Chapitre 3

Robustesse logicielle par analyse statique vis à vis de valeurs dysfonctionnelles

3.1 Introduction

Ce chapitre décrit comment démontrer la robustesse d'un logiciel vis-à-vis de valeurs dysfonctionnelles. Nous utilisons pour cela un outil d'analyse statique basé sur l'interprétation abstraite.

Notre approche est originale sur au moins deux points:

(1) Elle introduit l'utilisation d'une technique formelle non pas en phase de spécification, mais en phase de développement logiciel. La norme [CEN 01a] recommande l'utilisation des techniques formelles (A.2 point 1 page 48, A.4 point 1 page 50) telles que B, Z lors de la conception et la spécification, et l'analyse statique lors de la vérification du logiciel. Nous proposons de mettre en œuvre l'analyse statique de programme au cours du développement.

(2) Elle met en oeuvre l'analyse statique d'une façon non immédiate. L'analyse statique est en général utilisée pour détecter les erreurs ciblées : erreurs d'exécution, erreurs mémoire ou erreurs numériques. Nous proposons d'utiliser un outil d'analyse statique pour vérifier la cohérence entre les domaines fonctionnels spécifiés et le code source d'un logiciel, mais aussi pour calculer les domaines de valeurs des entrées.

Chapitre rédigé par Christèle FAURE, Jean-Louis BOULANGER et Samy AÏT KACI.

Section 3.2 nous positionnons notre approche vis-à-vis des référentiels associés aux systèmes critiques. Section 3.3 nous élaborons la méthode de preuve de robustesse logicielle que nous décrivons section 3.4. La section 3.5 explique comment utiliser l'analyse statique de programme pour automatiser une partie de notre méthode : calculer le « contrôle requis ». Section 3.6 nous présentons l'application de la méthode de vérification de robustesse sur le logiciel PING. Nous donnons des perspectives d'extension de la méthode proposée section 3..7, et en concluons section 3.8.

3.2. Contexte normatif

Dans les systèmes critiques (transport aérien, transport ferroviaire, centrale nucléaire), les défaillances peuvent mettre en cause la vie d'une ou plusieurs personnes et sont donc contraires à la sécurité (au sens "safety"). Pour cette classe de systèmes, les normes exigent la démonstration de l'absence de défaillances. Leur conception est donc soumise au respect de référentiels techniques (normes, documents métier, état de l'art) très strict.

Les systèmes électriques/électroniques sont utilisés depuis des années pour exécuter des fonctions liées à la sécurité dans la plupart des secteurs industriels. La norme IEC 61508 [IEC 98] présente une approche générique de toutes les activités liées au cycle de vie de sécurité de systèmes électriques/électroniques/électroniques programmables (E/E/PES) qui sont utilisés pour réaliser des fonctions de sécurité.

Dans la plupart des cas, la sécurité est obtenue par l'addition de plusieurs systèmes basés sur des technologies diverses (mécanique, hydraulique, pneumatique, électrique, électronique, électronique programmable). La stratégie de sécurité doit prendre en compte tous les éléments contribuant à la sécurité. Ainsi la norme IEC 61508 [IEC 98] fournit un cadre d'analyse qui s'applique à des systèmes relatifs à la sécurité basés sur d'autres technologies (mécanique, hydraulique, etc.), puis traite spécifiquement des systèmes E/E/PES.

Du fait de la grande variété des applications électriques/électroniques/ programmables et des degrés de complexité très divers, la nature exacte des mesures de sécurité à mettre en œuvre dépend de facteurs propres à l'application ; c'est pourquoi dans la norme IEC 61508 [IEC 98] il n'y a pas de règle générale mais des préconisations concernant les méthodes d'analyse à mettre en œuvre.

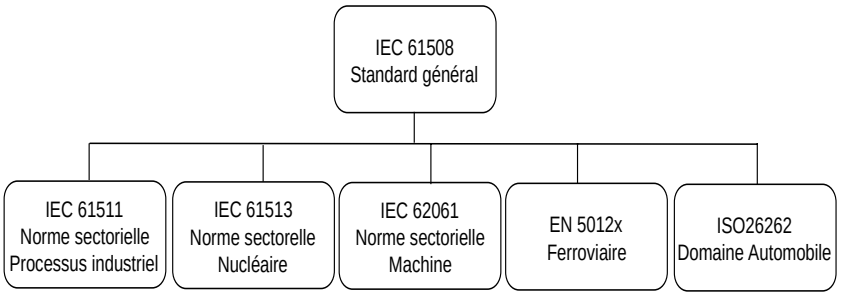

Figure 3.1. *IEC 61508 et les déclinaisons*[1]

Dans les normes, il existe des échelles qui permettent de définir le niveau de criticité de chaque système. Dans le domaine des systèmes complexes à base de composants électroniques et/ou programmés, la norme CEI 61508 [IEC 98] définit la notion de SIL (Safety Integrated Level). Le niveau SIL permet de quantifier le niveau de sécurité à atteindre pour un système et prend cinq valeurs: 0 (pas de danger, destruction de matériel), 1 (blessure légère), 2 (blessure grave), 3 (mort d'une personne) ou 4 (mort de plusieurs personnes).

Comme le montre la Figure 3.1, le référentiel ferroviaire CENELEC EN 5012x est une déclinaison de la norme générique IEC 61508 [IEC 98] qui tient compte des spécificités du domaine ferroviaire et des expériences réussies (SACEM, TVM, SAET-METEOR, etc.).

Le domaine du transport ferroviaire est donc principalement régi par trois normes dérivées de la CEI 61508 [IEC 98] qui couvrent différents aspects de la sécurité d'un système:

– la norme CENELEC EN 50126 [CEN 00] décrit les méthodes à mettre en œuvre pour spécifier et démontrer la fiabilité, disponibilité, maintenabilité et sécurité (FMDS) ;

– la norme CENELEC EN 50128 [CEN 01a] décrit les actions à entreprendre pour démontrer la sécurité des logiciels ;

– la norme CENELEC EN 50129 [CEN 03] décrit la structure du dossier de sécurité.

1. La norme ISO 26262 [ISO 09] n’est pas encore disponible, mais les industriels du domaine de l’automobile sont actuellement en train de préparer sa mise en œuvre comme le montre le chapitre 9 de [BOU 09]. A noter aussi que la norme IEC 61513 n’est pas vraiment rattachable à la norme IEC 61508 si on considère l’histoire des normes dans le domaine du nucléaire.

La Figure 3.2 présente le niveau couvert par chacune de ces normes ferroviaires : niveau système, sous-système.

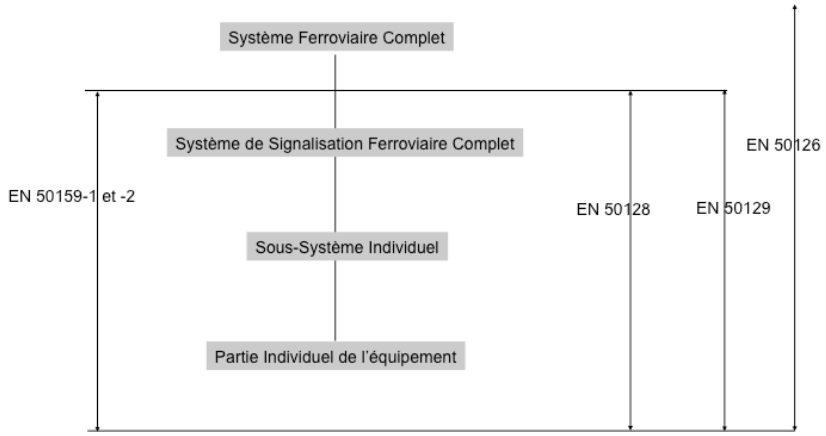

Figure 3.2. *Normes applicables aux systèmes ferroviaires*

Le développement des logiciels est dépendant d'un niveau de criticité propre SSIL (Software SIL) qui varie: du niveau 0 (pas de danger, aucun impact), au niveau 4 (critique, mort de plusieurs personnes). Le niveau SSIL est un niveau de confiance à atteindre qui s'appuie sur la maîtrise de la qualité du logiciel par un processus de développement logiciel préétabli et systématique. Cette norme propose un cycle de vie en V classique et requiert la mise en œuvre de techniques telles que : les méthodes formelles de spécification et de conception, la traçabilité des exigences, les tests unitaires, la couverture de tests, etc.

Si une mesure ou technique fortement recommandée (HR défini [CEN 01a] page 46) n'est pas utilisée, le raisonnement sous-tendant cette décision doit être détaillé : il est nécessaire de montrer que grâce au processus global mis en œuvre et/ou à l'utilisation d'autres techniques cette mesure/technique n'est pas nécessaire.

Il est important d'indiquer que dans le domaine ferroviaire et plus particulièrement pour les logiciels, il est obligatoire d'avoir une évaluation indépendante (voir le chapitre 14 de la norme CENELEC EN 50128 [CEN 01a]). Cette évaluation du logiciel est réalisée par une entité indépendante de la réalisation du logiciel : un ISA (Independant Safety Assessor). Lors de l'évaluation du logiciel vis à vis de la norme, il est attendu une démonstration de la conformité à la norme et chaque raisonnement lié à la non prise en compte d'une mesure/technique sera vérifié et validé ou non par l'organisme extérieur.

3.3. Elaboration de la méthode de preuve de robustesse

Nous nous plaçons dans le contexte courant d'une application de niveau SSIL 3-4 pour laquelle certaines recommandations fortes (HR) n'ont pas été suivies. Nous avons élaboré une méthode qui permet d'atteindre les objectifs manqués du fait du non respect des recommandations fortes, de couvrir les objectifs correspondants et de les justifier en montrant la pertinence et la complétude de la robustesse du logiciel vis-à-vis de valeurs dysfonctionnelles.

Le Tableau 3.3 ci-dessous liste trois choix de développement qui sont souvent faits lors de développement logiciel et qui entraînent le non respect des recommandations HR [CEN 01a]. La première colonne présente le choix de développement, la seconde contient la recommandation non suivie à cause de ce choix et la troisième associe une référence à la recommandation utilisée dans la suite de ce chapitre.

Choix de développement	*Recommandation forte (HR) non suivie*	*Référence*
Le langage de programmation choisi est le langage C qui n'offre pas de typage fort.	Utilisation d'un langage de programmation à fort typage (Tableau A.4 - point 7, page 50).	HR-1
Le logiciel est intégré en mode *bigbang* c'est à dire que l'ensemble des modules (ou de gros paquets) est intégré directement sans observable intermédiaire.	L'intégration des modules logiciels doit être le processus de regroupement progressif de chacun des modules logiciels testés au préalable (§10.4.17, page 24).	HR-2
Les tests unitaires et les tests d'intégration ne permettent pas de démontrer l'adéquation aux recommandations de la norme.	La fourniture pour chaque module d'un compte rendu de la couverture des tests montrant que toutes les instructions du code source sont exécutées au moins une fois (§10.4.14-ii, page 24).	HR-3
	L'exécution d'un dossier de tests à partir d'une analyse des valeurs aux limites (Tableau A.13 - point 1, page 55).	HR-4

Tableau 3.3 : *Correspondance entre des choix de développement courants et des recommandations HR [CEN 01a] non suivies*

Ne pas suivre la recommandation HR-1 entraîne une faiblesse de robustesse de l'application puisque le typage statique n'est pas réalisé. Pour pallier cette absence

de vérification automatique, les développeurs ajoutent du contrôle (dynamique) dans le code source des applications. Pour être comparable au typage statique, ce contrôle de valeur doit garantir que toutes les données manipulées restent dans leur domaine fonctionnel[2] tout au long des calculs et doit être intégré systématiquement au code source de l'application. Ceci ne peut être assuré qu'en vérifiant la présence et la correction des points de contrôle posés dans le logiciel puisqu'ils ne sont pas posés automatiquement. Nous utilisons l'analyse statique de programme pour réaliser une partie de cette vérification.

La Figure 3.4 présente dans la colonne de gauche un code source non contrôlé écrit en langage C, et dans celle de droite le code intégrant le contrôle : la valeur de la variable *etat* est contrôlée entre son calcul par *compute_etat* et son utilisation par *use_etat*. Le domaine fonctionnel {*DISPO, BUSY*} de la variable *etat* est défini par son type énuméré. Dans le code contrôlé, si la valeur de *etat* fait parti de ce domaine, l'exécution continue sans modification. Sinon, l'exécution est modifiée par l'appel de la fonction *ERREUR*.

typedef enum {DISPO,BUSY} EVT ; … *EVT etat ;* *etat = compute_etat(…) ;* *…=use_etat(etat) ;*	*typedef enum {DISPO,BUSY} EVT ;* … *EVT etat ;* *etat = compute_etat(…) ;* if ((*etat* == *DISPO*) || (etat == *BUSY*)) { *…=use_etat(etat) ;* } else { ERREUR(etat) ; }

Figure 3.4. *Exemple* ***de contrôle de valeur***

Le défaut d'intégration progressive HR-2 et ses effets sur les tests HR-3, HR-4 peuvent être couverts dynamiquement avec des tests unitaires, et des tests d'intégration hors bornes. Le nombre de tests requis pour tester toutes les combinaisons de valeurs de variable possibles est de l'ordre du produit des

[2] On appelle domaine fonctionnel un domaine spécifié de valeurs acceptables pour une variable. En C, les types scalaires sont stockés dans des espaces mémoire qui sont des multiples d'octets, ou à la limite dans des champs de bits qui permettent de représenter 2^n valeurs où n est le nombre de bits. Ces types sont utilisés pour toutes les variables quelque soit leur domaine fonctionnel et même les types énumérés sont en fait traités comme des *int*. De ce fait, le domaine des valeurs associé au type des variables est beaucoup plus large que leur domaine fonctionnel. Le langage C ne permet donc pas de contrôler automatiquement des domaines fonctionnels des variables.

cardinaux des domaines fonctionnels des entrées manipulées : ces tests sont fréquemment trop nombreux pour être réalisés dans les temps et moyens des projets. Nous utilisons l'**analyse statique** pour propager les **domaines fonctionnels** spécifiés pour les entrées du logiciel vers le **contrôle de valeur** lorsqu'il est présent dans le code source et vers les consommations de donnée en général. Si le contrôle vérifie un domaine différent de celui calculé par propagation, c'est qu'il est possible de produire des valeurs dysfonctionnelles à partir d'entrées correctes ou que le contrôle est plus stricte qu'attendu. Par contre, si le contrôle vérifie le domaine calculé par propagation, c'est que pour des valeurs correctes des entrées toutes les exécutions produisent des données appartenant aux domaines attendus. Ceci complète donc les tests aux limites, prouve que l'exécution ne sort pas du contrôle présent dans le code et que ce contrôle n'est pas plus restrictif qu'attendu

Finalement, nous considérons un code comme robuste vis-à-vis de valeurs dysfonctionnelles si il implémente un contrôle de valeurs, et si la correction (cohérent vis-à-vis des domaines fonctionnels) de ce contrôle est établie.

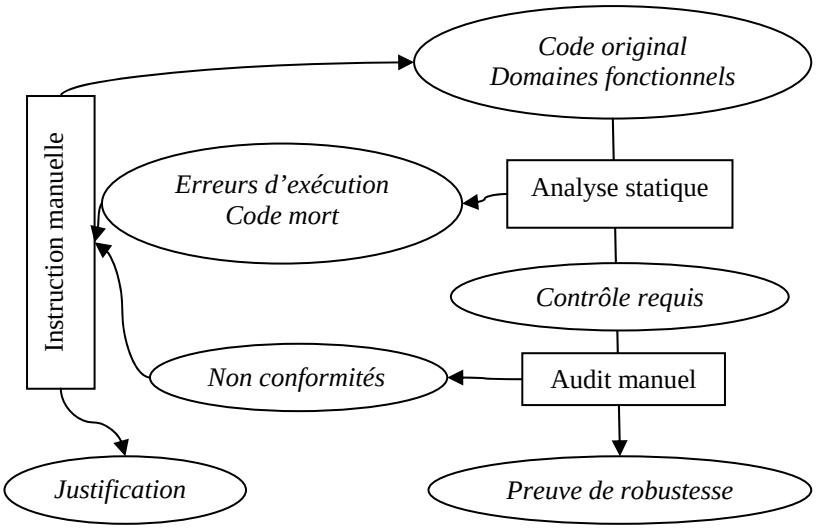

Figure 3.5. *Méthode de vérification de robustesse*

Nous avons élaboré une méthode spécifique pour vérifier la robustesse d'un logiciel implémentant le contrôle de valeur. Cette méthode présentée Figure 3.5

utilise l'analyse statique de programme, mais nécessite aussi des traitements manuels. Elle se compose de deux étapes principales: *(1)* une analyse statique qui calcule le contrôle requis pour assurer la robustesse du logiciel à partir des domaines fonctionnels, et *(2)* un audit manuel qui permet de montrer la conformité du contrôle de valeur. Il faut noter qu'ici analyse statique ne signifie pas utilisation directe d'un outil d'analyse statique, mais combinaison d'étapes manuelles et automatisées.

Cette méthode permet à chaque étape de détecter des incohérences entre le code source original et les domaines fonctionnels spécifiés : des erreurs d'exécution sont détectées durant la première étape, et des non conformités sont identifiées durant la seconde. Le Tableau 3.6 associe à chaque recommandation l'objectif à prouver par notre méthode de vérification, et les erreurs ainsi détectées : si une erreur est détectée, l'objectif n'est pas atteint et la recommandation n'est pas couverte. La conformité du contrôle du logiciel n'est donc prouvée que si toutes les erreurs détectées statiquement et toutes les non conformités résultat de l'audit sont instruites : c'est-à-dire qu'elles sont corrigées dans le code source ou dans les domaines spécifiés, ou qu'elles sont justifiées. L'étape d'instruction des erreurs trouvées est donc indispensable à la correction de la méthode, mais n'est pas décrite dans ce chapitre car elle dépend trop de l'application cible.

Référence	*Objectif à prouver*	*Erreurs à instruire*
HR-1	La production de données en dehors de leurs domaines fonctionnels est détectée – par une violation de contrôle.	Erreurs d'exécution Non conformités
HR-2	Les modules ne produisent pas de données erronées à partir de valeurs correctes des entrées	Erreurs d'exécution
HR-3	Les instructions du programme non exécutable sont détectées.	Code mort
HR-4	Les bornes des entrées des modules sont atteignables et atteintes par les modules qui les consomment ; et qu'une entrée qui prend valeur en dehors de son domaine fonctionnel est détectée comme incorrecte par la fonction qui la consomme.	Erreurs d'exécution

Tableau 3.6. *Erreurs à instruire*

De plus, cette méthode permet de mettre en œuvre des recommandations supplémentaires de la norme [CEN 01a] :
LR-5 la *programmation défensive* (Tableau A.3 - point 1, page 49) ;

LR-6 la *programmation par assertion* (Tableau A.3 - point 5, page 49) ;
LR-7 l'utilisation de l'*analyse statique* (Tableau A.5 - point 3, page 51) ;
LR-8 l'*analyse des valeurs aux limites* (Tableau A.19 - point 1, page 58).

3.4 Description générale de la méthode

Cette section définit informellement les notions de « contrôle requis » pour assurer la robustesse du logiciel et de contrôle effectivement posé « contrôle effectif ». Elle présente ensuite les points clés de leur calcul sur le code source d'une application, ainsi que de la vérification de robustesse définie comme la cohérence entre le « contrôle requis » et le « contrôle effectif ».

3.4.1 Contrôle de valeur requis ou effectif

Le contrôle de valeur décrit succinctement, [FAU 09] est implanté dans le code par des points de contrôle qui vérifient au point de code choisi que la valeur de la variable est dans son domaine fonctionnel. Si le contrôle réalisé aboutit à un succès alors l'exécution continue sans changement sous l'hypothèse que la variable est fonctionnellement correcte, sinon une erreur est détectée et le logiciel effectue une action de sécurité pré-définie.

L'action de sécurité est définie à partir des règles de sécurité applicables au logiciel cible et peut correspondre à différents comportements : mettre le logiciel dans un état final (position de repli en ferroviaire), corriger l'état courant et continuer l'exécution, ou restaurer l'état initial et recommencer l'exécution. Cette action est en général similaire pour tous les points de contrôle puisque définie par les règles de sécurité applicables au logiciel cible. La Figure 3.7 présente deux exemples d'actions de sécurité possibles: dans la colonne de gauche l'exécution se poursuit après avoir signalé l'erreur par *Signal_Defaut*, dans la colonne de droite le logiciel se met en position de repli après avoir signalé l'erreur.

```
if (controle_correct)
{
      /* ne rien faire */
}
else
{
  Signal_Defaut(etat) ;
}
```

```
 if (controle_correct)
{
  /* ne rien faire */
}
else
{
   Signal_Defaut(etat) ;
   POSITION_REPLI ;
}
```

Figure 3.7. *Exemple d'action de sécurité*

Une fois la stratégie de localisation connue et l'entrée à contrôler déterminée, la construction d'un point de contrôle nécessite de connaître :

- Sa localisation : Elle est déterminé par l'obligation de « vérifier la valeur avant utilisation » qui se traduit en terme de code par « avant les instructions de consommations ». Et comme la valeur n'est connue qu'après sa production, le point de contrôle doit être localisé entre chaque production et ses consommations. Ceci correspond à un ensemble de localisations possibles dans le programme.

- Son domaine fonctionnel : Si il est déterminé lors de la spécification du logiciel, il est alors connu et utilisé en entrée de la méthode. Si il n'est pas connu, il peut être calculé à partir des domaines fonctionnels spécifiés et du code exécuté. Il devra alors être calculé par la méthode.

Nous définissons « le contrôle requis » comme le contrôle devant être posé pour assurer la robustesse du logiciel. Comme sa localisation est choisie entre la production et les consommations, un point de contrôle requis est décrit par le quadruplé (*entrée*, *domaine de valeurs*, *localisations de production*, *localisations de consommation*) où l'*entrée* est un nom de variable ou un chemin d'accès mémoire, le *domaine de valeurs* est une description des valeurs correctes pour l'entrée entre chaque *production* et les *consommations* correspondantes, où une localisation est décrite par un triplet (*nom de fichier, numéro de ligne, numéro de colonne*). Il faut noter que les localisations de production et de consommation se situent potentiellement dans des fichiers et des fonctions différentes.

	typedef enum {DISPO,BUSY,UNK} EVT ; *…* *EVT etat ;*	
1 :	*etat = compute_etat(…) ;*	(etat,{DISPO,BUSY},{1},{3,5})
2 :	*if (condition)*	
3 :	*{ …=use_etat_1(etat) ;…}*	
4 :	*else*	
5 :	*{ …=use_etat_2(etat) ;…}*	
6 :	*…=use_etat_3 (etat) ;*	

Figure 3.8. *Exemple de point de contrôle requis*

La Figure 3.8 présente colonne de droite le point de contrôle requis pour le code original présenté colonne de gauche : l'entrée est *etat*, les valeurs fonctionnelles sont {*DISPO, BUSY*} et les localisations sont décrites par le numéro de ligne pour faciliter la lecture {1, 2, 3, 4, 5}. Le point de contrôle requis signifie qu'un point de

contrôle doit être posé pour protéger les instructions 3 et 5 d'une erreur dans la valeur calculée par l'instruction 1. On fait l'hypothèse ici que le contrôle de la valeur avant les instructions {3,5} assure que la valeur à l'instruction {6} est correcte, donc un point de contrôle n'est pas nécessaire entre les lignes 5 et 6.

Nous définissons « le contrôle effectif » comme le contrôle effectivement présent dans le code source d'une application. Il est décrit par le triplet (entrée, domaine de valeurs effectif, localisation effective). La localisation des points de contrôle effectifs est le résultat d'un choix dans l'ensemble des localisations entre production et consommations. Ce choix n'est en général pas laissé au développeur, mais dirigé par une stratégie choisie globalement pour un logiciel, un projet. Les deux stratégies extrêmes sont : la stratégie « au plus tôt » qui entraîne la pose d'un point de contrôle effectif juste après l'instruction de production, la stratégie « au plus tard » qui entraîne la pose de plusieurs points de contrôle effectifs placés juste avant les consommations.

La Figure 3.9 présente deux exemples de code contrôlés pour l'exemple Figure 3.8 suivant les stratégies: « au plus tôt » colonne de gauche (un point de contrôle) et « au plus tard » colonne de droite (deux points de contrôle).

1 : *2 :* *3 :* *4 :* *5 :* *6 :*	*etat = compute_etat(…) ;* if ((*etat == DISPO*) || (*etat == BUSY*)) { *if (condition)* *{ …=use_etat_1(etat) ;…} ;* *else* *{ …=use_etat_2(etat) ;…} ;* *…=use_etat_3(etat) ;* } else {ERREUR(etat) ;} ;	*1 :* *2 :* *3 :* *4 :* *5 :* *6 :*	*etat = compute_etat(…) ;* *if (condition)* *{* if ((*etat == DISPO*) || (*etat == BUSY*)) { *…=use_etat_1(etat) ;…* } else { ERREUR(etat) ;} ; *}* *else* *{* if ((*etat == DISPO*) || (*etat == BUSY*)) { *…=use_etat_2(etat) ;…* } else {ERREUR(etat) ;} *;} ;* *…=use_etat_3(etat) ;*
	{(*etat*, {*DISPO,BUSY*}, 1)}		{(*etat*, {*DISPO,BUSY*}, 3), (*etat*, {*DISPO,BUSY*}, 5)}

Figure 3.9. *Exemples de points de contrôle effectifs*

Pour vérifier qu'un logiciel est robuste, il faut donc calculer l'ensemble des points de contrôle requis, puis vérifier qu'ils sont tous implantés (présence et correction) par un ou plusieurs points de contrôle effectifs suivant la stratégie de localisation choisie.

Au cours du développement la complexité de l'implantation, les modifications du logiciel et de sa spécification peuvent entraîner des incohérences dans le contrôle du logiciel. Il est donc nécessaire de vérifier la correction du contrôle effectif vis-à-vis du contrôle requis. Un code est défini comme robuste vis-à-vis de valeurs dysfonctionnelles si tous les points de contrôle requis ont été implantés.

La Figure 3.10 présente une implantation erronée du point de contrôle requis présenté Figure 3.8. Le point de contrôle effectif est erroné pour deux raisons : la seconde consommation (ligne 5) n'est pas protégée de valeurs incorrectes de *etat*, et la première (ligne 3) est protégée de façon trop restrictive puisque toutes les valeurs correctes de *etat* ne sont pas acceptées.

```
1 :  etat = compute_etat(…) ;
2 :  if (condition)
         { if (etat == DISPO)
3 :        { …=use_etat_1(etat) ;… }
           else { ERREUR(etat) ;} ;
         }
4 :  else
5 :     { …=use_etat_2(etat) ;… } ;
6 :  …=use_etat_3(etat) ;
```

Figure 3.10. *Exemple de point de contrôle effectif erroné*

3.4.2. Calcul du contrôle requis

L'ensemble des points de contrôle requis peut être calculé manuellement à partir du code source de l'application et des domaines fonctionnels connus grâce aux trois étapes suivantes:

- Identification des entrées du logiciel et de chaque fonction

L'identification des entrées se fait par une analyse pour chaque fonction. Les entrées du logiciel sont les variables associées aux appels de fonction d'accès à l'environnement (IO telles que getc, fget ...). Les entrées d'une fonction sont d'une part les paramètres, les variables statiques, et les sorties consommées des fonctions appelées ; et d'autre part les variables globales utilisées directement ou indirectement par la fonction. Dans certains cas, les entrées ne sont pas implantées sous forme de variables, mais de composants de variables (champs de structure, case de tableau). De manière générale, une entrée est une zone mémoire décrite par un

chemin construit à partir d'un nom de variable et d'accesseurs du langage (accès à une case de tableau, à un champ de structure).

- Localisation des lieux de production et de consommation

La localisation des lieux de production et de consommation d'une entrée nécessite une analyse inter-procédurale complexe pour suivre les variables (chemins) renommées à travers les appels de fonctions. La production de la valeur des paramètres est considérée réalisée au début de la fonction. La production des variables globales ou locales est réalisée par l'appel de : fonctions d'accès à l'environnement (getc), ou de toute autre fonction de l'application. La consommation de toutes les valeurs correspond à un calcul explicite à partir de la valeur : on considère que le passage par paramètre ou le stockage dans une autre variable ne sont pas des consommations. Si l'entrée est un objet scalaire les productions/consommations sont atomiques. Par contre si l'entrée est composite comme une structure ou un tableau, les productions/consommations sont partielles et multiples: il existe alors des points de production/consommation pour chaque composante de l'entrée.

- Calcul des domaines de valeurs fonctionnels.

Les domaines de valeurs fonctionnels ne peuvent pas être calculés uniquement à partir du code source. Si des domaines fonctionnels sont connus pour toutes les entrées, aucun calcul n'est nécessaire. Mais en général, les domaines fonctionnels sont uniquement connus sur les entrées du logiciel car ils font partie de sa spécification: les valeurs de ces entrées sont produites par l'environnement et ont en général un domaine de valeurs restreint. Par contre, les domaines des entrées des fonctions sont souvent inconnus en particulier en cas d'intégration big-bang. Ils doivent alors être calculés à la main à partir de connaissances "métiers" par rétro-conception à partir du code source du programme. En particulier, ces domaines peuvent être calculés en propageant les domaines fonctionnels connus à travers le programme.

3.4.3. Vérification du contrôle effectif

Pour vérifier le contrôle effectif déjà présent dans le code cible, il faut calculer le contrôle requis comme décrit précédemment, puis parcourir le code pour vérifier si chaque point requis est implanté dans le code. Un point de contrôle effectif (*var*, *dom*, *loc*) implante un point de contrôle requis (*var**, *dom**, *prod**, *conso**), si il porte sur la même variable *var*==*var** avec le même domaine fonctionnel *dom*==*dom** et si sa localisation satisfait la stratégie de localisation *loc*∈strategy(*prod**, *conso**).

L'algorithme général de vérification des points de contrôle est le suivant : pour chaque point de contrôle requis (*var, dom*, prod*, conso**), on recherche les points de contrôle effectifs qui contrôlent la valeur de la variable *var*:

- Si aucun point effectif n'existe, alors on ajoute cette non-conformité
  (*var*, *dom*, prod*, conso**)? None
- Pour chaque point de contrôle effectif (*var*, *dom*, *loc*) ,
  On vérifie que la localisation *loc* est correcte vis-à-vis de la stratégie de pose et pour une localisation de production *prod* et de consommation *conso*
  - Si la localisation est incorrecte, alors on ajoute la non-conformité (*var, dom*, prod*, conso**) ? (*var, dom, loc*) | NC(*prod,conso,loc*).
  - Si la localisation *loc* est correcte, alors on vérifie les domaines
    - Si *dom =/= dom**, alors on ajoute la non-conformité (*var,dom*,prod*,conso**) ? (*var,dom,loc*) | NC(*prod,conso,loc,dom*).
    - Si *dom == dom**, alors on ajoute une conformité partielle (*var,dom*,prod*,conso**) ?(*var,dom,loc*) | CP(*prod,conso*).
- Puis on analyse l'ensemble des conformités pour évaluer leur couverture
  - Si tous les points de productions et de consommations, sont associés à un point de contrôle effectif, alors on ajoute une conformité totale (*var, dom*, prod*, conso**)?CT
  - Sinon, certains points de productions et de consommations ne sont pas associés à des points effectifs, on ajoute une non-conformité (var, *dom*, prod*, conso**)?CP

Une fois tous les points requis étudiés, on a calculé :

- les points requis non implantés dans le code,
- les non conformités potentielles.

Finalement, ces non conformités potentielles sont systématiquement instruits et peuvent amener à une modification du code source ou des domaines fonctionnels spécifiés, ou à une justification.

3.5 Méthode de calcul du contrôle requis

Le contrôle requis pour un logiciel est long et difficile à calculer car il nécessite des analyses inter procédurales sur des données complexes telles que les domaines de valeurs des variables. Les outils basés sur l'analyse statique (par interprétation abstraite) réalisent automatiquement de telles analyses, mais ne calculent pas directement les points de contrôle requis. Nous avons développé une méthode pour calculer le contrôle requis utilisant au mieux les capacités des outils d'analyse statique tels qu'ils existent.

Les outils d'analyse statique simulent toutes les exécutions du programme cible en une exécution symbolique sur laquelle des propriétés dynamiques sont vérifiées. Cette exécution symbolique nécessite l'abstraction de valeurs concrètes en valeurs abstraites (élément de treillis), et le calcul de points fixes pour couvrir la récursion souvent présente dans les programmes (boucle, fonction récursive). Ils calculent les valeurs abstraites des variables en chaque point du programme par une exécution symbolique: la valeur abstraite d'une variable *v* représente l'ensemble des valeurs que peut prendre *v* au cours des différentes exécutions possibles du programme. Sur ces valeurs abstraites, les outils vérifient des propriétés telles que l'absence d'erreur d'exécution, d'erreur numérique…

Nous faisons ici l'hypothèse que l'outil d'analyse statique utilisé:
- détecte les erreurs d'exécution,
- calcule un sous ensemble du code mort,
- définit un opérateur *observe_value* permettant extraire la valeur d'une variable,
- définit un opérateur *assume_value* dont la sémantique est assert puis assume (voir 3.5.2)**,**
- calcule le graphe d'appel du logiciel,
- calcule le dictionnaire de données.

Notre méthode de calcule des points de contrôle requis à partir du code source et des domaines fonctionnels opère en deux étapes principales comme présenté Figure 3.11: l'Etape 1 identifie les entrées et localise leurs points de production et de consommation, et l'Etape 2 calcule les domaines de valeurs des entrées aux points de productions. Ces deux étapes mettent en oeuvre l'analyse statique pour des objectifs différents.

Durant la première étape détaillée section 3.5.1, l'outil d'analyse statique est utilisé pour calculer le graphe d'appel et le dictionnaire de données du code original. Lors de ce calcul, l'outil vérifie de plus l'absence d'erreur de compilation et d'exécution (RTE - Runtime Error). Ces erreurs doivent être instruites avant de pouvoir utiliser les autres résultats: ceci est symbolisé dans la Figure 3.11 par un arc allant des erreurs d'exécution vers le code original. Une fois les erreurs corrigées, l'outil d'analyse statique produit le dictionnaire de données et le graphe d'appel de l'application.

Durant la deuxième étape détaillée section 3.5.2, l'analyse statique est utilisée pour calculer les domaines fonctionnels inconnus et ainsi compléter la définition des points de contrôle requis. Les erreurs détectées durant cette étape doivent elles aussi être instruites avant de pouvoir utiliser ces domaines non spécifiés des entrées.

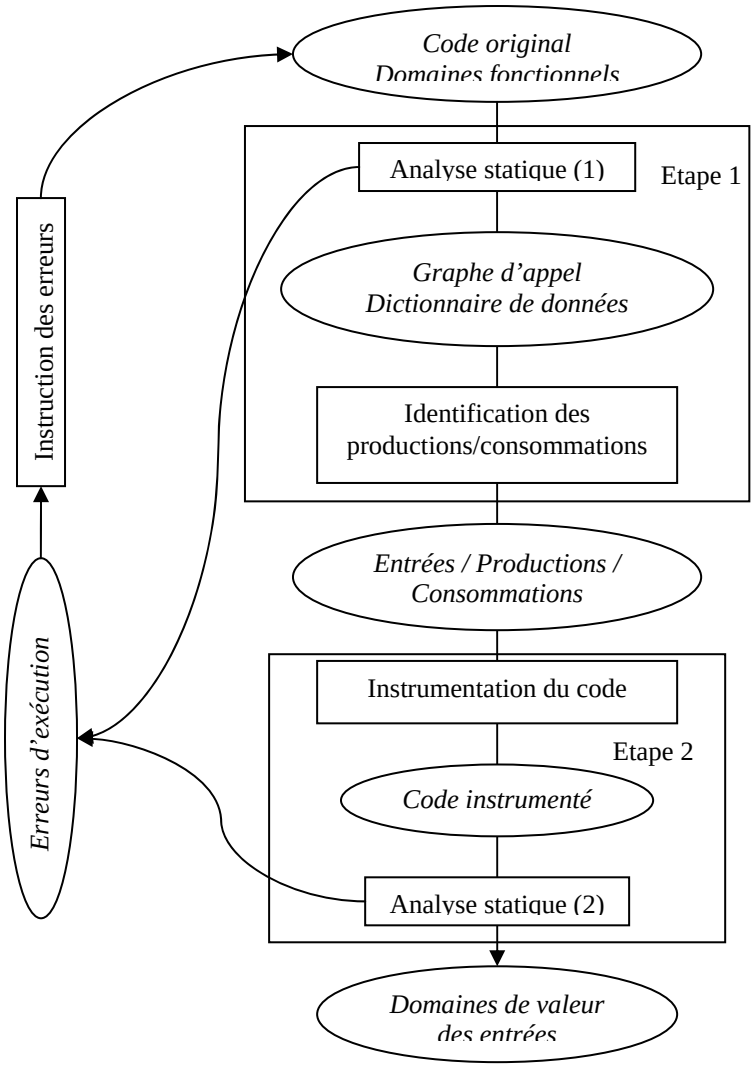

Figure 3.11. Calcul du contrôle requis

3.5.1. Identification des productions/consommations des entrées

L'étape 1 commence par une analyse statique du code original qui a pour but de calculer le graphe d'appel et le dictionnaire de données pour les variables globales et

statiques. Ces informations sont utilisées pour calculer les entrées et les localisations des productions et des consommations comme décrit ci-dessous.

Le traitement réalisé pour calculer les productions / consommations de chaque type d'entrée est décrit ci-dessous :

- Les entrées du logiciel : Les appels des fonctions d'entrée du langage (getc, scanf, fscanf ...) sont recherchés dans le graphe d'appel: ce sont les lieux de production des valeurs des entrées du logiciel. Par examen du code source, on détermine les variables affectées à partir de ces appels. On obtient ainsi les entrées du logiciel et leurs lieux de production. Ensuite, les lieux de consommation de leurs valeurs sont recherchés à travers les appels de fonction.
- Les entrées/sorties globales ou statiques des fonctions : Le dictionnaire de données contient la liste des variables globales ou statiques de l'application. Si il contient aussi leurs écritures (lieux de production) et leurs lectures (lieux de consommation) directes et indirectes, il n'y a rien de plus à calculer. Sinon, il faut tout d'abord trouver les utilisations directes dans le code source puis suivre le graphe d'appel pour trouver tous les accès indirects.
- Les paramètres des fonctions : Si le dictionnaire de données contient les paramètres des fonctions, et leurs lectures écritures directes et indirectes, il n'y a rien à calculer. Sinon la liste des paramètres d'une fonction est obtenue en regardant l'en-tête de sa définition. Pour calculer les entrées/sorties, il faut déterminer tous les accès aux paramètres. On considère que le lieu de production d'un paramètre se situe avant l'exécution de la première instruction. Les lieux de consommation d'un paramètre sont calculés en suivant sa valeur à travers des appels de fonction (et donc des re-nommages potentiels). Ce calcul est entièrement fait à la main si l'outil d'analyse statique ne donne pas d'information sur les paramètres des fonctions.

Si on ajoute un type d'entrée, il suffit de définir comment calculer ces entrées et leurs productions/consommations et le reste de la méthode s'applique sans changement.

3.5.2. Calcul des domaines de valeurs des entrées

L'étape 2 de la méthode décrite Figure 3.11 a pour objectif de calculer les domaines de valeurs des entrées compatibles avec les domaines spécifiés pour les entrées du logiciel. Mais elle vérifie aussi la cohérence entre les domaines fonctionnels d'entrée et le code source de l'application cible.

Elle commence par une phase d'instrumentation manuelle du code, suivie d'une analyse statique du programme instrumenté qui calcule les domaines fonctionnels inconnus des entrées.

La phase d'instrumentation a pour but de placer d'une part les contraintes sur les entrées dont le domaine est spécifié, et d'autre part les points d'observation sur les autres entrées. Les domaines spécifiés sont traduits en contraintes en utilisant l'opérateur *assume_value* de manière à ce que toute violation de la contrainte entraîne l'arrêt de l'exécution (sémantique courant de l'opérateur *assert*), sinon l'exécution continue en prenant la contrainte comme hypothèse (sémantique assume). Ces contraintes sont placées au plus tôt après la production de la valeur. Les points d'observation sont traduits grâce à l'opérateur *observe_value* placé dans le code aux localisations des points de production c'est-à-dire à l'entrée des procédures pour les paramètres et les variables globales (statiques). Le choix de la stratégie « au plus tôt » pour la pose des contraintes et des points d'observation minimise le volume d'instrumentation et nécessite uniquement la connaissance des lieux de production.

On peut voir Figure 3.12 un exemple de code dans la deuxième colonne et sa version instrumentée dans la troisième colonne. L'observation de la variable *in* est posée avant l'instruction *1* et la contrainte sur la variable *line* est posée après l'instruction *1*.

		observe_value(in) ;
1 :	*gets(line) ;*	*gets(line) ;*
		assume_value(line) ;
	…	…
2 :	*etat = compute_etat(line,in) ;*	*etat = compute_etat(line,in) ;*
3 :	*if (condition)*	*if (condition)*
4 :	*{x=use_etat_1(etat) ;…}*	*{ x=use_etat_1(etat) ; …}*
5 :	*else*	*else*
6 :	*{ x=use_etat_2(etat) ;…} ;*	*{ x=use_etat_2(etat) ; …} ;*
7 :	*y=use_etat_3(etat) ;*	*y=use_etat_3(etat) ;*

Figure 3.12. *Calcul de point de contrôle requis*

L'analyse statique du code instrumenté propage les domaines de valeurs spécifiés des entrées, vers les points d'observation. Le résultat de l'analyse contient la liste des domaines calculés pour les entrées dont le domaine de valeurs n'est pas spécifié : il s'agit d'approximations des domaines de valeurs atteignables à partir des domaines spécifiés en exécutant symboliquement le code et en collectant les contraintes liées à l'usage par ce code des données résultantes des entrées (division par zéro par exemple). Cette approximation est due à l'abstraction utilisée par l'analyse statique pour prendre en compte toutes les exécutions possibles

(sound) : par exemple si le domaine est [3..4] ∪ [5..7] est calculé, il est sur-approximé en [3..7] en utilisant le treillis des intervalles.

De plus, l'analyse statique vérifie automatiquement la cohérence entre les domaines des entrées du logiciel et le code instrumenté. Les erreurs d'exécution, le code mort ou les violations du contrôle présent signalent des incohérences entre le code source et les contraintes instrumentées. Ces erreurs sont instruites pour être finalement soit corrigées, soit justifiées. Si elles doivent être corrigées, la correction peut nécessiter la modification du code source, ou des domaines fonctionnels spécifiés. Il est parfois difficile de remonter de l'erreur trouvée à la contrainte violée si l'influence de cette dernière n'est pas immédiate. C'est seulement une fois toutes les erreurs instruites comme le montre la Figure 3.11, que les domaines fonctionnels calculés sont correctes et donc utilisables.

3.6 Vérification du contrôle effectif d'une application industrielle

La méthode de calcul du contrôle requis décrit section 3.5 a été appliquée à la vérification du contrôle des valeurs d'une application ferroviaire dont le développement n'a pas suivi les recommandations HR-1, 2, 3 et 4, mais qui implante le contrôle de valeur tel que défini précédemment.

3.6.1 Logiciel cible

Le logiciel cible PING implémente un système d'enclenchement ferroviaire SSIL3-4. Ce produit contribue en sécurité au traitement des fonctions d'enclenchements des postes de signalisation ferroviaire, au contrôle et commande des éléments à la voie (aiguilles, signaux, etc.), et aux échanges de contrôles/commandes avec les systèmes externes comme l'illustre la Figure 3.13.

Ce logiciel intègre le contrôle des valeurs de ses entrées. On appelle entrée d'une fonction : ses paramètres, ses variables locales statiques, les variables globales qu'elle utilise et les variables de sortie des fonctions appelées. La stratégie choisie impose que toute consommation d'une entrée d'une fonction soit protégée par un point de contrôle placé dans la fonction qui consomme la valeur « au plus tard » avant la consommation. On appelle entrée du logiciel les variables (paramètre, locale ou globale) affectées par appel d'une fonction d'entrée sortie. Ces entrées doivent être protégées par un point de contrôle placé dans la fonction qui produit la valeur « au plus tôt » après la production.

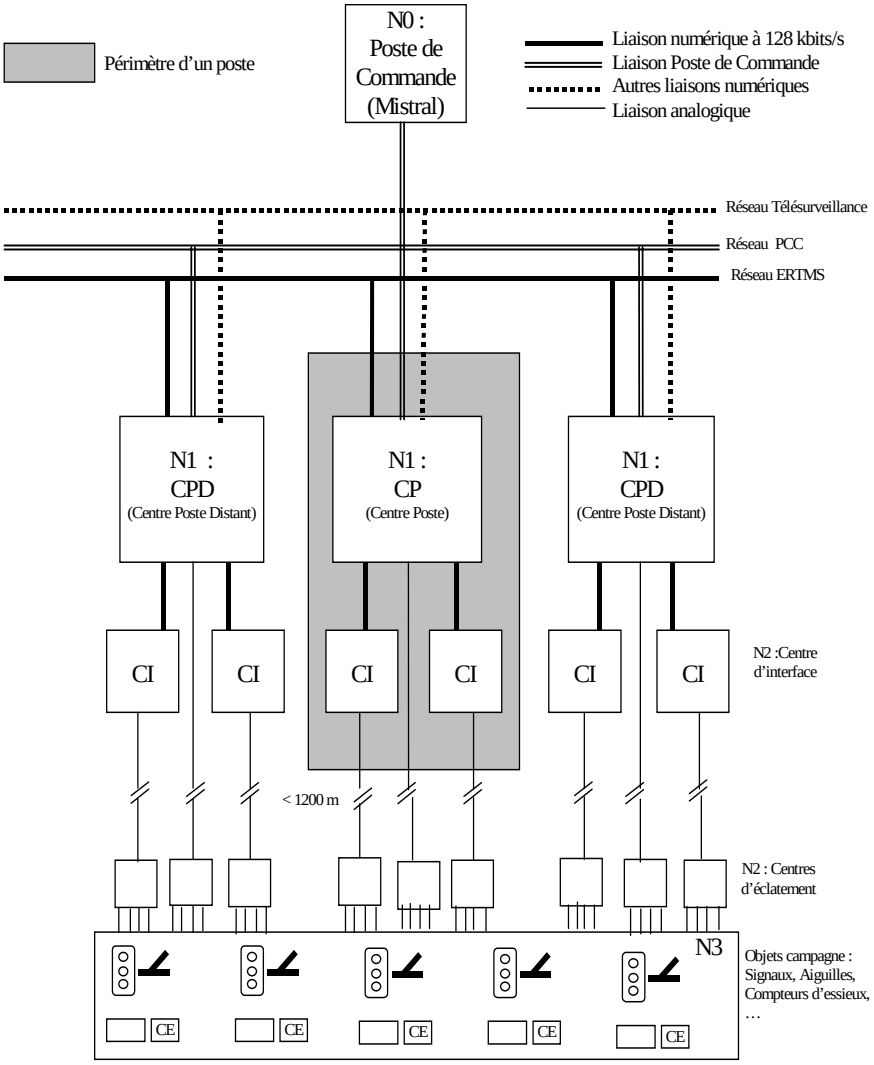

Figure 3.13. *Architecture du PING*

La forme générale du contrôle de valeur est donnée dans la Figure 3.14. L'appel de la fonction *Defaut_Fatal* commande le comportement restrictif du système et trace les erreurs détectées par le contrôle de valeur à partir du nom du module et du message lié au point de contrôle violé.

```
if ( Nbr_variables_securite >= MAX_VARIABLES_SECURITE)
  {
    Defaut_Fatal (module,  message_de_defaut );
    POSITION_REPLI
  }
  /* else Le traitement nominal continue */
```

Figure 3.14 : *Exemple de point de contrôle présent dans le logiciel PING*

On peut remarquer que la fonction *Defaut_Fatal* n'est pas réservée au contrôle de valeurs mais est utilisée pour traiter les cas d'erreur en général. De plus, la macro *ASSERT* qui définit un point de contrôle de façon univoque, n'est pas toujours utilisée pour poser les points de contrôle effectifs. De ce fait la recherche des points de contrôle effectifs théoriquement faisable automatiquement par recherche de forme sur le code source (pattern matching) doit être réalisée manuellement dans ce cas.

Les domaines fonctionnels sont spécifiés dans une table d'association nom de module, nom de fonction, nom de variable (ou chemin d'accès mémoire), domaine de valeurs. Ils sont donnés pour les entrées du logiciel. Les domaines fonctionnels sont décrits comme des ensemble de valeurs *{DISPO, BUSY}*, comme des intervalles *[3..12]*, ou comme des propriétés *non_null(p)*, *sizeof(s)==4*.

3.6.2 Mise en œuvre

Nous avons choisi PolySpace[3] comme outil d'analyse statique parce que c'est l'outil utilisé par l'industriel ayant développé le code source, et qu'il offre toutes les fonctionnalités que nous recherchons.

PolySpace est un outil d'analyse statique basé sur l'interprétation abstraite qui a pour but de détecter les erreurs d'exécution ainsi que les comportements non déterministes dans le code source d'une application écrite en langage C, C++ ou Ada. Une erreur d'exécution est un état d'erreur d'un programme parfaitement identifié dans la norme du langage cible comme un comportement non spécifié "unspecified behavior", indéfini "undefined behavior" ou dépendant du compilateur "implementation defined". L'outil calcule les points du programme cible pouvant provoquer un de ces comportements et leur associe un statut d'erreur : impossible

[3] © The MathWorks

(vert), potentielle (orange), certaine (rouge), code non atteignable (gris). PolySpace produit la liste d'association de ces points de programme et de leur statut d'erreur. De plus il calcule un sous ensemble du code mort : les instructions jamais exécutées, et les procédures jamais appelées.

PolySpace offre les deux opérateurs indispensables à l'instrumentation:
- les points d'observation (*observe_value*): Un point d'observation est traduit en point d'inspection (IPT) sur les variables aux points choisis. *L'instruction #pragma Inspection_Point var1 var2* permet de connaître la valeur des variables *var1 var2* au point de programme où elle est posée. Les domaines calculés sont ajoutés aux résultats produits. Il est à noter que les points d'inspection peuvent être posés uniquement sur des variables de type scalaire, ce qui n'a pas permis l'observation de toutes les variables : en particulier les structures et les tableaux ne peuvent être observés globalement mais peuvent être observés composante par composante.
- les contraintes (*assume_value*) : Une contrainte est traduite en assertions. La sémantique de l'instruction *assert(contrainte)* est pour PolySpace: si la *contrainte* est vérifiée au point ou elle est posée, elle contraint le reste de l'exécution, sinon elle arrête l'exécution. Le statut des assertions (certainement violé, jamais violé ou potentiellement violé) est aussi présent dans les résultats produits mais ne permet pas de statuer directement sur le contrôle : en particulier le fait que l'assertion ne soit jamais violée (verte), prouve que le domaine contrôlé est inclus dans le domaine calculé par PolySpace, mais ne montre pas l'égalité. L'audit manuel décrit (section 3.4.3) reste donc indispensable à l'atteinte de l'objectif de vérification.

PolySpace met à la disposition des utilisateurs les résultats intermédiaires nécessaires à l'application de notre méthode:

- les fonctions non exécutables (code mort),
- le graphe d'appel des fonctions,
- le dictionnaire de données globales statiques.

3.6.2.1 Analyse préliminaire de l'application

La première étape de l'analyse statique d'un code source est sa compilation par l'outil. Pour prendre en compte les spécificités de cette application et permettre sa compilation, nous avons paramétré PolySpace comme décrit dans le Tableau 3.15.

Options PolySpace	Spécificité de l'application
-target i386 -OS-target no-predefined-OS -I APPLICATIONS/WATCOMC_Includes	Compilateur WATCOMC
-dos	Présence de "\" au lieu de "/" dans les noms de fichiers inclus.

-discard-asm	Présence de morceaux d'assembleur non traités mais bouchonnés (stub) automatiquement.
-D INTERRUPT= -D __far= -D FAR=	Utilisation d'« interruptions » et de « far pointeur » non reconnus et donc non simulés automatiquement.

Table 3.15 : *Liste des options nécessaires à la compilation*

La compilation par PolySpace est plus stricte que celle réalisée par les compilateurs courants puisqu'elle vérifie systématiquement le respect de la norme C ANSI. De ce fait, certaines modifications ont été apportées au code source de l'application pour corriger les erreurs syntaxiques détectées.

Une fois ce paramétrage terminé, l'analyse de l'application par PolySpace a montré que les interruptions n'étaient pas simulées. En particulier des boucles infinies ont été détectées à tort. Nous avons supprimé ces boucles qui ne font que retarder l'exécution de la suite des instructions sans changer leur comportement.

Nous avons simplifié l'application pour permettre une analyse par PolySpace plus efficace. Pour diminuer le nombre de pointeurs manipulés, nous avons redéfini les fonctions d'archivage sans apport fonctionnel, en leur associant une sémantique vide. De même nous avons remplacé les fonctions *Defaut_Fatal*, *POSITION_REPLI et PSEUDO_ DEFAUT_FATAL*, par des arrêts certains de l'exécution. Nous avons défini la fonction *ALLOUER_MEMOIRE* à la fonction standard *malloc*. En faisant cela, nous avons divisé le nombre d'alias calculés par PolySpace par un facteur 5.8.

De plus, nous avons redéfini la macro *ASSERTION* qui implante les points de contrôle de façon univoque à l'appel de la fonction *assert* reconnue par PolySpace. Au cours de ce travail préliminaire, nous avons détecté et corrigé trois erreurs d'exécution dans le code source de l'application.

3.6.2.2 Instrumentation et analyse du code instrumenté

L'instrumentation du code source a pour but d'ajouter d'une part les contraintes spécifiées et d'autre part les points d'observation. Ces deux types d'instrumentations sont contrôlés indépendamment grâce aux macros *Active_contrainte* et *Active_obser vation* qui sont activées par les options de compilation *–D Active_ contrainte* et *–D Active_ observation.*

Les contraintes sont implémentées sous forme de fonctions C regroupant des assertions. La Figure 3.16 présente la traduction du domaine fonctionnel *[0..NB_TYPE_CARTE-1]* de la variable *typeCarte* en la fonction de contrainte *contrainte_typeCarte*. Certains domaines fonctionnels ne sont pas traduits en contraintes car, trop compliqués, ils ne sont pas propagés par PolySpace : par exemple les champs de bits (16 bits poids fort [0..4], poids faibles [0..255]), les structures d'adresse IP (192.168.[0..255].[0..255]).

```
void contrainte_typeCarte(E_TYPE_CARTE typeCarte)
{
#define TMP_typeCarte typeCarte
  assert(TMP_typeCarte >= 0);
  assert(TMP_typeCarte <= NB_TYPE_CARTE - 1);
}
```

Figure 3.16 : *Exemple de fonction de contrainte*

Ces fonctions sont dupliquées dans le cas où la contrainte doit être posée en plusieurs points du programme. En général les contraintes sont posées une fois entre la production et la consommation : juste après production par simplicité. Mais si il existe une opération de *Cast* entre la production et la consommation, pour que la propagation ne soit pas arrêtée par PolySpace, la contrainte est posée une fois juste après la production et une seconde fois avant la consommation si elle peut être traduite sur le nouveau type. Ceci permet de renforcer l'effet de ces contraintes surtout si elles portent sur des composants d'objets structurés (case de tableau, champ de structure). La Figure 3.16 présente l'utilisation de la fonction *contrainte_typeCarte* dans la fonction *GetIdCarte*.

Les points d'observations sont implémentés sous forme de macros qui s'expansent en *Inspection_Point* pour les entrées du logiciel et des fonctions. Il faut noter que PolySpace ne permet d'observer que le sous-ensemble des entrées qui sont scalaires. La Figure 3.17 présente l'utilisation de la macro OBS_DIP_GetIdCarte dans la fonction *GetIdCarte* de l'application originale.

```
1 : T_idCarte DIP_GetIdCarte(E_TYPE_CARTE typeCarte)
2 : {

    #ifdef Active_observation
      OBS_DIP_GetIdCarte(typeCarte)
    #endif /* Active_observation */
```

```
      #ifdef Active_contrainte
        contrainte_typeCarte(typeCarte);
      #endif /* Active_contrainte */

3 :     DIP_ASSERTION( typeCarte < NB_TYPE_CARTE);

5 :     return gLesIdCarte[typeCarte];
6 :   } /* FIN DIP_GetIdCarte */
```

Figure 3.17 : *Instrumentation de la fonction GetIdCarte*

Le code instrumenté a été ensuite analysé par PolySpace en utilisant les options de précision maximales (-03 –to pass4) en plus des options nécessitées par la compilation présentées dans le Tableau 3.18.

Options PolySpace	
-target i386	-D INTERRUPT=
-OS-target no-predefined-OS	-D __far=
-I APPLICATIONS/WATCOMC_Includes	-D FAR=
-dos	-O3
-discard-asm	-to pass4

Tableau 3.18 : *Options PolySpace d'analyse de PING instrumenté*

L'analyse du code instrumenté génère des erreurs d'exécution. L'erreur est un *assert* rouge si une contrainte fonctionnelle est directement violée dans le fichier de définition des fonctions de contrainte : c'est le cas si dans l'exemple Figure 3.16, la fonction *GetIdCarte* est appelée avec une valeur erronée (*typeCarte* ≥*NB_TYPE_ CARTE ou négative)* et les assertions contenues dans la fonction *contrainte_ typeCarte* (Figure 3.15.) seront violées. Les points de contrôle du code original peuvent être violés : dans l'exemple, si le point de contrôle en ligne *3* est mal défini (*typeCarte* > *NB_TYPE_CARTE*) une violation apparaît. Enfin, un accès hors borne ligne 5 peut être détecté si la taille de la variable *gLesIdCarte* est déclarée de taille trop faible par rapport au contrôle posé (size(*gLesIdCarte)* < *typeCarte)*.

Une fois ces incohérences corrigées dans le code source ou les domaines fonctionnels spécifiés, les résultats de l'analyse sont utilisés pour l'audit : les

domaines calculés aux points d'inspection servent de domaine de référence pour les domaines fonctionnels inconnus.

3.6.2.3 Audit de code

L'audit de code a pour but de vérifier que les points de contrôle effectifs implantent les points requis. En appliquant la stratégie pour la pose des points de contrôle (localisation choisie), on peut simplifier l'algorithme présenté section 3.4.3 comme décrit ci-dessous.

Le code source de chaque fonction exécutable est parcouru suivant un algorithme différent pour chaque type de point de contrôle :

- Pour le point de contrôle d'une entrée de la fonction *f* : il faut remonter dans la définition de *f*, de chaque consommation d'un paramètre ou d'une globale (statique) de *f*, jusqu'au début de la définition.
- Pour le point de contrôle des sorties d'une fonction *g* appelée dans la fonction *f* : pour chaque sortie de *g*, if faut descendre dans le code à partir de chaque appel de la fonction *g*, jusqu'à la fin de la définition de la fonction *f*.
- Pour le point de contrôle d'une entrée logiciel : il faut descendre dans le code à partir de l'appel de la fonction d'entrée qui produit la valeur de cette entrée, jusqu'à la fonction qui contient la consommation de cette valeur. Il s'agit d'une recherche complexe car inter-procédurale.

Par contre une fois ce parcours terminé, l'algorithme de conclusion sur la conformité est le même:

- Si aucun point de contrôle effectif n'est trouvé, c'est que la production n'est pas vérifiée et on l'ajoute à la liste des non conformités potentielles.
- Si au moins un point de contrôle effectif est trouvé la vérification est opérée en deux étapes successives:
  - On vérifie pour chaque point effectif que le domaine contrôlé est bien celui calculé par PolySpace.
    - Si les domaines ne sont pas égaux, alors on l'ajoute à la liste des non conformités potentielles.
    - Si les domaines sont égaux, alors on passe à l'étape suivante.
  - On vérifie que la protection offerte par ces points de contrôle est assurée pour toutes les exécutions possibles par calcul de la couverture des chemins présents dans le code:
    - Si les points effectifs protègent toutes les exécutions possibles, on les ajoute à la liste des conformités.
    - Sinon on les ajoute à la liste des non conformités potentielles.

Les non conformités ainsi obtenues, sont ensuite instruites une à une : les erreurs sont corrigées dans le code ou dans les domaines fonctionnels spécifiés, et les autres sont justifiées. Ce traitement est hors du propos de ce chapitre.

3.6.3. Résultats

3.6.3.1 Résultats directs

Nous avons appliqué la méthode à deux versions successives du logiciel PING (Poste Informatisé Nouvelle Génération)[4].

Le logiciel PING V45.05 implémenté en 92 Klignes de C se compose de 66 fichiers et de 717 fonctions. Les domaines fonctionnels de 95 entrées étaient spécifiés ce qui a permis le développement d'autant de fonctions de contrainte dans un fichier de 2397 lignes de code C. Nous avons placé les contraintes en 196 points de programme, et les points d'observation sur 1509 entrées scalaires dont les domaines fonctionnels sont inconnus.

L'outil PolySpace a détecté automatiquement 1 incohérence entre le code source et les domaines fonctionnels spécifiés. L'analyse statique du code contraint a entraîné 8 reprises de code et 1 reprise de domaine fonctionnel dans la documentation.

L'audit manuel de vérification du contrôle a nécessité l'examen de 2644 points de contrôle effectifs et a montré que 71% des points étaient corrects, 18% étaient non conforme et le statut des 11% restants a dû être établi par l'industriel car il requiert des connaissances fonctionnelles sur l'application.

Le logiciel PING V49.03 implémenté en 122 Klignes de C se compose de 87 fichiers et 1409 fonctions. Les domaines fonctionnels de 358 entrées étaient spécifiés ce qui a permis le développement d'autant de fonctions de contrainte dans un fichier de 8099 lignes de code C (contenant 1800 contraintes). Nous avons placé les contraintes en 255 points de programme et les points d'observation sur 4597 entrées scalaires dont les domaines fonctionnels sont inconnus.

L'outil PolySpace a détecté automatiquement une vingtaine d'incohérences entre le code source et les domaines fonctionnels spécifiés qui ont entraîné la reprise de 22 domaines fonctionnels dans la spécification. La validation des résultats est en cours, et l'audit manuel n'a pas commencé.

---

[4] Pour en savoir plus sur le logiciel PING, le lecteur pourra se reporter aux chapitres 4 et 5 de [BOU 09]

La vérification du logiciel n'est donc pas terminée, mais l'effet de l'augmentation du nombre de contraintes traitées sur la qualité des domaines calculés est visible : plus de 84% des domaines sont précis (différents du domaine associé au type de donnée) alors que pour la version précédente seulement 48% des domaines calculés étaient précis.

Ceci permet de dire que l'audit manuel permettra de statuer plus rapidement sur la correction des points de contrôle présents dans le code vis-à-vis des points requis.

3.6.3.2 Résultats indirects

L'application de cette méthodologie a permis de conforter la traçabilité verticale (voir la Figure 3.20) du cycle de développement en V (voir la Figure 3.19) et en particulier le raffinement de données des niveaux système et sous-systèmes vers les plus bas niveaux de l'application logicielle.

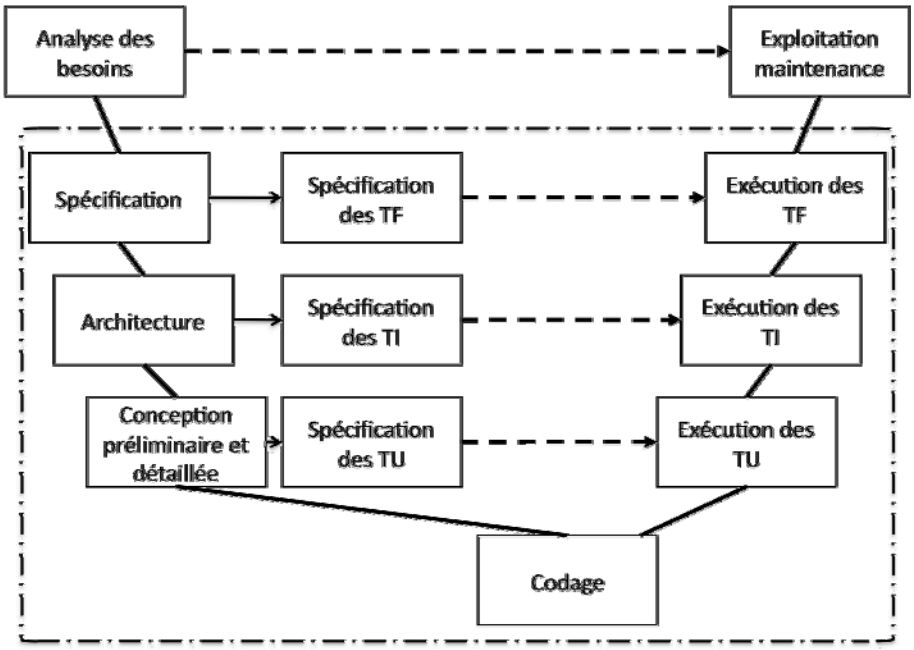

Figure 3.19 : *Cycle de développement en V*

En effet, vérifier manuellement la cohérence des domaines fonctionnels des données manipulées par le logiciel à partir des valeurs fournies par les niveaux système et sous-système est à la fois laborieux, complexe et aisément source d'erreurs, cela à cause de la multiplicité des productions et consommations potentielles d'une donnée et des dépendances de données qui peuvent remettre en cause, suite à une modification par exemple, tous les domaines établis.

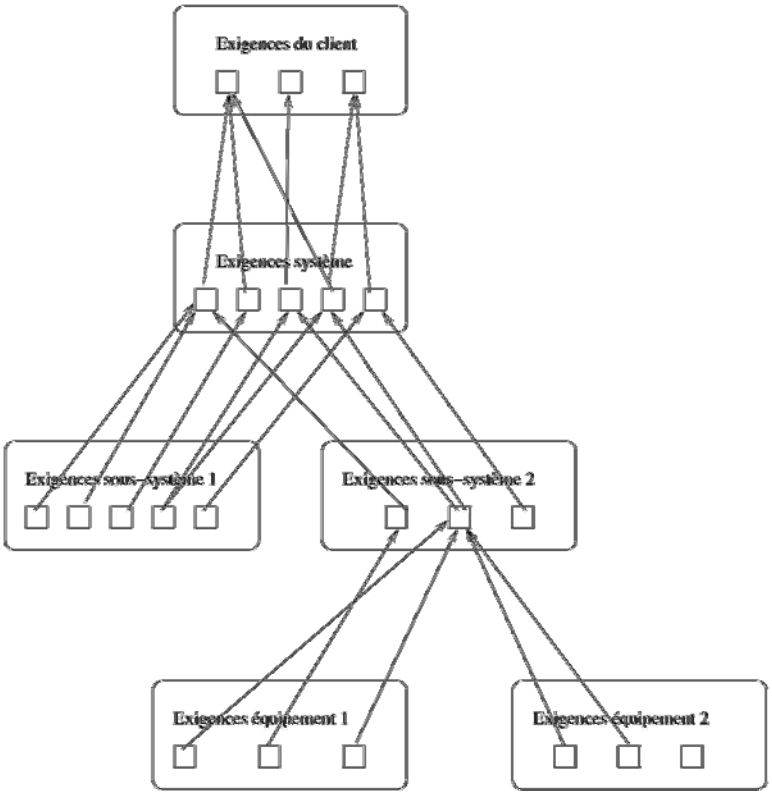

Figure 3.20 : *Traçabilité verticale*

Relativement à la traçabilité horizontale (voir la Figure 3.21), cette application a également eu pour but de montrer, statiquement, l'atteinte partielle ou complète des objectifs de :

- HR-1 (couverture des instructions) : En effet, la méthodologie en utilisant PolySpace détecte le code non atteignable et classifie cela comme une non-conformité.

- HR-2 (intégration progressive des modules du logiciel) : l'application de la méthodologie établit, ou non, que les bornes des domaines des sorties des modules peuvent être des valeurs produites par les modules.

- HR-3 (typage fort) : l'application de la méthode montre, ou non, que l'emploi de données en dehors de leurs domaines fonctionnels est détecté – par une violation de contrôle.

- HR-4 (tests à partir d'une analyse des valeurs aux limites) : l'application de la méthodologie établit, ou non, que les bornes des entrées des modules sont atteignables et attendues par les modules qui les consomment ; et qu'une entrée qui prend valeur en dehors de son domaine fonctionnel est détectée comme incorrecte par la fonction qui la consomme.

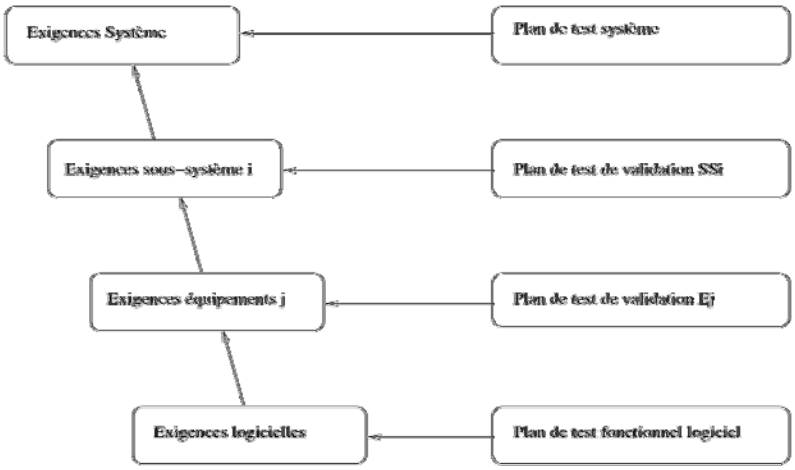

Figure 3.21 : *Traçabilité horizontale*

3.7. Discussion et perspectives

Ce chapitre décrit la vérification du contrôle de valeur posé dans le code source d'une application pour assurer la robustesse du logiciel vis-à-vis de valeurs dysfonctionnelles. Ce contrôle des valeurs traite les défaillances aléatoires (environnement perturbé, défaillance matérielle ...), mais prend aussi en compte les défaillances systématiques (bug dans l'application).

La pose du contrôle effectif est réalisée au cours du développement du logiciel, mais chaque modification du code source ou de son API entraînant des modifications du contrôle, la cohérence entre contrôle et application est difficile à assurer. Cette cohérence doit donc être assurée tout au long du développement et pas uniquement en phase de vérification. La méthode que nous proposons permet de vérifier cela de manière semi-automatique.

Notre méthode peut être étendue à la pose du contrôle car le « contrôle requis » calculé en préliminaire à la vérification du contrôle est une méthode générale. Dans ce cas, le code original ne contient pas de point de contrôle et l'étape de vérification finale est remplacée par une étape de pose des points effectifs à partir des points requis : chaque point requis est implanté en un ou plusieurs points effectifs suivant la stratégie choisie.

Une grande partie des opérations manuelles nécessaires au calcul du contrôle requis sont difficiles à réaliser sans erreurs sur un logiciel de taille et de complexité industrielles. La description section 3.5 (puis appliquées section 3.6) fait apparaître clairement les possibilités d'automatisation : les entrées des fonctions peuvent être calculées par une analyse inter procédurale connue sous le nom de calcul *in-out*, de même on peut calculer les entrées du logiciel, ou les lieux de production et de consommation en utilisant la notion de chaîne *def-use* sur les entrées. De même, l'instrumentation nécessaire au calcul du contrôle requis peut être générée automatiquement à partir des domaines fonctionnels. Finalement, les points requis pourraient être entièrement calculés automatiquement en combinant les résultats obtenus.

Il faut remarquer qu'une grande partie de ces calculs est faite de façon intermédiaire car utile à la vérification de l'absence des erreurs d'exécution. Ces informations ne sont pourtant pas accessibles aux utilisateurs pour l'instant. Ceci est vrai pour PolySpace, ainsi que pour nombre d'outils d'analyse statique qui "connaissent" beaucoup de choses sur le code qu'ils exécutent symboliquement, mais ne traduisent pas ces informations pour l'utilisateur. A notre connaissance, aucun outil d'analyse statique ne nous aurait permis d'automatiser tous les calculs que nous avons réalisés manuellement.

La vérification ou la pose automatique des points de contrôle est elle beaucoup plus difficile à automatiser du fait des différentes formes que peuvent prendre les points de contrôle effectif, du fait de la complexité de mise en œuvre de la stratégie de localisation choisie, mais aussi de la liberté laissée au développeur de choisir le meilleur compromis entre le respect de la stratégie de localisation et la minimisation du nombre de points de contrôle posés.

Pour permettre l'automatisation, il faut définir une règle de localisation plus précise et plus contraignante que celle utilisée actuellement, à savoir "dans la fonction qui consomme et au plus tard entre production et consommation". Une fois cette règle précisée, la pose des points de contrôle par instrumentation du logiciel original ou leur vérification par analyse du logiciel peut être entièrement automatisée. Nous étudions cette automatisation dans une plateforme d'analyse statique offrant les fonctionnalités de base et autorisant le développement de nouveaux modules.

3.8 Conclusion

Le travail de vérification de la robustesse de l'application industrielle PING aurait été impossible à réaliser à la main. En particulier le calcul des domaines fonctionnels inconnus ne peut être fait à la main de manière précise et exacte.

Nous avons montré qu'il est possible d'utiliser l'analyse statique de programme pour rendre cette vérification possible. Les outils d'analyse statique basés sur l'interprétation abstraite de programme sont en général utilisés pour démontrer l'absence d'erreur d'exécution, mémoire ou numérique et réalisent pour cela le calcul de valeurs abstraites. Nous avons utilisé cette fonctionnalité pour propager les domaines fonctionnels des données d'entrées du logiciel jusqu'aux entrées internes permettant d'élaborer les sorties. De cette manière la cohérence entre le code source et les domaines fonctionnels des entrées est prouvée automatiquement: si aucune erreur d'exécution ou aucune instruction non exécutable n'est trouvée, la cohérence est assurée automatiquement. De plus, nous avons montré qu'il est possible de spécifier l'ensemble des points de contrôle assurant la robustesse d'un logiciel. Ceci permet de définir une méthode de vérification : si un point de contrôle requis n'est pas ou est mal implanté dans le code source du logiciel, alors sa robustesse n'est pas assurée.

Ces deux activités ont permis comme attendu de démontrer la robustesse du logiciel industriel en conformité avec la norme CENELEC EN 50128 [CEN 01a].

Pour limiter le temps de mise en œuvre manuel, on peut utiliser la méthode au cours du développement en positionnant l'instrumentation (désactivé par défaut) en même temps que le contrôle et en opérant la vérification au fur et à mesure de la construction de l'application. Il est alors possible de faire la vérification du contrôle de façon incrémentale en conservant les résultats successifs de l'audit. Il restera une vérification finale à opérer par un audit de l'instrumentation nécessaire à la vérification, et un audit différentiel par rapport aux résultats préalables.

Un des points fort de la méthode proposée est qu'elle a pu être réalisée par des personnes n'ayant pas une connaissance fonctionnelle de l'application puisque les informations pertinentes ont été extraites automatiquement du code source de l'application.

Un autre point fort de notre méthode est de présenter un type d'utilisation complexe (non presse bouton) des outils d'analyse statique très peu illustré dans la littérature. Ce type d'utilisation prendra de plus en plus d'importance dans le futur car il apporte à la fois plus de confiance dans les résultats puisqu'une partie des informations est calculée automatiquement, et plus de confiance dans la méthode puisqu'elle peut elle-même être auditée.

Notre méthode suppose l'utilisation d'un outil d'analyse statique. Tout outil ou combinaison d'outils d'analyse statique possédant les fonctionnalités listées section 3.5 peut être utilisée. Nous avons utilisé PolySpace pour vérifier une application ferroviaire et projetons de réaliser la même vérification avec un ou plusieurs outils tels qu'Astrée [ASTREE] ou Frama-C [FRAMA-C] pour comparer la précision des résultats.

3.9. Bibliographie

[BOU 09] BOULANGER J.L., Sécurisation des architectures informatiques – exemples concret, Lavoisier 2009.

[CEN 00] CENELEC, NF EN 50126, Applications Ferroviaires. Spécification et démonstration de la fiabilité, de la disponibilité, de la maintenabilité et de la sécurité (FMDS), janvier 2000.

[CEN 01a] CENELEC, NF EN 50128, Applications Ferroviaires. Système de signalisation, de télécommunication et de traitement – Logiciel pour système de commande et de protection ferroviaire, juillet 2001.

[CEN 01b] CENELEC, EN 50159-1, Norme européenne, Applications aux Chemins de fer : Systèmes de signalisation, de télécommunication et de traitement - Partie 1 : communication de sécurité sur des systèmes de transmission fermés, mars 2001.

[CEN 01c] CENELEC, EN 50159-2, Norme européenne, Applications aux Chemins de fer : Systèmes de signalisation, de télécommunication et de traitement - Partie 2 : communication de sécurité sur des systèmes de transmission ouverts, mars 2001.

[CEN 03] CENNELEC, NF EN 50129, Norme européenne, Applications ferroviaires : systèmes de signalisation, de télécommunications et de traitement systèmes électroniques de sécurité pour la signalisation, 2003.

[FAU 09] Christèle Faure, *Computer Aided Extrinsic Robustness Verification.* Extended Abstract. SAFA Annual Workshop on Formal Techniques. 2009.

[IEC 98] IEC, IEC 61508 : Sécurité fonctionnelle des systèmes électriques électroniques programmables relatifs à la sécurité, Norme internationale, 1998.

[ISO 09] ISO, ISO/CD-26262, Road vehicles – Functional safety, non publié.

[ISO 90] Programming languages - C. International standard ISO/EIC9899:1990 (E).

[ASTREE] Astrée, http://www.absint.com/astree/index_fr.htm

[POLYSPACE] PolySpace, http://www.mathworks.com/products/polyspace

[FRAMAC] Frama C, http://frama-c.cea.fr

[COU 77] Patrick Cousot & Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238--252, Los Angeles, California, 1977. ACM Press, New York.

3.10. Glossaire

API	:	Application Programming Interface
CENELEC[5]	:	Comité Européen de Normalisation ELECtrotechnique
E/E/PES	:	électriques/électroniques/électroniques programmables
IEC[6]	:	International Electrotechnical Commission
ISA	:	Independant Safety Assessor
IPT	:	Points d'inspection
METEOR	:	METro Est Ouest Rapide
NF	:	Norme Française
PING	:	Poste Informatisé Nouvelle Génération
RTE	:	Runtime Error

5 Voir le site : http://www.cenelec.eu/Cenelec/Homepage.htm
6 Voir le site : http://www.iec.ch/

SACEM	*:*	Système d'Aide à la Conduite, à l'Exploitation et à la Maintenance
SAET	*:*	Système d'Automatisation de l'Exploitation des Trains
SIL	*:*	Safety Integrity Level
SSIL	*:*	Software Safety Integrity Level
TVM	*:*	Train Vital Management