

Sûreté de fonctionnement pour l'embarqué

Aspects logiciels

Sébastien Faucou & Pierre-Emmanuel Hladik

Une introduction très rapide à la SdF

Les concepts de base

Système

Une entité qui interagit avec d'autres entités, c'est-à-dire d'autres systèmes (logiciel, matériel, humains, etc.)

Fonction

Ce que le système doit faire, décrit en terme de fonctionnalité et de performance

Comportement

Ce que fait le système pour implémenter sa fonction. Décrit comme une succession d'états.

Structure / architecture

Ce qui permet au système d'engendrer son comportement.

Service

Le service fourni par un système correspond à son comportement tel qu'il est observé par un utilisateur.

Pour différencier service et comportement, on va donc distinguer :

- les *états externes* du système : ceux qui sont visibles des utilisateurs;
- et les *états internes*.

Le service est une séquence d'états externes. Le comportement est une séquence d'états internes et externes.

Définition de la SdF

On trouve deux définitions dans la littérature

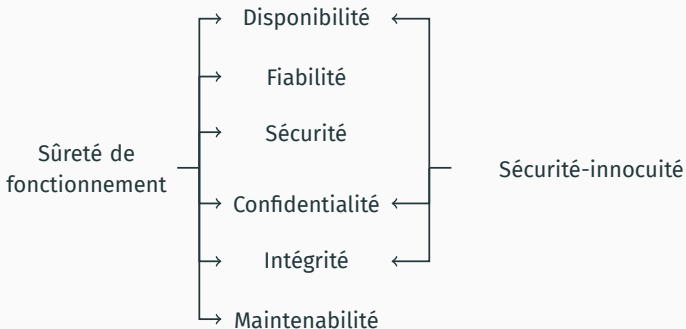
Définition 1 (accent sur la confiance justifiée)

La SdF (*Dependability*) est la propriété d'un système qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre.

Définition 2 (accent sur l'évitement des défaillances)

La SdF est, pour un système, l'aptitude à éviter les défaillances plus fréquentes ou plus sévères que ce qui est acceptable.

Les attributs de la SdF



Les attribut de la SdF

Disponibilité (*Availability*)

Aptitude à fournir le service attendu à un instant donné

Fiabilité (*Reliability*)

Aptitude à fournir le service attendu sur un intervalle de temps

Sécurité (*Safety*)

Absence de conséquences catastrophiques pour l'utilisateur ou l'environnement du système

Confidentialité (*Confidentiality*)

Aptitude à ne pas divulguer des informations sans autorisations

Intégrité (*Integrity*)

Absence de modifications inappropriées

Maintenabilité (*Maintainability*)

Capacité à être modifiée et réparé

Sécurité et security

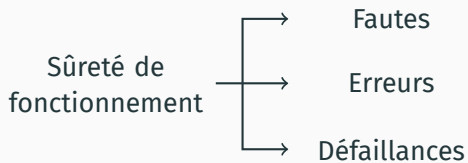
Warning : sécurité \neq security

- sécurité = *safety* (ou *functional safety*)
- sécurité-innocuité = *security*

Aujourd'hui, dans le cadre des systèmes informatisés, on lit/entend souvent sécurité pour sécurité-innocuité, et sûreté pour sécurité (*safety*).

Le terme cybersécurité désigne quant à lui un sous-ensemble de la sécurité informatique (celle concernant la protection des systèmes contre les utilisations malveillantes).

Les entraves à la SdF



Faute, erreur, défaillance

Défaillance

Une défaillance se produit lorsque le service fourni ne correspond pas au service attendu. Au moins un état externe n'est pas l'état attendu.

Erreur

Une erreur est la partie de l'état du système qui peut entraîner une défaillance. Elle peut être latente, ou activée.

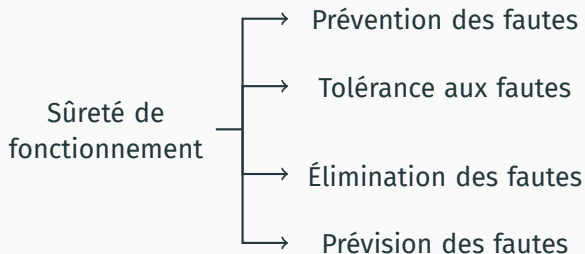
Une fois qu'elle est activée, l'erreur ne conduit pas toujours à la défaillance : la structure du système peut l'empêcher (redondance).

Faute

Une faute est un événement responsable (avéré ou supposé) de l'introduction d'une erreur dans le système.



Les moyens de la SdF



Les moyens de la SdF

Prévention des fautes

Ensemble des techniques et méthodes d'ingénierie qui visent à éviter l'introduction de fautes lors du développement ou de la maintenance.

Tolérance aux fautes

Part de la structure du système dédiée au traitement des fautes, dans le but de maintenir la fourniture du service (éventuellement dégradée) même en présence de fautes.

Élimination des fautes

Ensemble des techniques de vérification des propriétés du système, et des techniques de diagnostique et de correction associées.

Prévision des fautes

Évaluation du comportement du système en présence de fautes : qualitative (identification des modes de défaillance) et quantitative (mesure probabiliste des attributs du système).

A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr, *"Basic concepts and taxonomy of dependable and secure computing"*, in IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11-33, Jan.-March 2004, doi : 10.1109/TDSC.2004.2.

Organisation du cours

Dans la suite

Première partie : focus sur les aspects logiciels :

1. Prévention des fautes de développement : métriques, normes
2. Élimination des fautes : vérification
3. Les joies dangers de la programmation en C

4 séances, en incluant des présentations d'outils.

Seconde partie : éclairage sur la sûreté de fonctionnement au niveau système proposé par Kevin Delmas (ONERA) le 31 mai (M1-2, S1-2).

Le lanceur a commencé à se désintégrer à environ H0 + 39 secondes sous l'effet de charges aérodynamiques élevées dues à un angle d'attaque de plus de 20 degrés qui a provoqué la séparation des étages d'accélération à poudre de l'étage principal, événement qui a déclenché à son tour le système d'auto-destruction du lanceur; cet angle d'attaque avait pour origine le braquage en butée des tuyères des moteurs à propergols solides et du moteur principal Vulcain; le braquage des tuyères a été commandé par le logiciel du calculateur de bord (OBC) agissant sur la base des données transmises par le système de référence inertielle actif (SRI2).

À cet instant, une partie de ces données ne contenait pas des données de vol proprement dites mais affichait un profil de bit spécifique de la panne du calculateur du SRI 2 qui a été interprété comme étant des données de vol; la raison pour laquelle le SRI 2 actif n'a pas transmis des données d'attitude correctes tient au fait que l'unité avait déclaré **une panne due à une exception logiciel**; l'OBC n'a pas pu basculer sur le SRI 1 de secours car cette unité avait déjà cessé de fonctionner durant le précédent cycle de données (période de 72 millisecondes) pour la même raison que le SRI 2; l'exception logiciel interne du SRI s'est produite pendant une conversion de données de représentation flottante à 64 bits en valeurs entières à 16 bits. **Le nombre en représentation flottante qui a été converti avait une valeur qui était supérieure à ce que pouvait exprimer un nombre entier à 16 bits. Il en est résulté une erreur d'opérande. Les instructions de conversion de données (en code Ada) n'étaient pas protégées contre le déclenchement d'une erreur d'opérande** bien que d'autres conversions de variables comparables présentes à la même place dans le code aient été protégées; l'erreur s'est produite dans une partie du logiciel qui n'assure que l'alignement de la plate-forme inertielle à composants liés.

Dans le scénario de défaillance, les principales causes techniques sont l'erreur d'opérande lors de la conversion de la variable biais horizontal BH et l'absence de protection de cette conversion, qui a provoqué l'arrêt de fonctionnement du calculateur du SRI.

Il a été signalé à la Commission que les conversions n'étaient pas toutes protégées car un objectif de charge de travail maximale de 80% avait été assigné au calculateur du SRI. Afin de déterminer la vulnérabilité des codes non protégés, une analyse a été conduite sur chaque opération pouvant donner lieu à une exception, y compris une erreur d'opérande.

On a notamment analysé la conversion de représentations flottantes en nombres entiers et il s'est avéré que les opérations comportant sept variables risquaient de conduire à une erreur d'opérande. En conséquence, une protection a été ajoutée à quatre des variables, comme en témoigne le code Ada, alors que trois variables sont restées non protégées. Aucun élément justifiant cette décision n'a été retrouvé dans le code source proprement dit.

Bien que la source de l'erreur d'opérande ait été identifiée, elle n'est pas en soi la cause de l'échec de la mission. La spécification relative au mécanisme de traitement des exceptions a également contribué à la défaillance. En cas d'exception quelle qu'elle soit, la spécification système précise que la défaillance doit être indiquée sur le bus de données, que son contexte doit être enregistré dans une mémoire EEPROM (qui a été récupérée et lue à l'issue du vol Ariane 501) et que le processeur du SRI doit être arrêté.

C'est la décision d'arrêter le fonctionnement du processeur qui s'est finalement révélée fatale. Un redémarrage est impossible car le calcul de l'attitude est trop difficile pour être refait après l'arrêt du processeur, de sorte que le système de référence inertielle devient alors inutile. La raison qui sous-tend cette procédure draconienne tient à la philosophie adoptée par le programme Ariane, qui consiste à **ne prendre en compte que les défaillances aléatoires de matériels**. Dans cette optique, les mécanismes de traitement des exceptions ou des erreurs sont conçus pour faire face à une défaillance aléatoire de matériel, qui peut être efficacement prise en charge par un système de secours.

Bien que la défaillance ait été provoquée par une erreur de conception de logiciel à caractère systématique, il est possible d'introduire des mécanismes pour atténuer ce type de problème. Par exemple, les calculateurs présents dans les SRI auraient pu continuer de fournir leurs meilleures estimations des informations d'attitude requises. Il est préoccupant de constater qu'une exception de logiciel puisse être autorisée, voire requise pour provoquer l'arrêt d'un processeur alors que celui-ci commande des équipements critiques pour la mission. De fait, la perte d'une fonction logicielle correcte présente un risque car c'est le même logiciel qu'utilisent les deux unités SRI. Dans le cas d'Ariane 501, cela s'est traduit par la mise hors circuit de deux équipements critiques ne présentant pas de défaillance.

Quelles leçons tirer de ces éléments ?

Quelles leçons tirer de ces éléments ?

- Dans un système embarqué, un « *bug* » peut avoir des conséquences graves.
- Le logiciel peut être une des sources d'une défaillance complète du système, au même titre que le matériel.
- Comme souvent, une défaillance complète résulte de la présence de plusieurs fautes dans le système → toute hypothèse de conception doit être assortie d'un mécanisme de contrôle à l'exécution.
- Les fautes logicielles sont des fautes systématiques → une grande partie peut être évitée lors des phases de conception
 - En limitant la **complexité du code** (structure, présentation, etc.)
 - En explicitant les hypothèses.
 - En **vérifiant** a priori que les hypothèses sont valides.
 - En intégrant des **mécanismes de contrôle** à l'exécution.
- ...

Prévention des fautes

Règles de codage

Métriques

Normes de sûreté de fonctionnement logicielle

Pourquoi des règles de codage ?

Les **règles de codage** visent à couvrir les sources d'insécurité du développement logiciel :

- Erreur d'algorithmique
- Erreur de programmation
- Code vulnérable et/ou malveillant
- Utilisation de composants sur étagères non sûrs
- Mauvaise compréhension ou connaissance insuffisante de la sémantique du langage ou d'une API

Objectif : éviter les erreurs à l'exécution, qui, si elles ne sont pas gérées correctement, peuvent se propager et produire une défaillance.

- Erreurs à l'exécution : division par zéro, débordement de tampon, boucle infinie, erreurs de logique, etc.

Pourquoi des règles de codage ? (2)

C'est une demande explicite des normes de sûreté de fonctionnement logicielle pour les systèmes critiques.

- Faciliter l'élimination des fautes en favorisant la lisibilité du code.
- Éviter les fautes de codage en interdisant certaines constructions réputées dangereuses ...
- ...et en utilisant des constructions réputées fiables.

Que définissent les règles de codage ?

On va généralement retrouver :

- Des règles obligatoires (positives et négatives).
- Des règles recommandées (positives et négatives).
- Des préconisations visant à assurer l'homogénéité du code et faciliter sa lecture / compréhension / maintenabilité.
- Pour chaque règle : définition, motivation, faute ou erreur cible, exemples.
- Les cas dérogatoires et les processus à mettre en œuvre pour ces cas.

Règles de codage : l'exemple MISRA

MISTRA (*Motor Industry Software Reliability Association*) est un consortium regroupant constructeurs, équipementiers, et société d'ingénierie du domaine automobile.

Son activité principale est l'édition de standards décrivant des règles de programmation pour des logiciels embarqués sûrs et fiables.

Deux langages sont supportés :

- C++, avec le lancement de MISRA C++ en 2008
- C, avec la dernière édition MISRA C 2012 publiée en 2013 et amendée en 2019

Directive vs. Rules

MISRA C 2012, définit un ensemble de *guidelines*, décomposées entre :

- *directives* : indécidables, car faisant référence à des informations extérieures au code source (p. ex. doc de conception ou de spécification)
- *rules* : décidables, un outil d'analyse statique du code peut vérifier automatiquement la conformité.

On distingue ensuite :

- les *guidelines* obligatoires (aucune déviation autorisée).
- les *guidelines* requises (déviation autorisée si elle est justifiée et documentée).
- les *guidelines* recommandées.

Exemple de directive

Dir 4.4 Sections of code should not be “commented out”

Category Advisory

Applies to C90, C99

Amplification

This rule applies to both `//` and `/* . . . */` styles of comment.

Rationale

Where it is required for sections of source code not to be compiled then this should be achieved by use of conditional compilation (e.g. `#if` or `#ifdef` constructs with a comment). Using start and end comment markers for this purpose is dangerous because C does not support nested comments, and any comments already existing in the section of code would change the effect.

See also

Rule 3.1, Rule 3.2

Exemple de rule

Rule 11.5 A conversion should not be performed from pointer to *void* into pointer to object

C90 [Undefined 20], C99 [Undefined 22, 34]

Category Advisory
Analysis Decidable, Single Translation Unit
Applies to C90, C99

Rationale

Conversion of a pointer to *void* into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behaviour. It should be avoided where possible but may be necessary, for example when dealing with memory allocation functions. If conversion from a pointer to object into a pointer to *void* is used, care should be taken to ensure that any pointers produced do not give rise to the undefined behaviour discussed under Rule 11.3.

Exception

A *null pointer constant* that has type pointer to *void* may be converted into pointer to object.

Example

```
uint32_t *p32;  
void *p;  
uint16_t *p16;  
  
p = p32;          /* Compliant - pointer to uint32_t into  
                  * pointer to void */  
p16 = p;          /* Non-compliant */  
p = ( void * ) p16; /* Compliant */  
p32 = ( uint32_t * ) p; /* Non-compliant */
```

See also

Rule 11.2, Rule 11.3

Règles de codage : l'exemple SEI CERT

SEI CERT Coding Standard

- Règles pour écrire des programme moins vulnérables et plus sécurisés, couvrant plusieurs langages dont C et C++.
- Publié par le CERT (*Computer Emergency Response Team*) du SEI (*Softw. Eng. Institute, CMU*), un organisme de recherche étatsunien.

SEI CERT C Coding Standard

- Standard dédié à la programmation en C (couvre C99 et C11).
- De nombreuses entrées sont communes avec MISRA C et CWE (*Common Weakness Enumeration*).
- N'est pas considéré comme un standard orienté *safety*.
- Deux versions : livre/pdf (snapshot), et wiki (en constante évolution)
- Contient deux parties : les *rules* (must) et les *recommendations* (should).

Exemple de règle

EXP36-C. Do not cast pointers into more strictly aligned pointer types.

Comment appliquer les règles de codage ?

Les règles visent à contraindre la programmation : il faut convaincre les équipes de développement de leur intérêt

Pour une partie des règles (p.ex. les *rules* MISRA), des outils automatiques de vérification de la conformité du code peuvent être déployés

- Par exemple **Frama-C**, un outil libre et gratuit présenté lors d'une prochaine séance

Pour les autres règles, des revues de code et des audits de code peuvent être organisés.

Règles de codage

Métriques

Normes de sûreté de fonctionnement logicielle

Métriques logicielles

Une **métrique logicielle** est une fonction de mesure ou une combinaison de fonctions de mesure des propriétés techniques ou fonctionnelles d'un code source.

On distingue :

- Les métriques de base : sloc, complexité cyclomatique, nombre de commentaires, ...
- Les métriques composées : complexité de Halstead, indice de maintenabilité, ...

Les métriques composées sont issues d'études empiriques : leurs résultats doivent être interprété avec du recul

SLOC et autres métriques élémentaires

Plusieurs métriques élémentaires sont faciles à extraire du code source :

- SLOC : le nombre de ligne de code source
 - plusieurs décomptes possibles : avec/sans commentaires, lignes blanches, ...)
 - plusieurs découpes possibles : par fonction / fichier / projet, ...
 - pas de seuil indicatif universel
- Profondeur maximal d'appel
- Nombre de fonctions appelées
- ...

La complexité cyclomatique

La complexité cyclomatique M d'une fonction est une métrique proposée par Thomas J. McCabe en 1976. Elle vise à mesurer la difficulté à tester une fonction / la complexité de la fonction.

Elle est défini simplement comme étant égal au **nombre de cycles linéairement indépendants** dans le **graphe de flot de contrôle** G de la fonction.

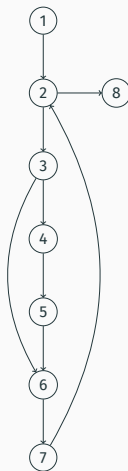
Elle se calcule automatiquement sur le graphe de flot de contrôle :

$$M = E - N + 2P$$

- E est le nombre de d'arc de G
- N est le nombre de nœuds de G
- P est le nombre de composantes connexe de G (à l'intérieur d'une fonction, $P = 1$)

Complexité cyclomatique : exemple

```
size_t popcount(unsigned int a) {  
    /* 1 */    size_t res = 0;  
    /* 2 */    while (a != 0) {  
        /* 3 */        if ((a & 1) == 1) {  
            /* 4 */            res = res + 1;  
            /* 5 */        }  
        /* 6 */        a = a >> 1;  
        /* 7 */    }  
    /* 8 */    return res;  
}
```



$$E = 9, N = 8, P = 1, M = 9 - 8 + 2 \times 1 = 3$$

Comment baisser M à 2 ? Est-ce plus lisible ?

Complexité cyclomatique : exemple

```
size_t f(const int a[], size_t len, int lo, int hi) {  
    size_t index;  
    size_t count;  
    for(index=0, count=0; index<len; index++) {  
        if (lo <= a[index] && a[index] <= hi) {  
            count = count + 1;  
        }  
    }  
    return count;  
}
```

Que vaut M ?

Interprétation de la complexité cyclomatique (1)

McCabe propose d'utiliser M pour mesurer le risque associé à un programme, et propose l'interprétation empirique suivante :

- $M \leq 10$: procédure simple, peu de risque
- $11 \leq M \leq 20$: plus complexe, risque modéré
- $21 \leq M \leq 50$: procédure complexe, risque élevé
- $51 \leq M$: procédure trop complexe, risque très élevé

Pour les systèmes critiques, on retient 2 classes : complexité normale (moins de 10) et complexité élevée (plus de 10).

Interprétation de la complexité cyclomatique (2)

M est aussi utilisé comme indicateur de testabilité / capacité à être analysé d'une fonction

- borne supérieure sur le nombre de cas de test pour couvrir toutes les branches;
- borne inférieure sur le nombre de chemins dans G ;
- corrélation forte entre la classe (complexité normale ou élevée) et la terminaison des techniques d'analyse statique.

Il n'est pas clair que M soit un meilleur indicateur que $SLOC$ ou d'autres métriques concernant la densité de fautes d'un programme. Dans le doute, les normes recommandent de limiter les valeurs des deux métriques.

Métrique composée : la complexité de Halstead (1)

Proposée par Halstead en 1977 qui cherchait à établir une science du développement logiciel fondée sur des propriétés mesurables, à l'image des propriétés de la matière.

Les différentes mesures sont dérivées de 4 quantités élémentaires :

- n_1 le nombre d'opérateurs uniques
- N_1 le nombre total d'opérateurs
- n_2 le nombre d'opérande uniques
- N_2 le nombre total d'opérandes

On en déduit :

- $N = N_1 + N_2$ la longueur du programme
- $n = n_1 + n_2$ la taille du vocabulaire du programme

Métrique composée : la complexité de Halstead (2)

Halstead dérive ensuite, de façon empirique, les mesures suivantes :

- $V = N \times \log_2(n)$ le volume du programme
- $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$ le niveau de difficulté d'un programme
- $L = \frac{1}{D}$ le niveau du programme
- $E = V \times D$ l'effort à l'implémentation
- $T = \frac{E}{18}$ estime le temps d'implémentation (en secondes)
- $B = \frac{E^2}{S}$ est une estimation du nombre de bugs du programme, avec S une constante liée à la compétence de la personne qui développe ($S = 3000$ pour une personne normale).

Métrique composée : la complexité de Halstead (3)

```
main()
{
    int a, b, c, avg;
    scanf("%d %d %d", &a, &b, &c);
    avg = (a+b+c)/3;
    printf("avg = %d", avg);
}
```

- opérateurs : main, (), {}, **int**, ,, ;, scanf, &, =, +, /, printf
- opérandes : a, b, c, avg, "%d %d %d", 3, "avg = %d"
- $n_1 = 12, n_2 = 7, n = 19, N_1 = 27, N_2 = 15, N = 42$
- $V = 42 \times \log_2(19) = 178.4$
- $D = \frac{12}{2} \times \frac{15}{7} = 12.85$
- $E = 12.85 \times 178.4 = 2292.44$
- $T = \frac{2292.44}{18} = 127.357$ secondes
- $B = \frac{2292.44^{2/3}}{3000} = 0.05$ bug

Conclusions sur les métriques

Les métriques logicielles sont un outil très simple d'utilisation, surtout que leur calcul est automatisé par des outils.

Elles permettent de comparer objectivement deux implémentations différentes d'un programme mais l'interprétation précise des valeurs calculées reste difficile.

De façon pragmatique, les normes de sûreté de fonctionnement logicielle recommandent de minimiser les valeurs des métriques de complexité.

Règles de codage

Métriques

Normes de sûreté de fonctionnement logicielle

Les normes de sûreté de fonctionnement logicielle

Comme souvent, chaque grand domaine industriel a proposé sa norme :

- Ferroviaire : EN 50128
- Automobile : ISO 26262-6
- Avionique : DO 178C
- Électronique (multi-domaine) : IEC 61508-3
- ...

Ces différentes propositions s'appuient sur les mêmes moyens d'action et formulent des recommandations souvent très similaires.

Les moyens d'action

- Choix du langage
- Règles de codage
- Choix des outils de développement
- Contrainte sur la taille et la complexité des fonctions (métriques)
- Programmation robuste (défensive, redondance)
- Techniques de test, taux de couverture à atteindre
- Analyse statique
- ...

Exemple : les langages de programmation

- **C reste une recommandation**, même pour les systèmes les plus critiques; son utilisation est contrainte via les autres moyens d'action.
- Une solution est d'**engendrer du code C à partir de programmes / modèles de haut niveau** (très utilisée en automobile, cf. AUTOSAR)
- D'autres langages réputés plus sûrs sont fortement recommandés pour les systèmes les plus critiques, p. ex. Ada (spatial) ou Scade/Lustre (avionique)
- Attention, il doit exister une **chaîne de développement complète et qualifiée** : frein à l'utilisation de nouveau langage comme Rust

Exemple : les langages de programmation (2)

Les critères de choix du langage sont souvent les mêmes :

- Largement utilisé (facilité à trouver des personnes compétentes pour le dév et la maintenance)
- Détection des erreurs à la compilation : typage fort, déterminisme, ...
- Support des assertions et de la programmation défensive
- Support pour la gestion des erreurs et des exceptions
- ...

Si un langage est faible sur un ou plusieurs critères, les autres moyens d'actions doivent être utilisés pour pallier ces faiblesses.

Les recommandations

En terme de programmation, outre les règles de codage, on retrouve des techniques ou familles de techniques :

Programmation défensive

- vérification des arguments
- tests de vraisemblance, de bornes
- utilisation de compteur pour borner les boucles **while**
- pas d'arithmétique de pointeur,
- contrôle strict de la gestion de la mémoire
- ...

Contrôle de validité des données

- utilisation de code détecteur voire correcteur

Détection des fautes matérielles ou d'interaction

- intégrité du flot de contrôle
- chien de garde
- tests en ligne, vérification en ligne

Adéquation fautes / réponses

Les techniques à mettre en œuvre dépendent bien sûr du modèle de faute retenu.

- Erreur matérielle (p. ex. SEU) : tests / vérification en ligne, intégrité du flot de contrôle, code détecteur/correcteur
- Erreur logicielle : programmation diversifiée, blocs de recouvrement
- Erreur temporelle : datation des données, surveillance des temps d'exécution

Illustration avec le domaine automobile (extrait de la norme ISO2626-2)

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	0	+	++	++
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.					