

CONCOURS D'ADMISSION 2026
Composition d'Informatique C, Filière MPI

1 Le sujet

Le sujet portait sur la vérification de systèmes concurrents. Il proposait une progression allant de notions générales sur les systèmes de transitions jusqu'à des techniques classiques de réduction de l'espace des exécutions dans les systèmes concurrents.

La partie I introduisait les systèmes de transitions, les propriétés de traces, les invariants et les propriétés de sûreté. Elle demandait de manipuler des notions à la fois mathématiques et algorithmiques : automates finis, préfixes interdits, régularité de langages, construction produit avec un automate et accessibilité dans un système de transitions. Elle comportait également une question de programmation en OCaml.

La partie II était consacrée à la propriété d'ultime invariance $EA\Phi$. Après un exemple permettant de fixer les idées, elle amenait les candidates et candidats à caractériser la non-satisfaction de cette propriété par l'existence d'un cycle accessible hors de Φ . La suite de la partie étudiait un algorithme de parcours en profondeur imbriqués, avec une attention particulière portée à l'ordre postfixe du parcours externe et à la preuve de correction de l'algorithme.

La partie III portait sur la programmation multi-threads en C. Elle rappelait l'utilisation de la bibliothèque pthread et demandait d'analyser un petit programme concurrent, puis d'en écrire une généralisation. Cette partie visait notamment à faire apparaître les difficultés liées aux accès concurrents à une variable globale partagée.

La partie IV revenait aux systèmes de transitions, cette fois pour modéliser et analyser des systèmes concurrents. Elle introduisait les notions d'indépendance d'actions, d'ensembles persistants, d'ordre induit par un mot et de mise en veille. Les dernières questions reliaient ces notions à la sélection de représentants lexicographiquement minimaux parmi les exécutions équivalentes.

Le sujet était long et exigeant. Les premières questions de chaque partie permettaient d'entrer dans les définitions, tandis que plusieurs questions de preuve ou de programmation demandaient une compréhension fine des objets introduits.

2 Remarques générales

Dans l'ensemble, les premières questions de chacune des quatre parties ont joué leur rôle d'entrée dans le problème : après des questions difficiles ou peu abordées, on observe à plusieurs reprises une

forte remontée du nombre de copies traitant le début de la partie suivante. Les questions de programmation ont été sensiblement mieux réussies que les questions de preuve les plus techniques. Les questions 9 et 11 ont constitué les principaux obstacles de la partie II, tandis que la fin de la partie IV n'a été atteinte que par un très petit nombre de candidats. Le sujet a ainsi permis une forte différenciation tout en laissant plusieurs possibilités de reprise au cours de l'épreuve.

Il est rappelé que la lisibilité de la copie est essentielle. Cela vaut pour les preuves, mais aussi pour le code : un programme doit être suffisamment aéré, commenté lorsque c'est utile, et les fonctions auxiliaires doivent être compréhensibles. Une réponse illisible, même partiellement correcte, ne permet pas aux correctrices et correcteurs d'en apprécier la valeur.

De nombreuses erreurs provenaient d'une lecture trop rapide des définitions. Il fallait distinguer soigneusement un système de transitions d'un automate fini, un état terminal d'un état final, une action d'une arête du graphe, ou encore un ensemble d'actions de l'ensemble des transitions effectivement présentes dans le système. Ces distinctions étaient particulièrement importantes dans les questions de complexité, de régularité et de concurrence.

Les questions de preuve de correction de programmes sont restées trop souvent descriptives. Pour prouver la correction d'un algorithme, il ne suffit pas de raconter le rôle général des fonctions : il faut dire ce que l'on suppose avant l'appel, ce qui est garanti après l'appel, quelles exceptions peuvent être levées, et pourquoi les appels successifs respectent bien ces conditions. Les termes de précondition, postcondition ou invariant n'étaient pas exigés en tant que tels, mais les idées correspondantes devaient apparaître.

Lorsque l'énoncé demandait explicitement une implémentation, notamment aux questions 5, 10 et 14, il était indispensable de respecter le langage demandé. Un pseudo-code très éloigné d'OCaml ou de C ne pouvait alors être considéré comme une implémentation satisfaisante. Lorsqu'une description de plus haut niveau était explicitement autorisée, comme à la question 13, celle-ci était bien entendu recevable. En OCaml, la gestion des sommets déjà visités était souvent le point déterminant pour assurer la terminaison. En C, il fallait être attentif aux allocations, aux indices passés aux threads, aux appels à `pthread_join` et aux libérations de mémoire.

Enfin, les questions de concurrence demandaient de raisonner localement. Deux actions ne sont pas indépendantes simplement parce qu'elles appartiennent à deux threads différents : il faut examiner ce qu'elles lisent, ce qu'elles écrivent et l'état dans lequel on se trouve. Cette dépendance au contexte a été une source fréquente d'erreurs dans les dernières parties.

3 Commentaires détaillés

Pour chaque question, sont indiqués entre crochets le pourcentage de copies ayant traité la question et le pourcentage de copies ayant obtenu la totalité des points.

Question 1. [99%, 46%] Cette première question demandait de relier la notion d'invariant à la satisfaction de la propriété I^ω , sous l'hypothèse qu'il n'y a pas d'état terminal. Le sens direct devait utiliser une récurrence sur les états successifs d'une exécution. Beaucoup de copies ont eu l'idée, mais l'ont parfois formulée de manière trop informelle. Le sens réciproque reposait sur l'hypothèse d'absence d'état

terminal : après une transition sortant de I , il faut pouvoir prolonger l'exécution en une exécution infinie afin d'utiliser la satisfaction de I^ω . Les raisonnements par l'absurde ou par contraposée étaient possibles, à condition de faire apparaître explicitement ce prolongement.

Question 2. [100%, 58%] Il fallait déterminer lesquelles des propriétés proposées étaient des propriétés de sûreté. Une réponse réduite à une liste, sans justification, était insuffisante : même lorsque le résultat est simple, la définition de sûreté demande d'exhiber un préfixe interdit ou, au contraire, de montrer qu'aucun préfixe fini ne peut empêcher un prolongement satisfaisant la propriété. Les erreurs les plus fréquentes portaient sur les quantificateurs, notamment sur la différence entre « il existe un état apparaissant une infinité de fois » et « tous les états apparaissent une infinité de fois ».

Question 3. [96%, 8%] Cette question combinait propriétés de sûreté et langages réguliers. Les propriétés régulières devaient être accompagnées d'un automate ou, au minimum, d'une description suffisamment précise d'un automate reconnaissant les préfixes interdits minimaux. Pour P_3 , il fallait au contraire démontrer la non-régularité. Les meilleures solutions utilisaient le lemme de l'étoile, un argument de type principe des tiroirs sur les états d'un automate, ou le théorème de Myhill–Nerode. En revanche, exhiber un langage non régulier strictement inclus dans l'ensemble étudié ne suffit pas à conclure. De même, l'utilisation d'une intersection avec un langage régulier devait être faite dans le bon sens et justifiée précisément.

Question 4. [79%, 9%] Le sens direct était relativement court : si les préfixes interdits minimaux forment un langage régulier, alors l'ensemble de tous les préfixes interdits s'obtient en les prolongeant par un mot quelconque. Le sens réciproque demandait plus de soin. Il fallait partir d'un automate reconnaissant tous les préfixes interdits, le rendre ou le supposer déterministe, puis le modifier pour accepter exactement les mots qui deviennent interdits pour la première fois. Plusieurs copies ont donné une formule ensembliste correcte ou presque correcte, mais sans justification suffisante.

Question 5. [94%, 22%] La fonction reachable appelait un parcours de graphe classique. Une bonne solution effectuait un parcours en profondeur ou en largeur à partir de l'état initial, testait la cible et mémorisait les états déjà visités afin d'assurer la terminaison. L'oubli du tableau de marquage conduisait à des boucles infinies sur les systèmes contenant des cycles. La complexité devait être exprimée en fonction du nombre d'états et du coût du parcours des listes de transitions. Il fallait éviter l'ambiguïté entre $|A|$, nombre d'actions, et le nombre d'arêtes du graphe de transitions.

Question 6. [36%, 4%] La construction attendue était un produit entre le système de transitions et l'automate reconnaissant les préfixes interdits minimaux. La réponse devait définir complètement le nouvel ensemble d'états, les actions disponibles, la fonction de transition, l'application d'initialisation et l'ensemble cible. Il fallait ensuite prouver les deux implications de l'équivalence : une trace atteignant un état cible fournit un préfixe interdit, et une exécution violant la propriété de sûreté donne accès à un tel état cible. Certaines réponses ont décrit l'idée générale sans définir précisément le système construit, ce qui ne suffisait pas.

Question 7. [97%, 37%] Cette question d'exemple a été globalement abordée, mais les justifications ont souvent été trop rapides. Pour la propriété Φ^ω , seuls les états dont toutes les exécutions restent dans Φ conviennent. Pour $EA\Phi$, il faut raisonner sur le comportement à long terme de toutes les exécutions.

L'état s_2 a été fréquemment oublié : malgré une transition possible vers un état hors de Φ , toutes les exécutions finissent bien par rester dans Φ .

Question 8. [96%, 53%] Le sens direct demandait d'extraire, d'une exécution infinie ayant une infinité d'occurrences hors de Φ , un état hors de Φ apparaissant au moins deux fois, donc appartenant à un cycle. L'argument repose sur la finitude de l'ensemble des états. Il ne suffisait pas d'affirmer qu'on peut toujours poursuivre vers un état ne satisfaisant pas la propriété. La réciproque était immédiate : un cycle accessible hors de Φ produit une exécution infinie qui ne satisfait pas $EA\Phi$.

Question 9. [69%, 1%] Cette question était technique et spécifique à l'algorithme de parcours imbriqués. Il fallait exploiter les deux propriétés de l'énumération issue du parcours en profondeur : l'ordre postfixe et l'ordre entre sous-arbres frères. De nombreuses copies ont traité correctement le cas où les deux états se trouvent dans des sous-arbres imbriqués, mais ont omis le cas des sous-arbres disjoints. Or ce dernier cas était précisément celui qui nécessitait l'utilisation fine de la seconde condition.

Question 10. [67%, 40%] L'implémentation de `outer_iter` devait produire un parcours en profondeur en ordre postfixe. Concrètement, il fallait marquer un état lors de sa première visite, parcourir récursivement ses successeurs, puis appeler la fonction `f` sur cet état. Les erreurs courantes étaient l'absence de récursion, l'absence ou la mauvaise portée du tableau de marquage, l'utilisation d'un ordre préfixe, l'oubli de l'appel à `f` sur l'état initial, ou encore l'oubli du fait que `t.next` renvoie une liste de paires dont il faut extraire le second composant.

Question 11. [45%, 0%] Cette question de spécification et de correction de `cycle_check` a été l'une des plus difficiles. Beaucoup de copies ont expliqué informellement ce que font `cycle_check` et `verify`, mais peu ont réellement démontré leur correction. Il fallait préciser l'état du tableau `marked` avant l'appel, expliquer qu'aucun des mauvais états déjà traités ne fait de cycle sur lui-même, caractériser exactement les cas où l'exception `Found` est levée, puis dire ce qui est marqué après un appel qui termine normalement. La correction globale de `verify` reposait alors sur l'enchaînement de ces préconditions et postconditions, et sur l'utilisation de la question 9 pour justifier que le partage du tableau `marked` entre les appels ne masque pas un cycle à détecter.

Question 12. [19%, 14%] La réponse était négative : remplacer l'ordre postfixe par un ordre préfixe rend l'algorithme incorrect. L'idée essentielle est qu'un premier appel à `cycle_check` peut marquer des états qui auraient dû être examinés plus tard comme points de départ de cycles. Un contre-exemple à deux états suffisait.

Question 13. [83%, 39%] L'analyse du programme C devait faire apparaître les deux affichages possibles, 20 et 40, et expliquer par un entrelacement d'instructions comment la valeur incorrecte peut être obtenue. La correction canonique consistait à protéger par un verrou l'accès critique à la variable globale partagée `m`, en verrouillant avant la lecture de `m` et en relâchant après son écriture.

Question 14. [75%, 34%] La fonction `main` demandée était une question de programmation C assez directe, mais elle imposait de respecter tous les détails de l'énoncé : allocation dynamique du tableau d'entiers, initialisation de chaque case par l'expression `rand()%max_item`, allocation ou déclaration des descripteurs de threads, lancement de chaque thread sur le bon sous-tableau, attente de tous les threads,

affichage du résultat au bon moment et libération de la mémoire allouée. Les erreurs d'indice dans l'adresse transmise à chaque thread ont été particulièrement pénalisantes.

Question 15. [66%, 19%] Il s'agissait de compter les entrelacements complets de deux threads comportant chacun quatre actions. Donner seulement une expression combinatoire sans sa valeur, ou donner la bonne valeur sans expliquer ce qui est compté, ne constituait pas une réponse complète.

Question 16. [65%, 12%] L'indépendance des actions devait être examinée état par état. Dans les premiers états considérés, les actions disponibles de chaque thread ne modifient pas l'information lue par l'autre et commutent donc. En revanche, dans le dernier état demandé, une action écrit dans la variable globale m alors que l'autre en lit la valeur : l'ordre d'exécution peut changer le résultat, et les actions ne sont donc pas indépendantes. Les réponses fondées uniquement sur l'idée que les actions appartiennent à deux threads différents étaient insuffisantes.

Question 17. [54%, 26%] La preuve de persistance demandait une récurrence simple sur les états successifs de l'exécution. Au départ, l'action considérée est disponible parce qu'elle appartient à l'ensemble persistant inclus dans les actions disponibles. À chaque étape, la condition (A2) donne l'indépendance avec l'action exécutée, ce qui permet de conclure que sa disponibilité est préservée. L'utilisation explicite de la définition de l'indépendance était attendue.

Question 18. [40%, 3%] Pour l'état initial du programme, un singleton bien choisi forme un ensemble persistant minimal, car l'action correspondante est indépendante des actions de l'autre thread tant que celles-ci sont exécutées avant elle. En revanche, dans l'état où les deux threads s'apprêtent à lire puis mettre à jour la variable globale, il fallait prendre les deux actions disponibles. Une réponse donnant seulement les ensembles, sans justifier la minimalité ni expliquer pourquoi un singleton ne suffit pas dans le second cas, restait incomplète.

Question 19. [25%, 3%] Cette question établissait la propriété fondamentale des ensembles persistants. Le premier point était de remarquer que toute exécution vers un état terminal doit finir par exécuter une action de l'ensemble persistant du point de départ ; sinon les actions de cet ensemble resteraient disponibles jusqu'à l'état terminal, contradiction. Il fallait ensuite déplacer une telle action vers le début de l'exécution en utilisant les commutations permises par l'indépendance, puis conclure par induction sur la longueur restante. Les preuves par récurrence pouvaient convenir, à condition de faire apparaître clairement cet argument de commutation.

Question 20. [21%, 7%] Cette question vérifiait la compréhension de la définition de l'ordre induit. Il fallait lister les contraintes imposées par les paires d'actions dépendantes apparaissant dans le mot, puis prendre la clôture transitive. Plusieurs réponses correctes ont été obtenues, mais il fallait faire attention à ne pas ajouter de comparabilité entre deux actions indépendantes et à distinguer deux actions incomparables de deux actions simplement non encore ordonnées explicitement.

Question 21. [10%, 0%] Le sens direct reposait sur le fait que permuter deux actions indépendantes adjacentes ne change pas l'ordre induit. Pour la réciproque, l'argument naturel était une récurrence sur la longueur du mot : on choisit une action minimale pour l'ordre commun, on la fait remonter au début du mot par permutations d'actions indépendantes, puis on applique l'hypothèse de récurrence au suffixe.

Une affirmation globale du type « deux mots qui donnent le même ordre partiel sont équivalents » devait être justifiée.

Question 22. [14%, 0%] Il fallait faire figurer dans un dessin les états correspondant aux positions des deux threads et aux valeurs déjà définies des variables locales et globale, ainsi que les actions étiquetant les transitions. Dans la seconde partie, le passage au système avec mise en veille ne modifiait que très légèrement toute la structure.

Question 23. [1%, 0%] La caractérisation des minima lexicographiques demandait les deux sens. Si un mot contient le motif interdit décrit dans l'énoncé, on peut déplacer l'action plus petite vers la gauche et obtenir un mot équivalent strictement plus petit. Réciproquement, si le mot n'est pas minimal, il faut considérer la première position où il diffère d'un mot équivalent plus petit et utiliser l'égalité des ordres induits pour montrer que l'action plus petite pouvait franchir le bloc intermédiaire. Le point délicat était précisément la condition d'indépendance avec tout ce bloc intermédiaire.

Question 24. [1%, 0%] La dernière question liait la mise en veille aux minima lexicographiques. Pour le premier point, il fallait montrer qu'une exécution dans le système avec mise en veille ne peut pas contenir le motif interdit de la question précédente : une action plus petite, indépendante de ce qui la sépare de sa première occasion d'être exécutée, aurait été placée en veille et ne pourrait plus être exécutée ensuite. Pour le second point, il fallait montrer qu'une exécution minimale peut toujours être rejouée dans le système avec mise en veille, car les actions nécessaires ne sont jamais bloquées. Très peu de copies sont arrivées jusque-là, mais les réponses les plus avancées ont bien mis en évidence le lien entre la construction algorithmique et la caractérisation de la question 23.