SAFE RIVER
Safety & Security Forge

Sûreté de Fonctionnement Logiciel & Analyse de Code

CAP'TRONIC et S2E2

Atelier Technique « Sûreté de Fonctionnement et Sécurité Fonctionnelle des Systèmes Electroniques »

ENSI de Bourges - 12 avril 2012

Mélina Brunet

SAFE RIVER
Safety & Security Forge

Plan de la présentation

- 1 / Recommandations normatives
- 2 / Métriques
- 3 / Règles de codage
- 4 / Analyse statique de code

Partie 1 :

RECOMMANDATIONS NORMATIVES

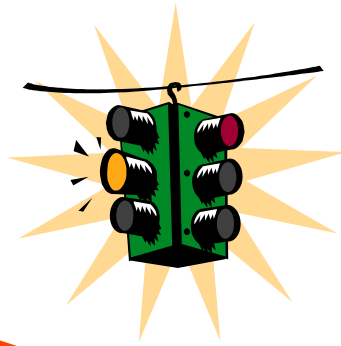
Code et normes de SdF Logiciel

- **Les normes de Sûreté de Fonctionnement Logiciel émettent des nombreuses contraintes, directes ou indirectes, sur le développement du code**

règles de codage

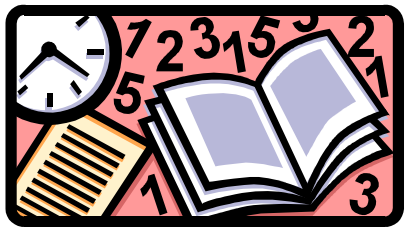
COTS

choix du langage de programmation



taille et complexité du code

identification des codes réutilisés



commentaires

prescriptions de sécurité

traçabilité

déterminisme

gestion de configuration du code

choix de la chaîne de développement

tolérance aux fautes

robustesse

Spécificités et convergences dans la nature des recommandations

Ferroviaire

- EN 50128

Avionique / Contrôle aérien

- DO 178C / DO 278A

Automobile

- ISO 26262-6

Générique (équipements électroniques tous domaines)

- CEI 61508-3

Nucléaire

- CEI 60880 / CEI 61513

Médical

- CEI 60601 -1-4, ISO 62034

Processus industriels, machines

- CEI 61511, ISO 13849, CEI 62061

Evolution des langages recommandés…

Table A.15 – Programming Languages (D.4)
Referenced by clause 10

TECHNIQUE/MEASURE	Ref	SWS IL0	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. ADA	B.62	R	HR	HR	R	R
2. MODULA-2	B.62	R	HR	HR	R	R
3. PASCAL	B.62	R	HR	HR	R	R
4. Fortran 77	B.62	R	R	R	R	R
5. 'C' or C++ (unrestricted)	B.62	R	-	-	NR	NR
6. Subset of C or C++ with coding standards	B.62 B.38	R	R	R	R	R
7. PL/M	B.62	R	R	R	NR	NR
8. BASIC	B.62	R	NR	NR	NR	NR
9. Assembler	B.62	R	R	R	-	-
10. Ladder Diagrams	B.62	R	R	R	R	R
11. Functional Blocks	B.62	R	R	R	R	R
12. Statement List	B.62	R	R	R	R	R

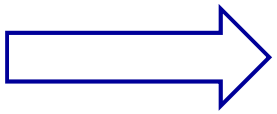
Requirements

1. At Software Safety Integrity Level 3 and 4 when a subset of languages 1, 2, 3 and 4 are used the recommendation changes to HR.
2. For certain applications the languages 7 and 9 may be the only ones available. At Software Safety Integrity Level 3 and 4 where a Highly recommended option is not available it is strongly recommended that to raise the recommendation to 'R' there should be a subset of these languages and that there should be a precise set of coding standards.
3. For information on assessing the suitability of a programming language see entry in the bibliography for 'Suitable Programming Language', B.62.
4. If a specific language is not in the table, it is not automatically excluded. It should, however, conform to B.62.

Les normes s'adaptent aux pratiques de programmation.

Exemple : « Génération automatique de code » traitée explicitement dans l'ISO 26262-6 car le développement à base de modèles est une pratique courante dans l'automobile.

EN 50128:2001



EN 50128:2011

Tableau A.14 – Langages de programmation textuels

TECHNIQUE/MESURE	Réf	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. ADA	D.62	R	HR	HR	HR	HR
2. MODULA-2	D.62	R	HR	HR	HR	HR
3. PASCAL	D.62	R	HR	HR	HR	HR
4. C ou C++	D.62 D.38	R	R	R	R	R
5. PL/M	D.62	R	R	R	NR	NR
6. BASIC	D.62	R	NR	NR	NR	NR
7. Assembleur	D.62	R	R	R	-	-
8. C#	D.62 D.38	R	R	R	R	R
9. JAVA	D.62 D.38	R	R	R	R	R
10. Liste d'instructions	D.62	R	R	R	R	R

SAFE RIVER
Safety & Security Forge

Mais les contraintes demandées sur les langages restent les mêmes !

- **Caractéristiques ou techniques utiles pour la sûreté :**
  - Langage largement utilisé plutôt que langage spécialisé
  - Orienté utilisateur plutôt que machine : l'assembleur doit être limité au logiciel bas-niveau qui gère directement le matériel, les interruptions…
  - Typage fort / vérification à l'exécution des types
  - Assertions
  - Programmation défensive…

  Les caractéristiques qui sont insuffisamment couvertes par le langage choisi doivent être renforcées via des règles de codage.

- **Caractéristiques ou techniques à éviter :**
  - Branchements inconditionnels
  - Entrées ou sorties multiples de boucles ou de fonctions
  - Récursivité
  - Pointeurs, collections ou tout autre type d'objet ou de variable dynamique
  - Initialisation ou déclaration implicite de variables
  - Conversions implicites de types…

SAFE RIVER
Safety & Security Forge

Objectifs de ces contraintes

Table 1 — Topics to be covered by modelling and coding guidelines

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity	++	++	++	++
1b	Use of language subsets[b]	++	++	++	++
1c	Enforcement of strong typing[c]	++	++	++	++
1d	Use of defensive implementation techniques	o	+	++	++
1e	Use of established design principles	+	+	+	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++

[a] An appropriate compromise of this method with other methods in ISO 26262-6 may be required.

[b] The objectives of method 1b are

- Exclusion of ambiguously defined language constructs which might be interpreted differently by different modellers, programmers, code generators or compilers.
- Exclusion of language constructs which from experience easily lead to mistakes, for example assignments in conditions or identical naming of local and global variables.
- Exclusion of language constructs which might result in unhandled run-time errors.

[c] The objective of method 1c is to impose principles of strong typing where these are not inherent in the language.

Automobile - ISO 26262

Techniques recommandées / Techniques déconseillées

Table A.3 – Software Architecture (clause 9)

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1.	Defensive Programming	B.15	-	R	R	HR	HR
2.	Fault Detection & Diagnosis	B.27	-	R	R	HR	HR
3.	Error Correcting Codes	B.20	-	-	-	-	-
4.	Error Detecting Codes	B.20	-	R	R	HR	HR
5.	Failure Assertion Programming	B.25	-	R	R	HR	HR
6.	Safety Bag Techniques	B.54	-	R	R	R	R
7.	Diverse Programming	B.17	-	R	R	HR	HR
8.	Recovery Block	B.50	-	R	R	R	R
9.	Backward Recovery	B.5	-	NR	NR	NR	NR
10.	Forward Recovery	B.32	-	NR	NR	NR	NR
11.	Re-try Fault Recovery Mechanisms	B.53	-	R	R	R	R

Ferroviaire – EN 50128

Techniques recommandées / Techniques déconseillées

12.	Memorising Executed Cases	B.39	-	R	R	HR	HR
13.	Artificial Intelligence - Fault Correction	B.1	-	NR	NR	NR	NR
14.	Dynamic Reconfiguration of software	B.18	-	NR	NR	NR	NR
15.	Software Error Effect Analysis	B.26	-	R	R	HR	HR
16.	Fault Tree Analysis	B.28	R	R	R	HR	HR

Requirements

1. Approved combinations of techniques for Software Safety Integrity Levels 3 and 4 shall be as follows:
   a) 1, 7 and one from 4, 5 or 12
   b) 1, 4 and 5
   c) 1, 4 and 12
   d) 1, 2 and 4
   e) 1 and 4, and one of 15 and 16

2. With the exception of entries 3, 9, 10, 13 and 14, one or more of these techniques shall be selected to satisfy the requirements for Software Safety Integrity Levels 1 and 2.

3. Some of these issues may be defined at the system level.

4. Error correcting codes may be used in accordance with the requirements of EN 50159-1 and EN 50159-2.

Ferroviaire – EN 50128

Les techniques « reines » de la sûreté de fonctionnement logiciel

- **Programmation défensive**
  - Initialisation restrictive
  - Tests de vraisemblance
  - Tests de bornes
  - Règles d'allocation / désallocation mémoire
  - Restrictions sur les pointeurs
  - Compteurs afin d'éviter les boucles infinies
  - …
- **Utilisation de codes détecteurs d'erreurs (CRC, MD5…)**
- **Détection des fautes**
  - Watchdogs, tests en ligne, contrôle de séquencement applicatif…

Choix des techniques vis à vis de la stratégie de gestion des erreurs

- **Un logiciel ne peut être sûr dans toutes les conditions d'utilisation (environnement, actions utilisateurs…)**
- **Choix des hypothèses de défaillance retenues**
- **Identifier les causes possibles d'erreur**
  - Erreurs du matériel => autotests, tests en ligne, traitement des erreurs
  - Erreurs du logiciel => programmation diversifiée ou blocs de recouvrement
  - Erreurs de données => tests de vraisemblance, codes détecteurs d'erreur
  - Erreurs temporelles – ordonnancement => watchdog, masquage des interruptions non utilisées, moniteur des tâches, gestion des exceptions, Worst Case Execution Time…
  - Erreurs temporelles – réseau / acquisition des entrées / synchronisation des différentes unités de traitement dans une architecture redondée => datation des données (time-stamp)…

L'Analyse statique de code : pilier du respect de ces caractéristiques

Ferroviaire - EN 50128

- Analyse statique - HR (Hautement Recommandé) des niveaux SSIL 1 à SSIL 4
- Vérification statique par interprétation abstraite (ajouté explicitement en version 2011)

Tableau A.18 – Analyse statique

TECHNIQUE/MESURE	Réf	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Analyse des valeurs aux limites	D.4	-	R	R	HR	HR
2. Listes de contrôle	D.8	-	R	R	R	R
3. Analyse de flux de commande	D.9	-	HR	HR	HR	HR
4. Analyse du flux de données	D.11	-	HR	HR	HR	HR
5. Supposition d'erreurs	D.21	-	R	R	R	R
6. Inspection de Fagan	D.24	-	R	R	HR	HR
7. Analyse des chemins insidieux	D.55	-	-	-	R	R
8. Exécution symbolique	D.63	-	R	R	HR	HR
9. Revues/Examens de la conception	D.66	HR	HR	HR	HR	HR
10. Vérification statique par interprétation abstraite	D.69	-	R	R	HR	HR

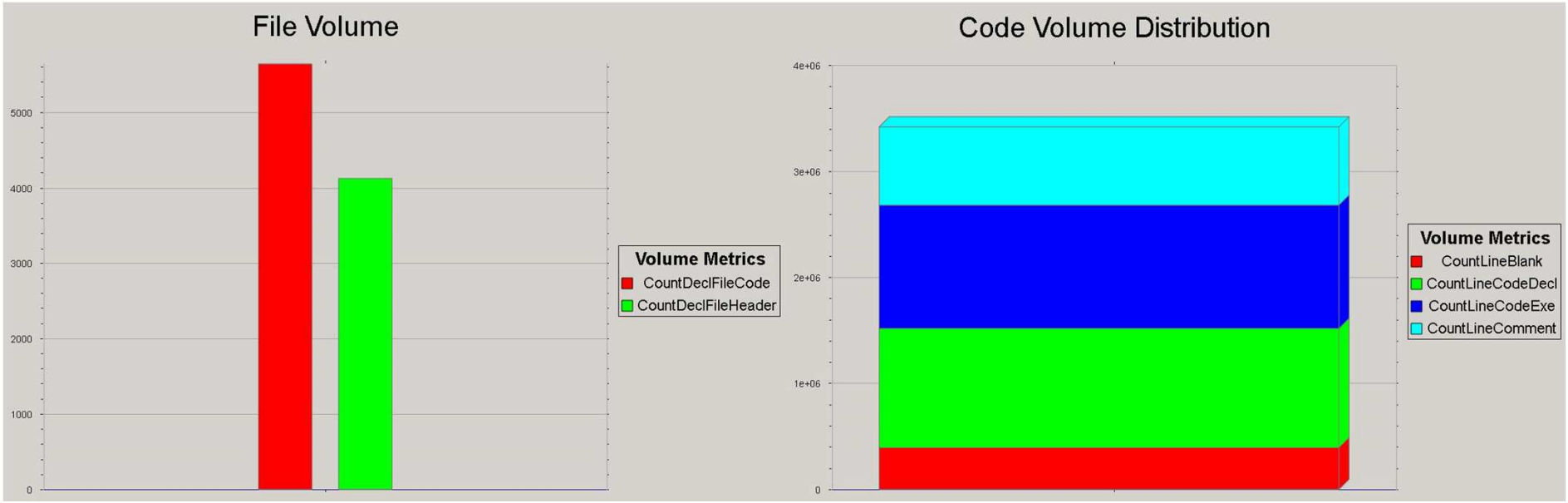
SAFE RIVER
Safety & Security Forge

Partie 2 :

METRIQUES

Métriques – niveau code global

- **Nombre de lignes de code**
- **Répartition des fichiers (.h / .c)**
- **Pourcentage de commentaires**



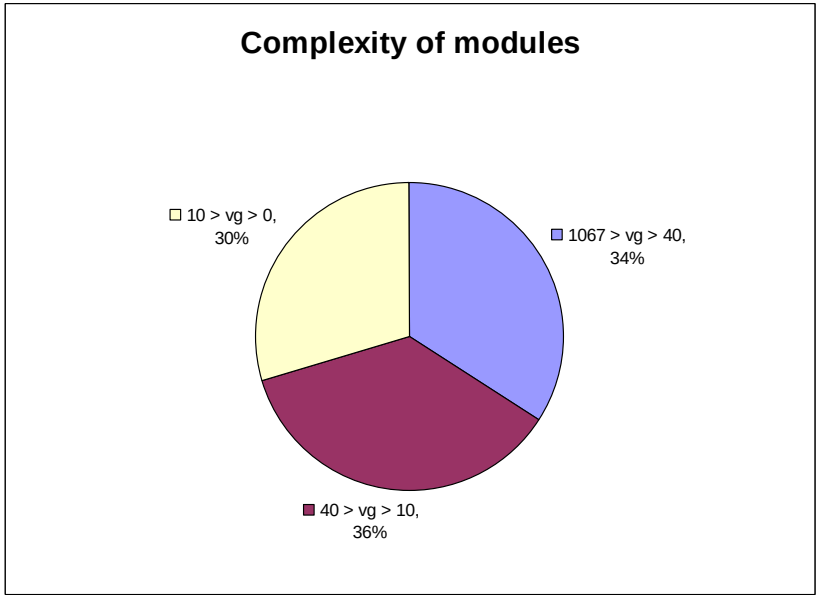
Métriques – niveau fonction

- Nombre de fonctions appelées
- Niveau d'imbrication des appels
- Nombre de « goto »
- Nombre de « return »
- Nombre de chemins
- Complexité cyclomatique (vg) - McCabe

Métriques de complexité

- **Les fonctions avec haute complexité induisent**
  - Plus de probabilité de contenir une erreur
  - Plus de difficulté de détecter ces erreurs lors d'une revue de pair
  - Plus d'efforts de tests car il y a davantage de chemins à couvrir
- **Faible complexité : vg < 10**
- **Haute complexité: vg >= 10**

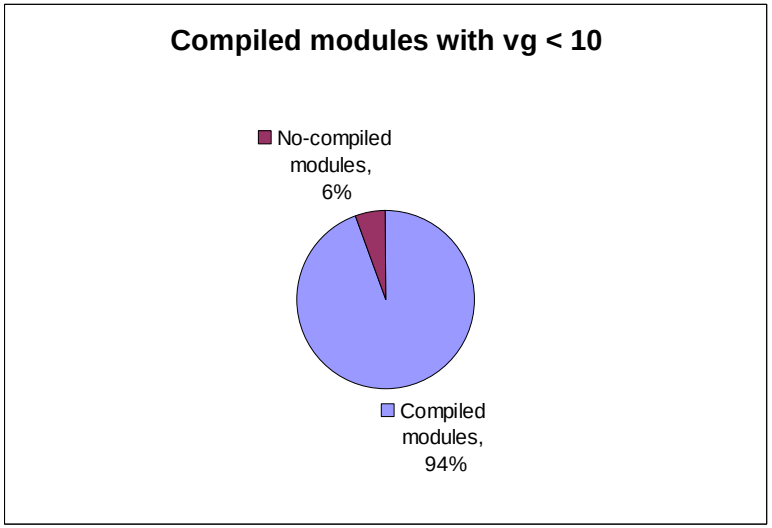
vg min	vg max	number of modules
0	10	53
10	40	64
40	1067	61

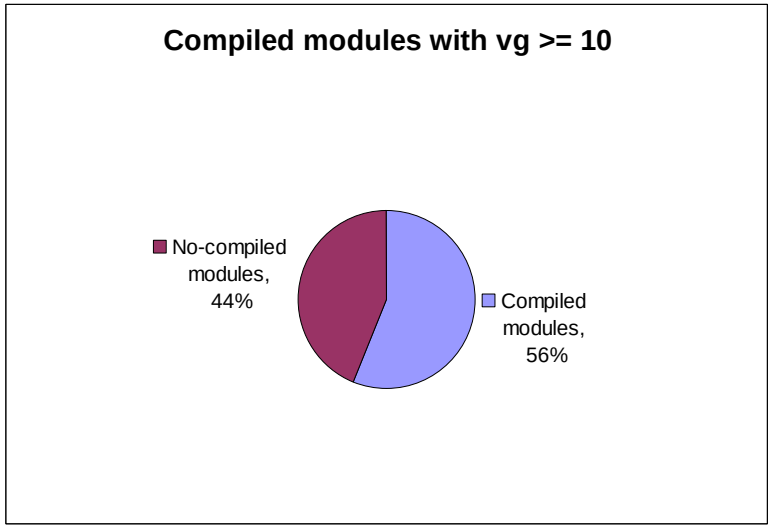


Exemple de résultats obtenus sur une application réelle

Complexité et analyse statique

Exemple :





- **94% des modules de faibles complexité sont analysables**
- **56% des modules de forte complexité sont analysables**

➔ **Les modules complexes sont plus difficile à analyser avec les outils, ce qui diminue également la probabilité de détection d'une erreur.**

Exemple de résultats obtenus sur une application réelle

SAFE RIVER
Safety & Security Forge

Partie 3 :

RÈGLES DE CODAGE

SAFE RIVER
Safety & Security Forge

Pré-requis

- **Sources d'insécurité dans le code**
  - Erreur du programmeur (de la simple erreur dans le nom d'une variable à la mauvaise implémentation d'un algorithme)
  - Introduction volontaire de code malveillant
  - Utilisation de COTS non sûrs (en cas d'utilisation de fonctions de la lib C, certaines fonctions sont à préférer à d'autres : strncpy est préféré à strcpy – prise en compte de la taille de la chaîne à copier)
  - Mauvaise compréhension d'une construction du langage par le programmeur : le compilateur ne fait pas ce que le programmeur avait prévu (notamment sur les règles de précédence des opérateurs)
  - Erreur du compilateur ou déviation non identifiée par rapport au standard du langage (l'ISO 9899:1990 est un standard de référence pour le C)
  - Erreurs d'exécution (run-time errors) : divisions par zéro, boucles infinies, buffer overflow…

Pré-requis

- **Des règles de codage sont explicitement demandées par les normes de sûreté de fonctionnement logiciel**
- **Elles doivent être mises en place pour couvrir les sources d'insécurité**
  - Faciliter la détection d'erreurs du programmeur (un code lisible permet de détecter et corriger plus facilement le code erroné ou malveillant)
  - Limiter le risque introduction d'erreurs, notamment en limitant les constructions autorisées (fixer un sous-ensemble applicable du langage, comme recommandé par les normes de sûreté de fonctionnement logiciel)
  - Fixer des « pattern » de codage sûr, limitant le risque de survenue d'erreurs d'exécution.

Définir dans les règles de codage :

- les pratiques interdites
- les pratiques déconseillées
- les pratiques obligatoires
- le processus à suivre en cas de dérogation à la règle

Types de règles de codage

- **Règles de style**
  - Présentation du code (indentation, accolades, présentation de l'entête de la fonction et des commentaires, date, auteur, adjonction d'un copyright…)
    - Remarque : généralement, dans un code destiné à un produit sûr de fonctionnement, il y a autant de lignes de code que de lignes de commentaires.
  - Règles de nommage (taille des noms, séparation des mots par un underscore ou une alternance majuscules/minuscules…)
    - Remarque : souvent le nombre maximal de caractères significatifs autorisés dans un identifiant est limité à 31, ce qui correspond à une limitation de nombreux compilateurs.
- **Règles de sûreté de fonctionnement ou de sécurité**
  - Règles internes à l'entreprise basées sur le retour d'expérience ou règles génériques issues de standards du métier (comme les règles Misra-C:2004 dans l'industrie automobile)

Faciliter la détection d'erreurs (1/2)

- Tous noms de variables et de fonctions doivent différer de plus de deux caractères : pour éviter qu'une erreur sur un nom de variable soit non détectée car on aboutit au nom d'une autre variable valide.
- Dans les conditions, écrire la valeur à gauche et la variable à droite : permet de détecter l'utilisation d'un symbole d'affectation (=) à la place du symbole de comparaison logique (==)
- Les paramètres doivent être les mêmes dans le prototype d'une fonction (déclaration) et dans sa définition (Misra 8.3)
- Les variables, notamment les codes retours des fonctions, doivent être initialisées à des valeurs restrictives lorsque cela est possible.
- Les codes retours des fonctions doivent être testés.
- Toute boucle d'attente doit être contrôlée par un timeout.

Faciliter la détection d'erreurs (2/2)

TYPAGE DES VARIABLES

- Le type défini doit convenir aux valeurs à stocker dans la variable.

- Les conversions de type implicites sont interdites.
- Les conversions de type explicites (cast) aboutissant à une perte d'information doivent être justifiés.
  - **Perte de valeur : valeur stockée dans un type trop petit pour la contenir**
  - **Perte de signe : valeur signée stockée dans un type non signé**
  - **Perte de précision : valeur flottante stockée dans un type entier**

char	**signed**	**-128**	**127**
	unsigned	**0**	**255**
short	**signed**	**-32 768**	**32 767**
	unsigned	**0**	**65 535**
long	**signed**	**-2^31**	**2^31 -1**
	Unsigned	**0**	**2^32**

Limiter l'introduction d'erreurs (1/2)

- La construction « condition?a:b » est déconseillée.
- Les opérateurs d'incrémentation (++) et de décrémentation (--) ne doivent pas être mélangés avec d'autres opérateurs dans une expression (règle Misra 12.13)
- Les opérateurs # et ## sont déconseillés (règle Misra 19.13)

- Les constantes octales sont interdites (règle Misra 7.1)

```
code[1] = 109;   /* équivalent à 109 en décimal */
code[2] = 100;   /* équivalent à 100 en décimal */
code[3] = 052;   /* équivalent à 42 en décimal */
code[4] = 071;   /* équivalent à 57 en décimal */
```

- Ne pas mélanger de valeurs décimales et hexadécimales dans le même calcul

Limiter l'introduction d'erreurs (2/2)

- Initialiser les variables à leur définition
- Ne pas laisser dans le code de variable inutilisée
- Limiter le nombre de variables globales
- Accéder aux cases des tableaux par des variables de type non signé, comme un unsigned char.
- Identifier les variables partagées entre plusieurs tâches du programme pour les protéger des accès concurrents.
- Tout 'case' doit se terminer par un 'break' inconditionnel.
- Tout 'switch' doit se terminer par un cas 'default', comportant un traitement de robustesse approprié (défini par la stratégie dysfonctionnelle choisie).

Document de règles de codage

- **Les règles de codage doivent être :**
  - adaptées au projet,
  - connues et maîtrisées par tous les développeurs participant du projet

- **Règles Misra-C (version 2004) : règles issues du domaine de l'automobile, mais largement adoptées dans d'autres domaines**
  - Sélection des règles applicables
  - Ajout de règles spécifiques selon le retour d'expérience de l'équipe

SAFE RIVER
Safety & Security Forge

Partie 4 :

ANALYSE STATIQUE DE CODE

SAFE RIVER
Safety & Security Forge

Compilateur : un premier niveau d'analyse statique

- **Souvent le compilateur fournit un premier niveau d'analyse du code, selon les options activées.**
- **Exemple : vérifications avec gcc**
- **Options de base activées par l'option –Wextra (anciennement appelée option –W) :**
  -Wclobbered -Wempty-body -Wignored-qualifiers -Wmissing-field-initializers
  -Wmissing-parameter-type (C only) -Wold-style-declaration (C only)
  -Woverride-init -Wsign-compare -Wtype-limits -Wuninitialized
  -Wunused-parameter (only with -Wunused or -Wall)
- **Options de base activées par l'option –Wall :**
  -Waddress -Warray-bounds (only with -O2) -Wc++0x-compat -Wchar-subscripts
  -Wenum-compare (in C/Objc; this is on by default in C++) -Wimplicit-int
  -Wimplicit-function-declaration -Wcomment -Wformat
  -Wmain (only for C/ObjC and unless -ffreestanding) -Wmissing-braces
  -Wnonnull -Wparentheses -Wpointer-sign -Wreorder -Wreturn-type
  -Wsequence-point -Wsign-compare (only in C++) -Wstrict-aliasing
  -Wstrict-overflow=1 -Wswitch -Wtrigraphs -Wuninitialized
  -Wunknown-pragmas -Wunused-function -Wunused-label
  -Wunused-value -Wunused-variable -Wvolatile-register-var
- **D'autres options intéressantes peuvent être activées comme : -Wswitch-default (qui indique quand un switch n'a pas de cas default) et -Wswitch-enum (qui indique quand il manque un cas dans un switch sur une variable qui prend les valeurs d'un enum).**
- **-Werror permet de transformer les warnings levés en erreurs et donc de forcer leur correction pour permettre la compilation du code.**

Différents niveaux d'analyse statique de code

Il n'y a pas d'exécution du code pour l'analyse statique.

Entrée de l'analyse statique : code source ou code compilé (objets ou exécutable)

- **1 / Analyse lexicale : utilisation d'expression régulières (regexp) pour rechercher des patterns de code**
- **2 / Parsing : utilisation d'une grammaire, spécifique au langage traité (décrit ses symboles), pour établir un arbre (parse tree)**
- **3 / Représentation interne : l'outil d'analyse statique doit se construire une représentation interne du code en transformant le code en un modèle. Généralement proche du modèle construit par un compilateur.**
- **4 / Analyse des flots de contrôle (control flow)**
- **5 / Analyse des flots de données (data flow)**
- **6 / Analyse sémantique : calcul des flots d'exécution du code (chemins) – nécessite la propagation de domaines de valeurs (union de domaines).**

Catégories d'outils d'analyse de code

Exemples d'outils d'analyse statiques de code :

- **Outils d'analyse lexicale : RATS, ITS4, CodeScan Labs, Flawfinder, PC-lint…**
- **Outils d'analyse basés sur le parsing et les flots de données et de contrôle : Klocwork, Ounce Labs, QAC, Logiscope…**
- **Outils d'analyse sémantique : Fortify, Coverity, Parasoft, PolySpace TMW, Astree…**

- **La nature des résultats, le temps d'analyse, le prix des licences et la complexité de mise en œuvre et d'interprétation des résultats sont très variables d'un outil à l'autre.**

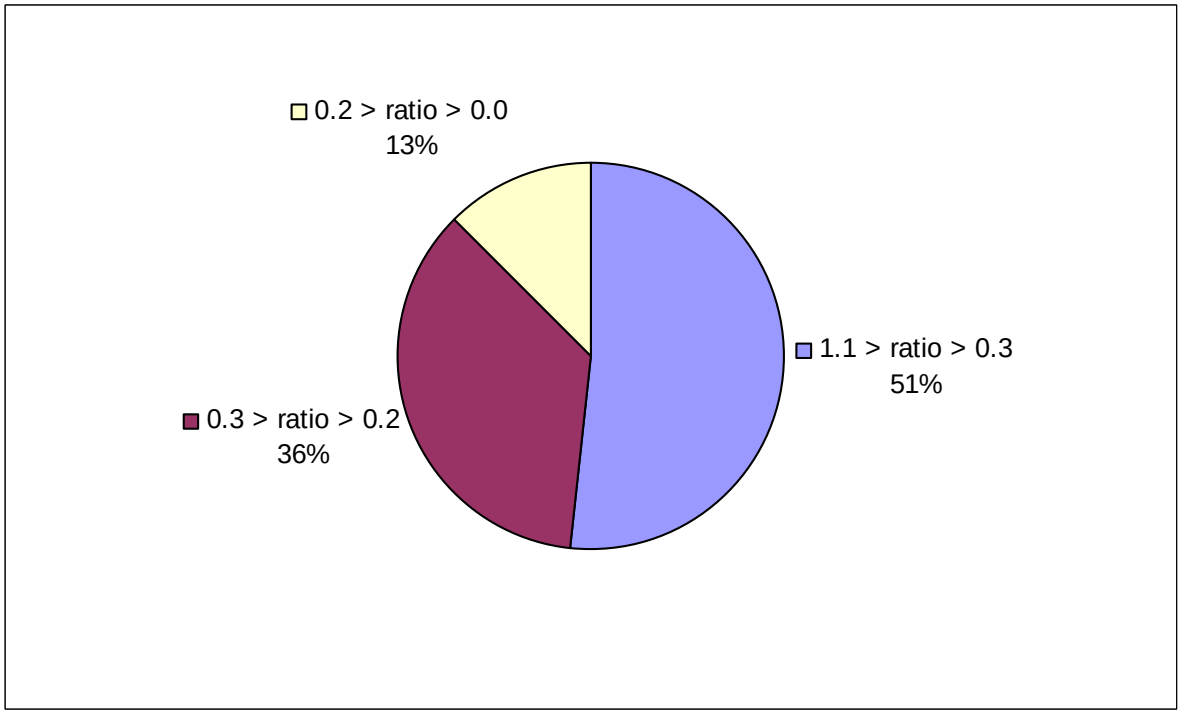
Vérification des règles MISRA C++

- **Le vérificateur MISRA C++ de PolySpace TMW permet de vérifier 167 des 228 règles MISRA C++ (certaines règles portent plus sur les processus que sur le code directement)**
- **Détermination du nombre de règles violées :**
  - Exemple : 129 règles violées (77% des règles détectables) and 240 492 violations
- **Etablissement du TOP 3 des modules avec le plus grand nombre de violations MISRA**

Module	Compiled	Number of lines	Number of violations	Ratio: MISRA / (nb lines)
network	YES	32 179	24 621	0.77
upgrade	YES	20 698	21 713	1.05
comm	YES	56 053	14 805	0.26

Exemple de résultats obtenus sur une application réelle

Ratio nombre de violations / nombre de lignes



- **1.1 > ratio > 0.3 ➔ 49 modules**
- **0.3 > ratio > 0.2 ➔ 34 modules**
- **0.2 > ratio > 0.0 ➔ 12 modules**

Exemple de résultats obtenus sur une application réelle

Utilisation des résultats pour cibler les tests, la relecture ou les reprises

- **TOP 3 des modules avec le plus fort nombre de violations MISRA violations par ligne de code**

Module	Compiled	Number of lines	Number of violations	Ratio: MISRA / (nb lines)
upgrade	YES	20 698	21 713	1.05
network	YES	32 179	24 621	0.77
audio	YES	90	61	0.68

- **TOP 3 des modules avec le plus faible nombre de violations MISRA violations par ligne de code**

Module	Compiled	Number of lines	Number of violations	Ratio: MISRA / (nb lines)
setup	YES	657	17	0.03
nav	YES	29	1	0.03
pilot	YES	5014	190	0.04

La disparité de ratio obtenue peut être très importante

=> d'une violation par ligne à une violation toutes les 20 lignes dans cet exemple

Exemple de résultats obtenus sur une application réelle

Utilisation des résultats pour cibler la formation de l'équipe de développement

- **Etablissement du TOP 3 des règles violées**
  - Rule 3-9-2 (Advisory): typedefs that indicate size and signedness should be used in place of the basic numerical types (46 019)
    - Example: network: `STATUS Init(void);`
  - Rule 5-0-4 (Required): An implicit integral conversion shall not change the signedness of the underlying type (32 543)
    - Example: Comm: `status=semTake(m_semaphore[p_sem_index], p_timeout);` type of `p_timeout` is `const unsigned short` whereas the signature of `semTake` is `STATUS semTake(SEM_ID semId, int timeout)`
  - Rule 0-1-7 (Required): The value returned by a function having a non-void return type that is not an overloaded operator shall always be used (25 056)
    - Example: Comm: `Clean();` whereas the signature of `Clean` is `STATUS Clean()`
- **Remarques : il y a 18 règles conseillées (dont 17 analysées par PolySpace). Selon les projets, on peut choisir de ne pas appliquer les règles conseillées, mais seulement celles qui sont recommandées.**

Exemple de résultats obtenus sur une application réelle

PolySpace TMW: un outil d'analyse par interprétation abstraite

- **Détection d'erreurs d'exécution**
- **Résultats par couleurs**
  - Rouge (!): erreur certaine dans le code
  - Vert (V): code sans erreur d'exécution
  - Gris (X): code non-atteignable (code mort)
  - Orange (?): code non prouvé (peut contenir une erreur)

Category	Acronym
Function Returns a Value	FRV
Non Null This-pointer	NNT
C++ related instructions	CPP
Object Oriented Programming check	OOP
Potential Call to (informative check)	INF
Non-Initialized Variable	NIV / NIVL
Non-Initialized Pointer	NIP
User Assertion Failure	ASRT
Overflows and Underflows	OVFL
Scalar or Float Division by zero	ZDV
Shift amount out of bounds	SHF
Array Index is Outside its Bounds	OBAI
Correctness condition	COR
Pointer is Outside its Bounds (Invalid dereference pointer)	IDP
Exception handling check	EXC
Unreachable Code	UNR
Non Terminations: Calls and Loops	NTC / NTL

- **Les points « rouges » sont toujours suivi de « gris »**
- **Le « vert » est propagé en sortie du « orange »**

Résultats PolySpace

Example: IHM (Polyspace_R2011a_polyspace_01_30_2011-14h08.log)

Checks statistics: (including internal files)

- OVFL => Green : 225, Orange : 4, Red : 0, Gray : 0 (98%)

…

- ASRT => Green : 23, Orange : 11, Red : 0, Gray : 0 (68%)

TOTAL: => Green : 9330, Orange : 1766, Red : 0, Gray : 39 (84%)

Number of NTL : 1

Number of NTC : 0

Number of UNR : 15

Certain (Red) errors summary:

- certain NTL, the loop is infinite or contains a run-time error, File IHM.cpp, line 852, column 5

Exemple de résultats obtenus sur une application réelle

Comment analyser des résultats PolySpace ?

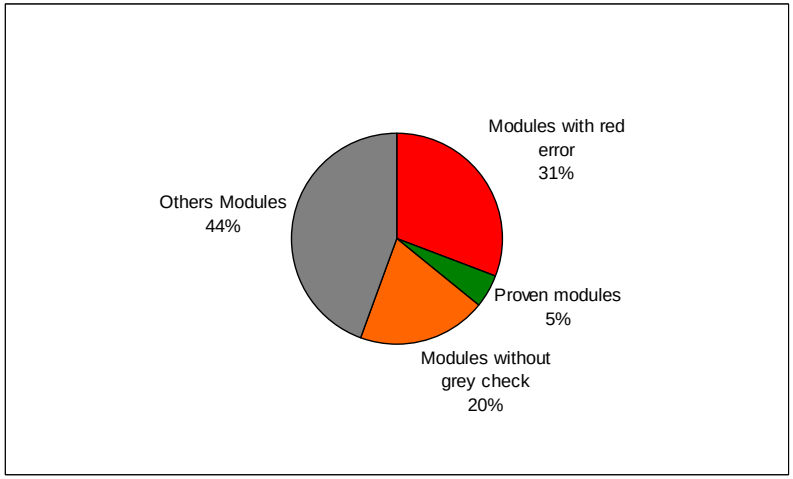
1. **Ordre de revue des points « rouges »**
   - OBAI
   - ASRT
   - NTC (fonctions de librairie également)
   - NIVL
   - NTL (avec un orange dans la boucle)
   - IDP
   - NNT
   - COR
   - NTL
   - NTC

2. **Ordre de revue des points « gris »**
   - Code défensif
   - Code mort lié à une configuration (#define)
   - Code mort suivant un « rouge »
   - Vrai code mort pouvant révéler une erreur de programmation

3. **Revue des points « orange »**
   - Les points orange doivent être analysés seulement après correction des points rouges et gris.

Analyse des résultats par module

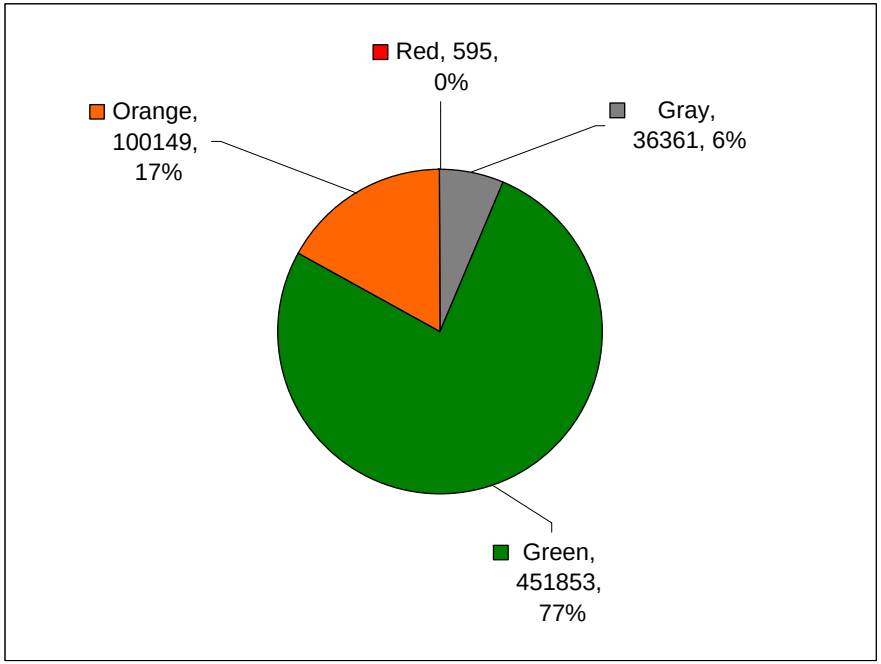
- **L'analyse permet d'identifier :**
  - les modules entièrement prouvés sans erreur d'exécution
  - les modules qui contiennent du « rouge » et du « gris »



- **Utilisé en cours de développement, ces résultats peuvent notamment contribuer à déterminer le degré de maturité des modules (prêts à être intégrés et testés ou à retravailler)**

Exemple de résultats obtenus sur une application réelle

Nombre de « checks »



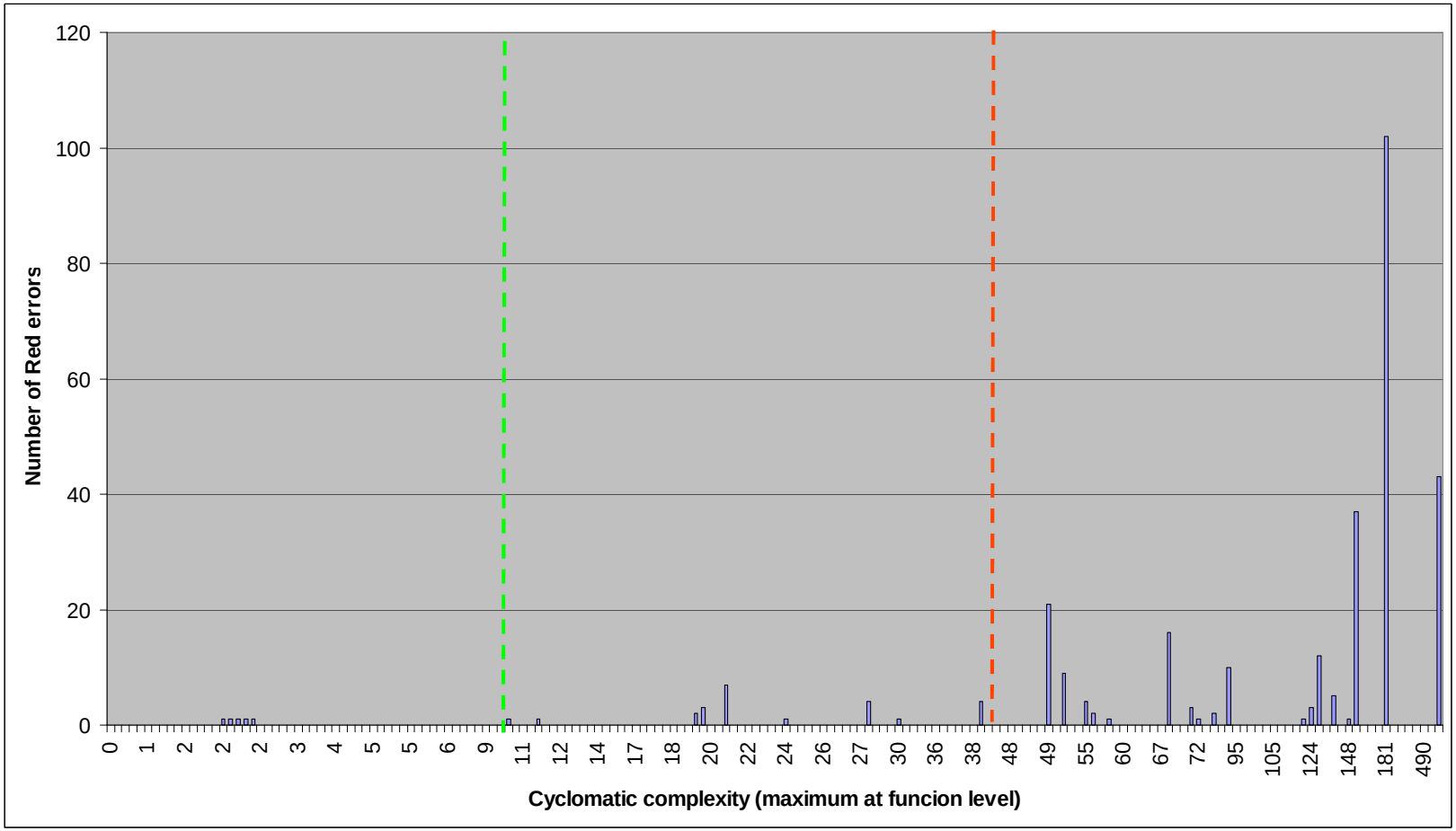
- **Nombre total de points analysés : 588 958**

Green	Orange	Red	NTC	NTL	Gray	UNR
451 853	100 149	298	249	48	27 581	8 780

Exemple de résultats obtenus sur une application réelle

Répartition des erreurs d'exécution certaines par complexité

- **L'expérience montre que les erreurs d'exécution certaines se situent particulièrement dans les modules complexes**



Exemple de résultats obtenus sur une application réelle

Les erreurs détectées dans les modules avec vg < 10 sont dues à des fichiers manquants (fonctions « extern » non fournies)

Répartition des erreurs certaines par catégories

- **443 points « rouges »**
  - 14 OBAI
  - 20 ASRT
  - 46 NTC with library functions
  - 5 NIVL
  - 10 IDP
  - 64 NNT
  - 43 COR
  - 37 NTL (loop with a red error)
  - 204 NTC (function with a red error)

- **Permet de cibler les tests (tests hors limite pour les tableaux, les entrées du logiciel…)**
- **Permet également de cibler la formation des équipes**

Exemple de résultats obtenus sur une application réelle

Exemple de points « rouge » liés à des problèmes de taille (1/4)

- **Certain OBAI, array index is outside its bounds : [0..254],**
- **"my_struct.diag_data[DIAG_REQUEST_LENGTH]" access with DIAG_REQUEST_LENGTH = 255 whereas the size of array (my_struct.diag_data) is DIAG_MESSAGE_SIZE_MAX = 255**

types.h

```
#ifdef SafeRiver_error
  #define DIAG_MESSAGE_SIZE_MAX 256
#else
  #define DIAG_MESSAGE_SIZE_MAX 255
#endif
```

Exemple de résultats obtenus sur une application réelle

Exemple de points « rouge » liés à des problèmes de taille (2/4)

- **Certain NTL, the loop is infinite or contains a run-time error,**
- **"for (int l_count ; l_count < (ANNOUNCEMENT_MAX+1) ; l_count++)" with ANNOUNCEMENT_MAX = 11**
  - info.AnnouncFlag[l_count] access with l_count = 11 whereas the size of (info.AnnouncFlag) is ANNOUNCEMENT_MAX = 11

IHM.cpp

```
#ifdef SafeRiver_error
  for (int l_count = 0 ; l_count < (ANNOUNCEMENT_MAX) ; l_count++)
    …
#else
  for (int l_count = 0 ; l_count < (ANNOUNCEMENT_MAX + 1) ; l_count++)
    …
#endif
```

Exemple de résultats obtenus sur une application réelle

Exemple de points « rouge » liés à des problèmes de taille (3/4)

- **Certain NTC, the called function __polyspace__stdstubs.memcpy__pst_inlined_8 (in the current context) either contains an error or does not terminate,**
- **memcpy((void*)l_header_array,(void*)&(l.headers_data[l_counter].header_data, MAX_HEADER_LENGTH);**
  - Size of (l.headers_data[l_counter].header_data) is smaller than size of (l_header_array) = MAX_HEADER_LENGTH

Interface.cpp

```
#ifdef SafeRiver_error
memcpy((void*)l_header_array,(void*)&(l.headers_data[l_counter]),MAX_HEADER_LENGTH);

#else
memcpy((void*)l_header_array,(void*)&(l.headers_data[l_counter].header_data),MAX_HEADER_LENGTH);

#endif
```

Exemple de résultats obtenus sur une application réelle

Exemple de points « rouge » liés à des problèmes de taille (4/4)

- **Certain NTL, the loop is infinite or contains a run-time error,**
- **For (count = 0 ; count <= 32 ; count++)**

  { unsigned long l_mask_temp = (m_mask_active&((unsigned long) 1 << count)); …}, count shall be less than or equal to 31

Parameter.cpp

```
#ifdef SafeRiver_error
  for (count = 0 ; count <= 32 ; count++)
{ unsigned long l_mask_temp = (m_mask_active&((unsigned long) 1 <<
count)); …}
#else
  for (count = 0 ; count < 32 ; count++)
{ unsigned long l_mask_temp = (m_mask_active&((unsigned long) 1 <<
count)); …}
#endif
```

Exemple de résultats obtenus sur une application réelle

SAFE RIVER
Safety & Security Forge

CONCLUSION

SAFE RIVER
Safety & Security Forge

Une solution à l'échelle industrielle

- **Analyse en fichier par fichier ou analyse de l'application intégrée**
- **Dernière application analysée : 1 120 000 lignes de code (sans les commentaires, sans les fichiers d'entête .h et .hpp)**
  - ~ 990 000 lignes de C++
  - ~ 130 000 lignes de C (ANSI)

Adaptée à l'analyse de codes critiques, comme à l'audit de code

- **L'analyse statique de code, technique recommandée par les normes de SdF, s'impose comme un outil capital pour la démonstration de la sûreté du logiciel**
- **Mais aussi comme un outil d'audit :**
  - Suivi d'une équipe de développement (points forts / points faibles pour cibler les formations)
  - Recette de logiciel sous-traité (critère de mesure objectif de la qualité du code reçu)
  - Recherche de « bug » (investigation de problème terrain dont la cause est indéterminée)

SAFE RIVER
Safety & Security Forge

Fin de la présentation

Questions ?

www.safe-river.com