Sûreté de fonctionnement des systèmes informatiques

Plan (1)

- Introduction
- Evitement des fautes
- Tolérance aux fautes
  - principes généraux
    - (détection, confinement, reconfiguration, diagnostic)
  - codes détecteurs et codes correcteurs d’erreurs
  - codes détecteurs à l’exécution
  - redondance
  - Tolérance aux pannes dans les systèmes répartis
    - paradigmes
      - election, consensus, appartenance à un groupe
  - Tolérance aux erreurs logiciel

Plan (2)

- Architectures pour la sûreté de fonctionnement
  - Algorithmes répartis pour la tolérance aux fautes
  - modèle de panne dans les systèmes répartis
  - architectures caractéristiques
- Validation d'architectures
  - Spécification de propriétés
    - (observateurs, logique temporelle, fonctions de mission)
  - Modélisation d'architectures par systèmes de transitions entre états
  - Vérification des propriétés par exploration de modèle
  - Validation probabiliste

Plan (3)

- Evaluation prévisionnelle
  - Composantes de la SDF
  - Evaluation par diagrammes de fiabilité
  - Evaluation par arbres de défaillance et calcul de coupes minimales
  - Méthodes de simulation

- Exemples d’architectures de systèmes informatiques
  - architectures orientées disponibilité
  - architectures orientées sécurité

Plan (4)

- Techniques de Conception de logiciels “sûrs” (évitement des erreurs de conception logicielle)
  - méthodes de spécification formelle
    - les différents langages et environnements disponibles
    - langages formels et preuve
      - processus de développement formel
      - exemple : Meteor (B) (processus mis en oeuvre, orésentation meteor et retour d’expérience MATRA TRANSPORT)
    - modélisation et exploration de modèle
      - processus de développement
      - exemple 2 : utilisation de l’approche synchrone (Aerospatiale) ;
    - les langages d’expression de propriétés et la vérification de propriétés

Plan (5)

- Détection des erreurs d'exécution ()
  - caractéristiques des erreurs d'exécution
  - analyse statique fondée sur l'interprétation abstraite (Polyspace Technologies)
  - exemple : Ariane 501 (CNES/DLA)
- Techniques d'élimination des erreurs
  - AEEL (MAGGALY)
  - Règles de codage et Relecture critique de code (MAGGALY)
  - TEST
    - politiques et stratégies de test
    - mesures de l'efficacité des tests
    - quand arrêter les tests ?

Plan (6)

- Processus de développement de systèmes sûrs à logiciel prépondérant les aspects système : caractérisation des exigences sdf
  - les choix d'architecture
  - le logiciel
    - logiciels développés
    - logiciels off the shelff
  - démonstration de la couverture des erreurs
    - démarche constructive
    - traçabilité du profil sdf

Plan (7)

- Les outils de la sûreté de fonctionnement
  - outils de modélisation comportementale et temporelles (AEF temporisés, rdp, files d'attente,...)
  - outils basés sur l'analyse des événements redoutés (AER, AMDEC, Coupes minimales)
  - outils d'évaluation quantitative : processus markoviens et semi markoviens, rdps, files d'attente, méthode Monte Carlo....
  - outils d'analyse statique de code
  - outils de suivi des tests
- Ateliers pour la sdf
  - ATL de la RATP (JP Rubaux)

Plan (8)

- Les techniques spécifiques de la sécurité (INRETS)

Bibliographie

- LIS : Guide de la Sûreté de Fonctionnement, CEPADUES Editions
    - Qui renvoie à une bibliographie très importante
- Bibliographie en annexe
- G. Mottet (référence à compléter)

1. Définitions et terminologie

- [Laprie89] : “la sûreté de fonctionnement d’un système informatique est la propriété qui permet de placer une confiance justifiée dans le service qu’il délivre.
- Service rendu par le système : comportement perçu par l’utilisateur
  - environnement fonctionnel (procédé,autre produit, opérateur)
  - environnement non fonctionnel (température, environnement electromagnétique,....)
  - mission et durée de mission
- Défaillance : service délivré non acceptable (par rapport à la fonction attendue du système)

Types de défaillances (1)

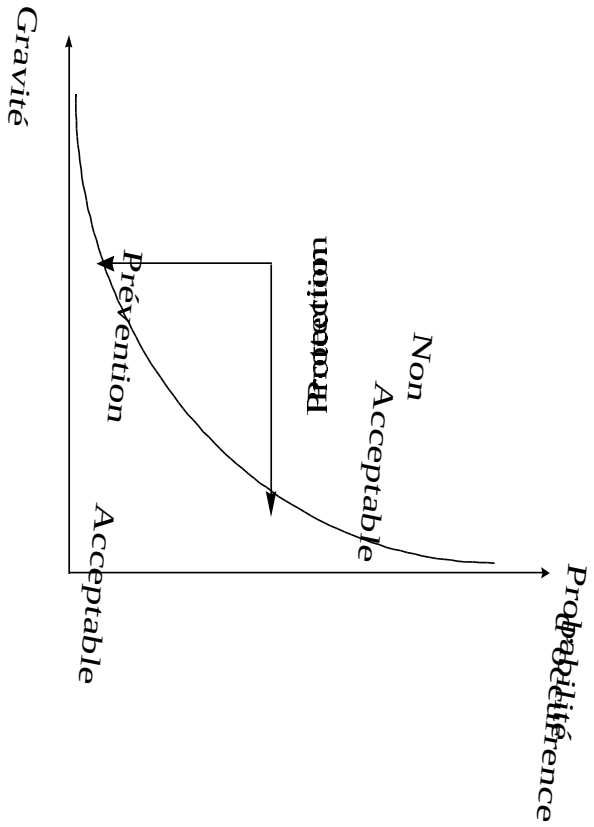
- Caractéristiques externes des défaillances
  - défaillance statique : le système produit un résultat non correct de manière permanente (aspects fonctionnels)
  - défaillance dynamique : le système a un régime transitoire pendant lequel le système produit un résultat faux puis atteint un régime permanent durant lequel les sorties sont correctes (aspects temporels)
  - défaillance durable : le système produit des résultats erronés de manière persistente
  - défaillance transitoire
  - défaillance cohérente ou incohérente suivant que la perception de la défaillance est identique pour les utilisateurs ou non.

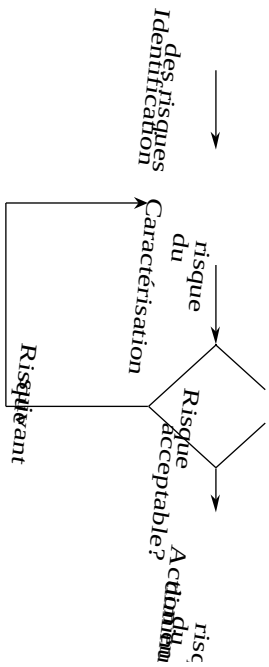
Types des défaillances (2)

- Classification des défaillances pour les systèmes répartis (Dolev, Cristian)
    - panne franche
    - panne d'omission
    - panne temporelle
    - panne byzantine
  - panne franche < panne d'omission < panne temporelle < panne byzantine

- Gravité ou mesure des effets sur la mission

Classement des conséquences des défaillances





Niveaux de gravité

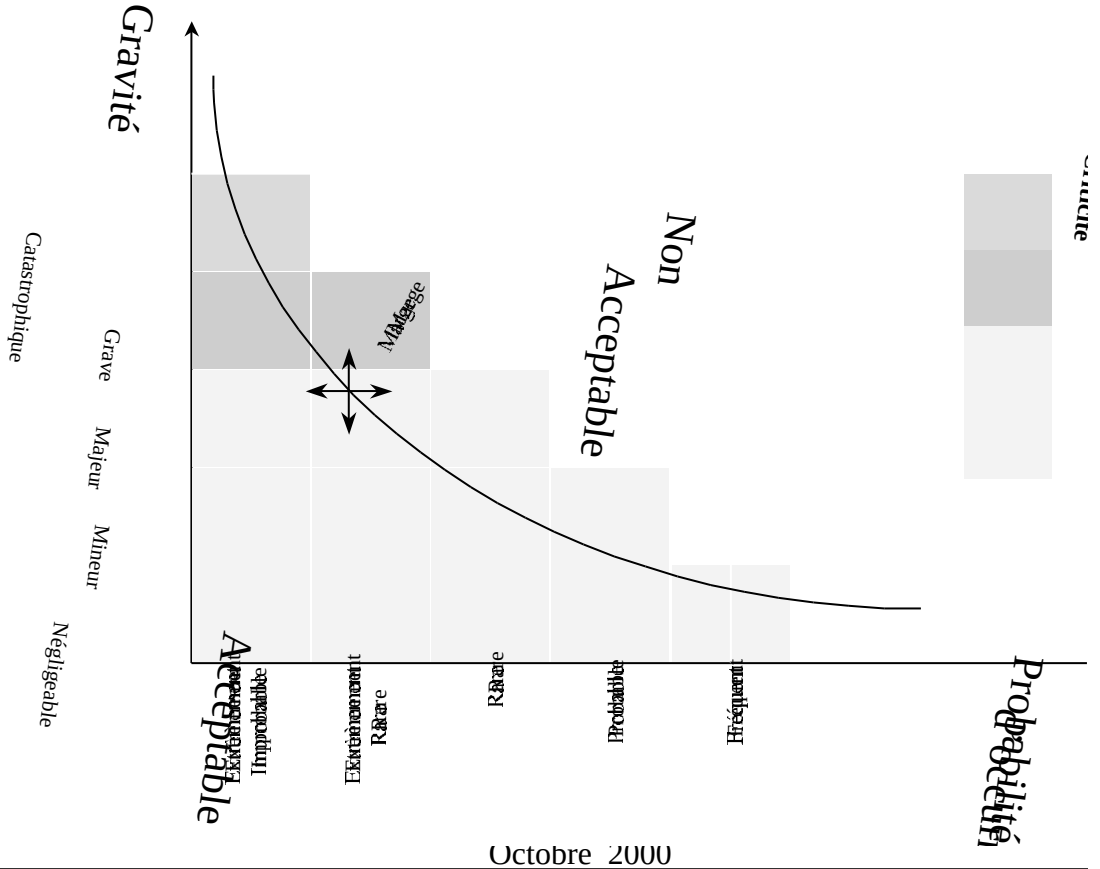
Classification des risques selon une échelle, qui peut dépendre du domaine applicatif

Niveau de gravité	Code de gravité	Description
Catastrophique	0A	Risque de perte de vie humaine ou agression sur le personnel d'exploitation.. Effets sur l'environnement à long terme.
Grave	0B	Destruction importante de biens qui interrompent les activités sur une longue période. Effets sur l'environnement à court terme.
Majeur	1	Perte de mission. Endommagement d'un bien.
Mineur	2	Mission dégradée.
Négligeable	3	Sans effet sensible sur le déroulement de la mission.

Niveaux de probabilité

Niveaux de probabilité	Code de Probabilité
Extrêmement improbable	PA
Extrêmement Rare	PB
Rare	PC
Probable	PD
Fréquent	PE

Niveaux de gravité et niveaux de probabilité



Domaines de criticité

Fréquence Gravité	PA	PB	PC	PD	PE
0A	15				Domaine
0B	13	14			Inacceptable
1	6	10	12		
2	3	5	9	11	
3	1	2	4	7	8

Echelle de criticité

Domaines et niveaux	Classes de criticité
De 1 à 6	1 : criticité la plus faible
De 7 à 12	2
De 13 à 14	3
15	4

Défaillances, Erreurs, Fautes

- Erreur : état (partiel) susceptible d'entraîner une défaillance
  - latente/détectée
  - propagation d'erreur, produit d'autres erreurs
- Faute : cause adjugée ou supposée d'une erreur
- Défaillance
  - manifestation d'une erreur qui par propagation traverse la frontière du système avec son environnement.
- ...defaillance -> faute -> erreur -> défaillance -> faute ->....

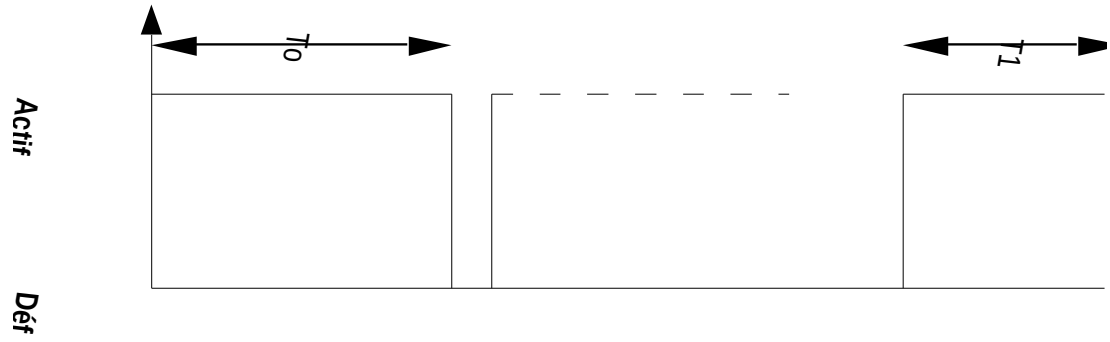
Les Fautes : caractéristiques

- Fautes [Laprie]
  - Cause phénoménologiques
    - fautes physiques
    - fautes dûes à l'homme
  - Nature
    - accidentelle
    - intentionnelle
  - Phase d'occurrence
    - conception
    - exploitation
  - Frontière du système
    - faute interne
    - faute externe
  - Persistence
    - temporaire
    - permanente

Prévision des fautes

- Evaluer prévisionnellement le comportement du système par rapport à l'occurrence des fautes
  - Examiner les défaillances des composants d'un système et leurs conséquences sur la sûreté de fonctionnement
- Etats observables du système
  - Service correct : accomplit la (les) fonctions du système
  - Service incorrect : non (accomplit la (les) fonctions du système)
- Attributs principaux
  - Fiabilité
  - Disponibilité
  - Sécurité-innocuité

Paramètres de la sûreté de fonctionnement



Fiabilité

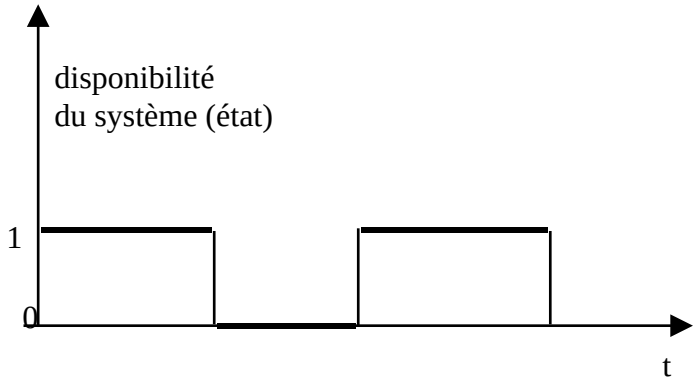
- Durée de mission
- Continuité de service

- Fiabilité stabilisée :
  · MTTF: temps de fonctionnement correct continu.
  · MDT: temps d'interruption de service admissible.

- Croissance de fiabilité

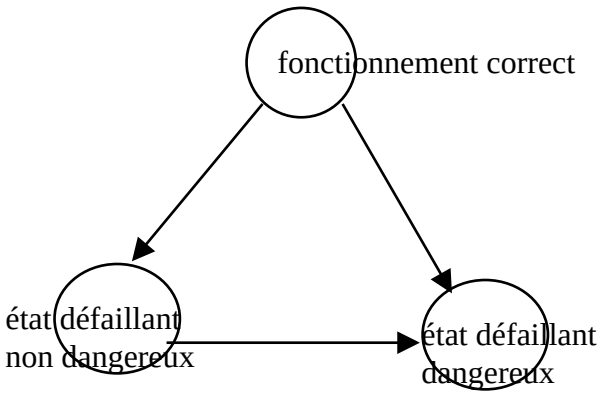
Disponibilité

- probabilité pour que le produit fonctionne à l'instant t sachant qu'il est en état de fonctionnement correct à t=0
- disponibilité instantanée
  - A(t) est donnée par la probabilité de l'état 1 à l'instant t
- disponibilité asymptotique (montrer que régime permanent existe => conditions d'ergodicité)
  - A*



sécurité (innocuité)

- distinguer les états défaillants mais non dangereux et les états dans lesquels un événement indésirable peut se produire.
- S(t) probabilité que le produit n'ait pas de défaillance catastrophique entre 0 et t



autres grandeurs de la sûreté de fonctionnement

- testabilité
  - facilité à déterminer les séquences de test
  - longueur des séquences de test, nombre de vecteurs d'entrée et de sortie à observer
  - couverture de test
  - efficacité de test (nombre de fautes détectées vs nombre total estimé de fautes dans le produit)
- Sécurité (security)
  - confidentialité
  - intégrité

Exigences SDF

- Le premier niveau d'exigences de SDF concerne la définition des événements redoutés et leur hiérarchisation en fonction de leur criticité.
- Une définition explicite des événements redoutés permet de préciser sans ambiguïté les priorités visées et objectifs donnés en termes de Disponibilité de service et de Sécurité. Elle reflète la politique et le niveau de maîtrise des risques dans le cadre d'un projet.

- Un objectif qualitatif, mais aussi généralement quantitatif, est affecté à chaque événement redouté. L'objectif quantitatif correspond généralement à la probabilité d'occurrence maximum admise pour cet événement durant la vie du système.

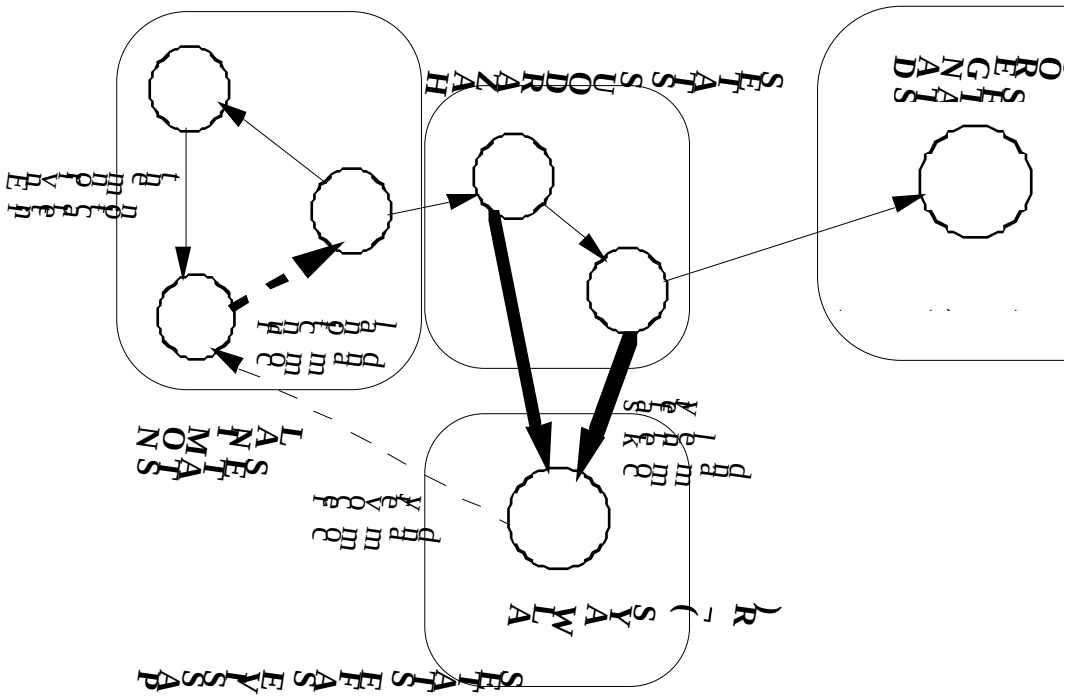
Illustration de la formalisation des objectifs

Exemple d’un système de transport terrestre

Objectifs généraux

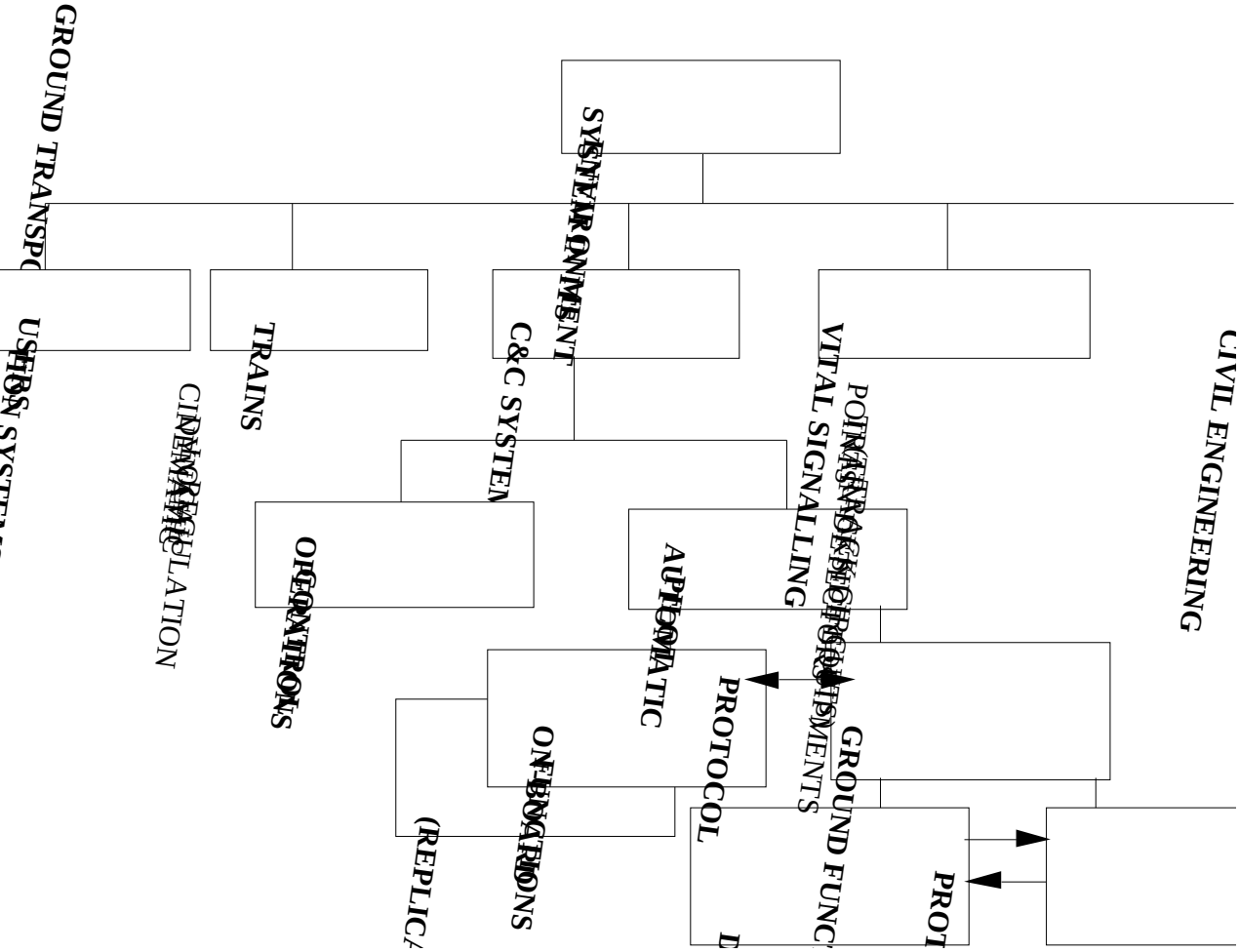
- Sécurité-innocuité
  - Événements redoutés et probabilité objectif
    - Accidents collectifs < 10-9/h en fonctionnement continu
    - Accidents « individuels »
    - ….
- Disponibilité du système
  - Probabilité d’interruption de service < 10-5/h

Sécurité : classification d’états pour les systèmes à états de sécurité passif

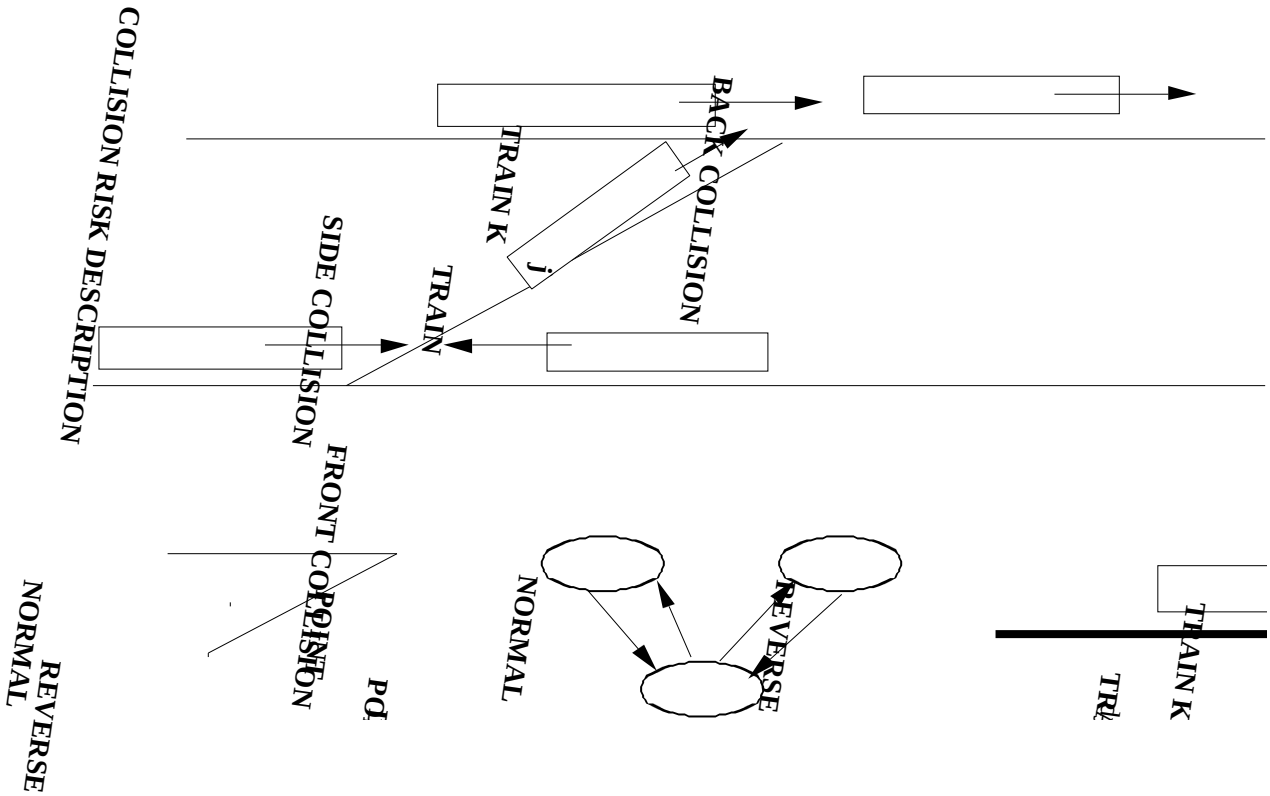


analyse du comportement souhaité vis à vis de la sécurité innocuité

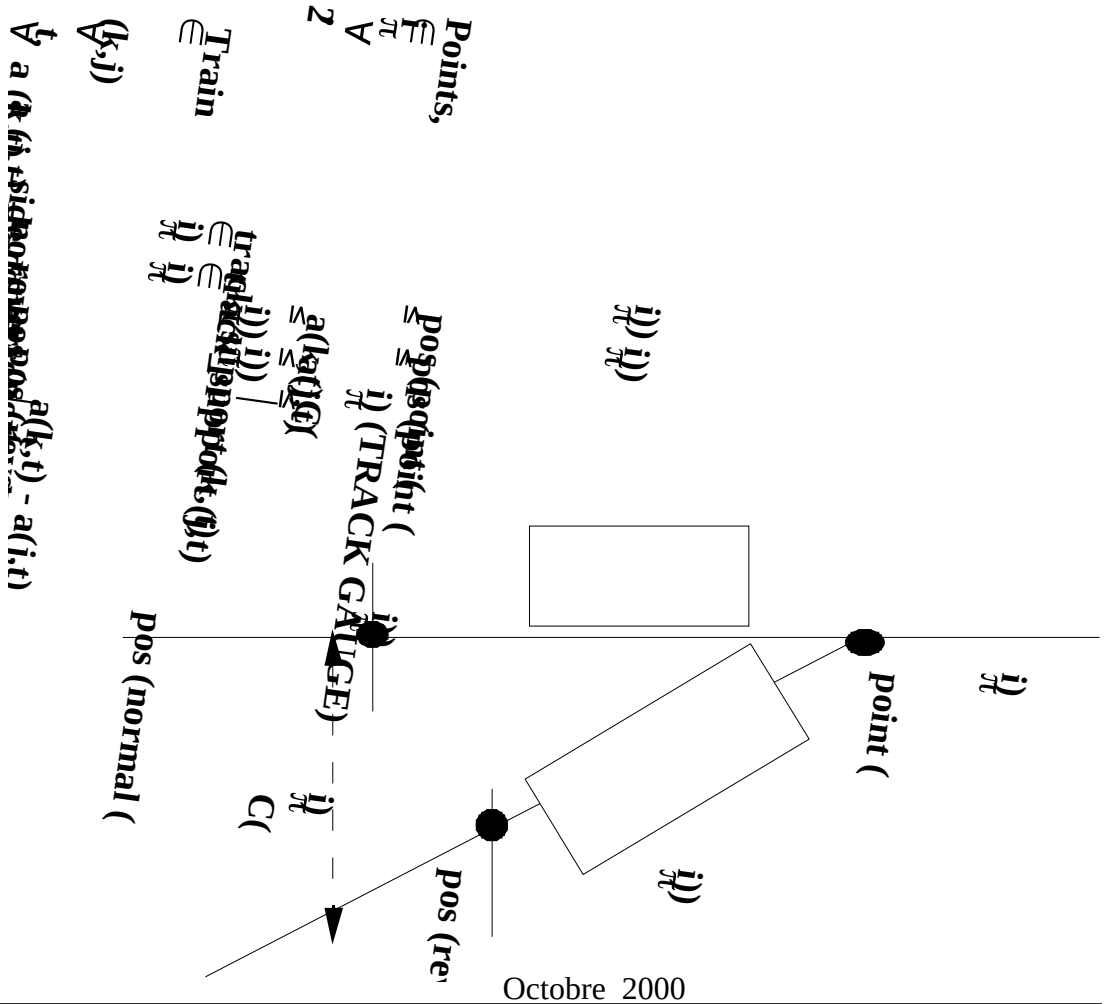
description structurelle



Identification du (des) risques



formalisation de l’événement redouté au niveau système

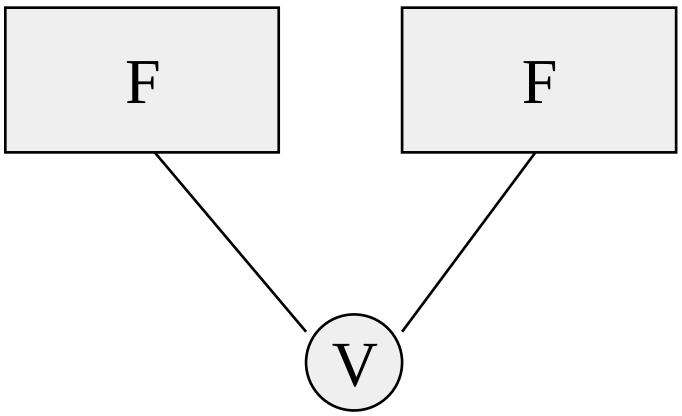


spécification des comportements des modules

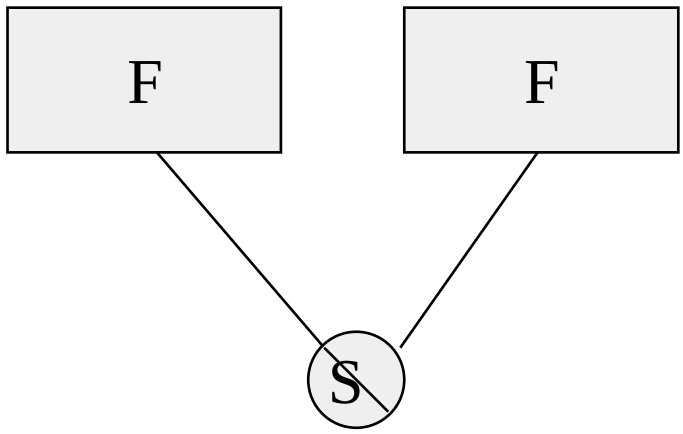
spécification des comportements des modules (2)

Projection sur la réalisation des modules

- Choix d’architecture fonctionnelle



Sémantique de panne
Fail Silent



Sémantique de disponibili

Modèles de fautes pour les systèmes à logiciel prépondérant

- Les prescriptions de sûreté de fonctionnement des systèmes à logiciel se déclinent à trois niveaux :

  – Au niveau des phases de spécification système,
  – Au niveau des choix d'architecture numérique,
  Au niveau des processus de développement logiciel.

Phase d'occurrence des fautes (1)

- fautes de spécification
  - erreur de frontière
  - incomplétude
  - incohérence
- fautes de conception
  - erreur de raffinement
    - faute au niveau d'un module (production de sorties non conformes à la spécification du module)
    - fautes dans les interfaces entre modules
  - non respect de contraintes technologiques (utilisation de ressources, partage de ressources,...)

Phase d’occurrence des fautes (2)

- fautes de production
  - matériel
  - logiciel
    - génération de code à partir d’un modèle de conception (manuel ou générateur)
    - passer du programme source à l’exécutable : l’exécutable doit produire un comportement équivalent à l’interprétation du du source (avec la sémantique du langage) => deux sources d’erreur la défaillance du compilateur et les ambiguités sémantiques du langage de programmation

modèles de fautes (1)

- Modèles de fautes en spécification
  - modèle comportemental (RDP, AEF, AEFC, équations)
    - fautes tel que oubli d'une transition, emission à tort d'un message, écriture d'une équation fausse
    - erreur :incohérence du modèle, apparition d'un interblocage, système non réinitialisable, état non atteignable, transitions non tirables ....
    - détection possible par analyse ou exploration du modèle comportemental(invariants, logique temporelle)
    - détection possible par simulation du modèle : recherches de scénarios nominaux et dégradés,

modèles de fautes (2)

- spécification : prise en compte des défaillances possibles
  - une fonction F ne produit pas de sorties (panne franche)
  - une fonction F, sur sollicitation ne produit pas de sortie (omission)
  - une fonction F produit des sorties intempestives (faute temporelle)
  - une fonction F produit des sorties incorrectes (faute byzantine).
    - Il faut analyser les modes de défaillance qui doivent être admis et donc traités: exemple dans un protocole de communication
      - réémission d'une requête si la réponse est tardive (faute temporelle) -> nécessité de désigner de manière unique une requête et d'associer la réponse correspondante
      - nécessité de retour à un état stable et traitement des pannes d'omission ou des pannes franches.

modèles de fautes (3)

- Conception générale
  - il s’agit d’implanter le modèle de spécification sur une architecture
    - erreurs d’interfaces entre modules (incohérence ou incomplétude)
    - mauvaise prise en compte des règles de communication et de synchronisation exprimées dans le modèle abstrait (e.g files bornées, délais,..) (non conformité avec le modèle de niveau supérieur)
    - non réalisation d’une fonction , ou ajout d’une fonction de conception qui introduit un état non prévu (acquisition des entrées, utilisation de constantes, de bases de données,...)
    - non couverture des modes de défaillance analysés en spécification

modèles de fautes (4)

- conception générale (suite)
  - détection des erreurs dans une conception :
    - matrices de traçabilité entre modèle de spécification et conception; au minimum : répartition des fonctions sur l’architecture.
    - justification des modes de réalisation des interfaces entre modules
    - ...

modèles de fautes (5)

- conception détaillée-codage
  - exemples d'erreurs
    - non initialisation de donnée,non retour de valeur lors d'un appel
    - non terminaison
    - appel à une fonction qui n'existe pas
    - absence de test de borne
    - non respect de type ou de domaine de valeur d'une variable
    - non respect d'une contrainte de causalité entre événements
  - détection possible : analyse de code (flot de contrôle, flot de données), RCC.

modèles de fautes (6)

- exécutable
  - exemples d’erreurs
    - branchement “intempestif”, débordement de pile,...
  - fautes possibles :
    - défaillance du compilateur
    - panne matérielle (tq modification de mot mémoire)
    - erreur de programmation

modèles de fautes (7)

- modèles de fautes matérielles
  - de conception au niveau du circuit : le circuit ne rend pas la fonction attendue
  - fautes matérielles de type collage ou transformation d'un 0 en 1 et réciproquement,
  - défaillances de contacts ou de tout autre composant électronique
  - court circuits affectant le circuit logique
  - ...

Face aux fautes

- Evitement/prévention
  - comment empecher, par construction, l'occurence ou l'introduction de fautes
- Tolerance aux fautes:
  - comment delivrer, par redondance, un service conforme à la specification
- Elimination des fautes:
  - comment reduire, par vérification, la presence des FAUTES
- Prevision des fautes:
  - comment estimer, par evaluation, la presence, la creation et les consequences des fautes.

2. Principes d’architectures pour la Tolérance aux fautes

Tolérance aux fautes

- objectif : continuité du service délivré en dépit de la présence de fautes
  - détection et confinement d'erreur
  - recouvrement : déterminer et trouver un état correct pour poursuivre les traitements
  - reconfiguration
- moyens de la tolérance aux fautes
  - codes détecteurs
  - replication et vote
  - contrôles en exécution
- nécessité d'architectures redondantes

Détection d’erreur

- Principes généraux
  - Deux grands types de techniques sont utilisées:
    - Le codage
      - Lors de la création d'une donnée c, une donnée r=h(c) est calculée. On peut vérifier ultérieurement que cette relation est vérifiée.
    - Le vote
      - Plusieurs versions d'une même donnée sont calculées "indépendamment". Une erreur est détectée lorsqu'une des versions au moins est distincte des autres.
      - Si l'on dispose de suffisamment d'information redondante et sous une hypothèse d'étendue limitée de l'erreur, les deux techniques précédentes permettent de masquer certaines erreurs.

Codes détecteurs d’erreur

- Principe général du codage (binaire)
  - Soit I l'ensemble des données à coder
  - On injecte I dans un ensemble C (le code)
  - avec m>n
  - On définit la distance du code par:
    - d est le nombre de bits distincts entre deux mots du code (distance de Hamming). Supposons que d=2t+1. Alors le code détecte 2t erreurs binaires et corrige t erreurs binaires
- Les principaux codes utilisés sont:
  - Les codes de Hamming (mémoires)
  - Les codes polynomiaux (mémoires secondaires et transmission des données)
  - Les codes arithmétiques (détection des erreurs d'exécution)

Détection d’erreur par replication et vote

- principes
  - unités redondantes, qui doivent être indépendantes vis à vis de la création et de l’activation de la faute => copies ou éléments diversifiés
  - vote par comparaison (dans le cas de copies) ou par fonction de décision dans le cas d’éléments diversifiés
- voteur
  - hyp1 : externe et “parfait”; il faut 2 copies pour détecter une erreur byzantine, 3 pour la corriger avec une hypothèse 2/3
  - hyp2 : vote réalisé par les unités après échanges de messages; dans ce cas il faut au moins trois copies pour détecter une erreur byzantine et quatre pour la corriger (cf“les généraux byzantins”)

Détection par contrôle dynamique

- Contrôles temporels
  - contrôle du séquencement des instructions
  - contrôle des séquences d'événements (ex protocole de communication)
- contrôles de vraisemblance
- contrôles d'intégrité portant sur la structure des données
- Contrôle de signature
  - détection des erreurs sur le séquencement et sur la sélection des opérandes d'une opération
- autotests
  - révéler les fautes dormantes

Confinement

- objectif : éviter la propagation d'un état erroné
  - propagation spatiale => barrières portent sur les interfaces. La structuration des données doit permettre de détecter les erreurs lors des transmissions de données entre modules
  - propagation temporelle => utilisation de tests en ligne permettant d'éviter les erreurs opérationnelles en activant une erreur latente avant son activation par l'application

Recouvrement d’erreur : reprise arrière

- points de reprise
  - sauvegarde périodique de l’état du système
    - mécanismes de mémorisation qui protège les informations, vis à vis des effets des erreurs tolérées
    - problème de cohérence de l’état global sauvegardé
  - détermination de l’état global cohérent “le plus proche” et reprise des traitements
    - mécanismes sous jacents (traités en exercice) :
      - calcul des états globaux cohérents dans un système réparti
        - horloges vectorielles et coupures
      - diffusion fiable dans un système transactionnel
      - protocoles à 2 phases (2PC)
      - protocoles de terminaison (d’une étape)

Recouvrement d’erreur : reprise avant

- reprise à partir d’un état cohérent “futur”
  - toujours spécifique d’une application
  - reconstruction de contexte à partir de données externes (exemple relecture des capteurs)
  - la programmation doit prévoir les traitements de cas erronés ou dégradés (exceptions, signaux d’erreur)

Recouvrement d’erreur : masquage

- utilisation de redondance de manière à ce que le système continue à fonctionner correctement même en présence d’erreur
  - exemple codes détecteurs et correcteurs (cf cours réseaux)
  - vote majoritaire
- application des codes détecteurs et correcteurs : disques RAID

Tolérance aux pannes et modes communs

- La tolérance aux pannes repose sur l'utilisation de données ou de matériels redondants.
- La principale faille de cette technique est l'existence de pannes "de mode commun" d'une part et l'efficacité imparfaite des dispositifs de détection et de reprise (couverture).

- Les parades aux pannes de mode commun sont:
  - - La diversification
  - - La localisation des unités redondées (défaillance liée à la proximité des unités redondées)
  - - La "desynchronisation des traitements" (défaillances liées à une faute transitoire)

Redondance temporelle

- Pour un composant soumis à des pannes (intermittentes mais éventuellement temporelles ou même quelconques) il est de pratique courante de tenter de masquer cette panne par un nombre fixé de tentatives successives.
- Lorsque cette stratégie réussit la défaillance du composant est masquée au niveau supérieur. En cas d'échec une exception est transmise

• exemples

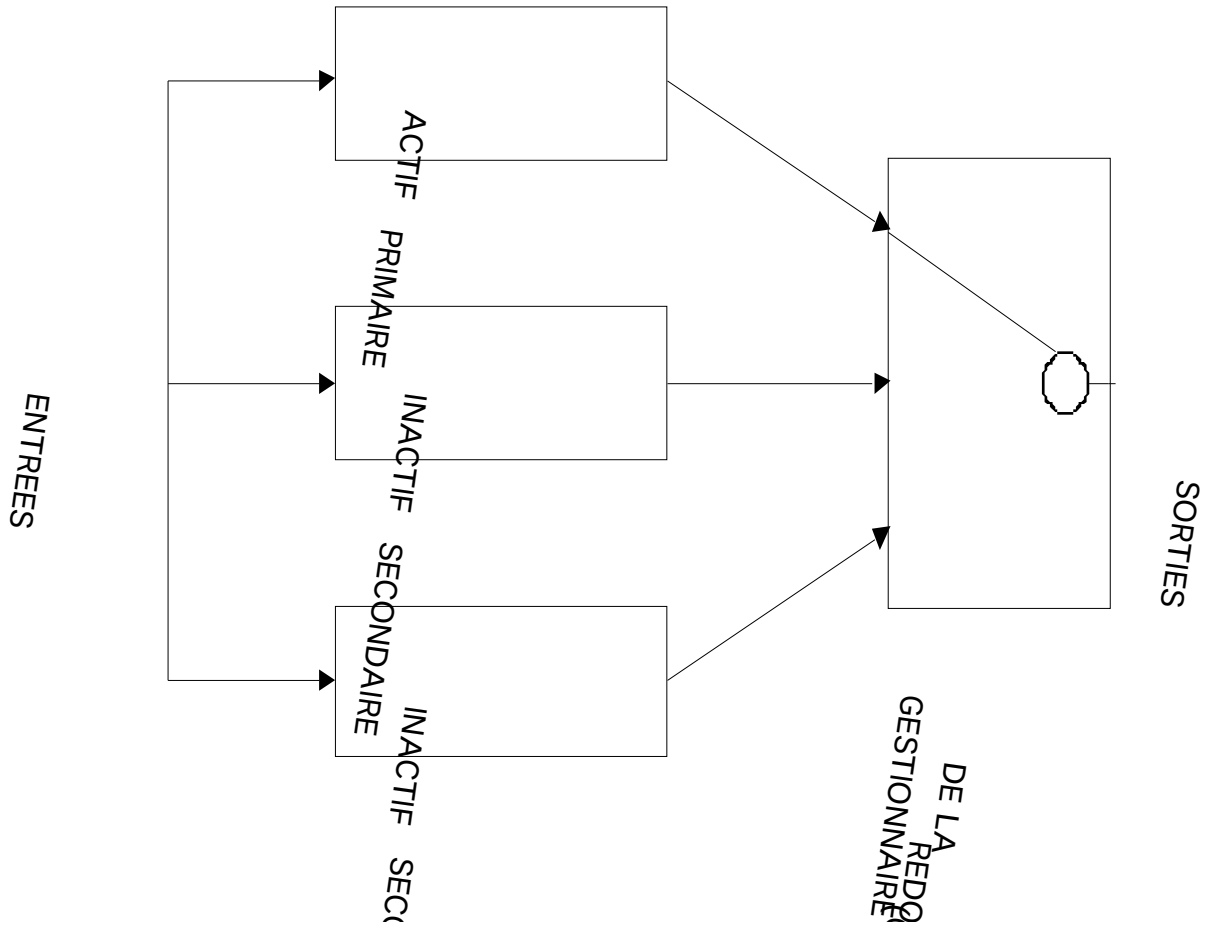
- Reprise dans les protocoles de communication (Lap B, TP4....)
- Reprise dans les opérations de transfert vers une mémoire secondaire.
- Dans les deux cas la détection est faite par codage.

Redondance spatiale

- Un groupe de composants redondants g peut-être conçu de telle façon que en dépit de la panne de certains membres de g le service offert du point de vue global par les membres de g continue (avec éventuellement des performances dégradées).

- g se comporte comme un composant unique ayant son propre niveau de défaillance supérieur au niveau de chacun des membres du groupe (les seules pannes observables par un utilisateur de g sont d'un niveau supérieur par exemple pannes d'omission si l'on masque les pannes franches).

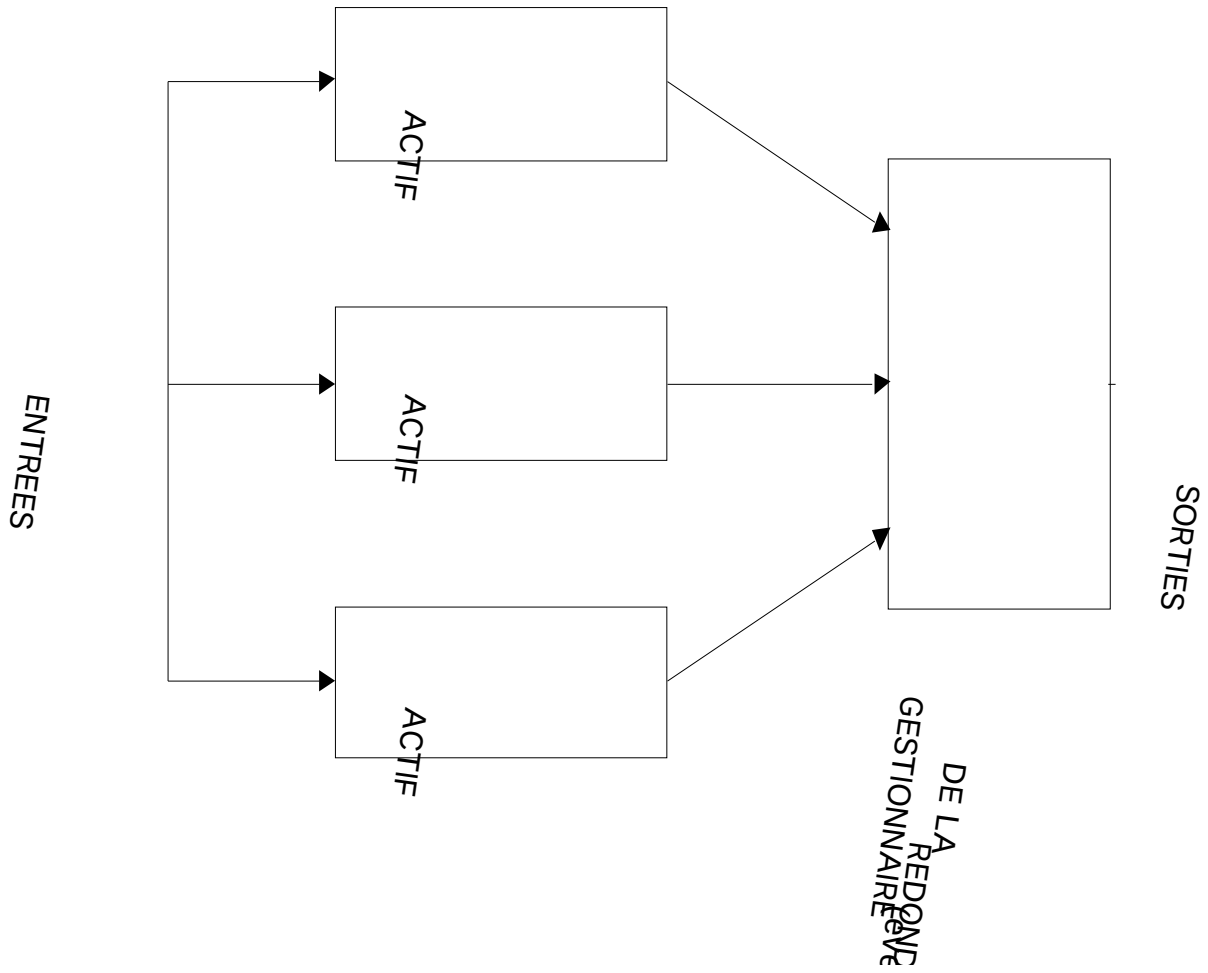
Redondance sélective passive



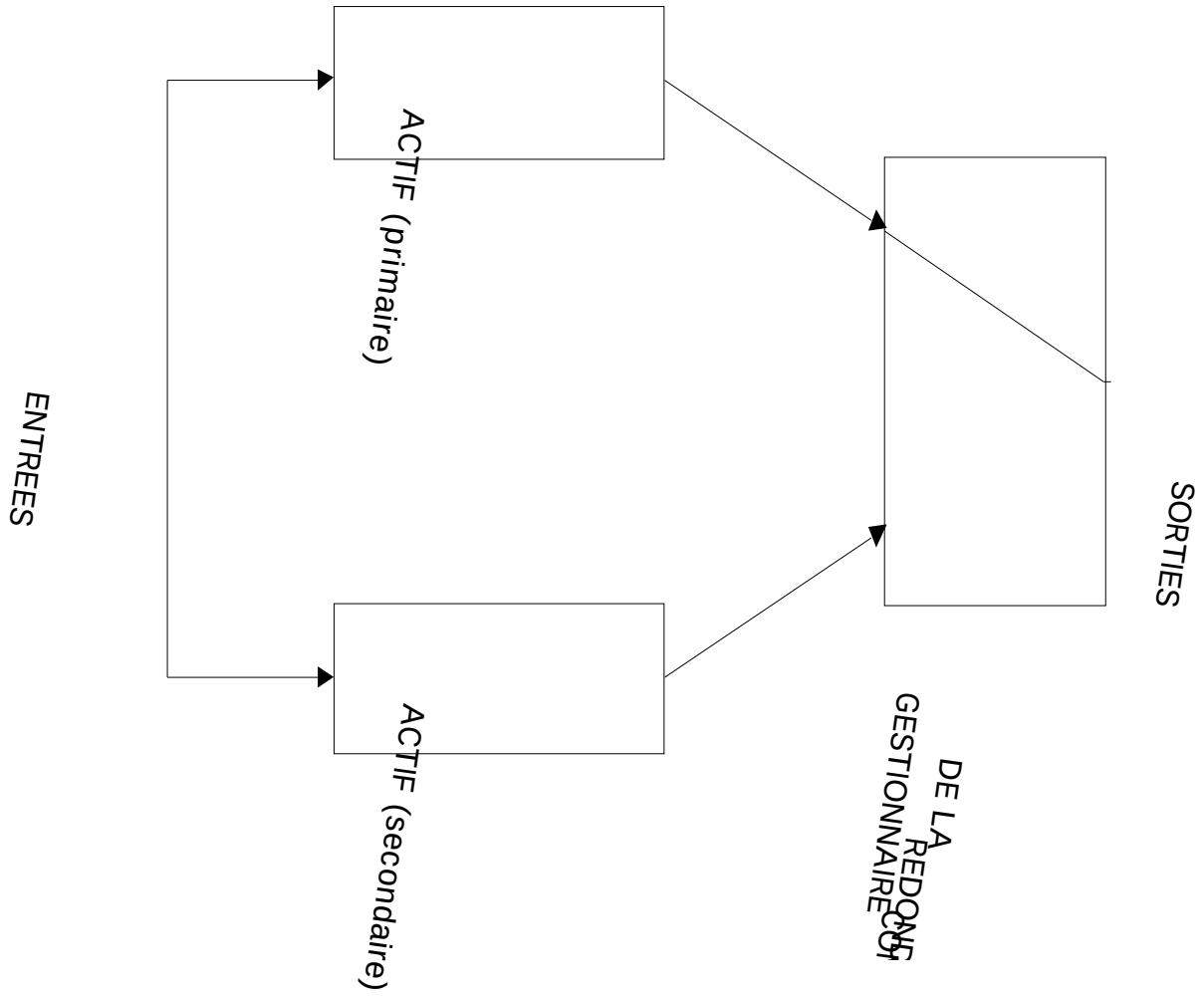
Redondance selective passive (pannes franches)

- tolérance aux pannes franches de calculateurs
  - Un seul des composants réalise effectivement les traitements et est affecté aux sorties (le primaire).
  - En cas de panne du primaire l'un des calculateurs inactifs (secondaire) est sélectionné et activé pour prendre en charge le service.
  - Pour le nouveau primaire il faut reconstituer un contexte d'exécution correct.
- Solution possible :
  - recopie périodique d'informations de reprise constituées par le primaire pour les secondaires.
  - rééxecution des services fournis depuis le dernier point de reprise en vue de reconstituer la partie manquante du contexte.

Redondance active (dynamique)



Redondance sélective active (pannes franches)



Architectures classiques

- Détection des erreurs
- Architectures redondantes
  - Architecture Duplex
  - Architecture TMR
  - Architecture 2/3
  - Architecture 2/4
  - Architecture N-MR

Détection des erreurs

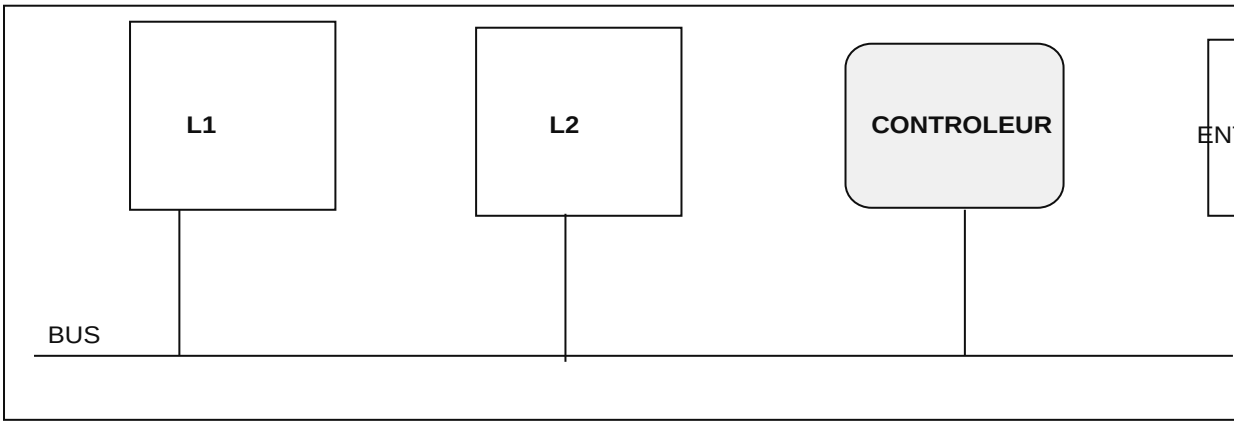
- La détection des erreurs dans les modules peut être obtenue en utilisant une des techniques suivantes :
- - tests périodiques
- - circuits autotestables
- - chiens de garde
-
- ***Tests cycliques*** : l'exécution d'un module est suspendue et on déroule une routine de test pour détecter la présence éventuelle d’erreurs dans le module. Ce mécanisme ne peut détecter les pannes temporaires que si elles apparaissent pendant la période de test ==> problème de couverture du test et d'optimisation de la fréquence de test.
-

Détection des erreurs (2)

- ***Circuits autotestables*** : les circuits autotestables sont conçus pour produire une sortie correcte ou indiquer la présence d'une faute dans le module.
-
- ***Chiens de garde*** : Les timers sont réglés pour effectuer des contrôles à des points pré-établis (***checkpoints)*** dans le programme qu'exécute un module. Pour un point de contrôle particulier, le chien de garde est armé et il décroit pendant que le module exécute ses fonctions. En l'absence d'erreur, le chien de garde est réarmé avant que le prochain point de contrôle soit atteint. L'occurrence d'une faute (hard ou soft) empêche le module de réarmer le chien de garde.

Architecture Duplex

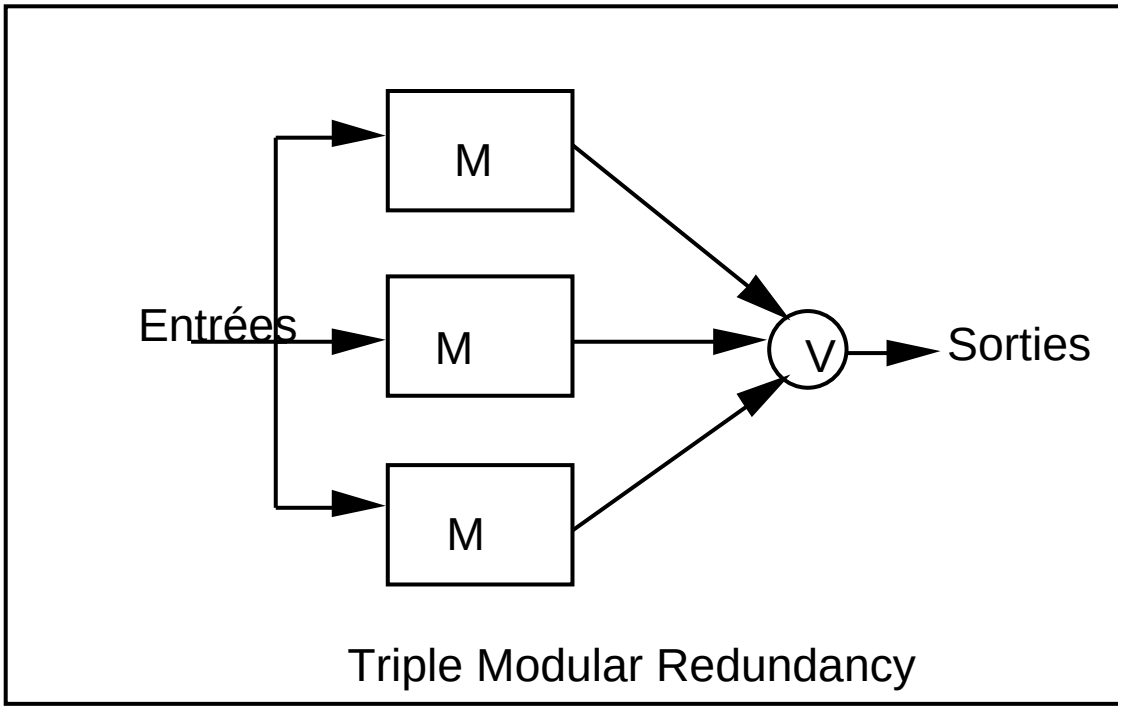
- Hypothèse de panne byzantine
  - 2 modules fonctionnellement identiques
  - 1 voteur
  - Le système fonctionne ⇔ les deux unités produisent des sorties cohérentes et le voteur fonctionne
  - Après détection d'une erreur le système s'arrête (on se ramène alors à une panne franche) => Il est donc nécessaire que l'application admette un état de panne franche (arrêt, état passif sûr)



Architecture triplex (1)

TMR en redondance statique

- TMR est la plus générale des techniques de masquage matériel.
- Le voteur lit les trois sorties et élabore une sortie unique



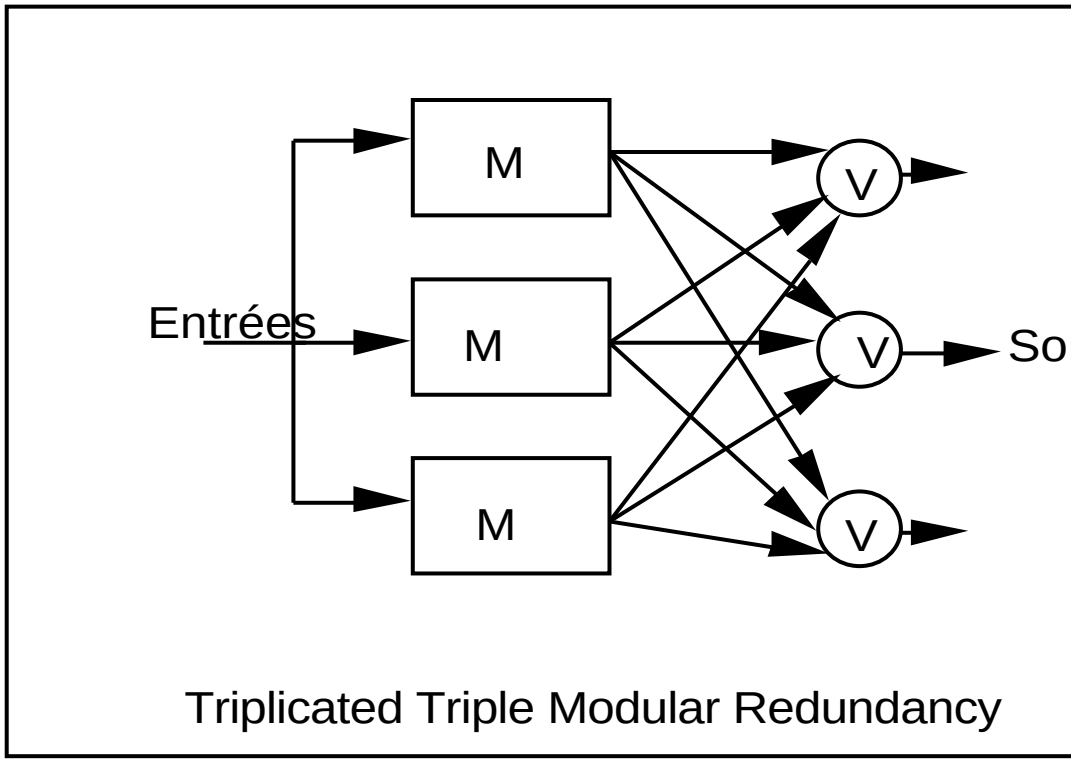
Architecture triplex (2)

- Caractéristiques du TMR
  - Toutes les fautes conduisant à des erreurs sur 1 SEUL module sont tolérées
  - Aucune faute concernant le voteur n'est tolérée
- Versions
  - Cas de pannes d'usure, possibilité d'utiliser des versions identiques
  - Cas des fautes de conception, nécessité de mettre en œuvre de la diversité
- Voteur
  - Acquisition des sorties des modules
  - Synchronisation des modules
  - Election de la valeur finale (résultat)

Architecture triplex (3)

Triplicated TMR

- Amélioration de la fiabilité du voteur
- Deux des trois sorties du système sont correctes si deux paires (M,V) fonctionnent correctement.



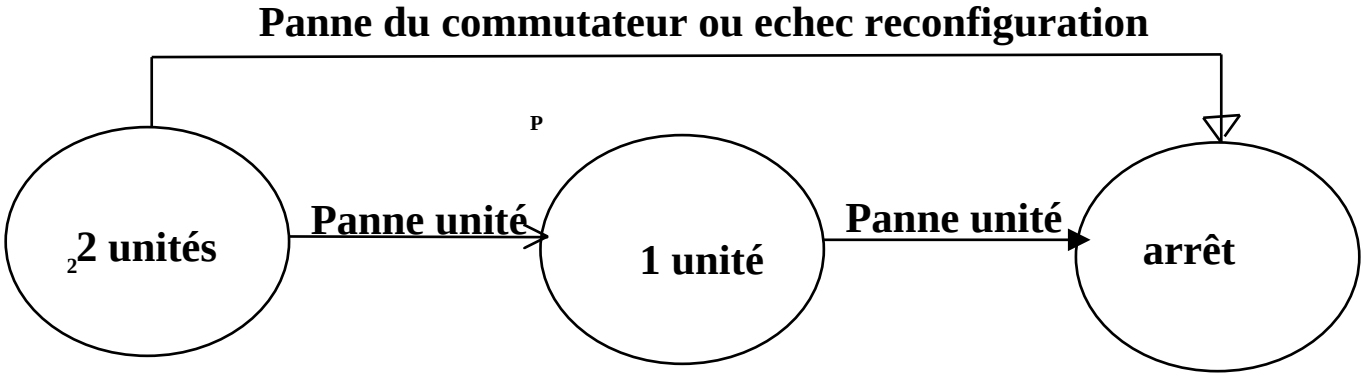
Triplicated Triple Modular Redundancy

Architecture triplex (4)

- ***Problèmes d'implantation du TMR***
  - Le problème majeur est la **synchronisation** entre modules. Une des approches consiste à avoir une **horloge physique commune,** ce qui fait de cette horloge un point critique.
  - La détection des fautes peut être implantée par un ensemble de détecteurs de désaccords : un détecteur de désaccord est activé si la sortie du module auquel il est attaché est différente de celle du voteur.
- ***Avantages et inconvénients***
  - Les avantages du schéma TMR ainsi implanté sont les suivants :
    - - masquage immédiat des fautes temporaires et permanentes
    - - détection intégrée au masquage
    - - conversion facile d'un système non redondant à un système TMR.
  - Les inconvénients sont liés :
    - - à la possibilité de pannes de mode commun, qui peuvent provoquer un vote réputé "bon" avec des résultats erronés
    - - aux difficultés de synchronisation si l'on veut utiliser un autre mécanisme qu'une horloge commune.

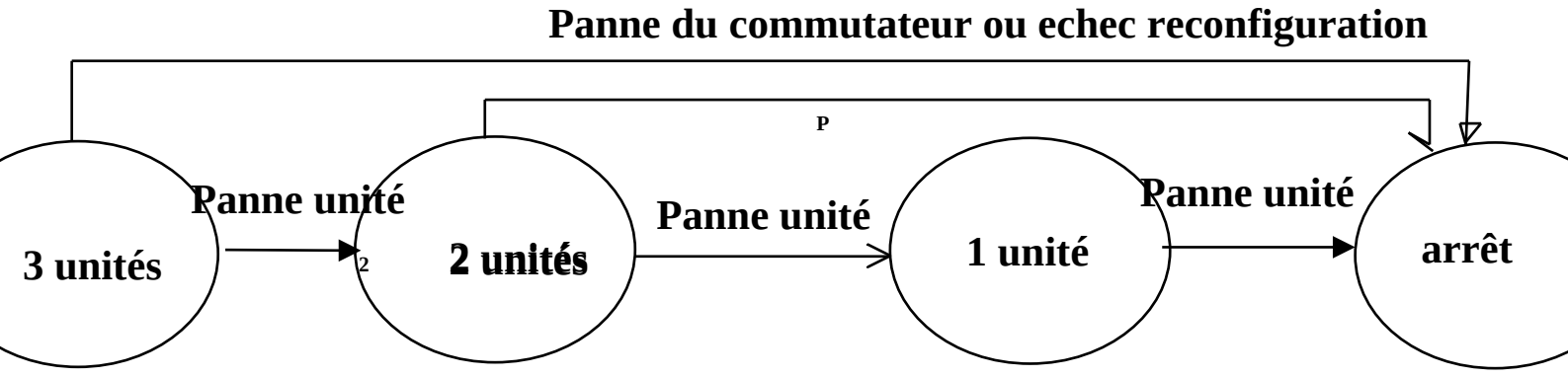
Architecture 1/2

- **Hypothèse de panne franche**
  - 2 modules fonctionnellement identiques
    - Eventuellement chacun des modules contient des autotests permettant de se ramener à une panne franche en cas de détection d'erreur
  - 1 commutateur
  - Le système fonctionne ⇔ les deux modules fonctionnent (produisent des sorties) et le commutateur fonctionne ou un module fonctionne et un module et le commutateurs sont défaillants
  - 1 panne franche tolérée (Fail operational/Fail Silent)



Architecture 1/3

- Hypothèse de panne franche ou omission
- 3 modules et 1 commutateur indépendant
  - Tolérance 2 pannes franches
  - Fail operational/Fail operational/Fail silent



Architecture 2/3

- Tolérance 1 panne byzantine ou 1 panne temporelle
- 3 modules actifs, 1 voteur indépendant
- Le système fonctionne correctement ⇔ deux unités sur trois au moins produisent des sorties cohérentes et le voteur fonctionne
  - Propriété Fail Operational à première erreur
- L'unité « défaillante » doit être passivée par le voteur
- Après détection d'une deuxième erreur, le système s'arrête => l'application doit admettre un état passif
  - Propriété Fail Operational, Fail Silent

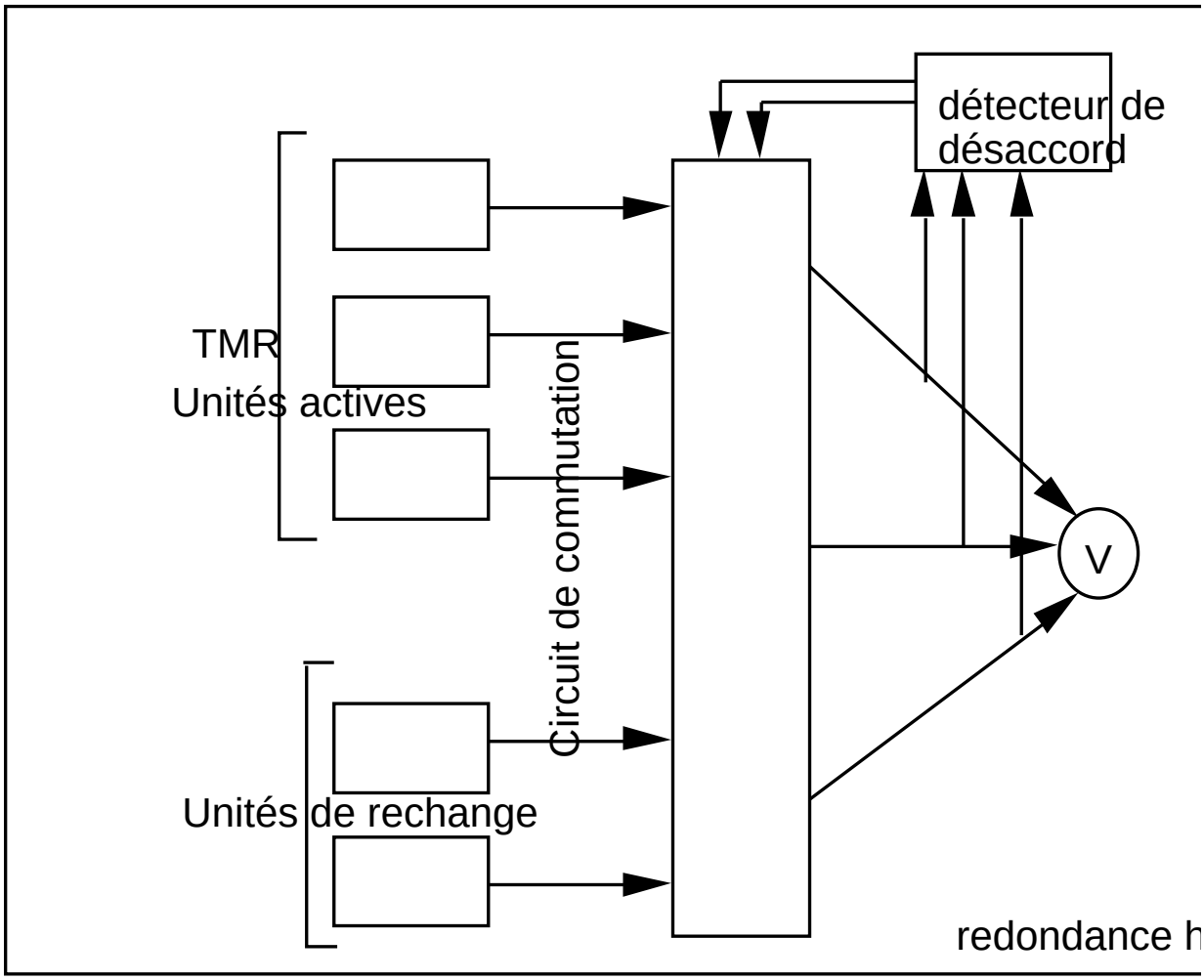
Architecture 2/4 (1)

- Tolérance 1 panne byzantine ou temporelle
- 4 modules actifs et un protocole de vote DISTRIBUE
  - Les modules doivent être synchronisés
  - A chaque étape, ils échangent leurs sorties et procèdent à un vote (en général vote majoritaire)
    - La valeur élue est celle qui correspond à 3 sorties cohérentes
  - Toute unité qui détecte que sa propre sortie est incohérente avec la valeur élue doit se passiver
  - En cas de passivation d'une deuxième unité, le système s'arrête ou se met en état de repli (Fail operational/fail silent)

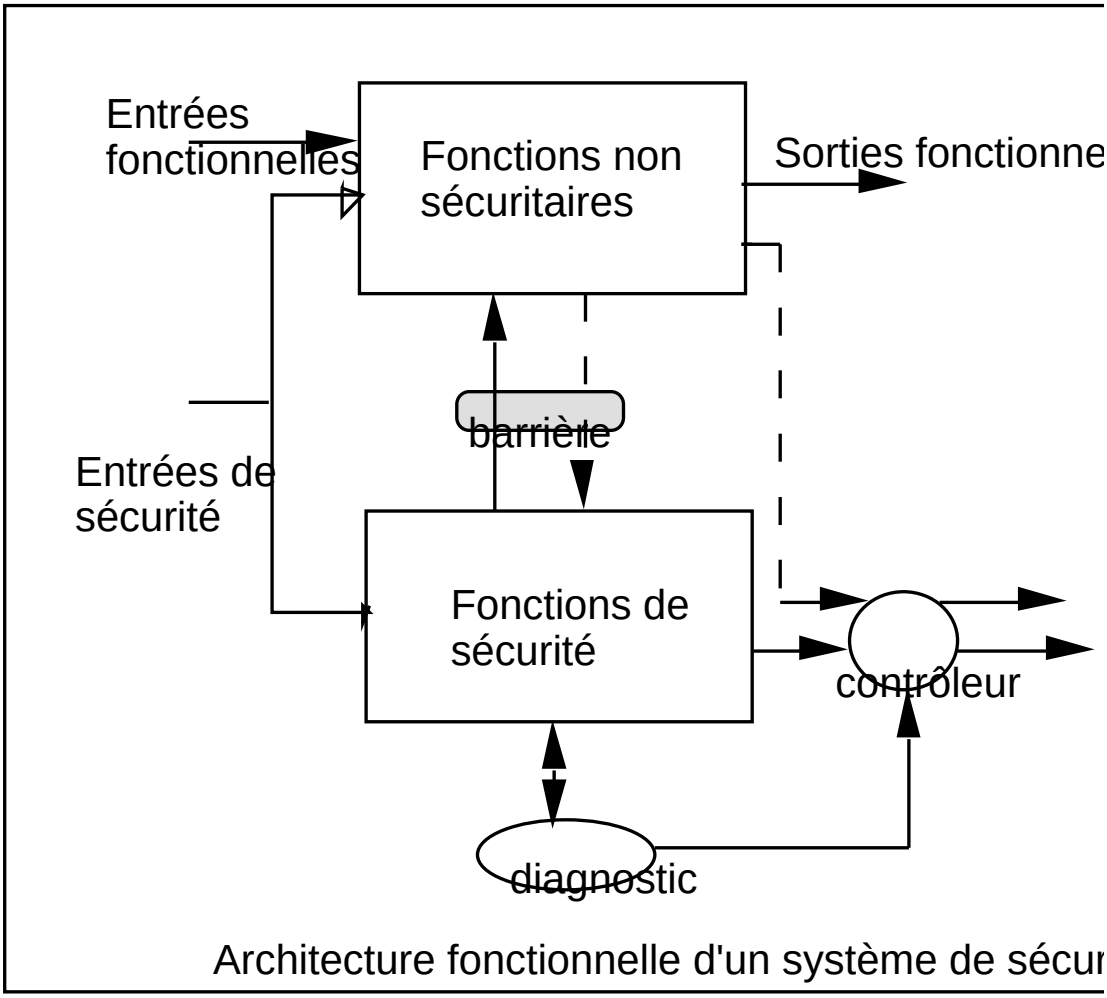
Architecture 2/4 (2)

- Tolérance 2 pannes
- 4unités actives et 1 voteur indépendant
- Vote majoritaire
- Passivation de l’unité défaillante
- Le système fonctionne tant que deux unités et le voteur fonctionne
  - FO/FO/FS

Architectures N-MR

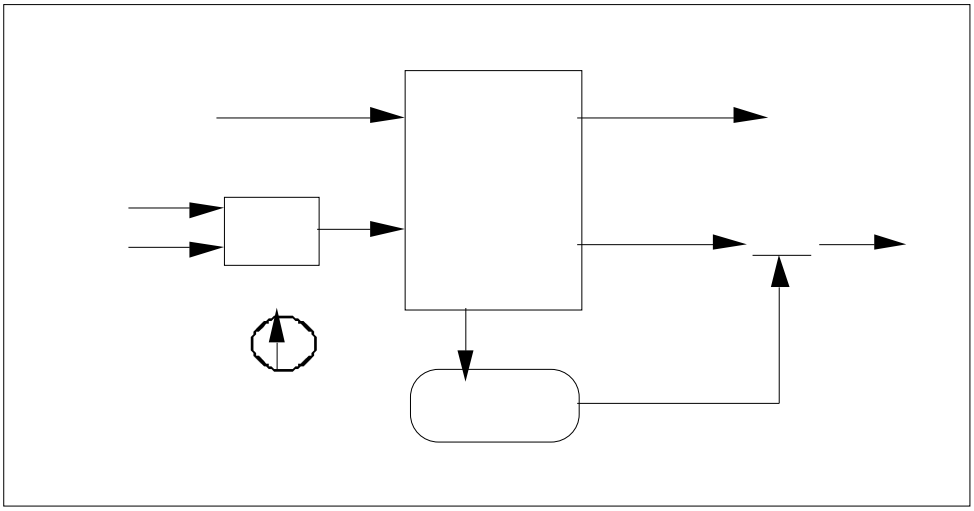


Architectures de sécurité



Architecture fonctionnelle d'un système de sécur

exemple de détection d’erreur par signature



Mise en œuvre : le monoprocesseur codé (1)

- Le traitement est réalisé en utilisant un jeu d'instructions particulières (les OPELs) qui réalisent le calcul et déterminent la signature de ce calcul, fonction des données, de la séquence et de la date.
- Le contrôleur dynamique détecte les erreurs en comparant les signatures produites à des signatures valides "précalculées".

- codage
  - Les données d'entrées sont codées suivant le principe suivant :
  - x est codée par X= (x, CTR(x))
  - avec CTR(x) = -rx +Bx +D où rx est le reste de la division de 2**k*x par un nombre premier A, Bx est la signature de X et D est la date.
- X est congru à Bx+D modulo A.

Mise en œuvre : le monoprocesseur codé (2)

- Détection
  - Lorsqu'on exécute un calcul, on produit la signature de ce calcul. Par exemple soit Z=X+Y; Z est congru à Bz+D modulo A.
  - Erreur sur les opérandes
    - Supposons que X ait été modifié : X = hA + S + D avec S<>Bx alors on aura avec une forte probabilité (S+By+D) moduloA <> (Bx+By+D)moduloA, ce qui permet de détecter une erreur dans les entrées.
  - Erreur d'exécution par exemple l'opération réalisée est - au lieu de +
    - avec une forte probabilité (Bx-By+D) modulo A <> (Bx+By+D) moduloA ce qui permet de détecter cette erreur.

Mise en œuvre : le monoprocesseur codé (3)

- Principe de mise en oeuvre :
  - Lors de la génération du code, les signatures sont précalculées à D près pour toutes les sorties puis stockées dans le contrôleur dynamique. En ligne les signatures sont recalculées.
- Démonstration de sécurité :
  - Basée sur l'évaluation probabiliste : on évalue la probabilité de non détection d'une défaillance.

Architectures de sécurité basées sur la redondance

- Architecture 2/3
  - Sécurisation du voteur (par exemple en utilisant une technologie MPC)
  - Diversification des unités (décalages temporels, gestion mémoiure diversifiée) pour éviter les modes communs
  - Sécurisation des transmissions
  - Sécurisation des E/S

Structures types pour le matériel

Sémantique de panne considérée	Aucune dégradation fonctionnelle . · ER = (sortie erronée ou sortie absente ou panne temporelle).	Panne d'omission admissible · Détection des sorties incorrectes et possibilité de reconfiguration. · ER = (sortie erronée ou panne temporelle)	Panne franche admissible · Détection des arrêts (non production) et possibilité de reprise · ER = sortie erronée.
Principes d'architecture	· N-plication , Vote · Détection et masquage d'erreur N>=3 si les UT sont non autotestables, N>=2 sinon · Synchronisation des UT. · Redondance des horloges.	· N-plication , Vote · Détection d'erreur N>=2 si les UT sont non autotestables, Synchronisation des UT. Redondance des horloges. N=1 si l'architecture est autotestable (exemple du mono processeur codé)	· N-plication , Vote · Détection d'erreur N>=2 si les UT sont non autotestables, N>=1 sinon · Synchronisation des UT.

Structures types pour le matériel (suite)

Nombre de pannes devant être tolérées	**0 panne**	**1 panne**	**2 pannes**
hypothèse de panne franche ou d'omission	matériel testé pour la durée de mission	2 processeurs fonction de détection d'unité défaillante et commutateur. Redondance active.	3 processeurs fonction de détection d'unité défaillante et commutateur. Redondance active.
hypothèse de panne byzantine ou temporelle	Architecture monoprocesseur matériel prouvé en sécurité intrinsèque s'il existe un état de sécurité passif. Architecture bi-processeurs actifs avec vote s'il existe un état de sécurité passif. Sinon faire une analyse de fiabilité et démontrer que la probabilité de défaillance non sûre est négligeable.	2/3 avec vote ou 2 architectures bi-processeurs avec vote, ces architectures étant en redondance active et commutées. Architecture à 4 processeurs en redondance active sans voteur matériel, avec une fonction de vote logiciel distribuée.	2/4 avec vote ou 3 architectures bi-processeurs avec vote, ces architectures étant en redondance active et commutées. Architecture à 7 processeurs en redondance active sans voteur matériel, avec une fonction de vote logiciel distribuée.

Tolérance aux fautes pour le logiciel

- Traitement d’erreur
  - Eviter par détection aussi rapide que possible la propagation d’une erreur
    - Assertions exécutables,
    - Tests d’acceptance
    - Traitements d’exception
    - Programmation défensive
- Mécanismes de redondance logicielle et ou de diversité
  - Diversification
  - Evitement des modes communs
  - Coûts de la diversité

Tolérance aux fautes logicielles (2)

- Diversification fonctionnelle
  - 2 (ou N>=2) variantes et 1 organe de décision (voteur) pour décider de l'acceptabilité des résultats
- Niveaux de diversification
  - Diversification des spécifications
    - Exemple de méthodes mathématiques ou numériques qui peuvent « se contrôler » mutuellement, calculs « inverses »…
  - Diversification de conception/codage
    - Langages différents,
    - Equipes distinctes

Redondance logicielle (fautes de conception)

- Blocs de recouvrement
  - ensure Assertion
    by
        Block1
    else by
        Block2
  - else raise exception

  - Assertion est une assertion exécutable (un code portant sur les résultats des calculs)
  - Block1 et Block2 sont deux versions d'un même programme, exécutées séquentiellement (et si test 1 passe, l'alternat 2 n'est pas exécuté).

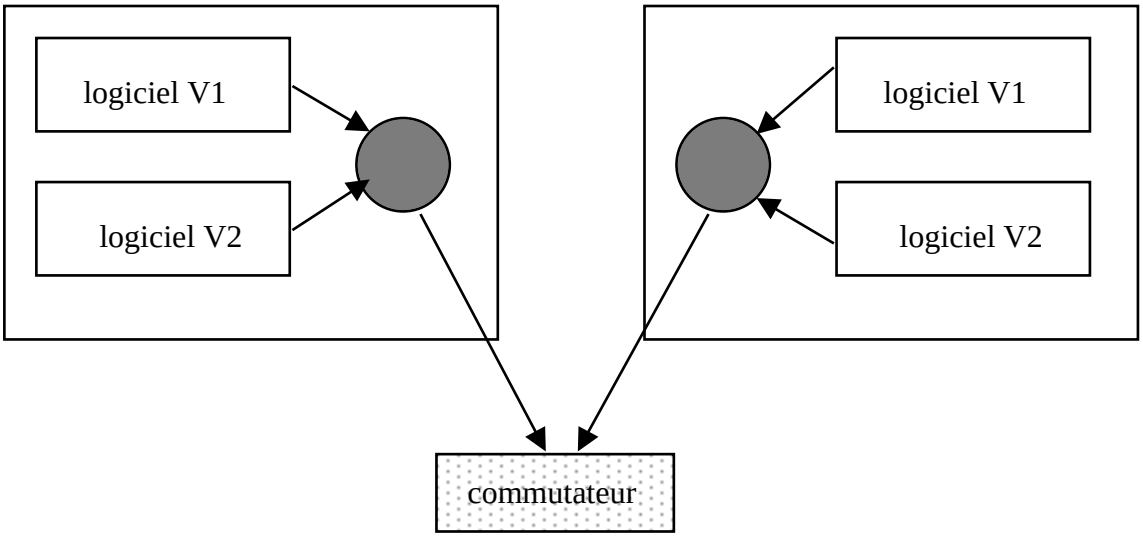
Redondance logicielle (suite)

- N-version programming
  - exécution parallèle des différentes versions
  - vote sur les résultats de toutes les versions

- programmation N-autotestable
  - exécution parallèle de deux (au moins) composants logiciels autotestables
  - un composant autotestable :
    - 1 version et une assertion (acceptance test)
    - 2 versions et un vote

Exemple d’architecture tolérant les fautes (faute de conception et physique)

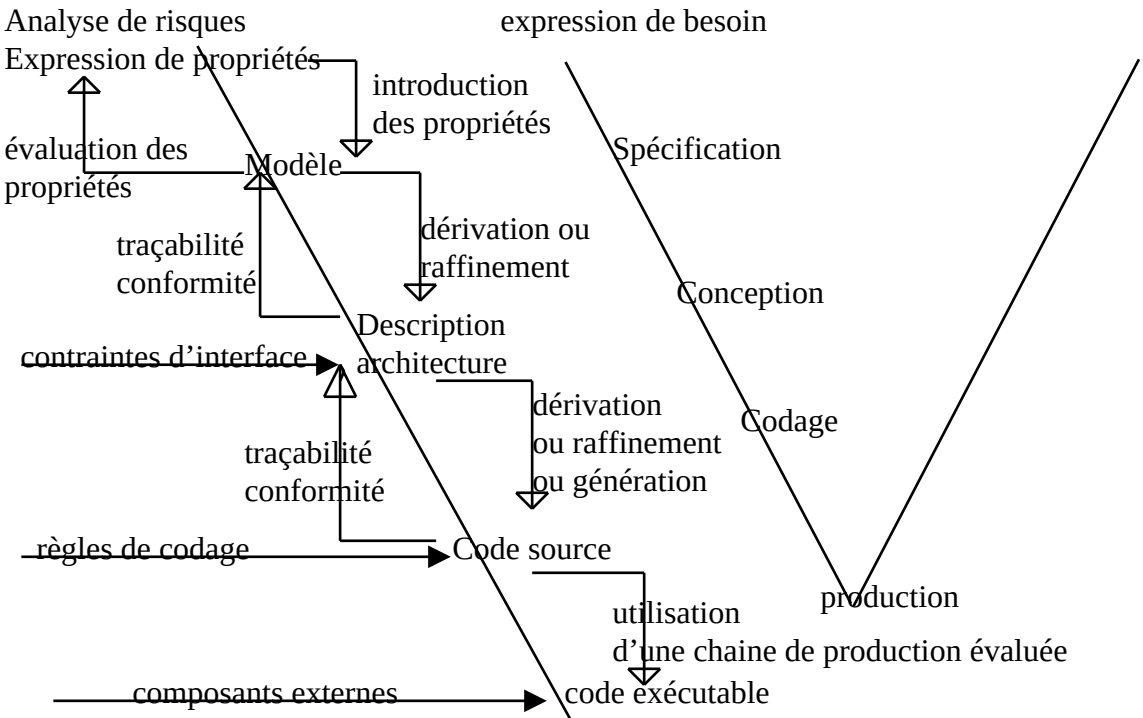
- PA 320
  - 2 versions logicielles en // avec vote
  - sur 2 calculateurs matériels avec commutation



3. Evitement des fautes : Construction de logiciels sûrs

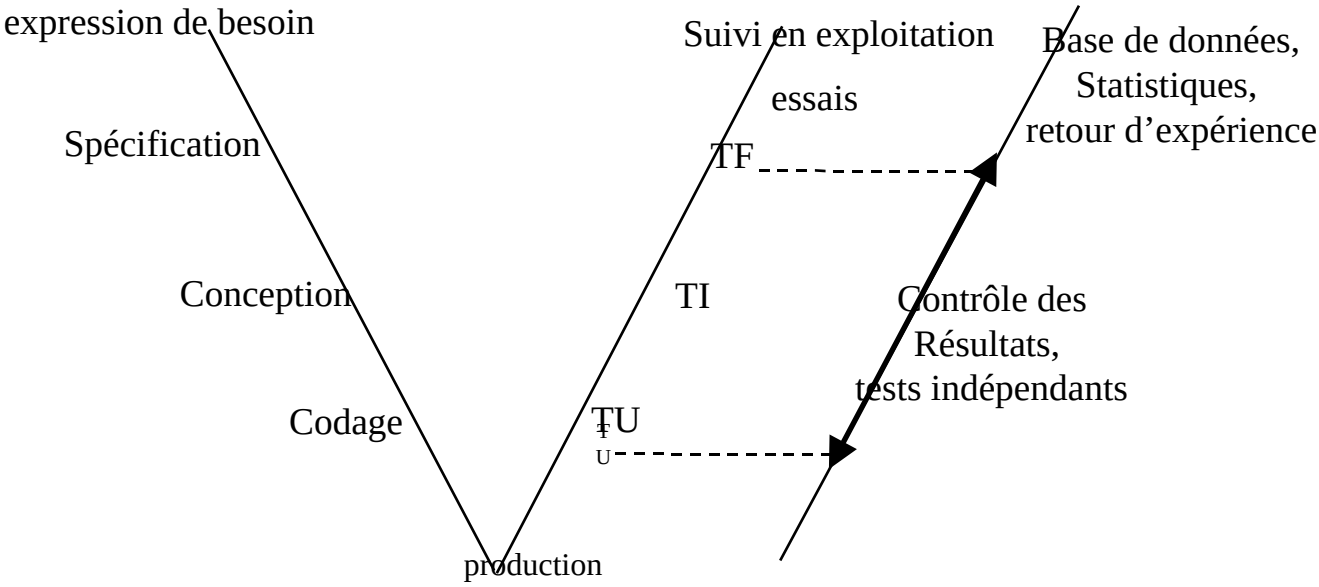
Techniques de prévention (1)

- Cycle de développement et de V&V
  - construction de la sûreté de fonctionnement dans les phases descendantes



Techniques de prévention (2)

- Cycle de développement et de V&V
  - construction de la sûreté de fonctionnement dans les phases de validation : élimination des erreurs et prévision



Techniques de prévention (3)

- analyse des besoins
  - evitement des erreurs de frontière avec l'environnement et des erreurs d'incomplétude ou non conformité aux besoins. Il s'agit d'une phase de reformulation des besoins. Les techniques de prévention relèvent de l'organisation et de l'analyse
    - qualité, expertise des équipes de spécification
    - étude de l'environnement (physique, humain)
    - analyse fonctionnelle (fonctions système, scénarios)
    - analyse objet (use case, scénarios)
    - analyse de risques (identification des événements redoutés, décomposition de ces événements sur les fonctions,...)
    - quantification des risques

Techniques de prévention (4)

- méthodes de spécification
  - Niveaux de formalisation
    - spécification textuelle + revue et prototypage
    - spécification semi-formelle (SADT, SART, OMT/UML) statique et dynamique + revue + prototypage ou déroulement manuel de scénarios
    - spécification formelle + exploration de modèle (RDP, AEFC + simulation exhaustive), évaluation de propriétés (prédicats, invariants ou logique temporelle),
    - spécification formelle et preuve de propriétés (B,)

Techniques de prévention (5)

- méthodes de spécification (suite)
  - de très nombreux langages et outils associés:
    - semi formel : SART, UML, StateCharts, HOOD
    - formel basé sur AEF communiquants RDP, Estelle, Lotos, SDL
    - formel basé sur approche synchrone
      - orienté flot de contrôle ESTEREL,
      - orienté flots de données LUSTRE,
      - SIGNAL
    - formel et preuve
      - Langage B,
      - VDM

Techniques de prévention (6)

- Conception et codage
  - de nombreux environnements de production basés sur des langages plus ou moins formels (ateliers de génie logiciel)
  - règles de dérivation du modèle de spécification vers le code
  - règles de codage et d'annotation des programmes
  - générateurs de code
  - utilisation de modèles de conception visant à simplifier la structure et l'écriture des applications : par exemple le modèle client/serveur avec utilisation de middleware offrant divers services (RPC, COSS de Corba,...)
  - utilisation d'assertions
  - inspection de code
  - analyse de code
  - analyse statique et interprétation abstraite

Techniques de prévention (7)

- Evitement des fautes liées à la technologie
  - Implantation des logiciels sur un noyau exécutif
    - gestion des tâches et ordonnancement
    - respect des échéances à l'exécution
      - => choix du noyau temps réel
      - => validation politique d'ordonnancement (démonstration)
      - => évaluation prévisionnelle politique d'ordonnancement (modèle de performance)
  - Chaine de production de l'exécutable
    - compilateur "validé" voire "certifié"
    - gestion de configurations (cohérence mutuelle des composants pris en compte pour produire l'exécutable)

Techniques de prévention (8)

- Evitement des fautes liées à la technologie (suite)
  - choix des composants matériels
    - technologie
    - processus de fabrication
    - robustesse vis à vis de l'environnement (charge, microcoupures, environnement electromagnétique...)
- Procédures de maintenance, notamment de maintenance préventive

4. Quelques Méthodes et Outils SDF logiciel pour la construction de logiciels sûrs

Thèmes abordés

- Spécification et Validation par exploration de modèles
- Environnements de conception formelle et de preuve
  - cf exemple METEOR dans Etudes de Cas
- Outils d ’analyse statique et de détection des erreurs d ’exécution
  - (à insérer)
- Méthodes et Outils de tests
  - à rédiger

Modélisation de spécifications par AEFC

- Modèle statique :
  - ensemble de modules (blocs pour lds) interconnectés par des canaux de communication (channels pour estelle, chan pour promela, routes pour lds)
  - lien entre modules et canaux : point d'interface (en entrée, en sortie)
  - caractérisation du mode de communication : RDV (multiplicité), File (taille)
  - identification des messages véhiculés par les canaux
- Premier niveau de vérification possible
  - cohérence du modèle
  - pour chaque flot identification de l'emetteur et des récepteurs

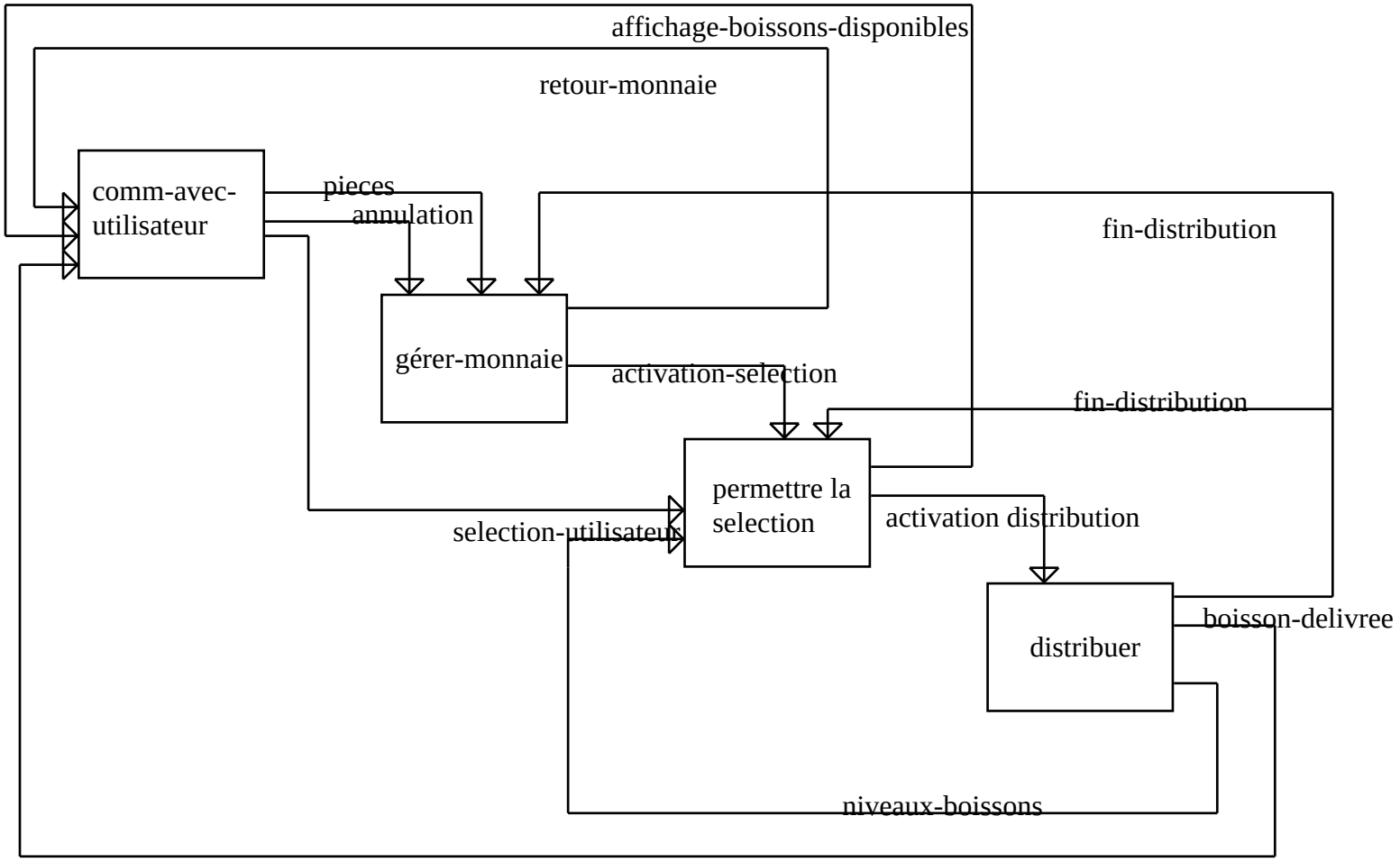
Modélisation des spécifications par AEFC

- Comportement local à chaque processus
  - Chaque module élémentaire est décrit par un AEFC
  - Déclaration de l'espace des états (local)
  - déclaration des variables (locales)
  - les messages sont (en principe) les seuls moyens de partager des données (pas de variables globales)
  - Transitions conditionnées par
    - variables d'état locales
    - réception d'un événement (horloge, message)
  - Les transitions tirables depuis un même état doivent être en exclusion mutuelle (on joue sur les conditions de tir des transitions)

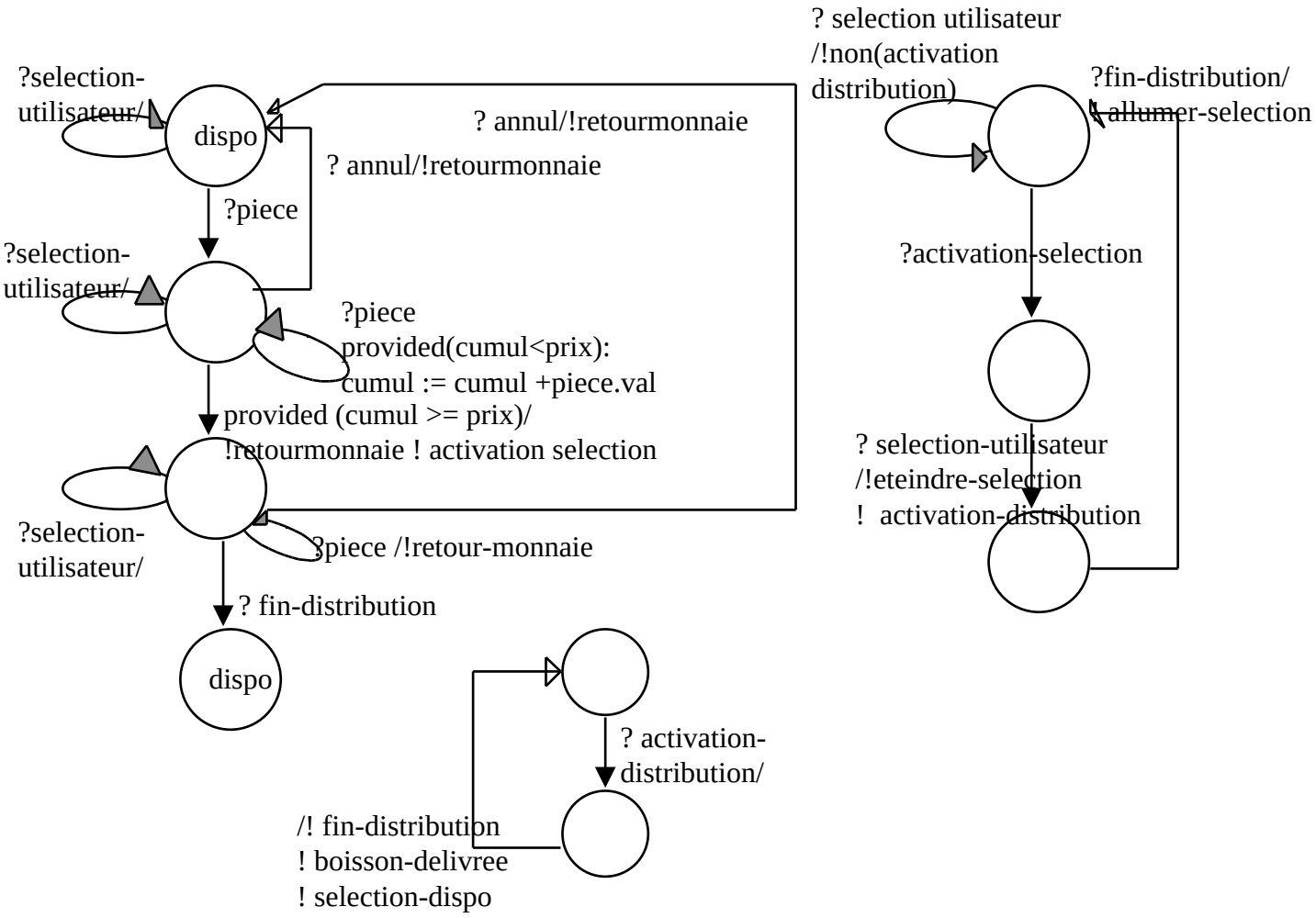
Modélisation des spécifications par AEFC

- Comportement global
  - produit des automates
    - tenir compte du mode de communication => produit synchrone si rdv
- Possibilité de simulation du modèle global ainsi réalisé
  - scénarios (séquences de transitions)
  - recherche d'états puits,
  - contrôle des états des variables et des séquences

spécification système



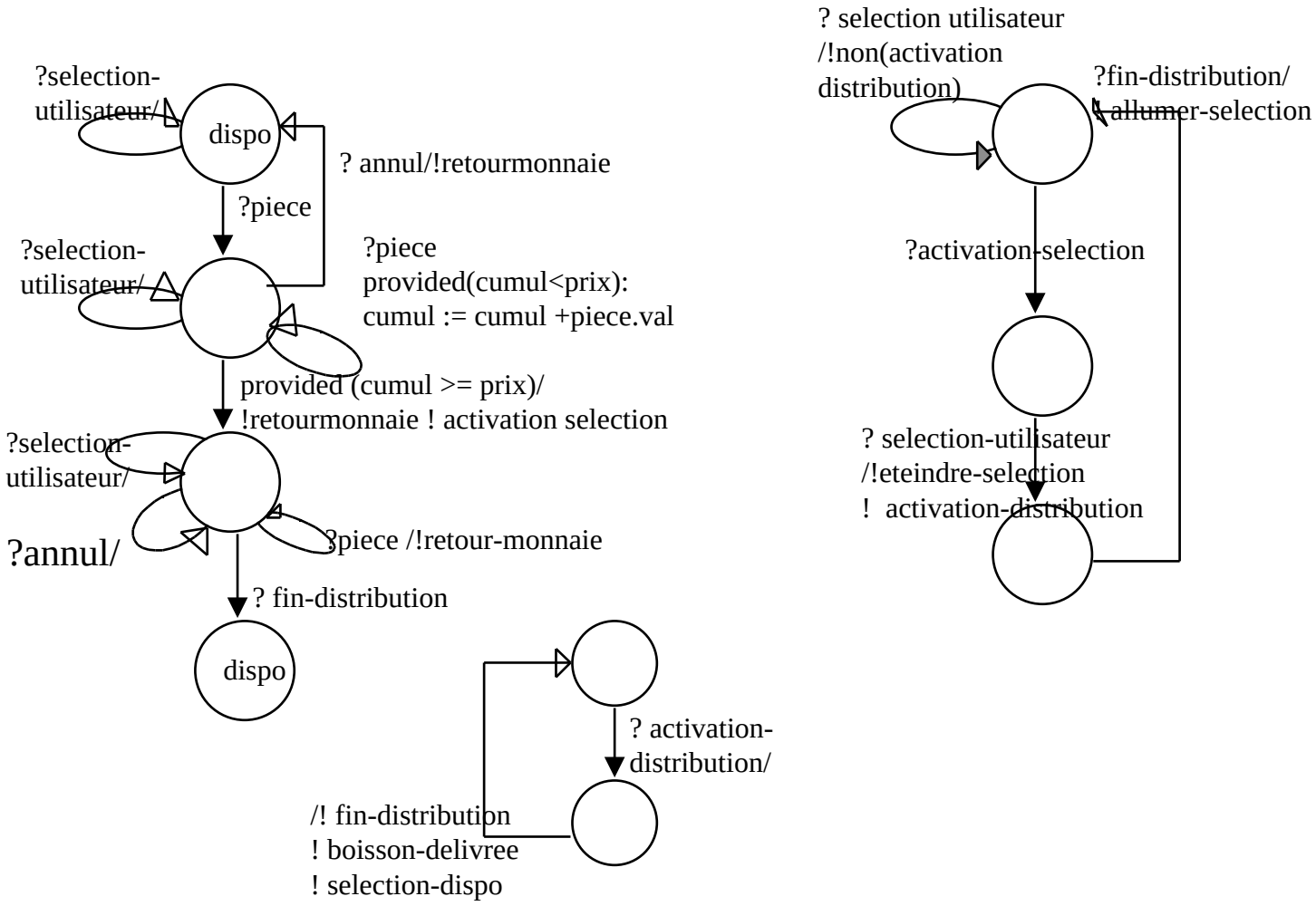
spécification système



Exercice étude du comportement sur AEFC

- Etude du comportement global
  - Produit des automates
    - Rdv
    - Files
  - Etat initial
  - Simulation
    - Construire le graphe des états
    - Trouver un comportement non spécifié (incomplétude)
    - Trouver un comportement erroné (incohérence, non conformité aux propriétés attendues)
    - Trouver un cas de blocage .
- Projection sur les messages externes uniquement
  - Quel est le comportement observé ?

Correction des incohérences dynamique/statique



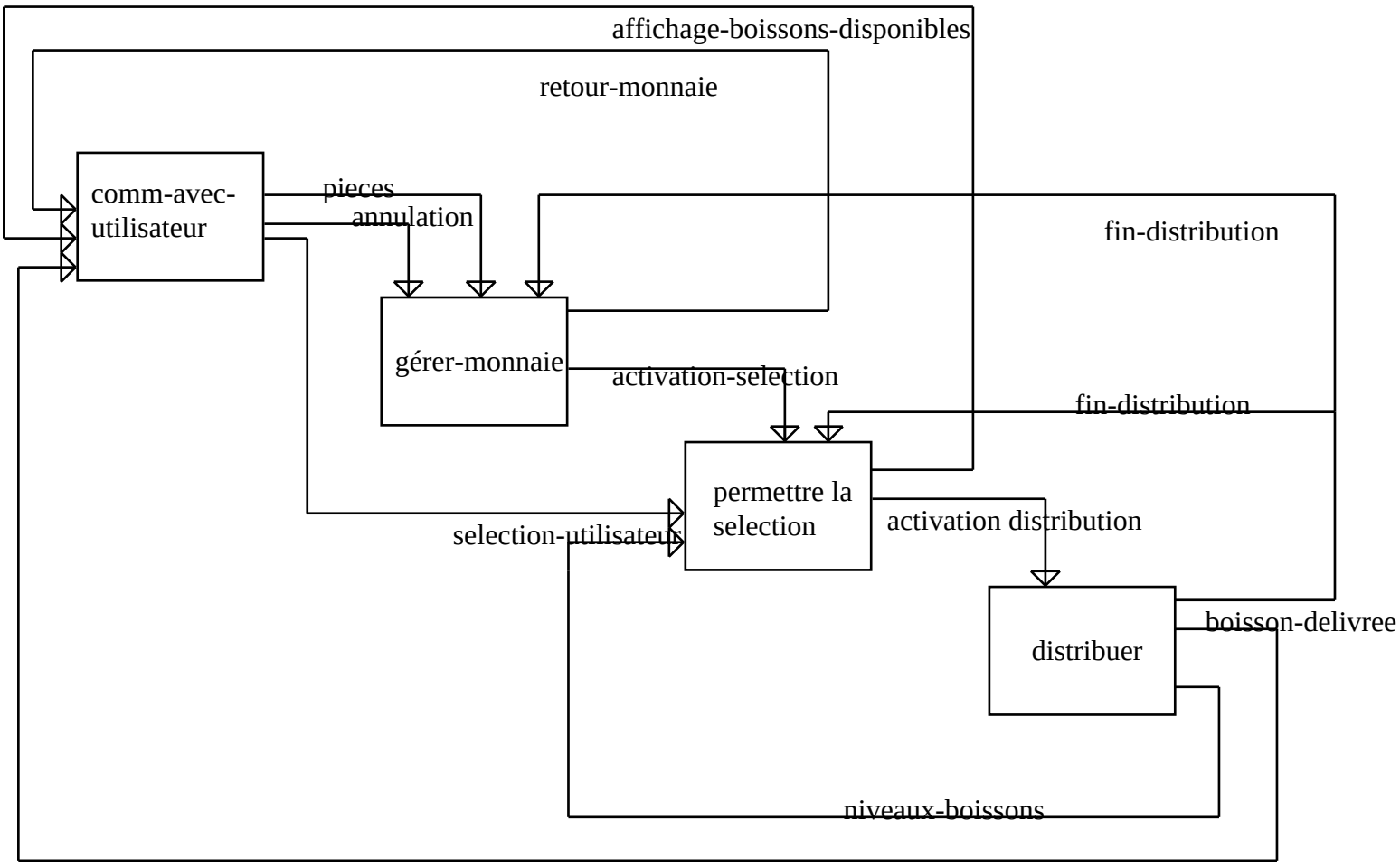
Etude des propriétés (1)

- Propriétés de fonctionnement (informel)
  - 1. Aucune boisson délivrée si le prix de la boisson n'est pas atteint
  - 2. Si le montant a été atteint et si la sélection est faite, la boisson sélectionnée est délivrée
  - 3. En cas d'annulation avant la sélection, la monnaie est rendue
- Traduction des propriétés sur le graphe
  - Analyse locale à chaque automate
    - La propriété 1est contrôlée par l'automate monnayeur : l'emission du message « !activation_selection » est gardée (provided cumul >=prix)
    - La propriété 3 est contrôlée par les transitions conditionnées par la réception du message « ?annul » du monnayeur
      - On constate qu'il subsiste une erreur : la transition dans l'état 3 du monnayeur ne devrait pas provoquer de retour monnaie, car la sélection a déjà été autorisée (ceci est plus restrictif que l'énoncé de P3, si on veut se conformer exactement à P3, il faudra rajouter des éléments (cf suite)

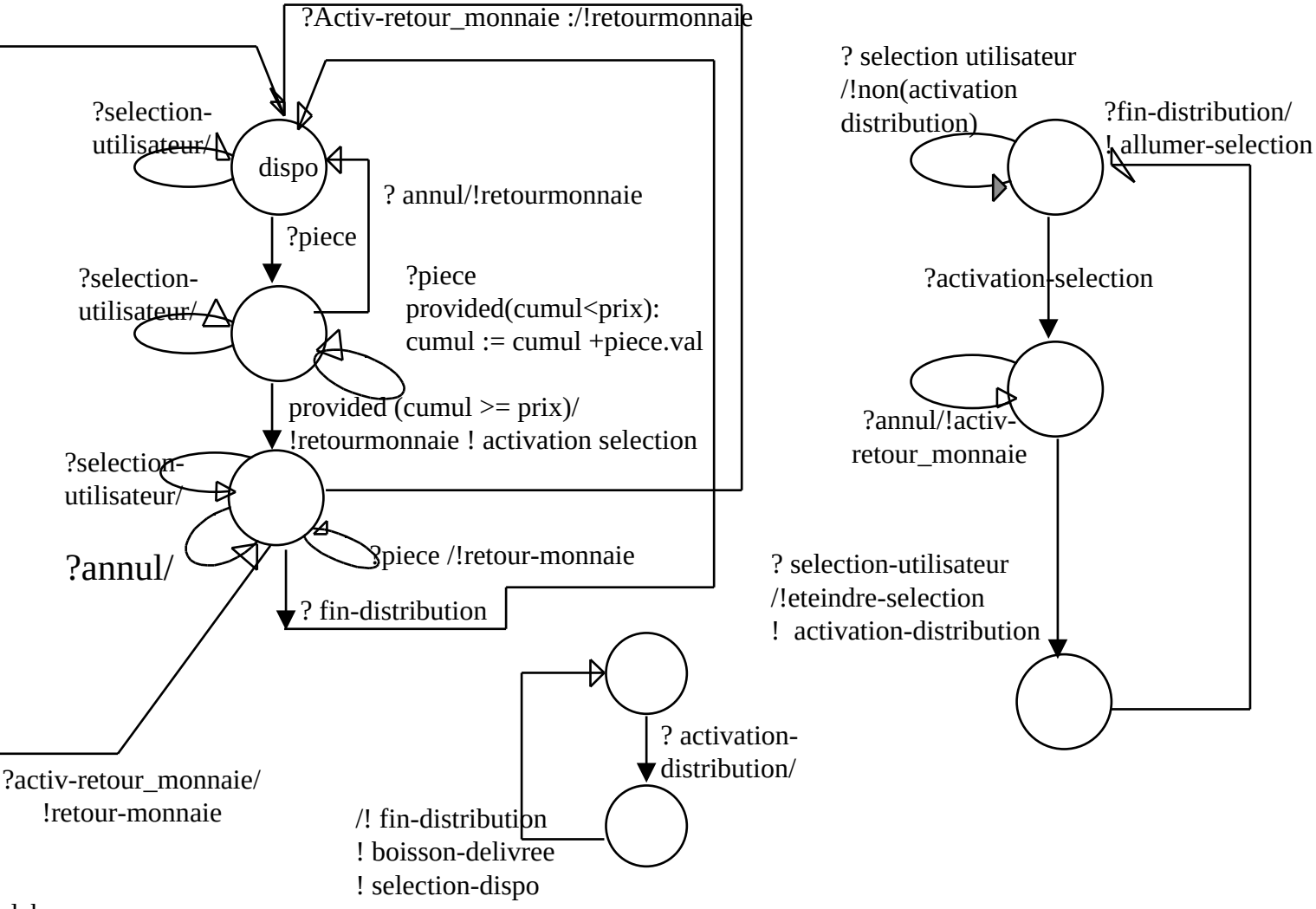
Etude des propriétés (2)

- Analyse du graphe
  - Analyse du comportement global
    - La propriété 2 est vérifiée lorsque le rendez vous « selection-utilisateur /!eteindre-selection !activation-distribution » est réalisé
    - La propriété 3 n'est pas vérifiée
      - Le monnayeur accepte de prendre en compte « ?annul » alors que la sélection est possible et le sélectionneur ne prend pas en compte la possibilité d'annuler
        - Si l'on veut permettre l'annulation avant la sélection, il faut compléter le sélectionneur avec une transition qui traite l'annulation et émet vers le monnayeur une demande de retour monnaie. (remarque cette transition pourrait être utile pour traiter également les cas de sélection de boisson « vide ».

Spécification modifiée (statique)

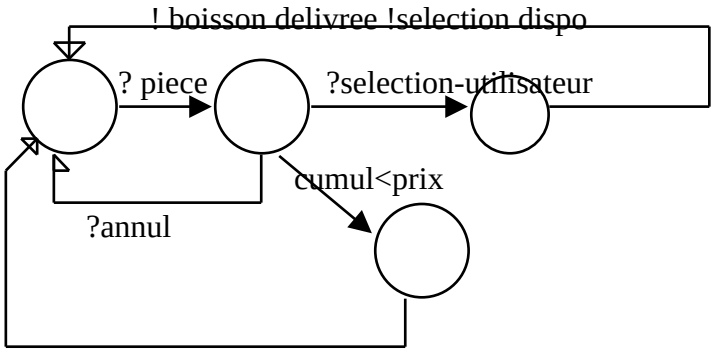


Spécification modifiée (dynamique)



Quelques propriétés attendues

- Scenario nominal vis à vis de l'utilisateur uniquement
  - blocage possible dans la spec actuelle (il manque des événements de délais)
- Scénarios non couverts
  - cumul n'atteint pas prix
- montrer que le nombre de boissons distribuées (occurrences d'activation distribution) est inférieur ou égal au nombre d'activation selection
- retour à l'état initial
- …



Comportements erronés

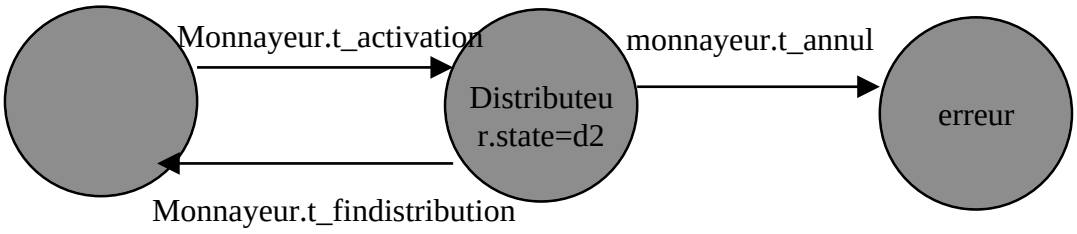
- Dysfonctionnements
  - Omission/panne franche
    - Sélection impossible
    - Arrêt complet de la machine
    - Non retour de monnaie
    - Non distribution de boisson
  - Pannes byzantines
    - Incohérence sélection/boisson
    - Retour monnaie incohérent
  - Pannes temporelles
    - ?
  - performance

Caractérisation des propriétés

- Comportements corrects
  - Vivacité
    - <= Retour à l'état initial global
    - Absence de blocage (deadlocks et livelocks)
  - Propriétés de séquencement / d'atteignabilité d'états
    - Si le système atteint un état où le monnayeur est en M3 (monnayeur.state=m3), alors le système atteint toujours un état où le distributeur est actif (distributeur.state=d2) en une transition ; on peut vérifier
    - (Monnayeur.state = m3) => (distributeur.state = d2)
      - (toujours, mais pas forcément juste après)
    - Never((monnayeur.state=m3)=> (distributeur.state NE d2))

Caractérisation de propriétés (2)

- Franchissabilité de transitions :
  - Si le distributeur est en état d2, il n'est pas possible d'annuler la sélection et de récupérer la monnaie
  - Not (fireable (monnayeur.t_annul, distributeur.state=dé))
  - Observateur associé (partiel) à la propriété



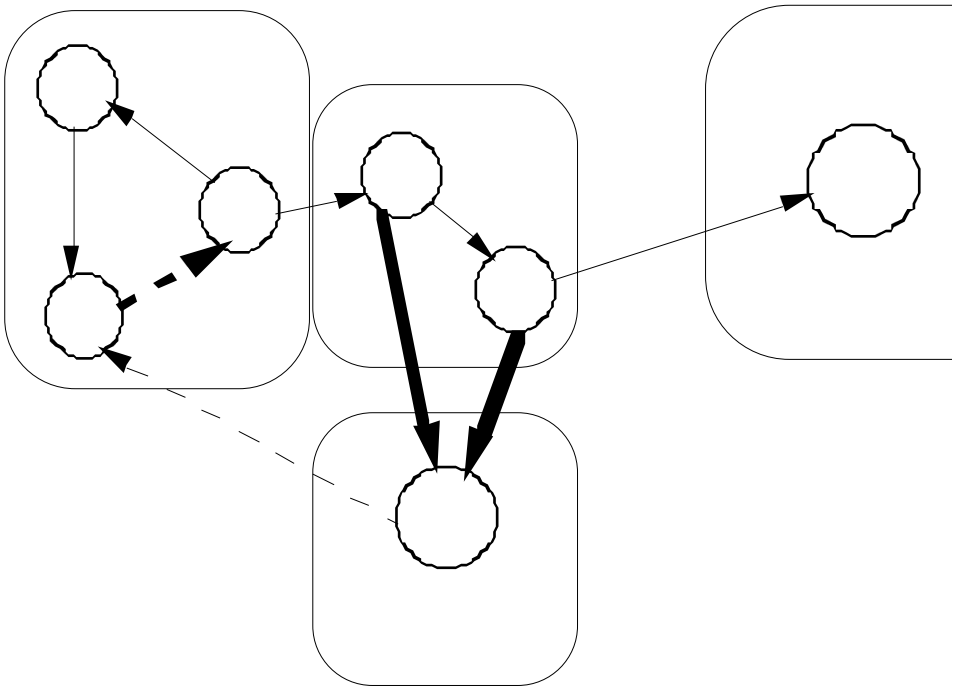
Liens avec les tests fonctionnels

- L’analyse des propriétés, et notamment des scénarios couvre :
  - l’analyse des scénarios vus par l’utilisateur => faire abstraction des transitions internes
  - l’analyse des séquences internes de transitions
- Tests
  - boîte noire : on applique les séquences des entrées et on observe les séquences des sorties
  - boîte grise : on connait la structure en module et on observe les séquences de transitions (messages entre modules) en plus des sorties perceptibles dans l’environnement
  - boîte blanche : on ajoute les variables d’états internes aux modules, ou encore des séquencements internes

Propriétés exprimées à l'aide de formules de logique temporelle

- Cf cours pour logique temporelle : LTL, CTL, PTL notamment
  - inevitabilité
    - il s'agit de montrer en général que quelque chose de bon est "inevitable"
      - application à la recherche de propriétés de sûreté
        - inev (appliquer-FU) rend les états dans lesquels seule cette transition est tirable. Il faut vérifier manuellement que l'ensemble de ces états est complet (recouvrement de toutes les alarmes)
  - potentialité
    - extrait les états origines d'une séquence de transition menant à un état cible donné. (dont on vérifie l'atteignabilité)

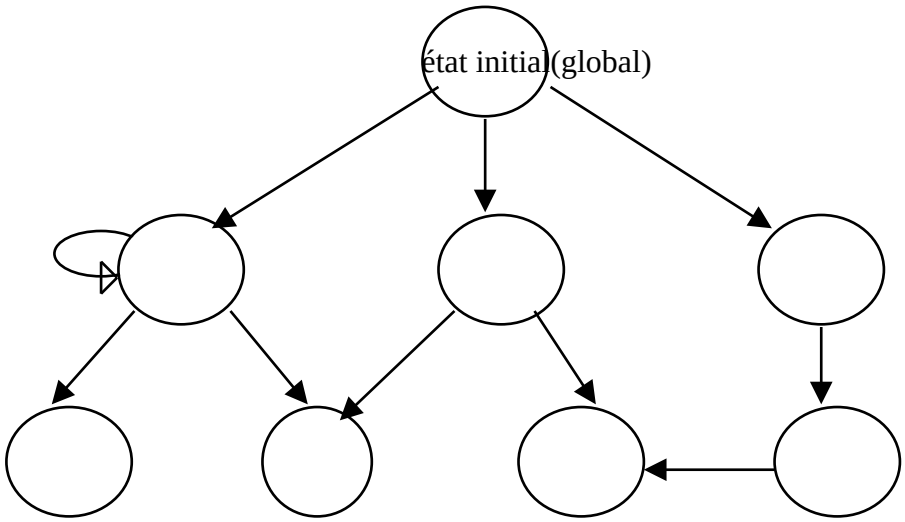
Application à l’étude d’événements redoutés



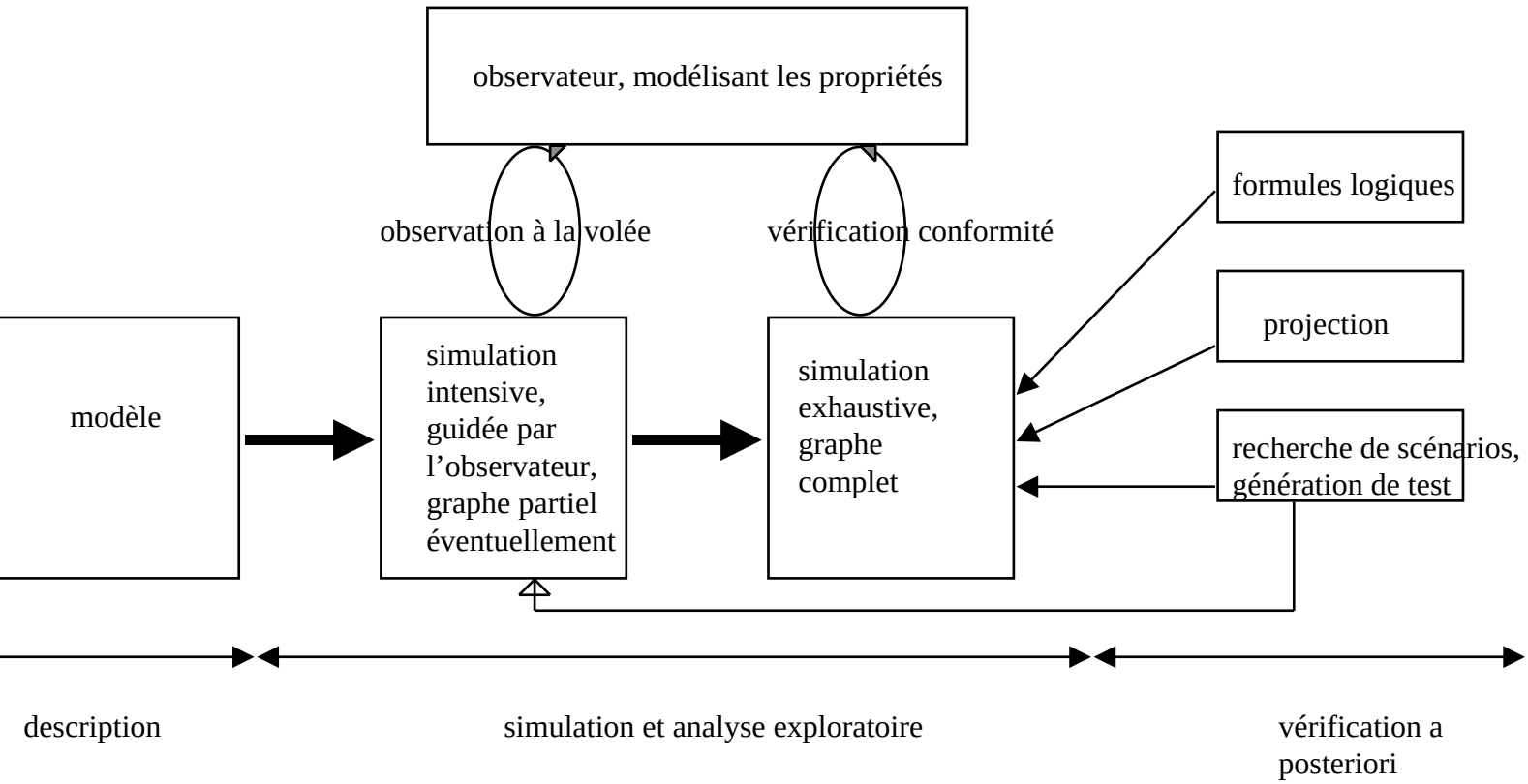
Validation basée sur l ’Exploration de graphe d’états (model checking)

- La simulation est insuffisante pour vérifier que
  - tous les comportements possibles du système sont conformes aux spécifications, notamment de sûreté de fonctionnement
  - aucun événement contraire à la sécurité par exemple ne peut se produire.
- Exploration de graphe d’états
  - simulation contrainte et guidée par un **observateur**
  - evaluation de propriétés
    - à la volée
    - sur le graphe complet a posteriori
  - construction de graphes “équivalents” par opérateurs sur les graphes : projection, réduction

Le graphe d’états

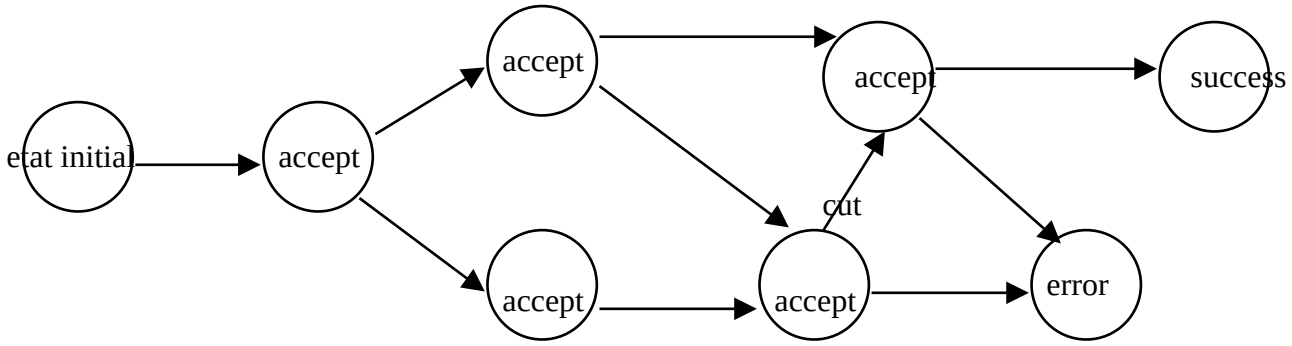


Les différentes techniques d’exploration



Structure d’un observateur

- un observateur est un AEF, qui a des sondes (points d’observation) sur le modèle (événements, variables internes, etats des modules)
- on définit des états accepteurs (qui sont des états de progression vers une situation donnée), des états de succès (propriété cible vérifiée) et des états d’erreur (propriété cible violée)
- on peut couper des branches d’exploration du graphe pour faire des observations particulières

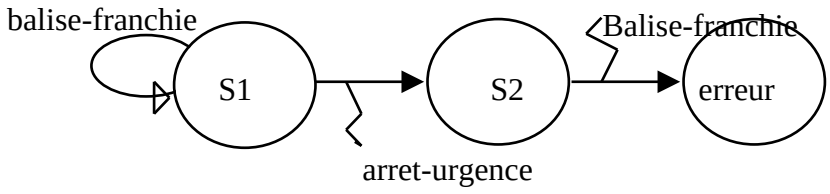


Observateur

- Evolution de l’observateur :
  - déterminée par l’observation des événements survenus lors du dernier pas de simulation
  - exécution synchrone du modèle et du (des) observateur(s)
  - A chaque pas, l’observateur évalue si une de ses transitions est tirable et si oui, il va dans l’état successeur adhoc
- description de scénario attendu par un observateur
  - init, debut du scenario, evénements observables, états de succès = déroulement complet du scénario, états de progression : états observables qui ne contredisent pas le déroulement du scénario, états d’erreur correspondant à l’occurrence d’un événement contraire au scénario.
- très grande simplicité d’utilisation

exemple d’observateur

- Propriété cherchée :
  - après réception du message “arret-urgence”, aucun message balise-franchie ne doit être reçu
  - il s’agit d’un invariant de séquence, qui permet de contrôler l’absence de comportement erroné (et de détecter une erreur résiduelle, violant cet invariant).



Evaluation de propriétés logiques

- Propriétés d’états
  - il s’agit souvent de conditions statiques de cohérence sur des variables d’états du système ou sur des variables du système et de l’environnement
    - à la volée, la simulation s’arrête dès lors qu’on rencontre un état qui vérifie « non (propriété) »
    - en exhaustif, on évalue la formule sur tous les états atteints

Propriétés de sûreté et propriétés de vivacité

- propriété de sûreté
  - exprime qu'un comportement "mauvais" ne se produit jamais au cours de l'exécution du système
    - exemple : absence de blocage, respect d'exclusion mutuelle ou d'une propriété d'état. il s'agit en général d'exprimer un **invariant du système.**
- propriété de vivacité
  - exprime qu'un comportement bon se produit toujours lors de l'exécution d'un système
    - exemple : terminaison, atteinte d'un état de succès (en particulier le retour à l'état initial)

Evaluation de propriétés logiques

- Propriétés d'atteignabilité
  - opérateurs portant sur les états
    - recherche des états à partir desquels une transition donnée est tirable (opérateur fireable)
    - recherche des états atteints directement par le tir d'une transition (opérateur after)
  - opérateurs portant sur les arcs
    - recherche des transitions tirables à partir d'un état source donné (opérateur src)
    - recherche des transitions aboutissant à un état cible donné (opérateur tgt)
- combinaison possible des formules

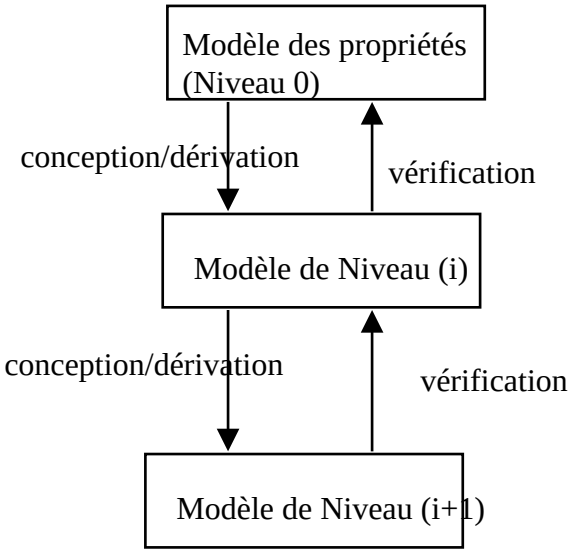
Autres méthodes d’exploration

- Projection et construction de graphe équivalent, réduit
- Logique arborescente
- Evaluation de formules prenant en compte le temps externe (horloge, unités de temps) (automates temporisés)
- Evaluation de formules par recurrence
- BDD (systèmes booléens)
- ...

Apport de la modélisation

- formalisation des propriétés
  - le modèle des propriétés peut servir **d'observateur** vis à vis de la spécification interne des fonctions
  - Il peut également servir à valider des choix de conception générale, notamment vis à vis du temps externe ou des E/S
- injection des propriétés dans la spécification interne

Utilisation des propriétés

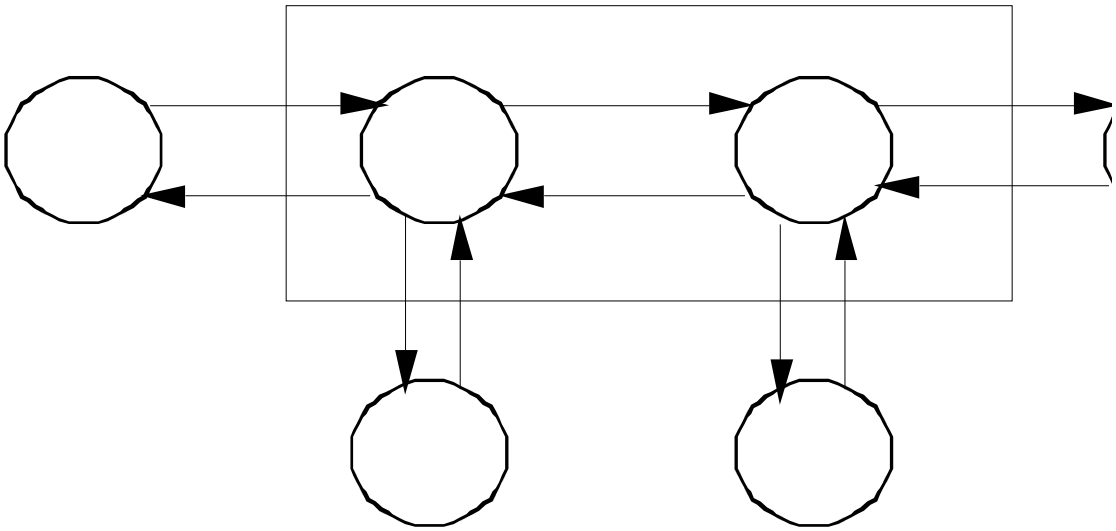


- Notion de niveaux de modélisation, et de raffinement
- Simulation et evaluation de propriétés, à l'aide d'opérateurs logiques
- construction exhaustive du graphe des états et utilisation d'opérateurs de comparaison de modèles (par exemple pour les automates) fondés sur la notion d'équivalence entre graphes
- preuve de raffinement (méthodes formelles)

Exemple de choix de conception

- Distributeur « réparti »
  - Monnayeur sur une machine
  - N (selectionneurs/distributeurs) de boisson sur des machines différentes
  - Un réseau d'échanges de messages entre monnayeur et selectionneur/distributeurs.
- => Fonctions de conception
  - Protocole ?
    - Acquitter chaque réception de message « activation »
    - Conserver les propriétés d'impossibilité d'annuler dès lors que activation émise
    - …
    - (exercice proposer un protocole de type bit alterné et montrer propriétés conservées)
  - Séquentialité des traitements dans la machine distributeur

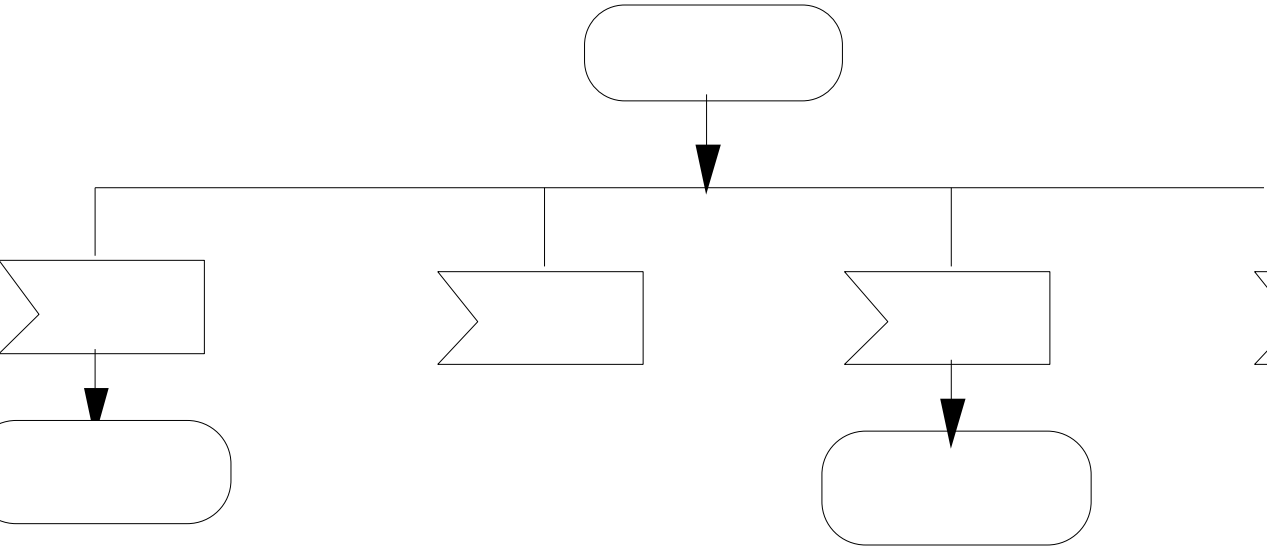
Exercice : formalisation protocole tftp



Exercice :Formalisation du protocole tftp (2)

- 1. Formaliser par diagramme(s) de séquencement de messages échangés entre les entités de l'architecture le déroulement d'une requête de lecture distante et celui d'une requête d'écriture distante. On ne s'intéresse qu'à la phase d'échanges de données (session). On supposera également que cette phase de transfert se fait en échangeant des blocs de données d'une taille fixe Taille_buffer et que le dernier bloc est reconnu parce qu'il a une taille strictement inférieure à "Taille_buffer".
- 2. Identifier les canaux entre processus et les messages véhiculés sur ces canaux avec leur type.
- 3. Spécifier le comportement du processus tftp pour la lecture distante

Exercice : formalisation du protocole tftp (3)



Exercice : formalisation du protocole tftp (4)

- 4. Spécifier les propriétés du protocole tftp.
- 5. Les formaliser par des propriétés de logique temporelle
- 6. Donner un observateur pour tftp (lecture distante seulement).

Exemples d ’environnements de spécification et vérification basée sur ce modèle

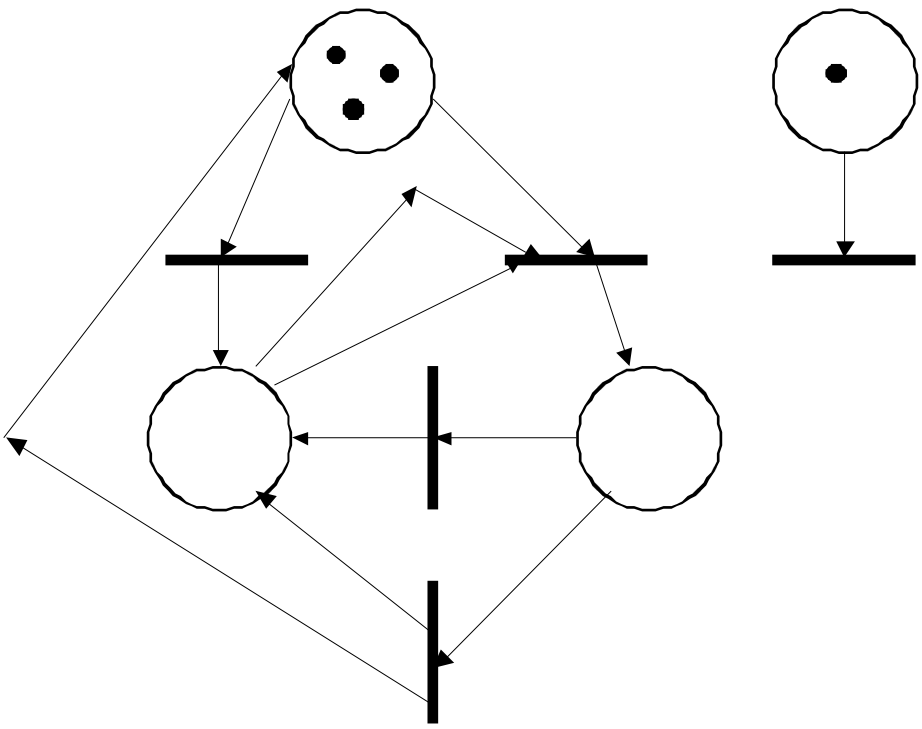
- PVS
  - PVS implante d ’une part des fonctions de vérification par exploration de modèles mais aussi des fonctions de preuve
- Object GEODE
  - langage sous jacent : LDS
  - simulateur interactif et exhaustif
  - évaluateur de propriétés logiques et logique temporelle
- SCADE
  - langage sous jacent LUSTRE
  - observateurs
- NP_PROVER
  - enoncé de propriétés
  - raisonnement par recurrence

Réseaux de Petri

Réseaux de Petri (présentation informelle)

- Modèle de base
  - R = (P,T,U), PxT TxP U
    - P ensemble fini de places
    - T ensemble fini de transitions
    - W : PxT TxP -> N (valuation)
    - M : P-> N (marquage)
    - M_0 marquage initial
  - Pre : restriction de W à PxT (C^-)
  - Post : restriction de W à TxP (C

Réseaux de Petri, illustration du modèle de base



failed p2
processors

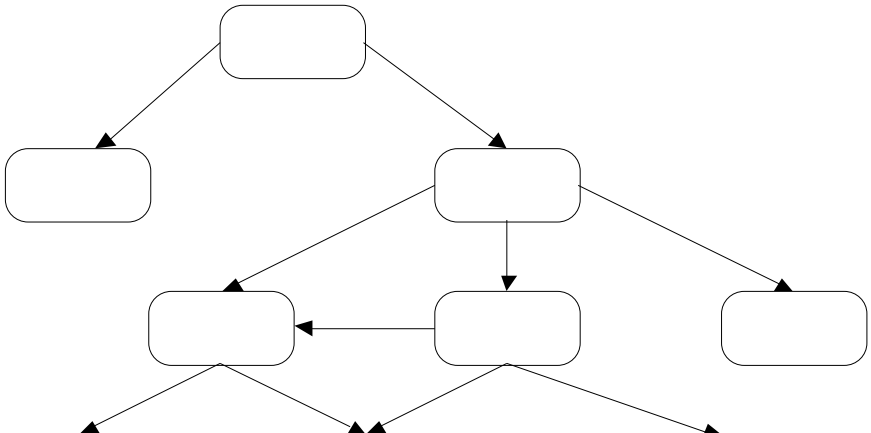
RDP, modèle de base

- Matrice d'incidence
  - $C = C^{+} - C^{-}$
- Etiquette
  - (R,h)
  - h : T -> A {ε}
  - étiquetage d'une suite de transitions est la suite des étiquettes
- Condition de franchissement d'une transition
  - R = (P,T,W) ; t T ; t est franchissable pour M si
    - p P, M(p) >= W(p,t) ; notation M(t >
    - si t est franchissable , M'(p) = M(p) + W(t,p) - W(p,t)
    - (M') = (M) + C (t)

RDP, modèle de base

- Séquence de transitions
  - σ séquence de transitions est franchissable à partir de M et conduit au marquage M' (noté M(σ > M' ssi :
    - si σ=ε, alors M' = M
    - si σ≠ε, σ=$t_1,t_2,...,t_n$
    - il existe une suite de marquages : M_0= M, M_1, M_2, ...,M_n=M'
      - telle que i [1,n-1] $M_i(t_{i+1} > M_{i+1}$.
- langage asssocié à un réseau : séquences franchissables ; expression des comportements possibles du système
- marquages accessibles
- construction du graphe des marquages accessibles

RDP, modèle de base



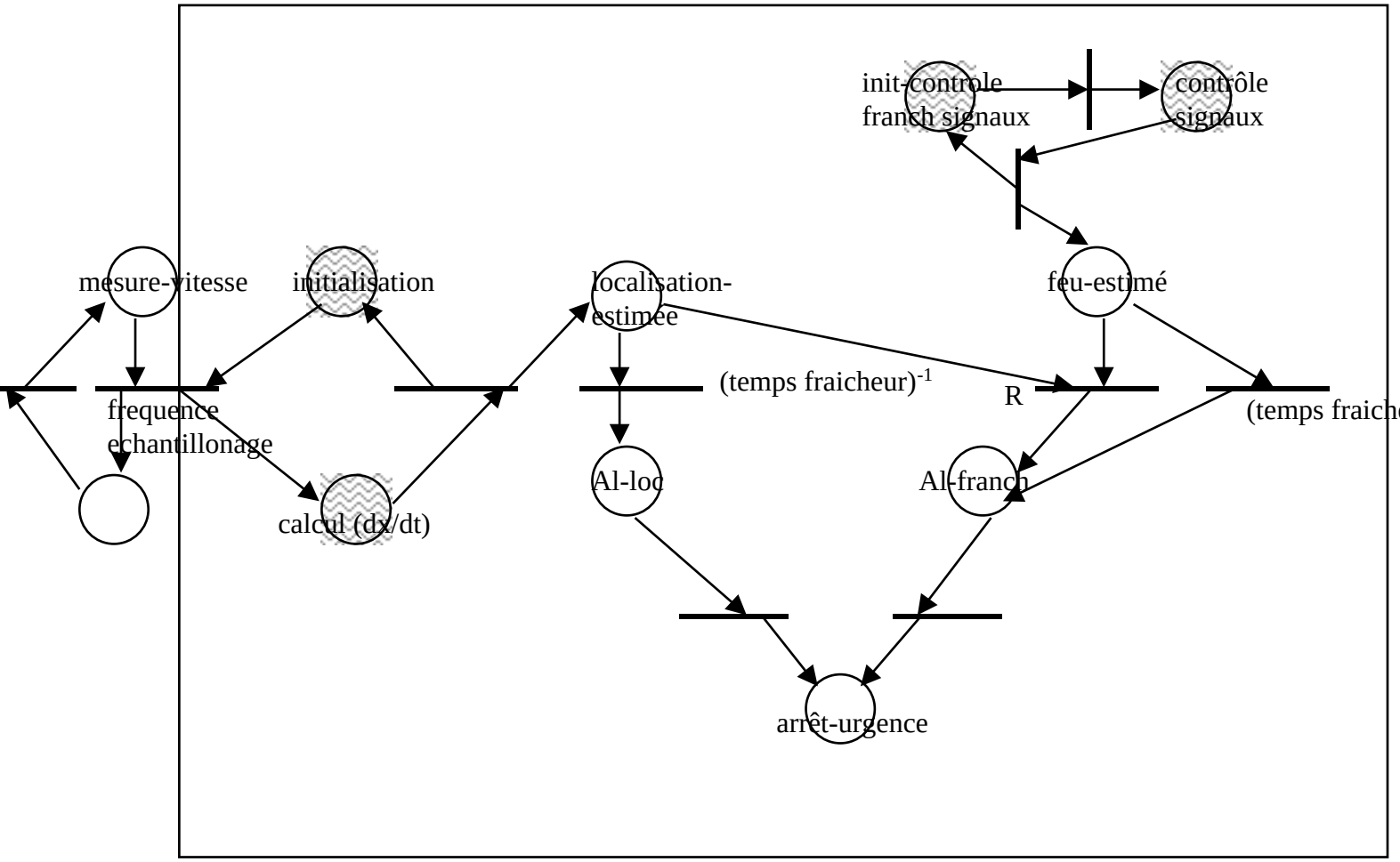
RDP, modèle de base

- Equations fondamentales
  - M=M(0) + CX où C est la matrice d'incidence (p,t) et X (t,1) un vecteur d'occurrences des tirs des transitions
  - conservation des jetons $F^t C = 0$
  - retour à l'état initial CX =0
- en utilisant ces invariants possibilité de montrer certaines propriétés sur le modèle de spécification, sans développer le graphe des marquages.

RDP, extensions

- conditions de franchissement des transitions
  - transitions en conflit potentiel
  - transitions en conflit effectif
  - franchissement simultané de transitions
  - priorités
- temporisation des transitions
  - va temps de séjour d'un jeton dans une place
  - taux de tir associé à une transition
- réseaux colorés

RDP, modélisation du comportement



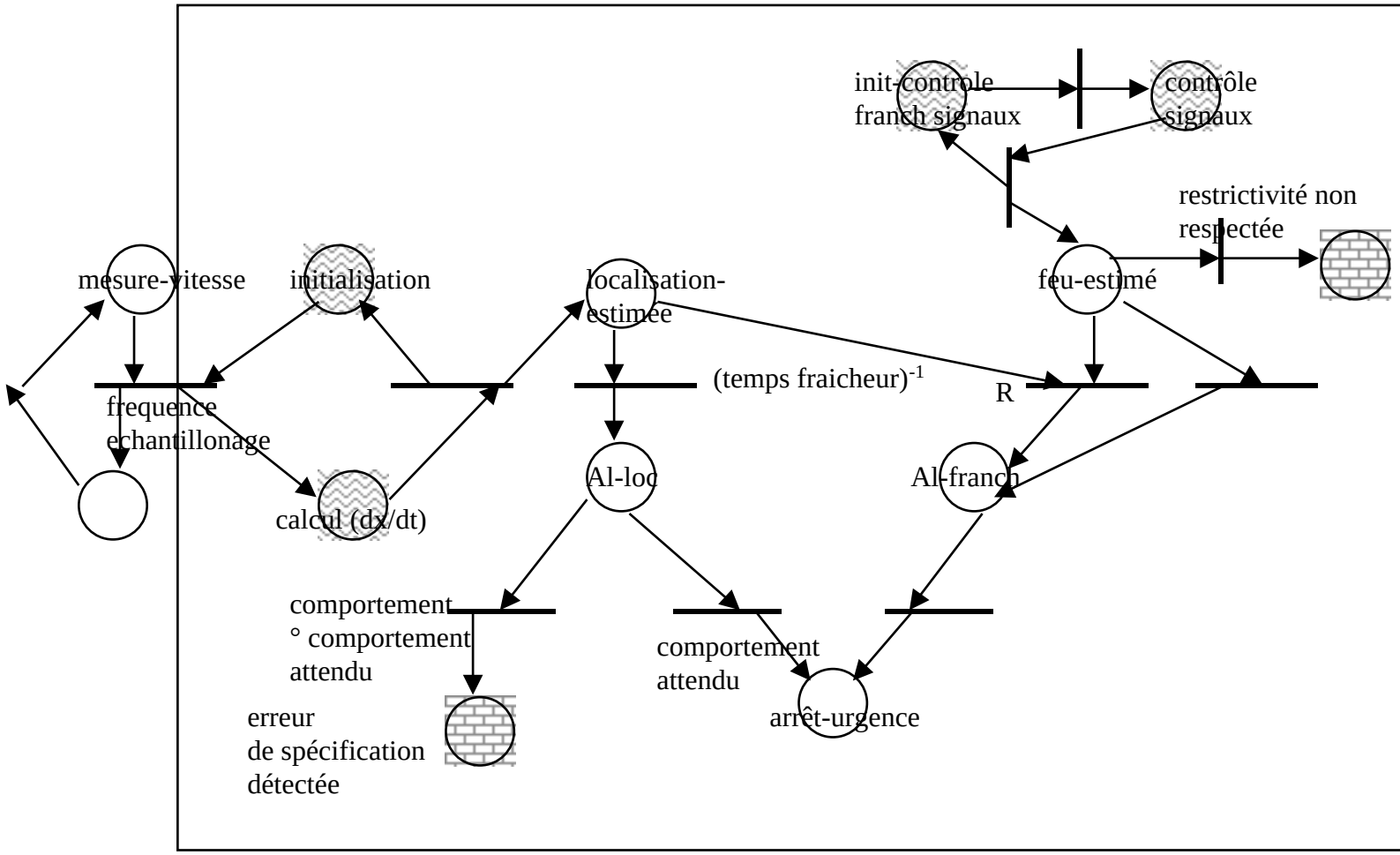
Evaluation des propriétés portant sur les états et les transitions

- Le modèle doit être validé
  - propriétés intrinsèques
  - propriétés de sûreté de fonctionnement
- Analyse directe du réseau de Petri
  - franchissabilité de transitions
  - existence de séquences particulières
  - retour à un état initial
  - propriétés de marquage
- Exploration du graphe des marquages atteints
  - caractérisation d'états cible
  - recherche des trajectoires menant à ces états cible
  - recherche d'états puits, de composantes connexes

identification des propriétés par modélisation

- propriétés globales
  - temps de calcul versus fréquence d'échantillonnage ?
  - temps entre apparition d'une situation dangereuse et l'arrêt
  - cohérence temporelle entre les variables d'états : l'estampille de "localisation estimée" et de "feu estimé" doivent être cohérentes (domaine de validité des estampilles à gérer)
- propriétés vis à vis de l'environnement
  - Sécurité : si (feu estimé = vert) alors (feu réel = vert) (restrictivité)
  - Sécurité : localisation-estimée "plus restrictive" que localisation-réelle
  - Sécurité : toute situation dangereuse provoque une alarme (puis l'arrêt)
  - Disponibilité : Prob (feu estimé = rouge) et (feu réel = vert) < seuil-f (généralisable à Prob (alarme à tort) < seuil-indispo)

exemple


init-contrôle
franch signaux
contrôle
signaux
restrictivité non
respectée
mesure-vitesse
initialisation
localisation-
estimée
feu-estimé
(temps fraicheur)-1
R
frequence
echantillonage
calcul (dx/dt)
Al-loc
Al-franch
comportement
° comportement
attendu
erreur
de spécification
détectée
comportement
attendu
arrêt-urgence

Utilisation des RDP

- Une méthode très répandue
  - domaine contrôle commande (voisinage GRAFCET)
  - de nombreux outils existent et sont basés sur les réseaux de Petri avec leurs extensions
- Toutefois, l'analyse est souvent difficile, d'où une utilisation fréquente de la simulation des spécifications modélisées par RDP
  - les séquences de transitions correspondent à des scénarios ou encore à des trajectoires du système modélisé
    - difficulté : atteindre tous les états (impact de la stratégie du simulateur pour la construction du graphe des états)
    - on peut simuler "pas à pas" des séquences cibles connues à l'avance

Utilisation des RDP

- Les RDP sont plus souvent utilisés pour modéliser
  - des architectures de systèmes
  - des protocoles de communication,
  - certains algorithmes parallèles (exclusion mutuelle , partages de ressources,2PC,...
  - plutôt que des comportements logiciels ; en effet, le nombre de variables est alors souvent très important et nécessite une formalisation par RDP colorés beaucoup plus difficiles à manipuler.
- Les extensions temporisées des RDP (RDPS par exemple, GSPN) en font un outil puissant d'analyse des architectures sûres de fonctionnement
  - référence avant vers le cours d'évaluation quantitative

Conception formelle et preuve

- Cf Cours sur langages formels
- Présentation du cas METEOR

Méthodes de vérification du code

- Analyse structurelle
- AEEL
- RCC
- Diagnostics Qualité
- Analyse statique

Analyse structurelle du code

- Approche « volumétrique » visant à maîtriser la complexité des logiciels
  - Volume
  - V(g)
  - Graphe d’appels

AEEL

- Analyse des effets des erreurs du logiciel
  - Méthode systématique
    - Pour chaque module, faire des hypothèses d'erreur
    - Étudier l'impact de chacune des hypothèses sur les sorties du module
    - Analyser la propagation sur les modules environnant le module analysé.
  - Sorties : recommandations de conception et de codage des modules (exemple vérification ou filtrage des entrées, recommandatios sur le séquencement,…) afin de garantir des propriétés énoncées en phase de spécification.

AEEL (2)

- AEEL en phase de conception
  - Constitue une analyse critique des interfaces, vis à vis de la propagation d'erreur
  - Permet de fixer des règles de structuration de la conception, et de classer les modules par niveau de « criticité »
  - Permet de donner des règles de programmation locales à un module.
  - Exemple : supposons que la vitesse mesurée d'un mobile soit élaborée à partir d'une moyenne de points de mesure. La moyenne est ensuite utilisée (uti) par un module de calcul de la vitesse et de la commande d'accélération à appliquer. On suppose qu'il existe un module de contrôle de vitesse INDEPENDANT qui sanctionne la survitesse. A quelles conditions peut on utiliser le module de moyenne pour le contrôle de vitesse ?

AEEL(3)

- Analyse en phase de conception
  - Hypothèses d'erreur : La moyenne est réalisée sur un nombre insuffisant ou erroné de points de mesure
    - Conséquence : erreur relative sur le résultat
    - Impact : si l'erreur est trop grande on peut
      - Avoir une vitesse estimée trop faible => le module vitesse va demander une accélération à tort, qui sera sanctionnée par le contrôle de vitesse.
      - SI CE MODULE A FAIT UN CALCUL D'ESTIMATION INDEPENDANT, L'impact se traduit alors en perte de disponibilité au niveau système ; SINON l'erreur ne sera pas détectée et il y a risque de collision.
      - Avoir une vitesse estimée trop forte => le module vitesse va demander une décélération à tort , ce qui donne une perte de performance au niveau système.

AEEL (4)

- AEEL en phase de conception
  - Exemple de recommandations suite à l'étude des effets de l'erreur
    - Le module ne peut être partagé par la commande et le contrôle ou bien il doit être classé en critique.
    - Pour contrer les effets de la défaillance étudiée (production d'une moyenne incorrecte), on peut prévoir de mettre un test d'acceptance (une barrière logicielle) sur l'erreur relative entre la vitesse réelle et la vitesse estimée.
      - Le test peut être fondé sur des variations acceptables (fourchette) entre la vitesse (n) et la vitesse (n+K) où n est un n° de cycle et K le nombre de cycles sur lequel on fonde l'estimation.
      - Si le résultat est hors fourchette, on sort une valeur par défaut (nil par exemple) qui provoque une alarme et la mise en arrêt du système.

AEEL(5)

Utilisation en phase de codage

- Peut servir de support à la RCC
  - Exemple de Règles à vérifier :
    - La moyenne est établie sur K mesures (n, n+K)
    - Le test d'acceptance est réalisé sur la base de ces K mesures et il n'existe pas de séquence ne passant pas par le test d'acceptance
    - Le résultat du test d'acceptance est réinitialisé à chaque cycle (n, n+K)
    - A chaque cycle de calcul entre n et n+K, les entrées servant à l'estimation sont initialisées à une valeur « restrictive ».
    - Le module n'a qu'un point de sortie, qui donne la valeur issue du test d'acceptance.

Relecture Critique de Code

- But de la RCC
  - Vérifier que le code est correct vis à vis des tâches amont :
    - DCD
    - AEEL éventuellement
    - Règles de codage
- Conformité aux règles de codage :
  - Se donner des critères de relecture critique de code (exemples)
    - respect de règles de nommage,
    - respect de définitions de types et modes d'utilisation associés,
    - respect de contraintes sur les valeurs des variables,
    - respect de règles d'écriture des traitements,
    - Respect de contraintes structurelles du code
    - Respect de règles syntaxiques

RCC (2)

- Cohérence spécifications code
  - Cohérence vis à vis des spécifications et des AEEL
    - Détection d'écarts fonctionnels
    - Prise en compte recommandations AEEL
  - Cohérence DCP/code
    - Conception générale versus graphe d'appel
    - Vérification des composants appelés et des conditions d'appel
  - Cohérence DCD/code
    - Vérification détaillée (en fait ligne à ligne)
      - Séquencement des traitements,
      - Vérification de l'initialisation des variables …
      - Vérification des contextes d'appels , vérification des paramètres d'appel
      - Vérification des valeurs des variables produites
      - …

RCC (3)

- Méthode de réalisation de la RCC
  - Entrées
    - critères de relecture
    - Documents amonts
    - Code et (si disponible) analyse statique structurelle
  - Sorties
    - Tableaux de résultats (critère de relecture, composant)

Diagnostic Qualité Logiciel

- Démarche qui recouvre les éléments précédents et qui consiste à « tracer » manuellement les objectifs assignés au logiciel sur les différents produits des phases de conception et de codage.
  - Cf Présentation « Session industriels » sur ce thème

Analyse statique et détection d’erreurs

- Objectif : détecter les erreurs résiduelles qui provoqueront des défaillances à l’exécution
  - Exemples :
    - variables non initialisées
    - Dépassement de bornes de tableaux
    - Dépassement de bornes sur des valeurs, divisions par 0, overflows,
    - Partage « illegal » de variables
    - Corruption mémoire,
    - Arithmétique de pointeurs erronée
- Constitue, en phase de test un moyen de compléter, voire de remplacer la RCC pour ces classes d’erreur.
- Cf cours analyse statique dess
- Cf présentation technique et outils sous jacents « session industriels ».

Exemple de code erroné : bornes de tableau

```
// Retourne le numero du premier thread a pouvoir etre cree
// -1 si on ne peut plus en creer
int SecuServApp::GetFreeThread (void) {
  int result;

  // recherche le numero du premier thread libre
  for (result = 0;
       (result < proxyConfig.GetThreadMax ()) && (threads[result] != NULL);
       result++);

  // si aucun thread n'est libre alors retourne -1
  if (result >= proxyConfig.GetThreadMax ()) {
    result = -1;
  }

  return result;
}
```

```
// Cree un thread
BOOL SecuServApp::CreateThread (int threadNumber) {
  BOOL result = FALSE;

  // creation de l'instance du thread
  threads[threadNumber] = new SecuServThread (userRecord, threadNumber);

  if (threads[threadNumber] != NULL) {
    // demarrage du thread
    result = threads[threadNumber]->CreateThread ();

    // attend que le thread ait fini de s'initialiser
    childStartedLock.Lock (INIT_THREAD_WAIT_TIMEOUT);
    childStartedLock.Unlock ();
  }

  return result;
}
```

Tests

- Tests
  - Stratégie de tests
  - Critères d ’arrêt des tests
  - Techniques de tests
  - Outils de test

Rôle du test

- Elimination des erreurs résiduelles
  - logiciel => erreurs de conception par opposition aux défaillances catalectiques
- Réduction des risques liés à l'utilisation du logiciel
  - orientation des tests vis à vis des événements redoutés (sécurité, indisponibilité)
- erreur -> défaut -> défaillance
  - logiciel correct : pas de défaut ("0-défaut")
  - logiciel fiable : pas de défaillance (notion d'observabilité de la défaillance, c'est à dire que le défaut traverse la frontière logiciel - environnement)
  - Le test participe à la démonstration de la correction d'un logiciel, a moins vis à vis de certaines propriétés. Il s'agit donc d'une démarch SYSTEMATIQUE

Le test dans un schéma de démonstration

- Un certain nombre d'activités complémentaires
  - Sur la branche descendante
    - Utilisation de méthodes de conception et de développement
    - Méthodes de spécification formelle et de production de code par raffinements successifs
  - Sur le code source
    - Inspection de code,
    - Relecture critique de code
    - Analyse statique
  - Sur la branche d'intégration
    - recette des modules
    - validation hôte
    - validation cible

Tests dans le cycle de vie

- Hierarchisation issue de la décomposition descendante
  - tests système correspondent au test du logiciel sur cible et dans un environnement simulé (baies de test) puis réel (essais)
  - tests de validation tests du logiciel sur hôte souvent (parfois sur cible également). on vérifie le comportement pour l'intégralité des fonctionnalités identifiées dans la spécification de besoin
  - tests d'intégration
    - intégration de modules jusqu'au niveau fonctionnel
    - stratégie d'intégration
  - tests de composants logiciels
  - tests unitaires

Tests dans le cycle de vie

- Prise en compte des itérations successives dans le cycle de développement
  - tests de non régression : vérifier que les modifications n'ont pas altéré le logiciel par rapport à sa version précédente. Il peut y avoir des tests de non régression à divers niveaux en fonction de la localisation et du niveau de la modification.

Quelques problèmes du test

- Plan de tests
  - comment déterminer les “jeux” de test pertinents ?
    - couverture des instructions
    - couverture des branches
    - couverture des CDD
    - couverture des domaines d’entrée
    - couverture des domaines de sortie
  - comment déterminer les oracles des tests ?
    - calcul du résultat attendu
    - propriétés
  - comment décider de l’arrêt des tests ?
    - objectif de couverture structurelle
    - objectif de couverture des entrées et des sorties
    - Nombre d’erreurs résiduelles estimé

Quelques problèmes du test

- Réalisation des tests
  - Ecriture de tests opérationnels
  - observation des variables et des résultats
  - Contrôle de l'effort de test
    - progression de la couverture des tests
      - instructions, branches
      - domaines des entrées et des sorties
    - Suivi de la croissance de fiabilité
    - Test aléatoire, injection de faute

analyse quantitative des bugs

Répartition par phase d'apparition

- spécifications + fonctionnalités : 25%
- structuration : 25%
  - séquencement
  - traitements
- données : 23%
- Codage : 10%
- Intégration : 10%

répartition spatiale des erreurs résiduelles

- points durs, parties oubliées et peu testées...

mesure de l'effet

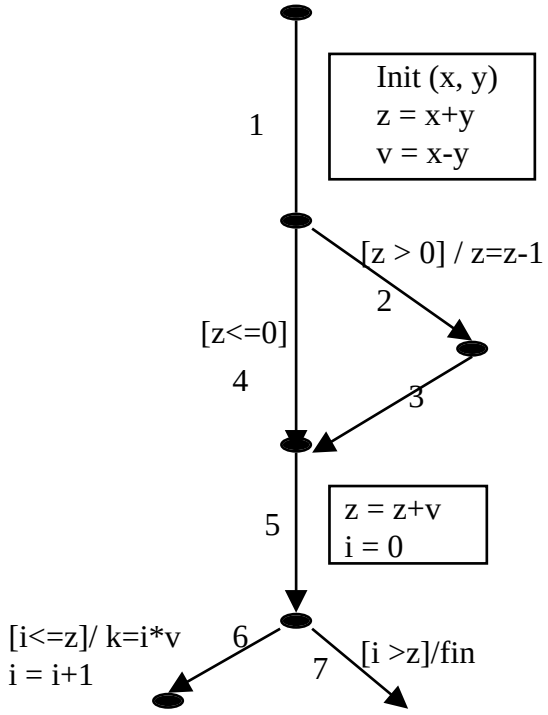
- fréquence d'apparition,
- coût de correction
- màj
- conséquences

Test structurel basé sur les chemins: flots de données et graphe de contrôle

Test structurel et graphe de contrôle

- Modèle sous jacent :
  - le programme est vu comme un graphe
    - les noeuds décrivent les points de décision
    - les arcs décrivent des branches et donc généralement des séquences d'instructions.
- objectif :
  - le test des chemins cherche à couvrir toutes les instructions et toutes les décisions
    - Couverture de chacune des instructions des blocs séquentiels
    - Parcours de tous les noeuds de décisions et de chacune des branches (CDD)

exemple



test structurel et sélection des entrées

- détermination des séquences permettant de parcourir toutes les branches
  - à chaque point de décision, déterminer les conditions à atteindre pour couvrir chacune des branches.
    - le calcul de la valeur des entrées pour positionner le test sur un chemin donné peut se faire par exécution manuelle ou symbolique.
      - exemple trouver x et y de manière à couvrir les différents chemins :
        - 1234567
        - 14567
        - 123567
        - 1457
        - 12357
      - pour le point de décision 1la branche 14 est atteinte pour (x,y) / x+y <=0 => y < -x avec x>=0. par exemple x=1 et y =-1.....

test structurel et sélection des entrées

- hypothèse sur les classes d'équivalence
  - si une variable prend ses valeurs dans un domaine borné [inf, sup], et si on fait l'hypothèse que le programme (ou un bloc donné) se comporte de manière identique sur cet intervalle, on testera les valeurs inf, sup, éventuellement un point milieu pour vérifier la conformité du comportement au résultat attendu.
- prise en compte des valeurs hors domaine de définition
  - vérifier qu'une telle valeur ne provoque pas de comportement erroné
- attention aux variables qui peuvent prendre des valeurs hors domaine suite à des opérations
  - => risque d'erreurs d'exécution telles que débordements de tableaux (un pointeur est incrémenté et dépasse les bornes du tableau), ou encore les divisions par 0 ou les débordements,....

test structurel et structures itératives ou récursives

- Trouver le nombre min de chemins pour couvrir différentes structures :
  - boucles simples
    - combien de tours ??
  - boucles concaténées
  - boucles imbriquées
    - trouver les combinaisons limites (inf d'une boucle, inf de la seconde, puis inf et sup, ...)

contrôle de la couverture de test structurel

- Détection de code mort
  - il s'agit de modules non accessibles dans une configuration opérationnelle des paramètres
  - mauvaise conception
- Contrôle du taux de couverture obtenu
  - couverture des instructions
  - couverture des chemins de écision à décision
- Contrôle des résultats et du chemin suivi.
- Complexité et taux de couverture
  - limiter la complexité du programme si on veut atteindre un taux de couverture structurel de 100% sur les chemins
    - limiter le nombre cyclomatique (v(g)<10 par exemple)

test structurel et flots de données

- Le test des chemins ne rend pas toujours compte de l'évolution des données
  - on peut orienter la selection des chemins par les flots de données, en suivant les différents modes d'utilisation d'une variable donnée.

Test d’intégration

test d’intégration

- Objectif
  - vérifier la cohérence des interfaces entre modules
  - vérifier les propriétés de comportement au niveau des appels.
- Modèle sous jacent
  - le graphe d’appel de l’application
    - architecture : modules et liens structurels entre composants
    - comportement : relations d’appel ; il s’agit d’un graphe qui explicite les passages de paramètres.
- Hypothèse
  - les modules unitaires à intégrer sont supposés testés avec un niveau suffisant de couverture pour chaque module.

test d’intégration

- Stratégie d’intégration
  - supposée définie dans le Plan de test d’intégration, réalisé en phase de conception générale
  - dépend de la structure de l’application
    - test de chaque module ou composant avec les modules qui interagissent avec lui, indépendamment
      - éventuellement utiliser des bouchons qui simulent les interactions.
    - puis intégration des modules sous forme de big bang, ou fonction à fonction.
  - l’intégration peut se faire selon une logique “fonctionnelle”
    - intégrer les modules fonction par fonction

test d’intégration

- Contrôle de l’effort de test
  - couverture des chemins d’appel
  - couverture des fonctionnalités
- difficultés spécifiques de l’intégration
  - mémorisation de variables ou “décalages” temporels entre modules
    - tester tous les cas de figure et vérifier les propriétés de cohéren globale peut être très difficile
      - exemple des protocoles de communication
      - modèles à files d’attente
- la conception de l’application doit être la plus simple possible
  - exemple des modèles FDS pour les applications critiques
  - RDV plutôt que files d’attente
  - ....

Test fonctionnel

Test fonctionnel

- Objectif
  - vérifier la conformité du comportement du logiciel à sa spécification
    - test de bout en bout, en boîte noire ou grise (car un test fonctionnel peut être l'aboutissement d'un test d'intégration!!)
- Modèle sous jacent : le modèle de spécification!
  - Flots de données de bout en bout
  - Scénarios fonctionnels et chemins formalisés par des observateurs pour les spécifications basées sur les AEF
  - Diagrammes de séquencement de messages pour les spécifications en SDL, UML,..

Test fonctionnel

- Stratégie de test
  - couverture fonctionnelle
    - couvrir les scénarios identifiés dans le modèle de spécification
    - couvrir les domaines d'entrée pertinents de l'environnement (frontière identifiée)
    - couvrir les comportements aux limites de l'environnement (et des opérateurs)
  - architecture de test
    - sur machine hôte
    - sur machine cible
    - environnement simulé ou environnement réel
  - difficulté de l'observation en environnement réparti
    - cohérence globale
    - reproductibilité des résultats

Test fonctionnel

- Contrôle de l'effort de test
  - croissance de fiabilité
  - épuisement de l'imagination des testeurs et des utilisateurs
  - épuisement du budget
- Difficultés
  - l'oracle
    - propriétés de logique temporelle
      - formules
      - observateur
  - conditions initiales d'observation d'un scénario dans l'environnement

5. Spécificités SDF logiciel

Facteurs SDF pour un logiciel (1)

- Facteurs que l'on trouve plutôt dans des normes qualité
  - Fiabilité
    - probabilité que le comportement du logiciel soit conforme à sa spécification
  - Maintenabilité
    - aptitude du logiciel à être corrigé
  - Robustesse
    - aptitude du logiciel à être utilisé dans des conditions d'utilisation dégradées
  - Tolérance aux pannes
    - il s'agit d'une contrainte externe qui fixe le nombre de défaillances observables que doit tolérer le logiciel
  - Performance
    - Respect des délais et temps de réponse,
    - critères de consommation des ressources

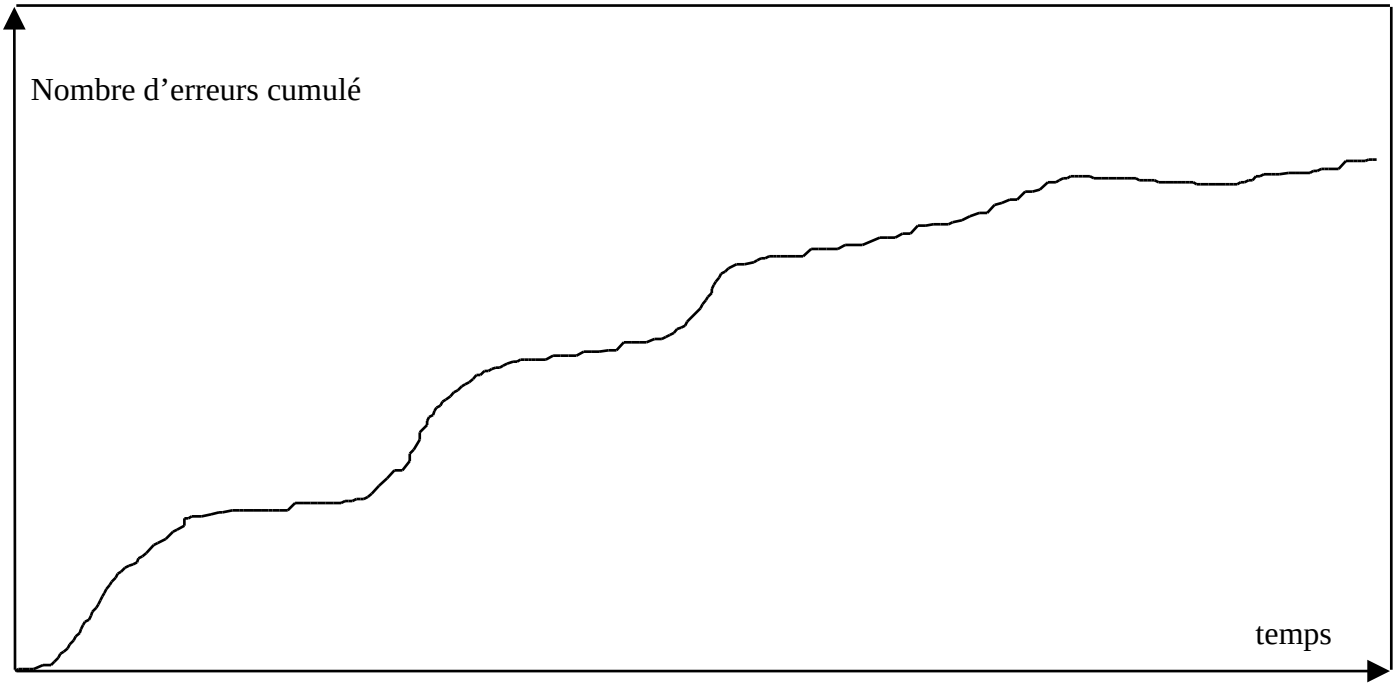
Facteurs SDF pour un logiciel (2)

- Eléments issus des normes orientées safety ou systèmes sûrs
  - Niveau d'intégrité du logiciel (Software Integrity Level), qui mesure
    - implicitement un niveau de probabilité de défaillance dangereuse tolérable à garantir (seuil de tolérance)
    - explicitement le SIL correspond à un ensemble de techniques qui doivent être utilisées pour justifier du respect de ce niveau de probabilité de défaillance.
  - Probabilité de non propagation d'erreur
    - ce critère a un impact sur la conception générale du logiciel (approche par niveaux et etanchéité entre niveaux)

Approche “historique” d’évaluation SDF du logiciel

- Années 80
  - Modèles de croissance de fiabilité
    - Principe : pendant la phase de test, on mesure le temps qui s’écoule entre deux apparitions d’erreur. En notant U la va égale au temps écoulé entre deux occurrences d’erreurs successives, on essaie d’approcher le processus de défaillance par un processus semi markovien et d’estimer le nombre d’erreurs résiduelles dans le logiciel, ainsi que la “date” estimée de la prochaine panne.
    - De nombreux modèles ont été proposés (Musa, Goel Okumoto, Osaki,Littlewood, Kitchenham, Kanoun, Basili, Troy...)
      - difficulté de la validation (collecte des données et validation du modèle)
      - prise en compte de l’effet “stress” des phases de test ?

Modèles de croissance du logiciel



Modèles utilisés plutôt pour planifier les efforts de tests et décider de l’arrêt des tests lors des corrections ou évolutions du logiciel dans sa phase d’exploitation.

Les approches basées sur la structure du logiciel

- Recherche sur les liens entre “principes de conception du logiciel” et fiabilité du produit en exploitation
  - de nombreuses expériences (collecte de données) pour faire le lien entre des métriques telles que
    - complexité, niveau du langage, notamment
    - fiabilité observée du logiciel
  - Les résultats de cette approche sont en général décevants, mais ils permettent de faire porter l’effort de test sur les modules compte tenu de métriques telles que la complexité par exemple.
  - Pratiquement il en résulte plutôt des règles simples pour le développement de logiciels sûrs telles que
    - limiter le v(g) d’un module à K
    - limiter le parallélisme
    - limiter le nombre de variables et les types utilisés...

Approches basées sur l'évaluation du processus de développement

- Le principe consiste à montrer que le niveau de fiabilité obtenu résulte d'un niveau de rigueur du processus de développement
  - Evaluation de l' Efficacité des principes de conception et des méthodes employées pour mettre en oeuvre ces principes
  - Evaluation de la Conformité, c'est à dire de l'implantation des principes dans le produit logiciel.
- Il s'agit d'une formalisation d'une pratique très utilisée dans l'aéronautique et le spatial (DO178B notamment), et plus généralement pour le développement de systèmes critiques (AED :DO178B, JAR, ferroviaire : EN50128, nucléaire: IEC 880)
- de nombreux travaux en cours
  - certification des systèmes de sécurité- confidentialité(Critères Communs)
  - projets européens
  - expérimentations industrielles

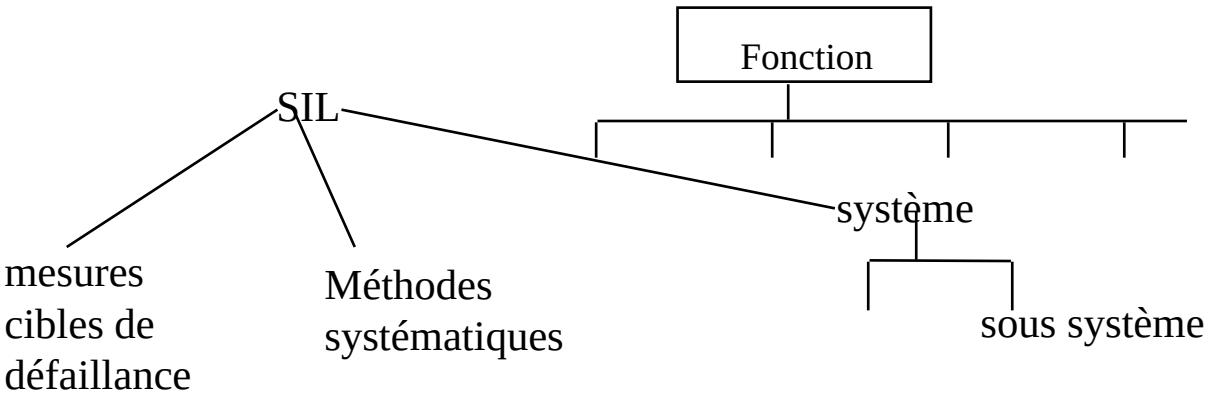
Synthèse

- pour le logiciel, on doit se ramener à une pure fiabilité en termes de probabilité de conformité à la spécification
- association d'un niveau de processus logiciel à un niveau de démonstration (rigueur)
- association entre les niveaux de démonstration et la réduction des risques de non conformité
- l'objectif de fiabilité assignés à chaque composant logiciel résulte d'une analyse architecturale

Niveaux d ’intégrité

- Normes IEC 61508, EN 50128, IEC 880 font le lien entre cible de défaillance et niveau d ’intégrité

Apport des SIL



Association entre processus de développement et rigueur de démonstration (1)

- L1 : la spécification du logiciel considéré et les tests fonctionnels tendent à montrer la satisfaction des objectifs.

- L2 : L1 + description informelle de la conception détaillée des logiciels. L'efficacité du test fonctionnel doit être évaluée. Principes et procédures applicables de mise en exploitation des logiciels.

- L3 : L2 + traçabilité établie entre tous les niveaux. Test unitaires et d'intégration avec contrôles de non régression sur tous les tests. Gestion en configuration des versions du logiciel et de son environnement

Association entre processus de développement et rigueur de démonstration (2)

- L4 : L3 + la spécification du logiciel est dérivée de la spécification des objectifs RAMS, la description de la solution (architecture logicielle et conception détaillée) est semi formelle. L'efficacité des tests à tous les niveaux est évaluée. Le risque lié à la défaillance des outils de production du code est évalué.

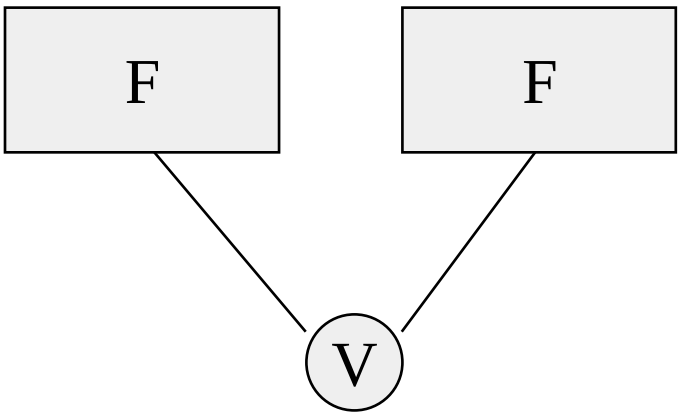
- L5 : L4 + le processus de validation des objectifs RAMS est réalisé par une équipe indépendante qui applique un processus formalisé. Les chaînes de production du code et des constantes du processus font l'objet d'un contrôle exhaustif ou d'une redondance de production.

- L6 : L5 + le processus de dérivation du code à partir des objectifs repose sur une technique formelle et des outils validés.

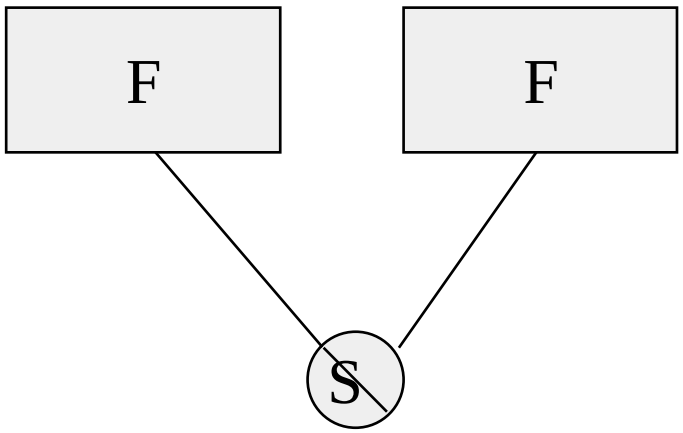
Association entre objectifs de fiabilité et rigueur de démonstration

Niveau d'intégrité logiciel	$P < 10^{-2}$	$P < 10^{-3}$	$P < 10^{-5}$	$P < 10^{-7}$	$P < 10^{-9}$
0	0	0	1	3	3-4
1	1	2	3	4	5
2	2	3	4	5	
3	3	4	5		
4	4	5			

Projection des résultats AS sur l ’architecture numérique : analyse architecturale (1)



Sémantique de panne
Fail Silent



Sémantique de disponibili

Projection des résultats AS sur l ’architecture numérique : analyse architecturale (2)

- Choix de conception :
  - décomposition matériel/logiciel en fonction :
    - du niveau que l ’on peut attendre des composants (réutilisation, COTS, spécifiques)
    - de la complexité de réalisation
    - du coût

Types de composants logiciels

omposants de service (application, noyau)

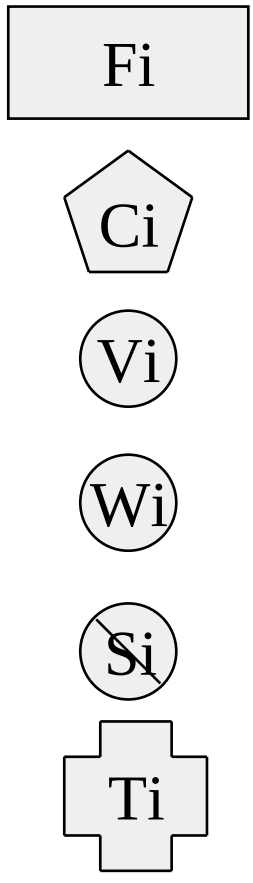
OTS

oteurs

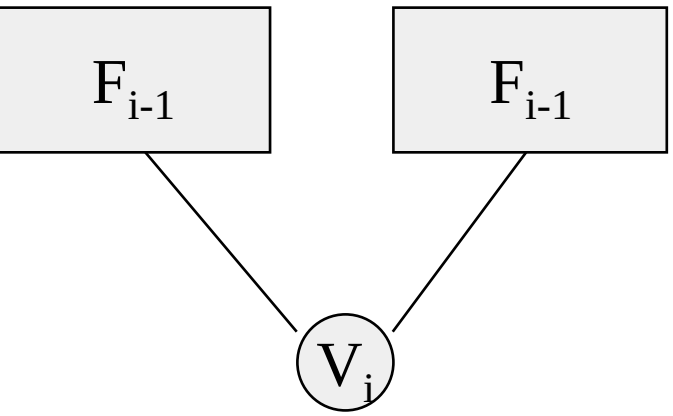
rappers

ommutateurs

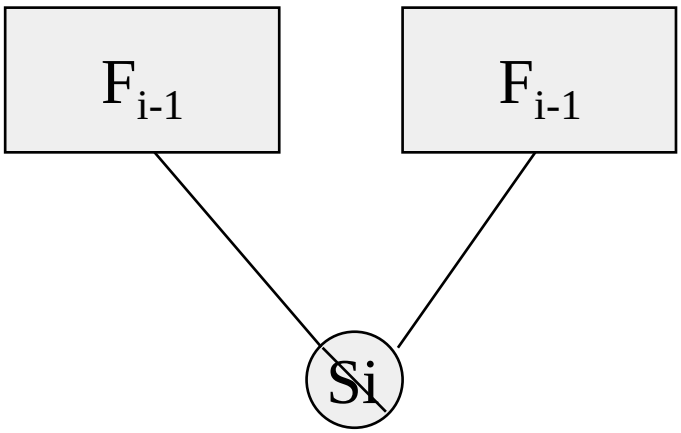
omposants de tests en ligne



Architecture logicielle à niveaux d ’intégrité



Sémantique de panne fail silent de niveau i



Disponibilité de niveau i

Formalisation des exigences du logiciel /niveau de processus (exemple-1)

Spécification de l'environnement

Exigences du logiciel

EX L5-1 :

EX L5-2 :

EX L5-3 :

EX L5-4 :

EX L5-5 :

Produire un plan

Formaliser les

Les objectifs

Les interfaces

ous ensemble doivent être décrits en termes de fon

Formalisation des exigences du logiciel /niveau de processus (exemple-2)

Démonstration de la couverture des erreurs par le processus de développement

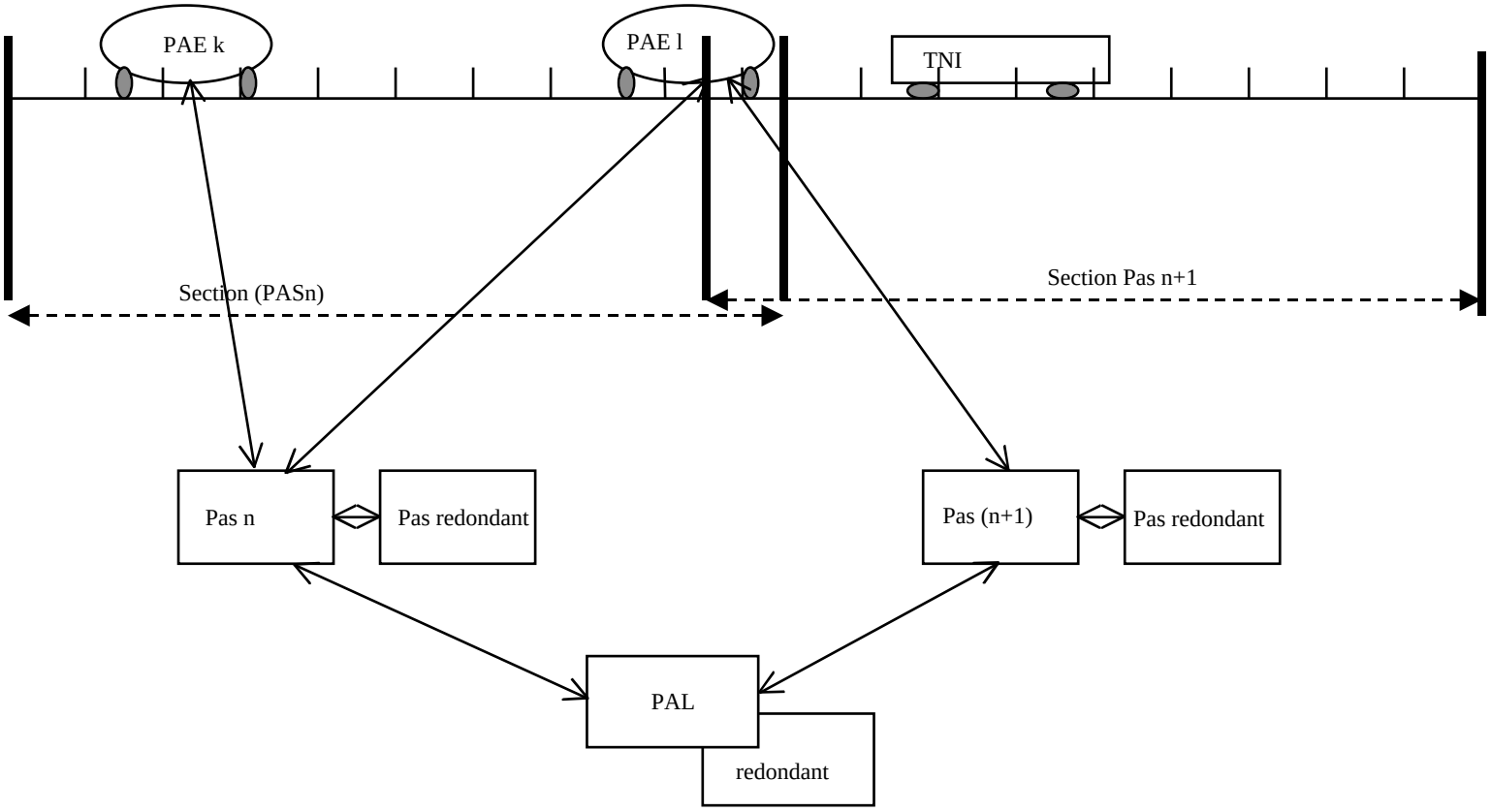
- Faire le lien entre les méthodes systématiques employées pour éviter l ’occurrence de fautes dans le processus logiciel et la couverture des erreurs qui peuvent affecter le produit logiciel
  - erreur d ’interface couverture par programmation défensive et orientation des entrées…
  - erreurs de spécification (non correction ou non complétude) couvert par formalisation et évaluation de propriétés
  - erreur de raffinement : couvert par
    - preuve de raffinement (exemple du langage B)
    - traçabilité et démonstration manuelle de la conservation des propriétés, RCC, AEEL, ...
    - génération de code à partir du modèle de specification
    - faute dans le passage au code cible
    - traduction « démontrée » ou utilisation de générateurs certifiés
  - ….

6. Etudes de cas

Orienté Safety : le cas METEOR

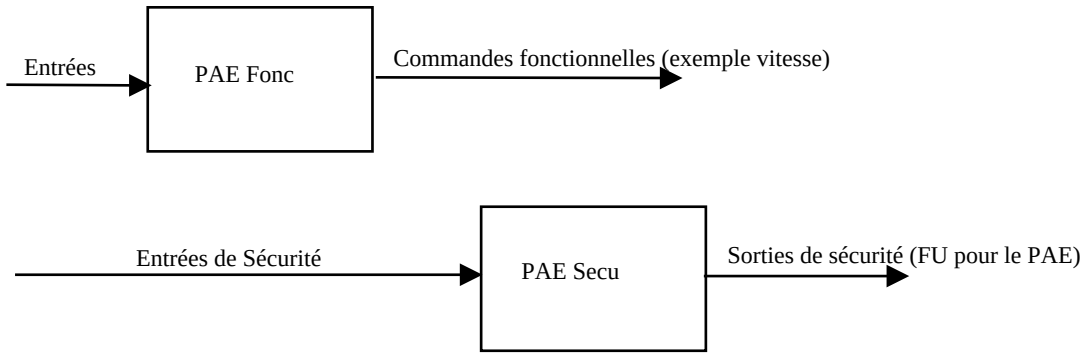
- Caractéristiques du système
  - Un PCC (Poste de Commande Centralisé) assure la régulation (niveau ligne), la surveillance et les fonctions de communiation
  - Des Pilotes Automatiques, qui réalisent le contrôle commande des mouvements des trains et des échanges voyageurs.
    - Pilote Automatique Ligne
    - Pilote Automatique Sol (un par section géographique)
    - Pilote Automatique Embarqué (un par Train Equipé)
  - Le système doit supporter des Trains Equipés (TE) et des trains Non Informatisés (TNI) => superposition de principes « informatiques » et de principes de sécurité intrinsèque, issus du matériel.

Architecture du PA METEOR



Caractéristiques des sous systèmes du PA

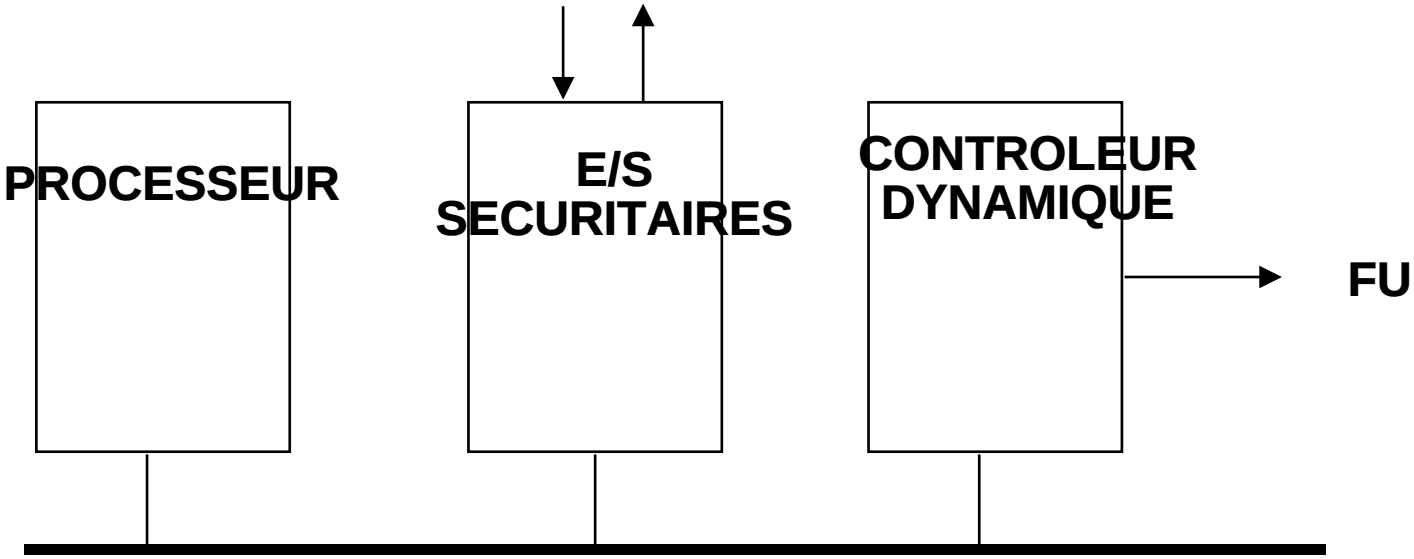
- Chaque sous système (Sol Bord) a une partie dite « fonctionnelle » et une partie dite « sécuritaire » qui doivent être INDEPENDANTES



Caractéristiques des PA de sécurité

- Existence d ’un état de sécurité passif
  - recours à l ’arrêt du train toujours possible
  - Ceci permet de se doter d ’un système de détection des erreurs et de se ramener à une panne franche (interruption de l ’exécution sur les calculateurs) et arrêt des équipements matériels (arrêt effectif du train, coupure effective de la tension)
  - Implantation du mécanisme de détection et de retour à l ’état de sécurité passif : Processeur Codé
- Pour le logiciel de sécurité: il doit être « parfait » ou encore « 0 défaut »
  - programme cyclique séquentiel
  - méthode formelle de conception et de codage

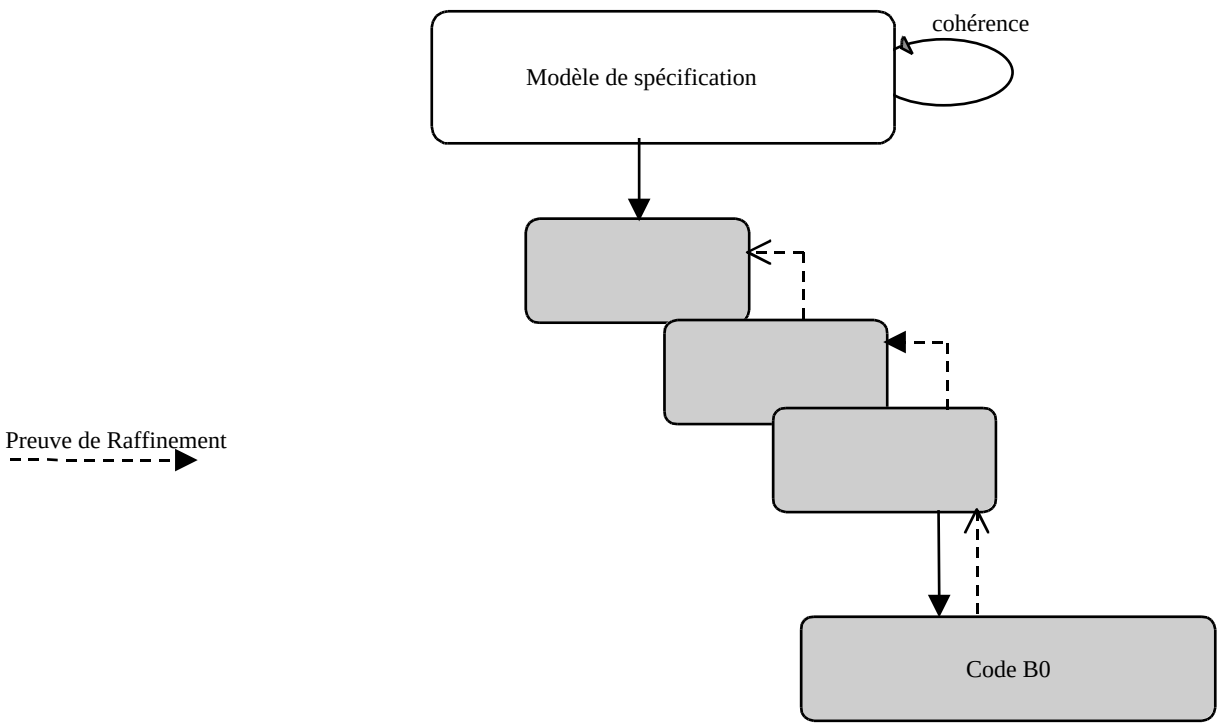
Principes d ’architecture matérielle



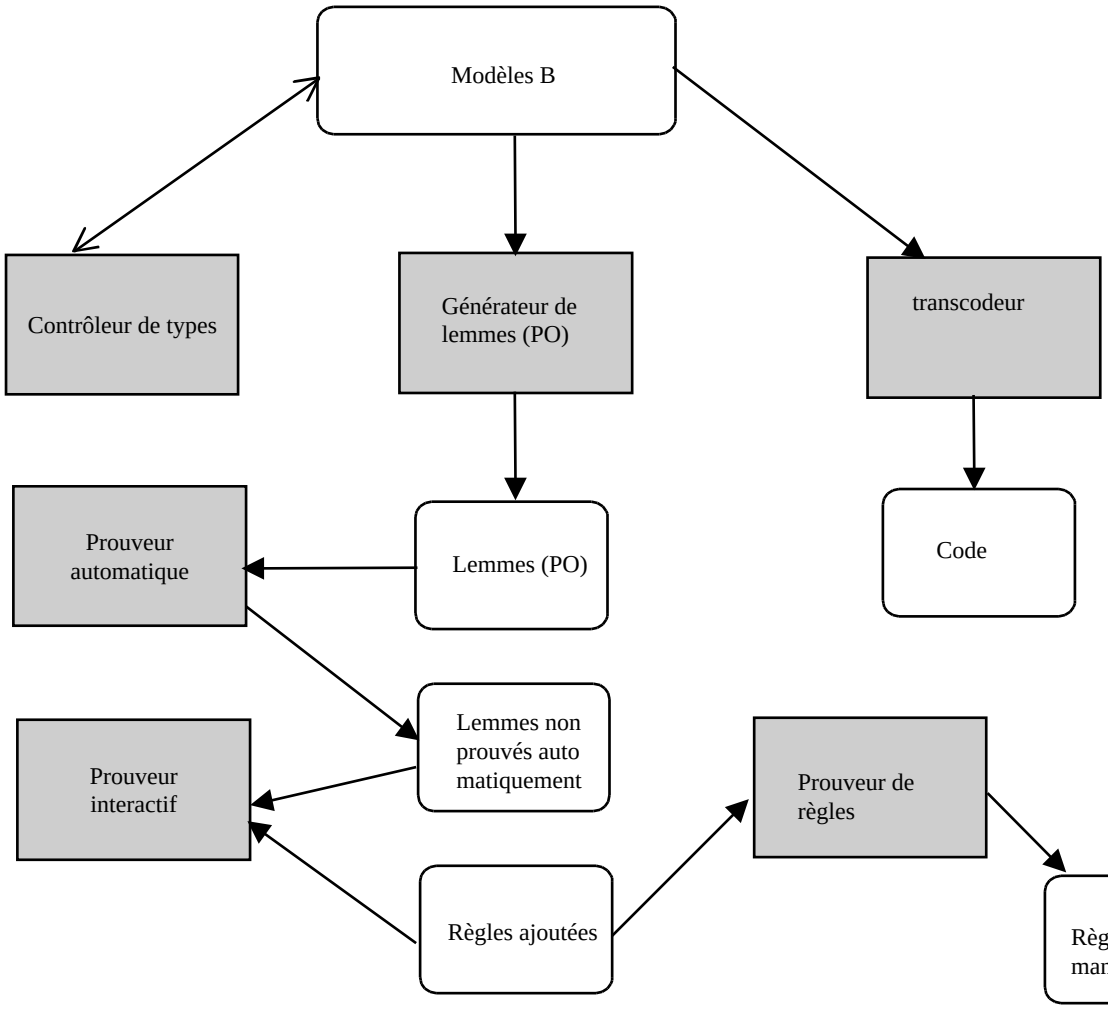
Maîtrise des risques en conception

- Conception logiciel
  - Utilisation d ’une méthode formelle de développement, supportée par un outil
    - Langage B
    - Preuve de raffinement
    - Obtention d ’un source B0, 0 défaut
  - Traduction B0 -> ADA, supposée parfaite
  - Compilation et Exécution sur Processeur Codé
    - Risque résiduel à 10^{-n} où n dépend du codage (en général 14 pour 48 bits de redondance, cf poly architecture)

Principe de développement B



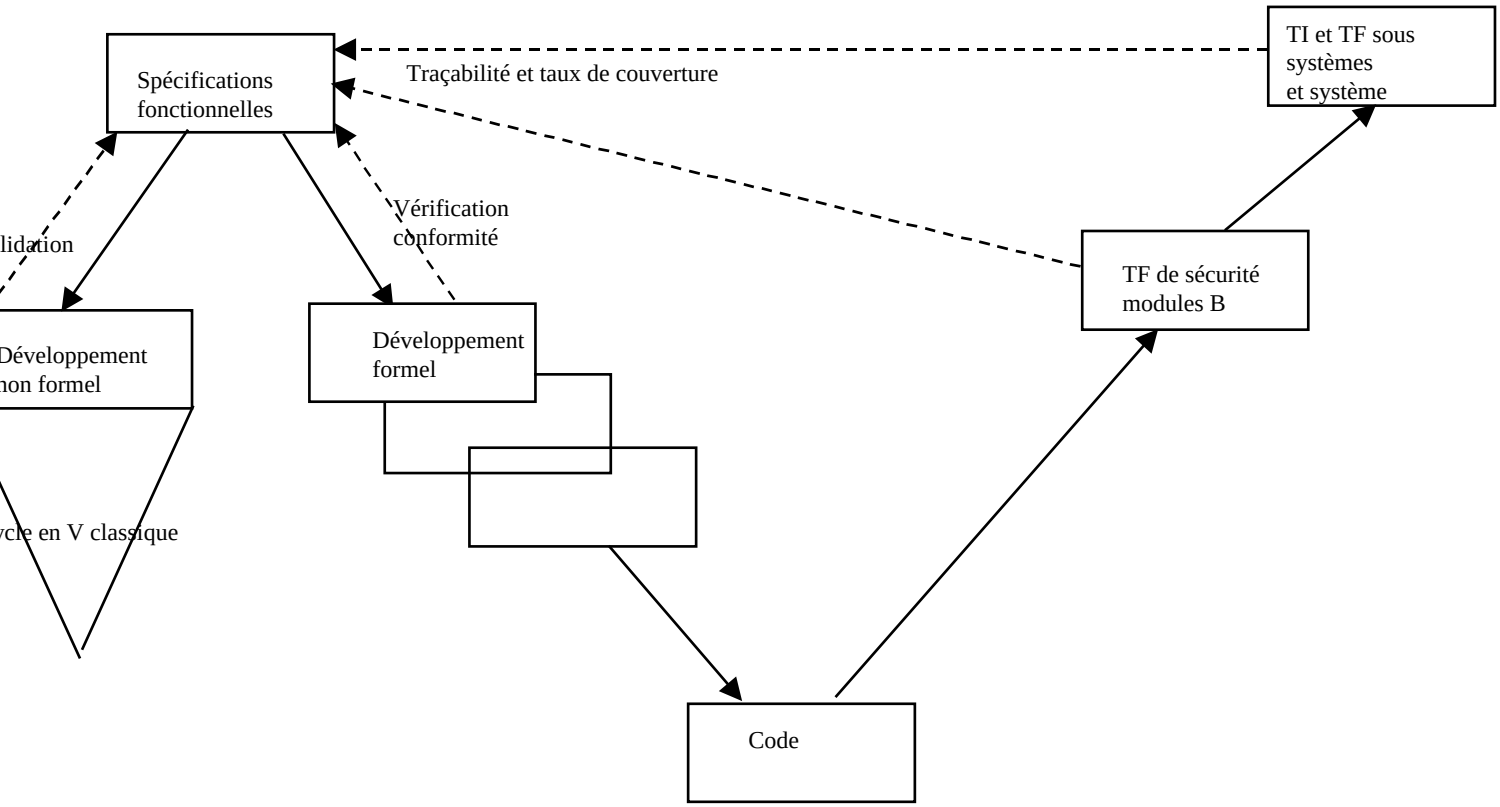
Outillage Développement B



Contrôle du développement B

- Conformité du modèle initial à la spécification : manuel
- Cohérence interne du modèle initial (automatique)
- Preuve de raffinement à chaque étape
  - Taux de preuve automatique des PO
  - Preuve des règles ajoutées nécessaires à la preuve des lemmes résiduels
    - règles context free
    - risque pour les règles context sensitive (qu ’il faut gérer en configuration…)
- Analyses de code (AEEL) sur code non prouvé et surtout sur interface entre code non prouvé et code prouvé (manuel)
- Transcodage B0 -> ADA parfait
- Tests fonctionnels
  - suppression TU TI sur code prouvé

Résumé du cycle de développement logiciel



Les actions de validation du client : cycle W

- Le client valide systématiquement et de manière indépendante toute la production de l ’industriel
  - Validation des productions
    - Spécifications fonctionnelles
    - Spécifications logicielles
    - Cahiers de tests
  - Validation des dossiers SDF fournis par l ’industriel
- En fait sur METEOR, la RATP a modélisé toutes les spécifications, refait ses propres analyses de risques et produit ses cahiers de tests indépendamment
- Les tests ont ensuite été réalisés indépendamment

Mise en œuvre double Validation sur les fonctions de securité du PA : étape spécification logicielle

- Pour toute fonction de sécurité (exemple anticollision, Contrôle Vitesse, Echange voyageurs,…)
  - modéliser sous forme AEFC (outillage ASA+) la spécification textuelle fournie par l ’industriel
  - enoncer des propriétés de sécurité
    - prédicats sur variables d ’états, invariants d ’états
    - formules de logique temporelle
  - valider scénarios
    - simulation interactive pour mise au point
    - simulation intensive pour recherche états puits
  - vérifier propriétés
    - évaluation de propriétés à la volée
    - simulation exhaustive + évaluation propriétés
  - générer les tests à partir du modèle AEFC

Mise en œuvre double validation : exemple de la validation des constantes

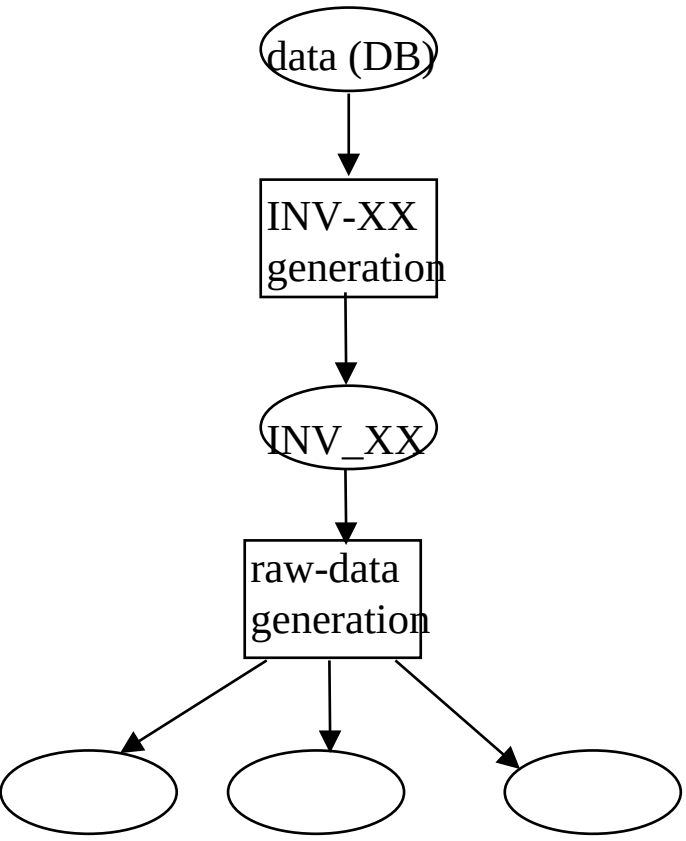
- La démarche de validation du logiciel doit couvrir
  - le logiciel « générique »
  - les données de ce logiciel
- Les données sont en réalité du code caché
  - réécriture manuelle de principes cinématiques
  - fourniture d ’une image de la topologie de la voie et de tous els points qui seront utilisés par les calculs embarqués
  - …
- Il s ’agit donc également de logiciel CRITIQUE, qui ne peut faire l ’objet de la même démarche de preuve
  - réels
  - complexité

Constantes

- 3 levels
  - DB : application oriented (points positions, distances between physical objects, ...)
  - INV_XX : system design oriented (type specification)

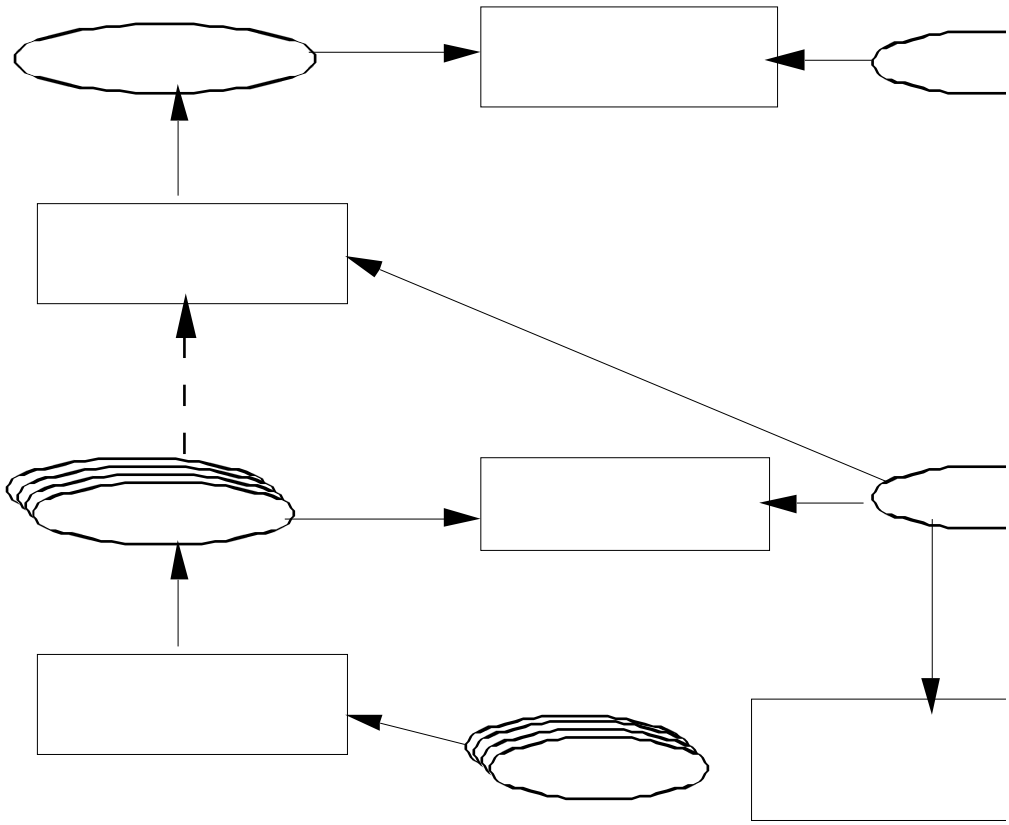
- Raw data : computer architecture dependant
    - one file per computer
    - ADA integers, compliant with ADA generated (code)

Génération des constantes (industriel)



- Textual specification (3 levels)
  - DB, INV_XX, Raw data
- Specification of constraints and traceability throughout the three levels (informal derivation process)
- Code Diversity (OGIM and OVIM)

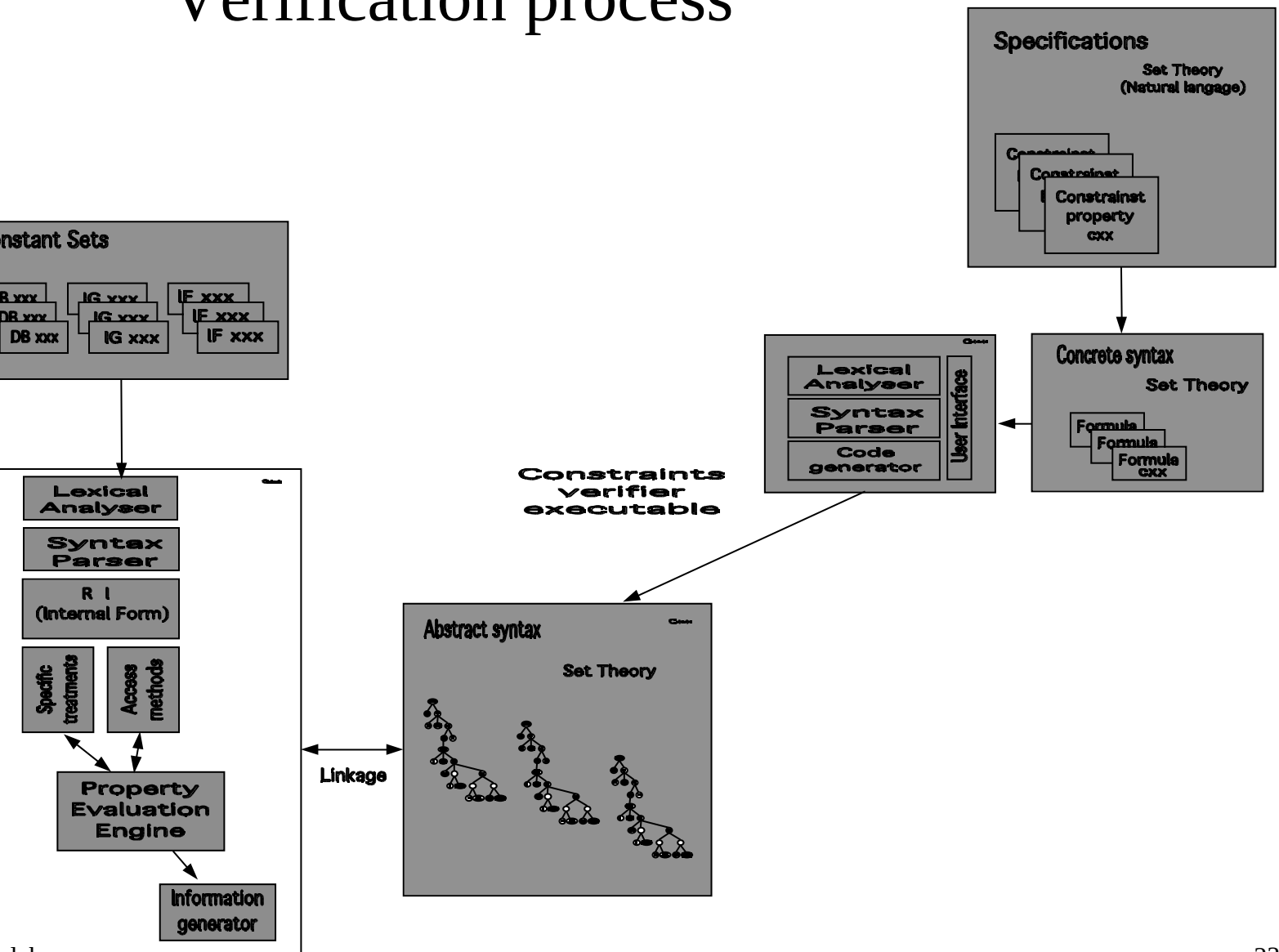
Validation des constantes (client)



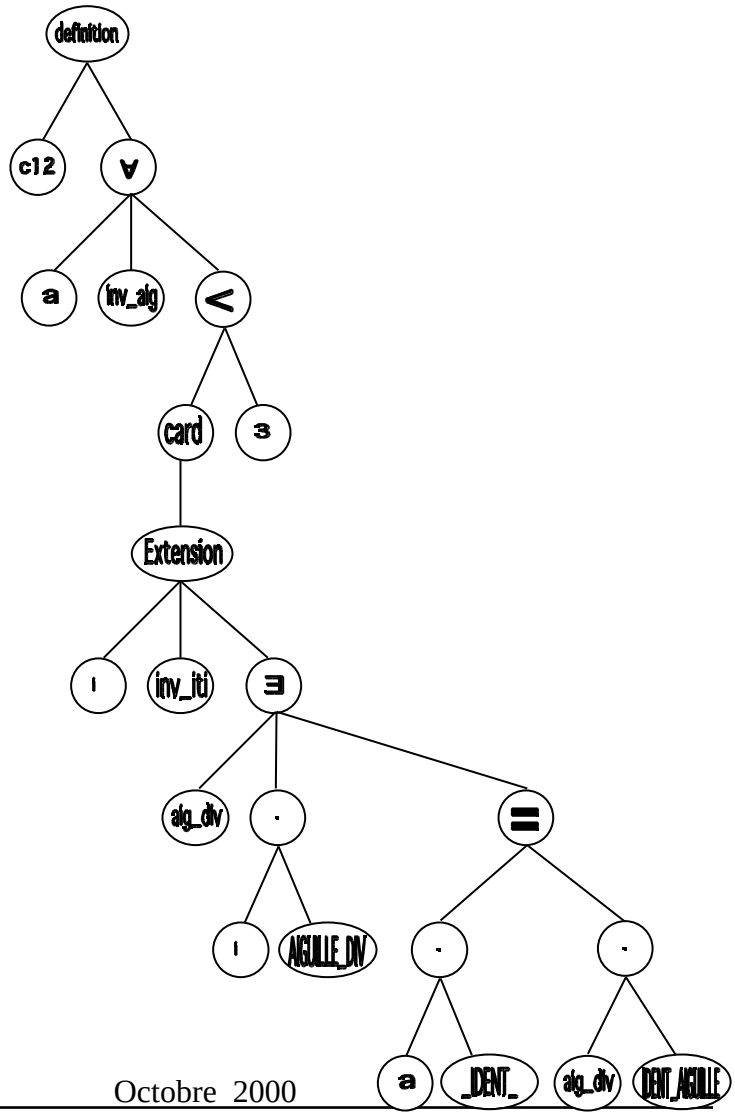
Exemple de propriété sur les constantes

- INV_POINT
  - ATT_1 : point_id
  - ATT_2 : list of Ground ATO the point belongs to
  - ATT_3 : Ground ATO in charge of transmitting the point position to the Embedded ATO
  - ATT_4 : chaining data (track supports)
- Constraints
  - existence and correctness of ATT_1
  - P INV_POINT, Card {ATT_3} =1 and ((x INV_PAS, x.ATT_1 P.ATT_3) => (x.ATT1 P.ATT_2))

Verification process



Mapping onto invariants



Quelques points d ’organisation des développements de systèmes critiques (1)

- La reconnaissance préalable de principes d ’architecture numérique et de méthodes de développement par la communauté
  - Les normes décrivent les méthodes par objectif de sécurité (CEI 61 508)
  - Chaque métier fait ses choix, selon également des critères géographiques
    - dans notre étude de cas franco-française, choix du processeur codé et de B
    - principe de logiciel séquentiel cyclique
    - mais il existe d ’autres choix à l ’étranger
      - architectures redondantes TMR ou 2X2
      - Spécifications Z ou VDM et analyse de code ...

Quelques éléments d ’organisation pour le développement de systèmes critiques (2)

- La mise en place d ’une organisation des actions de validation
  - chez l ’industriel : Validation par une équipe indépendante
    - Analyse de risque du processus lui même => actions à mettre en place
    - Analyse critique de toutes les productions de l ’équipe de développement
    - Tests de sécurité indépendants
  - chez le « client »
    - Double chaîne (exemple RATP sur METEOR)
    - Enoncé d ’exigences sur le processus et les productions en terme de niveau de démonstration de l ’atteinte des objectifs
    - Evaluation indépendante des éléments de preuve apportés par l ’industriel

4. Paradigmes de la tolérance aux pannes

A faire

5. Analyse et évaluation quantitative d ’architectures numériques

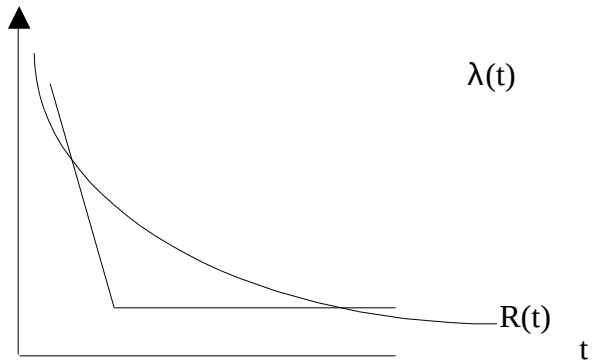
Mesure de la sûreté de fonctionnement

- Composantes de la sûreté de fonctionnement
  - Reliability R(t); fonction du temps qui estime l'aptitude d'un dispositif à accomplir sa mission (à délivrer son service) pour un intervalle de temps donné
    - R(t) = Prob (Z(τ) =1 , quelque soit τ appartenant à l'intervalle [0,t]) où Z est une variable de {0,1} qui exprime que le service délivré par le système système est correct (conforme à la spécification) ou défaillant
    - Il s'agit d'une fonction décroissante du temps
    - MTFF Temps moyen jusqu'à la première défaillance

$$MTTF = \int_0^\infty R(t)\,dt = \int_0^\infty -\frac{dR}{dt}\,t\,dt$$

fiabilité

- Lois utilisées pour les systèmes électroniques
  - matériel
    - loi exponentielle
    - loi de Weibull
  - logiciel
    - de nombreux modèles semi markoviens proposés dans les années 80.
    - en fait, les modèles décrivent la croissance de fiabilité en phase de test.
    - décider de la phase d'arrêt des tests

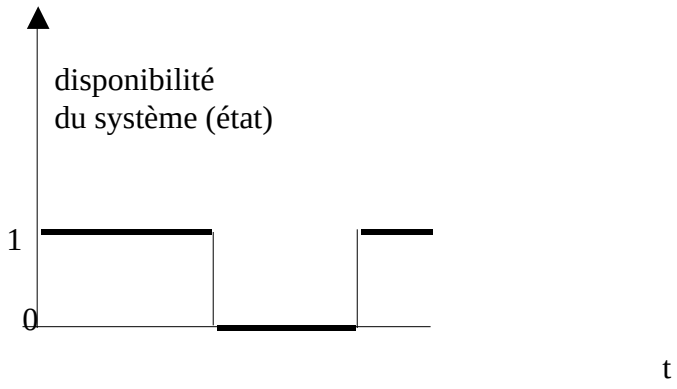


maintenabilité

- la maintenance est liée à la phase d'exploitation du système
- systèmes
  - réparables
  - non réparables
  - dégradables
- évaluation prévisionnelle
  - prise en compte des politiques de maintenace,
  - du dimensionnement des stocks et du taux de réparation
  - reconfiguration pour les structures redondantes
- modèles probabilistes
  - exponentiel
  - taux constant

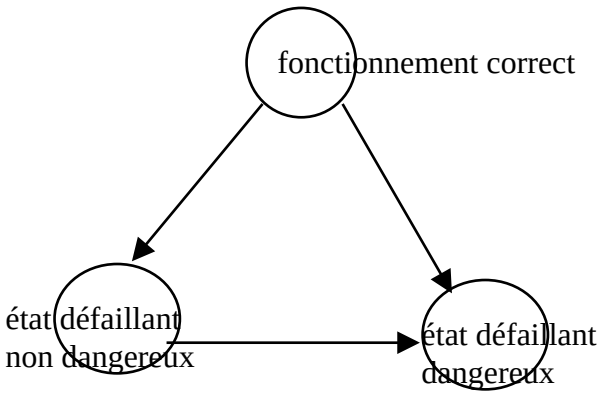
disponibilité

- probabilité pour que le produit fonctionne à l'instant t sachant qu'il est en état de fonctionnement correct à t=0
- disponibilité instantanée
  - A(t) est donnée par la probabilité de l'état 1 à l'instant t
- disponibilité asymptotique (montrer que régime permanent existe => conditions d'ergodicité)
  - A*



sécurité (innocuité)

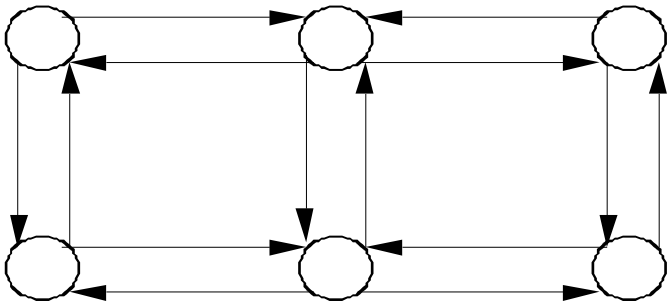
- distinguer les états défaillants mais non dangereux et les états dans lesquels un événement indésirable peut se produire.
- S(t) probabilité que le produit n'ait pas de défaillance catastrophique entre 0 et t



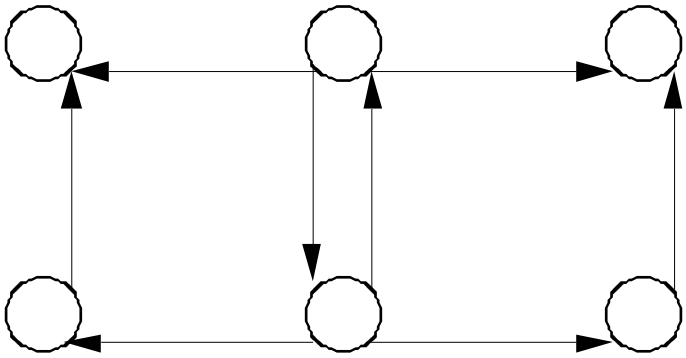
autres grandeurs de la sûreté de fonctionnement

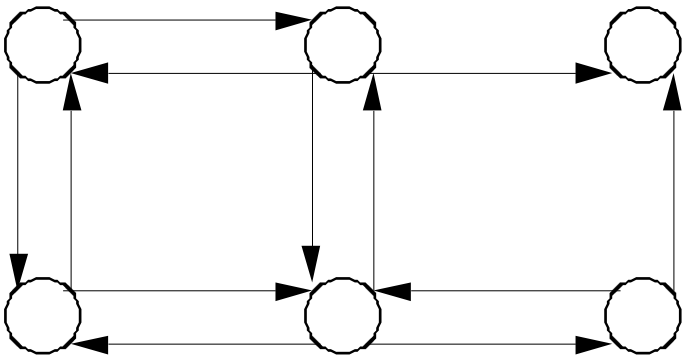
- testabilité
  - facilité à déterminer les séquences de test
  - longueur des séquences de test, nombre de vecteurs d'entrée et de sortie à observer
  - couverture de test
  - efficacité de test (nombre de fautes détectées vs nombre total estimé de fautes dans le produit)
- Sécurité (security)
  - confidentialité
  - intégrité

graphes de sûreté de fonctionnement

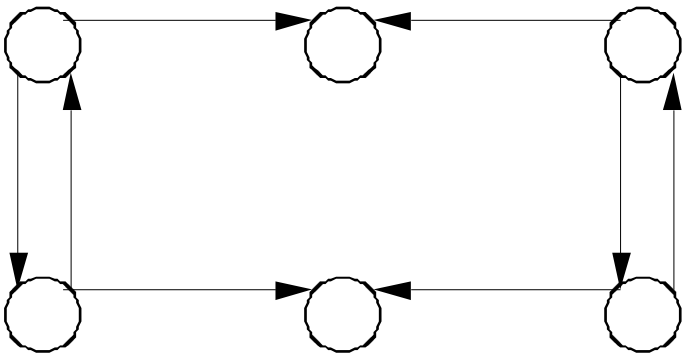


graphes de sûreté de fonctionnement





graphes de sûreté de fonctionnement



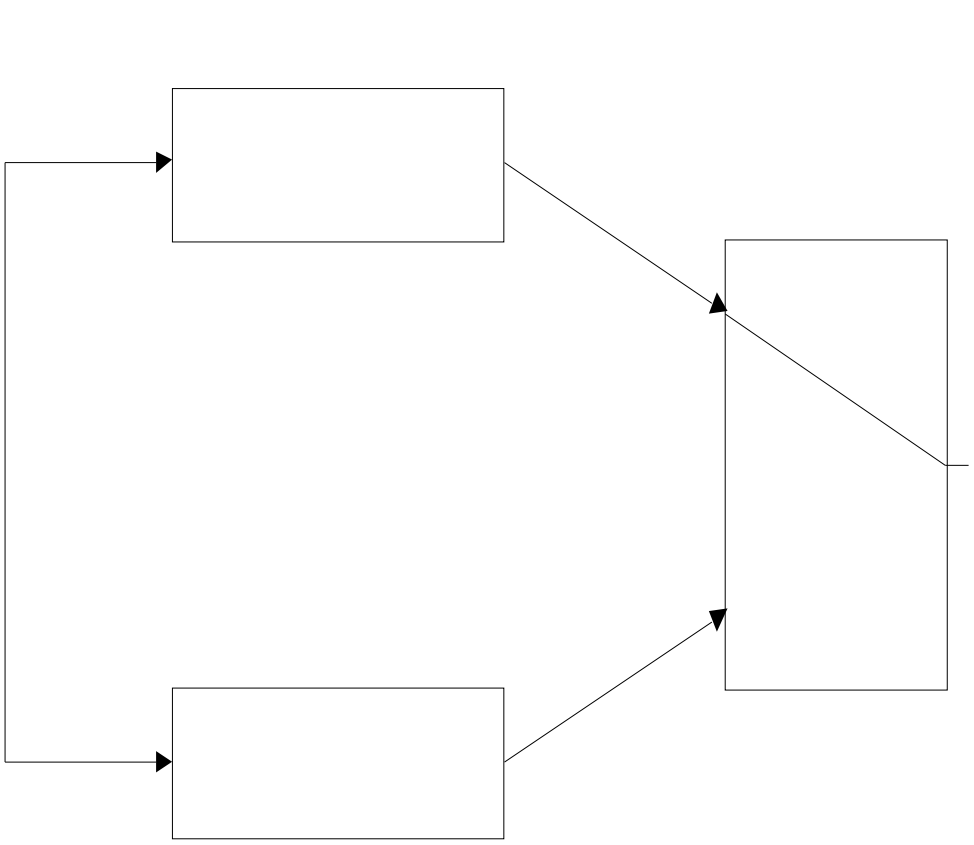
Etapes de l’analyse et de l’évaluation sdf

- spécification des objectifs de sûreté de fonctionnement
- analyse qualitative
  - AMDEC
  - arbre de causes (MAC)
  - RDP
  - ...
- modèles quantitatifs
  - diagrammes de fiabilité
  - modèles markoviens et semi markoviens
- modélisation projection des objectifs sur le modèle
- résolution, exploitation du modèle

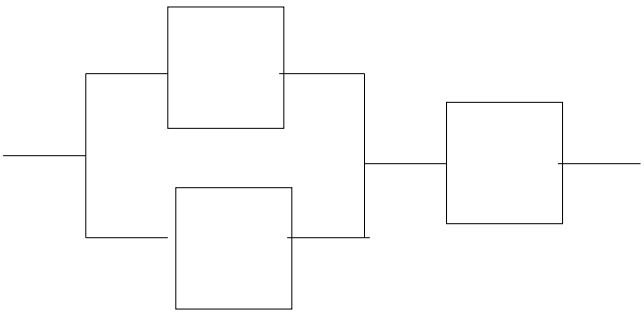
Analyse markovienne

- Présentation « tableau » des principaux résultats
  - Processus markovien
  - Régime permanent
  - Régime transitoire
  - Application aux calculs des composantes sdf

Cas d’école : RSA



Diagrammes de fiabilité et architectures serie parallèle



Modèles markoviens (1)

Modèles markoviens (2)

- matrice des taux
- décomposition de la matrice des taux suivant classification des états
- MTFF => calcul des temps de séjour en états transitoires
- Disponibilité asymptotique => calcul de régime permanent

Exemples d’évaluation d’architectures

- Exemple TMR

9. Spécification et prise en compte des exigences de sûreté de fonctionnement

Traçabilité entre exigences système et spécification du logiciel

Objectifs de la démarche

- Démarche systématique d'analyse système afin de projeter les exigences SDF sur la spécification des fonctions de sûreté réalisées par le logiciel
  - fonctions de sécurité ou fonctions "essentielles"
  - fonctions de gestion des états et des transitions entre états
    - détection des états dégradés, reconfigurations
    - détection des états dangereux, mise en état sûr
    - reconfiguration, reprise
  - allocation d'un niveau d'intégrité à ces fonctions
  - spécification de la sémantique de panne à prendre en compte
    - (franche, temporelle, byzantine)

Démarche : but et niveaux

But de la démarche :
ermettre au chef de projet et à l'ingénieur système de spécifier des objectifs
e sûreté de fonctionnement sur le logiciel implémenté dans le système.

Plusieurs niveaux sont identifiés selon les projets :
- Niveau système
- Niveau sous-système
- Niveau équipement
- Niveau logiciel.

Démarche : séquentielle et itérative

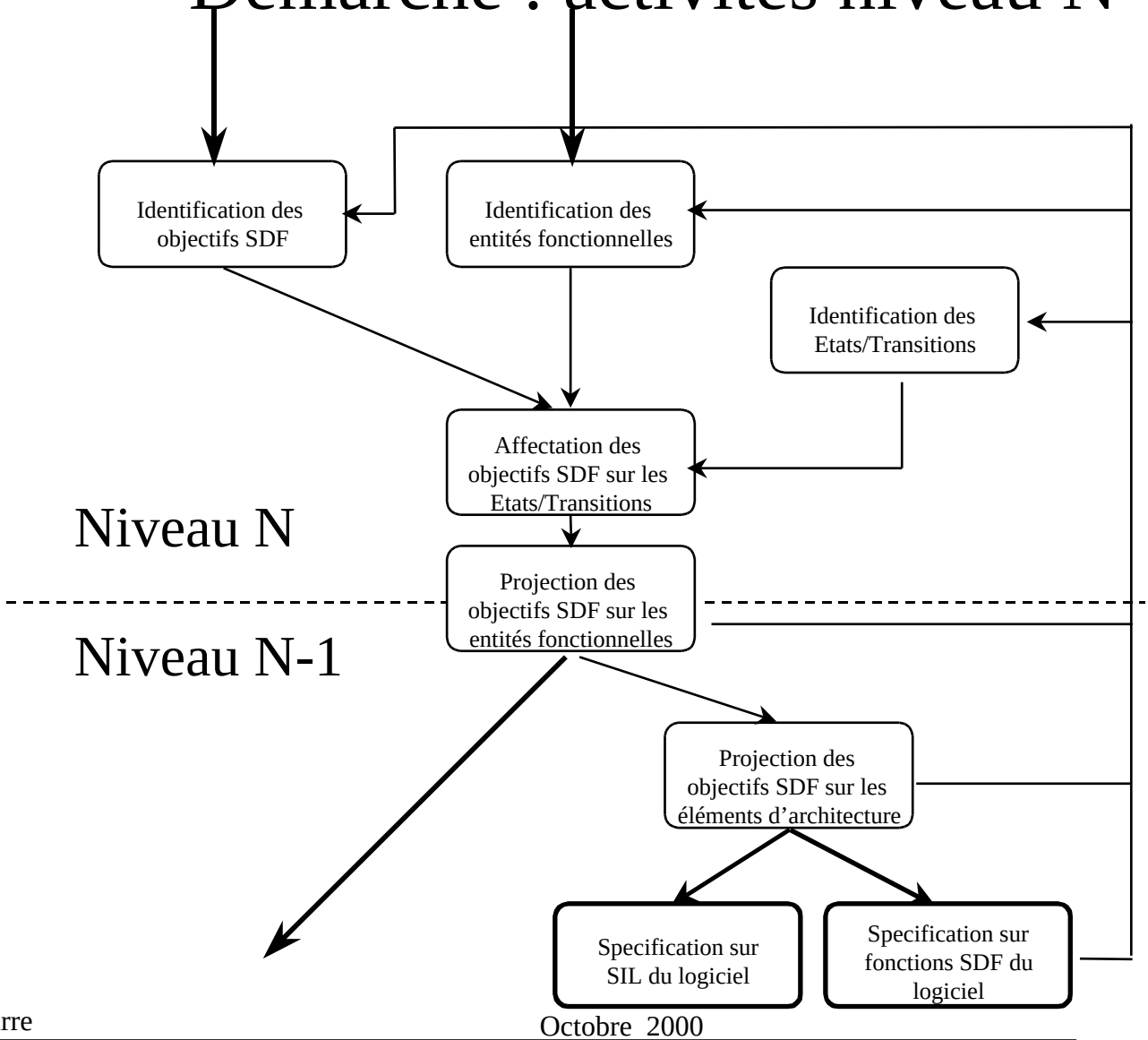
Séquentielle :
on ne peut projeter des objectifs à un niveau donné que si les objectifs de niveau supérieur ont été au préalablement identifiés.

Itérative sur 3 points :
au niveau donné : on vérifie que les objectifs alloués permettent de tenir l'objectif de niveau supérieur.
Cette vérification peut engendrer des ajustements et des raffinements sur les objectifs SDF des composants du niveau courant.

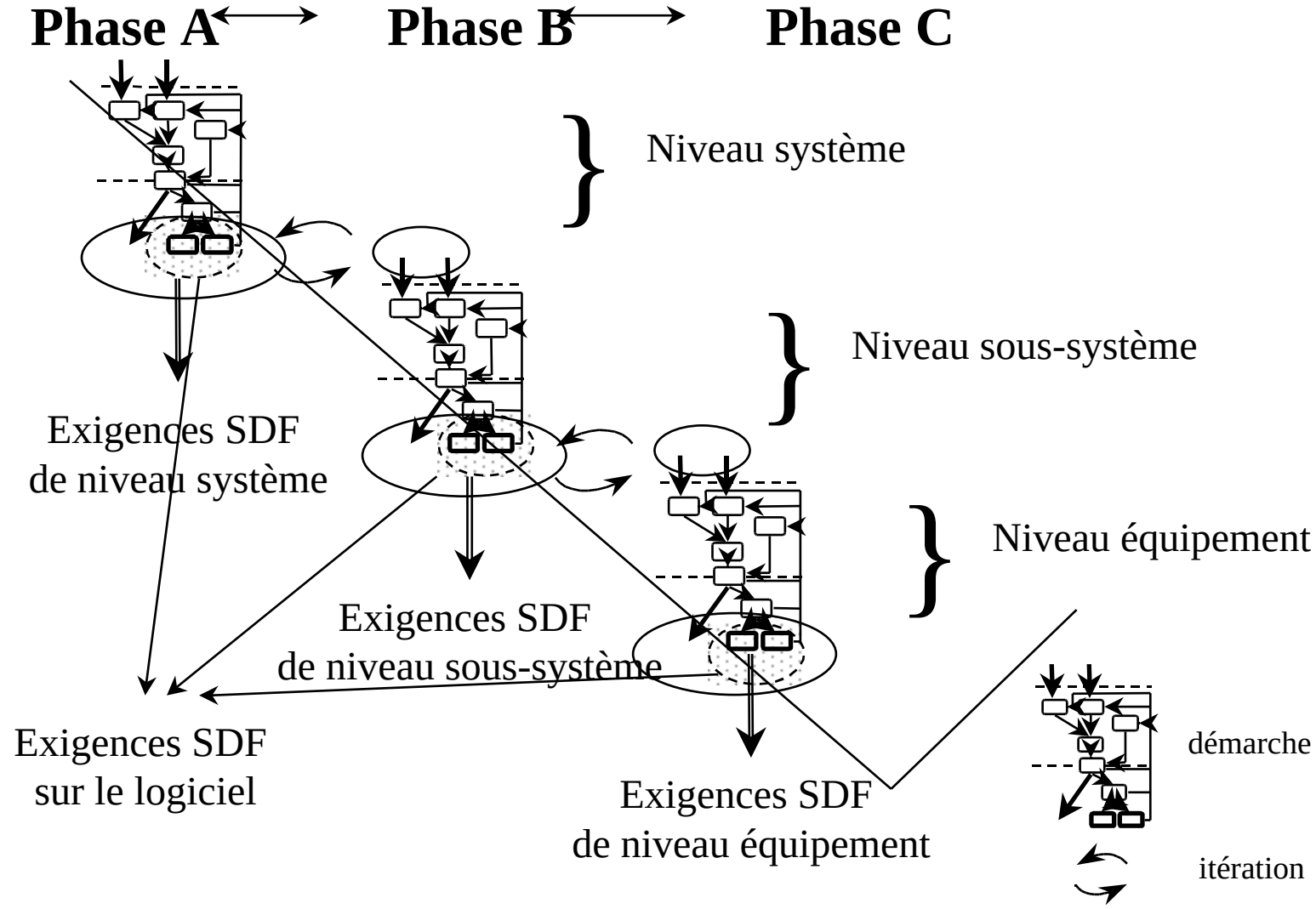
entre niveaux : les composants du niveau inférieur peuvent ne pas tenir les objectifs SDF et remettre en cause les objectifs identifiés.

au niveau système, les activités au niveau système portent sur plusieurs options d'architecture, et la convergence vers une solution optimum passe par les itérations

Démarche : activités niveau N



Démarche : instanciation sur un projet



Démarche : étape 1

ctivités	Identification des entités fonctionnelles élémentaires
bjectifs	Développer un niveau de compréhension suffisant du système et de ses fonctions au niveau considéré
esp.(*)	M/OU
ntrées	Tout document ou toute information nécessaire pour mener à bien une analyse fonctionnelle pertinente au niveau considéré
rties	Analyse fonctionnelle du niveau supérieur Diagrammes de flux

Démarche : étape 2

Activités	Identification des caractéristiques et objectifs SDF
Objectifs	Au niveau système : Déterminer les risques liés à la mission et à l'utilisation du système, les risques liés aux défaillances des fonctions du système. Evaluer la gravité de ces risques Aux niveaux inférieurs : Faire l'inventaire des risques identifiés aux niveaux supérieurs qui concernent le niveau auquel on se situe
Resp.	M/OU
Entrées	Domaine et phases d'utilisation du système, ou de la fonction étudiée, identification des risques externes potentiel Analyse fonctionnelle jugements d'experts Exigences qualitatives
Sorties	Identification des risques liés au système au niveau considéré (événements redoutés caractérisés gravité/probabilité d'occurrence) Evaluation de la gravité des risques Exigences qualitatives

Démarche : étape 3

Activités	Identification des Etats/Transitions
Objectifs	Elaborer un **graphe des états** du système sur le niveau considéré afin d'avoir une meilleure appréhension des évolutions du système, et de viser l'exhaustivité des spécifications à émettre
Resp.	M/OU M/OE ST
Entrées	Tout document ou toute information nécessaire pour me à bien un graphe d'états pertinent au niveau considéré
Sorties	Description des états et des transitions, graphe d'états

Démarche : détail étape 3

Etape 3 : Identification des Etats/Transitions (1/4)

on distingue :
•la classe des états nominaux,
•la classe des états de fonctionnements dégradés,
•la classe des états de sauvegarde
•l'état dans lequel la mission ne peut pas être réalisée.

Les transitions peuvent être entre états d'une classe ou entre classes.
Les transitions entre classes correspondent à des transitions entre modes de fonctionnement du point de vue de la sûreté de fonctionnement

Classification des états/SDF

Classe des états nominaux

Toutes les fonctions du système rendent le service conformément à sa spécification
On peut identifier les états selon la phase ou le domaine d'utilisation de l'entité au niveau considéré (ex : on distingue les phases de stand-by des phases " actives ")

Classe des Etats dégradés

Le système est dans un état appartenant à la classe des états dégradés :
·si le service est rendu partiellement (certaines fonctions sont non disponibles)
·si le service est rendu mais avec des performances inférieures aux performances du nominal.
Un sous ensemble des fonctions délivre le service attendu.
Ce sous ensemble est cohérent et rend globalement un service acceptable.

Classe des Etats Perte de mission technique

Le système ne peut plus assurer sa mission.
Aucun service n'est acceptable d'un point de vue fonctionnel ou les performances sont jugées inacceptables.
La réalisation de la mission dépend des phases et domaines d'utilisation.
Il peut donc y avoir plusieurs types de pertes techniques de mission.

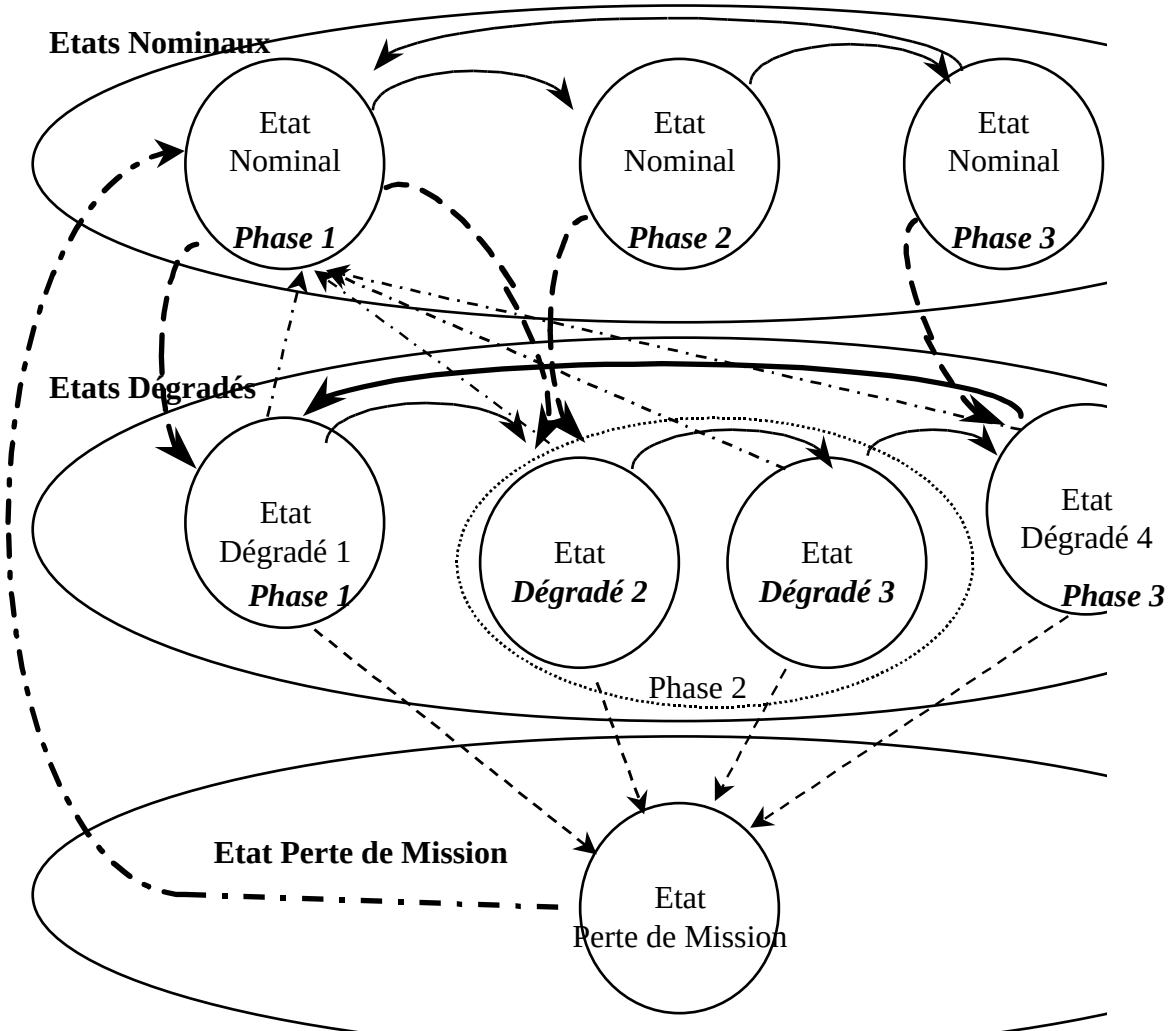
Classe des Etats de sauvegarde

L'état de sauvegarde est un état dans lequel aucun événement redouté jugé inacceptable (au sens de la sécurité) ne peut se produire.
Selon les phases et les domaines d'utilisation, on peut distinguer plusieurs états de sauvegarde.

Transitions SDF

Transitions opérationnelles
transitions liées au comportement nominal du système, passage d'une phase d'utilisation à une autre par exemple.
Transitions de dégradation
transitions qui font passer l'entité au niveau considéré dans un état final plus dégr
que l'état initial compte tenu de l'attribut SDF considéré (dégradation de service, perte de fonction, perte de barrière par exemple).
On peut spécifier sur ces transitions des contraintes de détection
(seuil de détection, taux de détection, alarmes) ou de temps d'action.
Transitions de reconfiguration
transitions qui font passer l'entité au niveau considéré dans un état final moins dé
que l'état initial compte tenu de l'attribut SDF considéré
(recouvrement de fonctions, retour à un état nominal, etc.).
Transitions de sécurité
transitions qui font passer l'entité dans l'état de sauvegarde.
Les contraintes appliquées à ces transitions peuvent être temporelles.
Les conditions de réalisation sont liées à la détection du danger.

Graphe des états (aspects disponibilité)



transitions opérationnelles transitions de dégradation transition de reconfiguration

Démarche : étape 4

Activités Affectation des objectifs SDF sur les Etats/Transitions

ObjectifsA partir des risques identifiés en 2, affecter les objectifs SDF sur certains états et toutes les transitions du graphe élaboré en

Resp.(*) M/OU M/OE ST

Entrées Graphe d'états

Sorties Analyse des défaillances du système et Evaluation de leur gravité
Graphe d'états avec affectation de caractéristiques sur les transitions (gravité, interdiction de se produire, caractéristiques temporelles, etc.)

Affectation des objectifs SDF sur les Etats/Transitions

L'analyse revêt deux aspects :

un aspect qualitatif, qui cherche à vérifier que l'architecture fonctionnelle atisfait aux exigences de sûreté de fonctionnement en terme d'existence de modes dégradés ou de redondances fonctionnelles,

un aspect quantitatif, qui cherche à allouer des objectifs de probabilité l'occurrence aux événements internes (dégradations, échecs des econfigurations,…).

Propriétés de comportement /SDF

nalyse Qualitative

entification pour chaque événement redouté “ racine ” de tous les chemins sur le raphe qui mènent à cet événement

ude des événements internes liés aux choix d’architecture fonctionnelle compte nu de la gravité de l’événement ciblé et de la combinatoire des transitions.

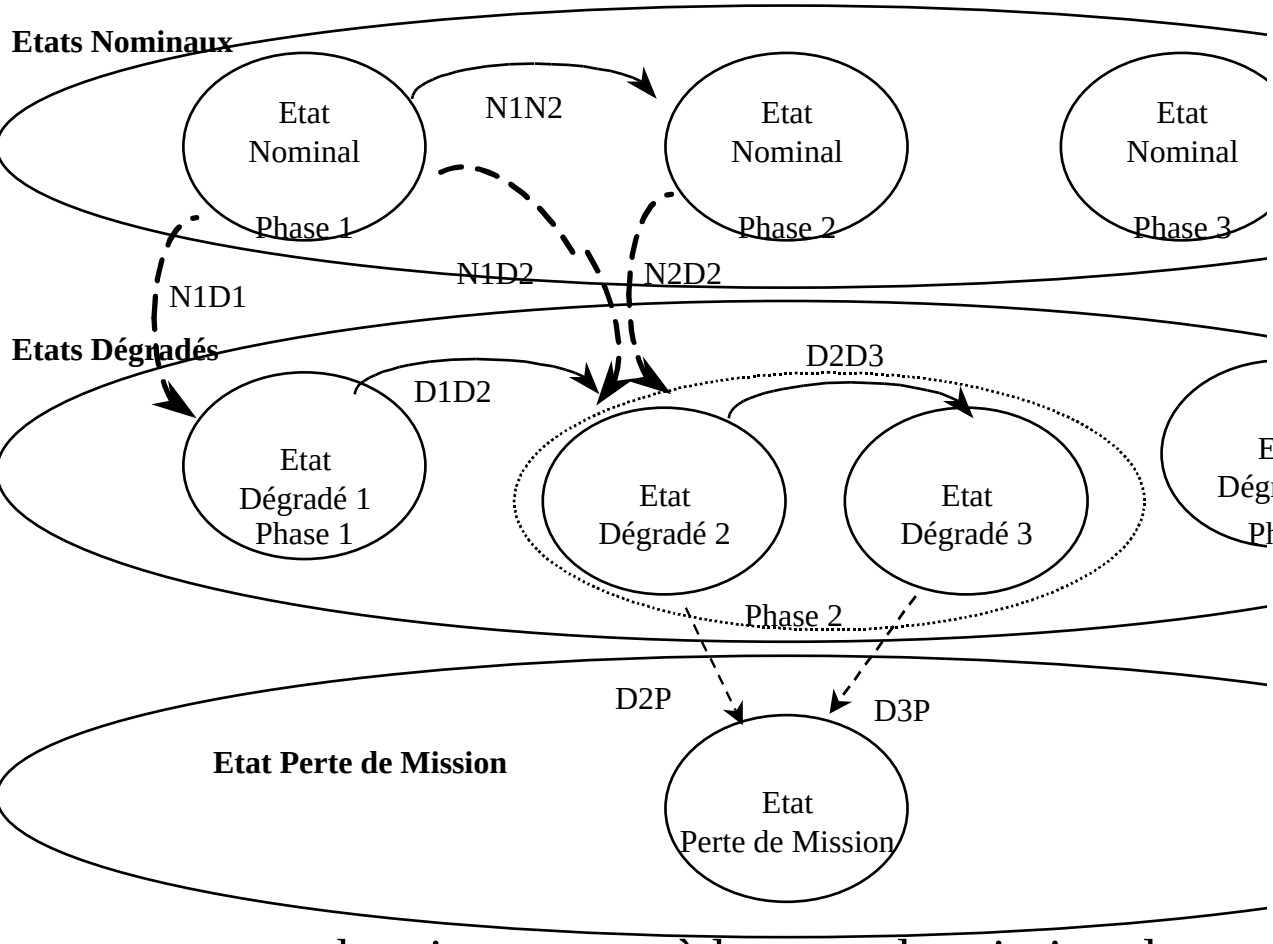
érification que les modes dégradés ou les redondances fonctionnelles spécifiées en hase d’analyse système sont suffisantes vis à vis des objectifs de tolérance aux pann

érification que les transitions entre modes sont correctement séquencées

érification de la cohérence mutuelle des modes

noncé de propriétés logiques telles que les séquences interdites ou s enchaînements nécessaires (logique temporelle)

Exemple comportement/SDF



chemins menant à la perte du mission du système

Exigences quantitatives /SDF

Analyse quantitative
envisageable si on peut faire des hypothèses sur les taux d'occurrence des différentes transitions.

Analyse par des techniques d'analyse sont assez classiques (réseaux de Pétri, graphes de Markov, arbres de défaillance, diagrammes de fiabilité).

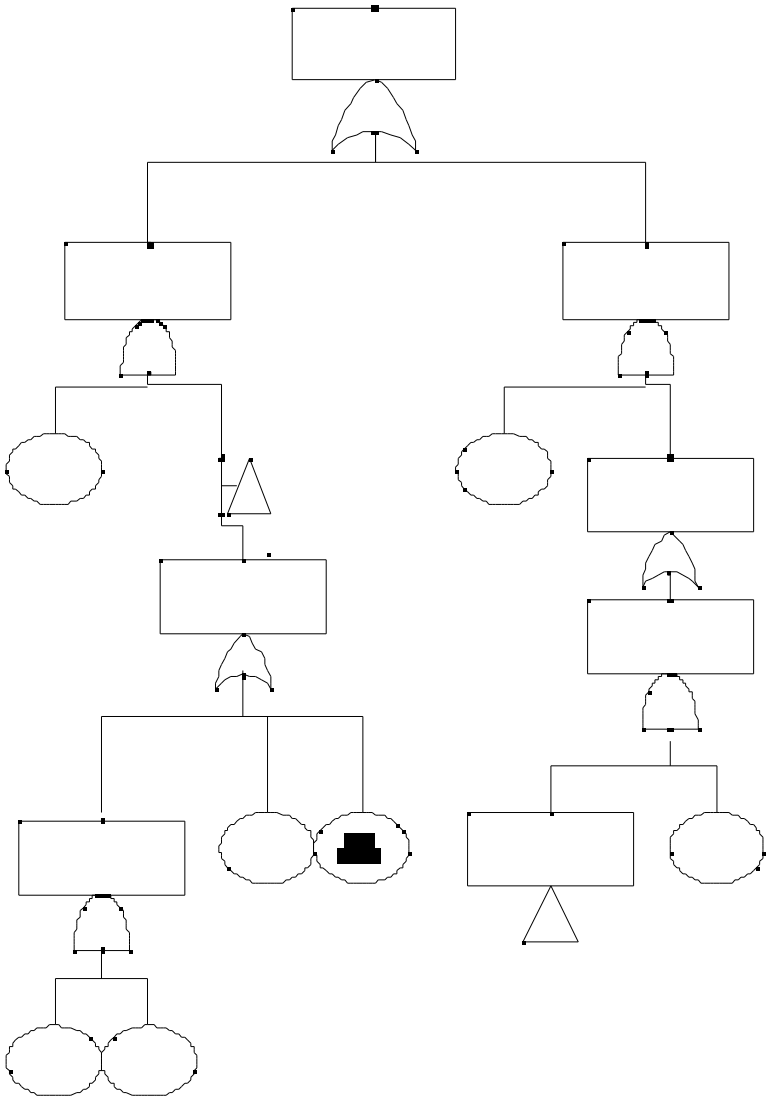
L'analyse peut servir à consolider :
les taux d'occurrence des événements vis à vis des objectifs globaux tels que le temps de mission (temps avant arrivée en
état "perte de mission")
la disponibilité de certains modes (temps de séjour en états transitoires)…

Plus spécifiquement aux critères de gravité et de criticité des événements, on peut procéder à une allocation par arbre de fautes probabilisé, en étudiant séparément les événements cibles.

Comme pour l'analyse qualitative, la démarche d'analyse quantitative devra être progressive en fonction de la nature de la taille et/ou de la criticité du projet

En sortie on aura affecté aux transitions du graphe un niveau de criticité

Exemple : Utilisation d ’AER



arbre de faute enrich
des allocations
quantitatives

Démarche : étape 5

Activités	Projection des caractéristiques SDF sur les entités fonctionnelles
Objectifs	Faire le lien entre les défaillances des entités fonctionnelles et les transitions; la gravité des défaillances des entités fonctionnelles est directement héritée de la gravité de la transition à laquelle elle est liée
Resp.(*)	M/OU M/OE ST
Entrées	Graphe d'état enrichi
Sorties	Analyse fonctionnelle Analyse des défaillances du système et Evaluation de leur gravité Synthèse donnant selon les phases d'utilisation et les risques externes, les objectifs SDF sur les défaillances des entités fonctionnelles

Démarche : détail étape 5

Trois types de résultats sont attendus :
•l’identification de nouvelles fonctions (le graphe met en évidence des fon
•à prendre en compte),
•les comportements prévus ou à spécifier des fonctions,
•le niveau d’intégrité requis sur les fonctions.

Activité menée en 8 étapes :
•Classement des fonctions
•Caractérisation des comportements des fonctions
•Réalisation du lien entre fonctions système et transitions du graphes d’éta
•Allocation d’Exigences fonctionnelles
•Identification des modes communs
•Création de nouvelles fonctions
•Caractérisation du niveau d’intégrité
•Validation des allocations

Démarche : détail étape 5

Classement des fonctions :

Fonctions de sécurité

fonctions qui servent à éviter l'occurrence d'un événement dangereux ou à maintenir le système dans un état sûr.

Fonctions de Disponibilité

Les fonctions de disponibilité sont 3 types :

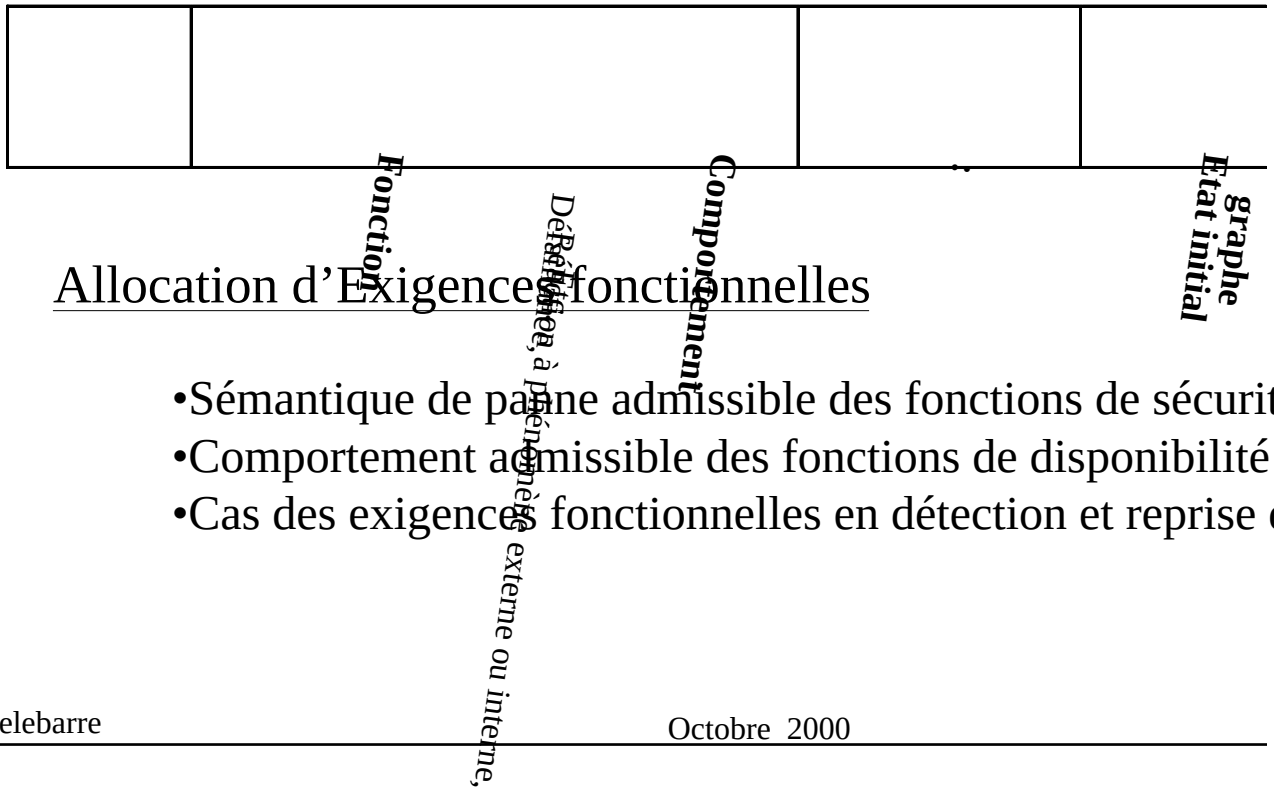
- les fonctions opérationnelles proprement dites
- les fonctions de détection
- les fonctions de maintien dans un état de fonctionnement ou de reconfiguration.

Caractérisation des comportements des fonctions

il faut se focaliser sur ceux qui amènent un changement d'état du système.
Les comportements étudiés peuvent être des défaillances, des sollicitations anormales d'autres fonctions du système, des sollicitations anormales du milieu extérieur.
Une même fonction peut avoir plusieurs modes de défaillance.

Démarche : détail étape 5

Lien entre fonctions système et transitions du graphe d’états



Allocation d’Exigences fonctionnelles

- Sémantique de panne admissible des fonctions de sécurité
- Comportement admissible des fonctions de disponibilité
- Cas des exigences fonctionnelles en détection et reprise d'erreur

Démarche : détail étape 5

Identification des modes communs

Certaines fonctions concernent plusieurs transitions.
On se focalise en particulier sur les fonctions transverses comme :
les fonctions de supervision et gestion des modes de fonctionnement :

- Fonction de gestion de la dégradation
- Fonction de reconfiguration,
- Fonction de maintien dans un état en présence de défaillance inte

les fonctions de maintien de la sécurité/surveillance

- Fonctions de contrôle des états pertinents de la sécurité
- Fonctions de détection des évènements/états dangereux
- Fonctions de réalisation des actions de sauvegarde

Plus proche du matériel, on vérifie aussi les fonctions transverses comme l'alimentation du système, ou les fonctions interface E/S.

On peut aussi à cette étape spécifier/valider les grandes lignes de procédure d'utilisation.

Démarche : détail étape 5

Création de nouvelles fonctions

Après confrontation entre les transitions du graphe des états et les fonctions connues du système, certaines transitions indispensables peuvent ne pas être couvertes.

Caractérisation du niveau d’intégrité

Pour toute fonction on alloue un niveau d ’intégrité qui correspond à la fo
à la criticité de l’événement redouté qui est couvert par ce sous ensemble e
à un objectif de défaillance “ fonctionnel ”.
Le niveau d’intégrité de la fonction est directement hérité de la transition à laquelle la fonction est liée.
Dans le cas ou la fonction est liée à plusieurs transitions de niveaux d’intégrité différents, on affecte à la fonction le niveau d’intégrité le plus fort des transitions considérées.

Démarche : détail étape 5

Valider les allocations

•montrer que les risques (axe sécurité) sont clos ou que le risque résiduel est acceptable (analyse exhaustive)
•montrer que la politique de réduction des risques est respectée,
•montrer que les fonctions liées à la fiabilité sont mises en oeuvre conformém à la politique de sûreté de fonctionnement.
•montrer que les mécanismes proposés de reconfiguration sont conformes à l politique de sûreté de fonctionnement, par traçabilité avec les transitions de reconfiguration identifiées.

Démarche : étape 6

Activités	Projection des caractéristiques SDF sur les éléments d'architecture
Objectifs	Faire le lien entre les éléments d'architecture et les entités fonctionnelles qu'ils supportent; la gravité de leur défaillance est directement héritée de la gravité des défaillances des entités fonctionnelles supportées.
Resp.(*)	M/OE ST
Entrées	Synthèse élaborée en étape 5
Sorties	Architecture définie Synthèse donnant selon les phases d'utilisation et les risques externes, les objectifs SDF sur les défaillances des éléments d'architecture

Démarche : étape 7

Activités	Critère d'arrêt
Objectifs	Décision d'itérer Evaluer l'homogénéité des gravités des entités fonctionnelles au niveau considéré et juger de l'intérêt d'une itération à un niveau inférieur
Resp.(*)	M/OE ST
Entrées	Synthèse donnant selon les phases d'utilisation et les risques externes, les objectifs SDF sur les défaillances des entités fonctionnelles
Sorties	Décision de poursuivre d'une itération

Démarche : étape 8

ctivités	Spécification des Fonctions SDF du logiciel
bjectifs	Spécifier les fonctions SDF du logiciel (existence, propriétés, domaine d'application, interface, …)
esp.(*)	M/OE ST
ntrées	Synthèses donnant selon les phases d'utilisation et les risques externes, les objectifs SDF sur les défaillances des entités fonctionnelles
orties	Spécification fonctionnelle et opérationnelles sur les fonctions SDF

Démarche : étape 9

Activités	Spécification des IL des modules du logiciel
Objectifs	Spécifier l'IL des modules logiciel
Resp.(*)	M/OE ST
Entrées	Synthèse donnant selon les phases d'utilisation et les risques externes, les objectifs SDF sur les défaillances des éléments d'architecture
Sorties	IL des modules logiciel

Formalismes recommandés

- Pour effectuer les étapes de spécification des exigences sdf, on utilise principalement :
  - les RDP et dérivés
  - les AEFC et outils d ’exploration de graphe.