Vérification formelle des propriétés graphiques des systèmes informatiques interactifs

Pascal Béger

▶ **To cite this version:**

Pascal Béger. Vérification formelle des propriétés graphiques des systèmes informatiques interactifs. Interface homme-machine [cs.HC]. INSA Toulouse, 2020. Français. NNT : . tel-02990362v1

HAL Id: tel-02990362

https://theses.hal.science/tel-02990362v1

Submitted on 5 Nov 2020 (v1), last revised 12 Apr 2022 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - ShareAlike 4.0 International License

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *30/10/2020* **par :**

PASCAL BÉGER

Vérification formelle des propriétés graphiques des systèmes informatiques interactifs

---

JURY

SOPHIE DUPUY-CHESSA	Professeure des Universités, LIG	Présidente du Jury
DOMINIQUE MÉRY	Professeur des Universités, LORIA	Rapporteur
CHARLOTTE SEIDNER	Maître de conférences, LS2N	Examinatrice
SÉBASTIEN LERICHE	Professeur, ENAC	Directeur de thèse
DANIEL PRUN	Enseignant-Chercheur, ENAC	Co-directeur de thèse

École doctorale et spécialité :
EDSYS : Informatique 4200018

Unité de Recherche :
Laboratoire d'Informatique Interactive (ENAC, Université de Toulouse, France)

Directeur(s) de Thèse :
Sébastien Leriche et *Daniel Prun*

Rapporteurs :
Sophie Dupuy-Chessa et *Dominique Méry*

RÉSUMÉ

Les systèmes critiques, particulièrement aéronautiques, contiennent de nouveaux dispositifs hautement interactifs. Dans ce contexte, les processus de certification décrits dans la DO-178C offrent une place importante à la vérification formelle des exigences de ces systèmes. Cependant, il est difficile avec les méthodes formelles actuelles de vérifier le respect des exigences concernant les éléments graphiques d'une interface telles que la couleur, la superposition, etc. De ce fait, notre objectif est de proposer une approche pour l'expression et la vérification formelle des exigences relatives à la scène graphique des interfaces humain-machine afin de profiter des apports des méthodes formelles dans un processus de développement.

Nous avons identifié un premier ensemble d'opérateurs graphiques de base nous permettant de décrire formellement des exigences graphiques. Le langage de programmation réactive Smala, supportant les éléments du format graphique SVG, est notre point d'entrée pour la mise en œuvre de cette étude. En effet, ce langage permet de décrire et d'animer une scène graphique en fonction d'événements d'entrée divers et variés (clic souris, compteur, ordre vocal, etc.). Nous avons conçu et développé un algorithme qui, par analyse statique du graphe de scène enrichi des applications Smala, permet de vérifier des propriétés graphiques exprimées préalablement avec notre formalisme. Le résultat est un système d'équations sur les variables d'entrée du système pour lesquelles la propriété vérifiée est vraie. Ce système d'équations peut alors être résolu par un outil d'analyse symbolique ou par simulation numérique.

Comme cas d'étude de nos travaux, nous utilisons le TCAS (Traffic alert and Collision Avoidance System), un système aéronautique ayant pour objectif d'améliorer la sécurité aérienne. Par l'intermédiaire de GPCheck, l'outil implémentant notre algorithme, pour chaque propriété graphique attendue, nous construisons le système d'équations portant sur les variables d'entrée de l'interface.

ABSTRACT

Critical systems, particularly in aeronautics, contain new, highly interactive devices. In this context, the certification processes described in DO-178C offer an important place for formal verification of the requirements of these systems. However, it is difficult with current formal methods to ensure requirements for graphical elements of an interface such as color, overlay, etc., which are not always easy to verify. Therefore, our objective is to propose an approach for the expression and formal verification of requirements for the graphical scene of human-machine interfaces in order to take advantage of the contributions of formal methods in a development process.

We have identified a first set of basic graphical operators allowing the verifier to formally describe graphical requirements. The Smala reactive programming language, supporting elements of the SVG graphical format, is our entry point for the implementation of this study. Indeed, this language allows the developer to describe and animate a graphic scene according to various input events (mouse click, counter, voice command, etc.). We have developed an algorithm which, by static analysis of the enriched scene graph with Smala applications, allows the verifier to verify graphical properties previously expressed with our formalism. The result is a system of equations on the input variables of the system for which the verified property is true. This system of equations can then be solved by a symbolic analysis tool or by numerical simulation.

As a case study of our work, we use the TCAS (Traffic alert and Collision Avoidance System), an aeronautical system whose objective is to improve air safety. Through GPCheck, the tool implementing our algorithm, for each expected graphical property, we build the system of equations dealing with the input variables of the interface.

REMERCIEMENTS

À l'issue de ce travail de thèse, je tiens à remercier chaleureusement toutes les personnes ayant contribué à le rendre possible.

Je tiens tout d'abord à remercier mes deux directeurs de thèse, Sébastien Leriche et Daniel Prun. Je les remercie pour leur confiance tout au long de ces trois années de thèse et pour leur pédagogie pendant tout ce processus de découverte et d'apprentissage du monde de la recherche. Cette expérience a été enrichissante et a existé en premier lieu grâce à eux.

Je remercie très sincèrement l'ensemble des membres de mon jury qui ont accepté d'évaluer ce travail. Ainsi, je remercie Sophie Dupuy-Chessa et Dominique Méry, rapporteurs de ce manuscrit, et Charlotte Seidner, examinatrice de ce travail.

Je remercie les membres de l'Équipe Informatique Interactive de l'ENAC pour l'accueil et les moments de partage qui ont contribué à ce travail.

Je remercie tout particulièrement les doctorants du bureau C117 (passés, présents, futurs), Valentin Becquet (docteur à l'heure de l'écriture de ces lignes), Alice Martin, Nicolas Nalpon et Florine Simon, qui m'ont permis de décompresser, plaisanter, avancer et qui ont été le moteur des conditions les plus favorables pour le bon déroulement de cette thèse.

Enfin, je remercie ma famille qui m'aura accompagné dans cette aventure.

À mon père

TABLE DES MATIÈRES

1 Introduction **1**

1.1 Systèmes informatiques interactifs 2

1.1.1 Définition 2

1.1.2 Les systèmes informatiques interactifs critiques 2

1.2 Vérification formelle 3

1.3 Questions de recherche 5

1.3.1 Comment formaliser les éléments de la scène graphique des interfaces humain-machine ? 7

1.3.2 Comment mécaniser la vérification des exigences graphiques des systèmes informatiques interactifs ? 7

1.4 Plan du manuscrit 8

2 Concepts de base **11**

2.1 Paradigmes de programmation 12

2.1.1 Paradigme impératif 12

2.1.2 Paradigme fonctionnel 13

2.1.3 Paradigme objet 14

2.1.4 Paradigme réactif 15

2.1.5 Synthèse 16

2.2 Propriétés liées à l'interaction 17

2.2.1 Comportement de l'utilisateur 18

2.2.2 Principes cognitifs 21

2.2.3 Interfaces humain-machine 22

2.2.4 Sécurité 26

2.2.5 Modélisation des systèmes interactifs 28
2.2.6 Synthèse 28
2.3 Nomenclature des formalismes 28
2.3.1 Algèbre de processus 29
2.3.2 Langage de spécification 30
2.3.3 Processus de raffinement 30
2.3.4 Système de transition d'états 32
2.3.5 Logique temporelle 33
2.3.6 Synthèse 35
2.4 Synthèse 35

3 État de l'art **37**
3.1 Objectif de ce chapitre 38
3.2 Revue de la littérature 38
3.2.1 Méthodologie 38
3.2.2 Propriétés liées au comportement de l'utilisateur 39
3.2.3 Propriétés liées aux principes cognitifs 42
3.2.4 Propriétés liées aux interfaces humain-machine 42
3.2.5 Propriétés liées à la sécurité 44
3.2.6 Modélisation des systèmes interactifs 45
3.2.7 Synthèse 50
3.3 Analyse de la littérature 50
3.3.1 Grande proportion de travaux sur la modélisation des systèmes 52
3.3.2 Utilisation privilégiée de certains formalismes 52
3.3.3 Émergence de nouveaux formalismes "ad hoc" 52
3.3.4 Maturité des cas d'étude 53
3.3.5 Propriétés de la scène graphique peu étudiées 54
3.4 Problématique abordée par cette thèse 55
3.4.1 Comment formaliser les éléments de la scène graphique des interfaces humain-machine ? 55
3.4.2 Comment mécaniser la vérification des exigences graphiques des systèmes informatiques interactifs ? 56

4 Cas d'étude : Smala et TCAS **57**
4.1 Contexte des travaux de thèse 58
4.2 Smala 59

4.2.1 Gestion du contrôle 59
4.2.2 Composants graphiques 61
4.2.3 Graphe d'activation 61
4.2.4 Lien entre composants graphiques et graphe d'activation . . 62
4.3 TCAS 64
4.3.1 Données d'entrée de l'interface 66
4.3.2 Exemple de scénario : Approche frontale 67
4.3.3 Exigences graphiques du TCAS 70
4.4 Synthèse 72

5 Vérification des propriétés graphiques 75
5.1 Processus de vérification 76
5.1.1 Processus de vérification complet 76
5.1.2 Présentation des actions du processus de vérification 79
5.2 Formalisme 88
5.2.1 Définitions préalables 88
5.2.2 Opérateurs graphiques 91
5.3 Algorithme de vérification 92
5.3.1 Définitions préalables 92
5.3.2 Algorithme 96
5.4 GPCheck : Implémentation de l'algorithme 112
5.4.1 Définition de la structure des nœuds du graphe de scène enrichi 112
5.4.2 smax_to_node : Génération des données d'entrée de Smala 114
5.4.3 fgp_to_node : Génération des données d'entrée des propriétés graphiques 116
5.4.4 node_to_java : Génération des données d'entrée finales . . . 118
5.4.5 find_proof : implémentation de l'algorithme 118
5.5 Synthèse 120

6 GPCheck : Application au TCAS 121
6.1 Rappels sur le TCAS 122
6.1.1 Données d'entrée de l'interface 122
6.1.2 Exigences graphiques du TCAS 123
6.2 GPCheck : Application au TCAS 124
6.3 Exigence E_1 : Propre position relative 125
6.3.1 Formalisation de l'exigence graphique 125

6.3.2 Vérification de l'exigence 126
6.3.3 Visualisation du résultat de la vérification 127
6.4 Exigence E_2 : Portée de 2 NM autour de *own aircraft* 128
6.4.1 Formalisation de l'exigence graphique 128
6.4.2 Vérification de l'exigence 128
6.5 Exigence E_3 : TA - Trafic de type *threat aircraft* 129
6.5.1 Formalisation de l'exigence graphique 129
6.5.2 Vérification de l'exigence 130
6.5.3 Visualisation du résultat de la vérification 130
6.6 Exigence E_4 : TA - Trafic de type *intruder aircraft* 131
6.6.1 Formalisation de l'exigence graphique 131
6.6.2 Vérification de l'exigence 132
6.6.3 Visualisation du résultat de la vérification 133
6.7 Exigence E_5 : TA - Trafic de type *proximate aircraft* 134
6.7.1 Formalisation de l'exigence graphique 134
6.7.2 Vérification de l'exigence 135
6.7.3 Visualisation du résultat de la vérification 135
6.8 Exigence E_6 : TA - Trafic de type *other aircraft* 136
6.8.1 Formalisation de l'exigence graphique 136
6.8.2 Vérification de l'exigence 137
6.8.3 Visualisation du résultat de la vérification 137
6.9 Exigence E_7 : TA - Altitude relative 138
6.9.1 Formalisation de l'exigence graphique 138
6.9.2 Vérification de l'exigence 139
6.9.3 Visualisation du résultat de la vérification 139
6.10 Exigence E_8 : TA - Sens vertical 140
6.10.1 Formalisation de l'exigence graphique 140
6.10.2 Vérification de l'exigence 141
6.10.3 Visualisation du résultat de la vérification 141
6.11 Synthèse 142

7 Conclusion et discussion **143**
7.1 Bilan des contributions de la thèse 144
7.1.1 Classification des propriétés liées à l'interaction 144
7.1.2 Vérification formelle des propriétés liées à l'interaction . . . 145
7.1.3 Développement d'un formalisme dédié à la scène graphique . 147

7.1.4 Développement d'un algorithme de vérification des propriétés graphiques 148
7.1.5 Développement de GPCheck, implémentation de l'algorithme de vérification 149
7.1.6 Vérification des exigences graphiques du TCAS 149
7.2 Discussion 151
7.2.1 Formalisme de Randell *et al.* [96] 151
7.2.2 Couverture de l'analyse 152
7.2.3 Adressage de la perception utilisateur 152
7.2.4 Dépendance à Smala 153
7.2.5 Utilisabilité de GPCheck 154
7.3 Perspectives 154
7.3.1 Formalisme de Randell *et al.* [96] 155
7.3.2 Couverture de l'analyse 156
7.3.3 Adressage de la perception utilisateur 157
7.3.4 Dépendance à Smala 158
7.3.5 Utilisabilité de GPCheck 159

CHAPITRE 1

INTRODUCTION

Depuis de nombreuses décennies, le développement des systèmes informatiques obéit à de nombreuses règles exprimées dans des référentiels méthodologiques, des guides de développement voire dans des normes. Pendant longtemps, ces référentiels méthodologiques ont accordé une large part aux méthodes de test pour vérifier que les systèmes obtenus fonctionnaient correctement et conformément à ce que l'utilisateur en attendait. Si les tests restent une méthode assez naturelle et facile à mettre en œuvre, elle n'en reste pas moins coûteuse et limitée quant à ses performances.

Plus récemment, des spécialistes ont développé des outils pour rendre plus sûre la vérification des exigences et du fonctionnement des systèmes informatiques : les méthodes formelles. Il s'agit généralement de formalismes mathématiques et d'outils associés qui permettent de vérifier certaines parties du système.

Pour des secteurs d'activité critiques tels que l'aéronautique et le médical, les méthodes formelles ont progressivement pris une place importante dans les processus de vérification. Cependant, ceci est vrai pour une majorité des propriétés des systèmes informatiques. Dans le cadre de cette thèse, nous verrons que les systèmes informatiques dits interactifs et en particulier les propriétés relatives à leur scène graphique n'ont pas bénéficié du même intérêt que les autres propriétés.

Dans ce premier chapitre, nous présentons une définition des systèmes informatiques interactifs. Nous présentons quelques exemples de ces systèmes, en particulier dans le cadre des systèmes critiques.

Nous posons ensuite les questions de recherches qui ont motivé ce travail de thèse. Ces deux questions concernent la vérification formelle des propriétés graphiques des systèmes informatiques interactifs.

1.1 Systèmes informatiques interactifs

Afin de comprendre les objectifs de cette thèse que nous avons intitulée "Vérification formelle des propriétés graphiques des systèmes informatiques interactifs", il est nécessaire de préalablement définir les différentes notions que ce titre contient. Cette section définit donc la notion de systèmes informatiques interactifs.

1.1.1 Définition

Les systèmes informatiques interactifs [18] sont des systèmes informatiques qui traitent des informations (clics de souris, entrées de données, etc.) provenant de leur environnement (autres systèmes ou humains) et produisent une représentation (notifications sonores, représentations visuelles, etc.) de leur état interne. Ces systèmes sont appelés interactifs du fait des interactions pouvant s'établir entre le système et l'utilisateur humain. Leur programmation est généralement basée sur le paradigme dit "réactif" [13] (l'exécution du programme dépend des événements et de leur propagation).

Historiquement, les développeurs pensaient les systèmes informatiques du point de vue de la fonction à réaliser. De nos jours, l'interface avec l'utilisateur est devenue centrale et elle occupe une place plus importante que la fonction réalisée. L'utilisabilité d'un système prime donc sur sa fonctionnalité. De ce fait, les systèmes informatiques interactifs sont devenus largement répandus dans différents secteurs critiques ou dans notre vie quotidienne à travers des applications mobiles, des distributeurs automatiques de billets, des interfaces présentes dans les habitacles des voitures, etc.

1.1.2 Les systèmes informatiques interactifs critiques

Nous entendons par "système critique" un système ayant un impact sur la santé ou la vie des humains présents dans son environnement. Il est donc crucial que le fonctionnement d'un tel système soit assuré et qu'il respecte toutes les règles de développement qui ont été préalablement définies.

Dans le secteur de l'aéronautique, nous pouvons trouver de tels systèmes dans les tours de contrôle ainsi que dans les centres de contrôle régionaux afin d'assurer l'écoulement du trafic aérien en toute sécurité. La recherche est encore en cours pour imaginer de nouvelles possibilités d'interaction par la réalité augmentée par

exemple [122]. Nous en trouvons également dans les cockpits d'avion et, au même titre que les tours de contrôle, des études sont en cours pour optimiser les interfaces des cockpits, par exemple en passant des interacteurs physiques (boutons, leviers, leds, etc.) aux interfaces tactiles [21, 20].

Dans le secteur médical, autre domaine critique, ces systèmes informatiques interactifs existent dans de nombreux dispositifs matériels comme par exemple les interfaces des pompes à infusion. Ce système agit directement sur la santé d'un patient en délivrant automatiquement des doses de produits préalablement sélectionnées.

Qu'il s'agisse du secteur médical ou du secteur aéronautique, le développement des systèmes informatiques critiques doit respecter un ensemble d'exigences afin d'être certifié et pour que les systèmes soient utilisés. Par exemple, le développement des systèmes informatiques de l'aéronautique doit démontrer le respect des normes comme la DO-178C [106].

Afin d'illustrer l'intérêt des méthodes formelles et leur pertinence pour le vérificateur, reprenons l'exemple des systèmes aéronautiques. Ils contiennent des dispositifs hautement interactifs : par exemple, les cockpits de nouvelle génération utilisent une électronique sophistiquée pilotée par des applications logicielles complexes et des interfaces graphiques riches. Ces systèmes sont utilisés dans des environnements critiques. Les normes ont pendant longtemps exigé l'utilisation de processus de tests afin de vérifier le respect des exigences par les systèmes informatiques. Cependant, cela n'était pas satisfaisant du fait des limites de ces processus de tests (coûts temporels et financiers, problème d'exhaustivité, risque élevé d'erreurs humaines).

De nos jours, pour aller au-delà des limites des processus de tests, les normes recommandent l'usage des méthodes formelles comme techniques de vérification. Dans ce contexte, les processus de certification décrits dans les normes aéronautiques telles que DO-178C [106] et DO-333 [107] accordent une place importante pour la vérification formelle des exigences de ces systèmes.

1.2 Vérification formelle

Dans la littérature, la vérification et la validation vont souvent ensemble. Elles peuvent être confondues car les nuances sont subtiles. Le guide PMBOK [2] donne les deux définitions suivantes pour ces deux notions :

- "Validation. The assurance that a product, service, or system meets the needs

of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers."

- "Verification. The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process."

Dans le cadre de cette thèse, nous nous intéressons à la vérification. Nous voulons donc uniquement nous assurer que les systèmes informatiques interactifs respectent les exigences des normes ou spécifications techniques qui leur sont applicables. Notre intérêt se porte donc sur les exigences techniques des systèmes.

Les méthodes formelles [127] sont des techniques basées sur les mathématiques ou la théorie des graphes par exemple. Grâce à des modèles mathématiques du futur système (élaborés à partir des spécifications) ou du code lui-même, elles permettent d'exprimer, d'analyser et de vérifier les propriétés de ces systèmes. Ces modèles peuvent être des formules mathématiques/logiques, des systèmes d'équations ou des graphes. Si le système est développé conformément au modèle alors la vérification des propriétés sur le modèle sera suffisante pour assurer la vérification de ces propriétés sur le système. Il faut donc assurer l'équivalence entre le modèle et le système.

Il existe différentes familles de méthodes de vérification formelle :

- La vérification basée sur la simulation de modèle [39] est une catégorie de méthodes basées sur l'élaboration de modèle état/transition du système où l'évaluation de propriétés logiques est effectuée au cours de chaque état atteint par simulation de ce modèle.
- La preuve [30] est une catégorie de méthodes basées sur l'élaboration d'un modèle décrit par un ensemble de variables, d'opérations, d'événements, de propriétés temporelles et d'invariants. Les opérations doivent préserver ces invariants et un ensemble d'autres propriétés (préconditions et/ou postconditions). Pour garantir l'exactitude de ces spécifications, un ensemble de preuves est généré et doit être prouvé.
- L'interprétation abstraite [50] consiste à donner une sémantique abstraite d'un langage de programmation ou d'une spécification afin d'effectuer des opérations sur la représentation abstraite : vérification des propriétés, optimisation, etc.
- L'analyse statique [65] est la principale application concrète de l'interprétation abstraite. Elle est appliquée en analysant le code source des programmes et

en extrayant automatiquement une sémantique abstraite qui peut être utilisée pour vérifier les propriétés et effectuer des optimisations.

Dans cette thèse, nos travaux portent sur la vérification basée sur l'analyse statique du code. D'une part, un modèle mathématique du système est construit à partir du code logiciel. D'autre part, un modèle des propriétés à vérifier est construit à partir de la spécification. Par la suite, ces deux modèles sont confrontés pour s'assurer que le premier (celui élaboré à partir du code) est cohérent avec le second (élaboré à partir des propriétés attendues du système).

1.3 Questions de recherche

Qu'il s'agisse d'un système informatique interactif critique ou non, que l'on s'occupe du développement de l'interface du système ou du comportement du système, il faut suivre les besoins du client ou de l'utilisateur final du système. Généralement, un cahier des charges retranscrit cette demande ou ces besoins. Cependant, il faut également prendre en compte les besoins du domaine et donc pour certains secteurs, cela prendra la forme d'une norme ou d'un guide de bonnes pratiques. De plus, pour l'aspect graphique de l'interface, en plus de respecter les demandes du client, l'usage est de suivre des règles de bonne conception [119, 123].

Historiquement, les premières propriétés étudiées par les méthodes formelles pour les systèmes informatiques concernaient les propriétés liées aux calculs (e.g. bornitude) ainsi que la vivacité des programmes (e.g. retour à un état donné, absence d'interblocage) [92]. Depuis ces trois dernières décennies, de nombreuses méthodes formelles outillées ont été proposées pour le développement de systèmes informatiques sûrs et corrects d'un point de vue fonctionnel. Cependant, l'évolution de ces systèmes et l'apparition des interfaces humain-machine (IHM) modernes font émerger de nouvelles propriétés (comportement de l'utilisateur, sciences cognitives, propriétés spécifiques à l'interface et cyber-sécurité par exemple) qui challengent ces méthodes. Ces systèmes informatiques interactifs n'ont pas bénéficié de la même attention par les méthodes formelles : la vérification de leurs propriétés liées à l'interaction repose encore principalement sur des évaluations de prototypes au cours des tests avec les utilisateurs finaux.

Lorsque des techniques de vérification formelle sont utilisées pour les propriétés des systèmes interactifs, celles-ci ne proposent très souvent qu'une couverture partielle des caractéristiques de ces systèmes. Par exemple, les éléments issus de

l'interface humain-machine restent peu pris en compte.

En particulier, si on considère la classe des propriétés graphiques des systèmes informatiques interactifs, leur vérification repose encore généralement sur une étude manuelle du système, en utilisant des check-lists pour s'assurer que pour un scénario d'utilisation donné du système, la scène graphique répond aux exigences imposées. Il s'agit donc là de processus de tests. Cependant, la vérification par tests pose plusieurs problèmes :

- Il n'existe aucune certitude que l'ensemble des comportements du système vérifie l'exigence. En effet, rien ne garantit que les scénarios testés représentent l'ensemble des scénarios possibles du système. Bien que les testeurs appliquent ces processus de tests au cours de la conception, sur des prototypes ou sur les étapes intermédiaires du système informatique interactif développé, ainsi que sur le système final, en essayant d'atteindre certains objectifs de couverture (données d'entrées, instruction, structure de contrôles, etc.), il est généralement impossible de couvrir toutes les exécutions possibles.

- D'un point de vue pratique et financier, les processus de test sont coûteux car ils interviennent à la fin de la phase de développement. Suite à l'observation d'un dysfonctionnement, la recherche de l'origine du bug, sa correction et le processus de revérification qui s'ensuivent restent lourds. Plus la scène graphique d'un système informatique interactif est riche et dynamique, plus le temps de vérification par test sera important et par conséquent coûteux d'un point de vue financier.

- *Errare humanum est.* L'erreur est humaine. Cette maxime représente le troisième problème. Les tests en général, et particulièrement des propriétés graphiques, restent encore majoritairement un processus manuel, que ce soit au cours de leur élaboration, leur passage ou l'analyse des résultats obtenus. Ils constituent ainsi une activité qui reste peu performante en matière de temps. Ils laissent aussi la place aux erreurs de réalisation ainsi que d'interprétation des résultats.

Les propriétés relatives à la scène graphique peuvent paraître simples. Cependant, nous estimons que le gain en couverture, en temps et en réduction d'erreurs humaines, apporté par un processus de vérification automatique serait important.

1.3.1 Comment formaliser les éléments de la scène graphique des interfaces humain-machine ?

Au cours des dernières décennies, de nombreuses méthodes formelles ont été proposées pour le développement de systèmes informatiques interactifs sûrs et fonctionnellement corrects. Toutefois, lorsque des techniques de vérification formelle sont utilisées, elles permettent souvent de ne couvrir que partiellement les problèmes liés aux interfaces humain-machine [53, 79, 32]. D'une part, elles adressent largement les propriétés comportementales des systèmes. D'autre part, nous constatons que les propriétés liées à la spécificité des systèmes informatiques interactifs, par exemple celles spécifiques à la scène graphique, sont peu étudiées. Par exemple, il y a peu d'exemples avec les méthodes formelles actuelles permettant de garantir les exigences relatives aux éléments graphiques d'une interface humain-machine impliquant la forme, la couleur, la position dans l'interface, etc.

De plus, nous avons précisé que la conception de la scène graphique suit souvent une norme ou des règles de conception imposées [119, 123]. Or, l'expression formelle de ces propriétés est encore peu développée ce qui laisse la place à des erreurs d'interprétation.

Ainsi, formaliser les éléments de la scène graphique des interfaces humain-machine permettrait de :

- définir sans ambiguïté les éléments de la scène graphique et leurs propriétés,
- mécaniser la vérification des exigences graphiques des systèmes informatiques interactifs.

1.3.2 Comment mécaniser la vérification des exigences graphiques des systèmes informatiques interactifs ?

Il existe deux approches pour vérifier la conformité d'un système à une spécification technique :

- Appliquer la vérification sur un modèle de la spécification puis traduire le modèle en code du système (par exemple, "Un modèle B événementiel décrit un système réactif par un ensemble d'événements qui modifient un état. Le raffinement permet de bâtir ce modèle étape par étape en introduisant de nouveaux événements tout en préservant les propriétés établies." [7]).
- Appliquer la vérification sur un modèle du système obtenu à partir de son code

(par exemple, Frama-C [109] possède des outils d'analyse de code permettant de vérifier des propriétés de programmes écrits en C et son plugin WP utilise notamment la logique de Hoare pour vérifier des propriétés par annotation de code).

La première approche est contraignante du fait de la nécessité de produire une traduction du modèle en du code informatique pour implémenter le système. Cependant, même si le modèle vérifie les exigences de la spécification, il faut assurer que la traduction est sûre et que le code résultant est équivalent au modèle de la spécification. Nous verrons aussi que les quelques approches de vérification formelle des propriétés graphiques des systèmes informatiques interactifs suivent jusqu'à présent cette approche.

Nous privilégions la seconde approche. Celle-ci reste en effet plus simple car elle prend en entrée le code final du système qui est effectivement exécuté. De plus, cette approche restant extrêmement peu explorée pour les systèmes informatiques interactifs, son étude nous apparaît comme la plus pertinente.

Ainsi, analyser statiquement le code des systèmes informatiques interactifs permettrait de vérifier formellement et de manière plus sûre les exigences graphiques de ces systèmes le tout en assurant l'exhaustivité de la vérification grâce à l'exploration systémique de tous les scénarios possibles.

1.4 Plan du manuscrit

Chapitre 2. Concepts de base. Ce chapitre introduit les différents concepts qui permettent de préciser notre problématique sur la vérification formelle des propriétés graphiques des systèmes informatiques interactifs. Nous présentons les quatre paradigmes de programmation usuels (impératif, fonctionnel, objet, réactif) et leurs particularités spécifiquement dans le contexte de leur vérification. Nous présentons ensuite notre classification des propriétés relatives aux systèmes informatiques interactifs et liées à l'interaction. Nous présentons enfin notre nomenclature des formalismes.

Chapitre 3. État de l'art. Ce chapitre constitue une revue de la littérature visant à extraire et classer les propriétés liées à l'interaction et formalismes utilisés pour la vérification de ces propriétés. Son objectif principal est de répondre à trois questions :

- Quelles sont les propriétés liées à l'interaction étudiées ?

- Quels sont les formalismes utilisés pour ce faire ?
- Quels sont les principaux cas d'étude utilisés pour démontrer l'approche ?

La deuxième partie de ce chapitre est une analyse de la revue précédente. Notre objectif ici est de répondre à deux questions :

- Quelles sont les propriétés liées à l'interaction les plus couvertes et celles qui sont les moins adressées ?
- Quels sont les formalismes les plus utilisés pour l'expression et la vérification des propriétés des systèmes informatiques interactifs, et y en a-t-il des nouveaux qui ont émergé spécifiquement pour ce type de propriétés ?

En répondant à ces questions, nous justifions nos questions de recherche.

Chapitre 4. Cas d'étude : Smala et TCAS. Ce chapitre introduit dans un premier temps le contexte de ces travaux de thèse.
Tout d'abord, nous présentons le secteur de l'aéronautique qui est au cœur des problématiques de notre équipe de recherche. Ce secteur définit de nombreuses normes que les systèmes informatiques interactifs doivent absolument respecter. L'usage des méthodes formelles est conseillé pour vérifier que les systèmes respectent ces normes. Nous présentons ensuite le langage de programmation réactive Smala développé par notre équipe et qui sert de base à nos travaux.
Enfin, nous présentons le cas d'étude que nous utilisons : le TCAS (*(Trafic alert and Collision Avoidance System)*), un système de prévention des collisions aériennes dont l'affichage peut être intégré dans différents équipements du cockpit des aéronefs.

Chapitre 5. Vérification des propriétés graphiques. Ce chapitre présente le processus de vérification des propriétés graphiques que nous avons développé au cours ces travaux de thèse. Nous présentons le formalisme que nous avons développé afin d'exprimer formellement les exigences graphiques des systèmes informatiques interactifs. Nous présentons ensuite l'algorithme qui analyse les différents opérateurs graphiques que nous avons définis. Enfin, nous présentons GPCheck, l'outil de vérification des propriétés graphiques que nous avons développé et qui implémente notre algorithme.

Chapitre 6. GPCheck : Application au TCAS. Dans ce chapitre, nous appliquons notre processus et utilisons notre outil sur notre cas d'étude : le TCAS intégré à l'IVSI (*Instantaneous Vertical Speed Indicator*). Nous appliquons le processus de vérification aux exigences graphiques de ce système informatique interactif.

Chapitre 7. Conclusion et discussion. Dans ce chapitre, nous synthétisons les différents "findings" de cette thèse et rappelons les réponses que nous avons

apportées à chacun d'entre eux à travers nos contributions. Nous discutons ensuite des résultats obtenus et envisageons les pistes futures permettant de prolonger ces travaux.

CHAPITRE 2

CONCEPTS DE BASE

Historiquement, les premières propriétés étudiées par les méthodes formelles pour les logiciels et systèmes informatiques ont concerné la sûreté (e.g. absence d'événement indésirable, bornitude) ainsi que la vivacité des programmes (e.g. retour à un état donné, absence d'interblocage) [92].

Depuis lors, les principales familles de méthodes permettant la vérification formelle des propriétés des systèmes informatiques sont la vérification de modèle [39], la preuve mathématique [30], l'interprétation abstraite [50] ainsi que l'analyse statique [65]. Cependant, l'évolution de ces systèmes et l'apparition des interfaces humain-machine modernes font émerger de nouvelles propriétés liées à l'interaction qui challengent ces méthodes.

À travers cette thèse, nous étudions la formalisation et la vérification des propriétés des interfaces humain-machine, des systèmes informatiques interactifs avec des scènes graphiques riches. Pour ce faire, nous introduisons trois concepts de base :

- *le premier concerne le paradigme de programmation interactive (ou orienté interactions) et ses différences avec les autres paradigmes,*
- *le deuxième présente les différentes propriétés relatives aux systèmes informatiques interactifs et liées à l'interaction,*
- *le troisième porte sur les différents formalismes et méthodes de vérification formelle classiquement utilisées pour les propriétés des systèmes informatiques interactifs.*

2.1 Paradigmes de programmation

Il existe de nombreux paradigmes de programmation [38] permettant de fournir un cadre conceptuel pour la conception et le développement des applications informatiques. Ces paradigmes constituent autant de manières de penser le fonctionnement ou l'objectif d'une application. De par leur nature, ils mettent chacun en avant différents concepts sur lesquels reposent les langages de programmation et, par là même, orientent les techniques à utiliser pour l'expression et la vérification de leurs propriétés.

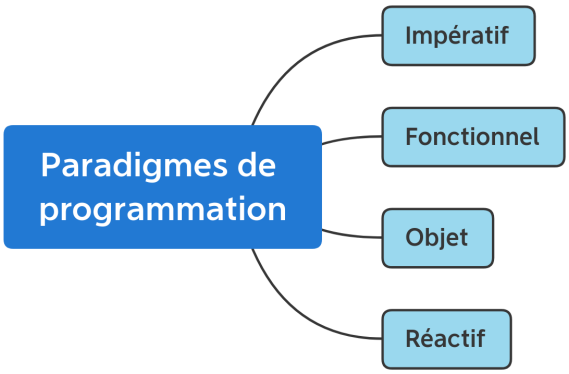


FIGURE 2.1 – Paradigmes de programmation usuels

Dans cette thèse, nous nous intéressons spécifiquement aux applications interactives, reposant principalement sur le paradigme dit "réactif". Afin de mettre en exergue les spécificités liées à ce paradigme, nous présentons les quatre paradigmes de programmation les plus usuels en détaillant leurs particularités : impératif, fonctionnel, objet et réactif (voire figure 2.1). Nous réalisons alors une synthèse où nous dégageons les spécificités du paradigme réactif.

2.1.1 Paradigme impératif

Le paradigme impératif consiste en l'écriture d'une séquence de commandes (les instructions) devant être réalisées par l'application et modifiant son état [56]. Des structures de contrôle (conditionnelle, boucle...) permettent de faire évoluer le flot d'exécution par des changements d'état (mémoire/variables et pointeur sur l'instruction courante). Cet état permet l'interruption sur l'exécution d'une instruction ou l'exécution contrôlée pas à pas [104].

De ce fait, le développement selon ce paradigme consiste en l'écriture d'une succession de commandes, détaillant les étapes devant être réalisées par l'application qui

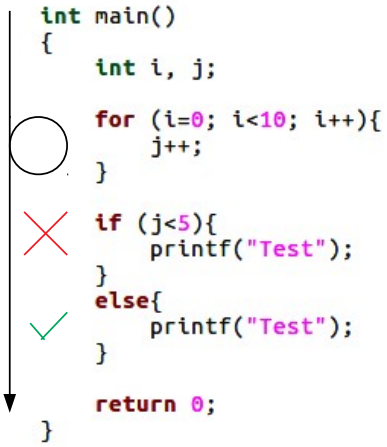


FIGURE 2.2 – Illustration de l'exécution d'un programme impératif

seront exécutées dans l'ordre de leur écriture en fonction des structures de contrôle (figure 2.2).

Dans ce paradigme, la vérification se focalise sur l'état de la mémoire, les données et leur type (respect des bornes des variables par exemple) ainsi que sur le contrôle (absence de boucles infinies par exemple).

C, Fortran et Pascal sont des exemples de langages basés sur le paradigme impératif.

2.1.2 Paradigme fonctionnel

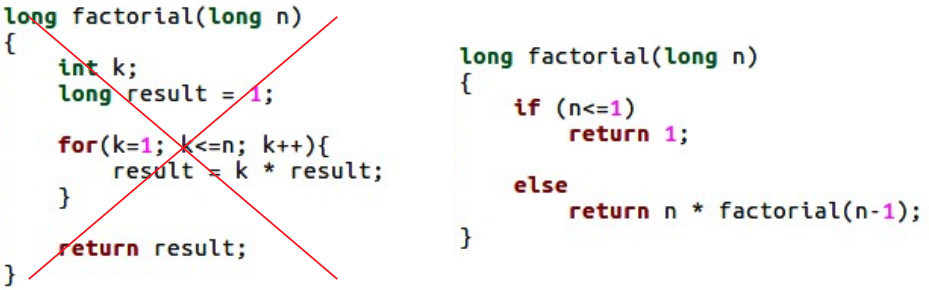


FIGURE 2.3 – Disparition des variables en langage fonctionnel

Un programme respectant le paradigme fonctionnel correspond à une fonction mathématique transformant son entrée en une sortie [121]. Il repose sur deux opérations fondamentales : la conversion, consistant en un renommage des paramètres (pouvant être également des fonctions) et la réduction, représentant les appels de fonctions. Les entrées et sorties des fonctions décrivent le flot d'exécution. Ainsi,

une application n'est plus décrite telle qu'elle doit s'exécuter, mais par le biais de l'objectif ou du résultat attendu par l'application ou ses différentes fonctionnalités.

En fonctionnel pur, les effets de bord sont interdits, l'affectation de valeur à des variables globales n'existant pas [121]. Le paradigme fonctionnel est donc sans état, ce qui simplifie la mise au point du programme puisqu'il est possible de tester et valider les fonctions indépendamment les unes des autres. Par rapport au paradigme impératif, il est important de noter que la notion de variable disparaît. La figure 2.3 montre ceci en présentant le fait que l'on utilise un appel récursif de la méthode sans passer par des variables intermédiaires.

Dans ce paradigme, les vérifications portent souvent sur la propriété de terminaison, les empilements des appels de fonctions (déterminant l'utilisation de la pile).

Caml, Haskell et F# sont des exemples de langages basés sur le paradigme fonctionnel.

2.1.3 Paradigme objet

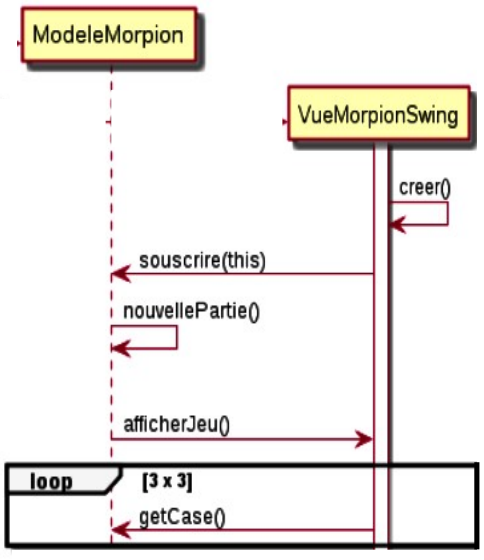


FIGURE 2.4 – Appels de méthodes entre objets

La programmation orientée objet est basée sur l'abstraction des données [52] de façon à obtenir une représentation proche des objets manipulés dans la réalité. Les concepts clés [74] sont une structuration du code et des données au sein de classes (structure contenant les méthodes et attributs communs aux objets instanciant une même classe), isolant ces données par encapsulation. L'héritage favorise la

réutilisation, le polymorphisme permet de manipuler des objets dont les types sont compatibles (issus d'une hiérarchie de classes). Les attributs d'un objet définissant son état interne, la programmation objet est de type avec état.

Afin de développer une application avec ce paradigme, il faut raisonner en matière d'objets s'échangeant des messages par appels de méthodes [89]. Le flot d'exécution d'une application orientée objet suit donc les appels de méthodes entre objets. Par exemple dans le programme représenté par la figure 2.4, *VueMorpionSwing* appelle sa propre méthode *creer()*, puis *VueMorpionSwing* appelle la méthode *souscrire()* de *ModeleMorpion* et ainsi de suite.

Les propriétés intéressant la vérification sont par exemple le respect du principe d'encapsulation, des relations entre objets et de la causalité des appels de méthodes.

Java et C# sont des exemples de langages basés sur le paradigme orienté objet.

2.1.4 Paradigme réactif

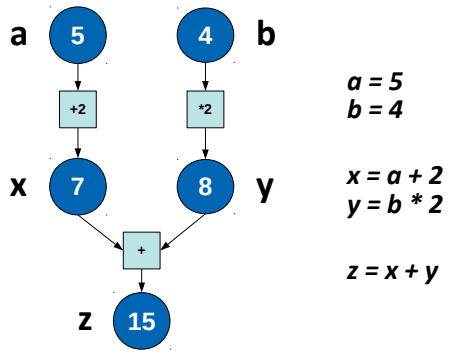


FIGURE 2.5 – Exécution d'un programme réactif dirigé par le flot de données

Dans une application respectant le paradigme réactif, des événements provenant de sources internes (horloge...) ou externes (clic de souris, appui sur le clavier, sortie d'un capteur...) déclenchent un comportement programmé. Ces événements se propagent en mettant à jour leurs dépendances par le concept de suite d'activation. Ainsi, la modification d'une variable peut provoquer la mise à jour d'autres variables et l'exécution de comportements qui en dépendent. Par exemple dans le programme représenté par la figure 2.5, si un événement met à jour les variables a et b ailleurs dans le programme, le programme exécutera à nouveau cet arbre et mettra ainsi à jour les dépendances.

L'exécution d'un programme réactif suit le flot de données, les événements extérieurs et la propagation automatique des changements se produisant en fonction des

dépendances entre les composants [13]. La propagation d'événements, à la base de ce paradigme, induit une définition sans état de la programmation réactive. L'exécution suivant le flot de données, il est nécessaire de définir un arbre de dépendances entre les différents composants de l'application.

Le respect des dépendances, le suivi du flot de données et de l'activation sont des propriétés pertinentes à vérifier pour les applications réactives.

Java FX et Scala.React [84] sont des exemples de langages utilisant le paradigme réactif.

2.1.5 Synthèse

La vérification formelle de propriétés des systèmes informatiques interactifs est notre objectif de base. Nous avons donc volontairement ciblé notre analyse (tableau 2.1) de manière à extraire les caractéristiques spécifiques à chaque paradigme. Ces caractéristiques spécifiques peuvent avoir un impact sur les techniques de vérification utilisées, notamment lorsqu'il s'agit d'analyse de code ou de la structure de l'implémentation de l'application.

Paradigme	Impératif	Fonctionnel	Orienté objet	Réactif
Flot d'exécution	Itératif, structure de contrôle	Suivi I/O fonctions	Contrôle, séquences appels méthodes	Données, arbre de dépendance
État	Avec état	Sans état	Avec état	Sans état
Définition de l'état	Variables en mémoire, locus de contrôle	*N/A*	État interne des objets (attributs) en mémoire	*N/A*
Propriétés pertinentes	État mémoire, bornes variables, boucles finies	Terminaison	Encapsulation, causalité appels méthodes	Dépendances, suivi flot de données et activation
Exemples de langages	Fortran, Pascal, C	Caml, Haskell,	Java, C#	Java FX, Scala.React

TABLE 2.1 – Synthèse des particularités des paradigmes de programmation

Dans le cadre de cette thèse, nous nous intéressons aux applications interactives basées sur le paradigme interactif. De ce fait, des techniques classiques d'analyse de code, généralement dédiées aux paradigmes impératif et fonctionnel, ne sont pas applicables. Dans le paradigme impératif, il est par exemple très facile de suivre le

flot d'exécution qui est plus ou moins linéaire et représentatif de son ordre d'écriture avec certains sauts autorisés par les structures de contrôle. En réactif, le flot d'exécution va varier en fonction des données d'entrée qui peuvent en plus provenir d'un utilisateur humain, ainsi qu'en fonction de l'arbre de dépendance des différents composants de l'application. L'analyse d'un tel système et de ses propriétés doit donc être différente de celles des systèmes plus classiques en étant centrée sur les dépendances entre composants.

2.2 Propriétés liées à l'interaction

Dans cette section, nous proposons une classification des propriétés liées à l'interaction.

L'objectif principal de cette classification pour cette thèse est de définir un premier ensemble représentatif de propriétés liées à l'interaction. Cela nous permet par la suite de vérifier la couverture de ces propriétés par les méthodes formelles. L'intérêt dans le cadre de cette thèse est de confirmer que les méthodes formelles couvrent peu certaines de ces propriétés.

Il n'existe pas une classification unique et actuellement aucune ne fait référence pour ces propriétés. De plus, nous constatons que plusieurs définitions des propriétés sont divergentes ou complémentaires.

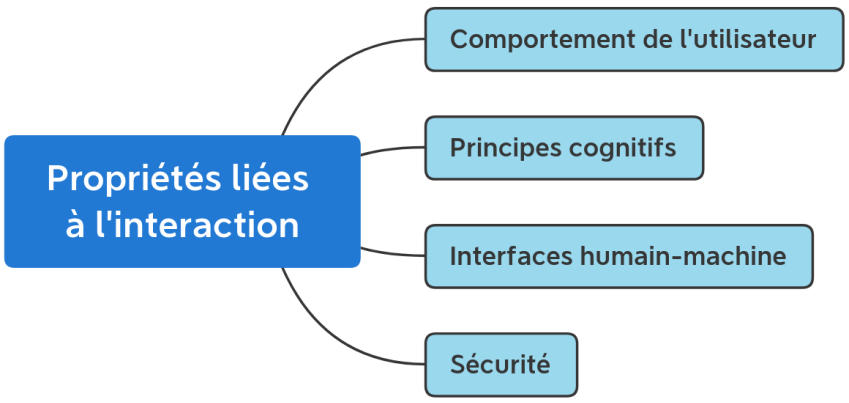


FIGURE 2.6 – Classes de propriétés liées à l'interaction

Nous avons donc choisi de classer ces nombreuses et nouvelles propriétés liées à l'interaction en quatre classes distinctes [23] (figure 2.6) que nous présentons ci-dessous : comportement de l'utilisateur [4], principes cognitifs [43], interfaces humain-machine [18] et sécurité [103].

Notre classification ne permet pas nécessairement de trier toutes les propriétés des systèmes informatiques interactifs. En effet, avant d'être interactifs, ceux-ci restent des systèmes informatiques. Aussi, nous n'avons pas pris en compte par exemple des propriétés liées aux calculs (e.g. bornitude) ou de vivacité des programmes (e.g. retour à un état donné, absence d'interblocage) [92].

2.2.1 Comportement de l'utilisateur

Cette classe regroupe les propriétés du système liées à la présence d'un utilisateur humain en interface [4]. Les propriétés de cette classe concernent les actions de l'utilisateur, les attentes de l'utilisateur par rapport au système, les objectifs et les restrictions de l'utilisateur.

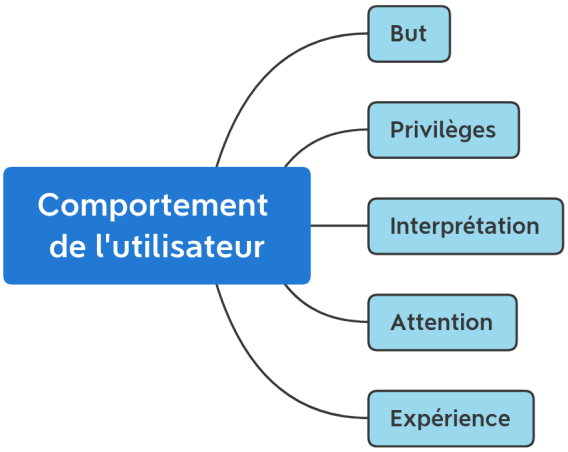


FIGURE 2.7 – Propriétés de la classe comportement de l'utilisateur

Au sein du comportement de l'utilisateur (figure 2.7), nous classons les propriétés suivantes : but de l'utilisateur [44], privilèges de l'utilisateur [46], interprétation de l'utilisateur [100], attention de l'utilisateur [116] et expérience de l'utilisateur [45].

2.2.1.1 But

Cerone et Elbegbayan [46] définissent le but de l'utilisateur comme les missions que l'utilisateur peut ou doit accomplir lorsqu'il utilise le système. Ces objectifs dépendent des motivations de l'utilisateur. Quelques années plus tard, Cerone [44] précise cette notion de but de l'utilisateur avec la notion d'action de but : un but est atteint lorsqu'une action de but est réalisée.

Rukšėnas *et al.* [98] définissent le but de l'utilisateur comme un modèle de plan partiel. Ce modèle est basé sur la connaissance de la tâche en cours d'exécution

lorsque l'utilisateur interagit avec le système et de tous les sous-objectifs qui doivent être atteints en fonction de cette tâche. Cette définition du but de l'utilisateur est complétée par Rukšėnas *et al.* [99]. Ils donnent les propriétés d'un but dont par exemple : la garde représentant l'activation du but ; le choix représentant la possibilité de choisir le but ; le prédicat atteint représente l'atteignabilité du but ; les sous-objectifs.

Un **but d'utilisateur** est donc une liste de sous-objectifs qu'un utilisateur doit réaliser pour atteindre un objectif plus important lié aux fonctionnalités du système utilisé.

Exemple. Prenons le but d'utilisateur "Retirer de l'argent à un distributeur automatique de billets". "Insérer la carte", "rentrer le code PIN" et "choisir le montant d'argent" sont des sous-objectifs de ce but d'utilisateur [82]. La garde pour atteindre le sous-objectif "choisir le montant d'argent" est ici "code PIN valide".

2.2.1.2 Privilèges

Cerone et Elbegbayan [46] définissent les privilèges de l'utilisateur comme les interactions que l'utilisateur peut ou ne peut pas effectuer avec le système en fonction de son statut dans le système. À partir de cette définition, les auteurs proposent de contraindre le système en modifiant la conception en fonction du statut de l'utilisateur. Ces changements de conception permettent au système de limiter les interactions disponibles afin d'appliquer des privilèges d'utilisateurs au sein du système.

Les **privilèges de l'utilisateur** sont donc un moyen d'empêcher un utilisateur ayant un niveau d'accréditation non autorisé d'atteindre des objectifs que l'utilisateur ne devrait pas atteindre.

Exemple. "Dans un système de gestion des articles de conférence, les utilisateurs ont des droits d'accès et d'action différents selon leur rôle (auteurs, réviseurs, organisateurs). Dans ce contexte, en fonction de leur rôle dans le système, les auteurs peuvent uniquement être en mesure de soumettre et de télécharger leur propre article, tandis que l'examinateur principal peut être en mesure de visualiser et de gérer plusieurs articles soumis par les auteurs." [62]

2.2.1.3 Interprétation

Rukšėnas et Curzon [100] présentent l'interprétation de l'utilisateur comme l'ensemble des hypothèses de l'utilisateur sur le système. Ces suppositions peuvent être

dues à l'expérience de l'utilisateur avec le système ou un système suffisamment proche. L'interprétation d'un système peut amener les utilisateurs à adapter leur comportement en fonction de leurs hypothèses.

Exemple. Nous sommes habitués au raccourci Ctrl+C afin de copier du texte. Un utilisateur novice d'un terminal pourrait l'utiliser pour copier du texte et fermer l'application en cours d'exécution parce que la fonctionnalité n'est pas la même, il s'agit là d'une mauvaise interprétation de ce raccourci.

2.2.1.4 Attention

Su *et al.* [116] s'intéressent aux limites de l'attention temporelle des utilisateurs humains principalement dans un scénario de test de présentation visuelle en série rapide. Ils rappellent que l'attention est liée à la signification des éléments affichés dans l'interface et à leur saillance. Ils précisent que le fait de s'occuper des *surprises* dans un schéma constant pourrait faire perdre l'attention de l'utilisateur.

Cerone [44] définit l'attention comme une activité de l'utilisateur qui doit se concentrer sur une caractéristique de l'environnement. L'utilisateur ignore les autres caractéristiques de l'environnement tout en se concentrant sur une caractéristique spécifique. L'utilisateur est plus susceptible de perdre son attention lors d'une exposition à des stimuli soudains non liés à l'activité en cours. Plus tard, Cerone [45] ajoute les définitions de l'attention implicite et explicite. D'une part, l'attention implicite est associée à des stimuli inattendus liés à l'état mental de l'utilisateur. D'autre part, l'attention explicite se réfère à l'attention portée aux stimuli pertinents pour la tâche en cours de réalisation.

L'**attention de l'utilisateur** est donc la capacité de l'utilisateur à se concentrer sur une activité spécifique sans être perturbé par des informations non pertinentes.

Exemple. Au volant d'une voiture, l'utilisation d'un GPS uniquement audio (une modalité perturbatrice) permet au conducteur de porter davantage d'attention à sa vitesse que l'utilisation d'un GPS audiovidéo (deux modalités perturbatrices) [77].

2.2.1.5 Expérience

Cerone et Zhao [47] expliquent que, dans le processus de conduite, les expériences de l'utilisateur influencent son comportement. Si l'utilisateur est confronté à un événement particulier, tel qu'un risque inattendu, dans une situation courante, il adaptera son comportement la prochaine fois qu'il sera confronté à la situation au cours de laquelle l'événement s'est produit. Ce phénomène fait référence à l'ap-

prentissage humain. Cerone [45] détaille ce processus d'apprentissage humain en trois phases : compréhension, pratique et expérience. La phase de compréhension permet à l'utilisateur d'apprendre la théorie sur le système. La phase de pratique permet à l'utilisateur d'acquérir des compétences. La phase d'expérience permet à l'utilisateur de modéliser mentalement le système.

L'**expérience utilisateur** concerne donc la connaissance, accumulée au fil du temps, de l'utilisateur sur le système.

Exemple. L'exemple donné dans l'interprétation de l'utilisateur illustre également l'expérience de l'utilisateur : un utilisateur expérimenté d'un terminal ne commettrait pas de mauvaise interprétation sur le raccourci Ctrl+C.

2.2.2 Principes cognitifs

Cette classe regroupe les propriétés liées aux sciences cognitives [43].

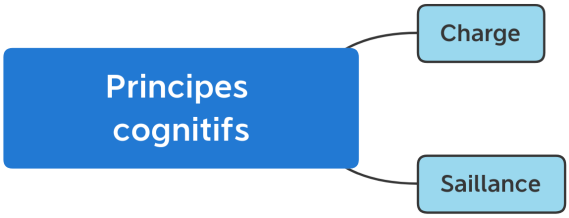


FIGURE 2.8 – Propriétés de la classe principes cognitifs

Au sein des principes cognitifs (figure 2.8), nous classons les propriétés suivantes : charge cognitive [99] et saillance cognitive [99].

2.2.2.1 Charge cognitive

Rukšėnas *et al.* [99] explique que la charge cognitive est liée à la tâche effectuée par l'utilisateur et plus particulièrement à sa complexité. Ils définissent deux types de charge cognitive : la charge cognitive intrinsèque qui se réfère directement à la complexité de la tâche ou de l'action exécutée ; la charge externe qui provient du contexte de la réalisation d'un objectif comme des informations non pertinentes ou des distracteurs.

La **charge cognitive** est donc liée à la tâche effectuée par l'utilisateur et plus particulièrement à sa complexité.

Exemple. Un utilisateur peut perdre son attention en interagissant avec une scène graphique trop riche.

2.2.2.2 Saillance cognitive

La **saillance cognitive** est une propriété qui consiste à faire en sorte qu'une idée, un principe, un schéma mental soit mis en avant par le système auprès de l'utilisateur.

Rukšėnas *et al.* [99] et Huang *et al.* [75] séparent la saillance cognitive de l'utilisateur lors de l'exécution d'une action en trois catégories : sensorielle, procédurale et cognitive.

Exemple. Un utilisateur sera plus concentré sur une action plus conforme à ses convictions.

2.2.3 Interfaces humain-machine

Nous identifions également la classe des propriétés relatives aux interfaces humain-machine adressant les propriétés induites par la présence d'interacteurs humain-machines (écrans, capteurs, etc.) [18].

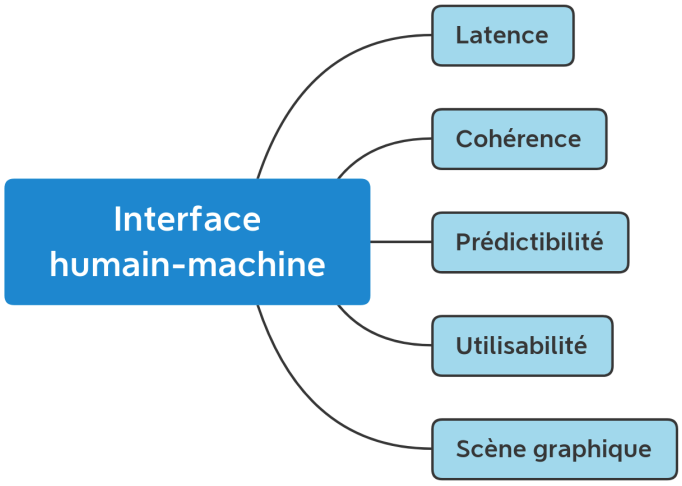


FIGURE 2.9 – Propriétés de la classe interface humain-machine

Au sein de la classe interfaces humain-machine (figure 2.9), nous identifions les propriétés suivantes : latence [81], cohérence [35], prédictibilité [86], utilisabilité [115], scène graphique [22], CARE [51].

2.2.3.1 Latence

Beckert et Beuster [19] expliquent l'importance du bon affichage des informations au bon moment afin de ne pas laisser les suppositions de l'utilisateur à propos

du système prendre le dessus sur l'état réel du système dans son esprit. Cela pourrait également conduire à ce que l'utilisateur ne voit pas du tout les informations affichées, par exemple si elles sont supprimées trop tôt de l'écran. Ce problème d'affichage peut résulter d'une erreur de conception ou d'un système trop lent, ce deuxième cas représentant la latence.

Leriche *et al.* [81] expliquent que les délais entre les interactions avec une application et le retour des informations de l'application avec des interactions tactiles peuvent rendre l'application inutilisable. Cette latence peut être gérée au niveau du matériel avec des outils conçus pour cette manipulation.

À l'origine, la latence est une problématique de réseau (temps de transmission d'un ordinateur à un autre) qui prend une autre forme dans le domaine des systèmes informatiques interactifs. "La **latence** d'un système interactif est le délai entre l'action d'un utilisateur et le moment où le retour d'information correspondant est présenté à l'utilisateur" [25].

Exemple. Si un ordinateur exécute plusieurs actions en même temps, il faudra quelques secondes pour lancer un navigateur web.

2.2.3.2 Cohérence

Bowen et Reeves [35] définissent la cohérence comme l'utilisation de la même terminologie pour les fonctions, les menus ou les écrans d'aide d'une application multiécrans, l'utilisation de commandes ou de séquences de commandes équivalentes afin d'atteindre des objectifs spécifiques tels que la fermeture d'une fenêtre dans l'interface.

Campos et Harrison [41] donnent trois propriétés liées à la cohérence des interfaces : le rôle et la visibilité des modes, la cohérence de la dénomination et de la finalité des fonctions telles que décrites précédemment, la cohérence du comportement des entrées de données. La première concerne la nécessité d'informer l'utilisateur du mode dans lequel se trouve le système, ce qui est surtout important pour les systèmes où l'utilisateur n'interagit qu'avec un dispositif physique. La troisième concerne les fonctions des touches et plus particulièrement le fait qu'elles ont un effet similaire quel que soit le mode.

Harrison *et al.* [67] définissent deux types de cohérence dans l'utilisation des touches programmables : le premier détaille quand une même touche est associée à une même fonction, le second explique qu'un affichage programmable apparaît lors de l'activation d'une touche particulière.

Harrison *et al.* [68] définissent la cohérence comme une propriété pour les actions. Une action telle qu'appuyer sur un bouton doit avoir le même effet quel que soit le mode du système et ne doit pas avoir d'autres effets que ceux prévus à l'origine par le comportement.

La **cohérence** représente donc un comportement constant du système, que ce soit pour un affichage ou une fonctionnalité, quel que soit le mode actuel du système.

Exemple. Il peut s'agir de l'utilisation de la même terminologie pour les fonctions ("Exit" ou "Quit" afin de définir une fonction "fermer une fenêtre").

2.2.3.3 Prédictibilité

Masci *et al.* [86] donnent une définition de la prédictibilité dans le cas d'utilisateurs expérimentés et compétents avec des systèmes interactifs. Ces utilisateurs sont censés avoir des modèles mentaux corrects du système. En d'autres termes, les utilisateurs connaissent les différentes caractéristiques des systèmes et leur fonctionnement. Dans ce cas précis, les auteurs définissent la prédictibilité comme la capacité de l'utilisateur à prévoir, à partir d'un état spécifique du système, le comportement futur du système et donc son état futur lorsqu'il interagit avec lui.

La **prédictibilité** est donc la capacité de l'utilisateur à prédire le comportement futur du système à partir de son état actuel et de la façon dont l'utilisateur interagira avec lui.

Exemple. Lorsqu'un utilisateur s'apprête à fermer un éditeur de texte contenant un document non sauvegardé, il sait qu'une fenêtre contextuelle s'affichera pour lui proposer un choix parmi : sauvegarder le document, annuler la fermeture ou fermer sans sauvegarder.

2.2.3.4 Utilisabilité

Rukšėnas *et al.* [98, 99] donnent des propriétés d'utilisabilité liées aux objectifs de l'utilisateur. En matière d'objectifs de l'utilisateur, les auteurs définissent l'utilisabilité comme l'assurance de l'atteignabilité du principal objectif de l'interaction, ou du moins celui que l'utilisateur perçoit comme principal. Rukšėnas *et al.* [98] précise également cette propriété avec l'assurance que tous les sous-objectifs de l'objectif principal seront également atteints à terme.

Coutaz *et al.* [51] présentent les propriétés CARE comme un moyen pour évaluer l'utilisabilité des interactions multimodales. Elles reposent sur des notions d'état (propriété qui peut être mesurée), d'objectif (état que l'utilisateur veut atteindre),

modalité (type d'interaction) et temporalité (temps passé sur une modalité). L'acronyme CARE correspond à :

- *complementarity* : les modalités doivent être utilisées de manière complémentaire (choix oral puis choix écrit par exemple),
- *assignment* : une seule modalité permet d'atteindre l'état suivant (absence de choix)
- *redundancy* : les modalités sont utilisées de manière redondante pour atteindre l'état suivant (peut représenter la confirmation d'un choix dans une modalité par une autre modalité)
- *equivalence* : les modalités sont équivalentes pour atteindre l'état suivant (choix écrit ou oral de la même phrase par exemple)

La norme ISO 9241-11 [115] définit l'**utilisabilité** comme "la mesure dans laquelle un produit peut être utilisé par des utilisateurs spécifiés pour atteindre des objectifs spécifiés avec efficacité, performance et satisfaction dans un contexte d'utilisation spécifié".

Exemple. Il est possible d'améliorer l'utilisabilité d'une fenêtre "accepter/refuser" en ajoutant des symboles liés aux deux notions, tels que ✓pour accepter et × pour refuser.

2.2.3.5 Propriétés liées à la scène graphique

Au niveau de la représentation graphique des données ou de l'état du système, Hjelmslev [70] donne deux plans permettant de définir un élément graphique et l'information qu'il contient. Cette information appartient au plan du contenu, aussi appelé plan des signifiés. L'élément graphique appartient au plan de l'expression ou plan des signifiants. Le signifiant peut être caractérisé par des paramètres associés à des éléments graphiques.

Bertin et Barbut [26] ont présenté des variables visuelles pour définir les éléments graphiques et les informations qu'ils représentent. Pour eux, un élément graphique peut être défini avec ses coordonnées, sa taille, sa valeur, son grain, sa couleur, son orientation et sa forme. Wilkinson [125] a ajouté la notion d'opacité à ces variables. Jacques Bertin, qui était cartographe, portait son intérêt principalement sur les éléments graphiques placés sur une feuille de papier. De ce fait, les coordonnées étudiées se limitaient au plan (x, y). Dans le cas des systèmes informatiques où les interfaces humain-machine ont un affichage dynamique, il faut également ajouter la

variable l correspondant à l'ordre ou couche d'affichage des éléments graphiques. La couche d'affichage et le dynamisme de l'affichage induisent des propriétés liées à la superposition des éléments graphiques.

Les variables visuelles définies par Bertin et Barbut [26] (coordonnées, taille, valeur, grain, couleur, orientation et forme) et Wilkinson [125] (opacité) peuvent alors caractériser le signifiant de Hjelmslev [70].

Randell et Cohn [94] ont décrit une logique d'intervalle qui peut être utilisée pour raisonner sur les positions spatiales des régions. Ils ont basé leur logique sur une relation dyadique primitive qui représente la connexion de deux régions.À partir de là, ils ont défini un ensemble de base de relations dyadiques pour deux régions telles que déconnectées, partielles, identiques, se chevauchant, se chevauchant partiellement.

La **scène graphique** [22] est définie par les variables visuelles proposées par Bertin et Barbut ainsi que Wilkinson qui permettent également de définir les différents composants de cette scène graphique : coordonnées, taille, valeur, grain, couleur, orientation, forme et opacité. À partir de ces variables, il est possible de raisonner sur la nature des éléments graphiques ainsi que des problématiques plus complexes de visibilité, perceptibilité, etc.

Exemple. Un texte rouge affiché au-dessus d'un fond rouge ne sera pas visible.

2.2.4 Sécurité

Cette classe prend en compte les propriétés liées à la sécurité informatique, telles que la prévention des menaces et le lien entre le comportement de l'utilisateur et les menaces éventuelles [103].

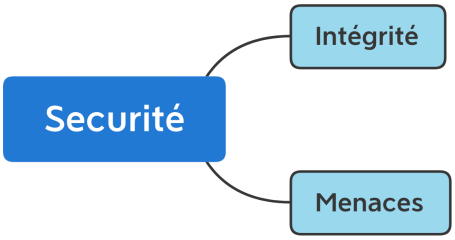


FIGURE 2.10 – Propriétés de la classe sécurité

Au sein de la sécurité (figure 2.10), nous classons les propriétés suivantes : intégrité [19] et menaces [78].

2.2.4.1 Intégrité

Beckert et Beuster [19] donnent deux définitions de l'intégrité. La première est générique aux systèmes qui peuvent être exposés à des menaces et explique que les hypothèses de l'utilisateur sur l'application sont correctes et que l'inverse est également vrai. La seconde est spécifique aux interfaces utilisateur et explique qu'il existe une correspondance entre les hypothèses de l'utilisateur concernant l'état et les données de l'application et sa configuration. Ils précisent cette deuxième propriété en expliquant la nécessité pour l'utilisateur, lorsqu'il effectue une tâche critique, d'avoir des hypothèses sur les propriétés critiques du système identiques à ses propriétés réelles.

La propriété d'**intégrité** indique donc que les hypothèses de l'utilisateur concernant l'application sont correctes et l'inverse est également vrai.

Exemple. Lorsque nous nous connectons à des interfaces avec deux champs de texte, l'inversion du champ nom d'utilisateur et du champ mot de passe est un défaut d'intégrité.

2.2.4.2 Menaces

Beckert et Beuster [19] rappellent que la fuite de données, la manipulation de données et la manipulation de programmes sont des menaces fondamentales en matière de sécurité informatique et que ces menaces peuvent être contrées en assurant les exigences de sécurité que sont la confidentialité, l'intégrité et la disponibilité.

Johnson [78] s'intéresse aux menaces dans le domaine des systèmes mondiaux de navigation par satellite. Dans ce domaine spécifique, les menaces sont plus criminelles qu'involontaires, comme l'utilisation de dispositifs de brouillage de la navigation par satellite pour perturber les signaux ou la diffusion de faux signaux des systèmes mondiaux de navigation par satellite pour modifier l'itinéraire d'un utilisateur de GPS.

La propriété de **menace** se concentre sur la définition des différentes menaces qui peuvent constituer un risque pour le système.

Exemple. Les fuites de données ou les manipulations de données sont des menaces pour un système informatique.

2.2.5 Modélisation des systèmes interactifs

Il est important de noter que souvent les articles n'adressent pas directement les propriétés liées à l'interaction mais plutôt les propriétés du modèle qui définit la structure et le comportement du système informatique interactif ou de ses exigences. Pour ces articles spécifiques, nous avons défini la catégorie **modélisation des systèmes interactifs** qui rassemble des articles adressant la modélisation formelle d'un système, et éventuellement les propriétés liées au modèle lui-même, et non centrées sur les propriétés liées à l'interaction.

2.2.6 Synthèse

Au cours de nos recherches afin de définir les propriétés liées à l'interaction, nous avons déterminé un premier finding.

Finding 1. Il n'existe pas de classification de référence des propriétés liées à l'interaction.

Suite à ce finding, nous avons donc créé notre classification de ces propriétés afin de donner un cadre à notre revue de littérature. Nous avons également ajouté une catégorie modélisation permettant de classer les articles adressant particulièrement la modélisation des systèmes informatiques interactifs.

2.3 Nomenclature des formalismes

La vérification formelle consiste à s'assurer que l'application est conforme à une spécification de référence en se basant sur l'élaboration d'un modèle dont la syntaxe et la sémantique reposent sur des bases mathématiques permettant ainsi des raisonnements objectifs.

De par sa nature, chaque formalisme se prête généralement à une méthode de vérification formelle préférentielle, pour laquelle il est le mieux adapté. Ainsi, pour chacun d'entre eux, nous préciserons cette méthode. Nous avons choisi d'utiliser la nomenclature suivante (figure 2.11) pour les présenter : algèbre de processus [11], langage de spécification [105], processus de raffinement [5], systèmes de transition d'états [12] et logique temporelle [64].

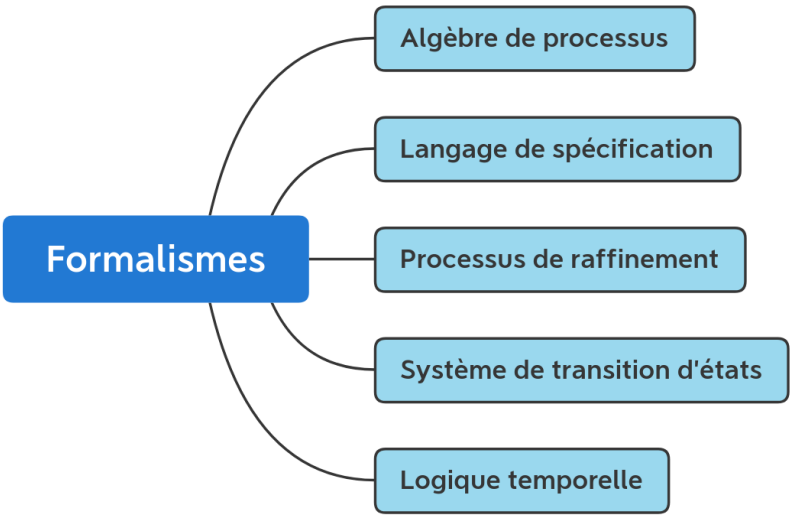


FIGURE 2.11 – Nomenclature des formalismes

2.3.1 Algèbre de processus

Baeten [11] donne l'historique et la définition de l'algèbre de processus. L'auteur donne également des exemples de certains formalismes de l'algèbre de processus tels que le calcul des systèmes communicants (CCS) ou les processus séquentiels communicants (CSP). Nous pouvons résumer à partir de cet article que l'algèbre de processus est un ensemble de moyens algébriques utilisés pour étudier et définir le comportement des systèmes parallèles.

Afin de fournir des exemples d'algèbres de processus nous proposons la liste suivante : syntaxe CWB-NC [49] pour la notation CSP de Hoare [72], Language Of Temporal Ordering Specification (LOTOS) [76], π-calcul probabiliste [90], π-calcul appliqué [102], Performance Evaluation Process Algebra (PEPA) [69].

Un exemple : la logique de Hoare. La logique de Hoare [71] repose sur la notion du triplet de Hoare $\{P\}Q\{R\}$ dans lequel P et R sont des formules logiques ou assertions représentant respectivement une précondition (celle-ci pourrait ne pas être imposée et considérée vraie par défaut et serait remplacée par le mot-clé true) et une postcondition et Q le programme typiquement impératif (instruction ou suite d'instructions). Ce triplet signifie que R sera vraie à la fin de l'exécution de Q si P était vraie à l'initialisation de Q.

Nous trouvons également la définition de l'axiome d'affectation indiquant, pour une affectation $x := f$, avec x l'identifiant d'une variable et f une expression pouvant contenir x, que si $P(x)$ doit être vraie après l'affectation alors $P(f)$ doit avoir été

vraie avant. Formellement, $\{P_0\}x := f\{P\}$ avec x l'identifiant d'une variable, f une expression et P_0 obtenu de P par substitution de toutes les occurrences de x par f.

À cela s'ajoutent des règles telles que :

- la conséquence : $\{P\}Q\{R\} \wedge R \supset S \Rightarrow \{P\}Q\{S\}$
  et $\{P\}Q\{R\} \wedge S \supset P \Rightarrow \{S\}Q\{R\}$,
- la composition : $\{P\}Q_1\{R_1\} \wedge \{R_2\}Q_2\{R\} \Rightarrow \{P\}Q_1; Q_2\{R\}$,
- l'itération : $P \wedge \{B\}S\{P\} \Rightarrow \{P\}while\ B\ do\ S\{\neg B \wedge P\}$, B étant une condition.

Cette modélisation permet de prouver qu'un programme vérifie une spécification, en garantissant la véracité de R à la fin de Q si P était vraie à l'initialisation de Q.

Key-Hoare et Hoare Advanced Homework Assistant sont des outils permettant d'utiliser la logique de Hoare.

Exemple. Le cas simple $\{0 \leq x \leq 1\}y := x^2\{y \leq x\}$ indique que la postcondition $y \leq x$ est vraie à la fin de l'instruction $y := x^2$ si la précondition $0 \leq x \leq 1$ est vraie avant l'exécution de l'instruction.

2.3.2 Langage de spécification

Un langage de spécification [105] est un langage formel qui peut être utilisé pour faire des descriptions formelles de systèmes. Il permet à un utilisateur d'analyser un système ou ses exigences (généralement par preuve) et d'améliorer ainsi sa conception.

Les langages de spécification les plus utilisés sont : SAL [55], Z [114], μCharts [63], Spec# [16], Promela [73], PVS [108], Higher-Order Processes Specification (HOPS) [57].

2.3.3 Processus de raffinement

Le perfectionnement d'un programme consiste en la concrétisation d'une description plus abstraite d'un système. Le but de cette méthode est de vérifier les propriétés dans un niveau abstrait de la description puis de concrétiser ce niveau tout en conservant les propriétés vérifiées. Ces étapes doivent être réalisées jusqu'à l'obtention de la description concrète du système.

Le raffinement constituant davantage un processus qu'un modèle, nous proposons ici une liste de modèles et langages de spécification utilisés pour appliquer des

processus de raffinement : modèles tels que la méthode B et B événementiel [5] et langages de spécification tels que Z [114] et μCharts [63].

Un exemple : la méthode B événementiel. La méthode B événementiel [6] repose sur l'utilisation de la théorie des ensembles comme notation de modélisation, du raffinement pour représenter les systèmes à différents niveaux d'abstraction et de la preuve mathématique pour vérifier la cohérence entre les niveaux de raffinement. "Un modèle B événementiel décrit un système réactif par un ensemble d'événements (clause EVENTS) qui modifient un état (clause VARIABLES). Le raffinement permet de bâtir ce modèle étape par étape en introduisant de nouveaux événements tout en préservant les propriétés établies (clauses INVARIANT et ASSERTION)." [7]

Exemple. La figure 2.12 présente l'exemple du modèle B événementiel et de sa première étape de raffinement pour l'ajout des minutes donné par Aït-Ameur *et al.* [7].

MODEL	**REFINEMENT**	
$Clock$	$ClockWMinute$	
VARIABLES	**REFINES**	
h	$Clock$	
INVARIANT	**VARIABLES**	
$h \in 0..23$	h, m	
ASSERTIONS	**INVARIANT**	
$h < 100$	$m \in 0..59$	
INITIALISATION	**ASSERTIONS**	
$h := 13$	$(h \neq 23) \vee (h = 23) \Rightarrow$	
	$(h \neq 23 \wedge m = 59) \vee (h = 23 \wedge m = 59) \vee (m \neq 59)$	
EVENTS	**VARIANT**	
$incr =$	$59 - m$	
SELECT $h \neq 23$	**INITIALISATION**	
THEN	$h := 13 \parallel m := 14$	
$h := h + 1$	**EVENTS**	
END;	$incr =$	
$zero =$	**SELECT** $h \neq 23 \wedge m = 59$	
SELECT $h = 23$	**THEN**	
THEN	$h := h + 1 \parallel m := 0$	
$h := 0$	**END;**	
END	$zero =$	$ticTac =$
END	**SELECT** $h = 23 \wedge m = 59$	**SELECT** $m \neq 59$
	THEN	**THEN**
	$h := 0 \parallel m := 0$	$m := m + 1$
	END;	**END**

FIGURE 2.12 – Modèle B événementiel d'une horloge et son raffinement [7]

2.3.4 Système de transition d'états

Les systèmes de transition d'états [12] consistent en des graphes dirigés composés d'états, représentés par des nœuds, et de transitions, représentées par des arêtes. Un état représente un instant dans le comportement du système ou, pour un programme, la valeur actuelle de toutes les variables et l'état actuel du programme. Le franchissement d'une transition implique un changement d'état. La vérification de propriétés sur ce type de modèle repose principalement sur les mécanismes de simulation du modèle ainsi que, pour certains, sur des techniques d'analyse de graphe.

Afin de fournir des exemples de systèmes de transition d'états nous proposons la liste suivante : UPPAAL [24], les réseaux de Petri [54], les modèles ICO (pour *Interactive Cooperative Objects*) [88], les automates à états finis, le système de transition étiqueté d'entrée/sortie (IOLTS pour *input-output labeled transition system*) [19].

Un exemple : les réseaux de Petri. Un réseau de Petri est un triplet $\langle P, T, F \rangle$ avec P un ensemble de places p, T un ensemble de transitions t et F la relation de flot [97]. Les places peuvent contenir des jetons qui conditionnent l'exécution (on parle de tirage) des transitions : un tirage se concrétise par la consommation et la production de jetons. On note M_0 le marquage (nombre de jetons dans l'ensemble du réseau) initial. Le marquage M correspondant au nombre de jetons présents dans l'ensemble des places.

Le marquage d'une (des) place(s) en amont d'une transition t_j détermine si cette dernière est tirable : si suffisamment de jetons sont présents (indication par le poids des arcs entrant dans t_j) dans la (les) place(s) en amont de t_j, celle-ci est sensibilisée et peut être tirée. t_j tirée consomme les jetons nécessaires de la (des) place(s) en amont et fournit à la (aux) place(s) en aval les jetons devant être transmis (indication par le poids des arcs sortant de t_j). La consommation et le don de jetons donnent le nouveau marquage.

David et Alla [54] présentent les propriétés de ce modèle ($[\exists s | M.s \rightarrow M']$ signifie qu'il existe une séquence s de tirs franchissable depuis M amenant à M', on dit que M' est accessible depuis M). Une place p_i est k-bornée si $\forall M | [\exists s | M_0.s \rightarrow M] \wedge M(p_i) \leq k$. Un réseau est k-borné si toutes ses places sont k-bornées, il sera dit sauf si $k = 1$. Une transition t_j est vivante pour M_0 si $\forall M | [\exists s | M_0.s \rightarrow M] \wedge [\exists s' | M.s' \rightarrow M"]$ avec $M" \geq Pre(, t)$. t_j est quasi vivante si $[\exists s | M_0.s \rightarrow M]$ avec $M \geq Pre(, t)$. Un réseau est vivant si toutes ses transitions sont vivantes. Un

réseau est réinitialisable si $\forall M|[\exists s|M_0.s \rightarrow M] \wedge [\exists s'|M.s' \rightarrow M_0]$. Un réseau est bloquant pour M_0 si $[\exists s|M_0.s \rightarrow M] \wedge [\forall t_j \in T, Pre(,t_j) > M]$.

Ce modèle permet de vérifier des propriétés génériques d'atteignabilité d'état (vivacité, non-blocage) et de bornes.

Exemple. Prenons le cas d'un espace de stockage pouvant contenir jusqu'à n éléments. Sa gestion, simple (stockage et retrait), est représentée par la figure 2.13 avec le formalisme réseau de Petri.

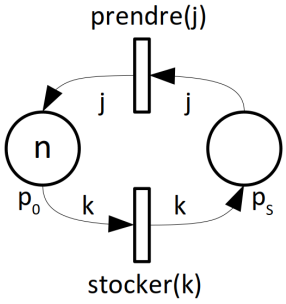


FIGURE 2.13 – Réseau de Petri de la gestion d'un espace de stockage

Sur cette figure, nous pouvons voir qu'il y a deux places : p_S représentant le nombre d'éléments dans l'espace de stockage et p_0 la place indiquant le nombre de places restantes dans l'espace de stockage. Il y a également deux transitions. Celle labélisée *stocker(k)* ayant un arc amont de poids k en provenance de la place p_0 et un arc aval de poids k à destination de p_S représente l'action de stocker k éléments dans l'espace de stockage. Celle labélisée *prendre(j)* ayant un arc amont de poids j en provenance de la place p_S et un arc aval de poids j à destination de p_0 représente l'action de prendre j éléments de l'espace de stockage. À chaque tir de transition (*stocker(k)*, *prendre(j)*), le nombre de jetons total dans le réseau reste constant à n, cela signifie que le réseau est n-borné et donc le stock peut bien accueillir au maximum n éléments.

2.3.5 Logique temporelle

Les propriétés à vérifier sont souvent exprimées sous forme de formules de logiques temporelles [64]. Ces formules sont basées sur des combinateurs booléens, des combinateurs temporels et pour certaines logiques sur des quantificateurs de chemin.

La logique temporelle permet de réaliser des assertions sur les états des exécutions (séquences d'états) grâce à des expressions utilisant les combinateurs booléens

true, *false*, $\neg$, $\wedge$, $\vee$, $\Rightarrow$ et $\Leftrightarrow$.

De plus, il existe des combinateurs temporels, permettant de raisonner sur les états d'une séquence :

- X : l'état suivant satisfait la propriété ϕ ($X\phi$, figure 2.14.c),
- F : il existera un état satisfaisant ϕ ($F\phi$, figure 2.14.d),
- G : tous les états vérifieront ϕ ($G\phi$, figure 2.14.e).

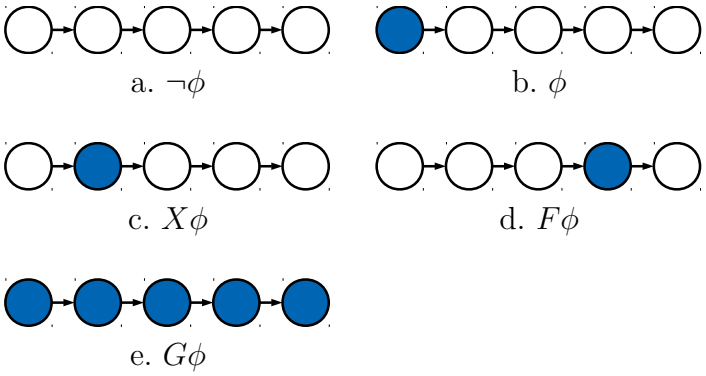


FIGURE 2.14 – Représentation de la logique temporelle linéaire

Enfin, nous avons des quantificateurs de chemin, permettant de raisonner sur l'ensemble des exécutions (arbre) :

- E : il existe une exécution satisfaisant ϕ ($E\phi$, figure 2.15.c),
- A : toutes les exécutions satisfont ϕ ($A\phi$, figure 2.15.d).

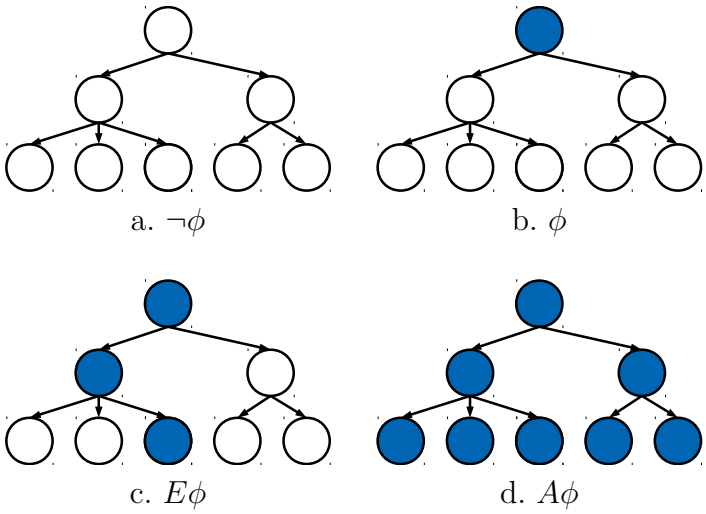


FIGURE 2.15 – Représentation de la logique du temps arborescent

Les deux principales logiques temporelles utilisées sont les suivantes : logique du temps arborescent (CTL pour *Computation Tree Logic*) et logique temporelle linéaire (LTL pour *Linear Temporal Logic*) [39].

Exemple. La propriété "Dès que la température devient supérieure à 100 degrés, une alarme sonore doit être levée" s'écrit en LTL : "$(Température > 100) \Rightarrow X(alarme\_sonore\_levée)$".

2.3.6 Synthèse

Nous avons présenté ici les principaux formalismes utilisés par les méthodes formelles. Bien que largement incomplète, cette nomenclature nous permettra cependant de classer efficacement les différents formalismes utilisés pour modéliser et vérifier les propriétés des systèmes interactifs.

2.4 Synthèse

À travers cet état de l'art, nous avons vu qu'afin de développer des systèmes informatiques, il existe de nombreux paradigmes de programmation dont les plus usuels sont l'impératif, le fonctionnel, l'objet et l'interactif (tableau 2.1) avec des spécificités qui leur sont propres : définition du flot d'exécution, présence ou absence de la notion d'état, définition de l'état (si existant), propriétés spécifiques (état de la mémoire, terminaison, encapsulation, dépendances, etc.).

Dans le cadre de cette thèse, nous nous intéressons aux systèmes informatiques interactifs et donc au paradigme réactif. De ce fait, nous utiliserons ce paradigme et ses spécificités dans la suite de nos travaux.

Nous avons ensuite présenté les propriétés liées à l'interaction (figure 2.6). Nous avons proposé une classification permettant d'organiser ces propriétés : comportement de l'utilisateur, principes cognitifs, interfaces humain-machine, sécurité. Ces classes comportent un nombre conséquent de propriétés qui peuvent être directement reliées au système en lui-même, à son environnement qu'il s'agisse de l'humain ou d'autres systèmes, ou alors aux interactions entre le système et son environnement. Ces propriétés ne se définissent pas et ne se vérifient pas nécessairement par les mêmes moyens.

C'est pourquoi nous avons proposé une nomenclature des formalismes (figure 2.11) : algèbre de processus, langage de spécification, processus de raffinement, système de transition d'états, logique temporelle.

À partir de cette présentation des concepts de base (classification des propriétés des systèmes interactifs et nomenclature de référence des formalismes), nous maintenant pouvons étudier et comprendre les travaux de recherche qui ont été menés dans le cadre de méthodes formelles appliquées aux propriétés des systèmes interactifs. Cela nous permettra de mettre en lumière les forces et les faiblesses des approches formelles appliquées aux systèmes informatiques interactifs. À partir de cela, nous expliciterons la problématique de cette thèse. Ce sera l'enjeu du chapitre suivant.

CHAPITRE 3

ÉTAT DE L'ART

Historiquement, les techniques classiques d'analyse formelle ont été développées pour être appliquées sur des langages impératifs ou fonctionnels. Celles-ci restent peu utilisées pour la prise en compte des spécificités des systèmes interactifs ainsi que celles des propriétés liées à l'interaction qui s'y rapportent.

L'objectif ici est de faire l'état de l'art de l'utilisation des méthodes formelles appliquées aux systèmes interactifs et de justifier le positionnement de cette thèse. Pour ce faire, dans un premier temps nous étudions la littérature en nous basant sur les questions suivantes :

- *Quelles sont les propriétés liées à l'interaction étudiées ?*
- *Quels sont les formalismes utilisés pour ce faire ?*
- *Quels sont les principaux cas d'étude utilisés pour démontrer l'approche ?*

La deuxième partie de ce chapitre est une analyse de cette revue de la littérature. Notre objectif ici est de répondre à deux questions :

- *Quelles sont les propriétés liées à l'interaction les plus couvertes et celles qui sont les moins traitées ?*
- *Quels sont les formalismes les plus utilisés pour l'expression et la vérification des propriétés des systèmes informatiques interactifs, et y en a-t-il des nouveaux qui ont émergé spécifiquement pour ce type de propriétés ?*

Pour répondre à ces questions et organiser cette revue de la littérature, nous utilisons les deux nomenclatures élaborées au chapitre précédent : la classification des propriétés liées à l'interaction (section 2.2, page 17) et la nomenclature des formalismes (section 2.3, page 28).

3.1 Objectif de ce chapitre

L'application des méthodes formelles aux systèmes informatiques interactifs est une problématique réellement prise en compte depuis peu. Prenons l'exemple de l'aéronautique qui constitue un secteur critique ou la vérification de la conformité d'un système à ses exigences est cruciale. En 1992, la DO-178B [111], qui décrit les contraintes de développement pour l'obtention de la certification d'un logiciel critique embarqué à bord des aéronefs d'aviation commerciale et générale, contenait deux paragraphes expliquant comment utiliser les méthodes formelles pour la certification des systèmes aéronautiques. En 2012, sa nouvelle version, la DO-178C [106], a présenté une annexe (DO-333 [107]) entièrement dédié à cette même problématique.

Il est donc intéressant aujourd'hui d'analyser les applications des méthodes formelles aux systèmes informatiques interactifs. L'objectif ici est de déterminer leur puissance d'expression et de vérification des propriétés de ces systèmes, leur passage à l'échelle par le biais des cas d'étude utilisés dans la littérature.

Nous voulons ici avoir une idée de la couverture d'étude formelle (expression et vérification) des propriétés liées à l'interaction. Pour ce faire, nous avons analysé un certain nombre de travaux représentatifs utilisant les méthodes formelles pour exprimer et vérifier des propriétés sur des cas d'étude (industriels ou non). Cela nous permettra de positionner et de justifier cette thèse.

3.2 Revue de la littérature

3.2.1 Méthodologie

L'objectif ici est de faire l'état de l'art de l'utilisation des méthodes formelles appliquées aux systèmes interactifs et de justifier le positionnement de cette thèse. Pour ce faire, nous étudions la littérature en nous basant sur les questions suivantes :

- "Quelles sont les propriétés liées à l'interaction étudiées ?" Cette question concerne la nature des propriétés liées à l'interaction qui ont été étudiées et est au centre de notre travail pour déterminer si certaines propriétés n'ont pas été étudiées.
- "Quels sont les formalismes utilisés pour ce faire ?" Cette question nous permet de montrer quels formalismes peuvent être utilisés pour étudier les propriétés liées à l'interaction.

- "Quels sont les principaux cas d'étude utilisés pour démontrer l'approche ?" Cette question concerne le système utilisé comme cas d'étude pour illustrer l'utilisation des méthodes formelles et ses particularités.

Nous nous sommes principalement intéressés au workshop *Formal Methods for Interactive Systems* (FMIS) pour cette revue de littérature. Ce workshop est au cœur de notre problématique puisqu'il regroupe des publications exclusivement liées à l'utilisation des méthodes formelles appliquées aux systèmes informatiques interactifs. Il a eu lieu 7 fois entre 2006 et 2018. Sur cette période, nous dénombrons 43 publications et un total de 94 auteurs différents sur l'ensemble de ces publications.

De plus, afin d'affiner les résultats de cette première revue de littérature, nous avons analysé les articles de la conférence *Engineering Interactive Computing Systems* (EICS). Cette conférence est également au cœur de la problématique que nous ciblons. Elle a eu lieu 11 fois entre 2009 et 2019. Sur cette période, nous dénombrons 458 publications et un total de 842 auteurs différents sur l'ensemble de ces publications.

Nous avons synthétisé les résultats de cette revue de littérature sous la forme de tableaux pour chaque type de propriété. Nous notons dans la première colonne le formalisme utilisé précédé de lettres entre parenthèses afin de spécifier de quel type de formalisme il s'agit : **AP** pour algèbre de processus, **LS** pour langage de spécification, **Ra** pour processus de raffinement, **STE** pour système de transition d'états, **LT** pour logique temporelle, **autre** pour les formalismes ne rentrant pas dans la nomenclature (figure 2.11, page 29) et **outil** lorsqu'il s'agit de la présentation d'outils de vérification.

À la suite de ces tableaux, nous donnons succinctement pour chaque propriété liée à l'interaction des exemples d'études qui ont été menées. Dans ces tableaux, nous renseignons exclusivement les travaux qui ont utilisé des formalismes pour travailler sur des propriétés des systèmes interactifs (ou sur la spécification des systèmes).

3.2.2 Propriétés liées au comportement de l'utilisateur

Le tableau 3.1 synthétise les études de la classe de propriétés relatives au comportement de l'utilisateur. Il répertorie les articles en fonction des propriétés étudiées (objectifs, privilèges, interprétation, attention et expérience) et des formalismes utilisés pour ce faire.

	Buts	Privilèges	Interprétation	Attention	Expérience
(AP) CWB-NC	[46]	[46]			
(AP) CSP	[44]			[44]	[44]
(AP) LOTOS				[116]	
(AP) PEPA					[47]
(LS) SAL	[98] [99]		[98] [100]		
(LS) HOPS	[58]				
(autre) HTDL				[45]	[45]
ad hoc					

TABLE 3.1 – Études des propriétés relatives au **comportement de l'utilisateur**

3.2.2.1 But

Cerone et Elbegbayan [46] définissent les buts des utilisateurs dans l'utilisation d'une interface web qui comporte un forum de discussions et une liste de membres. Les auteurs les définissent avec la syntaxe CWB-NC pour CSP (algèbre de processus). Ces définitions leur permettent de modéliser plus précisément les cas d'utilisation prévus et non prévus.

Rukšėnas *et al.* [98] traitent de l'utilisation d'une interface d'authentification avec deux zones de texte (nom d'utilisateur et mot de passe). Ils définissent les buts de l'utilisateur avec SAL (langage de spécification) par la définition d'une architecture cognitive du comportement de l'utilisateur. Elle permet aux auteurs de définir les actions qu'un utilisateur peut effectuer. Rukšėnas *et al.* [99] approfondissent la notion de buts de l'utilisateur à travers leur architecture cognitive.

Cerone [44] base son travail sur l'étude de deux cas d'utilisation : un utilisateur qui conduit une voiture jusqu'à son travail et doit la garer sur un eplacement réservé et un utilisateur qui interagit avec un distributeur automatique de billets. Il modélise les objectifs de l'utilisateur avec la notation d'Hoare sous la forme d'algèbre de processus CSP. Cela lui permet d'étudier les activités cognitives telles que la terminaison.

Dittmar et Schachtschneider [58] utilisent les modèles HOPS (langage de spécification) pour définir les tâches et les actions de l'utilisateur pendant qu'il résout un puzzle.

3.2.2.2 Privilèges

Cerone et Elbegbayan [46] définissent les privilèges des utilisateurs avec la syntaxe CWB-NC pour CSP (algèbre de processus). Ainsi, les auteurs peuvent mo-

déliser les actions que les utilisateurs identifiés ou non sont autorisés à effectuer. Cela permet aux auteurs de limiter le comportement de l'utilisateur en ajoutant de nouvelles propriétés dans le modèle d'interface web.

3.2.2.3 Interprétation

Rukšėnas *et al.* [98] traitent de l'interprétation par l'utilisateur d'une interface d'authentification. Ils la définissent avec SAL (langage de spécification) par la définition d'une architecture cognitive du comportement de l'utilisateur. Elle permet aux auteurs de souligner le risque pour l'utilisateur de mal comprendre l'interface en fonction de l'affichage des deux zones de texte. Rukšėnas et Curzon [100] étudient le comportement plausible des utilisateurs qui interagissent avec la saisie de numéros sur les pompes à perfusion. Ils supposent que les utilisateurs ont leurs propres convictions sur les valeurs incrémentales. Ils modélisent séparément le comportement des utilisateurs en fonction de leur interprétation et de la contrainte des disparités cognitives avec LTL (logique temporelle) et le vérificateur de modèle SAL.

3.2.2.4 Attention

Su *et al.* [116] étudient la limitation temporelle de l'attention en présence de stimuli sur des interfaces réactives riches en stimuli (IRRS). Les auteurs définissent le modèle cognitif des opérateurs humains avec LOTOS (algèbre de processus). Le modèle des IRSS est basé sur l'étude d'une tâche AB [60]. Ce travail présente des résultats de simulation axés sur les performances de l'interface en matière d'attention de l'utilisateur.

Cerone [44] aborde les attentes des utilisateurs, ce qui repose sur l'attention et l'expérience des utilisateurs. Il étudie les activités cognitives telles que la terminaison, la programmation des conflits et l'activation de l'attention. Il modélise celles qui utilisent la notation d'Hoare pour décrire le CSP (algèbre de processus).

3.2.2.5 Expérience

Cerone et Zhao [47] utilisent PEPA (algèbre de processus) pour modéliser un carrefour à trois voies sans feux de circulation et avec une situation de circulation donnée. Ils étudient l'expérience des usagers en matière de conduite dans de tels carrefours. Ils utilisent le plug-in Eclipse PEPA (algèbre de processus) pour analyser le modèle et déterminer par exemple la probabilité d'une éventuelle collision.

Cerone [45] propose une architecture cognitive pour la modélisation du comportement humain. Cet article présente le langage de description des tâches humaines (HTDL pour *Human Task Description Language*). Il l'utilise pour modéliser des propriétés liées au comportement de l'utilisateur telles que le contrôle automatique (tâches quotidiennes) et délibéré (motivé par un objectif) et l'apprentissage, l'attention et l'expérience de l'humain.

3.2.3 Propriétés liées aux principes cognitifs

Le tableau 3.1 synthétise les études de la classe de propriétés relatives aux principes cognitifs. Il répertorie les articles en fonction des propriétés étudiées (saillance, charge) et des formalismes utilisés pour ce faire.

	Saillance	Charge
(LS) SAL	[99] [75]	[99] [75]
(autre) GUM	[75]	[75]

TABLE 3.2 – Études des propriétés relatives aux **principes cognitifs**

Rukšėnas *et al.* [99] définissent deux principes cognitifs, la saillance et la charge cognitive. Ils les ajoutent à leur architecture cognitive SAL (langage de spécification). Les auteurs définissent également le lien entre ces deux principes. Ils illustrent ces principes à travers l'étude de cas d'une tâche de répartition des pompiers.

Huang *et al.* [75] essayent de vérifiquer que leur modèle utilisateur générique (GUM en anglais) peut encapsuler tous les principes cognitifs présentés dans l'expérience de la machine à beignets d'Ament *et al.* [8].

3.2.4 Propriétés liées aux interfaces humain-machine

Le tableau 3.3 synthétise les études de la classe de propriétés relatives aux interfaces humain-machine. Il répertorie les articles en fonction des propriétés étudiées (latence, cohérence, prédictibilité, utilisabilité, et scène graphique) et des formalismes utilisés pour ce faire.

3.2.4.1 Latence

Leriche *et al.* [81] explorent la possibilité d'utiliser le délai d'exécution dans le pire des cas (WCET pour *Worst-Case Execution-Time*) [93] basé sur des arbres pour étudier la latence des systèmes interactifs. Ils présentent également quelques

	Latence	Cohérence	Prédictibilité	Utilisabilité	Scène graphique
(LS) SAL			[86]	[98] [99]	
(LS) PVS		[67] [68]			
(LS/Ra) μCharts				[36]	
(LS/Ra) Z		[35]			
(STE) IOLTS	[19]	[19]			
(LT) LTL	[19]	[19]		[99]	
(LT) CTL		[67] [41]			
(autre) WCET basé sur arbre	[81]				

TABLE 3.3 – Études des propriétés relatives aux **interfaces humain-machine**

travaux qui ont été réalisés avec des graphes d'activation pour modéliser des systèmes interactifs.

3.2.4.2 Cohérence

Bowen et Reeves [35] utilisent leurs modèles de présentation et leurs processus de raffinement avec Z (langage de spécification) pour vérifier l'équivalence et la cohérence entre deux conceptions d'interface utilisateur. Les modèles de présentation leur permettent de s'assurer que des contrôles ayant la même fonction portent le même nom et inversement.

Beckert et Beuster [19] proposent un modèle IOLTS (système de transition d'états) d'une application textuelle pour garantir les contraintes de cohérence. Leur premier modèle ne satisfait pas les contraintes de cohérence. Ils raffinent ce modèle afin de les satisfaire.

Campos et Harrison [41] proposent une définition formelle de la cohérence de l'interface de la pompe volumétrique Alaris GP avec CTL (logique temporelle). La cohérence globale comprend : le rôle et la visibilité des modes, la relation entre le nommage et le but des fonctions, la cohérence du comportement des touches de saisie des données. Ils présentent également une partie d'une spécification MAL de la pompe à perfusion Alarais GP.

Harrison *et al.* [67] explorent la cohérence dans l'utilisation des touches de fonction logicielles des pompes à perfusion par l'utilisation de modèles MAL traduits en PVS (langage de spécification). Ils définissent les propriétés de cohérence avec CTL (logique temporelle) et les traduisent en théorèmes PVS (langage de spécification).

Harrison *et al.* [68] crée un modèle d'un distributeur de pilules à partir d'une spécification dans PVS (langage de spécification). Ils utilisent cette spécification avec l'outil PVSio-web pour étudier la cohérence des actions possibles.

3.2.4.3 Prédictibilité

Masci *et al.* [86] analysent la prédictibilité du système de saisie de données numériques des pompes à perfusion Alaris GP et B-Braun Infusomat Space. Ils utilisent des spécifications en SAL (langage de spécification) pour préciser la prédictibilité du système de saisie de données numériques de la pompe B-Braun.

3.2.4.4 Utilisabilité

Rukšėnas *et al.* [98] utilisent leur modèle de comportement de l'utilisateur dans SAL pour vérifier les propriétés d'utilisabilité d'une interface d'authentification. Ils vérifient la satisfaction de la propriété "l'utilisateur finit par atteindre le but perçu". Rukšėnas *et al.* [99] explorent davantage l'utilisation de leur modèle d'utilisateur avec SAL et des propriétés exprimées en LTL (logique temporelle). Ils vérifient satisfaction de la propriété "l'utilisateur finit par atteindre le but principal" dans une tâche de répartition des pompiers lors de l'utilisation d'une interface.

Bowen et Reeves [36] présentent un moyen d'appliquer le langage de spécification μCharts et les processus de raffinement aux conceptions d'interfaces utilisateur. Ils utilisent des modèles de présentation pour comparer deux conceptions d'interfaces utilisateur et si ces dernières restent utilisables. Ils décrivent également de manière informelle le processus de raffinement lié à la conception d'interfaces utilisateur.

3.2.5 Propriétés liées à la sécurité

Le tableau 3.4 synthétise les études de la classe de propriétés relatives à la sécurité.

	Intégrité	Menaces
(LS) SAL	[98]	
(STE) IOLTS	[19]	
(LT) LTL	[19]	
(autre) BDMP		[78]
ad hoc	[10]	

TABLE 3.4 – Études des propriétés relatives à la **sécurité**

Il répertorie les articles en fonction des propriétés étudiées (intégrité, menaces) et des formalismes utilisés pour ce faire.

Rukšėnas *et al.* [98] vérifient le risque de violation de la sécurité dans une interface d'authentification avec des propriétés SAL (langage de spécification). Cela met en évidence le fait que l'interprétation de l'utilisateur peut avoir un impact sur la sécurité en saisissant le mot de passe dans la mauvaise zone de texte par exemple.

Beckert et Beuster [19] produisent un modèle générique IOLTS (système de transition d'états) d'une application textuelle. Ils utilisent LTL (logique temporelle) pour décrire les propriétés des composants et les interpréter avec les IOLTS. Ils affinent le modèle pour garantir l'intégrité et pour prendre en compte le problème de la multiplicité des entrées (si l'utilisateur saisit à nouveau une donnée alors que le système n'a pas encore traité la dernière, lien avec les propriétés de latence vue précédemment) qui risque d'entraîner des failles de sécurité.

Arapinis *et al.* [10] présentent des propriétés de sécurité liées à l'utilisation d'un système de livraison de nourriture MATCH (*Mobilising Advanced Technology for Care at Home*). Ils définissent ces propriétés en utilisant différents formalismes tels que le langage de contrôle d'accès RW et la logique temporelle (LTL, TCTL, PCTL).

Johnson [78] étudie les propriétés de sécurité en matière de menaces qui peuvent se produire sur les systèmes mondiaux de navigation par satellites (SMNG). Il modélise les SMNG avec des processus Markoviens booléens (BDMP pour *Boolean Driven Markov Processes*) et intègre les menaces de sécurité au modèle.

3.2.6 Modélisation des systèmes interactifs

Les tableaux suivant synthétisent les études relatives à la modélisation des systèmes et des propriétés inhérentes aux formalismes utilisés. Ils répertorient donc les articles n'adressant pas (uniquement) la modélisation et la vérification des propriétés liées à l'interaction (présentées précédemment). Cela nous permet également de présenter certains cas d'études utilisés dans les travaux de recherche associés.

Des références de ces tableaux (celles en gras) sont des articles qui adressent uniquement la modélisation des systèmes. Ces articles ne couvrent pas les propriétés présentées précédemment. Nous ne présentons dans cette section que ces articles.

Nous trions les articles uniquement axés sur la modélisation en fonction du formalisme utilisé. L'objectif ici est de présenter succinctement les cas d'étude pour avoir une idée de leur maturité ainsi que de la précision de leur modélisation.

3.2.6.1 Algèbre de processus

Le tableau 3.5 synthétise les études relatives à la modélisation des systèmes par le biais de l'algèbre de processus.

	Modélisation systèmes
(AP) CSP	[44]
(AP) LOTOS	**[15]**
(AP) π-calculus	[10]
(AP) Prob. π-calc.	**[9]**
(AP) PEPA	[47]
(AP) TCBS'	**[27]**

TABLE 3.5 – Études relatives à la **modélisation** par l'**algèbre de processus**

Barbosa *et al.* [15] représentent un système de contrôle du trafic aérien constitué d'une tour de contrôle et trois avions modélisés par des interacteurs CNUCE. Ils utilisent un formalisme ad hoc, une approche générique de l'algèbre de processus, pour définir cette représentation.

Anderson et Ciobanu [9] construisent une abstraction du processus de décision Markovienne d'une spécification de programme exprimée avec une algèbre de processus probabiliste (en utilisant π-calculus). Ils utilisent ensuite cette abstraction pour vérifier la structure de la spécification, analyser la stabilité à long terme du système et fournir des conseils pour améliorer les spécifications si elles s'avèrent instables.

Bhandal *et al.* [27] présente le langage TCBS', fortement basé sur le *Timed Calculus of Broadcasting Systems* (TCBS). Ils donnent un modèle formel d'un modèle de coordination, le système Comhordú, dans ce langage.

3.2.6.2 Langage de spécification

Le tableau 3.6 synthétise les études relatives à la modélisation des systèmes par le biais des langages de spécification.

Calder *et al.* [40] étudient le *MATCH Activity Monitor* (MAM), un système omniprésent basé sur des règles et piloté par des événements. Ils modélisent séparément le comportement du système et sa configuration (ensemble de règles) avec Promela. Ils dérivent les règles Promela en des propriétés LTL pour vérifier les règles redondantes du système avec le vérificateur de modèle SPIN.

	Modélisation systèmes
(LS) SAL	[86] **[17]**
(LS) PVS	**[85]** [67] **[91]** [68]
(LS) Promela	**[40]**
(LS) HOPS	[58]
(LS/Ra) μCharts	[36]
(LS/Ra) Z	[35] [37]

TABLE 3.6 – Études relatives à la **modélisation** par les **langages de spécification**

Bowen et Hinze [34] présentent les premières étapes d'un travail utilisant des modèles de présentation pour concevoir un système d'information touristique. Ce système affiche une carte sur un support mobile de type smartphone.

Bass *et al.* [17] spécifient dans SAL les trois sous-systèmes de la protection de la vitesse de l'A320 : automatisation, avion et pilotes. Ce système hybride interactif a le potentiel de surprendre l'utilisateur en matière d'automatisation.

Masci *et al.* [85] spécifient le modèle de flux d'informations du DiCoT [29] en utilisant PVS. Ils utilisent trois concepts de modélisation (état du système, activité, tâche) pour cette spécification. Les auteurs utilisent l'exemple du service d'ambulance de Londres pour illustrer leur travail.

3.2.6.3 Processus de raffinement

Le tableau 3.7 synthétise les études relatives à la modélisation des systèmes par le biais des processus de raffinement.

	Modélisation systèmes
(LS/Ra) μCharts	[36]
(LS/Ra) Z	[35] [37]
(Ra) B/B événementiel	**[42]** **[101]** **[61]**

TABLE 3.7 – Études relatives à la **modélisation** par les **processus de raffinement**

Cansell *et al.* [42] spécifie une interface de vote électronique correspondant au modèle de vote unique transférable sans spécifier l'algorithme de comptage. Pour ce faire, ils utilisent la méthode B et un processus de raffinement.

Rukšėnas *et al.* [101] étudient les exigences globales liées aux interfaces de saisie de données des pompes à perfusion. Ils utilisent la méthode B événementiel et un processus de raffinement avec la plateforme Rodin pour spécifier ces exigences. Ces processus de raffinement permettent aux auteurs de vérifier si la spécification de

saisie du numéro de la pompe à perfusion Alaris GB valide les exigences globales d'une pompe à perfusion.

Geniet et Singh [61] étudient une interface humain-machine constituée de composants graphiques sous forme de widgets. Ils utilisent la méthode B événementiel et des processus de raffinement pour modéliser le système et analyser son comportement.

3.2.6.4 Système de transition d'états et logique temporelle

Le tableau 3.8 synthétise les études relatives à la modélisation des systèmes par le biais des systèmes de transition d'états et de la logique temporelle.

	Modélisation systèmes
(STE) FSM	[**117**]
(STE) UPPAAL	[**66**]
(STE) Colored PN	[**110**]
(STE) GTS	[**124**]
(STE) FSA	[**120**]
(STE) Event act. graph	[81]
(STE) ICO	[**110**]
(LT) LTL	[10] [**40**]
(LT) CTL	[67]

TABLE 3.8 – Études relatives à la **modélisation** par les **systèmes de transition d'états** et la **logique temporelle**

Harrison *et al.* [66] modélisent le système GAUDI [80] avec UPPAAL. Grâce au modèle UPPAAL, les auteurs peuvent explorer des scénarios de cas d'utilisation et vérifier les propriétés d'accessibilité par exemple.

Westergaard [124] utilise des systèmes de transition de jeu pour définir les visualisations du comportement des modèles formels. L'exemple d'un protocole d'interopérabilité pour les réseaux mobiles ad hoc pour mettre en évidence l'utilisation des visualisations.

Thimbleby et Gimblett [117] modélisent les possibilités d'interactions avec les entrées de données numériques des pompes à perfusion. Ils utilisent les machines à états finis et spécifient ceux-ci avec des expressions régulières pour modéliser les interactions.

Silva *et al.* [110] définit formellement un système et ses interactions WIMP et Post-WIMP avec les modèles ICO et les réseaux de Pétri colorés. Ces modèles leur

permettent d'analyser les propriétés inhérentes aux formalismes : invariants des transitions de place, vivacité et équité, et accessibilité.

Turner *et al.* [120] génèrent des modèles de présentation décrivant les séquences d'interactions basées sur des tâches et des widgets d'une pompe à perfusion. Cinq boutons (Up, Down, YesStart, NoStop, OnOff) et un affichage permettant des interactions avec l'utilisateur composent la pompe. Ils utilisent des automates à états finis pour modéliser ces séquences.

3.2.6.5 Autres

Le tableau 3.9 synthétise les études relatives à la modélisation des systèmes par le biais de formalismes ne rentrant pas dans la nomenclature que nous avons proposée ou par le biais d'outils.

	Modélisation systèmes
(autre) SAT	**[40]**
(autre) Mark. proc.	**[9]**
(autre) MAL	[41] [67]
(autre) GUM	[75]
(autre) BDMP	[78]
(outil) PVSio web	**[91]** [68]
ad hoc	[112] [28] [59] [10] [44] **[34]** **[117]** **[14]**

TABLE 3.9 – Études relatives à la **modélisation** par d'**autres** formalismes

Bhattacharya *et al.* [28] modélisent des claviers logiciels (claviers à l'écran) avec scanning et utilisent le modèle Fitts-Digraph [113] pour évaluer les performances de leur modèle et du système.

Sinnig *et al.* [112] décrivent un nouveau formalisme basé sur des ensembles d'ensembles partiellement ordonnés. Ils l'utilisent pour définir formellement des cas d'utilisation et des modèles de tâches.

Dix *et al.* [59] utilisent un formalisme ad hoc pour modéliser les états logiques de dispositifs physiques (interrupteurs, bouilloire électrique, etc.) et leurs effets numériques dans un autre modèle.

Oladimeji *et al.* [91] présentent PVSio-web, un outil qui étend le module PVSio de PVS avec un environnement graphique. Ils démontrent son utilisation en réalisant un prototype du système de saisie de données des pompes à perfusion.

Enfin, Banach *et al.* [14] envisagent d'utiliser un modèle B événementiel en conjonction avec un solveur SMT afin de prouver certains invariants sur un composant matériel, dédié à l'acquisition et à la fusion des entrées de divers capteurs vers la canne blanche d'une personne aveugle ou malvoyante (projet INSPEX).

3.2.7 Synthèse

Dans cette section, nous avons répertorié et trié les articles de la littérature spécialisée qui décrivent l'utilisation des méthodes formelles appliquées aux systèmes interactifs.

Grâce à cette section, nous avons vu que de très nombreux travaux existent quant à l'utilisation des méthodes formelles pour exprimer et vérifier les propriétés des systèmes interactifs ou pour spécifier ces systèmes. Toutes les classes de propriétés (comportement de l'utilisateur, principes cognitifs, interface humain-machine, sécurité) ont été étudiées, et ce, en prenant des exemples concrets. Tous les types de formalismes (algèbre de processus, langage de spécification, processus de raffinement, système de transition d'états, logique temporelle) de la nomenclature (figure 2.11, page 29) ont été utilisés pour vérifier des propriétés. Nous avons également vu des formalismes ad hoc ne rentrant pas dans cette nomenclature que nous avons proposée.

3.3 Analyse de la littérature

Dans cette section, nous analysons cette revue de la littérature afin de dégager les forces et faiblesses des méthodes formelles appliquées aux systèmes interactifs. Pour ce faire, nous synthétisons les tableaux que nous avons présentés dans la section précédente.

De cette synthèse, nous tirons les différentes conclusions de l'analyse des travaux existant afin de répondre à différentes questions :

- Quelles sont les propriétés liées à l'interaction les plus couvertes et celles qui sont les moins adressées ?
- Quels sont les formalismes les plus utilisés pour l'expression et la vérification des propriétés des systèmes informatiques interactifs, et y en a-t-il des nouveaux qui ont émergé spécifiquement pour ce type de propriétés ?

L'objectif ici est de confirmer la problématique expliquant nos questions de recherches concernant la formalisation des propriétés liées à la scène graphique et l'automatisation de leur vérification formelle.

Plusieurs classes de propriétés ont été étudiées et couvrent différents aspects des interactions.

À l'issue de notre revue de la littérature, nous avons mis en évidence plusieurs éléments que nous détaillons ci-dessous. Pour plus de clareté, ces éléments sont présentés sous la forme de "findings".

Finding 2. Il n'existe pas de cartographie de la couverture des propriétés liées à l'interaction par les méthodes formelles.

À travers cette revue de la littérature, nous remarquons qu'il n'existe pas de travaux synthétisant la couverture des propriétés liées à l'interaction par les méthodes formelles, qu'il s'agisse d'expression formelle ou de vérification formelle de ces propriétés. Nous avons donc réalisé ces travaux.

	Comportement utilisateur	Principes cognitifs	IHM scène graphique	IHM autres	Sécurité	Modélisation systèmes
Algèbre de processus	✓					✓✓
Langage de spécification	✓	✓		✓✓	✓	✓✓✓
Processus de raffinement				✓		✓
Système de transition d'états	✓			✓	✓	✓✓
Logique temporelle		✓		✓	✓	✓
Autres ad hoc	✓	✓		✓	✓	✓✓

TABLE 3.10 – Étude des méthodes formelles appliquées aux systèmes interactifs

Le tableau 3.10 synthétise les travaux que nous avons présentés dans le chapitre précédent. Nous avons répertorié 43 articles provenant des 7 éditions de FMIS ayant eu lieu sur la période 2006-2018. Il donne une répartition des articles dans une grille

d'analyse de notre choix. Nous notons : ✓ : 1-5 articles ; ✓✓ : 6-10 articles ; ✓✓✓ : 10+ articles.

3.3.1 Grande proportion de travaux sur la modélisation des systèmes

Nous soulignons la grande proportion d'articles qui adressent la modélisation des systèmes interactifs (tableaux 3.5, 3.6, 3.7 et 3.8 et colonne "Modélisation" du tableau 3.10).

En effet, nous constatons qu'environ 80% des articles adressant les méthodes formelles appliquées aux systèmes interactifs et leurs propriétés que nous avons répertoriés (34 articles sur 43) s'intéressent à la modélisation de cas d'étude. Ces modélisations concernent tout ou partie des cas d'études.

Nous notons également que plus de la moitié d'entre eux adresse spécifiquement la modélisation des propriétés inhérentes aux formalismes utilisés (invariant pour B, accessibilité pour les systèmes de transition, etc.). Ces articles n'adressent donc pas les propriétés liées à l'interaction.

3.3.2 Utilisation privilégiée de certains formalismes

En examinant les formalismes utilisés, principalement pour la modélisation des systèmes interactifs (tableaux 3.5, 3.6, 3.7 et 3.8), il apparaît que certains sont utilisés de manière majoritaire.

Nous pouvons voir que PVS et SAL sont les langages de spécification les plus utilisés. Sur les 14 articles qui utilisent des langages de spécification, nous constatons que SAL est le plus utilisé avec 5 articles qui l'utilisent. PVS est également très répandu, avec 4 articles qui l'utilisent. Ces deux derniers couvrent plus de la moitié des articles utilisant le langage de spécification.

Les modèles B et B événementiels sont les plus utilisés pour les processus de raffinement. 6 articles présentent des processus de raffinement et la moitié d'entre eux utilisent les modèles B et event-B. Nous trouvons 2 articles utilisant Z et 1 article utilisant μCharts.

3.3.3 Émergence de nouveaux formalismes "ad hoc"

À travers cette analyse, nous observons que la majorité des formalismes s'inscrivent dans la nomenclature (figure 2.11) que nous avons établie. Mais d'autres

formalismes n'ont pas pu être classés facilement dans l'une des familles proposées. Nous avons identifié 8 articles qui utilisent des formalismes ad hoc ou des formalismes que nous considérons hors de la nomenclature.

Parmi ces articles, nous trouvons, par exemple, la définition formelle de modèles de tâches et de cas d'utilisation en utilisant un formalisme ad hoc basé sur des ensembles d'ensembles partiellement ordonnés [112]. Nous trouvons également la modélisation de plusieurs dispositifs physiques avec un nouveau formalisme ad hoc permettant de représenter leur état logique (position d'un interrupteur par exemple) ainsi que leur effet numérique [59]. Un autre article présente la définition formelle de différentes émotions en utilisant un formalisme ad hoc [31]. Un article présente des propriétés de sécurité et les différents moyens (langage de contrôle d'accès RW, langage de requête ProVerif, appliqué π-calculus) de les formaliser [10].

3.3.4 Maturité des cas d'étude

16 des 43 articles se concentrent sur des cas "classiques" (ou cas d'école) et abordent la partie interface utilisateur (application web, application pour smartphone, système de vote électronique, distributeur automatique de billets, etc.) Ils permettent aux auteurs d'illustrer l'utilisation de plusieurs méthodes formelles et les propriétés inhérentes à celles-ci. Les systèmes sont modélisés, plusieurs propriétés, inhérentes aux formalismes ou aux systèmes, sont étudiées. Cependant, ces articles n'illustrent que les méthodes formelles et ne permettent pas aux auteurs de démontrer le passage à l'échelle potentiel de ces méthodes à l'aide de cas d'étude industriels conséquents comme des systèmes aéronautiques par exemple.

La pompe à perfusion est un dispositif médical critique pour la sécurité et est utilisée par 7 articles sur 43. Sur l'ensemble de ces articles, il s'agit souvent des mêmes auteurs. 3 d'entre eux étudient la partie données du système entier en le modélisant et vérifient certaines propriétés sur un sous-système seulement. 3 autres articles étudient le système complet. Ils modélisent un appareil en particulier (d'un industriel donné par exemple) ou ses spécifications afin de vérifier si cet appareil ou ces spécifications respectent les exigences globales d'une pompe à perfusion. Un autre article étudie les interactions possibles entre un utilisateur et le système. Les auteurs les modélisent sous la forme de séquences d'interaction correspondant aux tâches de l'utilisateur humain.

Cette approche a démontré la faisabilité des méthodes proposées, mais reste limitée. Nous notons que même si une pompe à perfusion est un système critique

pour la sécurité, les études réalisées pour ce système ne traitent pas nécessairement de ces aspects critiques pour la sécurité. En effet, seuls 3 articles se concentrent sur le système complet et ses exigences orientées vers la certification. Seuls ceux-là démontrent le passage à l'échelle des formalismes utilisés.

3.3.5 Propriétés de la scène graphique peu étudiées

Nous pouvons noter que même si plusieurs propriétés relatives aux interfaces humain-machines ont été étudiées (latence, utilisabilité, etc.), les propriétés de la scène graphique restent quant à elles extrêmement peu étudiées (0 article sur 43 pour FMIS), en tout cas, sur des cas d'étude concrets. Bien que certains articles proposent des formalismes permettant de définir des éléments graphiques ou des opérations entre éléments graphiques, nous n'avons pas identifié de travaux les appliquant à des cas d'étude concrets.

Nous avons souhaité consolider cette observation en procédant à l'analyse exhaustive des publications d'une autre conférence de référence dans le développement des applications interactives : EICS (*Engineering Interactive Computing Systems*). Parmi les 458 publications des occurrences d'EICS, nous dénombrons 5 publications (environ 1.1% des publications EICS) abordant des problématiques liées aux propriétés de la scène graphique.

Yanagida *et al.* [128], Tissoires et Conversy [118], Mirlacher *et al.* [87] ainsi que Bouzit *et al.* [33] présentent des outils et techniques permettant d'assister les concepteurs et programmeurs à concevoir des interfaces graphiques. Ces outils et techniques leur permettent aussi d'auto-générer des interfaces graphiques à partir des fichiers graphiques ainsi que de la modélisation du graphe de scène. Ces quatre publications ne correspondent donc pas à des techniques de vérification formelle de propriétés graphiques. Notons tout de même que Bouzit *et al.* [33] utilisent les variables graphiques de Bertin et Barbut [26] afin de définir un espace de conception pour les menus graphiques adaptatifs.

Chatty *et al.* [48] proposent une approche pour vérifier la visibilité des composants graphiques à partir du code du système informatique interactif. Cette approche est basée sur la vérification de la propriété d'opacité des éléments graphiques. Les auteurs analysent le code de l'application pour vérifier la valeur initiale de l'opacité d'un élément graphique et si cette propriété est mise à jour dans le programme. Cet article représente la seule publication de la conférence EICS adressant la vérification des propriétés graphiques. De plus, ces travaux ont été réalisés par l'équipe

de recherche dans laquelle nos travaux de thèse se déroulent. Ils ont constitué une première approche pour la vérification formelle des propriétés de la scène graphique que nos travaux ont contribué à élargir fortement.

Finding 3. Il n'existe pas de formalisme adéquat pour exprimer et vérifier formellement les propriétés de la scène graphique.

3.4 Problématique abordée par cette thèse

La dernière partie de l'analyse des travaux associés est celle qui nous intéresse le plus. Nous constatons en effet que les propriétés de la scène graphique des interfaces humain-machine ne sont pas étudiées. Afin de conclure sur notre état de l'art, nous pouvons justifier nos questions de recherche :

- Comment formaliser les éléments de la scène graphique des interfaces humain-machine ?
- Comment mécaniser la vérification des exigences graphiques des systèmes informatiques interactifs ?

3.4.1 Comment formaliser les éléments de la scène graphique des interfaces humain-machine ?

La partie des concepts de base adressant les propriétés liées à la scène graphique (section 2.2.3.5, page 25) a montré que quelques travaux existaient quant à la formalisation de propriétés graphiques ou à des pistes de formalisation.

Nous rappelons les variables graphiques de Bertin et Barbut [26] et Wilkinson [125] : coordonnées, taille, valeur, grain, couleur, orientation, forme et opacité. Ces variables constituent une première piste quant à la formalisation des éléments graphiques afin de vérifier par la suite des propriétés et exigences graphiques.

Nous rappelons également la logique d'intervalle de Randell et Cohn [94]. Celle-ci permet de raisonner quant à la spatialité en deux dimensions de régions que nous pouvons associer à des éléments graphiques. Cependant, ce raisonnement a pour base la notion de connexion entre deux régions. De ce fait, nous n'avons pas accès à la connexion des régions par rapport à leurs coordonnées, mais simplement à la connexion en tant que telle, qui constitue la relation de base de cette logique. De

plus, ces travaux permettent de raisonner sur des problématiques de position des éléments graphiques et non pas sur leur forme ou leur couleur par exemple.

Finalement, nous voyons qu'il existe quelques travaux de formalisation déjà existant. Cependant, pour les raisons exposées ci-dessus, nous pensons que ceux-ci ne sont pas assez précis pour adresser l'ensemble des propriétés graphiques qui pourraient nous intéresser et que nous pourrions baser sur les variables graphiques de Bertin et Barbut [26] et Wilkinson [125].

3.4.2 Comment mécaniser la vérification des exigences graphiques des systèmes informatiques interactifs ?

À travers l'état de l'art, nous avons identifié des travaux apportant des pistes quant à la formalisation des propriétés de la scène graphique. Cependant, dans les travaux associés nous n'avons repéré aucun article adressant d'une application concrète de ces pistes.

L'absence de formalisme dédié à toutes les variables graphiques des propriétés graphiques confirme qu'il n'existe pas de méthodes de vérification formelle des propriétés graphiques. De ce fait, il n'existe pas non plus d'outils implémentant ces méthodes afin de vérifier formellement des propriétés et exigences graphiques.

C'est pourquoi nous focaliserons également nos travaux sur cet objectif de mécanisation de vérification des exigences graphiques.

CHAPITRE 4

CAS D'ÉTUDE : SMALA ET TCAS

Nous avons justifié dans le chapitre précédent l'intérêt pour nous de répondre aux deux questions de recherche concernant la formalisation des propriétés graphiques et la vérification automatique des exigences graphiques des systèmes informatiques interactifs. Bien que des pistes existent quant à la formalisation des propriétés graphiques, celles-ci n'ont pas été suffisamment développées sur des cas concrets pour permettre l'automatisation de la vérification d'exigences graphiques.

Avant de nous pencher sur ces deux questions, nous introduisons dans ce chapitre le système qui servira de cas d'étude à nos travaux. L'intérêt ici pour nous est d'avoir un système critique avec des exigences graphiques que nous pourrons exprimer formellement et vérifier automatiquement. Étant chercheurs au sein de l'École Nationale de l'Aviation Civile, nous avons fait le choix de nous orienter vers un système aéronautique. Nous expliquerons également l'intérêt du choix du secteur aéronautique dans le cadre de nos travaux.

L'équipe Informatique Interactive développe un langage de programmation réactive, Smala. Nous profitons de ce jeune langage et des possibilités qu'il offre en y basant nos travaux. Ce chapitre sera également l'occasion de présenter ce langage et quelques-unes de ses particularités.

Le système d'alerte de trafic et d'évitement de collision ou TCAS (Trafic alert and Collision Avoidance System) est un système critique multimodal (interactions tactiles, visuelles et sonores) pouvant être embarqué dans certains instruments des cockpits d'aéronefs. Nous présenterons une implémentation de ce système réalisée avec le langage Smala, les spécificités de son interface ainsi que les exigences graphiques que nous avons extraites d'une norme et qu'il doit absolument respecter.

4.1 Contexte des travaux de thèse

Le laboratoire de l'École Nationale de l'Aviation Civile contribue à la recherche dans le domaine aéronautique. De ce fait, l'aéronautique est au cœur de ces travaux de thèse.

Le secteur aéronautique fait intervenir des métiers et systèmes très nombreux et variés. Le transport de personnes étant un des objectifs de ce secteur, cela en fait un secteur critique. De ce fait, c'est également un secteur où la certification est omniprésente. L'ISO [1] définit la certification comme suit : "Assurance écrite (sous la forme d'un certificat) donnée par une tierce partie qu'un produit, service ou système est conforme à des exigences spécifiques."

L'aéronautique est donc un secteur dans lequel les exigences des systèmes sont écrites et ces systèmes doivent absolument s'y conformer. Pour ce faire, il est possible de soumettre les systèmes à des phases de tests. Cependant, depuis 1992 la norme DO-178B [111] autorise et présente, par le biais de deux paragraphes, l'usage des méthodes formelles pour la certification de normes aéronautiques. En 2011, son évolution, la norme DO-178C [106] présente cet usage des méthodes formelles dans une annexe (DO-333 [107]) qui y est entièrement dédiée.

Le secteur de l'aéronautique a donc su profiter de l'avancée de la recherche scientifique en matière de vérification formelle afin d'améliorer ses processus de certification. Ceci rend donc légitime l'utilisation des méthodes formelles pour vérifier les exigences, graphiques ou non, des systèmes informatiques humain-machine dans le secteur aéronautique.

L'équipe Informatique Interactive poursuit des recherches sur les systèmes humain-machine aéronautiques, en combinant applications concrètes et travaux théoriques, le tout en développant Smala, un langage de programmation réactive. Nous incluons donc nos travaux dans le cadre de ce langage de programmation. Deux axes constituent cette équipe :

- L'axe "Interaction Humain-Machine" (IHM) conduit des recherches visant à améliorer la qualité et la performance des visualisations et des interactions au sein du système hybride humain-machine. Il s'agit notamment de concevoir des interactions robustes qui ne s'effondrent pas dans les situations opérationnelles non nominales, au sein des systèmes cyber-humains, pour lesquelles les frontières entre humains et technologies s'amenuisent au point où ces dernières

1. `https://www.iso.org/`

deviennent des extensions naturelles de l'humain.

- L'axe "Ingénierie des Systèmes Interactifs" (ISI) conduit des recherches visant à améliorer la conception de systèmes interactifs par l'élaboration d'outils conceptuels (modèles, théories) et concrets (langages de programmation, éditeurs) utilisables. Son approche originale consiste à refonder les principes de la programmation avec un modèle unifié des systèmes interactifs, fondé sur le concept de couplage entre processus.

Les travaux de cette thèse se situent à mi-chemin entre les deux axes dans le sens où l'objectif est de faire le lien entre les méthodes formelles (ingénierie des systèmes interactifs) et les propriétés de la scène graphique (interaction humain-machine).

4.2 Smala

L'équipe Informatique Interactive développe le langage de programmation réactive Smala [2] [83]. Smala permet de décrire de manière déclarative une application interactive selon le paradigme réactif. Une application développée en Smala contient un ensemble de composants où le contrôle est réduit à la transmission d'activation entre ces composants. Ce langage nous permet d'accéder facilement aux composants graphiques et à leurs propriétés graphiques.

Smala est constitué de composants qui sont organisés sous forme d'arbre. Lors de l'exécution, un composant peut être activé, ce qui déclenche une action effectuée par le composant. Afin de présenter Smala, nous introduisons trois principes de ce langage.

4.2.1 Gestion du contrôle

Le premier principe concerne la gestion du contrôle qui est dédié aux applications interactives. Trois composants de base dédiés à la gestion du contrôle existent dans le langage :

- le couplage (composant **binding**, noté `->`) : une relation d'activation unidirectionnelle entre deux composants,
- l'assignation (composant **assignment**, noté `=:`) : une transmission de la valeur d'un composant à un autre composant,

2. `http://smala.io/`

- le flot de données (composant **connector**, noté `=:>`) : une relation d'activation unidirectionnelle entre deux composants et une transmission d'une valeur de composant à un autre composant.

```
7  Component root {
8      Frame f ("Smala example", 0, 0, 400, 400)
9
10     FillColor fc (255, 0, 0)
11     Rotation r (0, 0, 0)
12     Rectangle rect (0, 0, 0, 0, 0, 0)
13
14     f.width / 2 =:> rect.width
15     f.height / 2 =:> rect.height
16     rect.width / 2 =:> rect.x
17     rect.height / 2 =:> rect.y
18     rect.width / 2 + rect.x =:> r.cx
19     rect.height / 2 + rect.y =:> r.cy
20
21     AssignmentSequence as_press (1) {
22         45 =: r.a
23         0 =: fc.r
24         0 =: fc.g
25         255 =: fc.b
26     }
27     rect.press -> as_press
28
29     AssignmentSequence as_release (1) {
30         0 =: r.a
31         255 =: fc.r
32         0 =: fc.g
33         0 =: fc.b
34     }
35     rect.release -> as_release
36 }
```

Code 4.1 – Exemple de code Smala

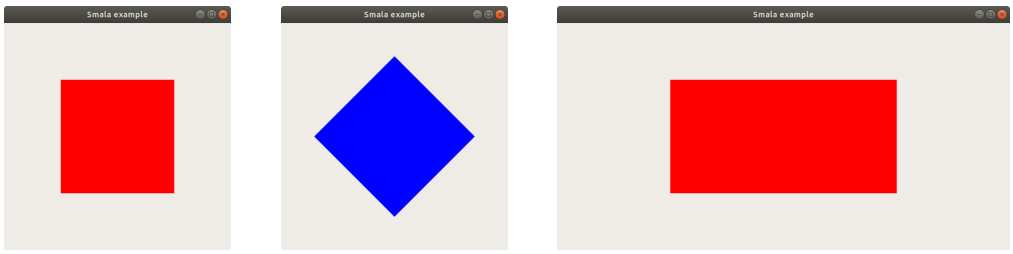
Le code 4.1 ci-dessus présente l'utilisation des trois composants de contrôle de base. La figure 4.1.a présente l'application interactive dans son état initial. Lorsque nous pressons (resp. relâchons) le rectangle `rect`, le **binding** active la séquence d'assignation `as_press` (resp. `as_release`) et tous les **assignements** dans celui-ci.

La figure 4.1.b (resp. 4.1.a) présente l'effet de chaque assignation contenue dans la séquence d'assignation `as_press` (resp. `as_release`). Lorsque nous modifions la

hauteur (resp. la largeur) de la fenêtre `f`, un **connector**, met à jour la hauteur (resp. la largeur) du rectangle `rect`. La sémantique des connectors qui impose de propager les activations De ce fait, lorsque la hauteur (resp. la largeur) du rectangle `rect` est mise à jour, la propagation de l'activation met à jour l'ordonnée (resp. l'abscisse) de l'origine de `rect`. La figure 4.1.c présente l'effet des connectors lorsque nous agrandissons la fenêtre.

4.2.2 Composants graphiques

Le deuxième principe concerne les composants graphiques. Les composants graphiques de Smala sont basés sur la norme SVG, nous avons donc accès à plusieurs composants graphiques de base (rectangle, ellipse, chemin, ...) et à leurs variables graphiques : coordonnées (axes x et y), taille (longueur, largeur, rayon x et y), couleur (système RGB), orientation (transformations géométriques) et transparence (coefficient d'opacité).



a. État initial b. Rectangle pressé c. Largeur augmentée

FIGURE 4.1 – Exemple d'application interactive de base en Smala

La figure 4.1 et le code associé ci-dessus (code 4.1) présentent les différentes variables graphiques auxquelles nous avons accès grâce aux composants graphiques Smala basés sur la norme SVG. Dans cet exemple, nous pouvons voir les effets lors de la modification des coordonnées (axe des x), de la taille (largeur), de la couleur (rouge/bleu) et de l'orientation (rotation de 45°).

4.2.3 Graphe d'activation

Le troisième principe concerne le graphe d'activation de Smala. Il s'agit d'une structure contenant tous les composants (composants graphiques inclus) du programme et qui permet de modéliser la propagation des événements et donc l'ordre d'exécution. Nous avons fait une abstraction de ce graphe afin d'obtenir un graphe

de scène enrichi par les informations liées au flot de données et utiles pour la vérification des propriétés graphiques. Ce graphe de scène enrichi, dans lequel nous représentons la scène graphique et le flux de données, permet de modéliser une application Smala.

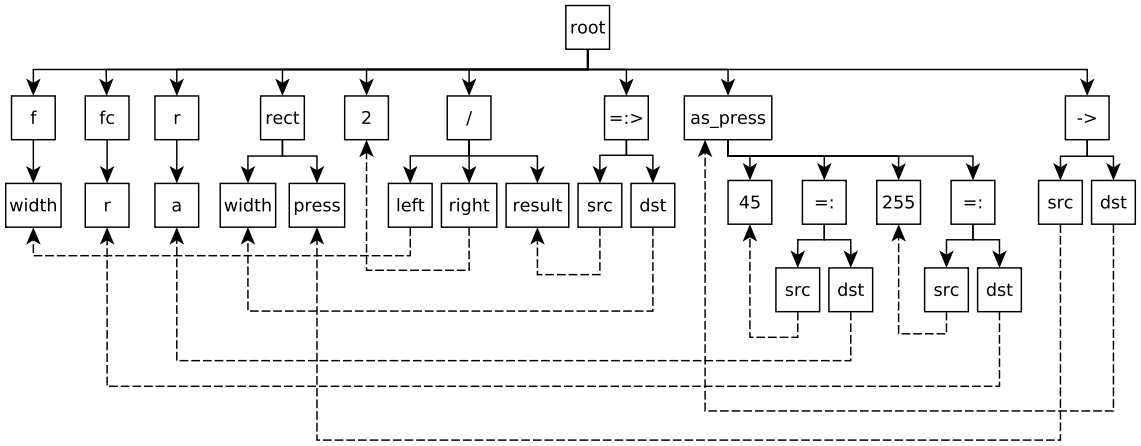


FIGURE 4.2 – Graphe de scène enrichi de l'exemple Smala (version allégée)

La figure 4.2 donne une version allégée du graphe de scène enrichi pour l'exemple Smala présenté ci-dessus (code 4.1 et figure 4.1).

Une flèche vers le bas représente les liens parents-enfants. La source de la flèche est le parent, la destination est l'enfant. Un enfant ne peut avoir qu'un seul parent, mais un parent peut avoir plusieurs enfants. Une flèche en pointillés représente un lien de référence. La destination de la flèche est le nœud utilisé comme référence pour la source de la flèche.

4.2.4 Lien entre composants graphiques et graphe d'activation

Le graphe d'activation, et plus précisément la sémantique de son parcours, a une influence sur l'ordre des activations des dépendances et aussi sur l'affichage de la scène graphique. La sémantique de parcours du graphe est le parcours *first depth* : parcours d'abord en profondeur puis de gauche à droite.

Nous présentons deux codes (codes 4.2 et 4.3) Smala proches avec une légère différence dans le graphe d'activation. Cela nous permet d'illustrer le parcours du graphe et son impact sur la scène graphique.

La scène graphique de cette application contient trois composants graphiques dont le contour est noir (`OutlineColor oc`) : un cercle rouge (`FillColor fc1` suivi

de `Circle c`), un carré vert (`FillColor fc1` suivi de `Rectangle carre`) et un rectangle bleu (`FillColor fc2` suivi de `Rectangle rect`).

Dans les deux codes, nous avons créé un composant non graphique qui contient les composants graphiques (`Component rectangles`). Celui-ci permet notamment de contrôler la propagation de la couleur et ainsi, un `FillColor` créé dans un `Component` n'impactera aucun élément graphique hors de ce `Component`. Ainsi, le `OutlineColor` qui n'est dans aucun `Component` agit sur le reste du code.

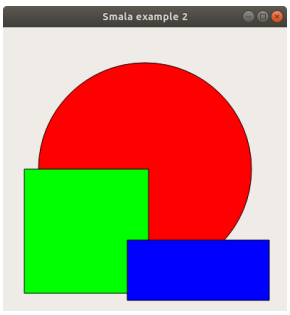
```
_main_
Component root {
    Frame f ("Smala example 2", ↩
        0, 0, 400, 400)
    OutlineColor black (0, 0, 0)
    FillColor red (255, 0, 0)
    Circle c (200, 200, 150)
    Component rectangles {
        FillColor green (0, 255, ↩
            0)
        Rectangle carre (30, 200, ↩
            175, 175, 0, 0)
        FillColor blue (0, 0, 255)
        Rectangle rect (175, 300, ↩
            200, 85, 0, 0)
    }
}
```

Code 4.2 – Exemple de code Smala

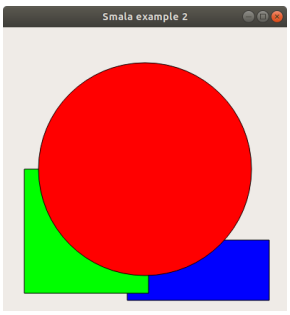
```
_main_
Component root {
    Frame f ("Smala example 2", ↩
        0, 0, 400, 400)
    OutlineColor black (0, 0, 0)
    Component rectangles {
        FillColor blue (0, 0, 255)
        Rectangle rect (175, 300, ↩
            200, 85, 0, 0)
        FillColor green (0, 255, ↩
            0)
        Rectangle carre (30, 200, ↩
            175, 175, 0, 0)
    }
    FillColor red (255, 0, 0)
    Circle c (200, 200, 150)
}
```

Code 4.3 – Exemple de code Smala

La figure 4.3.a (resp. 4.3.b) donne la scène graphique de l'exemple de code Smala 4.2 (resp. 4.3).



a. Scène graphique du code 4.2

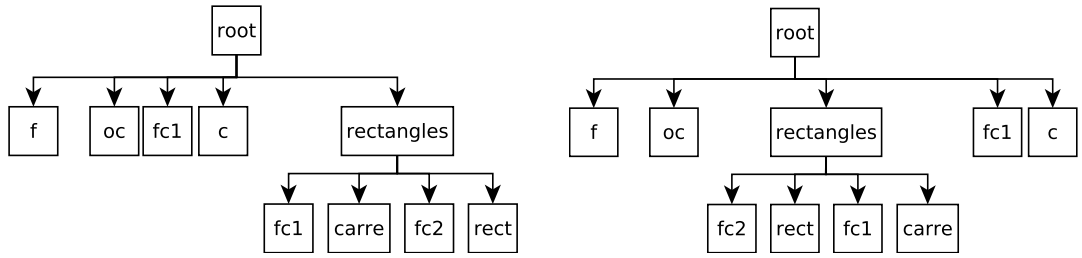


b. Scène graphique du code 4.3

FIGURE 4.3 – Scène graphique des exemples de codes Smala 4.2 et 4.3

La scène graphique correspondant au code 4.2 (figure 4.3.a) affiche en premier (ici, affiché en premier signifie élément graphique présent sur la première couche d'affichage et donc au dernier plan de l'interface) le cercle rouge, en deuxième le carré vert et en troisième le rectangle bleu. Au contraire, la scène graphique correspondant au code 4.3 (figure 4.3.b) affiche en premier le rectangle bleu, en deuxième le carré vert et en troisième le cercle rouge.

La figure 4.4.a (resp. 4.4.b) donne le graphes de scène enrichi (graphe d'activation avec des informations de la scène graphique) de l'exemple de code Smala 4.2 (resp. 4.3).



a. Graphe de scène enrichi du code 4.2 b. Graphe de scène enrichi du code 4.3

FIGURE 4.4 – Graphe de scène enrichi des exemples de codes Smala 4.2 et 4.3

Sur ces deux graphes de scène enrichis, `oc` (correspondant à `OutlineColor oc`) est à gauche des composants graphiques (`c`, `rect` et `carre`). Nous voyons sur les figures 4.3.a et 4.3.b que tous les composants graphiques possèdent des contours noirs.

La scène graphique du code 4.3 (figure 4.3.b) contient le cercle rouge au premier plan. Ainsi, cela présente le parcours en profondeur du graphe d'activation, partie du graphe de scène enrichi, en premier car les composants graphiques `carre` et `rect` sont contenus dans le `Component rectangles` qui est au même niveau dans le graphe de scène enrichi que le cercle `c`.

4.3 TCAS

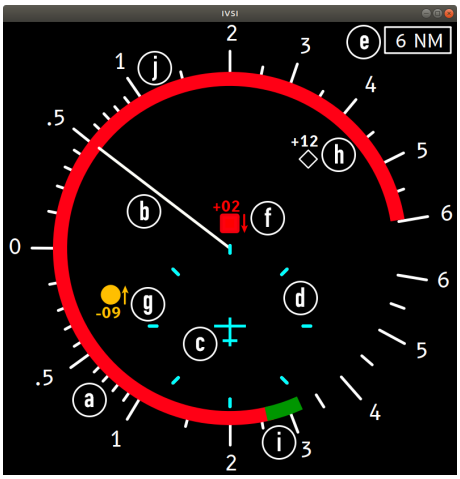
Le TCAS (*Trafic alert and Collision Avoidance System*) est un système aéronautique ayant pour objectif d'améliorer la sécurité aérienne en prévenant les collisions entre aéronefs (avions, hélicoptères, etc.). Il est obligatoire sur tous les avions commerciaux (transportant plus de 19 passagers ou pesant plus de 5.7 tonnes en Europe). La spécification technique ED-143 [3] le définit et on peut trouver dans

[126] des informations détaillées de différents scénarios d'utilisation, la logique de fonctionnement ou les différents affichages du système.

Par le biais de calculs effectués sur les informations reçues du trafic environnant, le TCAS fournit deux services aux pilotes : les TAs (*trafic alerts*) et les RAs (*resolution advisories*). Les TAs permettent d'informer les pilotes du type de trafic environnant (*threat aircraft*, *intruder aircraft*, *proximate aircraft*, *other aircraft*) en y associant une symbolique (forme et couleur). Les RAs permettent de notifier les pilotes des manœuvres à effectuer afin d'éviter tout risque de collision (atteindre une vitesse verticale donnée, conserver la vitesse verticale actuelle). Le système fournit ces deux services sous forme de notifications sonores et visuelles, ce qui fait du TCAS un **système critique multimodal**.

Un des composants du TCAS est l'instrument intégré au cockpit fournissant la visualisation. Le TCAS peut être intégré par exemple :

- au PFD (*Primary Flight Display*) où il ne fournira des visualisations que pour les RAs par le biais de trapèzes rouges indiquant le tangage de l'avion à éviter ainsi qu'une jauge rouge (resp. verte) indiquant la vitesse verticale à éviter (resp. à suivre),
- au ND (*Navigation Display*) où il ne fournira que des visualisations pour les TAs par le biais d'une symbolique particulière (identique à celle de l'IVSI présentée ci-après),
- à l'IVSI (*Instantaneous Vertical Speed Indicator*) où il pourra fournir les RAs, les TAs ou les deux en simultané.



a. IVSI - Compteur de vitesse verticale (ppm×1000)
b. IVSI - Aiguille de vitesse verticale
c. TA - Propre position relative ou avion de référence (*own aircraft*)
d. TA - Portée de 2 NM autour de *own aircraft*
e. TA - Échelle d'affichage du trafic
f. TA - *threat aircraft*
g. TA - *intruder aircraft*
h. TA - *proximate aircraft*
i. RA - Vitesse verticale à atteindre
j. RA - Vitesse verticale à éviter

FIGURE 4.5 – TCAS intégré à l'IVSI et réalisé en Smala

Nous présentons l'IVSI (*Instantaneous Vertical Speed Indicator*, figure 4.5), qui indique aux pilotes la vitesse verticale actuelle de leur aéronef, c'est-à-dire la vitesse instantanée de montée ou de descente.

Cet instrument accueille aussi la visualisation des informations élaborées par le TCAS. Ainsi, l'affichage conjoint d'un arc rouge, indiquant au pilote la plage de vitesses verticales à éviter, et d'un arc vert désignant la vitesse cible qu'il doit atteindre, représente la visualisation d'un RA. Sur la figure 4.5 par exemple, le RA informe le pilote que la plage de vitesses verticales $[-2500; +6000]$ ppm doit être évitée et que la plage de vitesses verticales $]-3250; -2500]$ ppm doit être atteinte. En effet, afin d'éviter la collision frontale avec le *threat aircraft* (figure 4.5 symbole f) qui est actuellement en descente et vole à peine 200 pieds au-dessus de l'aéronef de référence (*own aircraft*), le pilote doit amorcer une descente d'urgence.

Concernant l'affichage du trafic, prenons pour exemple le symbole f :

- La forme et la couleur indiquent le type de trafic environnant, son niveau de menace. Par ordre de menace croissante, nous avons : *other aircraf* (losange blanc ou cyan vide), *proximate aircraf* (losange blanc ou cyan plein), *intruder aircraf* (cercle jaune ou ambre plein), *intruder aircraf* (carré rouge plein).
- Le texte associé indique l'altitude relative du trafic, c'est-à-dire l'altitude séparant le trafic de l'aéronef de référence. Si le trafic est en dessous (resp. au-dessus) de l'aéronef de référence, l'altitude relative est négative (resp. positive) et indiquée par un `-` (resp. `+`) suivi de deux digits (représentant des centaines de pieds), le tout dans la couleur du symbole du trafic.
- La flèche associée indique la vitesse verticale du trafic. Une flèche vers le haut (resp. vers le bas) indique une vitesse verticale supérieure à 500 ppm (resp. inférieure à -500 ppm).

4.3.1 Données d'entrée de l'interface

L'évolution de la scène graphique de l'interface du TCAS intégré à l'IVSI est dynamique. Cette évolution et cette dynamicité entrainent le changement, l'apparition ou la disparition des composants graphiques en fonction de données d'entrée. Nous présentons certaines de ces données et illustrons leur représentation sur l'interface dans le tableau 4.1.

Nous cherchons à vérifier les exigences de notre interface, et ce, sur l'ensemble de ses scénarios de fonctionnement. De ce fait, il faut nous intéresser à ce qui la fait évoluer, car la vérification pourra dépendre de ces données d'entrée.

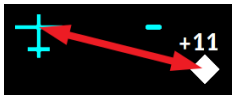
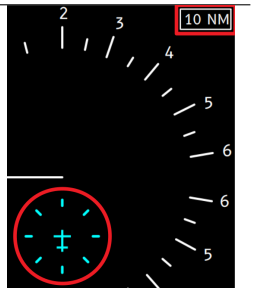
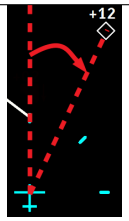
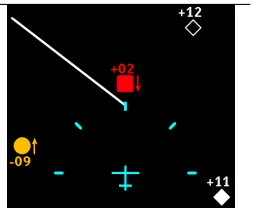
Donnée d'entrée	Informations	Plage de valeur	Illustration
Portée (*range*)	Distance entre l'avion de référence (*own_aircraft*) et le trafic environnant.	$[0; 128]NM$ (pas de $1/12NM$)	
Altitude relative (*relative altitude*)	Distance verticale entre l'avion de référence (*own_aircraft*) et le trafic environnant.	$\pm 12700NM$ (pas de $1/16NM$)	
Sens vertical (*vertical sense*)	Sens de la vitesse verticale	$\{0, 1, 2\}$	
Portée actuelle (*current range*)	Échelle de l'affichage de la portée.	$[6; 40]$ NM	
Relevèment (*bearing*)	Angle entre le cap de l'avion de référence (*own aircraft*) et le trafic environnant.	$\pm 180°$ (pas de $0.2°$)	
Niveau de menace (*threat level*)	Type de trafic environnant (*other*, *proximate*, *intruder*, *threat*)	$\{0, 1, 2, 3, 4\}$	

TABLE 4.1 – Données d'entrée de l'interface du TCAS intégré à l'IVSI

4.3.2 Exemple de scénario : Approche frontale

Avant de donner des exigences graphiques du TCAS, ce qui est le point nous intéressant dans le cadre de cette thèse, nous présentons rapidement un exemple de scénario du TCAS. Cet exemple permet d'expliquer comment l'interface réagit

à l'environnement extérieur du système. Nous expliquons chacune des phases de ce scénario à l'aide d'une schématisation des aéronefs (le cyan représentant l'aéronef de référence, le gris l'aéronef correspondant au trafic) et de leur course verticale, d'une représentation de l'interface du TCAS intégré à l'IVSI et de détails écrits avec entre parenthèses les équivalences sur l'interface.

Nous prenons le cas d'une approche frontale entre deux aéronefs. Ce cas est hypothétique et ne devrait en théorie jamais se produire en conditions réelles.

Dans ce cas hypothétique, nous supposons qu'en phase initiale (figure 4.6) les deux aéronefs sont relativement à la même altitude, le trafic étant légèrement au-dessus de l'aéronef de référence (texte blanc `+01` au-dessus du symbole du trafic indiquant une altitude relative de 100 pieds). Leur vitesse verticale est négligeable (aiguille du compteur quasiment à 0 ppm et aucune flèche à droite du symbole du trafic), ils se rapprochent donc frontalement sur le même plan. L'autre aéronef ne représente à ce moment précis aucune menace (trafic représenté par un losange blanc vide correspondant à *other aircraft*). Cela s'explique par le fait qu'il est à environ 6 NM de l'aéronef de référence (indication de l'échelle 6 NM en haut à droite de l'interface et losange quasiment au bord de l'indicateur de vitesse verticale).

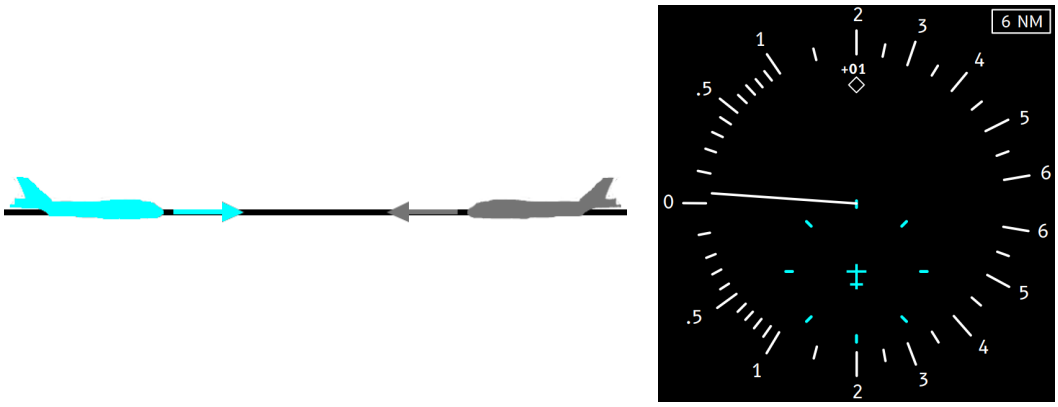


FIGURE 4.6 – Phase initiale de l'approche frontale

Lorsque l'autre aéronef est trop proche de celui de référence en distance et altitude (figure 4.7, celui-ci devient un trafic de niveau *threat aicraft* (représenté par un carré rouge plein). Dans cette configuration, ceci déclenche le RA *Descend. Descend.* (arc rouge sur la plage de vitesse verticale $[-1500; +6000]$ ppm et arc vert sur la plage de vitesse verticale $[-2000; -1500]$). Ce choix de RA est logique, le trafic est à 100 pieds au-dessus de l'aéronef de référence (texte rouge `+01` au-dessus du symbole du trafic) et n'indique toujours pas de sens vertical de navigation (aucune flèche à droite du symbole du trafic).

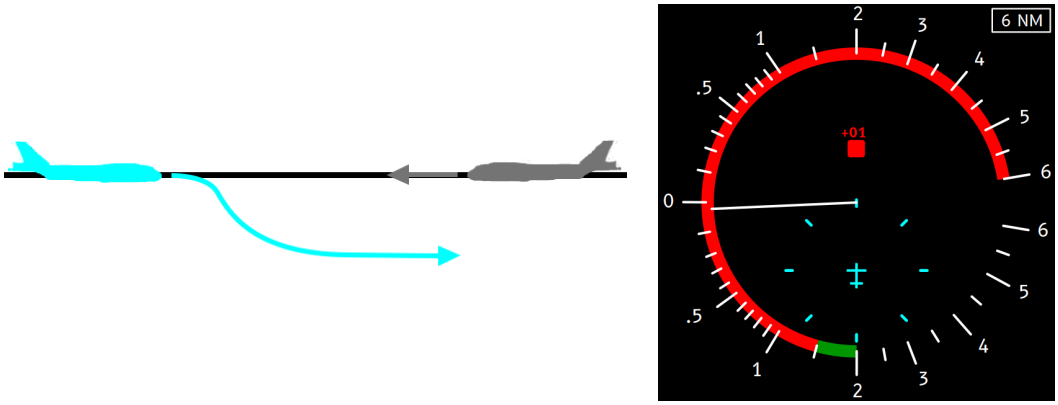


FIGURE 4.7 – Premier RA : *Descend. Descend.*

Dans la phase suivante (figure 4.8), nous avons la vue sur la manœuvre opérée par le trafic. Il s'agit une manœuvre de descente (flèche rouge vers le bas à droite du symbole du trafic). Le trafic s'est déjà rapproché de l'aéronef de référence et est désormais à la même altitude (texte rouge `+00` au-dessus du symbole du trafic). Cette situation est critique car il est impossible de savoir à quelle vitesse descend le trafic et le risque de collision est donc accentué.

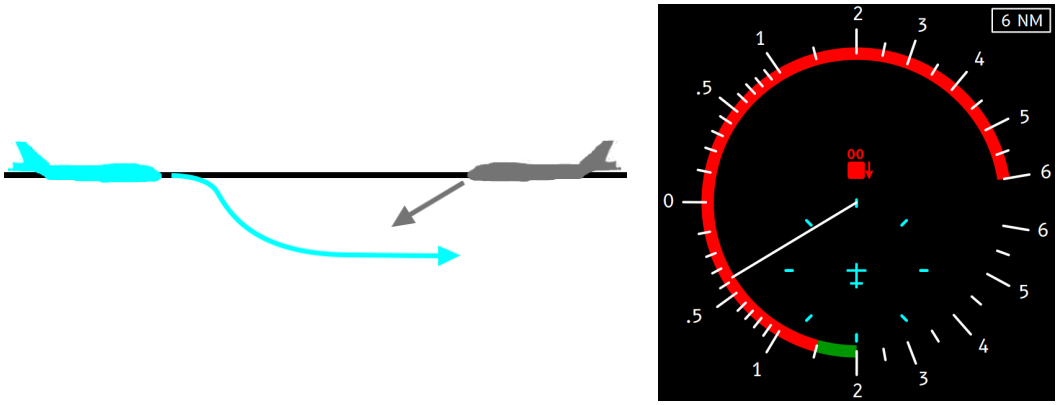


FIGURE 4.8 – Indication de la vitesse verticale du trafic : descente

De ce fait, le TCAS propose un nouveau RA, *Climb. Climb.* (arc rouge sur la plage de vitesse verticale $[-6000; +1500]$ ppm et arc vert sur la plage de vitesse verticale $[+1500; +2000]$), à l'aéronef de référence (figure 4.9) afin de prévenir la collision. Le trafic est toujours en phase de descente (flèche rouge vers le bas à droite du symbole du trafic) et est désormais passé à 100 pieds sous l'aéronef de référence (texte rouge `-01` en dessous du symbole du trafic). Il convient donc pour l'aéronef de référence de respecter le RA proposé par le TCAS afin d'éviter la collision avec le traffic environnant présentant un risque de collision.

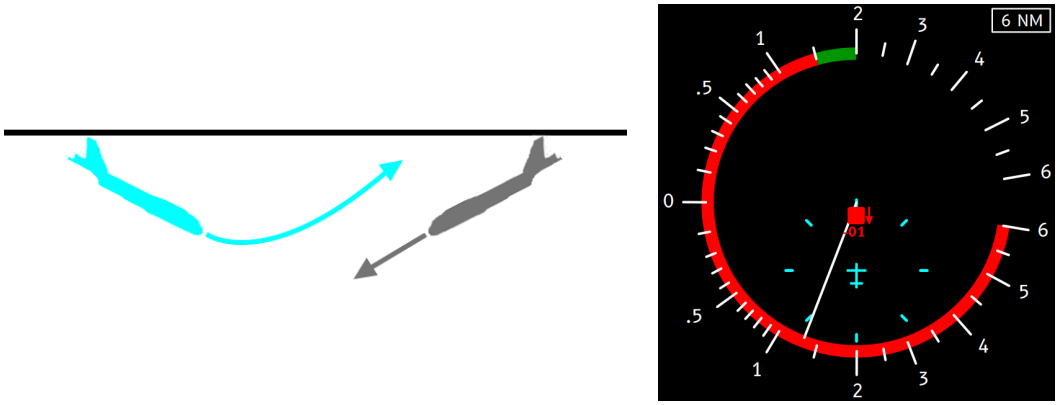


FIGURE 4.9 – Deuxième RA : *Climb. Climb.*

Enfin, après la manœuvre de montée de l'aéronef de référence, le risque de collision est écarté (figure 4.10). Nous constatons que le RA a disparu (plus aucun arc rouge ou vert sur le compteur de vitesse verticale) et le danger causé par le trafic est écarté, celui-ci redevient de niveau *intruder aicraft* (représenté par un cercle ambre plein).

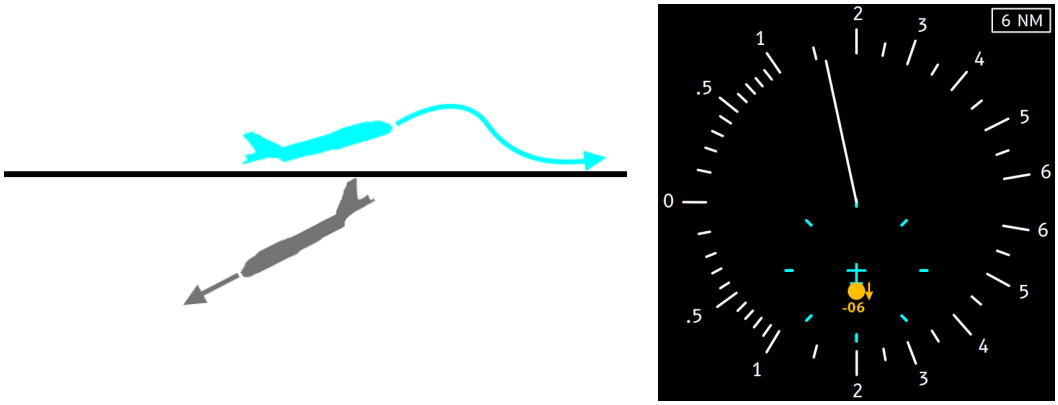


FIGURE 4.10 – Risque de collision écarté

4.3.3 Exigences graphiques du TCAS

Nous nous intéressons ici à l'intégration du TCAS à l'IVSI. Nous avons donc extrait de la spécification technique ED-143 [3] des exigences relatives à la visualisation du TCAS, applicables à l'IVSI :

- E_1 : *The traffic display shall contain a symbol representing the location of the own aircraft. The colour of the symbol shall be either white or cyan and different than that used to display proximate or other traffic.*

- E_2 : *The inner range ring shall not be solid and shall be comprised of discrete markings at each of the twelve clock positions. The markings shall be the same colour as the own aircraft symbol and of a size and shape that will not clutter the display.*

- E_3 : *The symbol for an RA shall be a red filled square.*

- E_4 : *The symbol for a TA shall be an amber or yellow filled circle.*

- E_5 : *The symbol for Proximate Traffic shall be a white or cyan filled diamond. The colour of the Proximate Traffic symbol should be different than that used for the own aircraft symbol to ensure the symbol is readable.*

- E_6 : *The symbol for Other Traffic shall be a white or cyan diamond, outline only. The colour of the Other Traffic symbol should be different than that used for the own aircraft symbol to ensure the symbol is readable.*

- E_7 : *For an intruder above own aircraft, the tag shall be placed above the traffic symbol and preceded by a "+" sign ; for one below own aircraft, the tag shall be place below the traffic symbol and preceded by a "-" sign. The colour of the data tag shall be the same as the symbol.*

- E_8 : *A vertical arrow shall be placed to the immediate right of the traffic symbol if the vertical speed of the intruder is equal to or greater than 500 fpm, with the arrow pointing up for climbing traffic and down for descending traffic. The colour of the arrow shall be the same as the traffic symbol.*

- E_9 : *A green "fly-to" arc shall be used to provide a target vertical speed whenever a change in the existing vertical speed is desired or when an existing vertical speed must be maintained.*

- E_{10} : *The red arcs on the RA/VSI shall indicate the vertical speed range that must be avoided to maintain or attain the TCAS-desired vertical miss distance from one or more intruders.*

Le tableau 4.2 présente ces exigences, représentatives de la complexité de la norme graphique. Nous avons choisi d'utiliser trois variables graphiques de Bertin et Barbut [26] afin de découper nos exigences graphiques et faire apparaître leurs différentes composantes séparément : la forme, la couleur et la position.

Ex.	Information	Forme	Couleur	Position
E_1	Propre position relative (*own aircraft*)	Avion (3 traits)	Blanc ou cyan ($\neq$ couleur *proximate / other aircraft*)	Centré horizontalement, 1/3 hauteur affichage
E_2	Portée de 2 NM autour de *own aircraft*	8 traits en cercle	Couleur *own aircraft*	Centré sur *own aircraft*, rayon en fonction de la portée
E_3	TA - Trafic de type *threat aircraft*	Carré plein	Rouge	Zone délimitée par le compteur de vitesse
E_4	TA - Trafic de type *intruder aircraft*	Cercle plein	Jaune ou ambre	Zone délimitée par le compteur de vitesse
E_5	TA - Trafic de type *proximate aircraft*	Losange plein	Blanc ou cyan ($\neq$ couleur *own aircraft*)	Zone délimitée par le compteur de vitesse
E_6	TA - Trafic de type *other aircraft*	Losange vide	Blanc ou cyan ($\neq$ couleur *own aircraft*)	Zone délimitée par le compteur de vitesse
E_7	TA - Altitude relative (en centaine de pieds)	"+" / "-" et deux chiffres	Couleur trafic	Au-dessus / en dessous du symbole du trafic
E_8	TA - Sens vertical	Flèche verticale (haut ou bas)	Couleur trafic	À droite du symbole du trafic
E_9	RA - Vitesse à atteindre	Arc de cercle	Vert	Compteur de vitesse
E_{10}	RA - Vitesse à éviter	Arc de cercle	Rouge	Compteur de vitesse

TABLE 4.2 – Liste des exigences graphiques du TCAS intégré à l'IVSI

4.4 Synthèse

Dans ce chapitre, nous avons présenté le contexte de la thèse qui s'inscrit dans les travaux à la connexion des axes Interaction Humain-Machine et Ingénierie des Systèmes Interactifs de l'équipe Informatique Interactive du laboratoire de l'École Nationale de l'Aviation Civile.

D'une part, ce contexte de travaux de l'équipe Informatique Interactive a augmenté l'intérêt porté sur l'étude formelle (formalisation et vérification) des propriétés de la scène graphique des systèmes interactifs. D'autre part, le contexte du secteur

aéronautique a porté le choix du cas d'étude vers un système critique de ce secteur : le TCAS intégré à l'IVSI.

Nous avons introduit le langage de programmation réactive Smala. Ce langage nous permet d'accéder facilement aux composants graphiques et à leurs propriétés graphiques.

Nous avons extrait de la spécification technique ED-143 [3], décrivant le TCAS intégré à l'IVSI, dix exigences graphiques de cette interface. Nous les avons ordonnées dans un tableau en séparant dans les colonnes leurs différentes composantes graphiques selon leur forme, leur couleur et leur position. Celles-ci serviront de base comme exigences à vérifier dans le cadre de ces travaux de thèse.

CHAPITRE 5

VÉRIFICATION DES PROPRIÉTÉS GRAPHIQUES

Les travaux de recherche que nous avons analysés au cours de l'état de l'art qui traitent des propriétés de la scène graphique ne vont pas assez en profondeur dans leur formalisation et leur vérification. Notre premier objectif dans ce chapitre est donc de proposer un langage permettant d'exprimer formellement des exigences graphiques.

Nous présentons notre formalisme et nos opérateurs graphiques : ce sont les opérateurs qui nous permettront de définir formellement des exigences graphiques. Pour cela, nous définissons au préalable les formes qui existent dans la norme SVG ainsi que leurs grandeurs caractéristiques. Nous définissons également les variables graphiques qui nous permettent d'exprimer les exigences relatives à la scène graphique des systèmes informatiques interactifs : couche d'affichage, couleur, opacité, affichage et domaine des éléments graphiques.

Nous présentons ensuite l'algorithme qui permet de vérifier formellement les exigences graphiques que nous avons définies précédemment. Cet algorithme permet d'obtenir soit une réponse booléenne à l'exigence que nous souhaitons vérifier, soit une équation dépendant des données d'entrée permettant de déterminer quels cas vérifient l'exigence graphique.

Enfin, nous présentons GPCheck : l'outil de vérification automatique des propriétés graphiques qui implémente notre algorithme.

Notons que nos propositions s'intègrent dans un processus global de vérification formelle des propriétés graphiques. Aussi, au préalable, nous commençons ce chapitre par une présentation de ce processus global.

5.1 Processus de vérification

Un des objectifs à long terme de l'équipe Informatique Interactive est de permettre la création d'interfaces humain-machine certifiées et développées avec le langage réactif Smala. C'est pour cela que nous avons présenté ce langage ainsi que quelques-uns de ses concepts clés. Afin de participer à son développement, nous avons choisi de baser notre processus de vérification des propriétés graphiques sur de l'analyse de code Smala.

Pour décrire le processus, nous utilisons une notation graphique basée sur SPEM [1]. La figure 5.1 donne la légende des éléments des diagrammes.

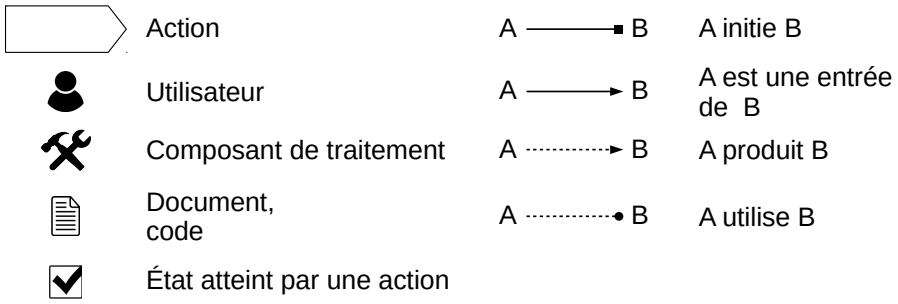


FIGURE 5.1 – Légende des diagrammes de processus

5.1.1 Processus de vérification complet

L'**analyse de code Smala** est la base du processus de vérification que nous avons développé. Nous donnons la représentation complète de notre processus de vérification dans la figure 5.2. Celui-ci sera détaillé dans la suite.

Notre processus de vérification est constitué de trois parties :

- Le **prétraitement** nous permet de formater les exigences graphiques formalisées ainsi que le code (et donc le graphe de scène enrichi) du système informatique interactif. Cela nous permet d'obtenir en sortie une classe Java que nous intégrons dans notre outil de vérification.
- L'**analyse** nous permet de parcourir le graphe de scène enrichi, contenu dans la classe Java générée pendant le processus de vérification. Cela nous permet d'obtenir en sortie du texte correspondant à une équation, résultat de la vérification des exigences graphiques.
- L'**exploitation** nous permet d'obtenir et de visualiser les valeurs des données d'entrée pour lesquelles l'exigence graphique est vraie.

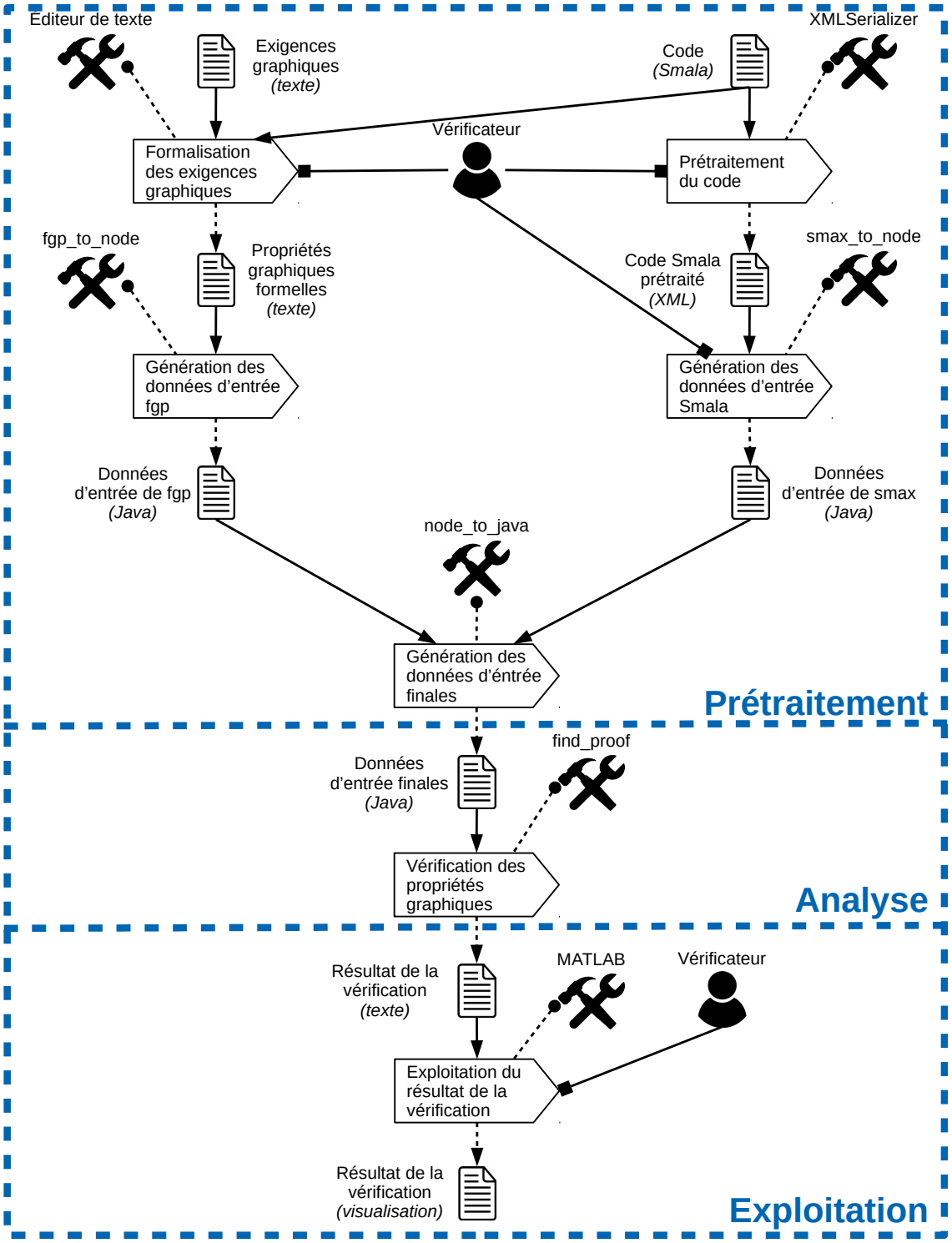


FIGURE 5.2 – Vue d'ensemble du processus de vérification des exigences graphiques

Nous présentons ci-après des informations sur les données d'entrée et de sortie du processus, utilisateurs, langages et composants de traitement qui interviennent

dans ce processus.

Les **données d'entrée** de notre processus de vérification sont :

- l'expression des propriétés graphiques à vérifier (extrait des normes ou de document de spécification),
- le code logiciel du système informatique interactif sur lequel les propriétés doivent être vérifiées.

Les **données de sortie** de notre processus sont un système d'équations portant sur les variables d'entrée du système informatique interactif pour lesquelles l'exigence graphique est vraie.

Un **utilisateur** intervient dans ce processus : le vérificateur. Celui-ci utilise trois **langages** différents :

- un langage formel (formalisme de la section 5.2.2, page 91) afin d'exprimer formellement les exigences graphiques à vérifier,
- un langage informel afin de lire les exigences graphiques à vérifier depuis les documents d'entrée (normes, spécifications, etc.),
- un langage de programmation (Smala) afin de faire le lien avec l'implémentation et donc reconnaître les identifiants des composants (éléments graphiques) dont on cherche à vérifier le respect des exigences graphiques.

Sans compter les éditeurs de texte, cinq **composants de traitement** interviennent dans ce processus :

- Le sérialiseur XML (*XMLSerializer*), un outil interne au langage Smala, traduit une application Smala en un graphe de scène enrichi (scène graphique, composants de couplage, dynamicité, données d'entrée, etc.).
- L'outil *smax_to_node* traduit le graphe de scène enrichi extrait du code Smala en une partie de code Java constituée de créations de nœuds et d'ajout de nœud à une liste donnée.
- L'outil *fgp_to_node* (fgp pour *formal graphical property*) traduit les propriétés graphiques formelles (exprimées dans un langage compréhensible par l'outil GPCheck) en une partie de code Java constituée de créations de nœuds et d'ajout de nœud à une liste donnée.
- L'outil *node_to_java* concatène les résultats produits par *smax_to_node* et *fgp_to_node*) puis met en forme le résultat.
- L'outil *find_proof* exécute notre algorithme de vérification et produit soit une solution booléenne indiquant si l'exigence graphique est vérifiée ou non, soit

une équation qui donne les contraintes sur les données d'entrée de telle sorte que l'exigence graphique est vérifiée.

- Un outil de résolution d'équations identifie les valeurs des données d'entrées pour lesquelles l'exigence graphique est vraie. Dans notre cas, nous utlisons un outil de résolution numérique (ici le simulateur MATLAB [1]).

Enfin, le processus est exécuté automatiquement par GPCheck, notre outil de vérification, pour les actions *Génération des données d'entrée fgp*, *Génération des données d'entrée Smala*, *Génération des données d'entrée finales*, *Vérification des propriétés graphiques*. Un script nous permet d'exécuter automatiquement l'action *Exploitation du résultat de la vérification* en suite des actions précédentes.

5.1.2 Présentation des actions du processus de vérification

Nous détaillons ci-après chacune des actions du processus.

Nous illustrons la sortie de chaque action par l'exemple d'une des exigences graphiques du TCAS : l'exigence E_3 du tableau 4.2 (page 72) concernant le symbole d'un trafic de type *threat aircraft* et plus particulièrement la composante couleur de cette exigence : *L'intrus à l'origine d'un RA est affiché sous la forme d'un carré plein rouge*. Nous utilisons des notations et codes que nous présentons dans les sections suivantes.

5.1.2.1 Prétraitement

L'étape de prétraitement nous permet de formaliser les exigences graphiques informelles et de formater cette formalisation en du code Java. Elle nous permet également de sérialiser au format XML le code Smala d'un système informatique interactif sous forme d'un graphe de scène enrichi et de formater ce graphe en du code Java. Enfin, cette étape nous permet de concaténer ces deux fichiers et de les formater en une classe Java que nous utilisons avec notre outil de vérification.

5.1.2.1.1 Formalisation des exigences graphiques Cette action (figure 5.3) permet d'exprimer formellement (au format texte) avec nos opérateurs graphiques (tableau 5.1, page 91) des exigences graphiques informelles (au format texte).

Le vérificateur formalise les exigences graphiques informelles rédigées par le spécificateur. Il rédige pour cela un document en utilisant un éditeur de texte. Le code

1. `https://fr.mathworks.com/products/matlab.html`

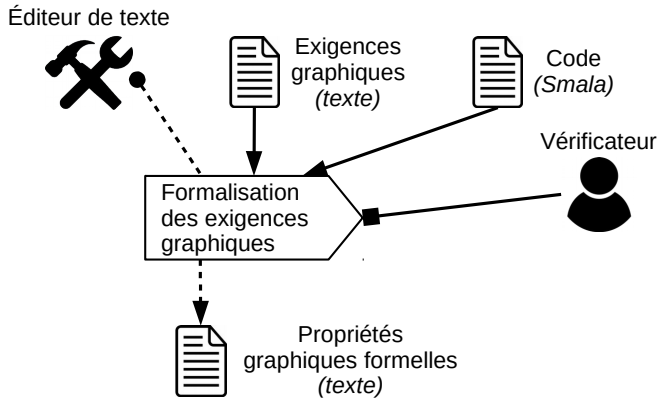


FIGURE 5.3 – Formalisation des exigences graphiques

Smala en entrée permet d'avoir l'identifiant exact des éléments graphiques afin de traduire la propriété graphique dans un langage manipulable par l'outil que nous développons.

En sortie de cette action, nous obtenons les exigences graphiques exprimées avec nos opérateurs graphiques formels (tableau 5.1, page 91).

Exemple. Nous utilisons la notation $threat\_aircraft$ (resp. red) pour le symbole d'un trafic de type *threat aircraft* (resp. la couleur rouge), nous avons la propriété graphique suivante : $threat\_aircraft =_{c,fill} red$. Le "*fill*" dans "$=_{c,fill}$" signifie que nous appliquons l'opérateur d'égalité des couleurs à la variable de couleur de remplissage, l'élément graphique n'ayant pas de contour.

5.1.2.1.2 Prétraitement du code Cette action (figure 5.4) permet de formater l'ensemble du code du système informatique interactif (tous les fichiers de code Smala et toutes les images SVG) en un seul fichier XML contenant le graphe de scène enrichi.

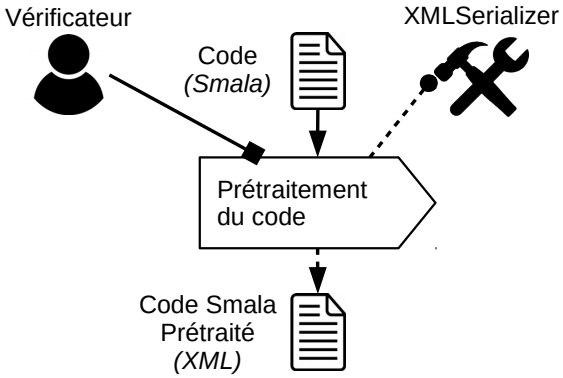


FIGURE 5.4 – Prétraitement du code

Le vérificateur prétraite également le code Smala à l'aide du sérialiseur XML de Smala. Cette action permet notamment de regrouper tous les fichiers de code ainsi que tous les fichiers SVG dans un même fichier ainsi que de les traduire dans un même format.

En sortie de cette action, nous obtenons le code Smala sous la forme d'un graphe XML, il s'agit là de ce que nous appelons le graphe de scène enrichi (scène graphique, informations de couplage entre les composants, dynamicité, etc.) Ce graphe permet une séparation claire de chaque composant ainsi que de chaque variable graphique des éléments graphiques.

Exemple. Le code 5.1 présente une partie du graphe XML résultant du fichier SVG correspondant à l'affichage du trafic après prétraitement.

```
860         <core:component id="threat_aircraft" >
861           <core:doubleproperty id="opacity" value="1" />
862           <gui:fill id="fill" value="ff0000" >
875           <gui:rectangle id="rect1935" displayed="0" >
```

Code 5.1 – Partie du graphe XML du symbole TA *threat aircraft* après prétraitement

Notons ici l'apparition d'un champ *displayed*. Celui-ci correspond à la variable d'affichage que nous définissons en section 5.2.1.5 (page 90). Ici, le champ est égal à 0. Cela signifie que le programme n'affiche pas l'élément graphique à l'initialisation.

5.1.2.1.3 Génération des données d'entrée Smala Cette action (figure 5.5) permet de formater le graphe de scène enrichi (au format XML) en une liste de nœuds qui correspondent à ceux que nous définissons en section 5.3.1 (page 92). Ce fichier de sortie est du code Java et nous permet de l'utiliser dans notre outil que nous avons développé en Java.

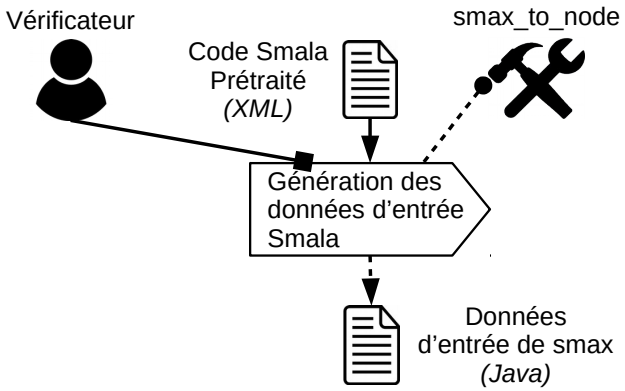


FIGURE 5.5 – Génération des données d'entrée Smala

Le vérificateur démarre la génération des données d'entrée provenant du code Smala prétraité. Pour ce faire, nous avons développé un premier outil, *smax_to_node*, qui traduit le code Smala prétraité (sous sa forme de graphe de scène enrichi XML) en une liste de nœuds $n_i \in N$. Cette action est la première de la partie automatique du processus. C'est ici que le vérificateur exécute l'outil GPCheck, dont nous présentons l'implémentation dans les parties suivantes.

En sortie de cette action, nous obtenons une partie de code Java correspondant à une liste de nœuds que l'outil *smax_to_node* a permis de traduire à partir du code Smala prétraité.

Exemple. Le code 5.2 présente une partie du code Java résultant de la traduction du code Smala prétraité concernant le symbole *threat aicraft*. Les mots-clés rouges (`Node_const`, `Node_prop` et `Node_rect`) correspondent aux types de nœuds (nœud de constante $n \in N_{const}$, nœud de propriété $n \in N_{prop}$ et nœud de rectangle $n \in N_{rect}$) que nous définissons en section 5.3.1 (page 92).

```
487 Node_const ↩
    root_traffic_sw_display_traffic_threat_aircraft_threat_aircraft_ ↩
    fill_ r = new Node_const("root.traffic.sw_display_traffic.↩
    threat_aircraft.threat_aircraft.fill.r", "255");
488 nodes.add(↩
    root_traffic_sw_display_traffic_threat_aircraft_threat_aircraft_ ↩
    fill_r);
505 Node_prop ↩
    root_traffic_sw_display_traffic_threat_aircraft_threat_aircraft_ ↩
    rect1935_displayed = new Node_prop("root.traffic.↩
    sw_display_traffic.threat_aircraft.threat_aircraft.rect1935.↩
    displayed", "false");
506 nodes.add(↩
    root_traffic_sw_display_traffic_threat_aircraft_threat_aircraft_ ↩
    rect1935_displayed);
507 Node_rect ↩
    root_traffic_sw_display_traffic_threat_aircraft_threat_aircraft_ ↩
    rect1935 = new Node_rect("root.traffic.sw_display_traffic.↩
    threat_aircraft.threat_aircraft.rect1935", ↩
    root_traffic_sw_display_traffic_threat_aircraft_threat_aircraft_ ↩
    rect1935_displayed);
508 nodes.add(↩
    root_traffic_sw_display_traffic_threat_aircraft_threat_aircraft_ ↩
    rect1935);
```

Code 5.2 – Partie de la traduction Java du code 5.1

Nous détaillons dans les parties suivantes les informations contenues dans ce code.

5.1.2.1.4 Génération des données d'entrée fgp Cette action (figure 5.6) permet de formater les exigences graphiques formalisées (au format texte) en une liste de nœuds qui correspondent à ceux que nous définissons en section 5.3.1 (page 92). Ce fichier de sortie est du code Java et ce format nous permet de l'utiliser dans notre outil que nous avons développé en Java.

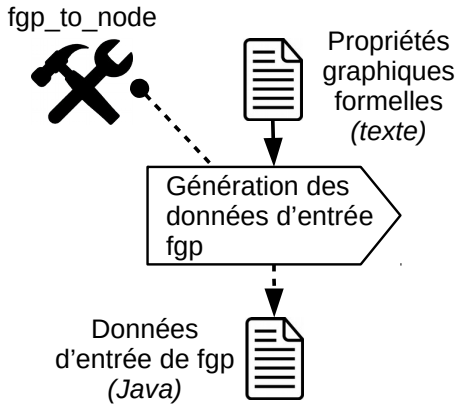


FIGURE 5.6 – Génération des données d'entrée fgp

Le processus continue avec la génération des données d'entrée provenant des propriétés graphiques. Pour ce faire, nous avons développé un deuxième outil, *fgp_to_node*, qui traduit les propriétés graphiques en une deuxième liste de nœuds $n_j \in N$.

En sortie de cette action, nous obtenons une partie de code Java correspondant à une liste de nœuds que l'outil *fgp_to_node* a permis de traduire à partir des propriétés graphiques.

```
13 Node_const RED = new Node_const("RED", "");
14 nodes.add(RED);
47 Node_and n0_and0 = new Node_and("n0_and0", n0_eq0, n0_eq1);
48 nodes.add(n0_and0);
49 Node_eq n0_eq2 = new Node_eq("n0_eq2", ↩
    root_traffic_sw_display_traffic_threat_aircraft_threat_aircraft_ ↩
    fill_b, RED_fill_b);
50 nodes.add(n0_eq2);
51 Node_and n0 = new Node_and("n0", n0_and0, n0_eq2);
52 nodes.add(n0);
```

Code 5.3 – Partie de la traduction Java de l'exigence $E_{3,couleur}$ formalisée

Exemple. Le code 5.3 présente une partie du code Java résultant de la traduction de l'exigence, formalisée, graphique en couleur que doit respecter le symbole *threat aicraft*. Les mots-clés rouges (`Node_const`, `Node_eq` et `Node_and`) correspondent aux types de nœuds (nœud de constante $n \in N_{const}$, nœud d'égalité $n \in N_{eq}$ et nœud de et logique $n \in N_{and}$) que nous définissons en section 5.3.1 (page 92).

Nous détaillons dans les parties suivantes les informations contenues dans ce code.

5.1.2.1.5 Génération des données d'entrée finales Cette action (figure 5.7) nous permet de concaténer les deux fichiers Java générés par les actions *Génération des données d'entrée fgp* et *Génération des données d'entrée Smala*. Elle nous permet également de formater ce fichier résultant afin de créer une classe Java *Java_to_check* contenant une méthode Java *java_to_check()*. Cette classe Java représente le graphe de scène enrichi.

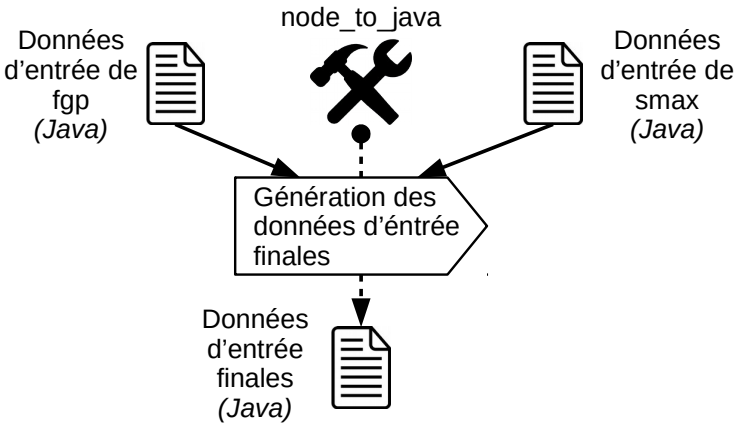


FIGURE 5.7 – Génération des données d'entrée finales

Une fois les deux actions précédentes effectuées, le processus continue avec la génération des données d'entrée finales. Pour ce faire, nous avons développé un troisième outil, *node_to_java*, qui concatène la liste des nœuds renvoyée par *smax_to_node* et la liste des nœuds renvoyée par *fgp_to_node*. Cela nous donne la version finale de la liste de nœuds, correspondant au graphe de scène enrichi de l'interface, que nous devons étudier pour vérifier l'exigence graphique. En outre, il formate le fichier pour qu'il corresponde à une classe Java.

En sortie de cette action, nous obtenons le fichier de la classe Java *Java_to_check* (code 5.4).

Nous détaillons dans les parties suivantes les informations contenues dans ce

code.

```
9  public class Java_to_check {
11     public static String java_to_check() {
12        List<Node_base> nodes = new ArrayList<Node_base>();
1857      Node_and n0 = new Node_and("n0", n0_and0, n0_eq2);
1858      nodes.add(n0);
1859      String result = Find_proof.find_proof (n0, nodes);
1860      return result;
1861   }
1862 }
```

Code 5.4 – Classe Java (partie) résultante de la génération des données d'entrée finales

5.1.2.2 Analyse

L'étape d'analyse nous permet de parcourir le graphe de scène enrichi, contenu dans la classe Java *Java_to_check* générée pendant le processus de vérification. D'un point de vue implémentation, cette étape exécute la méthode Java *java_to_check()*. Cela nous permet d'obtenir en sortie du texte correspondant à une équation, résultat de la vérification des exigences graphiques.

5.1.2.2.1 Vérification des propriétés graphiques Cette action (figure 5.8) nous permet de parcourir le graphe de scène enrichi en prenant comme point de départ l'exigence graphique à vérifier. Nous obtenons en sortie une équation au format texte représentant le résultat de la vérification des exigences graphiques, pouvant dépendre des données d'entrée du système.

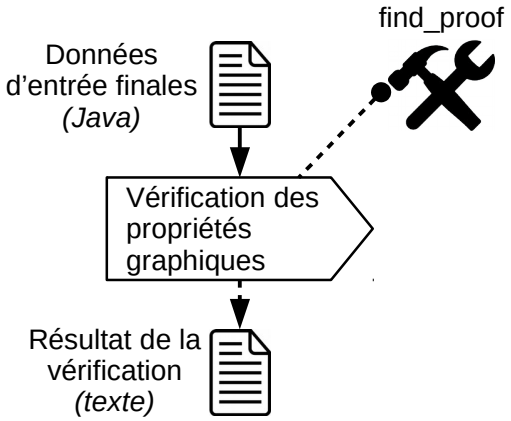


FIGURE 5.8 – Vérification des propriétés graphiques

L'algorithme que nous avons implémenté à l'intérieur de l'outil *find_proof* s'exécute sur la liste de nœuds contenant les données d'entrée finales, présente dans la classe *Java_to_check*, afin de procéder à la vérification.

En sortie de cette action, nous obtenons la solution de la vérification : soit une réponse booléenne, soit une équation permettant de déterminer dans quelles conditions d'utilisation (valeurs des données d'entrée de l'interface) l'exigence est vérifiée. Ce résultat est donné dans un langage formel. Le vérificateur doit donc renvoyer la solution booléenne ou les contraintes sur les données d'entrée dans le langage informel compris par le spécificateur et/ou le développeur.
Exemple. La réponse à l'exigence $E_{3,couleur}$ renvoyée par cette dernière action est `TRUE`. Dans ce cas, cela signifie que notre exigence *Le symbole pour un trafic de type threat aircraft est rempli de rouge* est toujours vérifiée.

5.1.2.3 Exploitation

L'étape d'exploitation nous permet d'obtenir et de visualiser les solutions au système d'équation généré au cours de l'étape d'analyse. Nous avons choisi d'utiliser un outil permettant la résolution numérique du système d'équation. D'autres possibilités existent et seront discutées par la suite.

5.1.2.3.1 Exploitation du résultat de la vérification Cette action (figure 5.9) nous permet de visualiser le résultat de la vérification de l'exigence graphique contenu dans l'équation au format texte. Nous obtenons un nuage de points qui correspond aux couples des données d'entrée passées en paramètres. Un point vert correspond à un couple qui vérifie l'exigence, un point rouge correspond à un point qui ne vérifie pas l'exigence.

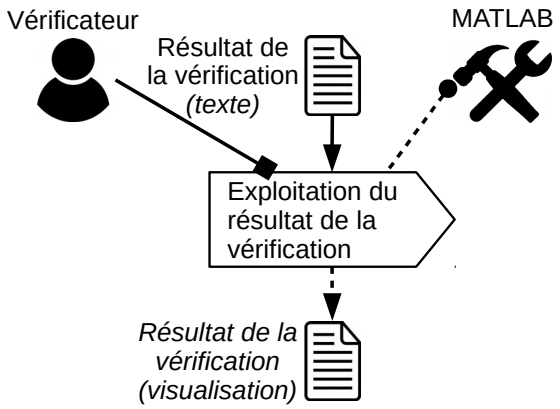


FIGURE 5.9 – Exploitation du résultat de la vérification

Enfin, le vérificateur résout l'équation obtenue avec l'outil de résolution numérique MATLAB.

En sortie de cette action, nous obtenons la visualisation de la solution de la vérification. Nous visualisons le résultat dans un graphe en deux dimensions permettant de représenter les résultats de l'équation pour les couples de données que nous désirons calculer.

Exemple. La figure 5.10 donne la visualisation du résultat de la vérification de l'exigence $E_{3,couleur}$ (un point vert correspond à l'exigence vérifiée, un point rouge correspond à l'exigence non vérifiée).

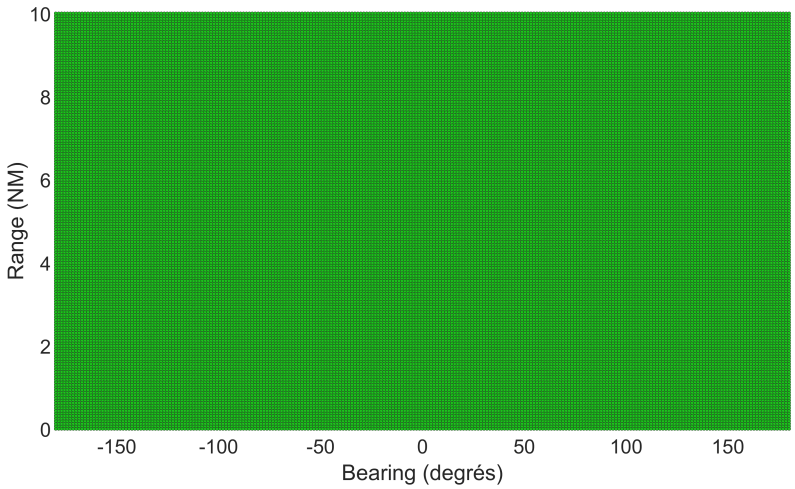


FIGURE 5.10 – Résultat de la vérification de $E_{3,couleur}$ avec $level = 4$ et $current\_range = 6$

Pour l'exemple du TCAS, les domaines des données d'entrée (tableau 4.1, page 67) principales sont les suivants :

- `current_range` : $[6; 40]$ (nombre de valeurs possibles indéterminé, supposons 34 ici)
- `bearing` : ± 180 avec un pas de 0.2 (1800 valeurs possibles)
- `range` : $[0; 128]$ avec un pas de $1/12$ (1536 valeurs possibles)
- `level` : $\{0, 1, 2, 3, 4\}$ (5 valeurs possibles)

Il faudrait donc 470016000 simulations afin de couvrir l'ensemble des scénarios pour ces quatre données d'entrée.

Afin d'optimiser le temps de calcul, nous avons réduit l'espace des variables que nous utilisons dans nos exploitations des résultats :

- `current_range` : 6 (1 valeur possible)

- `bearing` : ± 180 avec un pas de 1.44 (250 valeurs possibles)
- `range` : $[0; 10]$ avec un pas de $1/12$ (120 valeurs possibles)
- `level` : $\{0, 1, 2, 3, 4\}$ (fixé en fonction du cas, 1 valeur possible)

Avec cet espace des variables, il nous faut 30000 simulations afin de couvrir nos scénarios.

5.2 Formalisme

La définition d'un formalisme nous permet de réaliser l'action *Formalisation des exigences graphiques* (section 5.1.2.1.1, page 79).

Afin d'exprimer formellement les propriétés liées à la scène graphique, nous considérons les variables graphiques de Bertin et Barbut [26] et Wilkinson [125] : **coordonnées**, **couche**, **taille**, **couleur**, **orientation**, **forme** et **transparence**. Celles-ci sont à la base de tout élément graphique et existent dans la norme SVG. Les éléments graphiques peuvent être définis complètement avec un point spécifique (origine) et un ensemble d'attributs. Par exemple, un rectangle peut être défini en utilisant son origine (coin supérieur gauche), sa hauteur h et sa largeur w.

Les coordonnées x et y représentent des pixels et sont des nombres naturels. La couche d'affichage l est différente pour chaque élément graphique. $l = 0$ indique la couche la plus profonde et donc la première affichée : il faut donc comprendre que les éléments graphiques avec une couche $l > 0$ s'affichent au-dessus de l'élément graphique de couche $l = 0$.

5.2.1 Définitions préalables

Afin de faciliter la compréhension des opérateurs, nous proposons un ensemble de définitions des termes et symboles utilisés. La plupart des informations proviennent de la norme SVG. Cependant, certaines ont été ajoutées pour rendre compte de la dynamicité des interfaces humain-machine.

5.2.1.1 Ensembles des formes

Avant toute chose, nous définissons la forme que peut avoir un élément graphique selon la norme SVG. Les différentes formes possibles dans cette norme sont les suivantes : rectangle, cercle, segment, ellipse et texte.

- $\mathcal{R} = \{\langle x_0, y_0, h, w, l, c, o, d\rangle\}$: ensemble des rectangles,

- $\mathcal{C} = \{\langle cx, cy, r, l, c, o, d\rangle\}$: ensemble des cercles,
- $\mathcal{P} = \{\langle x_0, y_0, x_f, y_f, l, c, o, d\rangle\}$: ensemble des segments,
- $\mathcal{E} = \{\langle cx, cy, rx, ry, l, c, o, d\rangle\}$: ensemble des ellipses

Nous devons ajouter trois précisions concernant les définitions de ces ensembles de formes. Dans le cadre de cette thèse, nous considérons la forme d'un texte par le biais du rectangle dans lequel il est inscrit. Aussi, nous considérons toute forme géométrique particulière n'étant ni un rectangle, ni un cercle, ni un segment, ni une ellipse, qu'il s'agisse d'un polygone ou non, comme un ensemble de segments. Par exemple, un triangle sera l'ensemble de trois segments formant ce triangle, et un arc de cercle sera un segment avec des propriétés de rayon et centre de courbure. Enfin, actuellement, nous ne considérons pas les formes aux coins arrondis et conservons l'abstraction de leur forme globale sans arrondi.

5.2.1.2 Grandeurs caractéristiques des formes

De nombreuses grandeurs apparaissent dans les ensembles de formes que nous avons définis précédemment. Pour un élément graphique e_i, nous définissons :

- $x_0(e_i), y_0(e_i) \in \mathbb{N}$: coordonnées de l'origine de $e_i \in \mathcal{R} \cup \mathcal{P}$,
- $h(e_i), w(e_i) \in \mathbb{N}$: hauteur et largeur de $e_i \in \mathcal{R}$,
- $x_f(e_i), y_f(e_i) \in \mathbb{N}$: coordonnées de la deuxième extrémité de $e_i \in \mathcal{P}$,
- $cx(e_i), cy(e_i) \in \mathbb{N}$: coordonnées du centre de $e_i \in \mathcal{C} \cup \mathcal{E}$,
- $r(e_i) \in \mathbb{N}$: rayon de $e_i \in \mathcal{C}$,
- $rx(e_i), ry(e_i) \in \mathbb{N}$: rayon en x et y de $e_i \in \mathcal{E}$.

Concernant la notion de coordonnées de l'origine, celle-ci varie en fonction de la forme étudiée : il s'agit du coin supérieur gauche pour un rectangle et du premier point tracé pour un segment.

5.2.1.3 Couche d'affichage

La couche d'affichage est une notion à considérer dans le cadre des interfaces humain-machine informatiques. En effet, l'affichage pouvant être dynamique, cela peut induire des problématiques de superposition des éléments graphiques.

- $l(e_i) \in \mathbb{N}$: ordre ou couche d'affichage de $e_i \in \mathcal{R} \cup \mathcal{C} \cup \mathcal{P} \cup \mathcal{E}$ et $(e_1 \neq e_2) \Leftrightarrow (l(e_1) \neq l(e_2))$.

Il est important de noter que deux éléments graphiques distincts ne peuvent avoir une couche d'affichage identique.

5.2.1.4 Couleur et opacité

La couleur et l'opacité sont étroitement liées, cependant nous les définissons ici dans deux grandeurs à part afin de respecter la norme SVG et de correspondre au maximum à ce que le graphe XML extrait d'un dessin SVG donne. C'est également pour cela que nous avons choisi de définir la couleur dans le système RGB (rouge/vert/bleu) plutôt que dans le système HSV (teinte/saturation/valeur) par exemple.

- $c(e_i) = \langle c_r(e_i), c_g(e_i), c_b(e_i) \rangle, c_j(e_i) \in [0, 255]$: couleur de $e_i \in \mathcal{R} \cup \mathcal{C} \cup \mathcal{P} \cup \mathcal{E}$ en RGB,
- $o(e_i) \in [0, 100]$: cœfficient d'opacité de $e_i \in \mathcal{R} \cup \mathcal{C} \cup \mathcal{P} \cup \mathcal{E}$.

Une forme pouvant avoir une couleur de remplissage ainsi qu'une couleur de contour, il serait plus exact de raffiner cette définition afin de faire apparaître ces deux possibilités. Aussi, nous pourrons utiliser $c_{fill}(e_i)$ (resp. $c_{stroke}(e_i)$) pour exprimer la couleur de remplissage (resp. contour) de l'élément graphique e_i.

5.2.1.5 Affichage

Étant donné que nous raisonnons ici dans l'objectif de vérifier des exigences liées à la scène graphique d'interfaces humain-machine informatiques, il est important de prendre en compte la notion d'affichage. En effet, il est possible qu'un élément graphique e_i ne soit affiché qu'à certains moments particuliers de l'exécution du système du fait de la dynamicité de l'interface.

- $d(e_i) \in \{true; false\}$: affichage de $e_i \in \mathcal{R} \cup \mathcal{C} \cup \mathcal{P} \cup \mathcal{E}$.

La dynamicité d'une interface influe grandement sur la position et l'affichage des éléments graphiques, c'est pour cela qu'il est important pour nous que cette grandeur apparaisse.

5.2.1.6 Domaine

Enfin, tout élément graphique occupe un espace particulier sur l'interface graphique, indépendamment de sa couche d'affichage. Aussi, nous pensons nécessaire de définir une notion de domaine qui correspond à cela. Nous calculons ce domaine grâce aux coordonnées de l'origine et aux grandeurs caractéristiques de e_i.

- $\mathcal{D}(e_i) = \{(x, y)\} \in e_i$: domaine de e_i.

À titre d'exemple, pour un rectangle dont la largeur est parallèle à l'axe des abscisses : $\mathcal{D}(e_i) = \{(x, y) \mid x \in [x_0(e_i), x_0(e_i) + w(e_i)], y \in [y_0(e_i), y_0(e_i) + h(e_i)]\}$

Nous considérons le domaine d'un élément graphique comme l'ensemble des couples (x, y) le composant. Nous levons une exception pour les textes pour lesquels nous considérons actuellement le rectangle englobant.

5.2.2 Opérateurs graphiques

Après avoir présenté les différentes spécificités des éléments graphiques (formes et grandeurs), nous pouvons à présent définir les opérateurs graphiques de base qui nous serviront à formaliser les exigences graphiques. Le tableau 5.1 présente chacune de ces propriétés avec une définition dans notre formalisme et une illustration.

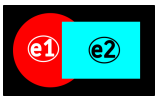
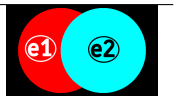
Opérateur	Formalisation	Illustration
ordre d'affichage de e_1 inférieur à celui de e_2	$e_1 <_d e_2 \Leftrightarrow l(e_1) < l(e_2)$	
intersection de e_1 et e_2	$e_1 \cap_? e_2 \Leftrightarrow \neg d(e_1) \vee \neg d(e_2) \vee (\mathcal{D}(e_1) \cap \mathcal{D}(e_2)) \neq \emptyset$	
inclusion de e_1 dans e_2	$e_1 \subset_? e_2 \Leftrightarrow \neg d(e_1) \vee \neg d(e_2) \vee \mathcal{D}(e_1) \subset \mathcal{D}(e_2)$	
égalité des formes de e_1 et e_2	$e_1 =_s e_2 \Leftrightarrow \neg d(e_1) \vee \neg d(e_2) \vee \mathcal{D}(e_1) = \mathcal{D}(e_2)$	
égalité des couleurs de e_1 et e_2	$e_1 =_c e_2 \Leftrightarrow c_i(e_1) = c_i(e_2), \forall i \in [r, g, b]$ $e_1 \neq_c e_2 \Leftrightarrow \exists i \in [r, g, b] \mid c_i(e_1) \neq c_i(e_2)$	

TABLE 5.1 – Opérateurs graphiques formels de base

Ce premier ensemble d'opérateurs graphiques de base permet de vérifier une grande partie des exigences graphiques. Nous pouvons également combiner certains de ces opérateurs pour réfléchir à des problématiques de visibilité par exemple.

Si nous réfléchissons au masquage partiel ou non d'un élément par un autre, nous pouvons combiner les propriétés d'ordre d'affichage et d'intersection, d'inclusion ou d'égalité des formes. En ce sens, nous pouvons exprimer formellement le masquage comme suit : $e_1 <_d e_2 \wedge (e_1 \cap_? e_2 \vee e_1 \subset_? e_2 \vee e_1 =_s e_2)$.

Il est également possible de trouver un masquage partiel d'un élément graphique par un autre induit par des problématiques de couleur : $e_1 =_c e_2 \wedge e_1 \cap_? e_2$. En

effet, qu'importe l'élément graphique au-dessus, si les deux ont la même couleur, ils peuvent se confondre en leur intersection.

Actuellement, nous utilisons une grande partie des variables graphiques dans ces opérateurs :

- l'opérateur d'ordre d'affichage $e_1 <_d e_2$ fait intervenir la variable de coordonnée avec la couche d'affichage,
- l'opérateur d'intersection $e_1 \cap_? e_2$ fait intervenir les coordonnées en x et y, la taille, la forme et l'orientation,
- l'opérateur d'inclusion $e_1 \subset_? e_2$ fait intervenir les coordonnées en x et y, la taille et la forme et l'orientation,
- l'opérateur d'égalité des formes $e_1 =_s e_2$ fait intervenir les coordonnées en x et y, la taille et la forme et l'orientation,
- l'opérateur d'égalité des couleur $e_1 =_c e_2$ fait intervenir la couleur.

5.3 Algorithme de vérification

La section précédente a introduit le formalisme que nous utiliserons pour exprimer formellement les exigences graphiques des systèmes informatiques interactifs. Cependant, cette formalisation n'est que la première étape d'un processus global de vérification des propriétés graphiques.

Après cette étape de formalisation, nous devons déterminer comment nous allons vérifier les propriétés. Notre objectif est d'effectuer de l'analyse de code Smala par le biais du graphe de scène enrichi que nous obtenons à partir du code Smala. Pour cela, nous avons développé un algorithme qui traite les différents opérateurs graphiques que nous avons définis par rapport aux types de nœuds qui apparaissent dans le graphe de scène enrichi.

5.3.1 Définitions préalables

Avant de présenter l'algorithme, nous donnons les différents termes et symboles que nous utilisons. Nous présentons ici les différents types de nœuds qui existent dans le graphe de scène enrichi que nous extrayons du code Smala de notre application ainsi que des propriétés graphiques que nous définissons pour l'application.

Nous nommons N un ensemble de nœuds que nous définirons ci-dessous. *id* représente l'identifiant unique de l'élément. *val* représente la valeur associée au nœud.

layer représente la couche d'affichage de l'élément graphique. l (*left*), r (*right*), *src* (*source*), *dst* (*destination*), *input*, *base*, *exp* (*exponent*), x_0, x_f, y_0, y_f, w (*width*), h (*height*), cx, cy, r (*radius*), tx, ty, a (*angle*) et *disp* (*displayed*) représentent des nœuds sans spécifier leur type.

5.3.1.1 Données du graphe de scène enrichi

Dans la section 4.1, nous avons présenté avec la figure 4.2 le graphe de scène enrichi que nous utilisons pour la vérification des propriétés graphiques. Nous avons choisi de définir le lien parent-enfant à travers l'identifiant unique de chaque composant. Ainsi, l'identifiant d'un composant est de la forme :

```
grandParent.parent.enfant
```

Nous avons traduit les liens de référence en incluant les nœuds référencés dans la structure des nœuds concernés. Nous présentons cela ci-après.

Aussi, la liste contenant les nœuds nous permet d'obtenir la relation d'ordre du graphe de scène enrichi.

5.3.1.2 Variables du programme

Nous cherchons à couvrir la totalité des exécutions possibles de l'application interactive. Ainsi, il nous faut considérer toutes les variables qui créent un scénario d'exécution. Nous considérons ici trois cas de variables du programme :

- $N_{const} = \{\langle id, val\rangle\}$: ensemble des variables **constantes**,
- $N_{entry} = \{\langle id, val\rangle\}$: ensemble des variables correspondant aux **entrées** (par exemple les interactions entre l'environnement et l'application)
- $N_{prop} = \{\langle id, val\rangle\}$: ensemble des autres variables, considérées comme des **propriétés**.

5.3.1.3 Éléments graphiques

Nous nous intéressons à l'étude des propriétés graphiques. De ce fait, nous devons considérer les éléments graphiques et plus particulièrement leur forme. Nous considérons ici les formes suivantes :

- $N_{rect} = \{\langle id, x_0, y_0, w, h, layer, disp\rangle\}$: ensemble des **rectangles**,
- $N_{circle} = \{\langle id, cx, cy, r, layer, disp\rangle\}$: ensemble des **cercles**,

- $N_{path} = \{\langle id, x_0, x_f, y_0, y_f \rangle\}$: ensemble des **segments**,
- $N_{path_{set}} = \{\langle id, \{\langle id, x_0, x_f, y_0, y_f \rangle\}, layer, disp \rangle\}$: ensemble des **ensembles de segments**.

5.3.1.4 Opérations géométriques

Les éléments graphiques peuvent évoluer dans l'espace en deux dimensions de l'interface par le biais de modifications de leurs coordonnées ou par le biais de transformations géométriques. Nous considérons ici les transformations géométriques suivantes :

- $N_{trans} = \{\langle id, tx, ty \}$: ensemble des **translations**,
- $N_{rot} = \{\langle id, a, cx, cy \rangle\}$: ensemble des **rotations**.

5.3.1.5 Opérateurs du formalisme

Afin de vérifier des propriétés par le biais de notre formalisme, nous définissons les nœuds suivants :

- $N_{low_d} = \{\langle id, l, r \rangle\}$: ensemble des opérateurs de **comparaison ascendante** de l'ordre d'affichage des éléments graphiques,
- $N_{inter} = \{\langle id, l, r \rangle\}$: ensemble des opérateurs d'**intersection** d'éléments graphiques.
- $N_{in} = \{\langle id, l, r \rangle\}$: ensemble des opérateurs d'**inclusion** d'éléments graphiques,

5.3.1.6 Liens entre les composants

Nous avons vu dans la section 4.1 qu'il existe, en Smala, des composants non graphiques qui propagent valeurs et activations entre d'autres composants du programme. Ces composants de lien existent dans d'autres langages tels que le Python ou le Java FX. Ils sont le moteur de la dynamicité de l'application interactive. Nous déclarons ici trois types de nœuds qui correspondent à ces composants de liens :

- $N_{bind} = \{\langle id, src, dst \rangle\}$: ensemble des **bindings**, couplant la source du binding (src) à sa destination (dst),
- $N_{assign} = \{\langle id, src, dst \rangle\}$: ensemble des **assignments**, assignant la valeur de la source de l'assignment (src) à la valeur de sa destination (dst),
- $N_{connect} = \{\langle id, src, dst \rangle\}$: ensemble des **connectors**, assignant la valeur de la source du connector (src) à la valeur de destination (dst) et couplant src à dst.

5.3.1.7 Opérateurs binaires et unaires

Il est possible de définir des opérations mathématiques ou logiques dans le programme afin de passer d'une branche d'exécution du programme à une autre ou de traduire les données d'entrée en données d'affichage. Pour cela, nous pouvons utiliser des opérateurs binaires et unaires.

5.3.1.7.1 Opérateurs binaires Nous déclarons ici les types de nœuds qui correspondent aux différents opérateurs binaires que nous utilisons actuellement.

Dans un premier temps, nous déclarons les **opérateurs logiques** :

- $N_{eq} = \{\langle id, l, r\rangle\}$: ensemble des comparateurs d'**égalité**, comparant l'égalité de l'opérande de gauche (l) à celle de l'opérande de droite (r),
- $N_{neq} = \{\langle id, l, r\rangle\}$: ensemble des comparateurs de **non-égalité**, comparant la non-égalité de l'opérande de gauche (l) à celle de l'opérande de droite (r),
- $N_{and} = \{\langle id, l, r\rangle\}$: ensemble des opérateurs **et**, réalisant un et logique entre l'opérande de gauche (l) et l'opérande de droite (r),
- $N_{or} = \{\langle id, l, r\rangle\}$: ensemble des opérateurs **ou**, réalisant un ou logique entre l'opérande de gauche (l) et l'opérande de droite (r),
- $N_{cmp} = \{\langle id, l, r\rangle\}$: ensemble des comparateurs d'**ascendance**, comparant l'ascendance de l'opérande de gauche (l) à celle de l'opérande de droite (r),
- $N_{cmp_s} = \{\langle id, l, r\rangle\}$: ensemble des comparateurs d'**ascendance stricte**, comparant l'ascendance stricte de l'opérande de gauche (l) à celle de l'opérande de droite (r).

Dans un second temps, nous déclarons les **opérateurs mathématiques** :

- $N_{add} = \{\langle id, l, r\rangle\}$: ensemble des opérateurs d'**addition**, additionnant l'opérande de gauche (l) à l'opérande de droite (r),
- $N_{sub} = \{\langle id, l, r\rangle\}$: ensemble des opérateurs de **soustraction**, soustrayant l'opérande de droite (r) à l'opérande de gauche (l),
- $N_{mult} = \{\langle id, l, r\rangle\}$: ensemble des opérateurs de **multiplication**, multipliant l'opérande de gauche (l) et l'opérande de droite (r),
- $N_{div} = \{\langle id, l, r\rangle\}$: ensemble des opérateurs de **division**, divisant l'opérande de gauche (l) par l'opérande de droite (r),
- $N_{pow} = \{\langle id, base, exp\rangle\}$: ensemble des opérateurs de **puissance**, élevant l'opérande de base ($base$) à l'exposant (exp).

5.3.1.7.2 Opérateurs unaires Enfin, nous déclarons ici les types de nœuds qui correspondent aux différents opérateurs unaires que nous utilisons actuellement :

- $N_{cos} = \{\langle id, input\rangle\}$: ensemble des opérateurs de **cosinus**, calculant le cosinus de l'opérande *input*,
- $N_{sin} = \{\langle id, input\rangle\}$: ensemble des opérateurs de **sinus**, calculant le sinus de l'opérande *input*,
- $N_{sqrt} = \{\langle id, input\rangle\}$: ensemble des opérateurs de **racine carrée**, calculant la racine carrée de l'opérande *input*,
- $N_{abs} = \{\langle id, input\rangle\}$: ensemble des opérateurs de **valeur absolue**, calculant la valeur absolue de l'opérande *input*.

5.3.1.8 Ensemble final des nœuds

Nous avons défini les différents ensembles qui correspondent aux différents types de nœuds que nous pouvons obtenir dans le graphe de scène enrichi de notre application. Ainsi, l'ensemble N des nœuds est : $N = N_{const} \cup N_{entry} \cup N_{prop} \cup N_{rect} \cup N_{circle} \cup N_{path} \cup N_{path_{set}} \cup N_{trans} \cup N_{rot} \cup N_{low_d} \cup N_{inter} \cup N_{in} \cup N_{bind} \cup N_{assign} \cup N_{connect} \cup N_{eq} \cup N_{and} \cup N_{or} \cup N_{cmp} \cup N_{cmp_s} \cup N_{add} \cup N_{sub} \cup N_{mult} \cup N_{div} \cup N_{pow} \cup N_{cos} \cup N_{sin} \cup N_{sqrt} \cup N_{abs}$

5.3.2 Algorithme

Nous avons développé un algorithme pour vérifier les exigences graphiques, exprimées dans notre formalisme, pour les interfaces graphiques implémentées en Smala et dont nous avons extrait le graphe de scène enrichi. Il consiste en une fonction récursive $\mathcal{F}$ prenant en paramètre un nœud n_c du graphe final (graphe de scène enrichi et graphe des propriétés graphiques formelles) correspondant à l'exigence graphique que le vérificateur vise à vérifier.

Nous notons $\mathcal{F}^{type(n_c)}(n_c, val)$ le cas de la fonction récursive où le nœud n_c est de type *type* à laquelle nous ajoutons un paramètre *val* de type *String*.

$$\mathcal{F} : N \times String \rightarrow String$$

Par souci de concision, lorsque la chaîne est vide (aucune valeur passée en paramètre), nous notons $\mathcal{F}^{type(n_c)}(n_c)$ au lieu de $\mathcal{F}^{type(n_c)}(n_c, "")$ et pouvons considérer la définition suivante pour ce cas particulier :

$$\mathcal{F} : N \rightarrow String$$

Comme notre objectif est d'obtenir une solution booléenne ou une équation, $\mathcal{F}$ renvoie une chaîne de caractères. Ainsi, il faut garder à l'esprit que les opérateurs (binaires et unaires) n'affectent pas $\mathcal{F}$ mais sont renvoyés sous forme de caractères.

Afin d'illustrer cela, prenons par exemple les nœuds suivants :

$$x = \langle x, "" \rangle \in N_{entry}$$
$$int3 = \langle int3, "3" \rangle \in N_{const}$$
$$adder = \langle adder, x, int3 \rangle \in N_{add}$$

Ces nœuds correspondent à l'instruction `x + 3`. Appliquons maintenant l'algorithme sur le nœud *adder* :

$$\mathcal{F}(adder) := \mathcal{F}^{add}(adder)$$
$$\mathcal{F}(adder) := \mathcal{F}(adder.l) \text{ '+' } \mathcal{F}(adder.r)$$
$$\mathcal{F}(adder) := \mathcal{F}(x) \text{ '+' } \mathcal{F}(int3)$$
$$\mathcal{F}(adder) := \mathcal{F}^{entry}(x) \text{ '+' } \mathcal{F}^{const}(int3)$$
$$\mathcal{F}(adder) := x \text{ '+' } 3$$

Le résultat de $\mathcal{F}(adder)$ est la chaîne de caractères `"x + 3"`. Dans tout le déroulement de l'algorithme que nous avons présenté ci-dessus, le symbole `+` ne correspond alors pas à un opérateur d'addition ou de concaténation mais simplement au caractère `+`. Notons aussi l'appel de $\mathcal{F}(adder.l)$ qui permet d'appeler la fonction $\mathcal{F}$ avec l'enfant l du nœud *adder*. Cela se rapporte à la définition de la relation parent-enfant que nous avons définie en section 5.3.1.1 (page 93).

5.3.2.1 Algorithme complet

Nous présentons ici l'ensemble des cas (de récursion et non terminaux) de la fonction récursive que nous avons définie. Nous détaillons chacun de ces cas dans les sections suivantes.

- si $n_c \in N_{const}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{const}(n_c) := n_c.val$
- si $n_c \in N_{entry}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{entry}(n_c) := n_c.id$
- si $n_c \in N_{prop}$,
  - $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{prop}(n_c) := \mathcal{F}(n_l), \forall n_l(n, n_c) \in N_{assign} \cup N_{connect}$

  - $\mathcal{F}(n_c, val) \Leftrightarrow \mathcal{F}^{prop}(n_c, val) := \mathcal{F}(n_l, val)$,
    $\forall n_l(n, n_c) \in N_{assign} \cup N_{connect} \mid n_c.src \in N_{const} \wedge n_c.src.val = val$

- si $n_c \in N_{low_d}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{low_d}(n_c) := \mathcal{F}(n_c.l.layer)$ '<' $\mathcal{F}(n_c.r.layer)$

- si $n_c \in N_{inter}, \mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{inter}(n_c) :=$ "¬ (" $\mathcal{F}(n_c.l.disp)$ ") ∨ ¬ (" $\mathcal{F}(n_c.r.disp)$ ") ∨"
$$\begin{cases} f^{RR}_{\cap?}(n_c.l, n_c.r) \mid n_c.l, n_c.r \in N_{rect} \\ f^{CC}_{\cap?}(n_c.l, n_c.r) \mid n_c.l, n_c.r \in N_{circle} \\ f^{RC}_{\cap?}(n_c.l, n_c.r) \mid n_c.l \in N_{rect} \wedge n_c.r \in N_{circle} \end{cases}$$

- si $n_c \in N_{in}, \mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{in}(n_c) :=$ "¬ (" $\mathcal{F}(n_c.l.disp)$ ") ∨ ¬ (" $\mathcal{F}(n_c.r.disp)$ ") ∨"
$$\begin{cases} f^{RR}_{\subset?}(n_c.l, n_c.r) \mid n_c.l, n_c.r \in N_{rect} \\ f^{CC}_{\subset?}(n_c.l, n_c.r) \mid n_c.l, n_c.r \in N_{circle} \\ f^{RC}_{\subset?}(n_c.l, n_c.r) \mid n_c.l \in N_{rect} \wedge n_c.r \in N_{circle} \\ f^{CR}_{\subset?}(n_c.l, n_c.r) \mid n_c.l \in N_{circle} \wedge n_c.r \in N_{rect} \\ f^{PR}_{\subset?}(n_c.l, n_c.r) \mid n_c.l \in N_{path} \wedge n_c.r \in N_{rect} \\ f^{PC}_{\subset?}(n_c.l, n_c.r) \mid n_c.l \in N_{path} \wedge n_c.r \in N_{circle} \end{cases}$$

- si $n_c \in N_{bind}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{bind}(n_c) := \mathcal{F}(n_c.src)$

- si $n_c \in N_{assign}$,

  - $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{assign}(n_c) := \mathcal{F}(n_b), \forall n_b(n, n_c) \in N_{bind}$

  - $\mathcal{F}(n_c, val) \Leftrightarrow \mathcal{F}^{assign}(n_c, val) := \mathcal{F}(n_b), \forall n_b(n, n_c) \in N_{bind}$

- si $n_c \in N_{connect}$,

  - $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{connect}(n_c) := \mathcal{F}(n_c.src)$ '∧' $\mathcal{F}(n_b), \forall n_b(n, n_c) \in N_{bind}$

  - $\mathcal{F}(n_c, val) \Leftrightarrow \mathcal{F}^{connect}(n_c, val) := \mathcal{F}(n_b), \forall n_b(n, n_c) \in N_{bind}$

- si $n_c \in N_{eq}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{eq}(n_c) :=$
$$\begin{cases} \mathcal{F}(n_c.l, n_c.r.val) \mid n_c.l \in N_{prop} \wedge n_c.r \in N_{const} \\ \mathcal{F}(n_c.r, n_c.l.val) \mid n_c.l \in N_{const} \wedge n_c.r \in N_{prop} \\ \text{"¬ (" } \mathcal{F}(n_c.l, n_c.r.val) \text{ ')' } \mid n_c.l \in N_{prop} \wedge n_c.r \in N_{const} \wedge n_c.r.val = false \\ \text{"¬ (" } \mathcal{F}(n_c.r, n_c.l.val) \text{ ')' } \mid n_c.l \in N_{const} \wedge n_c.r \in N_{prop} \wedge n_c.l.val = false \\ \mathcal{F}(n_l, \emptyset), \forall n_l(n, n_c.l) \in N_{assign} \cup N_{connect} \mid \exists n_{l2}(n, n_c.r) \wedge n_c.l, n_c.r \in N_{prop} \\ \mathcal{F}(n_c.l) \text{ "==" } \mathcal{F}(n_c.r) \mid n_c.l, n_c.r \in N_{const} \cup N_{entry} \cup N_{prop} \end{cases}$$

- si $n_c \in N_{neq}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{neq}(n_c) :=$

$$\begin{cases} \text{"}\neg\ (\text{"}\ \mathcal{F}(n_c.l, n_c.r.val)\ \text{')'} \mid n_c.l \in N_{prop} \wedge n_c.r \in N_{const} \\ \text{"}\neg\ (\text{"}\ \mathcal{F}(n_c.r, n_c.l.val)\ \text{')'} \mid n_c.l \in N_{const} \wedge n_c.r \in N_{prop} \\ \mathcal{F}(n_c.l, n_c.r.val) \mid n_c.l \in N_{prop} \wedge n_c.r \in N_{const} \wedge n_c.r.val = false \\ \mathcal{F}(n_c.r, n_c.l.val) \mid n_c.l \in N_{const} \wedge n_c.r \in N_{prop} \wedge n_c.l.val = false \\ \text{"}\neg\ (\text{"}\ \mathcal{F}(n_l, \emptyset)\ \text{')'}\ , \forall n_l(n, n_c.l) \in N_{assign} \cup N_{connect} \mid \exists n_{l2}(n, n_c.r) \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad n_c.l, n_c.r \in N_{prop} \\ \mathcal{F}(n_c.l)\ \text{'}\neq\text{'}\ \mathcal{F}(n_c.r) \mid n_c.l, n_c.r \in N_{const} \cap N_{entry} \cap N_{prop} \end{cases}$$

- si $n_c \in N_{and}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{and}(n_c) := \mathcal{F}(n_c.l)\ \text{'}\wedge\text{'}\ \mathcal{F}(n_c.r)$
- si $n_c \in N_{or}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{or}(n_c) := \mathcal{F}(n_c.l)\ \text{'}\vee\text{'}\ \mathcal{F}(n_c.r)$
- si $n_c \in N_{cmp}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{cmp}(n_c) := \mathcal{F}(n_c.l)\ \text{'}\leq\text{'}\ \mathcal{F}(n_c.r)$
- si $n_c \in N_{cmp_s}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{cmp_s}(n_c) := \mathcal{F}(n_c.l)\ \text{'}<\text{'}\ \mathcal{F}(n_c.r)$
- si $n_c \in N_{add}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{add}(n_c) := \mathcal{F}(n_c.l)\ \text{'}+\text{'}\ \mathcal{F}(n_c.r)$
- si $n_c \in N_{sub}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{sub}(n_c) := \mathcal{F}(n_c.l)\ \text{"}-\ (\text{"}\ \mathcal{F}(n_c.r)\ \text{')'}$
- si $n_c \in N_{mult}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{mult}(n_c) :=\ \text{'('}\ \mathcal{F}(n_c.l)\ \text{"}) * (\text{"}\ \mathcal{F}(n_c.r)\ \text{')'}$
- si $n_c \in N_{div}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{div}(n_c) :=\ \text{'('}\ \mathcal{F}(n_c.l)\ \text{"}) / (\text{"}\ \mathcal{F}(n_c.r)\ \text{')'}$
- si $n_c \in N_{pow}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{pow}(n_c) :=\ \text{'('}\ \mathcal{F}(n_c.base)\ \text{"}) \hat{}\ (\text{"}\ \mathcal{F}(n_c.exp)\ \text{')'}$
- si $n_c \in N_{cos}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{cos}(n_c) :=\ \text{"}cos(\text{"}\ \mathcal{F}(n_c.input)\ \text{')'}$
- si $n_c \in N_{sin}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{sin}(n_c) :=\ \text{"}sin(\text{"}\ \mathcal{F}(n_c.input)\ \text{')'}$
- si $n_c \in N_{sqrt}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{sqrt}(n_c) :=\ \text{"}sqrt(\text{"}\ \mathcal{F}(n_c.input)\ \text{')'}$
- si $n_c \in N_{abs}$, $\mathcal{F}(n_c) \Leftrightarrow \mathcal{F}^{abs}(n_c) :=\ \text{"}abs(\text{"}\ \mathcal{F}(n_c.input)\ \text{')'}$

5.3.2.2 Cas de base de la fonction

La fonction que nous développons pour notre algorithme est récursive, de ce fait, il nous faut définir les conditions d'arrêt de la récursion. Nous avons défini deux cas de base permettant d'arrêter la récursion.

5.3.2.2.1 Variable constante ($n_c \in N_{const}$) Nous notons $\mathcal{F}^{const}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{const}$.

$$\mathcal{F}^{const}(n_c) := n_c.val$$

Le premier des deux cas de base pour $\mathcal{F}$ se produit si n_c est un nœud correspondant à une variable constante. Dans ce premier cas de base, $\mathcal{F}$ renvoie la valeur *val* associée au nœud n_c.

5.3.2.2.2 Variable de donnée d'entrée ($n_c \in N_{entry}$) Nous notons $\mathcal{F}^{entry}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{entry}$.

$$\mathcal{F}^{entry}(n_c) := n_c.id$$

Le second des deux cas de base pour $\mathcal{F}$ se produit si n_c est un nœud correspondant à une donnée d'entrée. Dans ce second cas de base, $\mathcal{F}$ renvoie l'identifiant id du nœud n_c. Ce second cas permet notamment d'obtenir les contraintes sur les données d'entrée concernées.

5.3.2.3 Cas non terminaux

Tous les autres cas que nous présentons ci-après sont des cas non terminaux. Pour certains types de nœuds, nous avons défini la fonction $\mathcal{F}$ avec un second paramètre correspondant à une valeur val.

5.3.2.3.1 Variable de type propriété sans valeur ($n_c \in N_{prop}$) Nous notons $\mathcal{F}^{prop}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{prop}$.

En ce qui concerne les nœuds représentant des variables du programme qui ne correspondent ni à des constantes ni à des données d'entrée, nous avons deux cas de figure dépendants du fait qu'une valeur val est passée en paramètre de la fonction $\mathcal{F}$.

Nous nous intéressons ici à la mise à jour de la valeur de ces propriétés. Celle-ci existe pour tout assignment ou connector dont la variable est destination. De ce fait, la récursion appelle ces assignments ou connectors.

$$\mathcal{F}^{prop}(n_c) := \mathcal{F}(n_l), \forall n_l(n, n_c) \in N_{assign} \cup N_{connect}$$

Si n_c est un nœud correspondant à une variable de type propriété, l'algorithme appelle la récursion avec un nœud $n_l \in N_{assign} \cup N_{connect}$ comme paramètre de $\mathcal{F}$ pour chaque $n_l \in N_{assign} \cup N_{connect}$ qui a n_c comme nœud dst. S'il y a plusieurs n_l correspondants, l'algorithme retourne le caractère "$\vee$" entre chaque appel de $\mathcal{F}$.

5.3.2.3.2 Variable de type propriété avec valeur ($n_c \in N_{prop}$) Nous notons $\mathcal{F}^{prop}(n_c, val)$ la fonction $\mathcal{F}(n_c, val)$ avec $n_c \in N_{prop}$.

$$\mathcal{F}^{prop}(n_c, val) := \mathcal{F}(n_l, val),$$
$$\forall n_l(n, n_c) \in N_{assign} \cup N_{connect} \mid n_c.src \in N_{const} \wedge n_c.src.val = val$$

Si nous ajoutons une valeur val comme deuxième paramètre de $\mathcal{F}$, l'algorithme appelle la récursion avec :

- un nœud $n_l \in N_{assign} \cup N_{connect}$ comme paramètre de $\mathcal{F}$ pour chaque $n_l \in N_{assign} \cup N_{connect}$ qui a le nœud $src \in N_{const}$ de n_c comme nœud dst et si la valeur de src est égale à val,
- la valeur val.

S'il y a plusieurs n_l correspondants, l'algorithme retourne le caractère "$\vee$" entre chaque appel de $\mathcal{F}$.

5.3.2.3.3 Comparaison ascendante d'ordre d'affichage ($n_c \in N_{low_d}$) Nous notons $\mathcal{F}^{low_d}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{low_d}$.

Le comparateur ascendant d'ordre d'affichage des éléments graphiques est un comparateur ascendant entre leurs positions dans le graphe de scène enrichi.

$$\mathcal{F}^{low_d}(n_c) := \mathcal{F}(n_c.l.layer) \text{ '<' } \mathcal{F}(n_c.r.layer)$$

L'algorithme retourne donc la chaîne de caractères '<' entre deux appels récursifs de $\mathcal{F}$.

5.3.2.3.4 Intersection et inclusion des éléments graphiques Afin de définir l'intersection et l'inclusion des éléments graphiques dans notre algorithme, nous avons défini différentes fonctions représentant les intersections et inclusions entre les types de formes par rapport à leurs grandeurs caractéristiques. Nous présentons ces fonctions en considérant les éléments graphiques suivants :

- le rectangle $r_1 = \langle r_1, x_{10}, y_{10}, w_1, h_1, layer_{r1}, disp_{r1} \rangle$,
- le rectangle $r_2 = \langle r_2, x_{20}, y_{20}, w_2, h_2, layer_{r2}, disp_{r2} \rangle$,
- le cercle $c_1 = \langle c_1, cx_1, cy_1, r_{c1}, layer_{c1}, disp_{c1} \rangle$,
- le cercle $c_2 = \langle c_2, cx_2, cy_2, r_{c2}, layer_{c2}, disp_{c2} \rangle$,
- le segment $p = \langle p, x_{p0}, x_{pf}, y_{p0}, y_{pf} \rangle$,
- l'ensemble de segments $ps = \langle ps, \{\langle p, x_{p0}, x_{pf}, y_{p0}, y_{pf} \rangle\}, layer_p, disp_p \rangle$.

Nous présentons également les définitions des translations et des rotations :

- la translation $tr = \langle tr, tx, ty \rangle$ avec tx la translation en x et ty celle en y,
- la rotation $rot = \langle rot, rotx, roty, rota \rangle$ avec $rotx$ le centre de rotation en x, $roty$ celui en y et $rota$ l'angle de la rotation.

Pour chaque élément graphique, nous effectuons un calcul pour déterminer les coordonnées en fonction des potentielles transformations géométriques impactant cet élément. Ainsi, nous avons les impacts de la translation sur les éléments graphiques :

- $r_1 : x_{10} = x_{10} + tx$ et $y_{10} = y_{10} + ty$,
- $r_2 : x_{20} = x_{20} + tx$ et $y_{20} = y_{20} + ty$,
- $c_1 : cx_1 = cx_1 + tx$ et $cy_1 = cy_1 + ty$,
- $c_2 : cx_2 = cx_2 + tx$ et $cy_2 = cy_2 + ty$,
- $p : x_{p0} = x_{p0} + tx$, $x_{pf} = x_{pf} + tx$, $y_{p0} = y_{p0} + ty$ et $y_{pf} = y_{pf} + ty$

Nous avons également les impacts de la rotation sur les éléments graphiques (nous utilisons la notation x' pour désigner la nouvelle valeur de x) :

- $r_1 : x'_{10} = \sqrt{(rotx - x_{10})^2 + (roty - y_{10})^2} * cos(rota)$
  et $y'_{10} = \sqrt{(rotx - x_{10})^2 + (roty - y_{10})^2} * sin(rota)$,
- $r_2 : x'_{20} = \sqrt{(rotx - x_{20})^2 + (roty - y_{20})^2} * cos(rota)$
  et $y'_{20} = \sqrt{(rotx - x_{20})^2 + (roty - y_{20})^2} * sin(rota)$,
- $c_1 : cx'_1 = \sqrt{(rotx - cx_1)^2 + (roty - cy_1)^2} * cos(rota)$
  et $cy'_1 = \sqrt{(rotx - cx_1)^2 + (roty - cy_1)^2} * sin(rota)$,
- $c_2 : cx'_2 = \sqrt{(rotx - cx_2)^2 + (roty - cy_2)^2} * cos(rota)$
  et $cy'_2 = \sqrt{(rotx - cx_2)^2 + (roty - cy_2)^2} * sin(rota)$,
- $p : x'_{p0} = \sqrt{(rotx - x_{p0})^2 + (roty - y_{p0})^2} * cos(rota)$,
  $y'_{p0} = \sqrt{(rotx - x_{p0})^2 + (roty - y_{p0})^2} * sin(rota)$,
  $x'_{pf} = \sqrt{(rotx - x_{pf})^2 + (roty - y_{pf})^2} * cos(rota)$,
  et $y'_{pf} = \sqrt{(rotx - x_{pf})^2 + (roty - y_{pf})^2} * sin(rota)$.

Les transformations géométriques sont cumulatives. Aussi, nous devons effectuer ces calculs pour toute transformation géométrique impactant l'élément graphique concerné dans l'ordre d'impact de ces transformations. Nous considérons le contenu de ce paragraphe après calcul des opérations de transformations géométriques.

Afin de simplifier l'écriture des expressions mathématiques, nous écrivons :

- $x_{1f} = x_{10} + w_1$ et $\mathcal{F}(x_{1f}) := \mathcal{F}(x_{10})$ '+' $\mathcal{F}(w_1)$,
- $x_{2f} = x_{20} + w_2$ et $\mathcal{F}(x_{2f}) := \mathcal{F}(x_{20})$ '+' $\mathcal{F}(w_2)$,
- $y_{1f} = y_{10} + h_1$ et $\mathcal{F}(y_{1f}) := \mathcal{F}(y_{10})$ '+' $\mathcal{F}(h_1)$,
- $y_{2f} = y_{20} + h_2$ et $\mathcal{F}(y_{2f}) := \mathcal{F}(y_{20})$ '+' $\mathcal{F}(h_2)$.

Le symbole "+" qui apparaît entre deux appels de la fonction $\mathcal{F}$ ne correspond pas à l'opérateur d'addition ou de concaténation mais simplement au caractère "+".

Nous précisons également que pour simplifier davantage les expressions, nous notons ici x_{10} au lieu de $r_1.x_{10}$ et de même pour l'ensemble des variables et formes. Nous faisons abstraction de l'identifiant du composant bien qu'il doive figurer, c'est pourquoi nous avons pris le soin de nommer différemment chaque variable interne des nœuds considérés pour les formes.

5.3.2.3.4.1 Intersection rectangle - rectangle Nous définissons dans l'algorithme la fonction d'intersection rectangle - rectangle comme suit :

$$f^{RR}_{\cap_?}(r_1, r_2) = \mathcal{F}(x_{20}) \text{ '}\leq\text{' } \mathcal{F}(x_{1f}) \text{ '}\wedge\text{' } \mathcal{F}(x_{10}) \text{ '}\leq\text{' } \mathcal{F}(x_{2f}) \\ \text{'}\wedge\text{' } \mathcal{F}(y_{20}) \text{ '}\leq\text{' } \mathcal{F}(y_{1f}) \text{ '}\wedge\text{' } \mathcal{F}(y_{10}) \text{ '}\leq\text{' } \mathcal{F}(y_{2f})$$

5.3.2.3.4.2 Intersection cercle - cercle Nous définissons dans l'algorithme la fonction d'intersection cercle - cercle comme suit :

$$f^{CC}_{\cap_?}(c_1, c_2) = \mathcal{F}(r_{c1}) \text{ '+' } \mathcal{F}(r_{c2}) \text{ "}\leq sqrt((\text{" } \mathcal{F}(cx_2) \text{ "}- (\text{" } \mathcal{F}(cx_1) \text{ "})) \hat{} \ 2 + (\text{" } \\ \mathcal{F}(cy_2) \text{ "}- (\text{" } \mathcal{F}(cy_1) \text{ "})) \hat{} \ 2)\text{"}$$

5.3.2.3.4.3 Intersection rectangle - cercle Nous définissons dans l'algorithme la fonction d'intersection rectangle - cercle comme suit :

$$
\begin{aligned}
f^{RC}_{\cap_?}(r_1, c_1) = \ & "sqrt((" \ \mathcal{F}(cx_1) \ "- (" \ \mathcal{F}(x_{10}) \ ")) \hat{} \ 2 + (" \ \mathcal{F}(cy_1) \\
& "- (" \ \mathcal{F}(y_{10}) \ ")) \hat{} \ 2 \leq" \ \mathcal{F}(r_{c1}) \\
& "\vee \ sqrt((" \ \mathcal{F}(cx_1) \ "- (" \ \mathcal{F}(x_{1f}) \ ")) \hat{} \ 2 + (" \ \mathcal{F}(cy_1) \\
& "- (" \ \mathcal{F}(y_{10}) \ ")) \hat{} \ 2 \leq" \ \mathcal{F}(r_{c1}) \\
& "\vee \ sqrt((" \ \mathcal{F}(cx_1) \ "- (" \ \mathcal{F}(x_{10}) \ ")) \hat{} \ 2 + (" \ \mathcal{F}(cy_1) \\
& "- (" \ \mathcal{F}(y_{1f}) \ ")) \hat{} \ 2 \leq" \ \mathcal{F}(r_{c1}) \\
& "\vee \ sqrt((" \ \mathcal{F}(cx_1) \ "- (" \ \mathcal{F}(x_{1f}) \ ")) \hat{} \ 2 + (" \ \mathcal{F}(cy_1) \\
& "- (" \ \mathcal{F}(y_{1f}) \ ")) \hat{} \ 2 \leq" \ \mathcal{F}(r_{c1}) \\
& '\vee' \ \mathcal{F}(x_{10}) \ '\leq' \ \mathcal{F}(cx_1) \ '\wedge' \ \mathcal{F}(cx_1) \ '\leq' \ \mathcal{F}(x_{1f}) \\
& '\wedge' \ \mathcal{F}(y_{10}) \ '\leq' \ \mathcal{F}(cy_1) \ '+' \ \mathcal{F}(r_{c1}) \\
& '\wedge' \ \mathcal{F}(cy_1) \ "- (" \ \mathcal{F}(r_{c1}) \ ") \leq" \ \mathcal{F}(y_{1f}) \ '\vee' \ \mathcal{F}(y_{10}) \ '\leq' \ \mathcal{F}(cy_1) \\
& '\wedge' \ \mathcal{F}(cy_1) \ '\leq' \ \mathcal{F}(y_{1f}) \\
& '\wedge' \ \mathcal{F}(x_{10}) \ '\leq' \ \mathcal{F}(cx_1) \ '+' \ \mathcal{F}(r_{c1}) \ '\wedge' \ \mathcal{F}(cx_1) \ "- (" \ \mathcal{F}(r_{c1}) \ ") \leq" \ \mathcal{F}(x_{1f}) \\
& '\vee' \ \mathcal{F}(x_{10}) \ '\leq' \ \mathcal{F}(cx_1) \ "- (" \ \mathcal{F}(r_{c1}) \ ") \wedge" \ \mathcal{F}(cx_1) \ '+' \ \mathcal{F}(r_{c1}) \ '\leq' \ \mathcal{F}(x_{1f}) \\
& '\wedge' \ \mathcal{F}(y_{10}) \ '\leq' \ \mathcal{F}(cy_1) \ "- (" \ \mathcal{F}(r_{c1}) \ ") \wedge" \ \mathcal{F}(cy_1) \ '+' \ \mathcal{F}(r_{c1}) \ '\leq' \ \mathcal{F}(y_{1f})
\end{aligned}
$$

5.3.2.3.4.4 Intersection ($n_c \in N_{inter}$) Nous notons $\mathcal{F}^{inter}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{inter}$.

Finalement, nous obtenons pour le cas de l'intersection pour la fonction récursive :

$$
\begin{aligned}
\mathcal{F}^{inter}(n_c) := \ & "\neg \ (" \ \mathcal{F}(n_c.l.disp) \ ") \vee \neg \ (" \ \mathcal{F}(n_c.r.disp) \ ") \vee" \\
& \begin{cases}
f^{RR}_{\cap_?}(n_c.l, n_c.r) \mid n_c.l, n_c.r \in N_{rect} \\
f^{CC}_{\cap_?}(n_c.l, n_c.r) \mid n_c.l, n_c.r \in N_{circle} \\
f^{RC}_{\cap_?}(n_c.l, n_c.r) \mid n_c.l \in N_{rect} \wedge n_c.r \in N_{circle}
\end{cases}
\end{aligned}
$$

5.3.2.3.4.5 Inclusion rectangle dans rectangle Nous définissons dans l'algorithme la fonction d'inclusion rectangle dans rectangle comme suit :

$$f^{RR}_{\subset_?}(r_1, r_2) = \mathcal{F}(x_{20})\ \text{'<'}\ \mathcal{F}(x_{1f})\ \text{'}\wedge\text{'}\ \mathcal{F}(x_{10})\ \text{'<'}\ \mathcal{F}(x_{2f})$$
$$\text{'}\wedge\text{'}\ \mathcal{F}(y_{20})\ \text{'<'}\ \mathcal{F}(y_{1f})\ \text{'}\wedge\text{'}\ \mathcal{F}(y_{10})\ \text{'<'}\ \mathcal{F}(y_{2f})$$

5.3.2.3.4.6 Inclusion cercle dans cercle Nous définissons dans l'algorithme la fonction d'inclusion cercle dans cercle comme suit :

$$f^{CC}_{\subset_?}(c_1, c_2) = \mathcal{F}(r_{c1})\ \text{"+ } sqrt(\text{(" } \mathcal{F}(cx_2)\ \text{"- (" } \mathcal{F}(cx_1)\ \text{")) ^ 2) + ("}$$
$$\mathcal{F}(cy_2)\ \text{"- (" } \mathcal{F}(cy_1)\ \text{")) ^ 2) <" } \mathcal{F}(r_{c2})$$

5.3.2.3.4.7 Inclusion rectangle dans cercle Nous définissons dans l'algorithme la fonction d'inclusion rectangle dans cercle comme suit :

$$\begin{aligned} f^{RC}_{\subset_?}(r_1, c_2) = \ & \text{"}sqrt(\text{(" } \mathcal{F}(cx_2)\ \text{"- (" } \mathcal{F}(x_{10})\ \text{")) ^ 2 + ("} \\ & \mathcal{F}(cy_2)\ \text{"- (" } \mathcal{F}(y_{10})\ \text{")) ^ 2) <" } \mathcal{F}(r_{c2}) \\ & \text{"}\wedge\ sqrt(\text{(" } \mathcal{F}(cx_2)\ \text{"- (" } \mathcal{F}(x_{1f})\ \text{")) ^ 2 + ("} \\ & \mathcal{F}(cy_2)\ \text{"- (" } \mathcal{F}(y_{10})\ \text{")) ^ 2) <" } \mathcal{F}(r_{c2}) \\ & \text{"}\wedge\ sqrt(\text{(" } \mathcal{F}(cx_2)\ \text{"- (" } \mathcal{F}(x_{10})\ \text{")) ^ 2 + ("} \\ & \mathcal{F}(cy_2)\ \text{"- (" } \mathcal{F}(y_{1f})\ \text{")) ^ 2) <" } \mathcal{F}(r_{c2}) \\ & \text{"}\wedge\ sqrt(\text{(" } \mathcal{F}(cx_2)\ \text{"- (" } \mathcal{F}(x_{1f})\ \text{")) ^ 2 + ("} \\ & \mathcal{F}(cy_2)\ \text{"- (" } \mathcal{F}(y_{1f})\ \text{")) ^ 2) <" } \mathcal{F}(r_{c2}) \end{aligned}$$

5.3.2.3.4.8 Inclusion cercle dans rectangle Nous définissons dans l'algorithme la fonction d'inclusion cercle dans rectangle comme suit :

$$f^{CR}_{\subset_?}(c_1, r_2) = \mathcal{F}(x_{20})\ \text{'<'}\ \mathcal{F}(cx_1)\ \text{"- (" } \mathcal{F}(r_{c1})\ \text{") }\wedge\text{" } \mathcal{F}(cx_1)\ \text{'+'}\ \mathcal{F}(r_{c1})\ \text{'<'}\ \mathcal{F}(x_{2f})$$
$$\text{'}\wedge\text{'}\ \mathcal{F}(y_{20})\ \text{'<'}\ \mathcal{F}(cy_1)\ \text{"- (" } \mathcal{F}(r_{c1})\ \text{") }\wedge\text{" } \mathcal{F}(cy_1)\ \text{'+'}\ \mathcal{F}(r_{c1})\ \text{'<'}\ \mathcal{F}(y_{2f})$$

5.3.2.3.4.9 Inclusion segment dans rectangle Nous définissons dans l'algorithme la fonction d'inclusion segment dans rectangle comme suit :

$$f_{\subset?}^{PR}(p, r_2) = \mathcal{F}(x_{20}) \text{ '<' } \mathcal{F}(x_{p0}) \text{ '}\wedge\text{' } \mathcal{F}(x_{pf}) \text{ '<' } \mathcal{F}(x_{2f}) \\ \text{ '}\wedge\text{' } \mathcal{F}(y_{20}) \text{ '<' } \mathcal{F}(y_{p0}) \text{ '}\wedge\text{' } \mathcal{F}(y_{pf}) \text{ '<' } \mathcal{F}(y_{2f})$$

5.3.2.3.4.10 Inclusion segment dans cercle Nous définissons dans l'algorithme la fonction d'inclusion segment dans cercle comme suit :

$$\begin{aligned} f_{\subset?}^{PC}(p, c_2) = \ & \text{"}sqrt((\text{" } \mathcal{F}(cx_2) \text{ "}- (\text{" } \mathcal{F}(x_{p0}) \text{ "})) \hat{} \ 2 + (\text{"} \\ & \mathcal{F}(cy_2) \text{ "}- (\text{" } \mathcal{F}(y_{p0}) \text{ "})) \hat{} \ 2) <\text{" } \mathcal{F}(r_{c2}) \\ & \text{"}\wedge\ sqrt((\text{" } \mathcal{F}(cx_2) \text{ "}- (\text{" } \mathcal{F}(x_{pf}) \text{ "})) \hat{} \ 2 + (\text{"} \\ & \mathcal{F}(cy_2) \text{ "}- (\text{" } \mathcal{F}(y_{p0}) \text{ "})) \hat{} \ 2) <\text{" } \mathcal{F}(r_{c2}) \\ & \text{"}\wedge\ sqrt((\text{" } \mathcal{F}(cx_2) \text{ "}- (\text{" } \mathcal{F}(x_{p0}) \text{ "})) \hat{} \ 2 + (\text{"} \\ & \mathcal{F}(cy_2) \text{ "}- (\text{" } \mathcal{F}(y_{pf}) \text{ "})) \hat{} \ 2) <\text{" } \mathcal{F}(r_{c2}) \\ & \text{"}\wedge\ sqrt((\text{" } \mathcal{F}(cx_2) \text{ "}- (\text{" } \mathcal{F}(x_{pf}) \text{ "})) \hat{} \ 2 + (\text{"} \\ & \mathcal{F}(cy_2) \text{ "}- (\text{" } \mathcal{F}(y_{pf}) \text{ "})) \hat{} \ 2) <\text{" } \mathcal{F}(r_{c2}) \end{aligned}$$

5.3.2.3.4.11 Inclusion ($n_c \in N_{in}$) Nous notons $\mathcal{F}^{in}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{in}$.

Finalement, nous obtenons pour le cas de l'inclusion pour la fonction récursive :

$$\mathcal{F}^{in}(n_c) := \text{"}\neg\ (\text{" } \mathcal{F}(n_c.l.disp) \text{ "}) \vee \neg\ (\text{" } \mathcal{F}(n_c.r.disp) \text{ "}) \vee\text{"} \\ \begin{cases} f_{\subset?}^{RR}(n_c.l, n_c.r) \mid n_c.l, n_c.r \in N_{rect} \\ f_{\subset?}^{CC}(n_c.l, n_c.r) \mid n_c.l, n_c.r \in N_{circle} \\ f_{\subset?}^{RC}(n_c.l, n_c.r) \mid n_c.l \in N_{rect} \wedge n_c.r \in N_{circle} \\ f_{\subset?}^{CR}(n_c.l, n_c.r) \mid n_c.l \in N_{circle} \wedge n_c.r \in N_{rect} \\ f_{\subset?}^{PR}(n_c.l, n_c.r) \mid n_c.l \in N_{path} \wedge n_c.r \in N_{rect} \\ f_{\subset?}^{PC}(n_c.l, n_c.r) \mid n_c.l \in N_{path} \wedge n_c.r \in N_{circle} \end{cases}$$

5.3.2.3.5 Composant binding ($n_c \in N_{bind}$) Nous notons $\mathcal{F}^{bind}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{bind}$.

Un binding permet de transmettre l'activation d'un composant. De ce fait, il faut rechercher la source de ce binding.

$$\mathcal{F}^{bind}(n_c) := \mathcal{F}(n_c.src)$$

Si n_c est de type binding, l'algorithme appelle la récursion avec le nœud src de n_c comme paramètre de $\mathcal{F}$.

5.3.2.3.6 Composant assignment sans valeur ($n_c \in N_{assign}$) Nous notons $\mathcal{F}^{assign}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{assign}$.

Un assignment permet de transmettre la valeur d'une destination à une source. De ce fait, il faut rechercher tout binding activant cet assignment pour suivre le chemin d'activation.

$$\mathcal{F}^{assign}(n_c) := \mathcal{F}(n_b), \forall n_b(n, n_c) \in N_{bind}$$

Si n_c est de type assignment, l'algorithme appelle la récursion avec un nœud $n_b \in N_{bind}$ comme paramètre de $\mathcal{F}$ pour chaque $n_b \in N_{bind}$ qui a n_c comme nœud dst. S'il y a plusieurs n_b correspondants, l'algorithme retourne le caractère "$\vee$" entre chaque appel de $\mathcal{F}$.

5.3.2.3.7 Composant assignment avec valeur ($n_c \in N_{assign}$) Nous notons $\mathcal{F}^{assign}(n_c, val)$ la fonction $\mathcal{F}(n_c, val)$ avec $n_c \in N_{assign}$.

$$\mathcal{F}^{assign}(n_c, val) := \mathcal{F}(n_b), \forall n_b(n, n_c) \in N_{bind}$$

Si nous ajoutons une valeur val comme deuxième paramètre de $\mathcal{F}$, la définition précédente reste la même.

5.3.2.3.8 Composant connector sans valeur ($n_c \in N_{connect}$) Nous notons $\mathcal{F}^{connect}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{connect}$.

Un connector permet de transmettre la valeur et l'activation d'une destination à une source. De ce fait, il faut donc rappeler la fonction récursive avec la source du connector pour suivre ce chemin d'activation. De plus, il faut également rechercher tout binding activant ce connector pour suivre ces potentiels chemins d'activations.

$$\mathcal{F}^{connect}(n_c) := \mathcal{F}(n_c.src) \text{ '}\wedge\text{' } \mathcal{F}(n_b), \forall n_b(n, n_c) \in N_{bind}$$

Si n_c est de type connector, l'algorithme appelle la récursion avec le nœud src de n_c comme paramètre de $\mathcal{F}$. Il écrit un caractère "$\wedge$" et l'algorithme appelle la récursion

avec un nœud $n_b \in N_{bind}$ comme paramètre de $\mathcal{F}$ pour chaque $n_b \in N_{bind}$ qui a n_c comme nœud *dst*. S'il y a plusieurs n_b correspondants, l'algorithme retourne le caractère "$\vee$" entre chaque appel de $\mathcal{F}$.

5.3.2.3.9 Composant connector avec valeur ($n_c \in N_{connect}$) Nous notons $\mathcal{F}^{connect}(n_c, val)$ la fonction $\mathcal{F}(n_c, val)$ avec $n_c \in N_{connect}$.

$$\mathcal{F}^{connect}(n_c, val) := \mathcal{F}(n_b), \forall n_b(n, n_c) \in N_{bind}$$

Si nous ajoutons une valeur *val* comme deuxième paramètre de $\mathcal{F}$, nous appliquons la même définition que $\mathcal{F}^{assign}(n_c)$.

5.3.2.3.10 Opérateur d'égalité ($n_c \in N_{eq}$) Nous notons $\mathcal{F}^{eq}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{eq}$.

L'opérateur d'égalité est un cas particulier dépendant du type de nœuds de ses opérandes *droite* et *gauche* : constante, donnée d'entrée ou propriété.

5.3.2.3.10.1 Avec une constante et une propriété

$$\mathcal{F}^{eq}(n_c) := \mathcal{F}(n_c.l, n_c.r.val) \mid n_c.l \in N_{prop} \wedge n_c.r \in N_{const}$$

$$\mathcal{F}^{eq}(n_c) := \mathcal{F}(n_c.r, n_c.l.val) \mid n_c.l \in N_{const} \wedge n_c.r \in N_{prop}$$

Si n_c est de type égalité et si un nœud (l ou r) de n_c est une constante et l'autre une propriété, l'algorithme appelle la récursion avec le nœud qui est une propriété comme premier paramètre de $\mathcal{F}$ et la valeur *val* de la constante comme second paramètre de $\mathcal{F}$.

5.3.2.3.10.2 Avec une constante de valeur "false" et une propriété

$$\mathcal{F}^{eq}(n_c) := \text{"}\neg\text{ (" } \mathcal{F}(n_c.l, n_c.r.val) \text{ ')'} \mid n_c.l \in N_{prop} \wedge n_c.r \in N_{const} \wedge n_c.r.val = false$$

$$\mathcal{F}^{eq}(n_c) := \text{"}\neg\text{ (" } \mathcal{F}(n_c.r, n_c.l.val) \text{ ')'} \mid n_c.l \in N_{const} \wedge n_c.r \in N_{prop} \wedge n_c.l.val = false$$

Si la valeur de la constante est le booléen $false$, alors nous ajoutons le caractère "$\neg$" avant $\mathcal{F}$.

5.3.2.3.10.3 Avec deux propriétés

$$\mathcal{F}^{eq}(n_c) := \mathcal{F}(n_l, \emptyset), \forall n_l(n, n_c.l) \in N_{assign} \cup N_{connect} \mid \exists n_{l2}(n, n_c.r) \wedge n_c.l, n_c.r \in N_{prop}$$

Si n_c est de type égalité et si les deux nœuds (l et r) de n_c sont des propriétés, l'algorithme appelle la récursion avec un nœud $n_l(n, n_c.l) \in N_{assign} \cup N_{connect}$ qui a la même source n qu'un nœud $n_{l2}(n, n_c.r) \in N_{assign} \cup N_{connect}$ comme premier paramètre de $\mathcal{F}$ et avec une valeur vide comme second paramètre de $\mathcal{F}$.

Ce cas permet de parcourir le chemin d'activation et de mise à jour des propriétés qui ont la même source d'activation ou de mise à jour.

5.3.2.3.10.4 Pour les autres cas

$$\mathcal{F}^{eq}(n_c) := \mathcal{F}(n_c.l) \text{ "==" } \mathcal{F}(n_c.r) \mid n_c.l, n_c.r \in N_{const} \cup N_{entry} \cup N_{prop}$$

Si n_c est de type égalité et si les deux nœuds (l et r) de n_c sont des constantes, l'algorithme appelle la récursion avec le nœud l de n_c comme paramètre de $\mathcal{F}$, écrit les caractères "==" et appelle la récursion avec le nœud r de n_c comme paramètre de $\mathcal{F}$.

5.3.2.3.11 Opérateur de non-égalité ($n_c \in N_{neq}$) Nous notons $\mathcal{F}^{neq}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{neq}$.

L'opérateur de non-égalité est un cas particulier dépendant du type de nœuds de ses opérandes *droite* et *gauche* : constante, donnée d'entrée ou propriété. Son traitement est proche de celui de l'opérateur d'égalité.

5.3.2.3.11.1 Avec une constante et une propriété

$$\mathcal{F}^{neq}(n_c) := \text{ "¬ (" } \mathcal{F}(n_c.l, n_c.r.val) \text{ ')' } \mid n_c.l \in N_{prop} \wedge n_c.r \in N_{const}$$

$$\mathcal{F}^{neq}(n_c) := \text{ "¬ (" } \mathcal{F}(n_c.r, n_c.l.val) \text{ ')' } \mid n_c.l \in N_{const} \wedge n_c.r \in N_{prop}$$

Si n_c est de type non-égalité et si un nœud (l ou r) de n_c est une constante et l'autre une propriété, l'algorithme appelle la récursion avec le nœud qui est une propriété comme premier paramètre de $\mathcal{F}$ et la valeur val de la constante comme second paramètre de $\mathcal{F}$. Nous ajoutons le caractère "¬" avant $\mathcal{F}$

5.3.2.3.11.2 Avec une constante de valeur "false" et une propriété

$$\mathcal{F}^{neq}(n_c) := \mathcal{F}(n_c.l, n_c.r.val) \mid n_c.l \in N_{prop} \wedge n_c.r \in N_{const} \wedge n_c.r.val = false$$

$$\mathcal{F}^{neq}(n_c) := \mathcal{F}(n_c.r, n_c.r.val) \mid n_c.l \in N_{const} \wedge n_c.r \in N_{prop} \wedge n_c.l.val = false$$

Si la valeur de la constante est le booléen $false$, alors nous retirons le caractère "¬" avant $\mathcal{F}$.

5.3.2.3.11.3 Avec deux propriétés

$$\mathcal{F}^{neq}(n_c) := \text{ "¬ (" } \mathcal{F}(n_l, \emptyset) \text{ ')', } \forall n_l(n, n_c.l) \in N_{assign} \cup N_{connect} \mid \exists n_{l2}(n, n_c.r) \wedge n_c.l, n_c.r \in N_{prop}$$

Si n_c est de type non-égalité et si les deux nœuds (l et r) de n_c sont des propriétés, l'algorithme appelle la récursion avec un nœud $n_l(n, n_c.l) \in N_{assign} \cup N_{connect}$ qui a la même source n qu'un nœud $n_{l2}(n, n_c.r) \in N_{assign} \cup N_{connect}$ comme premier paramètre de $\mathcal{F}$ et avec une valeur vide comme second paramètre de $\mathcal{F}$. Nous ajoutons le caractère "¬" avant $\mathcal{F}$

Ce cas permet de parcourir le chemin d'activation et de mise à jour des propriétés qui ont la même source d'activation ou de mise à jour.

5.3.2.3.11.4 Pour les autres cas

$$\mathcal{F}^{neq}(n_c) := \mathcal{F}(n_c.l) \text{ '}\neq\text{' } \mathcal{F}(n_c.r) \mid n_c.l, n_c.r \in N_{const} \cap N_{entry} \cap N_{prop}$$

Si n_c est de type non-égalité et si les deux nœuds (l et r) de n_c sont des constantes, l'algorithme appelle la récursion avec le nœud l de n_c comme paramètre de $\mathcal{F}$, écrit le caractère "≠" (ou les caractères " !=") et appelle la récursion avec le nœud r de n_c comme paramètre de $\mathcal{F}$.

5.3.2.3.12 Opérateurs binaires logiques et mathématiques Ici nous considérons les nœuds $n_c \in N_{and} \cup N_{or} \cup N_{add} \cup N_{sub} \cup N_{mult} \cup N_{div} \cup N_{pow}$. Si n_c est un nœud correspondant à un opérateur binaire (autre que l'opérateur d'égalité présenté ci-avant), l'algorithme retourne la chaîne de caractères correspondant à l'opérateur entre deux appels récursifs de $\mathcal{F}$. Dans certains cas, nous ajoutons des parenthèses afin de prendre en compte les problématiques d'opérations logiques ou mathéma-

tiques successives dans le chemin d'activation.

Nous notons $\mathcal{F}^{and}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{and}$. Cela va de même pour les autres opérateurs binaires.

$$\mathcal{F}^{and}(n_c) := \mathcal{F}(n_c.l)\ \text{'}\wedge\text{'}\ \mathcal{F}(n_c.r)$$

Par exemple, si n_c est de type opérateur et logique, l'algorithme appelle la récursion avec le nœud l de n_c comme paramètre de $\mathcal{F}$, écrit le caractère "$\wedge$" et appelle la récursion avec le nœud r de n_c comme paramètre de $\mathcal{F}$.

De manière analogue, nous obtenons :

$$\mathcal{F}^{or}(n_c) := \mathcal{F}(n_c.l)\ \text{'}\vee\text{'}\ \mathcal{F}(n_c.r)$$

$$\mathcal{F}^{cmp}(n_c) := \mathcal{F}(n_c.l)\ \text{'}\leq\text{'}\ \mathcal{F}(n_c.r)$$

$$\mathcal{F}^{cmp_s}(n_c) := \mathcal{F}(n_c.l)\ \text{'}<\text{'}\ \mathcal{F}(n_c.r)$$

$$\mathcal{F}^{add}(n_c) := \mathcal{F}(n_c.l)\ \text{'}+\text{'}\ \mathcal{F}(n_c.r)$$

$$\mathcal{F}^{sub}(n_c) := \mathcal{F}(n_c.l)\ \text{"}-\ (\text{"}\ \mathcal{F}(n_c.r)\ \text{')'}$$

$$\mathcal{F}^{mult}(n_c) :=\ \text{'('}\ \mathcal{F}(n_c.l)\ \text{")}*\ (\text{"}\ \mathcal{F}(n_c.r)\ \text{')'}$$

$$\mathcal{F}^{div}(n_c) :=\ \text{'('}\ \mathcal{F}(n_c.l)\ \text{")}\ /\ (\text{"}\ \mathcal{F}(n_c.r)\ \text{')'}$$

$$\mathcal{F}^{pow}(n_c) :=\ \text{'('}\ \mathcal{F}(n_c.base)\ \text{")}\ \hat{}\ (\text{"}\ \mathcal{F}(n_c.exp)\ \text{')'}$$

5.3.2.3.13 Opérateurs unaires Ici nous considérons les nœuds $n_c \in N_{cos} \cup N_{sin} \cup N_{sqrt} \cup N_{abs}$. Si n_c est un nœud correspondant à un opérateur unaire, l'algorithme retourne la chaîne de caractères correspondant à l'opérateur autour de l'appel récursif de $\mathcal{F}$. Dans certains cas, nous ajoutons des parenthèses afin de prendre en compte les problématiques d'opérations logiques ou mathématiques successives dans le chemin d'activation.

Nous notons $\mathcal{F}^{cos}(n_c)$ la fonction $\mathcal{F}(n_c)$ avec $n_c \in N_{cos}$. Cela va de même pour les autres opérateurs unaires.

$$\mathcal{F}^{cos}(n_c) :=\ \text{"}cos(\text{"}\ \mathcal{F}(n_c.input)\ \text{')'}$$

Par exemple, si n_c est de type cosinus, l'algorithme appelle la récursion avec le nœud *input* de n_c comme paramètre de $\mathcal{F}$ et en écrivant autour les chaînes de caractères

correspondantes "cos("$\mathcal{F}$")".

De manière analogue, nous obtenons :

$$\mathcal{F}^{sin}(n_c) := \text{"}sin(\text{"}\ \mathcal{F}(n_c.input)\ \text{')'}$$

$$\mathcal{F}^{sqrt}(n_c) := \text{"}sqrt(\text{"}\ \mathcal{F}(n_c.input)\ \text{')'}$$

$$\mathcal{F}^{abs}(n_c) := \text{"}abs(\text{"}\ \mathcal{F}(n_c.input)\ \text{')'}$$

5.4 GPCheck : Implémentation de l'algorithme

Notre objectif est maintenant d'implémenter notre algorithme mais également la partie automatique du processus de vérification des propriétés graphiques. Pour ce faire, nous avons développé GPCheck, un prototype de vérificateur automatique de propriétés graphiques. Nous avons choisi de développer ce premier prototype avec le langage Java pour ses propriétés d'héritage et de cast dynamique.

Nous consacrons cette section aux détails d'implémentation du prototype.

5.4.1 Définition de la structure des nœuds du graphe de scène enrichi

```
3  public class Node_base {
4      private String id;
6      Node_base (String id) {
7          this.id = id;
8      }
10     public String getId() {
11         return id;
12     }
13 }
```

Code 5.5 – Implémentation Java de la structure de base des nœuds ($n_c \in N$)

Les nœuds du graphe de scène enrichi sont au cœur de notre algorithme, nous avons donc défini une structure de base permettant de représenter un nœud quel que soit son type. Cela permet à l'outil qui implémente notre algorithme de pouvoir prendre en paramètre n'importe quel nœud que nous lui passerons. Nous avons défini une classe Java `Node_base` (code 5.5) pour représenter cette base.

L'ensemble des nœuds que nous définissons par la suite héritent de cette classe Java Node_base qui ne contient qu'un attribut, l'identifiant, unique à chaque nœud.

5.4.1.1 Exemples de nœuds

Nous présentons ici trois types de nœuds que nous retrouvons dans l'implémentation du prototype afin de présenter certains détails de cette implémentation.

5.4.1.1.1 Nœud de binding Un binding est un composant Smala ayant une source et une destination. Nous l'avons implémenté en Java selon le code 5.6.

```
3  public class Node_bind extends Node_base {
4      private Node_base src;
5      private Node_base dst;
7      public Node_bind(String id, Node_base src, Node_base dst) {
8          super(id);
9          this.src = src;
10         this.dst = dst;
11     }
13     public Node_base getSrc() {
14         return src;
15     }
17     public Node_base getDst() {
18         return dst;
19     }
20 }
```

Code 5.6 – Implémentation Java du nœud de binding ($n_c \in N_{bind}$)

Dans le cas de la définition de sa structure Java qui hérite de celle de `Node_base`, nous ne nous intéressons pas au type de la source et de la destination du binding. C'est pourquoi nous déclarons ici *src* et *dst* en tant que `Node_base`.

5.4.1.1.2 Nœud de donnée d'entrée Une donnée d'entrée est une structure contenant un identifiant et une valeur. Nous l'avons implémenté en Java selon le code 5.7.

De manière générale, la valeur d'un nœud peut prendre n'importe quel type (chaîne de caractères, entier, flottant, etc.), c'est pour cela que nous la déclarons de type `String`.

```
3  public class Node_entry extends Node_base {
4     private String val;
6     public Node_entry(String id, String val) {
7        super(id);
8        this.val = val;
9     }
11    public String getVal() {
12       return val;
13    }
14 }
```

Code 5.7 – Implémentation Java du nœud de donnée d'entrée ($n_c \in N_{entry}$)

5.4.1.1.3 Nœud de rectangle Dans cette implémentation, nous représentons un rectangle par une structure contenant un identifiant et la variable d'affichage. Nous l'avons implémenté en Java selon le code 5.8.

```
3  public class Node_rect extends Node_base {
4     private Node_base disp;
6     public Node_rect(String id, Node_base disp) {
7        super(id);
8        this.disp = disp;
9     }
11    public Node_base getDisp() {
12       return disp;
13    }
14 }
```

Code 5.8 – Implémentation Java du nœud de rectangle ($n_c \in N_{rect}$)

Actuellement, nous n'avons pas déclaré les grandeurs caractéristiques du rectangle comme étant une partie de la structure. Nous les retrouvons par les liens parent-enfant.

5.4.2 smax_to_node : Génération des données d'entrée de Smala

`smax_to_node` est le premier outil de la partie automatique du processus de vérification. Celle-ci traduit le code Smala prétraité (dans sa version XML) en une liste de nœuds Java.

```
5  public class Node_def {
7      public Node_def(Node _node, String _path_id, String _path) {
8          setNode(_node);
9          setPath_id(_path_id);
10         setPath(_path);
11     }
17     public void setNode(Node node) {
18         this.node = node;
19     }
37     private Node node;
38     private String path;
39     private String path_id;
40 }
```

Code 5.9 – Implémentation Java (partie) de la structure de prétraitement des nœuds

Dans un premier temps, nous utilisons XPath[2] afin de parcourir le graphe XML et récupérer les nœuds et leurs informations. Pour cela, nous avons défini une première structure, la classe `Node_def` (code 5.9), contenant un nœud de type `org.w3c.dom.Node`, un chemin `path` et un chemin d'identifiant `path_id` de type `String`.

Nous rangeons tous les nœuds du graphe de scène enrichi, prétraités et typés `Node_def`, dans une liste `all_nodes_w_id`.

L'outil parcourt ensuite cette liste et écrit dans le fichier de sortie la liste des nœuds provenant du graphe de scène enrichi en fonction de plusieurs paramètres comme le nom du nœud ou ses attributs. Nous présentons cela dans le code 5.10 qui représente une partie de l'écriture du fichier de sortie.

Ici, nous nous intéressons à l'écriture des nœuds correspondant à des données d'entrée, constantes et certaines propriétés. La première vérification permet de déterminer, grâce à l'attribut `entry`, si le nœud est une donnée d'entrée. S'il ne s'agit pas d'un nœud de donnée d'entrée, nous recherchons l'existence de liens (binding, assignment, connector) dont le nœud courant est destination. Si un tel lien existe, le nœud est une propriété, sinon, c'est une constante. Nous écrivons ensuite dans le fichier de sortie le nœud courant en fonction de ses paramètres : `node_type`, `path_id`, `path` et `node_value`.

Le code 5.2 (page 82) présente un exemple d'une partie du fichier de sortie généré par l'outil `smax_to_node`.

2. `https://www.w3.org/TR/xpath-31/`

```
284 if (node_name.equals("core:IntProperty") || node_name.equals("core:↩
    doubleproperty")
285         || node_name.equals("core:BoolProperty") || node_name.equals↩
            ("core:textproperty")) {
286     if (attributes.getNamedItem("entry") != null
287             && attributes.getNamedItem("entry").getNodeValue().↩
                equals("1")) {
288         node_type = "entry";
289     } else {
290         Boolean not_const = false;
291         for (int ii = 0; ii < all_links_length; ii++) {
292             Node n_l = all_links.item(ii);
293             String dst = ((String) xpath.evaluate("@destination", ↩
                n_l)).replace("/", ".");
294             if (dst.contains(path)) {
295                 not_const = true;
296                 break;
297             }
298         }
299         if (not_const) {
300             node_type = "prop";
301         } else {
302             node_type = "const";
303         }
304     }
305     node_smax.write("Node_" + node_type + " " + path_id + " = new ↩
        Node_" + node_type + "(\"" + path
306             + "\", \"" + node_value + "\");\n");
307     node_smax.write("nodes.add(" + path_id + ");\n");
308 }
```

Code 5.10 – Implémentation Java (partie) de l'écriture des nœuds provenant de Smala

5.4.3 fgp_to_node : Génération des données d'entrée des propriétés graphiques

`fgp_to_node` est le deuxième outil de la partie automatique du processus de vérification. Celle-ci traduit les propriétés graphiques en une liste de nœuds Java. Actuellement, le fichier d'entrée contient des lignes au format `Op1 Operateur Op2`.

Entre chaque ligne de ce fichier d'entrée, nous appliquons l'opérateur logique ET.

`fgp_to_node` contient plusieurs méthodes Java permettant d'écrire le fichier de sortie avec les nœuds correspondants. Chacune de ces méthodes Java prend trois paramètres : l'opérateur du formalisme `Operateur`, son opérande de gauche `Op1` et son opérande de droite `Op2`.

C'est également `fgp_to_node` qui génère les constantes de couleur par exemple.

Le code 5.11 illustre une partie de la génération des nœuds provenant des propriétés graphiques dans le cas où nous ne souhaitons vérifier qu'une seule propriété graphique.

Nous avons choisi de présenter le cas de l'opérateur d'égalité de couleur de remplissage. En effet, étant donné que la couleur dépend de trois composantes, nous déclinons cet opérateur `EQCF` (opérateur graphique d'égalité de couleur de remplissage : $=_{c,fill}$) en trois opérateurs d'égalité. Nous ajoutons aux opérandes de base les données nécessaires à la prise en compte des nœuds correspondant à la couleur de remplissage. Par exemple, pour la composante rouge, nous ajoutons `_fill_r` à l'identifiant des opérandes.

```
166   public static void write_op_1prop(String op, String l, String r, ↩
          BufferedWriter node_fgp) throws IOException {
167     if (op.equals("EQCF")) {
168       node_fgp.write("Node_eq n0_eq0 = new Node_eq(\"n0_eq0\", " +↩
              l.replace(".", "_") + "_fill_r , "
169                + r.replace(".", "_") + "_fill_r);\n");
170       node_fgp.write("nodes.add(n0_eq0);\n");
171       node_fgp.write("Node_eq n0_eq1 = new Node_eq(\"n0_eq1\", " +↩
              l.replace(".", "_") + "_fill_g , "
172                + r.replace(".", "_") + "_fill_g);\n");
173       node_fgp.write("nodes.add(n0_eq1);\n");
174       node_fgp.write("Node_and n0_and0 = new Node_and(\"n0_and0\",↩
              n0_eq0, n0_eq1);\n");
175       node_fgp.write("nodes.add(n0_and0);\n");
176       node_fgp.write("Node_eq n0_eq2 = new Node_eq(\"n0_eq2\", " +↩
              l.replace(".", "_") + "_fill_b , "
177                + r.replace(".", "_") + "_fill_b);\n");
178       node_fgp.write("nodes.add(n0_eq2);\n");
179       node_fgp.write("Node_and n0 = new Node_and(\"n0\", n0_and0, ↩
              n0_eq2);\n");
180     } else if (op.equals("EQCS")) {
```

Code 5.11 – Implémentation Java (partie) de l'écriture des nœuds provenant des propriétés graphiques

Le code 5.3 (page 83) présente un exemple d'une partie du fichier de sortie généré par l'outil `fgp_to_node`.

5.4.4 node_to_java : Génération des données d'entrée finales

L'outil `node_to_java` concatène les deux fichiers générés par `smax_to_node` (en premier) et `fgp_to_node` (en deuxième). Cet outil met ce fichier en forme pour avoir une structure de classe Java, la classe `Java_to_check`. Cette classe résultante contient une méthode Java `java_to_check` contenant la liste de l'ensemble des nœuds provenant du code Smala de l'application et des propriétés graphiques à vérifier, ainsi qu'un appel à l'outil `find_proof` implémentant l'algorithme et que nous détaillons dans la prochaine section.

Nous écrivons la classe Java `Java_to_check` à chaque exécution de l'outil GP-Check. De ce fait, il est nécessaire de la recompiler à chaque exécution. C'est le deuxième objectif de l'outil `node_to_java` qui lance une nouvelle compilation de la classe `Java_to_check` après l'avoir générée.

5.4.5 find_proof : implémentation de l'algorithme

L'outil `find_proof` peut prendre deux paramètres (le nœud courant de type `Node_base` et la liste des nœuds) ou trois paramètres (le nœud courant de type `Node_base`, la liste des nœuds et une valeur de type `String`). Nous retrouvons donc les formes $\mathcal{F}(n_c)$ et $\mathcal{F}(n_c, val)$.

Nous rappelons que nous avons pour objectif d'obtenir une chaîne de caractères correspondant à un résultat booléen ou une équation dépendant des données d'entrée permettant de déterminer quels cas vérifient l'exigence graphique. De ce fait, l'outil `find_proof` renvoie un `String`.

Le nœud n_0 est le premier appelé, il correspond soit à l'opérateur graphique renseigné, soit à un opérateur logique ET provenant de la conjonction de plusieurs propriétés graphiques. Au même titre que l'algorithme, la fonction vérifie le type du nœud passé en paramètre et exécute le fonctionnement désiré.

5.4.5.1 Cas de base de la récursion

Les deux conditions d'arrêt de notre récursion sont le cas où le nœud courant est une constante ($\mathcal{F}^{const}(n_c) := n_c.val$) ou une donnée d'entrée ($\mathcal{F}^{entry}(n_c) := n_c.id$).

Dans le premier cas, l'outil renvoie la valeur associée au nœud. Dans le second cas, l'outil renvoie l'identifiant du nœud. Le code 5.12 présente cette implémentation.

```
45      if (n_c.getClass().equals(Node_const.class)) {
47          return ((Node_const) n_c).getVal();
48      }
51      else if (n_c.getClass().equals(Node_entry.class)) {
53          return ((Node_entry) n_c).getId();
54      }
```

Code 5.12 – Implémentation Java des cas de base de la récursion

Nous notons sur ce code l'apport de Java dans l'héritage, qui permet de vérifier la classe de `n_c` qui hérite de `Node_base` ainsi que de permettre le cast de `n_c` dans cette classe héritée même si nous passons un `Node_base` en paramètre de l'outil.

5.4.5.2 Exemples de cas non terminaux

Le code 5.13 présente l'implémentation en Java du cas récursif de l'addition ($\mathcal{F}^{add}(n_c) := \mathcal{F}(n_c.l)$ '+' $\mathcal{F}(n_c.r)$).

```
349     else if (n_c.getClass().equals(Node_add.class)) {
352         return find_proof(((Node_add) n_c).getL(), nodes) + " + " + ↩
                find_proof(((Node_add) n_c).getR(), nodes);
353     }
```

Code 5.13 – Implémentation Java du cas de l'addition

Nous voyons ici que le `+` apparaissant dans l'algorithme est bien un caractère.

```
411     else if (n_c.getClass().equals(Node_connect.class)) {
413         String result = find_proof(((Node_connect) n_c).getSrc(), ↩
                nodes);
414         String s_and = " /\\ ";
415         for (var n_b : nodes) {
416             if (n_b.getClass().equals(Node_bind.class)) {
417                 if (((Node_bind) n_b).getDst().equals(n_c)) {
418                     result += s_and + find_proof(n_b, nodes);
419                 }
420             }
421         }
422         return result;
423     }
```

Code 5.14 – Implémentation Java du cas du connector

Le code 5.14 représente l'implémentation en Java du cas récursif du connector sans valeur ajoutée ($\mathcal{F}^{connect}(n_c) := \mathcal{F}(n_c.src)$ '$\wedge$' $\mathcal{F}(n_b), \forall n_b(n, n_c) \in N_{bind}$).

Nous voyons que pour tous les bindings dont le connector est destination, nous ajoutons un appel de l'outil avec le symbole $\wedge$ entre chaque appel.

5.5 Synthèse

Grâce à ce chapitre qui regroupe la contribution scientifique de cette thèse, nous avons apporté des réponses à nos questions de recherche.

La première de ces questions était *Comment formaliser les éléments de la scène graphique des interfaces humain-machine ?*. Une partie du formalisme que nous avons défini permet de répondre à cette première question. En effet, ce formalisme permet de définir tout élément graphique selon les différentes variables graphiques que l'on retrouve dans la norme SVG : coordonnées, couche, taille, couleur, orientation, forme et transparence. De plus, avec ce formalisme, nous avons formalisé des opérateurs entre les éléments de la scène graphique tels que l'intersection et l'inclusion.

Notre deuxième question de recherche était *Comment mécaniser la vérification des exigences graphiques des systèmes informatiques interactifs ?*. Le processus, l'algorithme et le prototype GPCheck que nous avons développés permettent de répondre à cette question. Le processus présente l'ensemble des actions à entreprendre lors du développement d'un système informatique interactif ainsi que celles nécessaires à la vérification des exigences reliées à sa scène graphique. GPCheck permet de réaliser l'automatisation de l'ensemble des actions de vérification : génération des données d'entrée à partir du code prétraité de l'application (graphe de scène enrichi au format XML) et des propriétés graphiques (exigences graphiques formalisées), vérification des propriétés graphiques en accord avec l'algorithme que nous avons développé, renvoie du résultat de la vérification (réponse booléenne à l'exigence ou équation dépendant des données d'entrée).

Après obtention du résultat, s'il s'agit d'une équation, il faut la résoudre afin d'obtenir les valeurs pour lesquelles les données d'entrée vérifient la propriété. Cette action peut se faire par des outils de résolution symbolique (Mathematica [3] par exemple) ou numérique (MATLAB par exemple).

3. `https://www.wolfram.com/mathematica/`

CHAPITRE 6

GPCHECK : APPLICATION AU TCAS

Nous avons répondu à nos questions de recherche grâce à la contribution scientifique que nous avons présentée dans le chapitre précédent. Nous avons présenté un processus visant à vérifier, en partie automatiquement, des exigences graphiques. Nous avons donc un formalisme qui permet d'exprimer formellement des exigences graphiques. Nous avons un algorithme et un prototype qui permettent d'analyser le code d'une application afin de vérifier les exigences graphiques considérées. Nous obtenons au choix la réponse booléenne de la vérification ou une équation à donner à un solveur.

Nous avons donné l'exemple d'une exigence graphique simple, non impactée par les données d'entrée de l'application interactive, pour illustrer notre processus. De ce fait, nous avons vérifié que l'exigence concernant la couleur du symbole threat aircraft de notre implémentation en Smala du TCAS intégré à l'IVSI était toujours vraie.

Nous nous intéressons dans ce chapitre à l'application de notre processus de vérification aux exigences graphiques du TCAS intégré à l'IVSI. Dans un premier temps, nous rappelons les données d'entrée de ce système. Nous rappelons également les exigences graphiques de ce système que nous avons extraites de la spécification technique ED-143 qui est applicable au TCAS.

Enfin, nous appliquons le processus aux exigences graphiques que nous sommes actuellement capables de vérifier. Cela permet de démontrer la couverture du processus de vérification ainsi que du prototype de vérificateur sur un cas d'étude concret. Ce chapitre représente donc le côté applicatif de la contribution scientifique de cette thèse.

6.1 Rappels sur le TCAS

6.1.1 Données d'entrée de l'interface

Le tableau 6.1 rappelle les données d'entrée du TCAS.

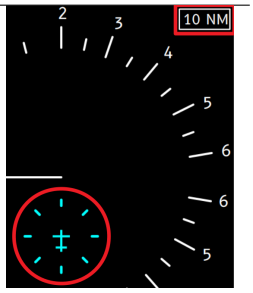
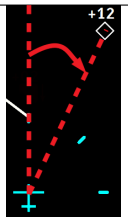
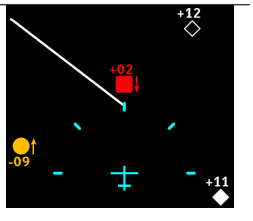
Donnée d'entrée	Informations	Plage de valeur	Illustration
Portée (*range*)	Distance entre l'avion de référence (*own_aircraft*) et le trafic environnant.	$[0; 128]NM$ (pas de $1/12NM$)	
Altitude relative (*relative altitude*)	Distance verticale entre l'avion de référence (*own_aircraft*) et le trafic environnant.	$\pm 12700NM$ (pas de $1/16NM$)	
Sens vertical (*vertical sense*)	Sens de la vitesse verticale	$\{0, 1, 2\}$	
Portée actuelle (*current range*)	Échelle de l'affichage de la portée.	$[6; 40]$ NM	
Relevèment (*bearing*)	Angle entre le cap de l'avion de référence (*own aircraft*) et le trafic environnant.	$\pm 180°$ (pas de $0.2°$)	
Niveau de menace (*threat level*)	Type de trafic environnant (*other*, *proximate*, *intruder*, *threat*)	$\{0, 1, 2, 3, 4\}$	

TABLE 6.1 – Données d'entrée de l'interface du TCAS intégré à l'IVSI

6.1.2 Exigences graphiques du TCAS

Nous avons extrait de la spécification technique ED-143 [3] des exigences relatives à la visualisation du TCAS, applicables à l'IVSI :

- E_1 : *The traffic display shall contain a symbol representing the location of the own aircraft. The colour of the symbol shall be either white or cyan and different than that used to display proximate or other traffic.*
- E_2 : *The inner range ring shall not be solid and shall be comprised of discrete markings at each of the twelve clock positions. The markings shall be the same colour as the own aircraft symbol and of a size and shape that will not clutter the display.*
- E_3 : *The symbol for an RA shall be a red filled square.*
- E_4 : *The symbol for a TA shall be an amber or yellow filled circle.*
- E_5 : *The symbol for Proximate Traffic shall be a white or cyan filled diamond. The colour of the Proximate Traffic symbol should be different than that used for the own aircraft symbol to ensure the symbol is readable.*
- E_6 : *The symbol for Other Traffic shall be a white or cyan diamond, outline only. The colour of the Other Traffic symbol should be different than that used for the own aircraft symbol to ensure the symbol is readable.*
- E_7 : *For an intruder above own aircraft, the tag shall be placed above the traffic symbol and preceded by a "+" sign ; for one below own aircraft, the tag shall be place below the traffic symbol and preceded by a "-" sign. The colour of the data tag shall be the same as the symbol.*
- E_8 : *A vertical arrow shall be placed to the immediate right of the traffic symbol if the vertical speed of the intruder is equal to or greater than 500 fpm, with the arrow pointing up for climbing traffic and down for descending traffic. The colour of the arrow shall be the same as the traffic symbol.*
- E_9 : *A green "fly-to" arc shall be used to provide a target vertical speed whenever a change in the existing vertical speed is desired or when an existing vertical speed must be maintained.*
- E_{10} : *The red arcs on the RA/VSI shall indicate the vertical speed range that must be avoided to maintain or attain the TCAS-desired vertical miss distance from one or more intruders.*

Le tableau 6.2 rappelle ces exigences graphiques du TCAS intégré à l'IVSI et permet de faire apparaître leurs différentes composantes séparément : la forme, la couleur et la position (trois variables graphiques de Bertin et Barbut [26]).

Ex.	**Information**	**Forme**	**Couleur**	**Position**
E_1	Propre position relative (*own aircraft*)	Avion (3 traits)	Blanc ou cyan ($\neq$ couleur *proximate / other aircraft*)	Centré horizontalement, 1/3 hauteur affichage
E_2	Portée de 2 NM autour de *own aircraft*	8 traits en cercle	Couleur *own aircraft*	Centré sur *own aircraft*, rayon en fonction de la portée
E_3	TA - Trafic de type *threat aircraft*	Carré plein	Rouge	Zone délimitée par le compteur de vitesse
E_4	TA - Trafic de type *intruder aircraft*	Cercle plein	Jaune ou ambre	Zone délimitée par le compteur de vitesse
E_5	TA - Trafic de type *proximate aircraft*	Losange plein	Blanc ou cyan ($\neq$ couleur *own aircraft*)	Zone délimitée par le compteur de vitesse
E_6	TA - Trafic de type *other aircraft*	Losange vide	Blanc ou cyan ($\neq$ couleur *own aircraft*)	Zone délimitée par le compteur de vitesse
E_7	TA - Altitude relative (en centaine de pieds)	"+" / "-" et deux chiffres	Couleur trafic	Au-dessus / en dessous du symbole du trafic
E_8	TA - Sens vertical	Flèche verticale (haut ou bas)	Couleur trafic	À droite du symbole du trafic
E_9	RA - Vitesse à atteindre	Arc de cercle	Vert	Compteur de vitesse
E_{10}	RA - Vitesse à éviter	Arc de cercle	Rouge	Compteur de vitesse

TABLE 6.2 – Liste des exigences graphiques du TCAS intégré à l'IVSI

6.2 GPCheck : Application au TCAS

Nous pouvons à présent vérifier les exigences graphiques (tableau 6.2) pour notre implémentation du TCAS intégré à l'IVSI. Nous suivons le processus pour chacune

des exigences mais choisissons de masquer dans cette partie les codes que nous estimons superflus.

Nous n'avons encore implémenté ni l'opérateur d'égalité de forme, ni les constantes de formes. De ce fait, nous ne pouvons pas vérifier à l'aide de GPCheck les exigences relatives à la forme des éléments graphiques. Cependant, nous pouvons actuellement les vérifier à l'état initial de l'application à l'aide du code des éléments graphiques.

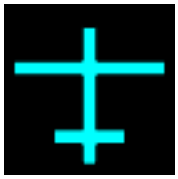
Dans la suite, nous notons *frame* l'élément graphique représentant la fenêtre de l'interface ainsi que *WHITE* pour la constante de couleur blanche (nous représentons les autres constantes de couleur de manière analogue).

6.3 Exigence E_1 : Propre position relative

6.3.1 Formalisation de l'exigence graphique

Les trois exigences concernant l'élément graphique représentant la propre position relative sont :

- forme : avion (3 traits),
- couleur : blanc ou cyan ($\neq$ couleur *proximate / other aircraft*),
- position : centré horizontalement, 1/3 hauteur affichage.



La formalisation de l'exigence E_1 du TCAS intégré à l'IVSI, sans la composante de forme, donne :

$$
\begin{aligned}
E_1 : (&own\_aircraft =_c WHITE \vee own\_aircraft =_c CYAN) \\
&\wedge own\_aircraft \neq_{c,stroke} other\_aircraft \wedge own\_aircraft \neq_c proximate\_aircraft \\
&\wedge own\_aircraft.center.x = frame.width/2 \\
&\wedge own\_aircraft.center.y = frame.height * 2/3
\end{aligned}
$$

La hauteur et la largeur de l'interface mesurent 800 pixels. Nous faisons l'hypothèse que le centre de l'élément graphique doit se trouver à ± 25 px de l'emplacement désiré. Ainsi, le centre en x doit se trouver à 400 ± 25 px. En considérant l'inversion de l'axe des ordonnées en informatique, le centre en y doit se trouver à 533 ± 25 px.

La formalisation de l'exigence E_1 du TCAS intégré à l'IVSI, sans la composante de forme, devient donc $E_1 = E_{1,white} \vee E_{1,cyan}$ avec :

$$
\begin{aligned}
E_{1,white} : & \, own\_aircraft =_c WHITE \\
& \wedge own\_aircraft \neq_c other\_aircraft \\
& \wedge own\_aircraft \neq_c proximate\_aircraft \\
& \wedge own\_aircraft.center.x < 425 \wedge 375 < own\_aircraft.center.x \\
& \wedge own\_aircraft.center.y < 558 \wedge 508 < own\_aircraft.center.y
\end{aligned}
$$

$$
\begin{aligned}
E_{1,cyan} : & \, own\_aircraft =_c CYAN \\
& \wedge own\_aircraft \neq_c other\_aircraft \\
& \wedge own\_aircraft \neq_c proximate\_aircraft \\
& \wedge own\_aircraft.center.x < 425 \wedge 375 < own\_aircraft.center.x \\
& \wedge own\_aircraft.center.y < 558 \wedge 508 < own\_aircraft.center.y
\end{aligned}
$$

6.3.2 Vérification de l'exigence

Le code 6.1 représente l'élément graphique dans le graphe de scène enrichi et présente principalement les formes qui le composent.

```
19  <core:component id="own_aircraft" >
37    <gui:path id="path975-3" d="M 400,522 400,573" displayed="1" />
38    <gui:path id="path979-6" d="M 387,563 413,563" displayed="1" />
39    <gui:path id="path981-7" d="M 380,537 420,537" displayed="1" />
40  </core:component>
```

Code 6.1 – Représentation XML de la forme de l'élément graphique *own aircraft*

Nous voyons que trois segments (`gui:path` dont l'attribut `d` représente les coordonnées de l'ensemble de segments) forment le composant `own_aircraft` :

- `path975-3` possède une abscisse fixe (400 dans les deux couples) donc il s'agit donc d'un segment vertical,
- `path979-6` possède une ordonnée fixe (563 dans les deux couples) donc il s'agit donc d'un segment horizontal,
- `path981-7` possède une ordonnée fixe (537 dans les deux couples) donc il s'agit donc d'un segment horizontal.

Nous passons ensuite les exigences graphiques en entrée de GPCheck et obtenons deux équations non simplifiées (comprenant 58335 caractères pour $E_{1,white}$ et 58337 caractères pour $E_{1,cyan}$) correspondant au résultat de la vérification pour chaque partie de l'exigence ($E_{1,white}$ et $E_{1,cyan}$). Par souci de place et de lisibilité, nous ne présentons pas ces équations non simplifiées.

6.3.3 Visualisation du résultat de la vérification

Nous avons calculé avec MATLAB les résultats de ces deux équations en faisant varier *range* de 0 à 10 avec un pas de 1/12 ainsi que *bearing* de -180 à 180 avec un pas de 1,44 et en fixant *level* à 1 et *current_range* à 6. La figure 6.1 (gauche) représente la visualisation du résultat de la vérification de $E_{1,white}$. La figure 6.1 (droite) représente la visualisation du résultat de la vérification de $E_{1,cyan}$.

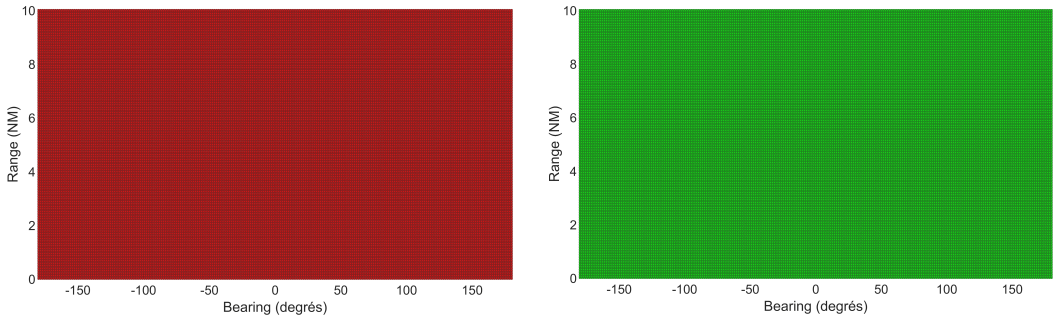


FIGURE 6.1 – Résultats des vérifications de $E_{1,white}$ (gauche) et $E_{1,cyan}$ (droite) avec $level = 1$ et $current\_range = 6$

Sur la figure 6.1 (gauche), nous voyons qu'aucune configuration des données d'entrée ne vérifie l'exigence $E_{1,white}$. Cependant, celle-ci ne représente qu'une partie de l'exigence E_1. Nous avons testé l'exigence de couleur pour que le symbole soit blanc. En testant cette exigence pour la couleur cyan, voire figure 6.1 (droite), l'exigence $E_{1,cyan}$ est toujours vérifiée.

La disjonction des deux résultats donne le résultat représenté par la figure 6.1 (droite) ($false \vee true = true$, le résultat de la figure 6.1 (gauche) n'a donc aucun impact sur la disjonction). Nous pouvons déduire de la seule différence entre les deux ensembles d'exigences graphiques que le symbole de l'*own aircraft* n'est jamais blanc et toujours cyan.

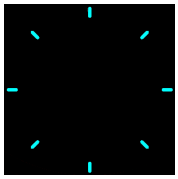
Nous pouvons donc conclure que l'exigence $E_1 = E_{1,white} \vee E_{1,cyan}$ est toujours vérifiée.

6.4 Exigence E_2 : Portée de 2 NM autour de *own aircraft*

6.4.1 Formalisation de l'exigence graphique

Les trois exigences concernant l'élément graphique représentant la portée de 2 NM autour de *own aircraft* sont :

- forme : 8 traits en cercle,
- couleur : couleur *own aircraft*,
- position : centré sur *own aircraft*, rayon en fonction de la portée.



La formalisation de l'exigence E_2 du TCAS intégré à l'IVSI, sans les composantes de forme et de position, donne :

$$E_2 : range\_circle =_c own\_aircraft$$

6.4.2 Vérification de l'exigence

Le code 6.2 représente l'élément graphique dans le graphe de scène enrichi et présente principalement les formes qui le composent et les transformations géométriques qui l'impactent.

```
108     <core:component id="range" >
109       <gui:translation id="t" >
113       <gui:path id="lineToClone" d="M 400,521 400,535" displayed="1↩
              " >
129     </core:component>
130     <gui:rotation id="r1" >
131       <core:doubleproperty id="a" value="45" />
137     <core:component id="range1" >
138       <gui:translation id="t" >
142       <gui:path id="lineToClone" d="M 400,521 400,535" displayed="1↩
              " >
```

Code 6.2 – Représentation XML de la forme de l'élément graphique *range circle*

Nous voyons qu'une translation `t` impacte les 8 segments (`gui:path`) qui la suivent (nous n'en montrons que deux ici par souci de place). Tous ces segments on le

même attribut `d` et donc les mêmes coordonnées de base. Des rotations permettent de mettre ces coordonnées à jour afin de former le cercle.

Nous passons ensuite les exigences graphiques en entrée de GPCheck et obtenons le résultat `true`. Cela prouve que cette exigence ne dépend pas des données d'entrée.

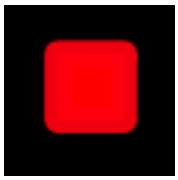
Ainsi, l'exigence $E_{2,couleur}$ est toujours vérifiée et la visualisation du résultat n'est pas nécessaire.

6.5 Exigence E_3 : TA - Trafic de type *threat aircraft*

6.5.1 Formalisation de l'exigence graphique

Les trois exigences concernant l'élément graphique représentant le trafic de type *threat aircraft* sont :

- forme : carré plein,
- couleur : rouge,
- position : zone délimitée par le compteur de vitesse.



La formalisation de l'exigence E_3 du TCAS intégré à l'IVSI, sans la composante de forme, donne :

$$E_3 : threat\_aircraft =_c RED \wedge threat\_aircraft \subset_? vsi$$

Un carré est par définition un rectangle dont la hauteur et la largeur sont égales. De ce fait, nous devons vérifier que $threat\_aircraft.height = threat\_aircraft.width$ (par définition, seuls les rectangles possèdent ces deux attributs).

La formalisation de l'exigence E_3 du TCAS intégré à l'IVSI, sans la composante de forme complète, devient donc :

$$\begin{aligned} E_3 : & threat\_aircraft.height = threat\_aircraft.width \\ & \wedge threat\_aircraft =_c RED \wedge threat\_aircraft \subset_? vsi \end{aligned}$$

6.5.2 Vérification de l'exigence

Nous passons ensuite l'exigence graphique en entrée de GPCheck et obtenons une équation non simplifiée (code 6.3) correspondant au résultat de la vérification.

```
((((((true) & (((((true) & (true))) & (true))))) & (((((true) & (true)
    )) & (true))))) & (~(((((level == 4) & (((range <= current_range) &
     (((abs((bearing)) <= 90) | (range <= 2)))))))))) | ~(true) | (sqrt
    ((400 - (22 + ((400 + ((sin((((bearing) * (3.14159)) / (180)))) * (
    range)) * (((435) / (current_range)))) - (35))))^2 + (400 - (40 +
    ((537 - (((cos((((bearing) * (3.14159)) / (180)))) * (range)) *
    (((435) / (current_range))))) - (52))))^2) < 300 & sqrt((400 - (22
    + ((400 + ((sin((((bearing) * (3.14159)) / (180)))) * (range)) *
    (((435) / (current_range)))) - (35)) + 25))^2 + (400 - (40 + ((537
    - (((cos((((bearing) * (3.14159)) / (180)))) * (range)) * (((435) /
     (current_range))))) - (52))))^2) < 300 & sqrt((400 - (22 + ((400 +
     ((sin((((bearing) * (3.14159)) / (180)))) * (range)) * (((435) / (
    current_range)))) - (35))))^2 + (400 - (40 + ((537 - (((cos((((
    bearing) * (3.14159)) / (180)))) * (range)) * (((435) / (
    current_range))))) - (52)) + 25))^2) < 300 & sqrt((400 - (22 +
    ((400 + ((sin((((bearing) * (3.14159)) / (180)))) * (range)) *
    (((435) / (current_range)))) - (35)) + 25))^2 + (400 - (40 + ((537
    - (((cos((((bearing) * (3.14159)) / (180)))) * (range)) * (((435) /
     (current_range))))) - (52)) + 25))^2) < 300)))
```

Code 6.3 – Résultat de la vérification de l'exigence E_3

6.5.3 Visualisation du résultat de la vérification

Nous avons calculé avec MATLAB les résultats de cette équation en faisant varier *range* de 0 à 10 avec un pas de 1/12 ainsi que *bearing* de -180 à 180 avec un pas de 1,44 et en fixant *level* à 4 et *current_range* à 6. La figure 6.2 représente la visualisation du résultat de la vérification de E_3.

Nous voyons ici que certaines configurations des données d'entrée ne vérifient pas l'exigence E_3 pour notre implémentation Smala du TCAS intégré à l'IVSI. Il s'avère en effet que le symbole du trafic peut apparaître hors de la zone du compteur de vitesse verticale. Cela indique que nous devons modifier notre implémentation afin de respecter cette exigence.

La figure 6.3 présente une configuration des données d'entrée ($range = 5$, $relative\_altitude = 0$, $vertical\_sense = 1$, $bearing = 75$, $level = 4$) qui ne vérifie pas l'exigence E_3.

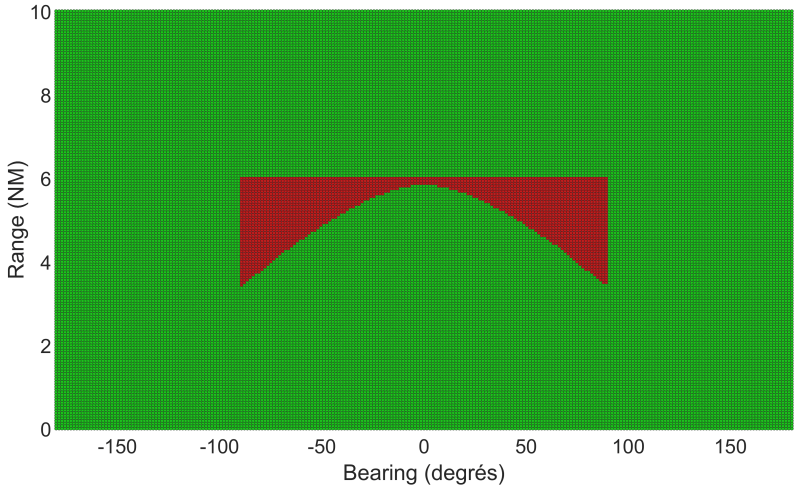


FIGURE 6.2 – Résultat de la vérification de E_3 avec $level = 4$ et $current\_range = 6$

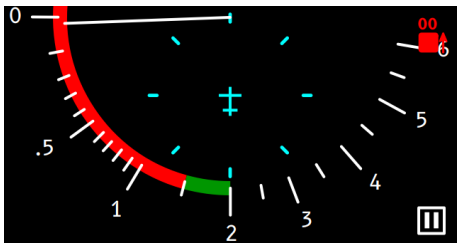


FIGURE 6.3 – Affichage du TCAS intégré à l'IVSI ne vérifiant pas l'exigence E_3

Ici, nous voyons que le symbole du *threat aircraft* se trouve en dehors du compteur de vitesse verticale. Il s'agit là d'une erreur d'implémentation du TCAS intégré à l'IVSI. Il faut donc corriger le code Smala du système afin de respecter cette exigence.

6.6 Exigence E_4 : TA - Trafic de type *intruder aircraft*

6.6.1 Formalisation de l'exigence graphique

Les trois exigences concernant l'élément graphique représentant le trafic de type *intruder aircraft* sont :

- forme : cercle plein,
- couleur : jaune ou ambre,
- position : zone délimitée par le compteur de vitesse.

La formalisation de l'exigence E_4 du TCAS intégré à l'IVSI, sans la composante de forme, donne :

$$E_4 : (intruder\_aircraft =_c YELLOW \vee intruder\_aircraft =_c AMBER) \wedge intruder\_aircraft \subset_? vsi$$

La formalisation de l'exigence E_4 du TCAS intégré à l'IVSI, sans la composante de forme, devient donc $E_4 = E_{4,yellow} \vee E_{4,amber}$ avec :

$$E_{4,yellow} : intruder\_aircraft =_c YELLOW \wedge intruder\_aircraft \subset_? vsi$$
$$E_{4,amber} : intruder\_aircraft =_c AMBER \wedge intruder\_aircraft \subset_? vsi$$

6.6.2 Vérification de l'exigence

Nous passons ensuite les exigences graphiques en entrée de GPCheck et obtenons une équation non simplifiée (code 6.4 pour $E_{4,yellow}$ et code 6.5 pour $E_{4,amber}$) correspondant au résultat de la vérification pour chaque partie de l'exigence ($E_{4,yellow}$ et $E_{4,amber}$).

```
(((((((((true) & (false))) & (true))) & (((((true) & (false))) & (true)
    )))) & (~(((((((level == 2) | (level == 3))) & (((range <=
    current_range) & (((abs((bearing)) <= 90) | (range <= 2)))))))))) |
    ~(true) | (15 + sqrt((400 - (35 + ((400 + ((sin((((bearing) *
    (3.14159)) / (180)))) * (range)) * (((435) / (current_range)))) -
    (35))))^2 + (400 - (53 + ((537 - (((cos((((bearing) * (3.14159)) /
    (180)))) * (range)) * (((435) / (current_range))))) - (52))))^2) <
    300)))
```

Code 6.4 – Résultat de la vérification de l'exigence $E_{4,yellow}$

```
(((((((((true) & (true))) & (true))) & (((((true) & (true))) & (true)))
    )) & (~(((((((level == 2) | (level == 3))) & (((range <=
    current_range) & (((abs((bearing)) <= 90) | (range <= 2)))))))))) |
    ~(true) | (15 + sqrt((400 - (35 + ((400 + ((sin((((bearing) *
    (3.14159)) / (180)))) * (range)) * (((435) / (current_range)))) -
    (35))))^2 + (400 - (53 + ((537 - (((cos((((bearing) * (3.14159)) /
    (180)))) * (range)) * (((435) / (current_range))))) - (52))))^2) <
    300)))
```

Code 6.5 – Résultat de la vérification de l'exigence $E_{4,amber}$

6.6.3 Visualisation du résultat de la vérification

Nous avons calculé avec MATLAB les résultats de ces deux équations en faisant varier *range* de 0 à 10 avec un pas de 1/12 ainsi que *bearing* de -180 à 180 avec un pas de 1,44 et en fixant *level* à 2 et *current_range* à 6. La figure 6.4 (gauche) représente la visualisation du résultat de la vérification de $E_{4,yellow}$. La figure 6.4 (droite) représente la visualisation du résultat de la vérification de $E_{4,amber}$.

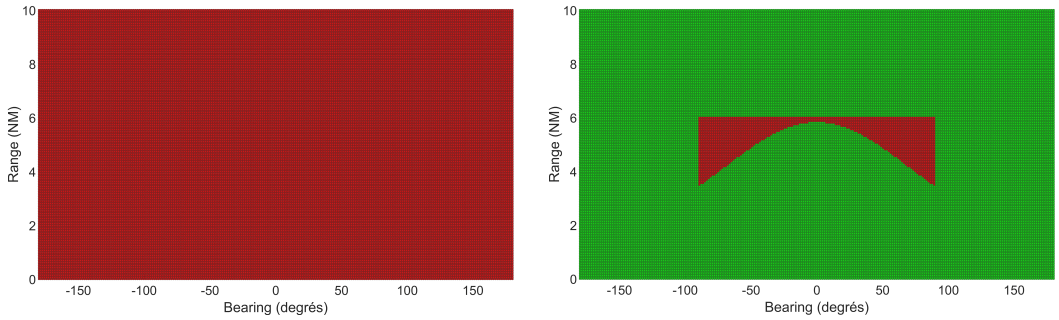


FIGURE 6.4 – Résultats des vérifications de $E_{4,yellow}$ (gauche) et $E_{4,amber}$ (droite) avec $level = 2$ et $current\_range = 6$

Ici, nous voyons qu'aucune configuration des données d'entrée ne valide l'exigence $E_{4,yellow}$. Cependant, il faut réaliser une disjonction avec le résultat suivant afin d'avoir le résultat final de la vérification de l'exigence E_4.

La disjonction des deux résultats donne le résultat représenté par la figure 6.4 (droite) : $false \vee true = true$, le résultat de la figure 6.4 (gauche) n'a donc aucun impact sur la disjonction. Nous pouvons déduire de la seule différence entre les deux ensembles de propriétés graphiques que le symbole de l'*intruder aircraft* n'est jamais jaune et toujours ambre. Au même titre que l'exigence E_3, le symbole du trafic peut apparaître hors de la zone du compteur de vitesse verticale.

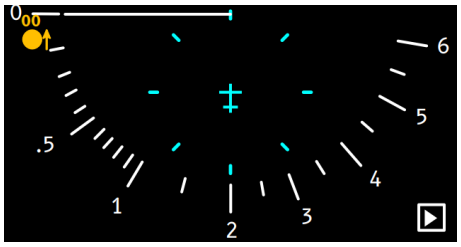


FIGURE 6.5 – Affichage du TCAS intégré à l'IVSI ne vérifiant pas l'exigence E_4

La figure 6.5 présente une configuration des données d'entrée ($range = 5$, $relative\_altitude = 0$, $vertical\_sense = 1$, $bearing = -75$, $level = 2$) qui ne vérifie pas l'exigence E_4.

Ici, nous voyons que le symbole du *intruder aircraft* se trouve en dehors du compteur de vitesse verticale. Il s'agit là d'une erreur d'implémentation du TCAS intégré à l'IVSI. Il faut donc corriger le code Smala du système afin de respecter cette exigence.

6.7 Exigence E_5 : TA - Trafic de type *proximate aircraft*

6.7.1 Formalisation de l'exigence graphique

Les trois exigences concernant l'élément graphique représentant le trafic de type *proximate aircraft* sont :

- forme : losange plein,
- couleur : blanc ou cyan ($\neq$ couleur *own aircraft*),
- position : zone délimitée par le compteur de vitesse.

La formalisation de l'exigence E_5 du TCAS intégré à l'IVSI, sans la composante de forme, donne :

$$\begin{aligned} E_5 : & (proximate\_aircraft =_c CYAN \vee proximate\_aircraft =_c WHITE) \\ & \wedge proximate\_aircraft \neq_c own\_aircraft \\ & \wedge proximate\_aircraft \subset_? vsi \end{aligned}$$

La formalisation de l'exigence E_5 du TCAS intégré à l'IVSI, sans la composante de forme, devient donc $E_5 = E_{5,cyan} \vee E_{5,white}$ avec :

$$\begin{aligned} E_{5,cyan} : & proximate\_aircraft =_c CYAN \\ & \wedge proximate\_aircraft \neq_c own\_aircraft \wedge proximate\_aircraft \subset_? vsi \end{aligned}$$

$$\begin{aligned} E_{5,white} : & proximate\_aircraft =_c WHITE \\ & \wedge proximate\_aircraft \neq_c own\_aircraft \wedge proximate\_aircraft \subset_? vsi \end{aligned}$$

6.7.2 Vérification de l'exigence

Nous passons ensuite les exigences graphiques en entrée de GPCheck et obtenons deux équations non simplifiées (comprenant 94345 caractères pour $E_{5,cyan}$ et 94525 caractères pour $E_{5,white}$) correspondant au résultat de la vérification pour chaque partie de l'exigence ($E_{5,cyan}$ et $E_{5,white}$). Par souci de place et de lisibilité, nous ne présentons pas ces équations non simplifiées.

6.7.3 Visualisation du résultat de la vérification

Nous avons calculé avec MATLAB les résultats de ces deux équations en faisant varier *range* de 0 à 10 avec un pas de 1/12 ainsi que *bearing* de -180 à 180 avec un pas de 1,44 et en fixant *level* à 1 et *current_range* à 6. La figure 6.6 (gauche) représente la visualisation du résultat de la vérification de $E_{5,cyan}$. La figure 6.6 (droite) représente la visualisation du résultat de la vérification de $E_{5,white}$.

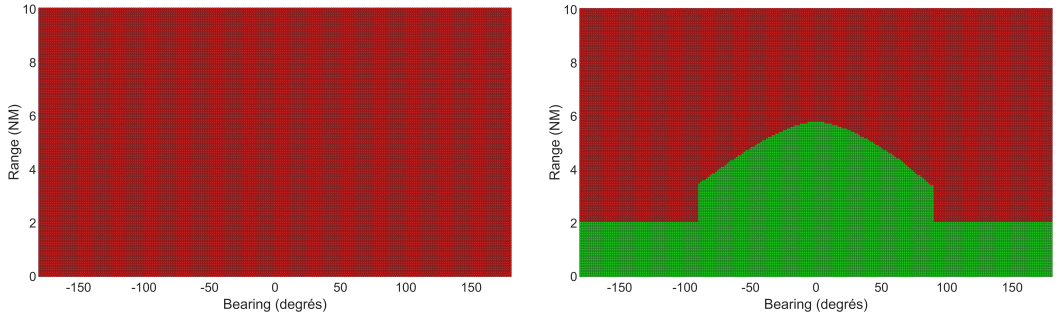


FIGURE 6.6 – Résultats des vérifications de $E_{5,cyan}$ (gauche) et $E_{5,white}$ (droite) avec $level = 1$ et $current\_range = 6$

Ici, nous voyons qu'aucune configuration des données d'entrée ne valide l'exigence $E_{5,cyan}$. Cependant, il faut réaliser une disjonction avec le résultat suivant afin d'avoir le résultat final de la vérification de l'exigence E_5.

La disjonction des deux résultats donne le résultat représenté par la figure 6.6 (droite) : $false \vee true = true$, le résultat de la figure 6.6 (gauche) n'a donc aucun impact sur la disjonction. Nous pouvons déduire de la seule différence entre les deux ensembles de propriétés graphiques que le symbole de l'*intruder aircraft* n'est jamais cyan et toujours blanc.

La figure 6.7 présente une configuration des données d'entrée ($range = 5$, $relative\_altitude = 0$, $vertical\_sense = 1$, $bearing = 90$, $level = 1$) qui ne vérifie pas l'exigence E_5.

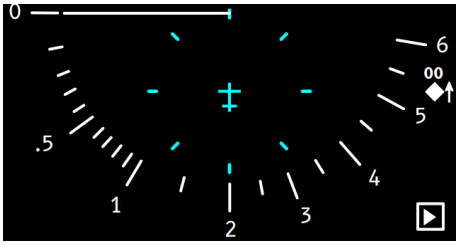


FIGURE 6.7 – Affichage du TCAS intégré à l'IVSI ne vérifiant pas l'exigence E_5

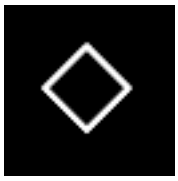
Ici, nous voyons que le symbole du *proximate aircraft* se trouve en dehors du compteur de vitesse verticale. Il s'agit là d'une erreur d'implémentation du TCAS intégré à l'IVSI. Il faut donc corriger le code Smala du système afin de respecter cette exigence.

6.8 Exigence E_6 : TA - Trafic de type *other aircraft*

6.8.1 Formalisation de l'exigence graphique

Les trois exigences concernant l'élément graphique représentant le trafic de type *other aircraft* sont :

- forme : losange vide,
- couleur : blanc ou cyan ($\neq$ couleur *own aircraft*),
- position : zone délimitée par le compteur de vitesse.



La formalisation de l'exigence E_6 du TCAS intégré à l'IVSI, sans la composante de forme, donne :

$$\begin{aligned} E_6 : (&other\_aircraft =_{c,stroke} CYAN \vee other\_aircraft =_{c,stroke} WHITE) \\ &\wedge other\_aircraft \neq_{c,stroke} own\_aircraft \\ &\wedge other\_aircraft \subset_? vsi \end{aligned}$$

La formalisation de l'exigence E_6 du TCAS intégré à l'IVSI, sans la composante

de forme, devient donc $E_6 = E_{6,cyan} \vee E_{6,white}$ avec :

$$E_{6,cyan} : other\_aircraft =_c CYAN \\ \wedge other\_aircraft \neq_c own\_aircraft \wedge other\_aircraft \subset_? vsi$$

$$E_{6,white} : other\_aircraft =_c WHITE \\ \wedge other\_aircraft \neq_c own\_aircraft \wedge other\_aircraft \subset_? vsi$$

6.8.2 Vérification de l'exigence

Nous passons ensuite les exigences graphiques en entrée de GPCheck et obtenons deux équations non simplifiées (comprenant 48739 caractères pour $E_{6,cyan}$ et 48829 caractères pour $E_{6,white}$) correspondant au résultat de la vérification pour chaque partie de l'exigence ($E_{6,cyan}$ et $E_{6,white}$). Par souci de place et de lisibilité, nous ne présentons pas ces équations non simplifiées.

6.8.3 Visualisation du résultat de la vérification

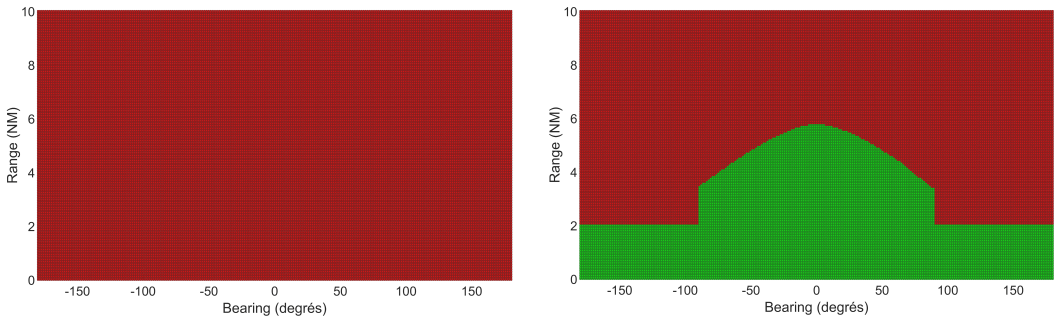


FIGURE 6.8 – Résultats des vérifications de $E_{6,cyan}$ (gauche) et $E_{6,white}$ (droite) avec $level = 1$ et $current\_range = 6$

Nous avons calculé avec MATLAB les résultats de ces deux équations en faisant varier *range* de 0 à 10 avec un pas de 1/12 ainsi que *bearing* de -180 à 180 avec un pas de 1,44 et en fixant *level* à 0 et *current_range* à 6. La figure 6.8 (gauche) représente la visualisation du résultat de la vérification de $E_{6,cyan}$. La figure 6.8 (droite) représente la visualisation du résultat de la vérification de $E_{6,white}$.

Ici, nous voyons qu'aucune configuration des données d'entrée ne valide l'exigence $E_{6,cyan}$. Cependant, il faut réaliser une disjonction avec le résultat suivant afin d'avoir le résultat final de la vérification de l'exigence E_6.

La disjonction des deux résultats donne le résultat représenté par la figure 6.8 (droite) : $false \vee true = true$, le résultat de la figure 6.8 (gauche) n'a donc aucun impact sur la disjonction. Nous pouvons déduire de la seule différence entre les deux ensembles de propriétés graphiques que le symbole de l'*other aircraft* n'est jamais cyan et toujours blanc.

La figure 6.9 présente une configuration des données d'entrée ($range = 5$, $relative\_altitude = 0$, $vertical\_sense = 1$, $bearing = -90$, $level = 0$) qui ne vérifie pas l'exigence E_6.

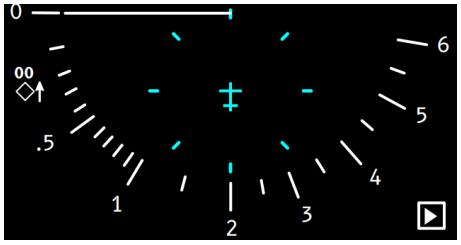


FIGURE 6.9 – Affichage du TCAS intégré à l'IVSI ne vérifiant pas l'exigence E_6

Ici, nous voyons que le symbole du *other aircraft* se trouve en dehors du compteur de vitesse verticale. Il s'agit là d'une erreur d'implémentation du TCAS intégré à l'IVSI. Il faut donc corriger le code Smala du système afin de respecter cette exigence.

6.9 Exigence E_7 : TA - Altitude relative

6.9.1 Formalisation de l'exigence graphique

Les trois exigences concernant l'élément graphique représentant l'altitude relative du trafic sont :

- forme : "+" / "-" et deux chiffres,
- couleur : couleur trafic affiché,
- position : au-dessus / en dessous du symbole du trafic.

La formalisation de l'exigence E_7 du TCAS intégré à l'IVSI, sans les composantes

de forme et de position, donne :

$$\begin{aligned} E_7 : relative\_altitude =_c threat\_aircraft \\ \vee\, relative\_altitude =_c intruder\_aircraft \\ \vee\, relative\_altitude =_c proximate\_aircraft \\ \vee\, relative\_altitude =_{c,stroke} other\_aircraft \end{aligned}$$

La formalisation de l'exigence E_7 du TCAS intégré à l'IVSI, sans la composante de forme, devient donc $E_7 = E_{7,threat} \vee E_{7,intruder} \vee E_{7,proximate} \vee E_{7,other}$ avec :

$$\begin{aligned} E_{7,threat} : relative\_altitude =_c threat\_aircraft \\ E_{7,intruder} : relative\_altitude =_c intruder\_aircraft \\ E_{7,proximate} : relative\_altitude =_c proximate\_aircraft \\ E_{7,other} : relative\_altitude =_{c,stroke} other\_aircraft \end{aligned}$$

6.9.2 Vérification de l'exigence

Nous passons ensuite les exigences graphiques en entrée de GPCheck et obtenons deux équations non simplifiées (comprenant 2976 caractères pour $E_{7,threat}$, 3052 caractères pour $E_{7,intruder}$, 556344 caractères pour $E_{7,proximate}$ et 278168 caractères pour $E_{7,other}$) correspondant au résultat de la vérification pour chaque partie de l'exigence ($E_{7,threat}$, $E_{7,intruder}$, $E_{7,proximate}$ et $E_{7,other}$). Par souci de place et de lisibilité, nous ne présentons pas ces équations non simplifiées.

6.9.3 Visualisation du résultat de la vérification

Nous avons calculé avec MATLAB les résultats de ces deux équations en faisant varier *range* de 0 à 10 avec un pas de 1/12 ainsi que *bearing* de -180 à 180 avec un pas de 1,44 et en fixant *level* à 4 pour $E_{7,threat}$, 2 pour $E_{7,intruder}$, 1 pour $E_{7,proximate}$, 0 pour $E_{7,other}$ et *current_range* à 6. La figure 6.10 représente la visualisation du résultat de la vérification de $E_{7,threat}$.

Les visualisations des résultats des vérifications de $E_{7,intruder}$, $E_{7,intruder}$ et $E_{7,intruder}$ sont les mêmes que celle de $E_{7,threat}$.

Nous supposons ici qu'il s'agit d'un correctif à apporter à notre outil car certains scénarios qui ne valident pas l'exigence d'après la vérification avec GPCheck valident l'exigence. Il s'agirait ici d'imaginer une nouvelle version de l'opérateur graphique

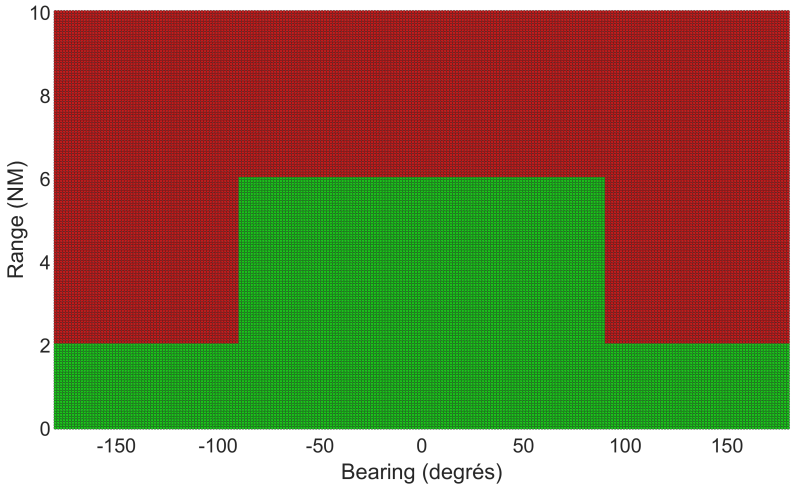


FIGURE 6.10 – Résultat de la vérification de $E_{7,threat}$ avec $level = 4$ et $current\_range = 6$

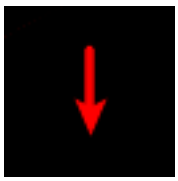
d'égalité des couleurs prenant en compte la notion d'affichage et d'implémenter cette mise à jour.

6.10 Exigence E_8 : TA - Sens vertical

6.10.1 Formalisation de l'exigence graphique

Les trois exigences concernant l'élément graphique représentant le sens vertical du trafic sont :

- forme : flèche verticale (haut ou bas),
- couleur : couleur trafic,
- position : à droite du symbole du trafic.



La formalisation de l'exigence E_8 du TCAS intégré à l'IVSI, sans les composantes de forme et de position, donne :

$$E_8 : vertical\_sense =_c threat\_aircraft \vee vertical\_sense =_c intruder\_aircraft$$
$$\vee\, vertical\_sense =_c proximate\_aircraft$$
$$\vee\, vertical\_sense =_{c,stroke} other\_aircraft$$

La formalisation de l'exigence E_8 du TCAS intégré à l'IVSI, sans la composante

de forme, devient donc $E_8 = E_{8,threat} \vee E_{8,intruder} \vee E_{8,proximate} \vee E_{8,other}$ avec :

$$E_{8,threat} : vertical\_sense =_c threat\_aircraft$$
$$E_{8,intruder} : vertical\_sense =_c intruder\_aircraft$$
$$E_{8,proximate} : vertical\_sense =_c proximate\_aircraft$$
$$E_{8,other} : vertical\_sense =_{c,stroke} other\_aircraft$$

6.10.2 Vérification de l'exigence

Nous passons ensuite les exigences graphiques en entrée de GPCheck et obtenons deux équations non simplifiées (comprenant 2976 caractères pour $E_{8,threat}$, 3169 caractères pour $E_{8,intruder}$, 557280 caractères pour $E_{8,proximate}$ et 278168 caractères pour $E_{8,other}$) correspondant au résultat de la vérification pour chaque partie de l'exigence ($E_{8,threat}$, $E_{8,intruder}$, $E_{8,proximate}$ et $E_{8,other}$). Par souci de place et de lisibilité, nous ne présentons pas ces équations non simplifiées.

6.10.3 Visualisation du résultat de la vérification

Nous avons calculé avec MATLAB les résultats de ces deux équations en faisant varier *range* de 0 à 10 avec un pas de 1/12 ainsi que *bearing* de -180 à 180 avec un pas de 1,44 et en fixant *level* à 4 pour $E_{8,threat}$, 2 pour $E_{8,intruder}$, 1 pour $E_{8,proximate}$, 0 pour $E_{8,other}$ et *current_range* à 6. La figure 6.11 représente la visualisation du résultat de la vérification de $E_{8,threat}$.

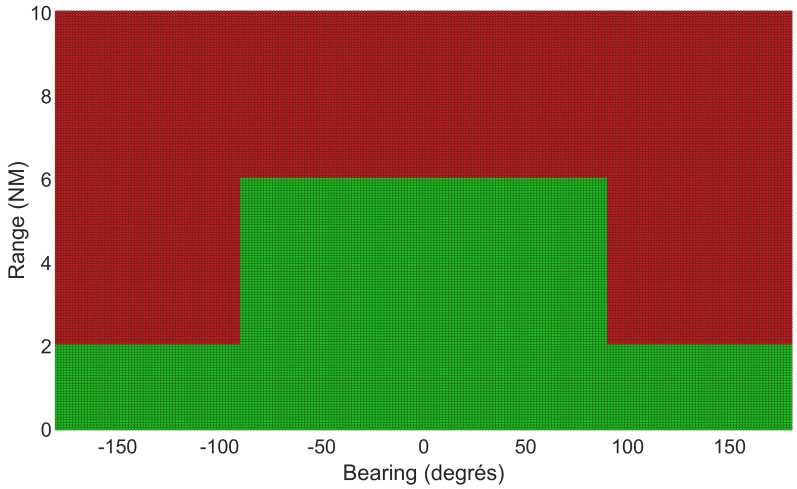


FIGURE 6.11 – Résultat de la vérification de E_8 avec $level = 4$ et $current\_range = 6$

Les visualisations des résultats des vérifications de $E_{8,intruder}$, $E_{8,intruder}$ et $E_{8,intruder}$ sont les mêmes que celle de $E_{8,threat}$.

Nous supposons ici qu'il s'agit d'un correctif à apporter à notre outil car certains scénarios qui ne valident pas l'exigence d'après la vérification avec GPCheck valident l'exigence. Il s'agirait ici d'imaginer une nouvelle version de l'opérateur graphique d'égalité des couleurs prenant en compte la notion d'affichage et d'implémenter cette mise à jour.

6.11 Synthèse

Nous avons appliqué GPCheck et notre processus de vérification à notre implémentation Smala du TCAS intégré à l'IVSI. Nous avons vérifié 8 des 10 exigences graphiques (tableau 6.2) du TCAS intégré à l'IVSI que nous avons extraites de la spécification technique ED-143 [3]. Ces résultats valident le passage à l'échelle de notre outil sur des systèmes informatiques interactifs de taille industrielle et présentant des aspects critiques.

Actuellement, nous avons vérifié les composantes en couleur de ces exigences graphiques ainsi que certaines composantes en forme et en position.

CHAPITRE 7

CONCLUSION ET DISCUSSION

Au cours de cette thèse, nous nous sommes intéressés principalement à deux questions :

- *Comment formaliser les éléments de la scène graphique des interfaces humain-machine ?*
- *Comment mécaniser la vérification des exigences graphiques des systèmes informatiques interactifs ?*

Nous avons présenté de nombreuses contributions que nous rappelons dans ce chapitre. Par rapport à la littérature, nous avons proposé une classification des propriétés liées à l'interaction. Afin de définir formellement la scène graphique, nous avons développé un formalisme. Afin de vérifier des exigences liées à la scène graphique, nous avons développé un algorithme de vérification de ces propriétés ainsi que GPCheck, un outil implémentant cet algorithme. Enfin, nous avons appliqué notre processus de vérification aux exigences d'un système informatique interactif aéronautique critique : le TCAS.

Nous discutons ici des résultats actuels de cette thèse. Nous abordons les points forts comme la proposition d'un formalisme pour définir les exigences graphiques des systèmes interactifs. Nous abordons également les limites et les pistes d'amélioration afin d'optimiser certains de nos résultats.

Enfin, nous nous intéressons aux possibles suites de cette thèse. Nous envisageons la prise en compte de nouveaux langages de programmation ou de nouveaux opérateurs graphiques. Nous avons aussi pour objectif d'améliorer le prototype de GPCheck.

7.1 Bilan des contributions de la thèse

L'objectif de cette thèse était d'adresser la problématique de la vérification formelle des propriétés graphiques des systèmes informatiques interactifs. Pour cela, nous avons cherché à répondre à deux questions de recherche :

- Comment formaliser les éléments de la scène graphique des interfaces humain-machine ?
- Comment mécaniser la vérification des exigences graphiques des systèmes informatiques interactifs ?

Nous faisons maintenant le bilan des réalisations et apports de cette thèse par rapport à ces deux questions. Pour cela, nous rappelons ci-dessous chaque contribution en organisant sa présentation de la manière suivante :

- rappel des problèmes principaux découverts à l'occasion de l'état de l'art sous la forme de "findings".
- bilan des principales contributions de la thèse.

7.1.1 Classification des propriétés liées à l'interaction

Finding 1. Il n'existe pas de classification de référence des propriétés liées à l'interaction.

Nous avons voulu nous intéresser aux propriétés graphiques des systèmes informatiques interactifs afin de pouvoir les vérifier.

Notre premier objectif était d'identifier les propriétés liées à l'interaction et d'étudier les méthodes formelles utilisées pour leur vérification. Pour cela, nous avons étudié les 7 occurrences (2006-2018) du workshop *Formal Methods for Interactive Systems* (FMIS) répertoriant 43 articles pour un total de 94 auteurs différents.

Dans cette littérature, nous n'avons pas trouvé de classification faisant référence et définissant les différentes propriétés liées à l'interaction. Cela nous a conduits à ce premier finding.

De plus, nous avons trouvé dans la littérature des propriétés ayant des définitions différentes d'un article à l'autre. Devant ce constat, nous avons pensé qu'une telle classification était nécessaire et pertinente. D'une part, elle permet de proposer un cadre unique de référence à la communauté scientifique. D'autre part, elle nous fournit le cadre de travail pour cette thèse.

De ce fait, nous avons proposé une classification (figure 7.1) de ces propriétés.

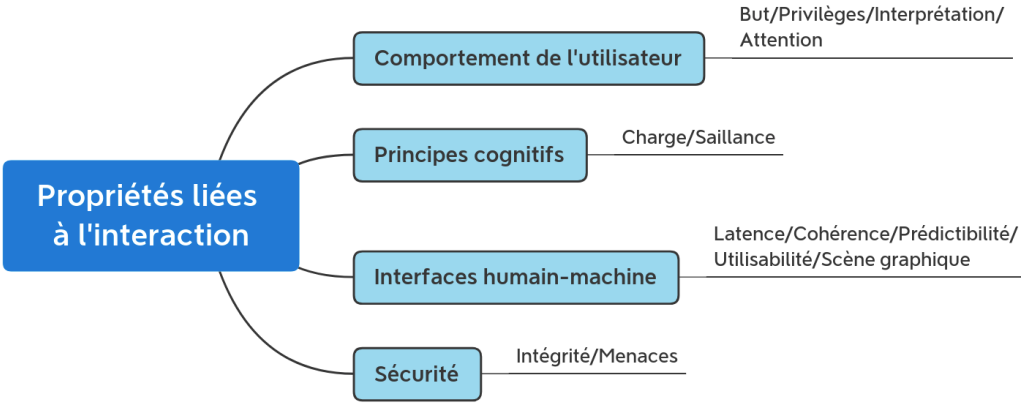


FIGURE 7.1 – Classification des propriétés liées à l'interaction

Nous avons ordonné ces propriétés liées à l'interaction selon les quatre classes suivantes :

- le **comportement de l'utilisateur** [4] : propriétés liées à un utilisateur humain,
- les **principes cognitifs** [43] : propriétés liées aux sciences cognitives,
- l'**interface humain-machine** [18] : propriétés liées aux problèmes induits par l'affichage et les divers interacteurs,
- la **sécurité** [103] : propriétés liées aux menaces de cyber-sécurité.

Cependant, notre classification ne permet pas nécessairement de trier toutes les propriétés des systèmes informatiques interactifs. En effet, avant d'être interactifs, ceux-ci restent des systèmes informatiques. Aussi, nous n'avons pas pris en compte par exemple des propriétés liées aux calculs (e.g. bornitude) ou de vivacité des programmes (e.g. retour à un état donné, absence d'interblocage) [92].

7.1.2 Vérification formelle des propriétés liées à l'interaction

Finding 2. Il n'existe pas de cartographie de la couverture des propriétés liées à l'interaction par les méthodes formelles.

De plus, dans cette littérature, nous n'avons pas trouvé de travaux indiquant la couverture de ces propriétés liées à l'interaction par les méthodes formelles, qu'il s'agisse d'expression formelle ou de vérification formelle de ces propriétés. Cela nous

a conduits à identifier notre deuxième et notre troisième finding ainsi qu'à proposer notre propre cartographie.

A partir de l'analyse des papiers publiés dans les workshops FMIS, nous avons proposé une cartographie qui permet de répartir les travaux des méthodes formelles appliquées aux systèmes interactifs (tableau 7.1, ✓ : 1-5 articles ; ✓✓ : 6-10 articles ; ✓✓✓ : 10+ articles). Le workshop FMIS est totalement dédié à la problématique qui nous intéresse pour cette cartographie : l'application des méthodes formelles aux propriétés des systèmes interactifs.

	Comportement utilisateur	Principes cognitifs	IHM scène graphique	IHM autres	Sécurité	Modélisation systèmes
Algèbre de processus	✓					✓✓
Langage de spécification	✓	✓		✓✓	✓	✓✓✓
Processus de raffinement				✓		✓
Système de transition d'états	✓			✓	✓	✓✓
Logique temporelle		✓		✓	✓	✓
Autres ad hoc	✓	✓		✓	✓	✓✓

TABLE 7.1 – Synthèse des méthodes formelles appliquées aux systèmes interactifs (étude FMIS)

Nous observons clairement qu'il n'y a pas de travaux relatifs à la scène graphique. Afin de renforcer cette principale information que nous avons extraite de cette étude bibliographique, nous avons également étudié les 11 occurrences (2009-2019) de la conférence *Engineering Interactive Computing Systems* (EICS) répertoriant 458 articles pour un total de 842 auteurs différents. Nous nous sommes uniquement intéressés dans cette étude à l'application des méthodes formelles (expression ou vérification) aux propriétés de la scène graphique. Nous avons dénombré 5 publications (environ 1.1% des publications EICS) abordant des problématiques liées aux propriétés de la scène graphique. Parmi celles-ci, seulement une adresse la vérification des propriétés graphiques et celle-ci correspond à des travaux de notre équipe.

7.1.3 Développement d'un formalisme dédié à la scène graphique

Finding 3. Il n'existe pas de formalisme adéquat pour exprimer et vérifier formellement les propriétés de la scène graphique.

Nous voyons grâce à la cartographie que les propriétés de la scène graphique n'ont pas bénéficié de la même attention que les autres propriétés liées à l'interaction dans l'utilisation des méthodes formelles.

Des travaux scientifiques, hors des conférences dédiées aux méthodes formelles, ont étudié les propriétés de la scène graphique. Rappelons notamment les travaux de Bertin et Barbut [26] et Wilkinson [125] qui ont proposé des variables graphiques permettant de définir les différentes composantes des éléments graphiques.

Randell et Cohn [94] ont présenté un formalisme composé d'opérateurs spatiaux permettant d'étudier la connexion de deux régions (que nous définissons comme l'intersection entre deux éléments graphiques) et un ensemble de propriétés qui dépendent de cette notion de connexion. Cependant, ces travaux ne s'intéressent qu'à la notion de position. De plus, leur raisonnement se base sur l'intersection et non pas les coordonnées des éléments graphiques. Nous avons estimé que cela n'était pas assez précis pour vérifier des exigences de la scène graphique. En effet, il est impossible avec la notion de position uniquement de vérifier des propriétés de couleur ou de forme. De plus, il est nécessaire d'avoir au moins accès à des propriétés dépendant des coordonnées des éléments graphiques pour utiliser une technique d'analyse de code par exemple.

En réponse au finding 3, nous avons donc créé notre formalisme (tableaux 7.2 et 7.3) basé sur des variables graphiques de Bertin et Barbut [26] : coordonnées, couleur, orientation, forme et taille.

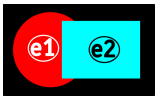
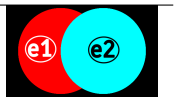
Opérateur	Formalisation	Illustration
ordre d'affichage de e_1 inférieur à celui de e_2	$e_1 <_d e_2 \Leftrightarrow l(e_1) < l(e_2)$	
intersection de e_1 et e_2	$e_1 \cap_? e_2 \Leftrightarrow \neg d(e_1) \vee \neg d(e_2) \vee (\mathcal{D}(e_1) \cap \mathcal{D}(e_2)) \neq \emptyset$	

TABLE 7.2 – Opérateurs graphiques formels

inclusion de e_1 dans e_2	$e_1 \subset_? e_2 \Leftrightarrow \neg d(e_1) \vee \neg d(e_2) \vee \mathcal{D}(e_1) \subset \mathcal{D}(e_2)$	
égalité des formes de e_1 et e_2	$e_1 =_s e_2 \Leftrightarrow \neg d(e_1) \vee \neg d(e_2) \vee \mathcal{D}(e_1) = \mathcal{D}(e_2)$	
égalité des couleurs de e_1 et e_2	$e_1 =_c e_2 \Leftrightarrow c_i(e_1) = c_i(e_2), \forall i \in [r, g, b]$ $e_1 \neq_c e_2 \Leftrightarrow \exists i \in [r, g, b] \mid c_i(e_1) \neq c_i(e_2)$	

TABLE 7.3 – Opérateurs graphiques formels (suite)

Ce formalisme nous permet de raisonner directement sur les variables graphiques mais aussi sur celles qui sont à la base du SVG : coordonnées (x, y), couleur (système RGB), orientation (rotations), forme (rectangle, cercle, segment), taille (largeur, hauteur, rayon), couche d'affichage (arbre XML).

Nous sommes donc capables de définir formellement des exigences graphiques des systèmes informatiques interactifs. Nous pouvons exploiter ces définitions formelles jusqu'au code de la scène graphique de ces systèmes.

7.1.4 Développement d'un algorithme de vérification des propriétés graphiques

Il n'existe pas d'algorithme de vérification formelle des propriétés de la scène graphique. Cette observation est très fortement liée à l'absence de formalisme dédié à l'expression formelle des propriétés de la scène graphique.

En réponse à cette observation, dans un premier temps, nous avons défini un algorithme permettant de vérifier formellement des propriétés graphiques.

Cet algorithme permet d'analyser le graphe de scène enrichi, au format XML, d'un système informatique interactif. Ce graphe de scène enrichi contient les informations de la scène graphique (éléments graphiques et transformations géométriques) ainsi que toutes les informations nécessaires à la propagation des événements internes ou externes du système (composants de couplage, variables constantes ou non, données d'entrée, etc.).

Cet algorithme part de la propriété à vérifier et remonte jusqu'aux données d'entrée du système.

Nous avons conçu une fonction récursive qui permet de traiter les différents types

de nœuds (en notant n_c le nœud courant) de ce graphe de scène enrichi :

$$\mathcal{F}^{type(n_c)}(n_c)$$

La première entrée de cette fonction est une propriété graphique, une exigence graphique définie formellement avec notre formalisme : le premier n_c correspond donc à un opérateur graphique ou à l'opérateur logique ET. La sortie de cette fonction est le résultat de la vérification et peut prendre une forme au choix entre :

- une réponse booléenne (vrai, faux),
- une équation permettant de déterminer quelles conditions d'utilisation (valeurs des données d'entrée de l'application interactive) vérifient l'exigence.

7.1.5 Développement de GPCheck, implémentation de l'algorithme de vérification

Afin d'implémenter la fonction récursive $\mathcal{F}^{type(n_c)}(n_c)$, nous avons développé GPCheck, un prototype de vérificateur d'exigences graphiques codé en Java. GPCheck prend en entrée le fichier XML du graphe de scène enrichi d'une application interactive Smala et un fichier qui contient l'exigence graphique ou la conjonction des exigences graphiques à vérifier. GPCheck renvoie en sortie un booléen (*true*, *false*) ou une équation dans laquelle les données d'entrée interviennent. Afin de résoudre cette équation, il faut la donner à un solveur pour obtenir les scénarios d'utilisation de l'application qui vérifient l'exigence.

7.1.6 Vérification des exigences graphiques du TCAS

Dans notre revue de littérature, nous n'avons pas d'application de vérification formelle de propriétés ou exigences de la scène graphique. Dans le cadre de cette thèse, nous avons donc appliqué GPCheck, notre vérificateur de propriétés graphiques, et l'algorithme qu'il implémente sur un cas d'étude concret industriel.

Nous avons présenté notre cas d'étude aéronautique : le TCAS (*Trafic alert and Collision Avoidance System*) intégré à l'IVSI (*Instantaneous Vertical Speed Indicator*), un équipement du cockpit des aéronefs. Il s'agit d'une interface qui contient un compteur de vitesse verticale, des informations sur le trafic alentour (niveau de menace, altitude relative, sens vertical) ainsi que des informations sur la manœuvre

à effectuer en cas de risque. Nous avons implémenté cette interface en Smala (figure 4.5).

Le TCAS est un système aéronautique qui doit respecter les exigences répertoriées dans la spécification technique ED-143 [3]. Nous avons extrait de la spécification technique ED-143 des exigences graphiques relatives à l'affichage des données dans l'IVSI. Nous les avons retranscrites dans le tableau 7.4 selon leurs composantes de forme, couleur et position.

Les cellules grisées correspondent aux exigences que nous avons vérifiées avec GPCheck pour notre implémentation Smala du TCAS intégré à l'IVSI.

Ex.	**Information**	**Forme**	**Couleur**	**Position**
E_1	Propre position relative (*own aircraft*)	Avion (3 traits)	Blanc ou cyan ($\neq$ couleur *proximate / other aircraft*)	Centré horizontalement, 1/3 hauteur affichage
E_2	Portée de 2 NM autour de *own aircraft*	8 traits en cercle	Couleur *own aircraft*	Centré sur *own aircraft*, rayon en fonction de la portée
E_3	TA - Trafic de type *threat aircraft*	Carré plein	Rouge	Zone délimitée par le compteur de vitesse
E_4	TA - Trafic de type *intruder aircraft*	Cercle plein	Jaune ou ambre	Zone délimitée par le compteur de vitesse
E_5	TA - Trafic de type *proximate aircraft*	Losange plein	Blanc ou cyan ($\neq$ couleur *own aircraft*)	Zone délimitée par le compteur de vitesse
E_6	TA - Trafic de type *other aircraft*	Losange vide	Blanc ou cyan ($\neq$ couleur *own aircraft*)	Zone délimitée par le compteur de vitesse
E_7	TA - Altitude relative (en centaine de pieds)	"+" / "-" et deux chiffres	Couleur trafic	Au-dessus / en dessous du symbole du trafic
E_8	TA - Sens vertical	Flèche verticale (haut ou bas)	Couleur trafic	A droite du symbole du trafic
E_9	RA - Vitesse à atteindre	Arc de cercle	Vert	Compteur de vitesse
E_{10}	RA - Vitesse à éviter	Arc de cercle	Rouge	Compteur de vitesse

TABLE 7.4 – Liste des exigences graphiques du TCAS intégré à l'IVSI

7.2 Discussion

Dans le cadre de cette thèse, nous avons développé un formalisme, un algorithme et un outil qui permettent de vérifier les exigences graphiques d'un système informatique interactif dès lors que nous en possédons un graphe de scène enrichi au format XML. Ce graphe de scène enrichi est indépendant du langage de programmation utilisé pour développer le système informatique interactif. Nous pouvons donc en théorie vérifier les exigences graphiques de tout système informatique interactif.

Cependant, il nous faut discuter des limites de ces contributions afin d'explorer les perspectives d'amélioration pour la suite.

7.2.1 Formalisme de Randell *et al.* [96]

Au cours de notre état de l'art (section 2.2.3.5, page 25), nous avons identifié les travaux de Randell *et al.* [96]. Les auteurs ont développé un formalisme afin d'exprimer formellement des propriétés graphiques liées à la notion de connexion entre des régions. Cette notion de connexion représente l'intersection des domaines de ces régions : si un point existe dans les deux régions, alors il y a connexion. Pour deux régions x et y, la connexion de x avec y s'écrit $C(x, y)$ et les auteurs ont introduit deux axiomes de cette relation : $\forall x C(x, x)$ et $\forall xy[C(x, y) \rightarrow C(y, x)]$.

Les auteurs ont défini plusieurs relations à partir de la relation de connexion : x est déconnectée de y, x est une partie (propre) de y, x est identique à y, x chevauche (partiellement) y, x est une partie propre tangentielle de y, etc. Nous voyons là un grand nombre de relations permettant d'exprimer formellement de nombreuses propriétés de positions relatives entre deux régions.

Concernant notre formalisme, nous sommes capables d'exprimer formellement l'intersection de deux éléments graphiques grâce à leurs coordonnées, pour autant nous ne sommes pas capables d'exprimer formellement une intersection ou une inclusion tangentielle par exemple. Même si notre formalisme permet d'exprimer des propriétés de positions des éléments graphiques à l'aide de leurs variables graphiques, nous n'avons cependant pas un éventail de propriété de position aussi complet que celui de Randell *et al.* [96] actuellement.

De plus, Randell *et al.* [95] ont défini des axiomes pour exprimer la position relative de deux corps par rapport à un point de vue donné. Concernant les propriétés de position, nous n'avons pas la possibilité d'exprimer formellement, et donc de vérifier formellement, des notions de positions relatives. Par exemple, nous ne

pouvons pas exprimer ou vérifier l'exigence $E_{8,position}$ du TCAS indiquant que la flèche du sens vertical doit se trouver à droite du symbole du trafic.

Dans la section 7.3.1 (page 155), nous nous positionnons par rapport aux travaux de Randell *et al.* [96].

7.2.2 Couverture de l'analyse

Nous avons défini dans notre formalisme un opérateur d'égalité des formes (pour deux éléments graphiques e_1 et e_2, $e_1 =_s e_2 \Leftrightarrow \neg d(e_1) \vee \neg d(e_2) \vee \mathcal{D}(e_1) = \mathcal{D}(e_2)$, avec $d(e_i)$ l'affichage de e_i et $\mathcal{D}(e_i)$ le domaine de e_i). Pour autant, dans notre processus de vérification des exigences graphiques du TCAS, nous n'avons pas vérifié d'exigences de forme par le biais de cet opérateur. Il nous est possible de vérifier qu'un rectangle est un carré en utilisant l'opérateur d'égalité mathématique avec sa hauteur et sa largeur en opérandes. Cependant, nous n'avons pas implémenté l'opérateur graphique d'égalité des formes dans GPCheck. Nous sommes donc actuellement dans l'impossibilité de vérifier des exigences de forme sur l'ensemble des scénarios d'utilisation d'un système informatique interactif. Par exemple, nous ne pouvons donc pas exprimer l'exigence $E_{3,forme}$ du TCAS indiquant que l'élément graphique *threat aircraft* est un carré plein.

De plus, nous ne prenons pas en compte toutes les formes géométriques existant dans la norme SVG et certaines ne sont pas considérées pour l'ensemble des opérateurs graphiques que nous avons définis. Par exemple, nous ne pouvons pas exprimer la position et le domaine d'un texte même en l'abstrayant à son rectangle englobant. La longueur d'un texte dépendant du nombre de caractères et d'informations dépendant de la police d'écriture utilisée, il nous est impossible d'avoir accès au domaine d'un texte.

Nous pensons qu'il ne s'agit pas de limites difficiles à dépasser. Dans la section 7.3.2 (page 156), nous nous positionnons par rapport aux limites de couverture de l'analyse.

7.2.3 Adressage de la perception utilisateur

Lorsque nous nous intéressons à des interfaces humain-machine, il faut également considérer l'humain qui utilise le système. De ce fait, en plus des exigences graphiques du système il est important de vérifier les problématiques de perceptibilités des éléments graphiques de l'interface.

La première problématique se pose sur la notion de masquage des éléments graphiques. Par exemple, même si ce n'est pas précisé dans un cahier des charges ou une norme, il peut être crucial qu'une alarme affichée soit toujours affichée au-dessus de tous les autres éléments graphiques de l'interface afin d'assurer une absence de masquage. Pour ce qui est du masquage, il faut aussi pouvoir assurer que l'élément graphique dont on veut assurer la perceptibilité est d'une couleur différente des éléments graphiques situés en dessous de celui-ci. Cet opérateur graphique de masquage n'existe pas dans notre formalisme et nous ne pouvons donc pas vérifier cette composante de la perceptibilité.

De plus, notre opérateur graphique d'égalité des couleurs est défini à l'aide d'opérateurs d'égalité mathématique stricte ($e_1 =_c e_2 \Leftrightarrow c_i(e_1) = c_i(e_2), \forall i \in [r, g, b]$). Nous pouvons vérifier qu'un composant est rouge. Il nous est cependant impossible de prendre en compte dans cette vérification des variations, même légères, de rouge.

Nous avons défini notre opérateur d'égalité des couleurs dans le système RGB, conformément à la norme SVG. Cependant, l'opacité apparaît dans les variables graphiques de Wilkinson [125] ainsi que dans la norme SVG. Actuellement, nous ne la prenons pas en compte dans notre opérateur d'égalité de couleur. Cependant, même si l'opacité n'apparaissait pas dans notre cas d'étude, cette variable graphique peut apparaître dans tout système informatique interactif. Nous devons donc la considérer tout en prenant en compte le(s) élément(s) graphique(s) dont la couche est inférieure à celle de l'élément graphique de référence afin d'obtenir la couleur résultante réelle.

Nous pensons qu'il ne s'agit pas de limites difficiles à dépasser. Dans la section 7.3.3 (page 157), nous nous positionnons par rapport à l'adressage de la perception utilisateur.

7.2.4 Dépendance à Smala

Nous avons défini notre processus de vérification sur la base de l'analyse d'un graphe de scène enrichi au format XML. Cependant, ce graphe est porteur d'une sémantique que nous avons empruntée au langage Smala. Ainsi, chaque nœud du graphe de type transformation géométrique ou colorimétrique impacte ses enfants et les nœuds à sa droite.

De plus, nous avons emprunté la sémantique des composants de couplage de Smala. Ainsi, un nœud de binding propage une activation sans pouvoir propager de valeur, un nœud d'assignment propage une valeur sans propager d'activation et un

nœud de connector propage une valeur tout en propageant une activation. Ceci est spécifique à Smala et peut ne pas l'être pour d'autres langages de programmation réactive comme Java FX et Python.

Il nous semble possible d'utiliser notre processus de vérification avec d'autres langages de programmation réactive si, d'un point de vue sémantique, le graphe de scène enrichi au format XML obtenu depuis la sérialisation d'un système informatique interactif codé en Java FX, par exemple, est équivalent à celui que nous obtenons depuis Smala. Dans la section 7.3.4 (page 158), nous nous positionnons par rapport à la dépendance à Smala.

7.2.5 Utilisabilité de GPCheck

Nous avons observé des limites d'utilisabilité lors de l'utilisation de GPCheck avec notre cas d'étude composé de relativement peu d'éléments graphiques et données d'entrée.

GPCheck prend en entrée une propriété graphique ou un ensemble de propriétés graphiques. Cependant, cet ensemble de propriétés graphiques ne peut être qu'une conjonction de propriétés graphiques. Ainsi, comme nous l'avons vu, nous ne pouvons pas vérifier en une seule utilisation de GPCheck la propriété $E_{4,couleur}$ du TCAS qui indique que le cercle plein du *intruder aircraft* doit être jaune **ou** ambre.

Concernant le résultat obtenu en retour de la vérification avec l'outil, nous avons décelé un problème lors de la résolution avec MATLAB dans un cas particulier. GPCheck ne fait pas de simplification de l'équation de sortie. De ce fait, les équations peuvent avoir une taille conséquente ainsi qu'une profondeur de parenthésage trop importante. En effet, en vérifiant plusieurs exigences simultanément, nous avons remarqué une profondeur de parenthésage supérieure à 32. Il s'agit de la profondeur maximale autorisée par MATLAB, nous n'avons donc pas pu visualiser ce résultat.

Dans la section 7.3.5 (page 159), nous nous positionnons par rapport au limites d'utilisabilité de GPCheck.

7.3 Perspectives

Cette thèse constitue un premier pas pour lier les propriétés graphiques du domaine de l'interface humain-machine et des méthodes formelles. Nous avons concentré nos travaux sur un langage de programmation spécifique, Smala. Nous avons créé un prototype de vérificateur d'exigences graphiques des systèmes informatiques in-

teractifs. Nous avons réussi à vérifier certaines exigences graphiques d'un système aéronautiques.

Nous proposons ici quelques réponses ou propositions envisageables face aux limites que nous avons développées précédemment.

7.3.1 Formalisme de Randell *et al.* [96]

Reprenons la relation de connexion ($C(x, y)$) de Randell *et al.* [96]. Celle-ci est basée sur deux axiomes : $\forall x C(x, x)$ et $\forall xy[C(x, y) \rightarrow C(y, x)]$.

Nous pensons que notre opérateur d'intersection ($e_1 \cap_? e_2$) ou la partie de cet opérateur correspondant à l'intersection des domaines ($(\mathcal{D}(e_1) \cap \mathcal{D}(e_2)) \neq \emptyset$) pourrait avoir une équivalence avec cette relation connexion $C(x, y)$.

$$e_1 \cap_? e_2 \Leftrightarrow \neg d(e_1) \vee \neg d(e_2) \vee (\mathcal{D}(e_1) \cap \mathcal{D}(e_2)) \neq \emptyset$$
$$e_1 \cap_? e_2 \Leftrightarrow \neg d(e_1) \vee \neg d(e_2) \vee \exists (x_1, y_1) \in \mathcal{D}(e_1), (x_2, y_2) \in \mathcal{D}(e_2) \mid (x_1, y_1) = (x_2, y_2)$$

Prenons le cas de l'intersection d'un élément graphique e avec lui-même :

$$e \cap_? e \Leftrightarrow \neg d(e) \vee \neg d(e) \vee \exists (x_1, y_1) \in \mathcal{D}(e), (x_2, y_2) \in \mathcal{D}(e) \mid (x_1, y_1) = (x_2, y_2)$$

Par définition, $\exists (x_1, y_1) \in \mathcal{D}(e), (x_2, y_2) \in \mathcal{D}(e) \mid (x_1, y_1) = (x_2, y_2)$ est toujours vraie. Nous vérifions donc :

$$\boxed{\forall e[e \cap_? e]}$$

Cela nous donne un premier lien avec la relation de connexion de Randell *et al.* [96] :

$$\forall x C(x, x)$$

L'opérateur $\vee$ est commutatif, nous pouvons donc écrire :

$$\neg d(e_1) \vee \neg d(e_2) \Leftrightarrow \neg d(e_2) \vee \neg d(e_1)$$

L'opérateur $=$ est commutatif, nous pouvons donc écrire :

$$(x_1, y_1) = (x_2, y_2) \Leftrightarrow (x_2, y_2) = (x_1, y_1)$$

Nous avons donc :

$$\begin{aligned}&\neg d(e_1) \vee \neg d(e_2) \vee \exists (x_1, y_1) \in \mathcal{D}(e_1), (x_2, y_2) \in \mathcal{D}(e_2) \mid (x_1, y_1) = (x_2, y_2)\\ \Leftrightarrow &\neg d(e_2) \vee \neg d(e_1) \vee \exists (x_2, y_2) \in \mathcal{D}(e_2), (x_1, y_1) \in \mathcal{D}(e_1) \mid (x_2, y_2) = (x_1, y_1)\end{aligned}$$

Cela nous donne la relation d'équivalence suivante :

$$\boxed{\forall e_1 e_2 [e_1 \cap_? e_2 \Leftrightarrow e_2 \cap_? e_1]}$$

Cela nous donne un deuxième lien avec la relation de connexion de Randell *et al.* [96] :

$$\forall xy[C(x, y) \rightarrow C(y, x)]$$

Notre opérateur graphique d'intersection ($e_1 \cap_? e_2$) semble donc vérifier les deux axiomes ($\forall e[e \cap_? e]$ et $\forall e_1 e_2 [e_1 \cap_? e_2 \Leftrightarrow e_2 \cap_? e_1]$) de la relation de connexion ($C(x, y)$) de Randell *et al.* [96] ($\forall x C(x, x)$ et $\forall xy[C(x, y) \rightarrow C(y, x)]$).

De ce fait, nous devrions pouvoir enrichir nos opérateurs graphiques en exprimant les opérateurs spatiaux de Randell *et al.* [96] dans notre formalisme. Cela offrirait la possibilité d'exprimer les opérateurs spatiaux de Randell *et al.* [96] en les exprimant à l'aide des variables graphiques et donc variables du code informatique.

De plus, nous pourrions utiliser les axiomes de Randell *et al.* [95] pour exprimer la position relative de deux corps par rapport à un point de vue donné. Cela nous permettrait de définir le fait qu'un élément graphique est à droite / gauche ou en haut / bas d'un autre élément graphique.

7.3.2 Couverture de l'analyse

Concernant l'opérateur d'égalité des formes, nous pensons qu'il faudrait ajouter à celui-ci la notion de constantes de formes. Ainsi, nous pourrions vérifier tout d'abord que le nœud relatif à l'élément graphique est bien du type qui nous intéresse. Nous pourrions ensuite comparer les différentes grandeurs caractéristiques de chacune de ces formes afin de vérifier que celles-ci sont égales. Cela nous permettrait notamment de séparer l'égalité des domaines (l'égalité des formes que nous avons définie actuellement) de l'égalité des formes indépendante de la position (coordonnées x et y) et des éventuelles transformations géométriques du type translation ou rotation.

Nous réfléchissons actuellement à une solution afin d'analyser le domaine d'un

texte. Pour ce faire, nous envisageons d'abstraire ce texte à son rectangle englobant. Les grandeurs caractéristiques de ce rectangle (longueur et largeur) dépendront alors du nombre de caractère, de la police ainsi que de la taille de la police.

Enfin, nous pourrions optimiser l'opérateur graphique d'inclusion des formes. L'opérateur d'inclusion est le suivant : $e_1 \subset_? e_2 \Leftrightarrow \neg d(e_1) \vee \neg d(e_2) \vee \mathcal{D}(e_1) \subset \mathcal{D}(e_2)$. Nous avons imaginé cet opérateur afin de mettre en avant le fait que si l'élément e_1 n'est pas affiché, il ne sera pas inclu dans e_2. Cependant, nous avons effectué le même principe avec l'affichage de e_2. Nous pensons qu'il serait plus intéressant d'assurer que si l'élément e_2 n'est pas affiché, alors l'élément e_1 ne doit pas être affiché non plus afin d'assurer son inclusion si celui-ci est affiché. Cela donnerait par exemple :

$$\boxed{e_1 \subset_? e_2 \Leftrightarrow \neg d(e_1) \vee \neg(d(e_1) \wedge \neg d(e_2)) \vee \mathcal{D}(e_1) \subset \mathcal{D}(e_2)}$$

7.3.3 Adressage de la perception utilisateur

Concernant la perception de l'utilisateur, il pourrait être intéressant de savoir pour un système critique si une alarme s'affiche au-dessus de tous les autres éléments graphiques de l'interface (notons que même si cette exigence est absente des exigences du TCAS, on la retrouve dans de nombreux autres systèmes informatiques interactifs). Pour ce cas, nous pourrions définir l'opérateur d'ordre d'affichage maximum qui représenterait le fait que la couche ($l(e_i)$) d'un élément graphique e_{ref} est la plus élevée de l'arbre XML ou des éléments graphiques affichés et affichables en même temps que cet élément :

$$\boxed{all <_{d,all} e_{ref} \Leftrightarrow \forall e[e <_d e_{ref}] \Leftrightarrow \forall e[l(e) < l(e_{ref})]}$$

Si nous nous intéressons à une mauvaise perceptibilité induite par le masquage d'un élément par un autre, nous pourrions créer un opérateur graphique représentant le masquage. Nous pourrions donc créer l'opérateur graphique $e_1 \odot_? e_2$ qui indique que e_1 est (partiellement) masqué par e_2. Ce masquage peut exister si e_1 et e_2 sont en intersection et si e_2 est au-dessus de e_1 ou si e_2 est en dessous de e_1 mais que leur couleur est identique. Nous aurions alors :

$$\boxed{e_1 \odot_? e_2 \Leftrightarrow e_1 \cap_? e_2 \wedge (e_1 <_d e_2 \vee (e_2 <_d e_1 \wedge e_1 =_c e_2))}$$

Pour prendre en compte la perception des couleurs, il serait intéressant de définir

autrement les constantes de couleur, voire l'opérateur d'égalité de couleur dans son ensemble, pour chaque composante (r, g et b). En effet, pour le moment nous nous intéressons à une égalité (ou inégalité) stricte pour chaque composante de couleur :

$$e_1 =_c e_2 \Leftrightarrow c_i(e_1) = c_i(e_2), \forall i \in [r, g, b]$$

Cependant, une couleur peut avoir plusieurs définitions dans le système RGB. Par exemple, le rouge ne sera pas forcément défini comme le rouge pur ($r = 255$, $g = 0$, $b = 0$) mais pourrait être défini comme le rouge cerise ($r = 187$, $g = 11$, $b = 11$). Il faut donc autoriser une certaine approximation sur chacune des composantes de la couleur afin de rendre l'opérateur d'égalité des couleurs plus permissive :

$$\boxed{e_1 =_c e_2 \Leftrightarrow c_i(e_1) = c_i(e_2) \pm val, \forall i \in [r, g, b]}$$

Il serait alors possible de déterminer une valeur *val* et de la fixer définitivement, à l'aide de travaux en ergonomie par exemple, ou alors de laisser la possibilité à l'utilisateur de la renseigner lors de l'utilisation de GPCheck.

7.3.4 Dépendance à Smala

Pour le moment, nous avons dédié GPCheck à l'analyse de code Smala et plus précisément de l'arbre XML qui correspond au graphe de scène enrichi d'une application Smala. Les éléments graphiques de Smala sont basés sur la norme SVG. Les composants de couplage (binding, assignment et connector) existent dans d'autres langages réactifs comme Java FX. Nous estimons qu'il est donc possible d'obtenir un graphe de scène enrichi, au format XML, d'un système informatique interactif implémenté dans un autre langage. De ce fait, il serait possible d'utiliser GPCheck pour analyser du code provenant d'autres langages que Smala.

Cependant, ce graphe de scène enrichi reste dépendant de la sémantique du langage de programmation utilisée. Aussi, il nous faudrait comparer les sémantiques de ces langages à celle de Smala afin d'adapter le graphe de scène enrichi résultant dans la sémantique de notre langage.

Afin de pouvoir utiliser notre processus de vérification avec un autre langage de programmation réactive, il faut :

- pouvoir traduire les éléments de la scène graphique dans le standard SVG (variables graphiques),

- posséder les composants de couplage, ou équivalents, de Smala (binding, connector, assignment)
- pouvoir obtenir un graphe de scène enrichi basé sur la même sémantique de parcours (d'abord en profondeur puis à droite)

7.3.5 Utilisabilité de GPCheck

Nous réfléchissons à un moyen simple de prendre en compte la disjonction qui existe dans certaines exigences graphiques. Nous avons déjà pensé dans l'algorithme et implémenté dans GPCheck l'opérateur logique OU. De ce fait, nous savons qu'il est possible de vérifier des disjonctions avec notre outil. Afin de vérifier des disjonctions de propriétés graphiques, nous pourrions ajouter le mot-clé `OR` au sein du fichier d'entrée dans lequel nous écrivons les exigences graphiques formalisées.

Concernant le problème de profondeur de parenthésage, spécifique à notre connaissance à MATLAB, plusieurs solutions s'offrent à nous. Nous pourrions dans un premier temps donner l'équation de sortie à un solveur autre que MATLAB qui autoriserait une profondeur de parenthésage supérieure à celle de MATLAB. Nous pourrions utiliser un simplificateur d'équations au sein de notre outil afin d'avoir une profondeur de parenthésage moins importante. De plus, notre résolution par MATLAB offre un résultat graphique par rapport aux plages des données d'entrée. Nous pourrions utiliser à la place un solveur symbolique qui donnerait alors les plages de valeurs autorisées d'un point de vue mathématique.

Enfin, GPCheck est encore au stade de prototype et tous les opérateurs graphiques ne sont pas encore pris en compte. Cependant, il serait intéressant pour nous en tant que développeurs de cet outil, de déterminer son efficacité et sa pertinence dans les processus de vérification. Un objectif que nous visons est donc d'évaluer GPCheck. Pour cela, nous devons trouver un cas d'étude (soit le TCAS, soit un nouveau système informatique interactif) ainsi qu'une liste d'exigences graphiques associées que nous savons vérifier avec GPCheck. Ensuite, nous proposerons à un panel d'utilisateurs de vérifier ces exigences sans GPCheck avec des méthodes de test ou autres méthodes à définir. Nous proposerons à un autre panel, formé à l'utilisation de l'outil, de faire de même avec GPCheck. Nous comparerons alors les résultats pour déterminer la pertinence de l'outil en l'état actuel des choses.

BIBLIOGRAPHIE

[1] Spem2.0. SPEM2.0 (2008). SPEM 2.0. http://www.omg.org/spec/SPEM/2.0/. Last access in january 2020., 2008.

[2] Ieee draft guide : Adoption of the project management institute (pmi) standard : A guide to the project management body of knowledge (pmbok guide)-2008 (4th edition). *IEEE P1490/D1, May 2011*, pages 1–505, 2011.

[3] Ed 143 - minimum operational performance standards for traffic alert and collision avoidance system ii (tcas ii), April 2013.

[4] John A Bargh. *The Four Horsemen of Automaticity : Awareness, Efficiency, Intention, and Control in Social Cognition.*, volume 2. 01 1994.

[5] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* 01 2005.

[6] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering.* 01 2010. doi:10.1017/CBO9781139195881.

[7] Yamine Aït-Ameur, Idir Aït-Sadoune, Mickael Baron, and Jean-Marc Mota. Vérification et validation formelles de systèmes interactifs fondées sur la preuve : application aux systèmes Multi-Modaux. *Journal d'Interaction Personne-Système*, 1(1) :1–30, September 2010. URL : https://hal.archives-ouvertes.fr/hal-00634186.

[8] MGA Ament, Anna Cox, Ann Blandford, and Duncan Brumby. Working memory load affects device-specific but not task-specific error rate. *CogSci 2010 : Proceedings of the Annual Conference of the Cognitive Science Society*, pages 91 – 96, 2010.

[9] Hugh Anderson and Gabriel Ciobanu. Markov abstractions for probabilistic pi-calculus. *Electronic Communications of the EASST*, 22, 01 2009.

[10] Myrtho Arapinis, Muffy Calder, Louise Denis, Michael Fisher, Philip Gray, Savas Konur, Alice Miller, Eike Ritter, Mark Ryan, Sven Schewe, Chris Unsworth, and Rehana Yasmin. Towards the verification of pervasive systems. *Electronic Communications of the EASST*, 22, 01 2009.

[11] Jos C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2) :131 – 146, 2005. Process Algebra. URL : `http://www.sciencedirect.com/science/article/pii/S0304397505000307`, `doi:https://doi.org/10.1016/j.tcs.2004.07.036`.

[12] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4), August 2013. URL : `https://doi.org/10.1145/2501654.2501666`, `doi:10.1145/2501654.2501666`.

[14] Richard Banach, Josph Razavi, Olivier Debicki, Nicolas Mareau, Suzanne Lesecq, and Julie Foucault. Application of formal methods in the inspex smart systems integration project. In *FMIS 2018*, 5 2018.

[15] Marco Antonio Barbosa, Luís Soares Barbosa, and José Creissac Campos. Towards a coordination model for interactive systems. *Electronic Notes in Theoretical Computer Science*, 183 :89 – 103, 2007. Proceedings of the First International Workshop on Formal Methods for Interactive Systems. URL : `http://www.sciencedirect.com/science/article/pii/S1571066107004306`, `doi:https://doi.org/10.1016/j.entcs.2007.01.063`.

[16] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system : An overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, January 2005.

[17] Ellen J. Bass, Karen M. Feigh, Elsa Gunter, and John Rushby. Formal modeling and analysis for interactive hybrid systems. 45, 01 2011.

[18] Michel Beaudouin-Lafon. Designing interaction, not interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 15–22, New York, NY, USA, 2004. ACM. URL : `http://doi.acm.org/10.1145/989863.989865`, `doi:10.1145/989863.989865`.

[19] Bernhard Beckert and Gerd Beuster. Guaranteeing consistency in text-based human-computer-interaction. 2007. Proceedings of the First International Workshop on Formal Methods for Interactive Systems.

[20] Valentin Becquet. *Conception d'une représentation graphique des gestes numériques pour le cockpit tactile fondée sur les dimensions participant à la conscience mutuelle entre les pilotes*. Theses, Institut Supérieur de l'Aéronautique et de l'Espace, December 2019. URL : `https://hal.archives-ouvertes.fr/tel-02492163`.

[21] Valentin Becquet, Catherine Letondal, Jean-Luc Vinot, and Sylvain Pauchet. How do Gestures Matter for Mutual Awareness in Cockpits ? Disclosing Interactions through Graphical Representations. In *DIS Designing Interactive Systems 2019*, DIS '19 Proceedings of the 2019 on Designing Interactive Systems Conference, pages 593–605, San Diego - California, United States, June 2019. ACM Press. URL : `https://hal-enac.archives-ouvertes.fr/hal-02134640`.

[22] Pascal Béger, Valentin Becquet, Sébastien Leriche, and Daniel Prun. Contribution à la formalisation des propriétés graphiques des systèmes interactifs pour la validation automatique. In *Afadl 2019, 18èmes journées Approches Formelles dans l'Assistance au Developpement de Logiciels*, Toulouse, France, June 2019. URL : `https://hal-enac.archives-ouvertes.fr/hal-02165690`.

[23] Pascal Béger, Sébastien Leriche, and Daniel Prun. A Survey of Papers from Formal Methods for Interactive Systems (FMIS) Workshops. In *FMIS 2019 : 8th Formal Methods for Interactive Systems workshop*, Porto, Portugal, October 2019. URL : `https://hal-enac.archives-ouvertes.fr/hal-02364845`.

[24] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. `doi:10.1007/978-3-540-30080-9\_7`.

[25] François Bérard and Renaud Blanch. Two touch system latency estimators : High accuracy and low overhead. In *Proceedings of the 2013 ACM International Conference on Interactive Tabletops and Surfaces*, ITS '13, page 241–250, New York, NY, USA, 2013. Association for Computing Machinery. URL : `https://doi.org/10.1145/2512349.2512796`, `doi:10.1145/2512349.2512796`.

[26] Jacques Bertin and Marc Barbut. *Sémiologie graphique : les diagrammes, les réseaux, les cartes*. Gauthier Villars, 1973. URL : https://books.google.fr/books?id=F4weAAAAMAAJ.

[27] Colm Bhandal, Melanie Bouroche, and Arthur Hughes. A process algebraic description of a temporal wireless network protocol. 45, 01 2011.

[28] Samit Bhattacharya, Anupam Basu, Debasis Samanta, Souvik Bhattacherjee, and Animesh Srivatava. Some issues in modeling the performance of soft keyboards with scanning. 2007. Proceedings of the First International Workshop on Formal Methods for Interactive Systems.

[29] Ann Blandford and Dominic Furniss. Dicot : A methodology for applying distributed cognition to the design of teamworking systems. In Stephen W. Gilroy and Michael D. Harrison, editors, *Interactive Systems. Design, Specification, and Verification*, pages 26–38, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[30] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Formalization of Real Analysis : A Survey of Proof Assistants and Libraries. *Mathematical Structures in Computer Science*, 26(7) :1196–1233, October 2016. URL : https://hal.inria.fr/hal-00806920, doi:10.1017/S0960129514000437.

[31] Jean-François Bonnefon, Dominique Longin, and Manh Hung Nguyen. A logical framework for trust-related emotions. *Electronic Communications of the EASST*, 22, 01 2009.

[32] Jullien Bouchet, Laya Madani, Laurence Nigay, Catherine Oriat, and Ioannis Parissis. Formal testing of multimodal interactive systems. In Jan Gulliksen, Morton Borup Harning, Philippe Palanque, Gerrit C. van der Veer, and Janet Wesson, editors, *Engineering Interactive Systems*, pages 36–52, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[33] Sara Bouzit, Gaëlle Calvary, Denis Chêne, and Jean Vanderdonckt. A design space for engineering graphical adaptive menus. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '16, page 239–244, New York, NY, USA, 2016. Association for Computing Machinery. URL : https://doi.org/10.1145/2933242.2935874, doi:10.1145/2933242.2935874.

[34] Judy Bowen and Annika Hinze. Supporting mobile application development with model-driven emulation. 45, 01 2011.

[35] Judy Bowen and Steve Reeves. Formal models for informal gui designs. *Electronic Notes in Theoretical Computer Science*, 183 :57 – 72, 2007. Proceedings of the First International Workshop on Formal Methods for Interactive Systems. URL : http://www.sciencedirect.com/science/article/pii/S1571066107004288, doi:https://doi.org/10.1016/j.entcs.2007.01.061.

[36] Judy Bowen and Steve Reeves. Refinement for user interface designs. *Electronic Notes in Theoretical Computer Science*, 208 :5 – 22, 2008. Proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems. URL : http://www.sciencedirect.com/science/article/pii/S1571066108002090, doi:https://doi.org/10.1016/j.entcs.2008.03.104.

[37] Judy Bowen and Steve Reeves. Ui-design driven model-based testing. *Electronic Communications of the EASST*, 22, 01 2009. URL : http://dx.doi.org/10.14279/tuj.eceasst.22.314.

[38] Pascal Béger, Sébastien Leriche, and Daniel Prun. Vers la certification de programmes interactifs Djnn. In *Afadl 2018, 17èmes journées Approches Formelles dans l'Assistance au Developpement de Logiciels* , Grenoble, France, June 2018. URL : https://hal-enac.archives-ouvertes.fr/hal-01815208.

[39] Béatrice Bérard, M. Bidoit, Alain Finkel, F. Laroussinie, A. Petit, Laure Petrucci, and Ph Schnoebelen. *Systems and Software Verification : Model-Checking Techniques and Tools*. 01 2001.

[40] Muffy Calder, Phil Gray, and Chris Unsworth. Tightly coupled verification of pervasive systems. *Electronic Communications of the EASST*, 22, 01 2009.

[41] José Campos and Michael Harrison. Modelling and analysing the interactive behaviour of an infusion pump. 45, 01 2011.

[42] Dominique Cansell, J. Paul Gibson, and Dominique Méry. Refinement : A constructive approach to formal software design for a secure e-voting interface. *Electronic Notes in Theoretical Computer Science*, 183 :39 – 55, 2007. Proceedings of the First International Workshop on Formal Methods for Interactive Systems. URL : http://www.sciencedirect.com/science/article/pii/S1571066107004276, doi:https://doi.org/10.1016/j.entcs.2007.01.060.

[43] Ula Cartwright-Finch and Nilli Lavie. The role of perceptual load in inattentional blindness. *Cognition*, 102(3) :321 – 340, 2007. URL : http://www.sciencedirect.com/science/article/pii/S0010027706000205, doi:https://doi.org/10.1016/j.cognition.2006.01.002.

[44] Antonio Cerone. Closure and attention activation in human automatic behaviour : A framework for the formal analysis of interactive systems. 45, 01 2011.

[45] Antonio Cerone. Towards a cognitive architecture for the formal analysis of human behaviour and learning. In Manuel Mazzara, Iulian Ober, and Gwen Salaün, editors, *Software Technologies : Applications and Foundations*, pages 216–232, Cham, 2018. Springer International Publishing.

[46] Antonio Cerone and Norzima Elbegbayan. Model-checking driven design of interactive systems. *Electronic Notes in Theoretical Computer Science*, 183 :3 – 20, 2007. Proceedings of the First International Workshop on Formal Methods for Interactive Systems. URL : http://www.sciencedirect.com/science/article/pii/S1571066107004252, doi:https://doi.org/10.1016/j.entcs.2007.01.058.

[47] Antonio Cerone and Yishi Zhao. Stochastic modelling and analysis of driver behaviour. *ECEASST*, 69, 2013.

[48] Stéphane Chatty, Mathieu Magnaudet, and Daniel Prun. Verification of properties of interactive components from their executable code. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '15, page 276–285, New York, NY, USA, 2015. Association for Computing Machinery. URL : https://doi.org/10.1145/2774225.2774848, doi:10.1145/2774225.2774848.

[49] Rance Cleaveland, Tan Li, and Steve Sims. The concurrency workbench of the new century. User's manual, SUNY at Stony Brook, Stony Brooke, NY, USA, URL : http://www.cs.sunysb.edu/~cwb, 2000.

[50] Patrick Cousot and Radhia Cousot. Abstract interpretation : Past, present and future. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery. URL : https://doi.org/10.1145/2603088.2603165, doi:10.1145/2603088.2603165.

[51] Joëlle Coutaz, Laurence Nigay, Daniel Salber, Ann Blandford, Jon May, and Richard M. Young. *Four Easy Pieces for Assessing the Usability of Multimodal Interaction : The Care Properties*, pages 115–120. Springer US, Boston, MA, 1995. URL : `https://doi.org/10.1007/978-1-5041-2896-4_19`, `doi:10.1007/978-1-5041-2896-4_19`.

[52] Brad J. Cox and Andrew Novobilski. *Object-Oriented Programming ; An Evolutionary Approach*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 1991.

[53] Bruno d'Ausbourg. Using model checking for the automatic validation of user interfaces systems. In Panos Markopoulos and Peter Johnson, editors, *Design, Specification and Verification of Interactive Systems '98*, pages 242–260, Vienna, 1998. Springer Vienna.

[54] René David and Hassane Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[55] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. Sal 2. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 496–500, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[56] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, USA, 1st edition, 1997.

[57] Anke Dittmar, Toralf Hübner, and Peter Forbrig. Hops : A prototypical specification tool for interactive systems. In T. C. Nicholas Graham and Philippe Palanque, editors, *Interactive Systems. Design, Specification, and Verification*, pages 58–71, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[58] Anke Dittmar and Reik Schachtschneider. Lightweight interaction modeling in evolutionary prototyping. *ECEASST*, 69, 2013.

[59] Alan Dix, Masitah Ghazali, and Devina Ramduny-Ellis. Modelling devices for natural interaction. *Electronic Notes in Theoretical Computer Science*, 208 :23 – 40, 2008. Proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems. URL : `http://www.sciencedirect.com/science/article/pii/S1571066108002107`, `doi:https://doi.org/10.1016/j.entcs.2008.03.105`.

[60] Jane E. Raymond, Kimron Shapiro, and Karen Arnell. Temporary suppression of visual processing in an rsvp task : An attentional blink ? *Journal of*

experimental psychology. Human perception and performance, 18 :849–60, 09 1992. doi:10.1037/0096-1523.18.3.849.

[61] Romain Geniet and Neeraj Kumar Singh. Refinement based formal development of human-machine interface. In Manuel Mazzara, Iulian Ober, and Gwen Salaün, editors, *Software Technologies : Applications and Foundations*, pages 240–256, Cham, 2018. Springer International Publishing.

[62] Panagiotis Germanakos and Marios Belk. *Personalization Categories and Adaptation Technologies*, pages 103–135. 02 2016. doi:10.1007/978-3-319-28050-9_4.

[63] Doug Goldson, Greg Reeve, and Steve Reeves. μ-chart-based specification and refinement. In *Proceedings of the 4th International Conference on Formal Engineering Methods : Formal Methods and Software Engineering*, ICFEM '02, pages 323–334, Berlin, Heidelberg, 2002. Springer-Verlag. URL : http://dl.acm.org/citation.cfm?id=646272.685799.

[64] Valentin Goranko and Antony Galton. Temporal logic. *The Stanford Encyclopedia of Philosophy* (Winter 2015 Edition), Edward N. Zalta (ed.), URL : https://plato.stanford.edu/archives/win2015/entries/logic-temporal/, 2015.

[65] Anjana Gosain and Ganga Sharma. Static analysis : A survey of techniques and tools. In Durbadal Mandal, Rajib Kar, Swagatam Das, and Bijaya Ketan Panigrahi, editors, *Intelligent Computing and Applications*, pages 581–591, New Delhi, 2015. Springer India.

[66] Michael D. Harrison, Christian Kray, and José Creissac Campos. Exploring an option space to engineer a ubiquitous computing system. *Electronic Notes in Theoretical Computer Science*, 208 :41 – 55, 2008. Proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems. URL : http://www.sciencedirect.com/science/article/pii/S1571066108002119, doi:https://doi.org/10.1016/j.entcs.2008.03.106.

[67] Michael D. Harrison, Paolo Masci, Jose Creissac Campos, and Paul Curzon. Automated theorem proving for the systematic analysis of an infusion pump. *ECEASST*, 69, 2013.

[68] Michael D. Harrison, Paolo Masci, and José Creissac Campos. Formal modelling as a component of user centred design. In Manuel Mazzara, Iulian

Ober, and Gwen Salaün, editors, *Software Technologies : Applications and Foundations*, pages 274–289, Cham, 2018. Springer International Publishing.

[69] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, New York, NY, USA, 1996.

[70] Louis Hjelmslev. *Prolegomena to a theory of language*. Number 7 in Prolegomena to a Theory of Language. University of Wisconsin Press, 1961. URL : `https://books.google.fr/books?id=HrtrAAAAIAAJ`.

[71] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, October 1969. URL : `http://doi.acm.org/10.1145/363235.363259`, `doi:10.1145/363235.363259`.

[72] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[73] Gerard Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.

[74] Maya Hristakeva and RadhaKrishna Vuppala. A survey of object oriented programming languages. Technical report, University of California, Santa Cruz, 2009. URL : `https://users.soe.ucsc.edu/~vrk/Reports/oopssurvey.pdf`.

[75] Huayi Huang, Rimvydas Rukšėnas, Maartje Ament, Paul Curzon, Anna Cox, Ann Blandford, and Duncan Brumby. Capturing the distinction between task and device errors in a formal model of user behaviour. 45, 01 2011.

[76] ISO-8807 :1989. Information processing systems - open systems interconnection - lotos - a formal description technique based on the temporal ordering of observational behaviour, 1989.

[77] Brit Susan Jensen, Mikael B. Skov, and Nissanthen Thiruravichandran. Studying driver attention and behaviour for three configurations of gps navigation in real traffic driving. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, page 1271–1280, New York, NY, USA, 2010. Association for Computing Machinery. URL : `https://doi.org/10.1145/1753326.1753517`, `doi:10.1145/1753326.1753517`.

[78] Chris W. Johnson. Using assurance cases and boolean logic driven markov processes to formalise cyber security concerns for safety-critical interaction with global navigation satellite systems. 45, 01 2011.

[79] Nadjet Kamel and Yamine Ait-Ameur. Modèle formel général pour le traitement d'interactions multimodales. In *IHM 04*, pages 219–222, Namure, Belgique, 03 2004.

[80] Christian Kray, Gerd Kortuem, and Antonio Krüger. Adaptive navigation support with public displays. In *Proceedings of the 10th International Conference on Intelligent User Interfaces*, IUI '05, pages 326–328, New York, NY, USA, 2005. ACM. URL : http://doi.acm.org/10.1145/1040830.1040916, doi:10.1145/1040830.1040916.

[81] Sébastien Leriche, Stéphane Conversy, Célia Picard, Daniel Prun, and Mathieu Magnaudet. Towards handling latency in interactive software. In Manuel Mazzara, Iulian Ober, and Gwen Salaün, editors, *Software Technologies : Applications and Foundations*, pages 233–239, Cham, 2018. Springer International Publishing.

[82] Dan Lockton, David Harrison, and Neville Stanton. The design with intent method : A design tool for influencing user behaviour. *Applied ergonomics*, 41 :382–92, 10 2009. doi:10.1016/j.apergo.2009.09.001.

[83] Mathieu Magnaudet, Stéphane Chatty, Stéphane Conversy, Sébastien Leriche, Célia Picard, and Daniel Prun. Djnn/Smala : A Conceptual Framework and a Language for Interaction-Oriented Programming. *Proceedings of the ACM on Human-Computer Interaction* , 2(EICS) :1 – 27, June 2018. URL : https://hal-enac.archives-ouvertes.fr/hal-01815222, doi:10.1145/3229094.

[84] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. 01 2010.

[85] Pablo Masci, Paul Curzon, Ann Blandford, and Dominic Furniss. Modelling distributed cognition systems in pvs. 45, 01 2011.

[86] Paolo Masci, Rimvydas Rukšėnas, Patrick Oladimeji, Abigail Cauchi, Andy Gimblett, Yunqiu Li, Paul Curzon, and Harold Thimbleby. On formalising interactive number entry on infusion pumps. 45, 01 2011.

[87] Thomas Mirlacher, Philippe Palanque, and Regina Bernhaupt. Engineering animations in user interfaces. *EICS'12 - Proceedings of the 2012 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 06 2012. doi:10.1145/2305484.2305504.

[88] David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. Icos : A model-based user interface description technique dedicated to inter-

active systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact.*, 16(4) :18 :1–18 :56, November 2009. URL : `http://doi.acm.org/10.1145/1614390.1614393`, `doi:10.1145/1614390.1614393`.

[89] Oscar Nierstrasz. Object-oriented concepts, databases, and applications. chapter A Survey of Object-oriented Concepts, pages 3–21. ACM, New York, NY, USA, 1989. URL : `http://doi.acm.org/10.1145/63320.66468`, `doi:10.1145/63320.66468`.

[90] G. Norman, C. Palamidessi, D. Parker, and P. Wu. Model checking the probabilistic π-calculus. In *Proc. 4th International Conference on Quantitative Evaluation of Systems (QEST'07)*, pages 169–178. IEEE Computer Society, 2007.

[91] Patrick Oladimeji, Paolo Masci, Paul Curzon, and Harold Thimbleby. Pvsioweb : a tool for rapid prototyping device user interfaces in pvs. *ECEASST*, 69, 2013.

[92] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3) :455–495, July 1982. URL : `http://doi.acm.org/10.1145/357172.357178`, `doi:10.1145/357172.357178`.

[93] Peter Puschner and Alan Burns. A review of worst-case execution-time analyses. *Real-time Systems - RTS*, 01 1999.

[94] David Randell and Anthony Cohn. Modelling topological and metrical properties in physical processes. pages 357–368, 01 1989.

[95] David Randell, Mark Witkowski, and Murray Shanahan. From images to bodies : Modelling and exploiting spatial occlusion and motion parallax. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'01, page 57–63, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[96] David A. Randell, Zhan Cui, and Anthony G. Cohn. A spatial logic based on regions and connection. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, KR'92, page 165–176, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[97] Wolfgang Reisig. *Understanding Petri Nets : Modeling Techniques, Analysis Methods, Case Studies*. Springer Publishing Company, Incorporated, 2013.

[98] R. Rukšėnas, P. Curzon, and A. Blandford. Detecting cognitive causes of confidentiality leaks. *Electronic Notes in Theoretical Computer Science*, 183 :21 – 38, 2007. Proceedings of the First International Workshop on Formal Methods for Interactive Systems. URL : http://www.sciencedirect.com/science/article/pii/S1571066107004264, doi:https://doi.org/10.1016/j.entcs.2007.01.059.

[99] Rimvydas Rukšėnas, Jonathan Back, Paul Curzon, and Ann Blandford. Formal modelling of salience and cognitive load. *Electronic Notes in Theoretical Computer Science*, 208 :57 – 75, 2008. Proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems. URL : http://www.sciencedirect.com/science/article/pii/S1571066108002120, doi:https://doi.org/10.1016/j.entcs.2008.03.107.

[100] Rimvydas Rukšėnas and Paul Curzon. Abstract models and cognitive mismatch in formal verification. 45, 01 2011.

[101] Rimvydas Rukšėnas, Paolo Masci, Michael D. Harrison, and Paul Curzon. Developing and verifying user interface requirements for infusion pumps : A refinement approach. *ECEASST*, 69, 2013.

[102] Mark D. Ryan and Ben Smyth. Applied pi calculus. In Véronique Cortier and Steve Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, chapter 6. IOS Press, 2011. doi:10.3233/978-1-60750-714-7-112.

[103] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1) :5–19, September 2006. URL : https://doi.org/10.1109/JSAC.2002.806121, doi:10.1109/JSAC.2002.806121.

[104] Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 796–807, New York, NY, USA, 2016. ACM. URL : http://doi.acm.org/10.1145/2884781.2884815, doi:10.1145/2884781.2884815.

[105] Donald Sannella and Martin Wirsing. Specification languages. *E. Astesiano, H.J. Kreowski, B. Krieg-Brückner (eds.) : Algebraic Foundation of Systems Specification. IFIP State-of-the-Art Reports, Berlin : Springer, 1999, 243-272.*, 07 1999. doi:10.1007/978-3-642-59851-7_8.

[106] RTCA (Firm). SC-205 and EUROCAE (Agency). Working Group 71. Rtca/do-178c software considerations in airborne systems and equipment certification, December 2011.

[107] RTCA (Firm). SC-205 and EUROCAE (Agency). Working Group 71. Rtca/do-333 formal methods supplement to do-178c and do-278a, December 2011.

[108] Natarajan Shankar. Pvs : Combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, pages 257–264, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[109] Julien Signoles, Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, and Boris Yakobowski. Frama-c : a software analysis perspective. volume 27, 10 2012. doi:10.1007/s00165-014-0326-7.

[110] José-Luis Silva, Camille Fayollas, Arnaud Hamon, Philippe Palanque, Célia Martiinie, and Eric Barboni. Analysis of wimp and post wimp interactive systems based on formal specification. *ECEASST*, 69, 2013.

[111] Ankit Singh. Rtca do-178b (eurocae ed-12b), 04 2011. doi:10.13140/RG.2.2.14788.42886.

[112] Daniel Sinnig, Patrice Chalin, and Ferhat Khendek. Towards a common semantic foundation for use cases and task models. *Electronic Notes in Theoretical Computer Science*, 183 :73 – 88, 2007. Proceedings of the First International Workshop on Formal Methods for Interactive Systems.

[113] R. William Soukoreff and I. Scott Mackenzie. Theoretical upper and lower bounds on typing speed using a stylus and a soft keyboard. *Behaviour & Information Technology*, 14(6) :370–379, 1995. doi:10.1080/01449299508914656.

[114] Michael J. Spivey. *The Z Notation : A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[115] I.O.F. Standardization. *ISO 9241-11 : Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs) : Part 11 : Guidance on Usability*. 1998. URL : https://books.google.fr/books?id=TzXYZwEACAAJ.

[116] Li Su, Howard Bowman, and Philip Barnard. Performance of reactive interfaces in stimulus rich environments, applying formal methods and cognitive frameworks. *Electronic Notes in Theoretical Computer Science*, 208 :95 – 111, 2008. Proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems. URL : http://www.sciencedirect.com/science/article/pii/S1571066108002144, doi:https://doi.org/10.1016/j.entcs.2008.03.109.

[117] Harold Thimbleby and Andy Gimblett. Dependable keyed data entry for interactive systems. 45, 01 2011.

[118] Benjamin Tissoires and Stéphane Conversy. Hayaku : Designing and optimizing finely tuned and portable interactive graphics with a graphical compiler. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, page 117–126, New York, NY, USA, 2011. Association for Computing Machinery. URL : https://doi.org/10.1145/1996461.1996505, doi:10.1145/1996461.1996505.

[119] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, USA, 1986.

[120] Jessica Turner, Judy Bowen, and Steve Reeves. *Using Abstraction with Interaction Sequences for Interactive System Modelling : STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*, pages 257–273. 06 2018. doi:10.1007/978-3-030-04771-9_20.

[121] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 1st edition, 2004.

[122] Jean-Luc Vinot, Catherine Letondal, Rémi Lesbordes, Stéphane Chatty, Stéphane Conversy, and Christophe Hurter. Tangible augmented reality for air traffic control. *Interactions*, 21(4) :54–57, July 2014. URL : https://doi.org/10.1145/2627598, doi:10.1145/2627598.

[123] Colin Ware. *Information Visualization : Perception for Design*. Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann, Amsterdam, 3 edition, 2012. URL : http://www.sciencedirect.com/science/book/9780123814647.

[124] Michael Westergaard. A game-theoretic approach to behavioural visualisation. *Electronic Notes in Theoretical Computer Science*, 208 :113 – 129, 2008. Proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems. URL : http://www.sciencedirect.com/science/article/pii/S1571066108002156, doi:https://doi.org/10.1016/j.entcs.2008.03.110.

[125] Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg, 2005.

[126] Ed Williams. Airborne collision avoidance system. In *Proceedings of the 9th Australian Workshop on Safety Critical Systems and Software - Volume*

47, SCS '04, pages 97–110, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc. URL : http://dl.acm.org/citation.cfm?id=1082338.1082349.

[127] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods : Practice and experience. *ACM Comput. Surv.*, 41(4), October 2009. URL : https://doi.org/10.1145/1592434.1592436, doi:10.1145/1592434.1592436.

[128] Takuto Yanagida, Hidetoshi Nonaka, and Masahito Kurihara. Personalizing graphical user interfaces on flexible widget layout. In *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '09, page 255–264, New York, NY, USA, 2009. Association for Computing Machinery. URL : https://doi.org/10.1145/1570433.1570481, doi:10.1145/1570433.1570481.

TABLE DES FIGURES

2.1 Paradigmes de programmation usuels 12
2.2 Illustration de l'exécution d'un programme impératif 13
2.3 Disparition des variables en langage fonctionnel 13
2.4 Appels de méthodes entre objets 14
2.5 Exécution d'un programme réactif dirigé par le flot de données . . . 15
2.6 Classes de propriétés liées à l'interaction 17
2.7 Propriétés de la classe comportement de l'utilisateur 18
2.8 Propriétés de la classe principes cognitifs 21
2.9 Propriétés de la classe interface humain-machine 22
2.10 Propriétés de la classe sécurité 26
2.11 Nomenclature des formalismes 29
2.12 Modèle B événementiel d'une horloge et son raffinement [7] 31
2.13 Réseau de Petri de la gestion d'un espace de stockage 33
2.14 Représentation de la logique temporelle linéaire 34
2.15 Représentation de la logique du temps arborescent 34

4.1 Exemple d'application interactive de base en Smala 61
4.2 Graphe de scène enrichi de l'exemple Smala (version allégée) 62
4.3 Scène graphique des exemples de codes Smala 4.2 et 4.3 63
4.4 Graphe de scène enrichi des exemples de codes Smala 4.2 et 4.3 . . 64
4.5 TCAS intégré à l'IVSI et réalisé en Smala 65
4.6 Phase initiale de l'approche frontale 68
4.7 Premier RA : *Descend. Descend.* 69
4.8 Indication de la vitesse verticale du trafic : descente 69
4.9 Deuxième RA : *Climb. Climb.* 70

4.10 Risque de collision écarté 70

5.1 Légende des diagrammes de processus 76
5.2 Vue d'ensemble du processus de vérification des exigences graphiques 77
5.3 Formalisation des exigences graphiques 80
5.4 Prétraitement du code 80
5.5 Génération des données d'entrée Smala 81
5.6 Génération des données d'entrée fgp 83
5.7 Génération des données d'entrée finales 84
5.8 Vérification des propriétés graphiques 85
5.9 Exploitation du résultat de la vérification 86
5.10 Résultat de la vérification de $E_{3,couleur}$ avec $level = 4$ et $current\_range = 6$ 87

6.1 Résultats des vérifications de $E_{1,white}$ (gauche) et $E_{1,cyan}$ (droite) avec $level = 1$ et $current\_range = 6$ 127
6.2 Résultat de la vérification de E_3 avec $level = 4$ et $current\_range = 6$ 131
6.3 Affichage du TCAS intégré à l'IVSI ne vérifiant pas l'exigence E_3 . 131
6.4 Résultats des vérifications de $E_{4,yellow}$ (gauche) et $E_{4,amber}$ (droite) avec $level = 2$ et $current\_range = 6$ 133
6.5 Affichage du TCAS intégré à l'IVSI ne vérifiant pas l'exigence E_4 . 133
6.6 Résultats des vérifications de $E_{5,cyan}$ (gauche) et $E_{5,white}$ (droite) avec $level = 1$ et $current\_range = 6$ 135
6.7 Affichage du TCAS intégré à l'IVSI ne vérifiant pas l'exigence E_5 . 136
6.8 Résultats des vérifications de $E_{6,cyan}$ (gauche) et $E_{6,white}$ (droite) avec $level = 1$ et $current\_range = 6$ 137
6.9 Affichage du TCAS intégré à l'IVSI ne vérifiant pas l'exigence E_6 . 138
6.10 Résultat de la vérification de $E_{7,threat}$ avec $level = 4$ et $current\_range = 6$ 140
6.11 Résultat de la vérification de E_8 avec $level = 4$ et $current\_range = 6$ 141

7.1 Classification des propriétés liées à l'interaction 145

LISTE DES TABLEAUX

2.1 Synthèse des particularités des paradigmes de programmation . . . 16

3.1 Études des propriétés relatives au **comportement de l'utilisateur** 40
3.2 Études des propriétés relatives aux **principes cognitifs** 42
3.3 Études des propriétés relatives aux **interfaces humain-machine** . 43
3.4 Études des propriétés relatives à la **sécurité** 44
3.5 Études relatives à la **modélisation** par l'**algèbre de processus** . 46
3.6 Études relatives à la **modélisation** par les **langages de spécification** 47
3.7 Études relatives à la **modélisation** par les **processus de raffinement** 47
3.8 Études relatives à la **modélisation** par les **systèmes de transition d'états** et la **logique temporelle** 48
3.9 Études relatives à la **modélisation** par d'**autres** formalismes . . . 49
3.10 Étude des méthodes formelles appliquées aux systèmes interactifs . 51

4.1 Données d'entrée de l'interface du TCAS intégré à l'IVSI 67
4.2 Liste des exigences graphiques du TCAS intégré à l'IVSI 72

5.1 Opérateurs graphiques formels de base 91

6.1 Données d'entrée de l'interface du TCAS intégré à l'IVSI 122
6.2 Liste des exigences graphiques du TCAS intégré à l'IVSI 124

7.1 Synthèse des méthodes formelles appliquées aux systèmes interactifs (étude FMIS) 146
7.2 Opérateurs graphiques formels 147
7.3 Opérateurs graphiques formels (suite) 148

7.4 Liste des exigences graphiques du TCAS intégré à l'IVSI 150