

Disponibilité des services et des ressources

Hervé Grall

Équipe OBASCO (EMN/INRIA – LINA)
École des mines de Nantes

10 mai 2005
Toulouse

Plan

- 1 La démarche
- 2 Modéliser la disponibilité de services
- 3 Modéliser la disponibilité de ressources

Plan

- 1 La démarche
- 2 Modéliser la disponibilité de services
- 3 Modéliser la disponibilité de ressources

Objectifs

- Une modélisation simple du comportement de composants logiciels
 - services, avec protocole d'accès (exemple : requête – réponse)
 - ressources (produites et consommées)
 - abstraction du comportement concret
 - sémantique opérationnelle concrète → traces d'exécution
 - sémantique abstraite → abstraction des traces
- Définir un langage logique pour exprimer des propriétés portant sur le comportement (abstrait) d'un composant
- Model-checking décidable : le comportement (abstrait) du composant vérifie-t-il une propriété logique ?

Une solution

- Système de transitions pour décrire les comportements abstraits possibles
→ Protocole $\stackrel{def}{=}$ ensemble des traces abstraites possibles
- Définition d'un transducteur pour exprimer la variation des ressources lors des transitions
- Logique du second ordre monadique pour exprimer les propriétés de disponibilité

→ Vérification de propriétés au niveau abstrait –
Concrètement ?

Plan

- 1 La démarche
- 2 Modéliser la disponibilité de services**
- 3 Modéliser la disponibilité de ressources

Exemple : un serveur, des clients, protocole requête-réponse

Le serveur : unique instance de la classe `Server`

Chaque client : instance d'une classe implémentant une interface `Client`

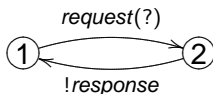
Code d'un client :

```
// appel direct
Service service1 = serverId.getService(this);

// appel en deux temps : requête puis réponse
Request requestId = serverId.request(this);
// -> évènement ``request (!)``
...
Service service2 = serverId.response(requestId);
// -> évènement ``? response``
```

Le protocole du serveur

- Le serveur reçoit des requêtes et y répond



- Deux aspects dans le comportement
 - la communication avec son environnement (les clients et d'autres composants éventuellement)
 - les calculs internes nécessaires pour réaliser les services
- Disponibilité : toute requête reçoit-elle une réponse ?

Des interfaces aux protocoles

- But d'une interface : déclarer les fonctions d'accès aux services
- Point de vue statique et abstrait : spécifier l'appel de la fonction (arguments à fournir, résultat attendu)
- Point de vue dynamique et concret : la sûreté de typage
si l'environnement utilise le composant conformément à son interface, il n'y aura pas d'erreurs (de types) à l'exécution

Des interfaces aux protocoles

Exemple

- enregistrer la demande de service

argument : le client

retourne un identifiant de requête

```
public Request request(Client clientId) ;
```

- offrir le service

argument : identifiant de requête

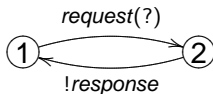
retourne le service

```
public Service response(Request requestId) ;
```

Des interfaces aux protocoles

- Hypothèse : les fonctions sont toujours disponibles
- Problème si la disponibilité des fonctions peut varier dans le temps → un composant réactif

Exemple



Messages liés aux appels de fonctions

? message (!) : envoi d'un argument, réception du résultat

→ exécution de code externe

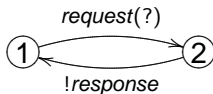
! message (?) : réception d'un argument, envoi du résultat

→ exécution de code interne

Des interfaces aux protocoles

- Hypothèse : les fonctions sont toujours disponibles
- Problème si la disponibilité des fonctions peut varier dans le temps → un composant réactif

Exemple



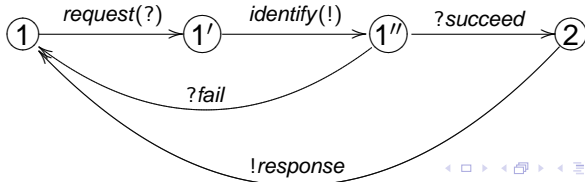
→ Une question de disponibilité des services :
toute requête reçoit-elle une réponse ?
Oui, si le protocole ne s'arrête pas en l'état 2

Une solution : utiliser un protocole

Un protocole modélise

- les communications avec l'environnement : messages correspondant à des événements liés aux fonctions d'accès aux services (appels, retours)
- les calculs internes, en particulier lorsqu'ils impliquent des communications.

Exemple : vérification de l'identité du client → fonction requise :
`boolean identify (Client clientId) ;`



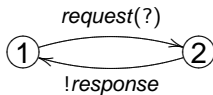
Protocole et abstraction

- Un protocole ne respecte pas la barrière d'abstraction d'une fonction d'accès – exemple : fonction `request` contenant des communications
- On devrait pouvoir définir un protocole à différents niveaux d'abstraction

→ Un protocole dépend des détails d'implémentation connus à un niveau d'abstraction donné

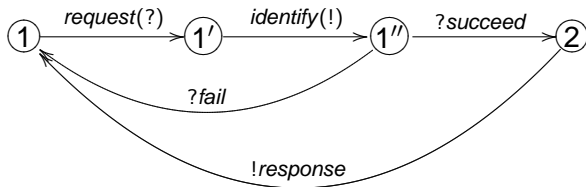
Protocole et abstraction

Protocole abstrait



Protocole et abstraction

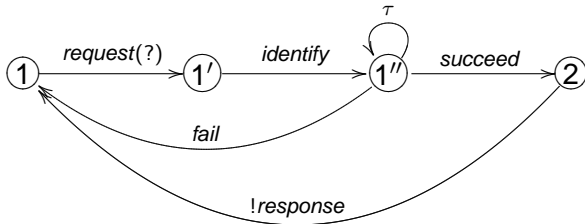
Protocole concret



Protocole et abstraction

Protocole concret : une alternative

→ Calculs internes sans communications : appel d'identify,
calculs τ , résultat fail ou succeed

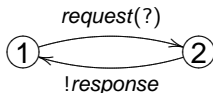


Première question : comment relier les différents niveaux d'abstraction ?

- Étudier le **raffinement** de protocoles
→ Relation de raffinement : un protocole concret raffine un protocole abstrait
- Étudier la relation entre le protocole (vue statique et abstraite du comportement) et l'exécution du composant (vue dynamique et concrète du comportement)
- Étudier la **préservation** de propriétés par **raffinement**

Première question : comment relier les différents niveaux d'abstraction ?

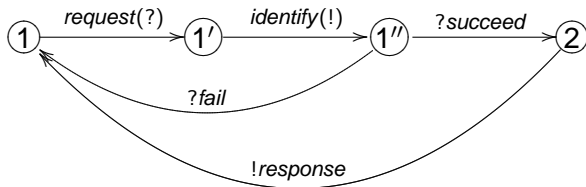
- Étudier la préservation de propriétés par raffinement
Toute requête reçoit-elle une réponse ?



Oui

Première question : comment relier les différents niveaux d'abstraction ?

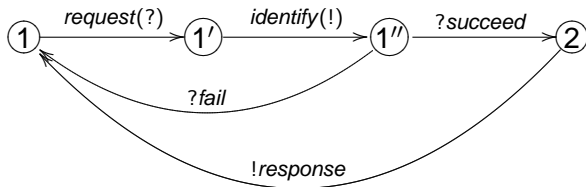
- Étudier la préservation de propriétés par raffinement
Toute requête reçoit-elle une réponse ?



Non.

Première question : comment relier les différents niveaux d'abstraction ?

- Étudier la préservation de propriétés par raffinement
Toute requête qui n'échoue pas reçoit-elle une réponse ?



Oui.

Première question : comment relier les différents niveaux d'abstraction ?

Dans la suite, on se situe à un **niveau donné d'abstraction**, et on oublie les autres niveaux.

Formalisation des protocoles

Automate (Q, I, F, Δ) défini sur un ensemble d'évènements A

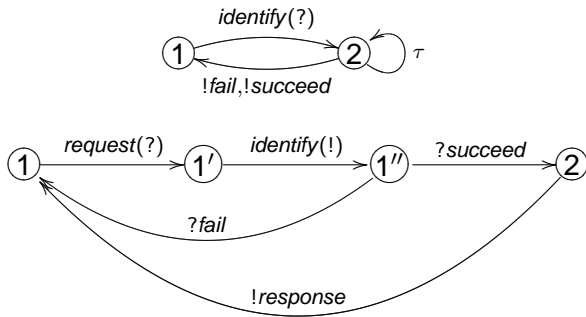
- Deux sortes d'évènement :
 - ① évènement de communication (messages avec émission et réception)
 - ② évènement interne (silencieux)
- Q : ensemble d'états (dénombrable)
- I : états initiaux
- F : états finals
- Δ : ensemble de transitions $\text{état}_1 \xrightarrow{\text{ev.}} \text{état}_2$
 → Trace reconnue par l'automate : suite des évènements associée à une exécution
 état initial ... *transitions* ... état final

→ **Protocole** $\stackrel{\text{def}}{=} \text{ensemble des traces reconnues par l'automate}$



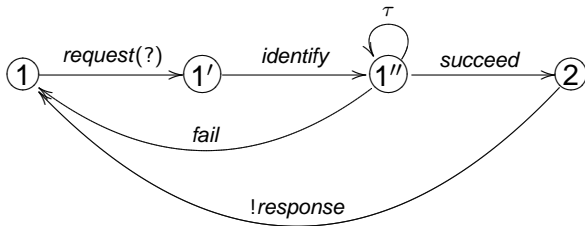
Composition de protocoles

Composition de protocoles : produit synchronisé des automates



Composition de protocoles

Composition de protocoles : produit synchronisé des automates

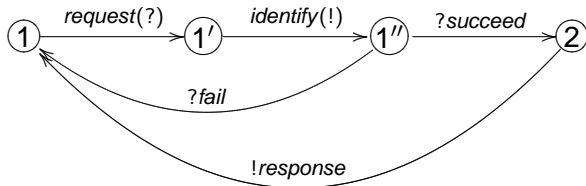


Seconde question : préservation de propriétés par composition

- 1 Toute requête qui n'échoue pas reçoit-elle une réponse ?
- 2 Entre deux requêtes, y a-t-il au plus trois transitions ?

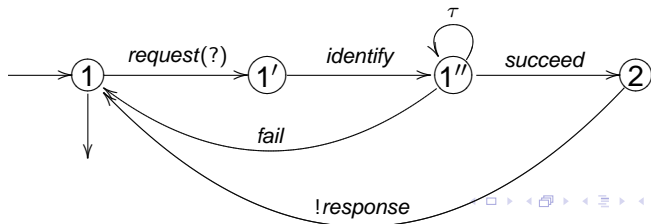
Seconde question : préservation de propriétés par composition

- 1 Toute requête qui n'échoue pas reçoit-elle une réponse ?
Oui
- 2 Entre deux requêtes, y a-t-il au plus trois transitions ?
Oui



Seconde question : préservation de propriétés par composition

- 1 Toute requête qui n'échoue pas reçoit-elle une réponse ?
Non si la boucle τ est indéfiniment parcourue, oui sinon
→ les **états finals** correspondent à la terminaison
- 2 Entre deux requêtes, y a-t-il au plus trois transitions ?
Non



Seconde question : préservation de propriétés par composition

- Étudier la **préservation** des propriétés par **composition**
- Exemple de propriété préservée : toute requête qui n'échoue pas reçoit une réponse
- Exemple de propriété non préservée : entre deux requêtes, il y a au plus trois transitions

États finals et vivacité

- Propriété $\stackrel{def}{=}$ un ensemble de traces (finies)

États finals et vivacité

- Propriété $\stackrel{def}{=}$ un ensemble de traces (finies)
- Propriété de **sûreté** (safety) – S
Pour toute trace n'appartenant pas à S , il existe un préfixe tel que toute trace prolongeant ce préfixe n'appartient pas à S :

$$\forall \beta \notin S. \exists \alpha \leq \beta. \forall \gamma. \alpha\gamma \notin S$$

Ou plus simplement :

L'ensemble S est fermé par préfixe :

$$\forall \beta \in S. \forall \alpha \leq \beta. \alpha \in S$$

→ Rien d'interdit n'est arrivé

États finals et vivacité

- Propriété $\stackrel{def}{=}$ un ensemble de traces (finies)
- Propriété de **vivacité** (liveness) – V
Toute trace peut être prolongée en une trace appartenant à V :

$$\forall \alpha . \exists \beta . \alpha \beta \in V$$

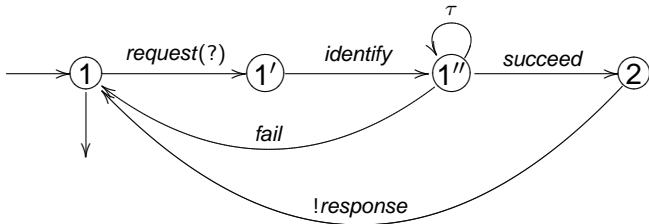
→ Il peut toujours arriver un évènement souhaitable.

États finals et vivacité

- Propriété $\stackrel{def}{=}$ un ensemble de traces (finies)
 - Théorème (Schneider) :
Toute propriété est égale à l'intersection d'une propriété de sûreté et d'une propriété de vivacité
- **Sans les états finals**, on ne définit que des protocoles correspondant à des propriétés de **sûreté**
- **Avec les états finals**, on peut définir **tout** protocole

Exemple

- Trace vide ou trace se terminant par $!response$: vivacité
- Fermeture préfixe du protocole défini par l'automate suivant :



→ Le protocole = intersection des deux propriétés

Démonstration du théorème de Schneider

P propriété quelconque

S : fermeture par préfixe de P , S' : complémentaire de S

$S \stackrel{\text{def}}{=} \{\alpha \mid \exists \gamma. \alpha \gamma \in P\}$, $S' \stackrel{\text{def}}{=} \{\alpha \mid \forall \gamma. \alpha \gamma \notin P\}$

$V \stackrel{\text{def}}{=} S' \cup P$

$V \cap S = P$ car $P \subseteq S$

V : propriété de vivacité ?

α quelconque – Deux possibilités

(i) $\alpha \in S$

→ (définition de S) $\exists \beta. \alpha \beta \in P$

→ ($P \subseteq V$) $\exists \beta. \alpha \beta \in V$

(ii) $\alpha \in S'$

→ ($S' \subseteq V$) $\exists \beta. \alpha \beta \in V$ (prendre $\beta = \varepsilon$)

Troisième question : Exprimer des propriétés de comportement

Critères pour un langage logique

- 1 Le langage exprime des propriétés de traces
- 2 Le problème du model-checking est décidable
 - Entrée : un protocole et une propriété logique
 - Résultat : le protocole vérifie la propriété ou non

Une réponse : la **logique du second ordre monadique**,
interprétée dans le monoïde A^* , formé des traces
Mère de **toutes** les logiques temporelles

Le langage logique

- Termes : ce sont les traces
 - trace vide
 - variable du premier ordre
 - la concaténation d'une trace avec un évènement
- Relations (unaires ou monadiques) : ce sont les ensembles de traces
 - variable du second ordre monadique
 - le protocole du composant noté P
- Formules atomiques
 - égalité entre traces, préfixe
 - appartenance d'une trace à un ensemble de traces
 - inclusion entre ensembles de traces
- Logique propositionnelle
- Logique du premier ordre (quantification sur les traces)
- Logique du second ordre monadique (quantification sur les ensembles de traces)

Model-checking – la décidabilité

- Théorème (Rabin) : la logique du second ordre monadique interprétée dans le monoïde A^* est décidable
La démonstration repose sur la correspondance suivante : une relation entre ensembles de traces est définissable si et seulement si elle est reconnue par un automate fini d'arbres (après codage)
- Corollaire : si le protocole P est défini par un automate fini de traces, alors notre problème de model-checking est **décidable**

Expressivité de la logique

- Entre deux requêtes, il y a au plus trois transitions

Pour toute trace t du protocole P ,
pour tout préfixe $u.request$ de t ,
s'il existe une trace $v.request$ préfixe de t et suffixe strict de
 $u.request$, alors au moins l'une des traces suivantes est un
préfixe de t :

- 1 $u.request.request$,
- 2 $u.request.*.request$,
- 3 $u.request.*.*.request$,
- 4 $u.request.*.*.*.request$

Expressivité de la logique

- Toute requête qui n'échoue pas reçoit une réponse avant qu'une autre requête puisse être émise

Pour toute trace t du protocole P ,
pour tout préfixe $u.request$ de t tel que
 $u.request.identify.\tau^*.fail$ n'est pas un préfixe de
 t ,

il existe un préfixe $v.response$ de t tel que

- 1 $u.request$ est un préfixe de $v.response$
- 2 entre $u.request$ et $v.response$, il n'y a pas d'évènement request

A quoi sert le second ordre ?

- Second ordre \rightarrow définitions inductives

$$\frac{\emptyset}{\text{trace}} \quad \frac{\text{trace}_1 \dots \text{trace}_n}{\text{trace}}$$

\rightarrow L'ensemble engendré inductivement est définissable

- Exemple : l'ensemble régulier $u . \text{request} . \tau^* . \text{fail}$

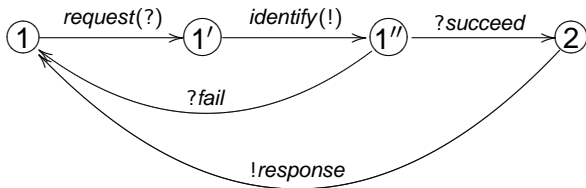
$$\frac{\emptyset}{u . \text{request} . \text{fail}} \quad \frac{v . \text{fail}}{v . \tau . \text{fail}}$$

Quatrième question : peut-on compter ?

- Existe-t-il un entier n tel qu'entre deux requêtes, il y ait au plus n transitions ?

Quatrième question : peut-on compter ?

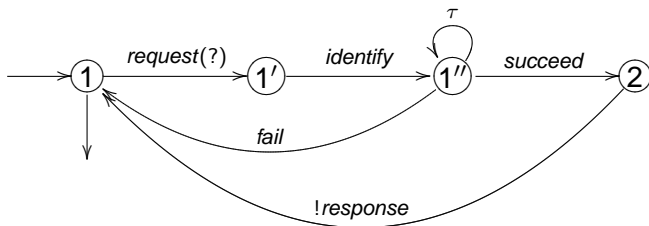
- Existe-t-il un entier n tel qu'entre deux requêtes, il y ait au plus n transitions ?



Oui

Quatrième question : peut-on compter ?

- Existe-t-il un entier n tel qu'entre deux requêtes, il y ait au plus n transitions ?



Non

Quatrième question : peut-on compter ?

- Existe-t-il un entier n tel qu'entre deux requêtes, il y ait au plus n transitions ?
- Impossible d'exprimer une telle propriété avec notre logique : on est obligé de fixer la valeur de n (par exemple $n = 3$)
→ introduire des **compteurs**

Plan

- 1 La démarche
- 2 Modéliser la disponibilité de services
- 3 Modéliser la disponibilité de ressources**

Automates avec ressources et compteurs

- Ensemble fini de **ressources** dénombrables
- Une ressource \rightarrow un **compteur** : le compteur indique la quantité de cette ressource
- L'état des ressources est décrit par un vecteur, indiquant la quantité de chaque ressource
- L'état initial des ressources est donné
- On ajoute au protocole un **transducteur** pour représenter la consommation et la production de ressources
 \rightarrow chaque transition du protocole est traduite par une fonction, qui transforme l'état des ressources

Automates avec ressources et compteurs

Automate avec compteurs (Q, I, F, Δ)

- Q : ensemble d'états (dénombrable)
- I : configurations initiales
 $\stackrel{def}{=} \{(\text{état initial de l'automate}, \text{vecteur initial})\}$
- F : états finals $\stackrel{def}{=} \{\text{état final de l'automate}\}$
- Δ : ensemble de transitions

$$\text{état}_1 \xrightarrow{\text{ev., tranfo.}} \text{état}_2$$

→ Exécution de l'automate :

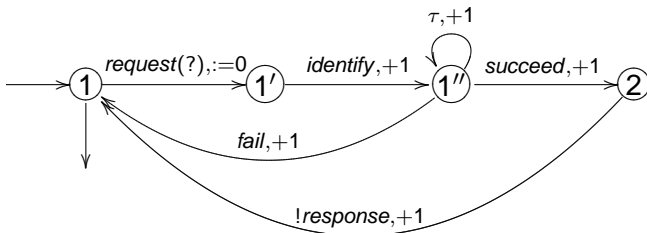
$$(\text{état initial}, \text{vecteur initial}) \dots (\text{état}_1, \text{vecteur}_1) \xrightarrow{\text{ev., tranfo.}} \\ (\text{état}_2, \text{transfo}(\text{vecteur}_1)) \dots (\text{état final}, \text{vecteur final})$$

Exemple : compter les transitions

Traduction utilisant

- une initialisation : mise à zéro $:= 0$
- une production de ressources :
incrément $+1$

État initial : compteur à zéro



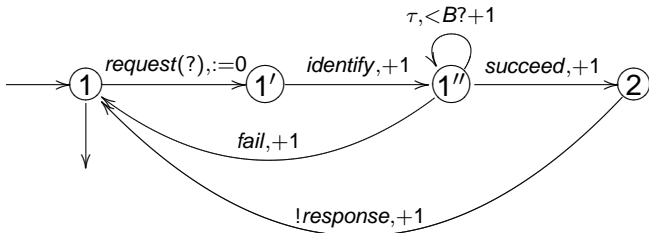
État 1 : 0, 2, 3, 4, ...

Exemple : compter les transitions

Traduction utilisant

- une initialisation : mise à zéro $:= 0$
- une production de ressources :
incrément conditionnelle $< B? + 1$

État initial : compteur à zéro



État 1 : $0, 2, \dots, B + 1, 3, \dots, B + 2$ ($B \geq 1$)

Propriétés d'atteignabilité

- L'ensemble des configurations (état, vecteur) est **infini**
→ Représentation finie d'ensembles particuliers de configurations
- Déterminer une représentation finie de
 - l'ensemble des configurations atteignables
 - l'ensemble des configurations atteignables en un état donné
- Déterminer l'existence d'un majorant pour les compteurs
→ S'il existe, possibilité du calcul exact par itération de l'ensemble fini des configurations

Quelques résultats de décidabilité

Automates avec un compteur

- Configurations initiales et transformations exprimées en logique du second ordre monadique interprétée sur \mathbb{N}
- Configurations initiales et transformations exprimées dans l'arithmétique de Presburger (premier ordre avec addition et ordre)

→ Généralisation des automates avec boîtes aux lettres (Royer)

Quelques résultats de décidabilité

Automates avec plusieurs compteurs (couplés)

- Configurations initiales : ensembles périodiques, transformations : $a_1x_1 + \dots + a_nx_n < k? + i$ ou $:= c$ (Boigelot, Wolper, Finkel (outil FAST))
- Configurations initiales et transformations : conjonctions de $x_i \text{ op } y_j + c$ ou $x_i \text{ op } c$, avec x_i, y_j valeurs des compteurs i et j avant ou après la transition et $\text{op} \stackrel{\text{def}}{=} < | =$, automates plats (Comon, Jurski)
- Résultats de décidabilité pour automates à deux compteurs, avec mise à zéro, in(dé)-crémentations et transfert (Finkel, Sutre)

→ Pas de méthode générale unifiant l'ensemble de ces travaux

Conclusion

Les questions

- 1 Préservation de propriétés par raffinement
- 2 Préservation de propriétés par composition
- 3 Disponibilité de services : expressivité et décidabilité
→ Logique du second ordre monadique
- 4 Disponibilité de ressources : expressivité et décidabilité
→ Atteignabilité pour des automates avec compteurs
- 5 Concrètement ? Le rapport entre code et protocole

Conclusion

Perspectives pour la vérification de propriétés (de disponibilité)

- Étude de la préservation
- Développement du cadre formel pour exprimer les propriétés de disponibilité et des algorithmes de décision
- Implémentation des algorithmes, utilisation d'outils

Conclusion

Rapport entre code et protocole

Le problème

- données : protocole vérifié + code (clients et serveur)
- résultat : un programme dont l'exécution est contrôlée par l'automate protocolaire

Différentes solutions

- Avec des composants (contrôleurs enveloppant le code et réalisant les communications)
- Avec le pattern Proxy
- Avec des aspects (événements déclenchant des transitions dans l'automate)