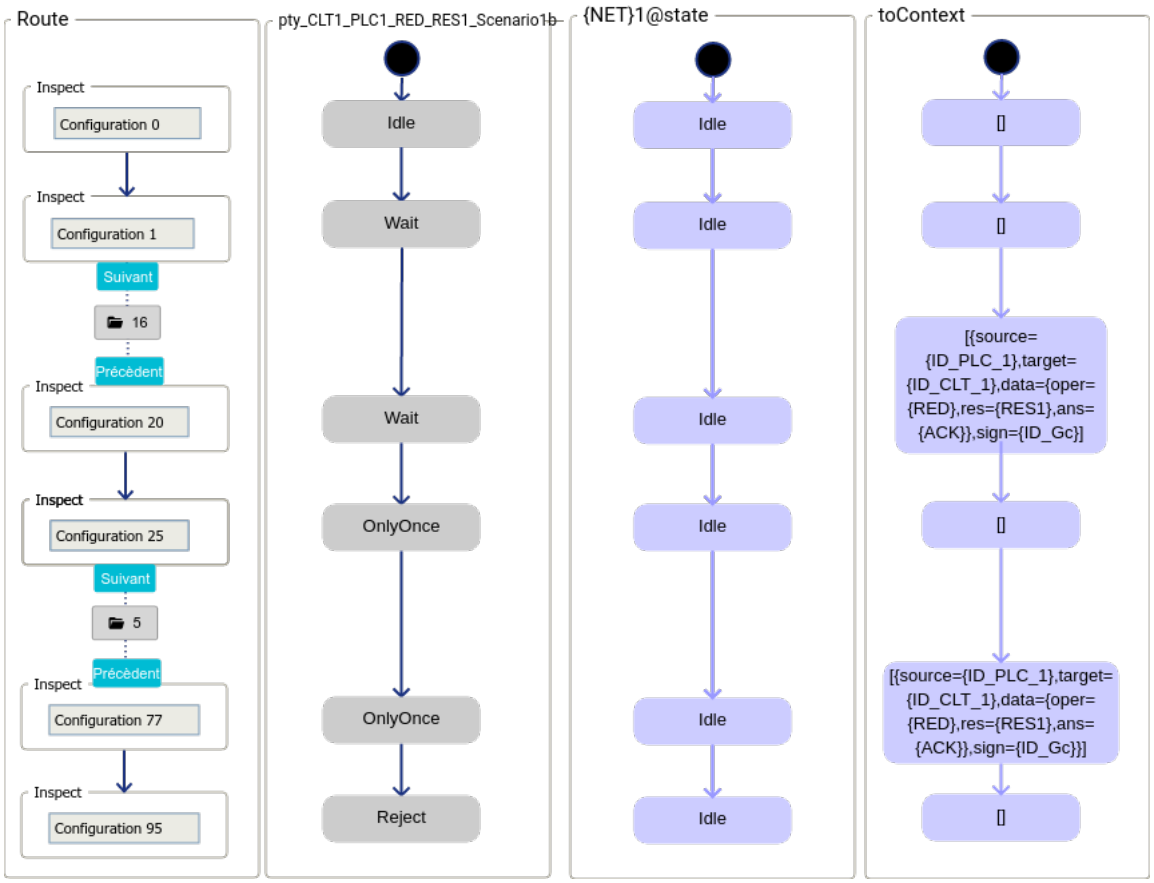


FIGURE 8.7 – Visualisation d'une route et de deux autres facettes

le contexte a reçu deux fois l'acquittement. Ces réceptions déclenchent les changements d'état de l'observateur.

### Combinaison de filtres de facettes

Un filtre de facette opère une restriction sur la trace et résulte en un ensemble de configuration, qu'on appelle une route. Combiner les filtres au moyen des opérateurs logiques classiques (OU, ET, NOT) revient à effectuer des opérations d'union, d'intersection et de différence sur les routes. En plus des opérateurs ensemblistes, il est possible des fournir des fonctions agrégatives sur les ensembles comme le count (qu'on utilise sur les transitions abstraites), le min ou le max.

### Masque de transitions

Une particularité des processus concurrents est les échanges de messages. C'est d'ailleurs pour cela que la représentation sous la forme d'un diagramme de séquence est intéressante. Des masques spécifiques aux transitions sont donc pertinents. Ce masque est similaire à celui des traces, à la différence que les informations présentées dans chaque configuration soient réduites à la différence avec les informations de la configuration précédente.

Pour l'observateur pris en exemple, les événements sont basés sur les envois et réceptions de messages. Il est fastidieux de chercher dans les traces ce qui s'est passé avec le message $send\_CLT_1\_PLC_1\_RED\_RES_1$ et $recv\_CLT_1\_PLC_1\_RED\_RES_1\_ACK$. Un masque dédié aux messages consiste à présenter une, plusieurs ou toutes les configurations où ces messages sont envoyés ou/et reçus. Ce masque de messages s'applique aux traces de transitions.

### Utilisation du VOS

La figure 8.8 présente l'usage du VOS pour l'outil d'analyse de trace par facettes.

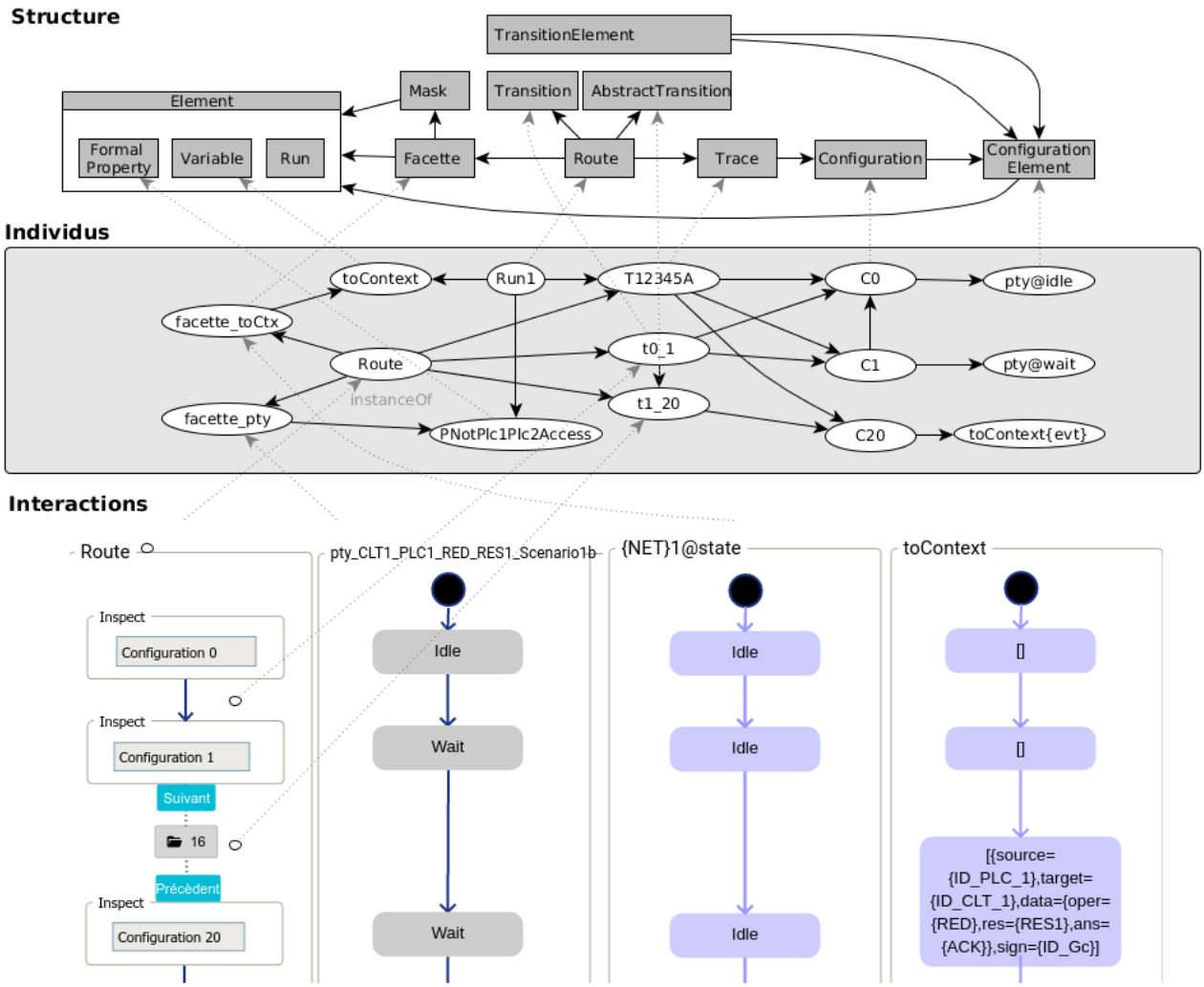


FIGURE 8.8 – Le VOS utilisé pour l'outil de trace

Chapitre 9

# Conclusions et perspectives

## 9.1 Conclusions

### 9.1.1 Enjeux

Il est généralement difficile d'établir le diagnostic en partant de contre-exemples issus de model checkers. Les principales causes que nous avons identifiées sont la complexité des activités cognitives menées sur des traces de très grandes tailles, le besoin d'exprimer des corrélations entre les informations issues de différents langages, le manque de connaissances de domaine, et dans le cas où celles-ci existent, la nécessité d'établir un processus permettant de les réutiliser. Orthogonalement, des moyens sont nécessaires pour organiser ces connaissances et les interactions qui en découlent. On constate que ces enjeux sont difficilement traitables séparément. Les activités cognitives par exemple, ne peuvent être facilitées qu'en s'appuyant sur un système organisant les connaissances et les interactions. Un tel système n'est efficace que s'il contient des connaissances mais pour que ces connaissances existent, il faut une méthode et un formalisme permettant de les capturer. Ces dépendances nous ont contraint à aborder l'ensemble de ces enjeux. Dans cette thèse nous ne répondons pas en détail à un enjeu particulier. En revanche, nous proposons un cadre général d'aide au diagnostic articulant nos propositions pour ces enjeux.

### 9.1.2 Contributions

#### Une dimension relative aux connaissances

Lorsque le diagnosticien vérifie le SAE, il met en œuvre un ensemble d'activités cognitives (vérifier, visualiser, isoler, générer...) chacune pouvant être apparentée à des outils ou à des techniques. Pour les catégoriser, notre proposition est de placer les activités cognitives dans la taxonomie de Bloom. L. W. Anderson et D. R. Krathwohl, lors de la révision de cette taxonomie, ont ajouté une deuxième dimension des connaissances. Nous avons repris l'idée de cette dimension, cependant, selon une organisation différente.

Le diagnostic est une des phases de la vérification par model checking. De nombreux travaux, principalement basés sur les contre-exemples, adressent certains enjeux du diagnostic. Cependant, les traces ne suffisent pas toujours pour comprendre le problème. Il

faut alors corréler plusieurs points de vues en utilisant des connaissances différentes. Par exemple, relier l'algorithme aux traces, ou combiner des visualisations de nature diverses. A ces fins, nous avons utilisé une approche par fédération de modèles.

L'ingénierie de domaine joue un rôle essentiel dans le développement logiciel, comme en atteste, entre autres, les approches du type Domain Driven Design [Avr+06]. Celles-ci portent la compréhension et la formulation du domaine au cœur de la conception logicielle. L'analyse du domaine fournit des descriptions de phénomènes, des invariants et des propriétés utiles pour la vérification. Il est donc sensé d'établir les connaissances de domaine comme un des niveaux de cette dimension.

L'activité de conception est perceptible de différentes manières. Nous pouvons la percevoir comme une structuration de problèmes mal définis, à l'image de H.A Simon [Sim73], pour lequel les spécifications d'un projet de conception sont habituellement d'un niveau abstrait. Nous pouvons aussi la percevoir comme une construction de représentations, à l'image de W. Visser [Vis09], incluant génération, transformation et évaluation des représentations. Quel qu'en soit sa forme, il est utile de conserver les connaissances portées par les activités de conception, que nous appelons connaissances de gestion.

Nous proposons une structure suivant trois niveaux de connaissances, celles liées au model checking, au domaine et à la gestion.

#### Articulation des activités d'ingénierie

Quand la solution à un problème est difficile à trouver, les cycles de conception-vérification s'accumulent. Il faut alors trouver un moyen de réduire le nombre de ces cycles. Nous proposons, à cet effet, de conserver les solutions satisfaisantes afin de les réutiliser ultérieurement. Nous considérons trois types de solutions, les *sample cases*, les *pattern cases* et les *component cases*. Chaque solution répond à un problème de domaine appelé *problem case*. Un *problem case* permet de comprendre et de capturer les problèmes passés. Problèmes et solutions constituent ainsi une base d'expertise réutilisable liée au domaine d'ingénierie.

Pour s'appuyer sur le domaine, il faut disposer en complément de l'ingénierie de l'application, d'une ingénierie du domaine. Ces ingénieries sont spatialement divisées, avec pour chacune d'un côté l'espace du problème et de l'autre l'espace de la solution. Pour faciliter le passage entre ces espaces et ingénieries, et donc capturer et réutiliser le domaine, nous proposons une méthode de résolution de problème. Celle-ci est découpée en deux espaces, l'espace du problème dans lequel on construit la représentation du problème, et l'espace de la solution où l'on génère et évalue la solution choisie.

Un processus de conception et vérification par model checking produit une profusion d'informations hétérogènes qui évoluent au cours du temps et corrélées. Or notre constat est que ces informations ne sont pas bien gérées. Les activités associées ne sont ni contrô-

lées ni enregistrées, et il n'y a pas de consensus sur la façon de les capturer. Nous proposons une infrastructure basée sur le concept d'*organizing system*, "un ensemble de ressources organisé intentionnellement et les interactions qu'elles supportent [Glu12]". L'infrastructure organise ainsi les différentes informations et propose des interactions, en particulier pour accompagner le processus de diagnostic.

### 9.1.3 Critique sur l'approche

#### Coût de mise en place

Il existe deux cas de figure, soit l'application est déjà conçue, soit elle est en cours de conception. Lorsqu'elle est déjà conçue, il est illusoire de penser que l'on va retrouver toutes les informations du domaine. Il faut alors réussir à refactorer l'application pour la faire correspondre aux problèmes, ou aux solutions existantes, lesquelles ramènent à leur tour des propriétés de problème à laquelle la solution répond. Lorsque l'application est en cours de conception, il faut, à chaque incrément réussir à capturer l'ensemble des solutions choisies dans une base de connaissance, les tracer au fil des améliorations ou abandons, afin de tracer l'état général des propriétés à un instant t qui permettent de prédire le comportement attendu de l'application.

Lorsqu'une solution est trouvée, celle-ci peut être capturée dans le domaine pour un usage ultérieur. Nous avons vu qu'il existe différentes façons de la capturer (*component case*, *pattern case* ou *sample case*). L'effort de réutilisation et de diagnostic est lié à la façon dont a été capturée la solution. Un *component case* est plus simple à réutiliser qu'un *sample case*. À l'inverse, l'effort de capture de la solution est plus simple pour un *sample case* qu'un *component case*.

Que l'application soit en cours de conception ou déjà conçue, utiliser le domaine peut donc avoir un coût non négligeable. Il faut également trouver le bon compromis entre l'effort à fournir pour capturer la solution et le bénéfice escompté de sa réutilisation. Un *component case* sera plus dur à capturer car il faudra notamment définir ses interfaces, mais il sera aussi plus facile de le réutiliser. Les expériences passées ont généralement montré le bénéfice de capturer le domaine et ses solutions quand celui-ci est utilisé sur le long terme (design patterns, bibliothèques graphiques...).

#### Généricité, Scalabilité

Bien qu'essentielle pour valider des systèmes critiques, nous avons constaté que le model checking reste peu appliqué dans l'industrie. Ce travail a été réalisé avec la volonté de promouvoir l'usage de cette approche chez les ingénieurs. Aussi, cette thèse s'est concentrée sur la recherche de méthodes et concepts simples, réalistes et praticables.

Nous pensons que l'approche est scalable pour différentes raisons, entre autres :

- Elle repose sur un cadre général qui est inspiré de Bloom, cadre mature et déjà appliqué à divers domaines depuis de très longues années. Les catégories présentes dans les deux dimensions peuvent être aisément adaptées au contexte industriel. Il est même envisageable d'utiliser ce cadre dans un tout autre domaine que le model checking, comme le test par exemple.

- La formalisation des connaissances s'appuie sur des formalisations préexistantes et admises (BEEM par exemple) pour définir les concepts clés. Il est possible de définir et d'associer de nouveaux concepts à ceux déjà définis.

- L'infrastructure repose sur des bonnes pratiques d'ingénierie. Elle sépare les préoccupations (informations, connaissances, interactions) facilitant ainsi les changements. Par le biais de la fédération de modèles et de l'outil Openflexo, elle permet d'ajouter de nouveaux connecteurs techniques, des concepts ou des relations.

- La montée en charge est un point fort de ce type d'approche. En effet, l'efficacité d'un système reposant sur les connaissances dépend de la quantité (et également qualité) des connaissances disponibles. Par analogie avec le CBR, il est possible de trouver des cas proches en recherchant des descripteurs, mais une telle technique sera bien plus efficace si elle dispose d'une large quantité de cas.

- Notre cadre semble s'appliquer au-delà de la construction d'un système logiciel. Prenons l'activité de résolution de problèmes que représente cette thèse. Celle-ci vise à "améliorer la qualité du système logiciel et faciliter le diagnostic". Cette phrase peut être vue comme une propriété du problème de l'application. Lors de l'activité de résolution de problèmes, nous avons trouvé des problèmes de domaine, "définir des interactions", ou "organiser les activités cognitives". Ceux-ci ramènent avec eux des propriétés de problèmes de domaine. Pour répondre au problème "organiser les activités cognitives", nous aurions pu produire une nouvelle classification, mais celle-ci aurait été trop difficile à évaluer. Nous avons opté pour une solution de domaine "la classification de Bloom" et ses propriétés de solutions de domaine. Si nous n'avions pas choisi Bloom, nous aurions pu résoudre le problème initialement posé de façon différente.

## 9.2 Perspectives

Le framework est principalement destiné au support d'interactions de diagnostics, comme par exemple l'outil d'analyse de traces par facettes. Cependant, de nouvelles interactions seraient envisageables. Le Model Based Diagnosis (MBD), par exemple, repose sur l'existence d'un modèle permettant de prédire le bon fonctionnement du système (décrit par des System Descriptions (SD)), et d'observations (OBS). Les *problem cases* et leurs propriétés formelles associées, représentent un modèle prédictif, donc des SD. Les explorations obtenues par model-checking sont des OBS. Les propriétés violées rentrant

en conflit avec les SD, sont des symptômes. L'approche MBD impose que le système soit décrit par des composants. Le système doit donc être un ensemble de *component cases*. Cela pose le problème de la complexité de la composition des *components case*, et de leur équivalence avec l'approche de R. Reiter [Rei87].

Un processus de diagnostic repose sur une succession d'activités cognitives complexes qu'il est difficile d'exprimer. Notre framework offre la capacité de les capturer et de les organiser. Il est donc envisageable d'exprimer le déroulement du processus de diagnostic. Cette capture pourrait permettre de réutiliser et rejouer ces processus dans des contextes différents.

Dans l'article [Lei+17], nous avons proposé un modèle de diagnostic simple. Il met en avant l'objectif du diagnostic. Il peut être curatif lorsque l'intention est de corriger un défaut, pédagogique lorsqu'il sert à comprendre le système, évolutif quand il permet de contrôler l'état d'un système. Une autre perspective intéressante est l'étude des corrélations entre intentions et interactions. Celle-ci pourrait, à terme, compléter le cadre de diagnostic.

# DISSÉMINATION

## Articles soumis et acceptés durant la thèse

— Vincent Leilde, Vincent Ribaud, Philippe Dhaussy. An Organizing System to Perform and Enable Verification and Diagnosis Activities. Intelligent Data Engineering and Automated Learning – IDEAL 2016, Oct 2016, Yangzou, China. pp.576-587.

— Vincent Leilde, Vincent Ribaud, Philippe Dhaussy. Model-based Diagnosis Patterns for Model Checking. PAME 2016, Oct 2016, St Malo, France. pp.7-12

— Vincent Leilde, Vincent Ribaud, Ciprian Teodorov, Philippe Dhaussy. A Diagnosis Framework for Critical Systems Verification. 15th International Conference on Software Engineering and Formal Methods, SEFM 2017, Sep 2017, Trente, Italy. Short Papers 1-6.

— Vincent Leilde, Vincent Ribaud, Ciprian Teodorov, Philippe Dhaussy. A Problem-Oriented Approach to Critical System Design and Diagnosis Support. 1st International Workshop on Modeling, Verification and Testing of Dependable Critical Systems (DETECT 2018), Oct 2018, Marrakesh, Morocco.

— Vincent Leilde, Vincent Ribaud, Ciprian Teodorov, Philippe Dhaussy. Domain-oriented Verification Management. 8th International Conference on Model and Data Engineering (MEDI 2018), Oct 2018, Marrakesh, Morocco. pp.30-39.

## Articles soumis et acceptés liés à la thèse

— Joël Champeau, Vincent Leilde, Papa Issa Diallo. Model Federation in toolchains. MODELS 2013, Sep 2013, Miami, United States.

— Weiqing Zhang, Vincent Leilde, Birger Moller-Pedersen, Christophe Guychard, Joël Champeau. Towards Tool Integration through Artifacts and Roles. Asia-Pacific Software Engineering Conference, 2012, Hong Kong, China.

# BIBLIOGRAPHIE


[Abr+08] Rui Abreu et al., « Automatic Software Fault Localization Using Generic Program Invariants », in : *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, event-place : Fortaleza, Ceara, Brazil, New York, NY, USA : ACM, 2008, p. 712-717.

[Abr+09] Rui Abreu et al., « A practical evaluation of spectrum-based fault localization », in : *Journal of Systems and Software*, SI : TAIC PART 2007 and MUTATION 2007 82.*11* (nov. 2009), p. 1780-1792, ISSN : 0164-1212.

[AE94] Aamodt Agnar et Plaza Enric, « Case-Based Reasoning : Foundational Issues, Methodological Variations, and System Approaches », in : *AI Communications 1* (1994), p. 39-59, ISSN : 0921-7126.

[Alr+13] D. Alrajeh et al., « Elaborating Requirements Using Model Checking and Inductive Learning », in : *IEEE Transactions on Software Engineering* 39.*3* (mar. 2013), p. 361-383, ISSN : 0098-5589.

[Alr+15] Dalal Alrajeh et al., « Automated support for diagnosis and repair », in : *Communications of the ACM* 58.*2* (2015), p. 65-72.

[Avi+04] Algirdas Avižienis et al., « Basic concepts and taxonomy of dependable and secure computing », in : *Dependable and Secure Computing, IEEE Transactions on* 1.*1* (2004), p. 11-33.

[Avr+06] Abel Avram et al., *Domain-driven design quickly : [a summary of Eric Evans' Domain-driven design*, English, OCLC : 540271783, LaVergne, TN : C4Media, 2006.

[AZV06] Rui Abreu, Peter Zoeteweij et Arjan Van Gemund, « An Evaluation of Similarity Coefficients for Software Fault Localization », en, in : *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, Riverside, CA, USA : IEEE, 2006, p. 39-46.

[Bay70] Thomas Bayes, « An essay towards solving a problem in the doctrine of chances », in : *Studies in the History of Statistics and Probability* 1 (1970), p. 134-153.

[BE03a] J. Buckley et C. Exton, « Bloom's taxonomy : a framework for assessing programmers' knowledge of software systems », in : *11th IEEE International Workshop on Program Comprehension, 2003.* Mai 2003, p. 165-174.

[BE03b] Jim Buckley et Chris Exton, « Blooms ? Taxonomy : A Framework for Assessing Programmers ? Knowledge of Software Systems. », in : jan. 2003, p. 165-174.

[Ben+] Reda Bendraou et al., « Definition of an eXecutable SPEM 2.0 », en, in : (), p. 9.

[Bie+03] Armin Biere et al., « Bounded Model Checking », en, in : *Advances in Computers*, t. 58, Elsevier, 2003, p. 117-148.

[Bjø06] Dines Bjørner, *Software Engineering 3*, en, Texts in Theoretical Computer Science An EATC Series, Berlin/Heidelberg : Springer-Verlag, 2006.

[BK08] Christel Baier et Joost-Pieter Katoen, *Principles of model checking*, Cambridge, Mass : The MIT Press, 2008.

[Blo+64] Benjamin Samuel Bloom et al., *Taxonomy of Educational Objectives : The Classification of Educational Goals*, English, 1st ed.], N. Y. : D. Mckay, 1964.

[BNR03] Thomas Ball, Mayur Naik et Sriram K. Rajamani, « From symptom to cause : localizing errors in counterexample traces », in : *ACM SIGPLAN Notices*, t. 38, ACM, 2003, p. 97-105.

[Boe79] B. W. Boehm, « Classics in Software Engineering », in : sous la dir. d'Edward Nash Yourdon, Upper Saddle River, NJ, USA : Yourdon Press, 1979, p. 323-361.

[Bol+14] Matthew L. Bolton et al., « Automatically Generating Specification Properties From Task Models for the Formal Verification of Human–Automation Interaction », en, in : *IEEE Transactions on Human-Machine Systems* 44.*5* (oct. 2014), p. 561-575, issn : 2168-2291, 2168-2305.

[Bou+18] Erwan Bousse et al., « Omniscient debugging for executable DSLs », en, in : *Journal of Systems and Software* 137 (mar. 2018), p. 261-288, issn : 01641212.

[Bou09] Mustapha Bourahla, « Model-Based Diagnostic Using Model Checking », in : IEEE, 2009, p. 229-236.

[Boz+98] Marius Bozga et al., « Kronos : A model-checking tool for real-time systems (Tool-presentation for FTRTFT '98) », in : jan. 1998, p. 298-302.

[Buc+99] Francesco Buccafurri et al., « Enhancing model checking in verification by AI techniques », in : *Artificial Intelligence* 112.*1* (1999), p. 57-104.

[Bur+92] J. R. Burch et al., « Symbolic model checking : 1020 States and beyond », in : *Information and Computation* 98.*2* (juin 1992), p. 142-170, issn : 0890-5401.

[Bus96] Frank Buschmann, éd., *Pattern-oriented software architecture : a system of patterns*, Chichester ; New York : Wiley, 1996.

[BW13] Dirk Beyer et Philipp Wendler, « Reuse of verification results », in : *International SPIN Workshop on Model Checking of Software*, Springer, 2013, p. 1-17.

[CA07] Betty HC Cheng et Joanne M. Atlee, « Research directions in requirements engineering », in : *2007 Future of Software Engineering*, IEEE Computer Society, 2007, p. 285-303.

[CE00] Krzysztof Czarnecki et Ulrich Eisenecker, *Generative programming : methods, tools, and applications*, Boston : Addison Wesley, 2000.

[CES09] Edmund M. Clarke, E. Allen Emerson et Joseph Sifakis, « Model checking : algorithmic verification and debugging », en, in : *Communications of the ACM* 52.*11* (nov. 2009), p. 74, issn : 00010782.

[Cha91] Eugene Charniak, « Bayesian networks without tears. », in : *AI magazine* 12.*4* (1991), p. 50.

[Chr+16] Maria Christakis et al., « Integrated environment for diagnosing verification errors », in : *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2016, p. 424-441.

[CK04] Edmund Clarke et Daniel Kroening, « Tutorial : Software model checking », in : *International Conference on Formal Engineering Methods*, Springer, 2004, p. 9-10.

[Cla+00] Edmund Clarke et al., « Counterexample-guided abstraction refinement », in : *Computer aided verification*, Springer, 2000, p. 154-169.

[CP99] Jie Chen et R. J. Patton, *Robust Model-Based Fault Diagnosis for Dynamic Systems*, en, The International Series on Asian Studies in Computer and Information Science, Springer US, 1999.

[CZ05] Holger Cleve et Andreas Zeller, « Locating causes of program failures », en, in : ACM Press, 2005, p. 342.

[DAC99] Matthew B. Dwyer, George S. Avrunin et James C. Corbett, « Patterns in property specifications for finite-state verification », in : *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, IEEE, 1999, p. 411-420.

[DDH72] O. J. Dahl, E. W. Dijkstra et C. A. R. Hoare, éd., *Structured Programming*, London, UK, UK : Academic Press Ltd., 1972.

[DDV01] Françoise Darses, Françoise Détienne et Willemien Visser, « Assister la conception : perspectives pour la psychologie cognitive ergonomique », fr, in : (2001), p. 11.

[EsS+14] Khaoula Es-Salhi et al., « KriQL : a query language for the diagnosis of transition systems », in : *AVOCS'14*, t. 70, Electronic Communications of the EASST, Enschede, Netherlands, sept. 2014.

[FL10] Eduardo B. Fernandez et Maria M. Larrondo-Petrie, « Designing Secure SCADA Systems Using Security Patterns », en, in : IEEE, 2010, p. 1-8.

[Gam95] Erich Gamma, éd., *Design patterns : elements of reusable object-oriented software*, Addison-Wesley professional computing series, Reading, Mass : Addison-Wesley, 1995.

[Gar+10] Ángel García-Crespo et al., « ODDIN : Ontology-driven differential diagnosis based on logical inference and probabilistic refinements », in : *Expert Systems with Applications* 37.*3* (mar. 2010), p. 2621-2628, issn : 0957-4174.

[GKL04] Alex Groce, Daniel Kroening et Flavio Lerda, « Understanding counterexamples with explain », in : *International Conference on Computer Aided Verification*, Springer, 2004, p. 453-456.

[Glu12] Robert J. Glushko, « . Foundations for "Organizing Systems" », in : *The Discipline of Organizing, edited by Robert J Glushko* (2012).

[GOA05] Simon F. Goldsmith, Robert O'Callahan et Alex Aiken, « Relational Queries over Program Traces », in : *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, New York, NY, USA : ACM, 2005, p. 385-402.

[Gon+12] Liang Gong et al., « Interactive fault localization leveraging simple user feedback », in : *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Trento, Italy : IEEE, sept. 2012, p. 67-76.

[GR94] Joseph C. Giarratano et Gary Riley, *Expert systems : principles and programming*, 2nd ed, The PWS series in computer science, Boston : PWS Pub. Co, 1994.

[Guy+13] Christophe Guychard et al., « Conceptual interoperability through Models Federation », in : *Semantic Information Federation Community Workshop*, 2013.

[GV03] Alex Groce et Willem Visser, « What went wrong : Explaining counterexamples », in : *Model Checking Software*, Springer, 2003, p. 121-136.

[Hal+02] J.G. Hall et al., « Relating software requirements and architectures using problem frames », en, in : IEEE Comput. Soc, 2002, p. 137-144.

[Har+88] D. Harel et al., « Statemate : A Working Environment for the Development of Complex Reactive Systems », in : *Proceedings of the 10th International Conference on Software Engineering*, ICSE '88, event-place : Singapore, Los Alamitos, CA, USA : IEEE Computer Society Press, 1988, p. 396-406.

[HCK92] Walter Hamscher, Luca Console et Johan de Kleer, éd., *Readings in Model-based Diagnosis*, San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1992.

[HHS08] Denis Hatebur, Maritta Heisel et Holger Schmidt, « A formal metamodel for problem frames », in : *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2008, p. 68-82.

[Hol03] Gerard Holzmann, *The SPIN Model Checker : Primer and Reference Manual*, Anglais, Reprint, Boston : Addison Wesley, sept. 2003.

[Hol94] Gerard J. Holzmann, « The Theory and Practice of A Formal Method : NewCoRe. », in : *IFIP Congress (1)*, 1994, p. 35-44.

[HPS04] Adil Hameed, Alun Preece et Derek Sleeman, « Ontology Reconciliation », en, in : *Handbook on Ontologies*, sous la dir. de PD Dr Steffen Staab et Professor Dr Rudi Studer, International Handbooks on Information Systems, Springer Berlin Heidelberg, 2004, p. 231-250.

[Ign91] James P. Ignizio, *Introduction to expert systems : the development and implementation of rule-based expert systems*, New York : McGraw-Hill, 1991.

[Iri+79] Masao Iri et al., « An algorithm for diagnosis of system failures in the chemical process », in : *Computers & Chemical Engineering* 3.*1* (1979), p. 489-493.

[Jac01] Michael Jackson, *Problem frames : analysing and structuring software development problems*, eng, OCLC : 247895444, Harlow : Addison-Wesley [u.a.], 2001.

[JHS] James A Jones, Mary Jean Harrold et John T Stasko, « Visualization for Fault Localization », en, in : (), p. 5.

[JM11] Manu Jose et Rupak Majumdar, « Cause clue clauses : error localization using maximum satisfiability », in : *ACM SIGPLAN Notices* 46.*6* (2011), p. 437-446.

[Joe03] S. Joe Qin, « Statistical process monitoring : basics and beyond », en, in : *Journal of Chemometrics* 17.*8-9* (août 2003), p. 480-502, issn : 0886-9383, 1099-128X.

[JRS02] HoonSang Jin, Kavita Ravi et Fabio Somenzi, « Fate and FreeWill in Error Traces », in : *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2002, p. 445-459.

[JS07] Lingxiao Jiang et Zhendong Su, « Context-aware Statistical Debugging : From Bug Predictors to Faulty Control Flow Paths », in : *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, New York, NY, USA : ACM, 2007, p. 184-193.

[KL90] Bogdan Korel et Janusz Laski, « Dynamic slicing of computer programs », en, in : *Journal of Systems and Software* 13.*3* (nov. 1990), p. 187-195, issn : 01641212.

[KM08] Andrew J. Ko et Brad A. Myers, « Debugging reinvented : asking and answering why and why not questions about program behavior », en, in : *Proceedings of the 13th international conference on Software engineering - ICSE '08*, Leipzig, Germany : ACM Press, 2008, p. 301.

[Kra02] David R. Krathwohl, « A Revision of Bloom's Taxonomy : An Overview », en, in : *Theory Into Practice* 41.*4* (nov. 2002), p. 212-218, issn : 0040-5841, 1543-0421.

[Lam86] Leslie Lamport, « The mutual exclusion problem : part I—a theory of interprocess communication », in : *Journal of the ACM (JACM)* 33.*2* (1986), p. 313-326.

[Lap95] Jean-Claude Laprie, « Dependable Computing : Concepts, Limits, Challenges », en, in : (1995), p. 13.

[Lei+17] Vincent Leilde et al., « A Diagnosis Framework for Critical Systems Verification », in : *15th International Conference on Software Engineering and Formal Methods, SEFM 2017*, Proceedings of the 15th International Conference on Software Engineering and Formal Methods, SEFM 2017, Trente, Italy, sept. 2017, Short Papers 1-6.

[Lem+16] Pirmin Lemberger et al., *Big data et machine learning : les concepts et les outils de la data science*, French, OCLC : 960196724, Malakoff : Dunod, 2016.

[Lib+05] Ben Liblit et al., « Scalable statistical bug isolation », in : *ACM SIGPLAN Notices* 40.*6* (2005), p. 15-26.

[Lin+17] Yun Lin et al., « Feedback-based debugging », in : *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, IEEE, 2017, p. 393-403.

[LPY97] Kim G. Larsen, Paul Pettersson et Wang Yi, « Uppaal in a nutshell », en, in : *International Journal on Software Tools for Technology Transfer* 1.*1-2* (déc. 1997), p. 134-152, issn : 1433-2779.

[Mac90] John L. Mackie, *The cement of the universe : a study of causation*, eng, 5. Dr., Clarendon library of logic and philosophy, OCLC : 258760915, Oxford : Clarendon Press, 1990.

[Mag+00] Jeff Magee et al., « Graphical animation of behavior models », in : *Proceedings of the 22nd international conference on Software engineering*, ACM, 2000, p. 499-508.

[Mao08] Shahar Maoz, « Model-based traces », in : *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2008, p. 109-119.

[May02] Richard E. Mayer, « Rote Versus Meaningful Learning », en, in : *Theory Into Practice* 41.*4* (nov. 2002), p. 226-232, issn : 0040-5841, 1543-0421.

[Mer04] Merriam-Webster, *Merriam-Webster's Collegiate Dictionary : Eleventh Edition*, en, Google-Books-ID : TAnheeIPcAEC, Merriam-Webster, 2004.

[Mil06] Alain Mille, « From case-based reasoning to traces-based reasoning », en, in : *Annual Reviews in Control* 30.*2* (jan. 2006), p. 223-232, issn : 13675788.

[MLL05] Michael Martin, Benjamin Livshits et Monica S. Lam, « Finding application errors and security flaws using PQL : a program query language », in : *ACM SIGPLAN Notices*, t. 40, ACM, 2005, p. 365-383.

[MR11] Peter Müller et Joseph N. Ruskiewicz, « Using debuggers to understand failed verification attempts », in : *International Symposium on Formal Methods*, Springer, 2011, p. 73-87.

[MT98] Lynette I. Millett et Tim Teitelbaum, « Slicing Promela and its applications to model checking, simulation, and protocol understanding », in : *Proceedings of the 4th International SPIN Workshop*, Citeseer, 1998, p. 75-83.

[MTZ16] Faiz UL Muram, Huy Tran et Uwe Zdun, « Counterexample Analysis for Supporting Containment Checking of Business Process Models », in : (2016).

[New+14] Chris Newcombe et al., « Use of formal methods at Amazon Web Services », in : *See http ://research. microsoft. com/en-us/um/people/lamport/tla/formal-methods-amazon. pdf* (2014).

[NR69] Peter Naur et Brian Randell, éd., *Software Engineering : Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1969.

[Nus01] Bashar Nuseibeh, « Weaving Together Requirements and Architectures », in : *Computer* 34 (avr. 2001), p. 115-119.

[OD17] Fadi Obeid et Philippe Dhaussy, « Model Checking of Security Patterns Implementation : Application to SCADA », en, in : (2017), p. 3.

[Pel07] Radek Pelánek, « BEEM : Benchmarks for explicit model checkers », in : *Model Checking Software*, Springer, 2007, p. 263-267.

[Pen02] Yannick Pencolé, « Diagnostic décentralisé de systèmes à événements discrets : application aux réseaux de télécommunications », fr, thèse de doct., Université Rennes 1, juin 2002.

[Pet81] James Lyle Peterson, *Petri Net Theory and the Modeling of Systems*, Upper Saddle River, NJ, USA : Prentice Hall PTR, 1981.

[PMT04] Luigi Portinale, Diego Magro et Pietro Torasso, « Multi-modal diagnosis combining case-based and model-based reasoning : a formal and experimental analysis », en, in : *Artificial Intelligence* 158.*2* (oct. 2004), p. 109-153, issn : 00043702.

[Pnu77] Amir Pnueli, « The temporal logic of programs », in : *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, Providence, RI, USA : IEEE, sept. 1977, p. 46-57.

[PT09] Guillaume Pothier et Éric Tanter, « Back to the Future : Omniscient Debugging », in : *IEEE Software* 26.*6* (nov. 2009), p. 78-85, issn : 0740-7459.

[PW03] Bernhard Peischl et Franz Wotawa, « Model-based diagnosis or reasoning from first principles », in : *IEEE Intelligent Systems* 18.*3* (2003), p. 32-37.

[PW04] Bernhard Peischl et Franz Wotawa, « Towards a Model for Automated Fault Localization in VHDL Designs : Exploring Counterexample-Traces Using a Model-Based Diagnosis Approach », in : (août 2004).

[Qui86] J. Ross Quinlan, « Induction of decision trees », in : *Machine learning* 1.*1* (1986), p. 81-106.

[Qui93] J. Ross Quinlan, *C4.5 : Programs for Machine Learning*, San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1993.

[RB03] Theo C. Ruys et Ed Brinksma, « Managing the verification trajectory », in : *International Journal on Software Tools for Technology Transfer (STTT)* 4.*2* (fév. 2003), p. 246-259, issn : 1433-2779, 1433-2787.

[RB98] Theo C. Ruys et Ed Brinksma, « Experience with literate programming in the modelling and validation of systems », in : *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 1998, p. 393-408.

[Rei87] Raymond Reiter, « A theory of diagnosis from first principles », in : *Artificial intelligence* 32.*1* (1987), p. 57-95.

[Rib+14] Vincent Ribaud et al., « Techniques and Challenges for Trace Processing from a Model-Checking Perspective », in : *CISSE 2014*, 2014.

[RR03] Manos Renieres et Steven P. Reiss, « Fault localization with nearest neighbor queries », in : *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, IEEE, 2003, p. 30-39.

[RS03] and C. R. Ramakrishnan et S. A. Smolka, « Model checking and evidence exploration », in : *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2003. Proceedings.* Avr. 2003, p. 214-223.

[SA97] Ramakrishnan Srikant et Rakesh Agrawal, « Mining generalized association rules », in : *Future Generation Computer Systems*, Data Mining 13.*2* (nov. 1997), p. 161-180, issn : 0167-739X.

[SB82] William Swartout et Robert Balzer, « On the Inevitable Intertwining of Specification and Implementation », in : *Commun. ACM* 25.*7* (juil. 1982), p. 438-440, issn : 0001-0782.

[Sch+13] Markus Schumacher et al., *Security Patterns : Integrating Security and Systems Engineering*, en, Google-Books-ID : 2ZwSAAAAQBAJ, John Wiley & Sons, juil. 2013.

[Sch01] Steve Schneider, *The B-Method*, English, 2001 edition, Basingstoke, Hampshire : Red Globe Press, oct. 2001.

[Sch86] Roger C. Schank, *Explanation Patterns : Understanding Mechanical and Creatively*, Hillsdale, NJ, USA : L. Erlbaum Associates Inc., 1986.

[Sha83] Ehud Y. Shapiro, *Algorithmic Program DeBugging*, Cambridge, MA, USA : MIT Press, 1983.

[Sho76] « Dedication », in : *Computer-Based Medical Consultations : Mycin*, sous la dir. d'Edward Hance Shortliffe, Elsevier, 1976, p. v.

[Sil+13] W. Silva et al., « Automatic property generation for formal verification applied to HDL-based design of an on-board computer for space applications », in : *2013 14th Latin American Test Workshop - LATW*, avr. 2013, p. 1-6.

[Sim73] Herbert A Simon, « The Structure of Ill Structured P coblems », en, in : *Artificial Intelligence* (1973), p. 21.

[SN71] Herbert A. Simon et Allen Newell, « Human problem solving : The state of the theory in 1970. », in : *American Psychologist* 26.*2* (1971), p. 145.

[Som11] Ian Sommerville, *Software engineering*, 9th ed, OCLC : ocn462909026, Boston : Pearson, 2011.

[SP01] Natasha Sharygina et Doron Peled, « A combined testing and verification approach for software reliability », in : *International Symposium of Formal Methods Europe*, Springer, 2001, p. 611-628.

[Spa01] Luca Spalzzi, « A Survey on Case-Based Planning », en, in : *Artificial Intelligence Review* 16.*1* (sept. 2001), p. 3-36, issn : 0269-2821, 1573-7462.

[SR04] Shaochun Xu et V. Rajlich, « Cognitive process during program debugging », en, in : *Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.* Victoria, BC, Canada : IEEE, 2004, p. 176-182.

[Swe+90] John Sweller et al., « Cognitive Load as a Factor in the Structuring of Technical Material », in : *Journal of Experimental Psychology : General* 119.*2* (1990), p. 176-192.

[TLD14] Ciprian Teodorov, Luka Leroux et Philippe Dhaussy, « Context-aware verification of a cruise-control system », in : *Model and Data Engineering*, Springer, 2014, p. 53-64.

[Vis+03] Willem Visser et al., « Model checking programs », in : *Automated Software Engineering* 10.*2* (2003), p. 203-232.

[Vis09] Willemien Visser, « La conception : de la résolution de problèmes à la construction de représentations [Design : From problem solving to the construction of representations] », in : *Le travail humain* 72 (2009).

[VRK03] Venkat Venkatasubramanian, Raghunathan Rengaswamy et Surya N Kavuri, « A review of process fault detection and diagnosis », en, in : *Computers & Chemical Engineering* 27.*3* (mar. 2003), p. 313-326, issn : 00981354.

[VVV17] Simon Van Mierlo, Yentl Van Tendeloo et Hans Vangheluwe, « Debugging Parallel DEVS », en, in : *SIMULATION* 93.*4* (avr. 2017), p. 285-306, issn : 0037-5497.

[Wei84] Mark Weiser, « Program Slicing », in : *IEEE Transactions on Software Engineering* SE-10.*4* (juil. 1984), p. 352-357, issn : 0098-5589.

[WL11] Grafton Whyte et Donovan Lindsay Mulder, « The Impact of Software Test Constraints on Software Test Effectiveness », in : jan. 2011.

[Won+16] W. Eric Wong et al., « A Survey on Software Fault Localization », in : *IEEE Transactions on Software Engineering* 42.*8* (août 2016), p. 707-740, ISSN : 0098-5589, 1939-3520.

[XR04] Shaochun Xu et Vaclav Rajlich, « Cognitive process during program debugging », in : *Cognitive Informatics, 2004. Proceedings of the Third IEEE International Conference on*, IEEE, 2004, p. 176-182.

[Zel02] Andreas Zeller, « Isolating Cause-Effect Chains from Computer Programs », en, in : *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* (2002), p. 13.

[Zel06] Andreas Zeller, *Why programs fail : a guide to systematic debugging*, OCLC : ocm60492044, Amsterdam ; Boston : Heidelberg : Elsevier/Morgan Kaufmann ; dpunkt.verlag, 2006.

**Titre :** Aide au diagnostic de vérification formelle de systèmes

**Mot clés :** Model checking, Diagnostic, Taxonomie de Bloom, Méthode de résolution de problèmes

**Résumé :** Le model checking est une technique de vérification formelle qui consiste à certifier que le comportement d'un système formel satisfait des propriétés formelles. Son principe est d'explorer l'ensemble des exécutions possibles du système pour découvrir des chemins d'exécution (traces) violant les propriétés. Si c'est le cas, l'ingénieur doit remonter aux causes qui ont produit la trace. L'objectif de la thèse est d'assister l'ingénieur lors de cette phase que l'on appelle diagnostic. Nous proposons un cadre combinant différents types de connaissances et activités cognitives, supporté par une méthode et une infrastructure. Nous illustrons l'approche sur la sécurisation d'un système SCADA.

Quand le diagnosticien est vérificateur du modèle, il doit faire face à des traces de grande taille. Il réalise un diagnostic en mobilisant une multitude d'activités cognitives complexes. Pour les outiller, nous proposons une classification de ces activités selon la taxonomie de Bloom. Quand la cause réelle opère sur des connaissances autres que celles du model checking, ces moyens sont alors insuffisants.

Quand le diagnosticien est le concepteur du modèle, il dispose ou non de connaissances de domaine permettant de le débloquer en lui offrant des nouveaux regards sur la trace. Pour y parvenir, il faut disposer du domaine et corréler les connaissances du domaine et du model checking pour réduire leur fossé sémantique. Nous proposons des structures pour capturer et réutiliser le domaine. D'un côté le *problem case* formule le problème que l'on cherche à résoudre et permet de préciser le diagnostic de la solution construite. D'un autre côté les *sample*, *pattern* et *component cases* capturent des éléments de solutions et permettent d'isoler le diagnostic.

Quand le diagnosticien est l'architecte du système, il combine des éléments de problèmes et de solutions provenant à la fois de l'ingénierie du domaine et de l'application. Pour progresser de manière fluide dans la solution et enrichir les propriétés à vérifier, nous proposons une méthode de résolution de problème. Alimentée par la base de connaissances issue du domaine, celle-ci réalise des allers-retours entre l'espace du problème et l'espace de la solution, traçant problèmes et solutions choisies, et augmentant la vérification et le diagnostic grâce à de nouvelles propriétés.

De manière transversale aux autres phases, le processus de vérification doit être organisé. Nous proposons une infrastructure permettant d'organiser, capitaliser et réutiliser les diverses connaissances (model checking, domaine, méthode). L'infrastructure est divisée en trois niveaux, le niveau physique regroupe les données brutes, le niveau connaissance regroupe des ontologies, et le niveau d'accès fournit des interactions supportées par les connaissances, dont les activités cognitives de diagnostic, organisées suivant la taxonomie de Bloom. Nous proposons un outil de simplification de traces par facettes reposant sur cette infrastructure.

**Title:** A diagnosis support for formal verification of systems

**Keywords:** Model checking, Diagnosis, Bloom's taxonomy, Problem solving method

**Abstract:** Model checking is a formal verification technique verifying that a system behavior satisfies formal properties. This technique explores all the possible executions of the system to discover execution paths (traces) violating formal properties. When a property is violated, the engineer must find the root causes that produced the trace. The goal of this work is to assist the engineer during this phase, which is called *diagnosis*. Our proposition is a framework combining various kinds of knowledge and cognitive activities, supported by a method and an infrastructure. We apply the approach to securing a SCADA system.

When the diagnostician is the model verifier, he generally faces large traces. The diagnosis is obtained by mobilizing a multitude of complex cognitive activities. To tool these activities, we propose to classify them according to the Bloom's taxonomy. Even so, these means are insufficient when the real cause doesn't operate on model checking knowledge.

When the diagnostician is the model designer, he may or may not have domain knowledge. This knowledge offers new perspectives about the trace, and may unblock the diagnosis. We propose a structure to capture and reuse this knowledge. Correlations can be explicited to reduce the semantic gap between domain and model checking knowledge. On the one hand, the *problem case* formulates the problem to be solved, and precises the diagnosis of the constructed solution. On the other hand, the *sample*, *pattern* and *component cases* capture elements of solutions, and isolate the diagnosis.

When the diagnostician is the system architect, he combines elements of problems and solutions from both domain and application engineering. To progress smoothly in the solution and enrich the properties to be checked, we propose a problem solving method. Fed by the domain knowledge base, the method goes back and forth between the problem space and the solution space, tracing chosen problems and solutions, and facilitating verification and diagnosis thanks to new properties.

We propose an infrastructure to organize, share and reuse various knowledge implied during the verification process (model checking, domain, method). This infrastructure is divided into three levels, the physical level gathers the raw data, the knowledge level gathers ontologies, and the access level provides interactions supported by knowledge. We offer a trace simplification tool diagnosis based on facets, and promoted by our infrastructure.