

Introduction aux systèmes temps réel (Partie 2)

Iulian Ober
GRIMM/ISYCOM
ober@univ-tlse2.fr

Sommaire

- Caractéristiques récurrentes des STR
- Notions de base en STR
- Types de problèmes soulevées
- Quelques résultats de base

En 2^{ème} partie:

- Modélisation et développement
modèles, langages, architectures
- Sûreté de fonctionnement
et comment l'obtenir / la prouver

Modèles computationnels

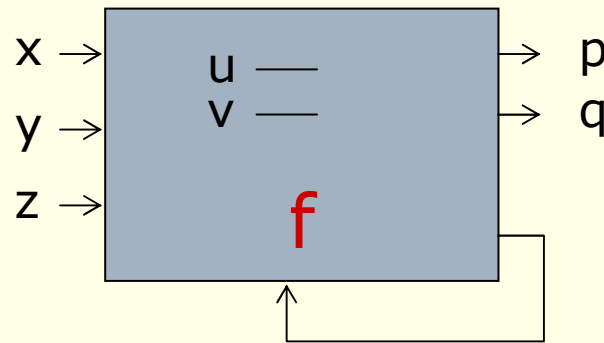
Quelques exemples^(*):

- CSP – threads concurrents avec RDV
- CT | DT – modélisation en temps continu / discret
- DE – système d'événements discrets
- DDF – flot de données dynamique
- SDF – flot de données synchrone
- SR – synchrone/réactif
- PN – réseaux de processus
- ...

(*) Source: Ptolemy project, UC Berkeley

Flot de données synchrone : SCADE

- formalisme adapté à la modélisation des tâches **périodiques, contrôlées par le temps** (mais pas seulement...)
- la construction de base est le **bloc fonctionnel**

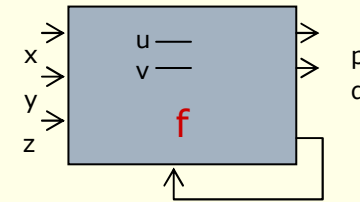


$$(p, q, u', v') = f(x, y, z, u, v)$$

Hypothèse synchrone

- le temps est une succession d'instant (temps discret): Temps = (t_0, t_1, t_2, \dots)
- toute variable ou lien devient une séquence de valeurs aux instants t_0, t_1, \dots

- $x = (x_0, x_1, x_2, \dots)$
- $u = (u_0, u_1, u_2, \dots)$



- les valeurs de l'instant t_k servent à calculer les valeurs de sortie de l'instant t_k et les valeurs de mémoire de l'instant t_{k+1}

$$(p_k, q_k, u_{k+1}, v_{k+1}) = f(x_k, y_k, z_k, u_k, v_k)$$

Hypothèse synchrone

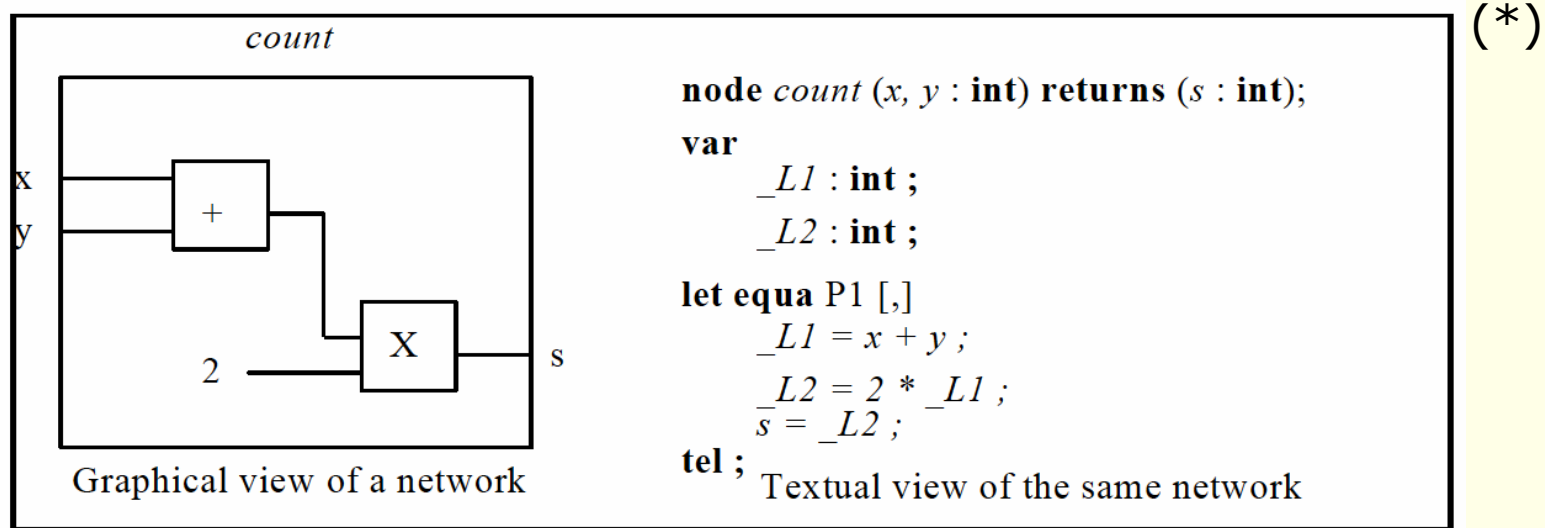
- **les moyens de calcul sont suffisants pour que les calculs du cycle k se terminent avant t_{k+1}**
- pour le programmeur, c'est comme si les calculs de son programme (f) étaient **instantanés**
- On *considère a priori* que la granularité du temps est suffisante pour que l'**hypothèse synchrone** soit satisfaite
- On *vérifie a posteriori* que c'est vrai

Flots de données

Un bloc est décrit par un ensemble d'équations

Avantages :

- **Parallélisme** maximisé : seules contraintes sont les *dépendances entre données*
 - optimisations et parallélisation faites par le compilateur
- Propriétés mathématiques \Rightarrow **vérification** formelle



(*) from the *SCADE Language Reference Manual*, Esterel Tech. 2005

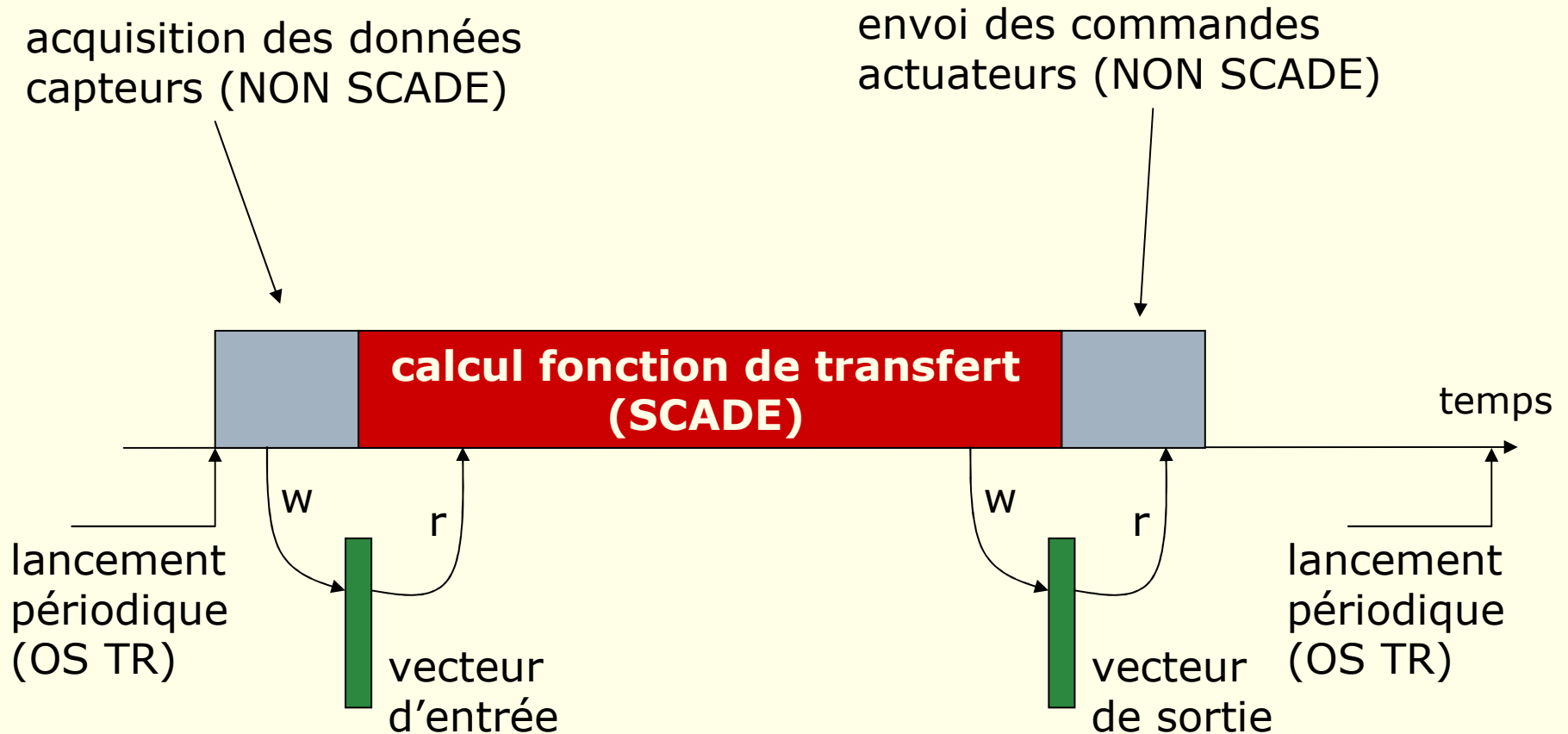
Flot de données (cont)

- le langage est **fonctionnel, déclaratif**
(les équation ne sont *pas séquencées*)
- **causalité** : les sorties d'un nœud *ne peuvent pas dépendre d'entrées du futur*
- les sorties dépendent d'une *quantité limitée d'entrées du passé*

Validation temporelle

- le comportement d'une application est **indépendant de la fréquence d'exécution** (tant que l'hypothèse synchrone reste vraie)
- a posteriori, on vérifie sur cible que le WCET est inférieur à la période
- résultat : un système SCADE est déterministe dpdv. **fonctionnel** et **temporel**

Utilisation typique



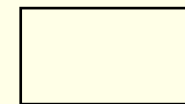
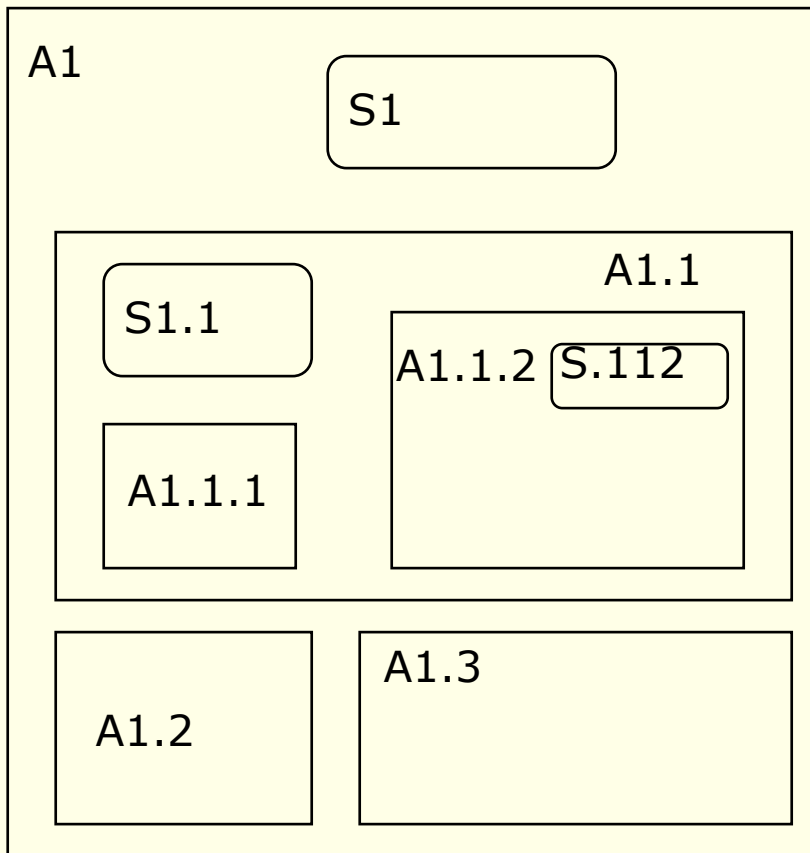
Quelques utilisations

- Airbus : calculateurs de bord A380 (part.)
– code certifié DO-178B
- EADS Space Tr. : séquenceur de vol Ariane-5, ATV, etc. (part.)
- Automobile, transport ferroviaire et équipement lourd

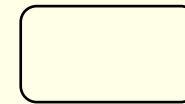
Réactif synchrone : STATECHARTS

- formalisme adapté à la modélisation de systèmes
 - **concurrents**
 - **ouverts** communicants (en interne et avec l'environnement) par des **événements** et/ou **données**
 - possibilité de **temporisation explicite**

Constructions de base



Activité

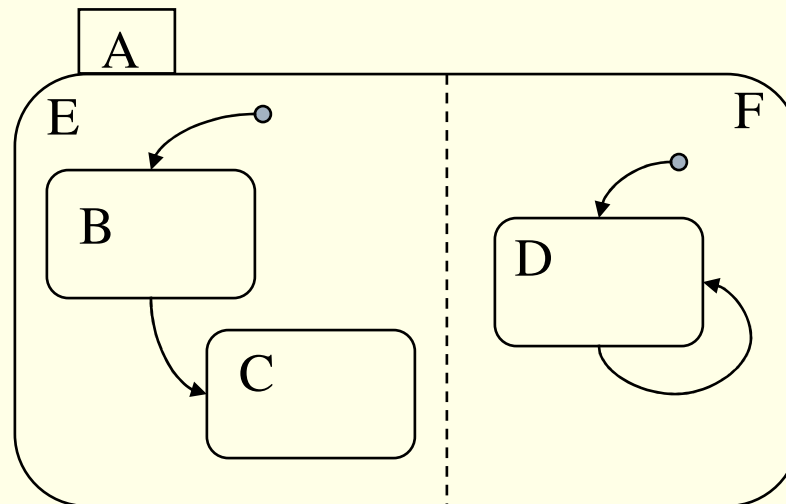


**statechart
(élément de
contrôle)**

- décomposition en *Activités* orientée flot de données
- maximum *1 Statechart par Activité* – gère le lancement des sous-activités

Constructions de base (cont)

- **Statechart** = automate hierarchique
 - états OR
 - états AND
 - états basiques



Étiquetage des transitions



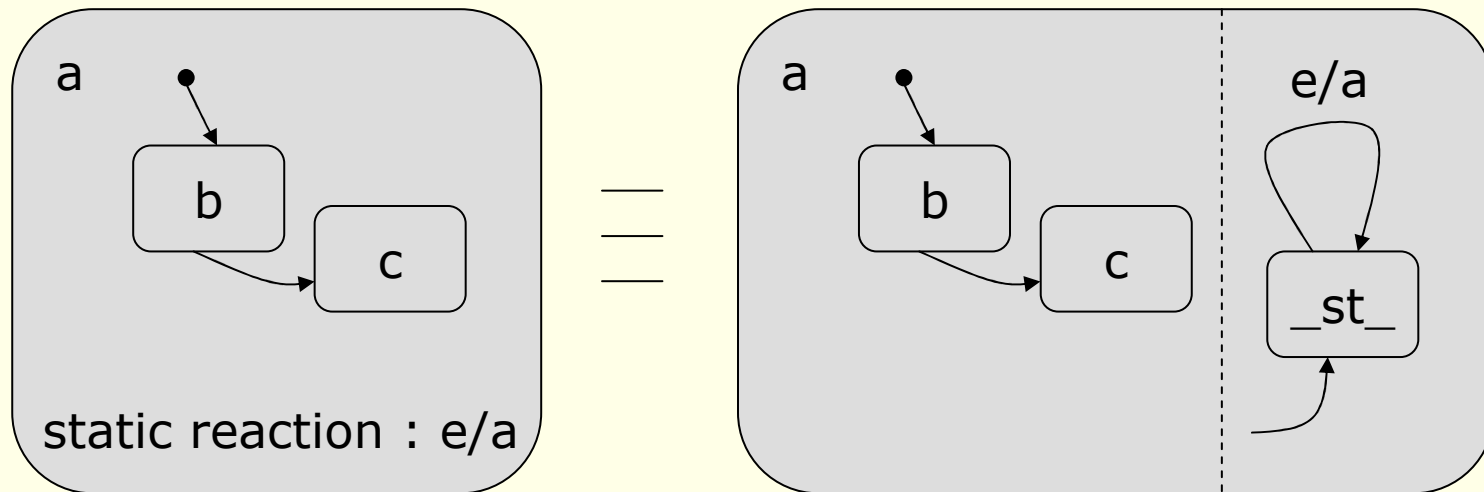
- e : événement
- c : condition booléenne sur les données (garde)
- a : action effectuée par la transition
 - affectation de variable
 - émission d'événement
 - start/stop *Activité*
 - ...

Plus d'actions

Actions sur état

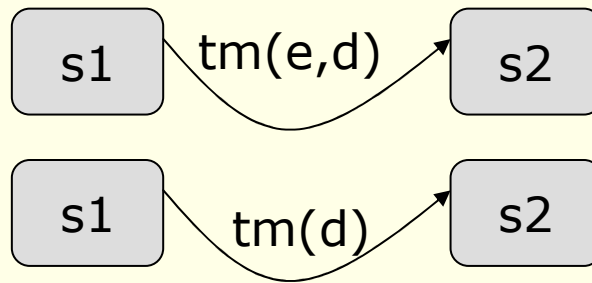
- entry
- exit
- réaction statique

Exemple:



Temps explicite

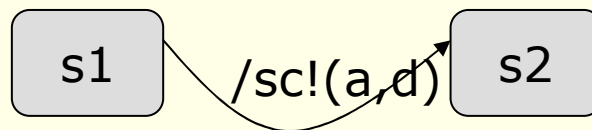
□ timeout



- a lieu d unités de temps après e

- a lieu d unités de temps après l'entrée dans $s1$

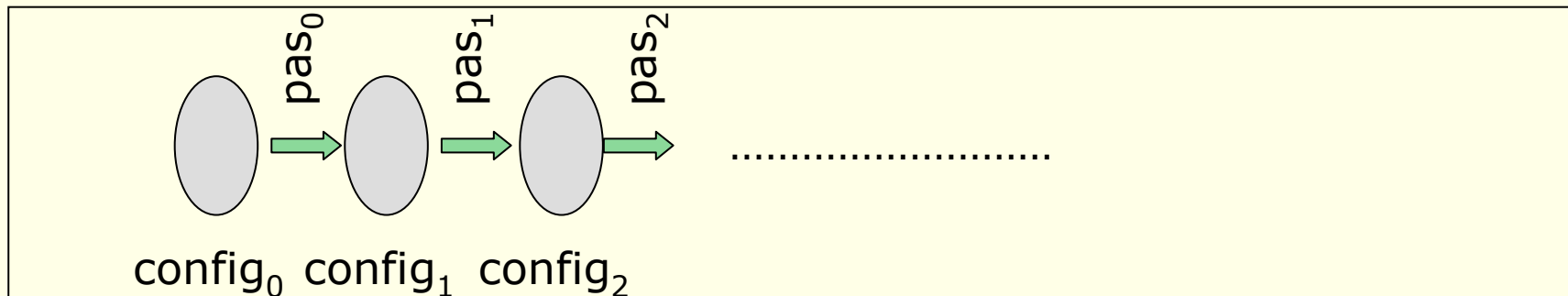
□ planification des actions (schedule)



- a aura lieu d unités de temps après l'exécution de cette action

Comportement du système

- une *exécution* est définie comme une séquence de *configurations* (status)

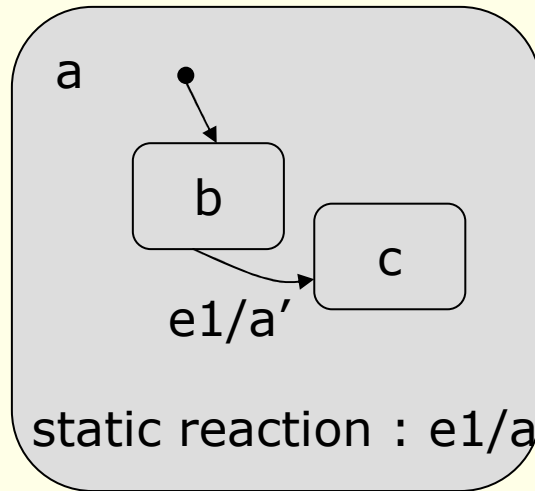


- **configuration** : ensemble d'information caractérisant l'état du système
 - états des statecharts
 - valeurs des variables
 - événements générés dans le dernier pas
- **pas** : changement de la configuration suite à un evt. extérieur ou à un changement interne

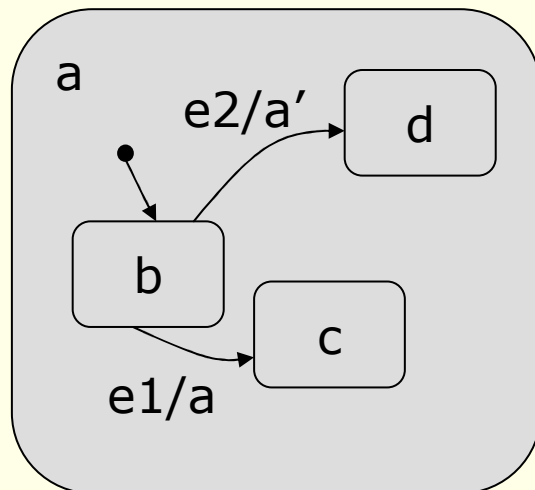
Pourquoi c'est synchrone?

- dans chaque pas, un **ensemble maximal** de transitions (sans conflit) est exécuté
- les événements générés dans un pas sont disponible uniquement dans le pas suivant (*ni avant, ni après*)
 - les événements sont visibles dans tous les sous-états (concurrents) d'un statechart
- les changements des variables sont visibles seulement dans le pas suivant
- toutes les décisions sont basées sur l'état avant le début du pas!
- abstraction : le pas s'exécute en **temps zero** (*Lire : le système se stabilise entre 2 événements successifs venant de l'environnement*)

Synchronie : exemples



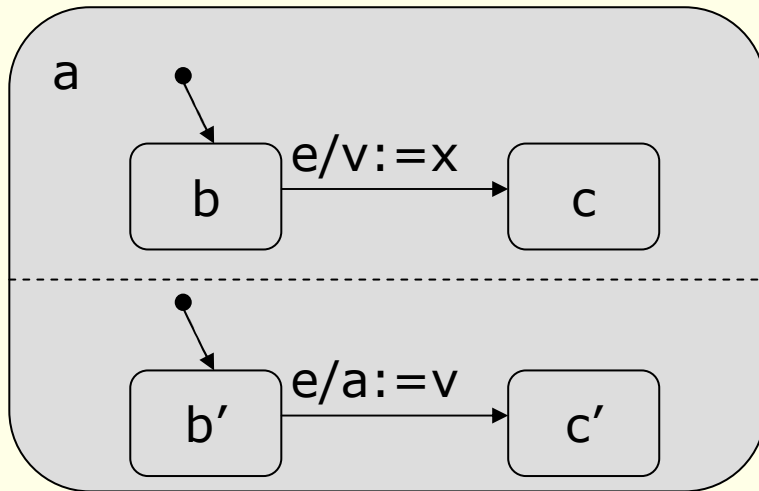
- si $e1$ arrive en b ,
 a et a' sont exécutées toutes les deux
(*ensemble maximal*)



- si $e1$ et $e2$ arrivent pendant b ,
seule une des a ou a' est exécutée
(*ensemble maximal sans conflit*)

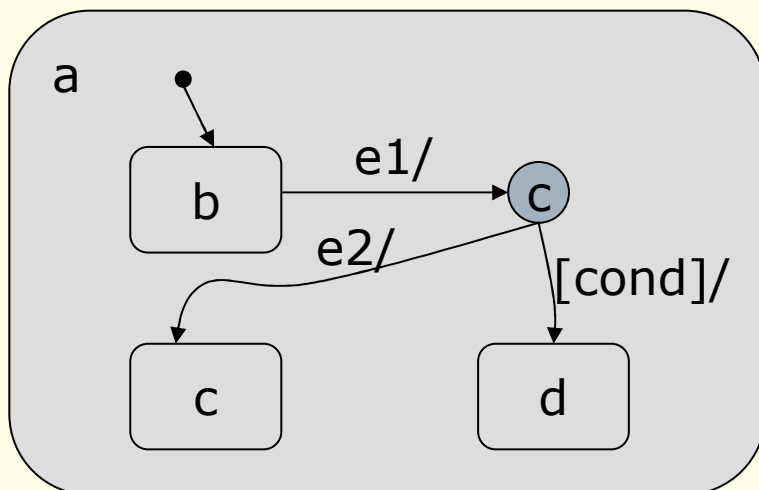
⇒ peut générer du *non-determinisme*

Synchronie : exemples



- si e arrive, pas de conflit :
 - a prend la valeur de v *du pas courant*
 - v prend la valeur de x *dans le pas suivant*

⇒ mais si les deux écrivaient v
il pourrait y avoir un conflit
(non-determinisme – race condition)

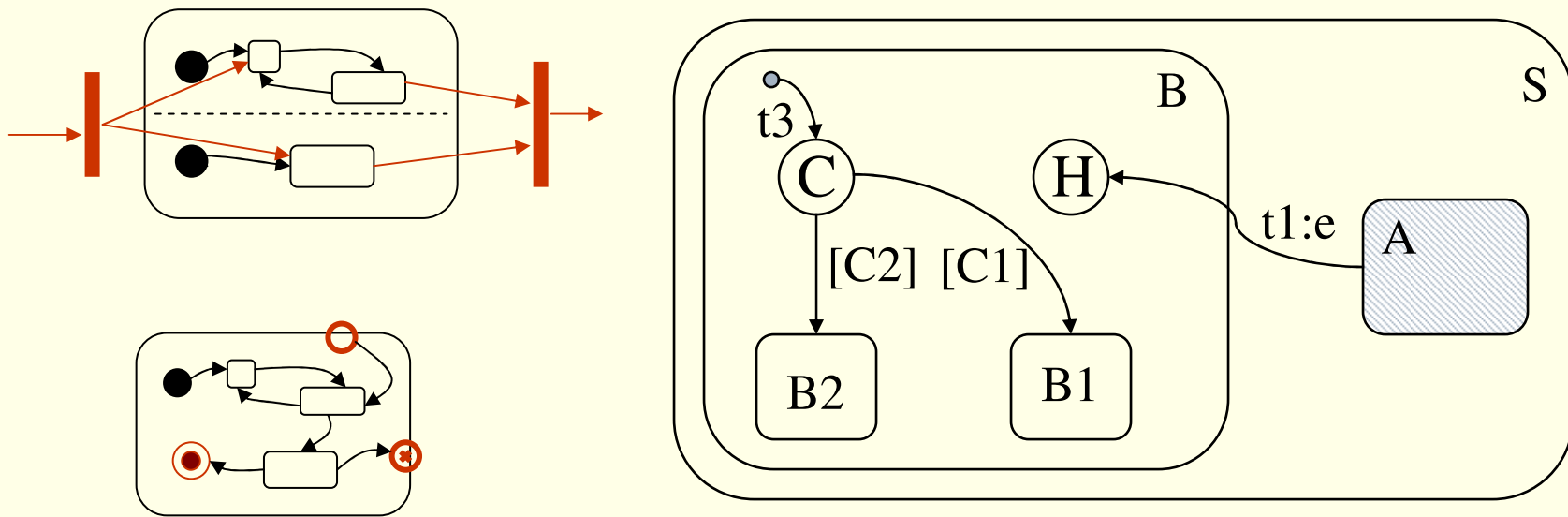


- états instables (e.g., conditions)
⇒ transitions composées
 - la faisabilité est vérifiée avant de faire le pas
 - si e1 & e2 présents ⇒ c
 - si e1 présent & cond vraie ⇒ d
- (possible non-determinisme)*

Sophistication

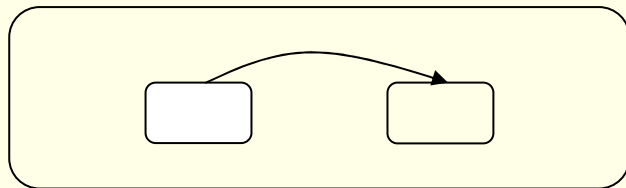
En réalité les choses sont encore plus compliquées...

- plus de constructions

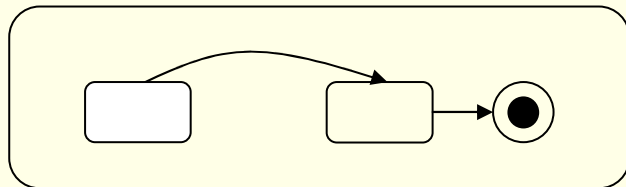
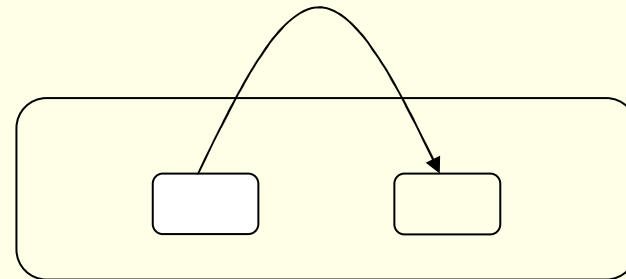


Variations sémantiques

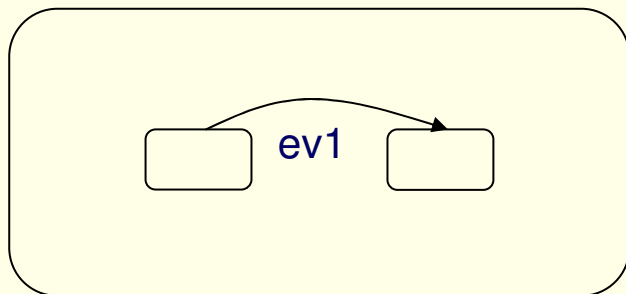
En réalité les choses sont encore plus compliquées...



vs.



/ no trigger here */*



Statecharts : en conclusion

- formalisme **très puissant**
- ...mais sémantique difficile, non-standard...

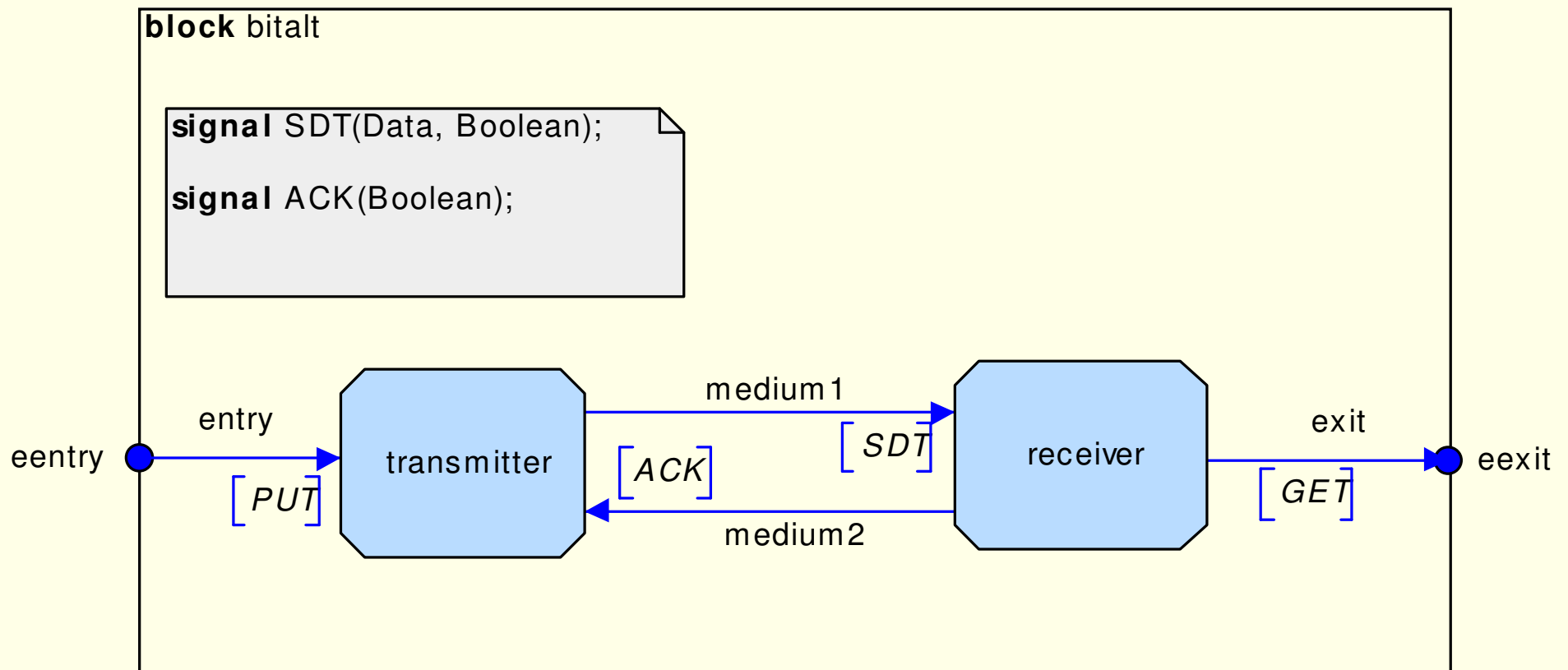
- implémenté dans Statemate, Rhapsody (UML2.0)
- très répandu et utilisé
 - **systèmes embarqués grande consommation**
 - **militaire, avionique, automobile, ...**

Réactif asynchrone : LDS

- LDS : formalisme adapté à la modélisation de systèmes
 - **concurrents**
 - **ouverts** communicants (en interne et avec l'environnement) par des **événements asynchrones**
 - possibilité de **temporisation explicite**
- peu adapté pour le contrôle-commande cyclique
- exemples : protocoles de communication
⇒ norme de définition pour les protocoles IUT-T, ETSI, etc. (UMTS, GSM, Bluetooth,...)

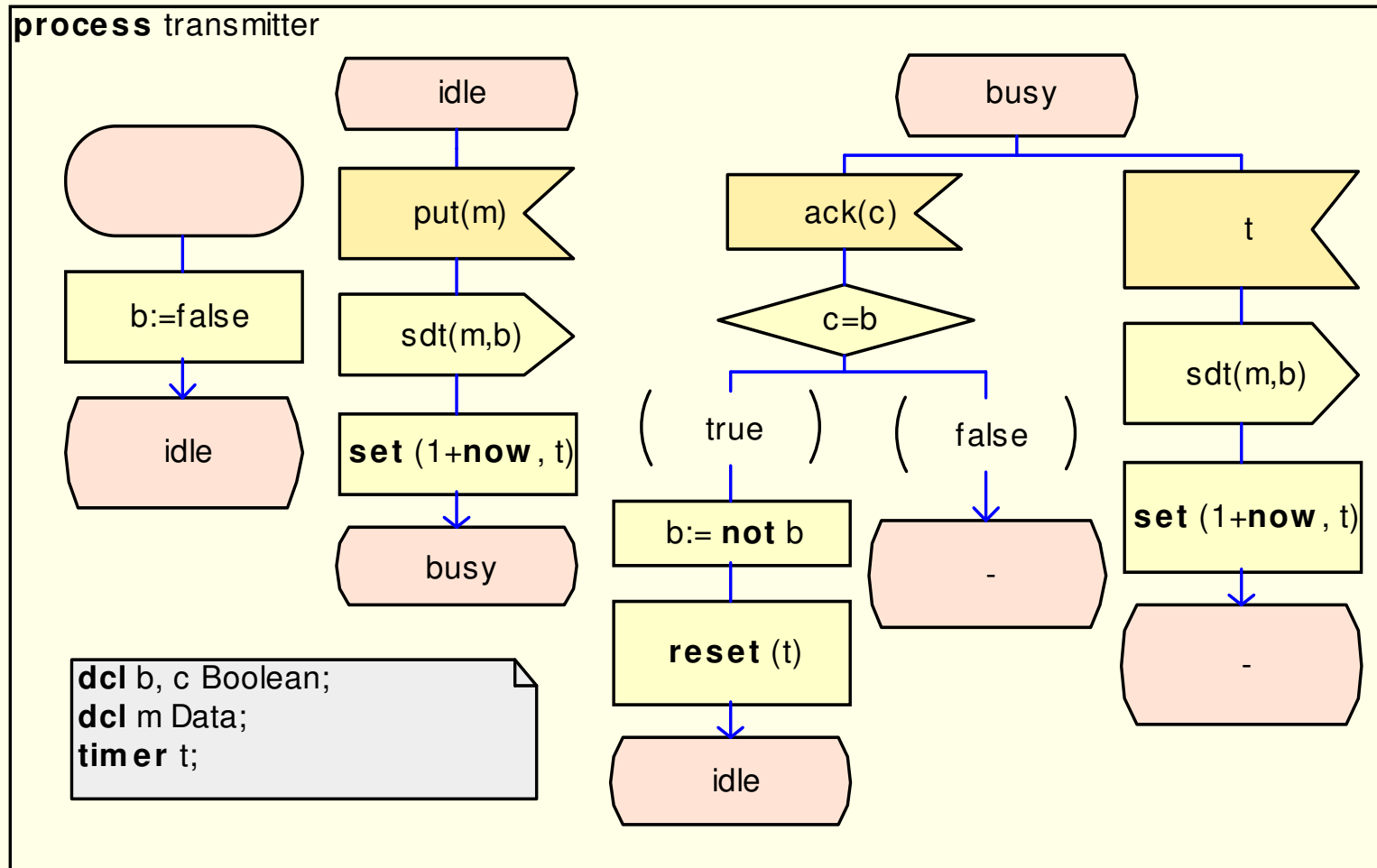
Concepts de base

- décomposition architecturale du système



Concepts de base

□ machines à états pour le comportement



Asynchronie

- ❑ vitesses relatives des processus inconnues
- ❑ pas de notion de *pas global*
- ❑ les signaux sont stockés dans des files d'attente – valables indéfiniment
- ❑ changements des variables valables immédiatement
(en contrepartie, variables locales, processus quasi-séquentiels)

⇒ non-déterminisme endémique

Caractéristiques

- sémantique formellement définie, normalisée par ITU-T
- outils : compilation, simulation, vérification, génération de tests, ...
- utilisation en baisse mais la plupart des concepts sont repris en **UML2.0**.
- utilisation principale : protocoles de communication
⇒ norme de définition pour les protocoles
IUT-T, ETSI, etc. (UMTS, GSM, Bluetooth, ...)

Systemes d'exploitation TR

Fonctions principales:

- gestion des tâches
- gestion de la mémoire
- E/S
- communication entre tâches (IPC)

⇒ comme tout OS, plus:

- support pour la prédictibilité temporelle
- rapidité, ressources limitées

OS RT : Typologie

- extensions RT UNIX (RT-LINUX, RT-MACH...)
 - + services, - modulaires/performants/prédictibles

- OS RT « recherche »

- OS commerciaux légers et rapides (QXN, pSOS, VxWorks, WindowsCE)
 - génériques, multi plateforme
 - microkernel - modulaires
 - services : ordonnancement par priorité, schéma d'allocation de mémoire rapide est déterministe, IPC asynchrone (mailboxes), gestion des interruptions en 2 phases, etc. ⇒ **POSIX RT**
 - parfois : versions + légères pour applications particulières (10KB mem., processeurs low-end,...)

- ...voire **pas de OS du tout (bare system)**

Sommaire

- ❑ Caractéristiques récurrentes des STR
- ❑ Notions de base en STR
- ❑ Types de problèmes soulevées
- ❑ Quelques résultats de base

En 2^{ème} partie:

- ❑ Modélisation et développement
modèles, langages, architectures
- ❑ **Sûreté de fonctionnement**
et comment l'obtenir / la prouver

Dépendabilité

= **confiance** dans le service livré par un système

Attributs :

- **Disponibilité**
(temps d'indisponibilité limité)
- **Fiabilité, robustesse**
(capacité de fonctionner sous des conditions dégradées)
- **Sûreté de fonctionnement**
(absence d'erreur catastrophique)
- **Sécurité**
(e.g., confidentialité)
- ...

Exemples:

- contrôle aérien : $< 10^{-9}$ échecs / heure
- système téléphonique : < 1 h indisponibilité / 40 ans

Hypothèses pour la dépendabilité

- Tout système à un pouvoir (de calcul) fini
⇒ les garanties qu'il fournit dépendent
d'hypothèses faites sur *son environnement*
 - **Hypothèses de charge maximale** générée par
l'environnement (e.g., fréquence des requêtes, etc.)
 - **Hypothèses sur la fréquence des pannes non
contrôlables**

- Dans le **scénario pire cas**, un système dépendable doit
être capable de *soutenir la charge maximale sous la
fréquence maximale de pannes*

- La spécification des hypothèses est fondamentale et
doit être **sans ambiguïté (formelle)**

Comment obtenir la dépendabilité

- Éviter les fautes par *construction*, avant qu'elles soient introduites
- Éliminer les fautes par *vérification* avant qu'elles deviennent problématiques
- Tolérer les fautes/pannes en utilisant la *redondance*
- Prévoir les fautes/pannes en *estimant* à l'avance leur nature, leur fréquence, etc.

Comment obtenir la dépendabilité

- ❑ Éviter les fautes par *construction*, avant qu'elles soient introduites
- ❑ Éliminer les fautes par *vérification* avant qu'elles deviennent problématiques
- ❑ Tolérer les fautes/pannes en utilisant la *redondance*
- ❑ Prévoir les fautes/pannes en *estimant* à l'avance leur nature, leur fréquence, etc.

Éviter & éliminer les fautes

- ⇒ méthodologies de conception
- ⇒ formalismes de modélisation

- ⇒ **validation** plus poussée
 - code inspection / walkthrough
 - test avec vérification de couverture
 - **preuve formelle**
 - ...

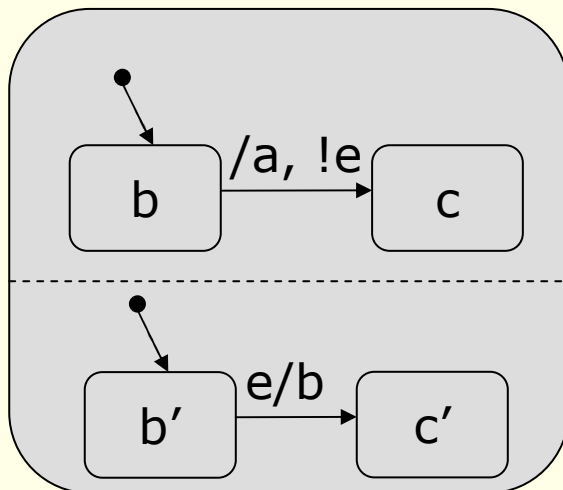
...formelle ?

= il existe une *interprétation* (ou *sémantique*)
du modèle du système
en terme d'**objets mathématiques**
(ensembles, fonctions, graphes, etc.)

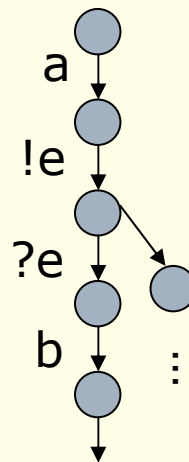
⇒ possibilité de **prouver des propriétés** sur le modèle

Exemple:

modèle



sémantique

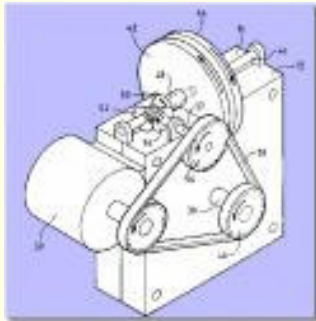


propriété

a arrive toujours
avant *b*

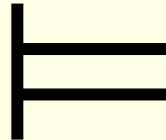
Vérification : ingrédients

modèle
du
système



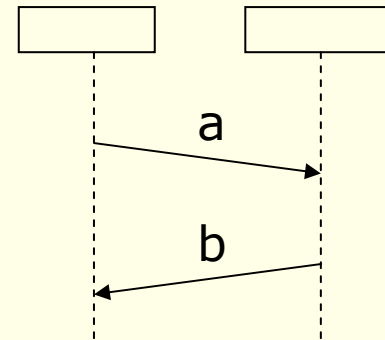
SCADE
LDS
Satatecharts
ADA
UML
...

relation de
satisfaction



Procédure
de
décision

propriété



Logique
(classique, temporelle...)
Automate abstrait
Traces (MSC)
...

Approches (principales)

Approche **déductive**

- système formel (axiomes + règles d'inférence) décrivant l'effet de chaque primitive du langage
- propriétés spécifiées par pré-/post-conditions $\{F1\} P \{F2\}$. Ex.:
 $\{x=(n-1)!\} y:=x*n \{y=n!\}$
- approche très générale...mais souvent **indécidable**
- expertise humaine + utilisation d'*assistants de preuve*

Approches (principales)

Approche **algorithmique** (model checking)

- modèle sémantique sous forme de **graphe d'états / transitions**
- vérification = parcours exhaustif du graphe à la recherche d'erreurs
- complètement automatisable...
- ...mais problèmes avec la taille du graphe (explosion combinatoire)
- *nombreux optimisations possibles* ⇒ applicable en pratique sur système de taille moyenne

Comment obtenir la dépendabilité

- Éviter les fautes par *construction*, avant qu'elles soient introduites
- Éliminer les fautes par *vérification* avant qu'elles deviennent problématiques
- **Tolérer les fautes/pannes** en utilisant la *redondance*
- Prévoir les fautes/pannes en *estimant* à l'avance leur nature, leur fréquence, etc.

Tolérance aux pannes

⇒ obtenue par la **redondance** :

- du matériel
 - méthode passive : réplication + vote
la panne n'est pas corrigée, mais masquée
 - méthode active :
réparation ou remplacement du composant en panne

- du logiciel
 - NVP (N-version programming) + vote
 - RB (recovery blocks) : récupération après exceptions

- des données

Tolérance aux pannes

⇒ obtenue par la **redondance** :

- du matériel
 - méthode passive : réplication + vote
la panne n'est pas corrigée, mais masquée
 - méthode active :
réparation ou remplacement du composant en panne

- du logiciel
 - NVP (N-version programming) + vote
 - RB (recovery blocks) : récupération après exceptions

- des données

TaP et systèmes distribués

Principe: concevoir le système tel qu'il n'y a pas de *point critique unique* (*single point of failure*)

⇒ système distribué (mais pas en étoile !!!)

- nombreux résultats théoriques sur
 - ce que l'on arrive à faire avec des processus coordonnés qui peuvent tomber en panne
 - ...et ce que l'on **ne peut pas faire**...
 - sous différentes hypothèses (e.g. topologie de communication, synchrone vs. async)

Conclusion

Conclusion

- système temps réel =
prédictibilité, dépendabilité

- pour y arriver, on fait appel à une panoplie de méthodes :
 - *résultats mathématiques*
(e.g., ordonnanceabilité)
 - *modèles* de conception spécifiques
 - *architectures, OS et langages* spécifiques
 - *validation* formelle ou très poussée
 - patronnes de conception supportant la tolérance aux pannes
 - ...