### D.1 Propriétés de Sûreté Structurelle

| Propriété | P1 |
|---|---|
| Formule | A[] not deadlock |
| Fonction | Vérifie l'absence de blocage global (deadlock) dans le système composé des automates maître et esclave. Cette propriété assure qu'aucune configuration du système ne conduit à un état bloqué dans lequel aucune transition ne peut être exécutée, garantissant ainsi la progression continue et effective du protocole. |
| Utilité FSM→C | Essentielle pour l'implémentation C : assure qu'il n'existe aucune combinaison d'états où le code pourrait se bloquer indéfiniment. Guide la conception des boucles d'événements et garantit qu'au moins une transition sortante est toujours définie dans chaque état. Permet d'identifier les états nécessitant un timeout ou un watchdog pour éviter les blocages en cas de non-réception de messages attendus. |

| Propriété | P2 |
|---|---|
| Formule | A[] ((MASTER_SDLC.slave_state_in_m == NRM and MASTER_BITBUS.irrec_err == 0) imply not deadlock) |
| Fonction | Vérifie l'absence de deadlock spécifiquement en mode nominal (NRM) lorsque la connexion SDLC est établie et qu'il n'y a pas d'erreur irrécupérable. Cette propriété assure que le fonctionnement normal du protocole ne conduit jamais à un blocage. |
| Utilité FSM→C | Cruciale pour le code nominal : confirme que les chemins d'exécution principaux (sans erreurs SDLC) sont toujours non-bloquants. Permet d'optimiser le code de production en séparant clairement la logique nominale (garantie sans deadlock) de la gestion d'erreurs. Justifie l'absence de certains timeouts dans les états nominaux, réduisant la complexité du code. |

### D.2 Intégrité des Données et Validité Protocolaire

| Propriété | P3 |
|---|---|
| Formule | MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s == cs_ref_s --> MASTER_BITBUS.PROCESS_BITBUS |
| Fonction | Vérifie que lorsque le maître reçoit une trame alphanumérique de l'esclave avec un checksum valide (cs_frame_s == cs_ref_s), il passe systématiquement à l'état de traitement PROCESS_BITBUS. Garantit le traitement correct des trames valides. |
| Utilité FSM→C | Dicte l'implémentation de la validation de checksum : la fonction C de vérification du checksum doit être appelée immédiatement après réception, et si valide, déclencher une transition inconditionnelle vers l'état PROCESS_BITBUS. Aucune vérification intermédiaire supplémentaire n'est nécessaire. Structure le code sous forme : if (checksum_valid(frame)) { state = PROCESS_BITBUS; }. |

| Propriété | P4 |
|---|---|
| Formule | E<> MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s == cs_ref_s |
| Fonction | Vérifie l'atteignabilité de l'état où le maître reçoit une trame alphanumérique valide de l'esclave. Confirme qu'il existe au moins un chemin d'exécution menant à la réception d'une trame valide, démontrant que le scénario nominal est possible. |
| Utilité FSM→C | Valide la testabilité du code : garantit qu'on peut créer un scénario de test unitaire où une trame valide est reçue. Sert de base pour les tests d'intégration et les cas de test nominaux. Indique que ce chemin d'exécution doit être couvert par les tests de couverture de code. |

| Propriété | P5 |
|---|---|
| Formule | (MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s != cs_ref_s) |

| | |
|---|---|
| | --> MASTER_BITBUS.SDLC_CONNECTION_REQUEST |
| Fonction | Vérifie que lorsque le maître reçoit une trame alphanumérique avec un checksum invalide (cs_frame_s != cs_ref_s), il passe systématiquement à l'état SDLC_CONNECTION_REQUEST. Garantit la détection et le traitement approprié des trames corrompues côté maître. |
| Utilité FSM→C | Détermine la gestion d'erreur de checksum : en cas de checksum invalide, l'implémentation C doit immédiatement effectuer un reset vers SDLC_CONNECTION_REQUEST sans tenter de récupération locale. Simplifie le code en évitant des mécanismes de retry à ce niveau. Structure : if (!checksum_valid(frame)) { state = SDLC_CONNECTION_REQUEST}; |

| Propriété | P6 |
|---|---|
| Formule | E<> (MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s != cs_ref_s) |
| Fonction | Vérifie l'atteignabilité de l'état où le maître reçoit une trame alphanumérique avec un checksum invalide. Confirme que le scénario d'erreur de transmission est représenté dans le modèle. |
| Utilité FSM→C | Guide l'implémentation des tests de robustesse : indique qu'il faut créer des tests avec injection de trames corrompues. Confirme que les chemins d'erreur sont atteignables et doivent être testés. Justifie l'inclusion de tests de corruption de données dans la suite de validation. |

| Propriété | P7 |
|---|---|
| Formule | (SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m == cs_ref_m and SLAVE_BITBUS.y == T_rep) --> SLAVE_BITBUS.SEND_ALPHANUM_S |

| | |
|---|---|
| **Fonction** | **Vérifie que lorsque l'esclave reçoit une trame de données valide du maître (checksum correct) et que le délai de réponse T_rep est atteint, il passe systématiquement à l'état d'envoi de réponse alphanumérique. Garantit le respect du timing de réponse pour les trames valides.** |
| **Utilité FSM→C** | **Impose une synchronisation temporelle stricte : l'implémentation C doit utiliser un timer (y) qui atteint exactement T_rep avant d'envoyer la réponse. Structure : if (checksum_valid && timer_y >= T_rep) { state = SEND_ALPHANUM_S; }. Évite les réponses prématurées et garantit un timing déterministe conforme au protocole industriel.** |

| **Propriété** | **P8** |
|---|---|
| **Formule** | **E<> (SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m == cs_ref_m and SLAVE_BITBUS.y == T_rep)** |
| **Fonction** | **Vérifie l'atteignabilité de l'état où l'esclave reçoit une trame de données valide et atteint le délai de réponse. Confirme que le scénario de communication nominal esclave→maître est possible.** |
| **Utilité FSM→C** | **Valide la testabilité du chemin nominal esclave : confirme qu'on peut simuler un timing correct. Guide l'implémentation de tests temporels avec un timer pout T_rep. Indique que ce scénario doit être couvert dans les tests d'intégration temporelle.** |

| **Propriété** | **P9** |
|---|---|
| **Formule** | **(SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m != cs_ref_m and SLAVE_BITBUS.y == T_rep) --> SLAVE_BITBUS.ERROR_IN_S** |
| **Fonction** | **Vérifie que lorsque l'esclave reçoit une trame de données avec un checksum invalide et atteint le délai de réponse, il passe** |

| | |
|---|---|
| | systématiquement à l'état d'erreur ERROR_IN_S. Garantit la détection des erreurs de transmission côté esclave. |
| Utilité FSM→C | Spécifie le comportement d'erreur avec timing : même si le checksum est invalide, l'esclave attend T_rep avant de signaler l'erreur. Évite les transitions trop rapides qui pourraient perturber le protocole. Code : if (!checksum_valid && timer_y >= T_rep) { state = ERROR_IN_S; log_error(); }. |

| Propriété | P10 |
|---|---|
| Formule | E<> (SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m != cs_ref_m and SLAVE_BITBUS.y == T_rep) |
| Fonction | Vérifie l'atteignabilité de l'état où l'esclave reçoit une trame de données invalide après le délai de réponse. Confirme que le scénario d'erreur est modélisé côté esclave. |
| Utilité FSM→C | Guide les tests d'erreur avec timing : confirme qu'on peut tester le comportement en cas de trame corrompue après attente du délai. Justifie les tests combinant corruption de données et respect de contraintes temporelles. |

| Propriété | P11 |
|---|---|
| Formule | (SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m == cs_ref_m) --> SLAVE_BITBUS.SEND_DATA_RESPONSE and invalid_data_m == 1) |
| Fonction | Vérifie que lorsque l'esclave reçoit une trame invalide (format incorrect) mais avec un checksum valide, il envoie comme même une |

| | |
|---|---|
| | **réponse de données, en gardant l'information de l'invalidité de la trame précédente. Garantit la signalisation des erreurs de format.** |
| **Utilité FSM→C** | **Distingue deux niveaux de validation : Le code C doit implémenter deux validations séquentielles. Si format invalide mais checksum OK, l'esclave répond avec conservation de l'état invalide de la trame reçue par le maitre silencieusement : if (checksum_valid && invalid_data_m) { response.invalid_data = 1; state = SEND_DATA_RESPONSE; }.** |

| **Propriété** | **P12** |
|---|---|
| **Formule** | **E<> (SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m == cs_ref_m)** |
| **Fonction** | **Vérifie l'atteignabilité de l'état où l'esclave reçoit une trame avec un format invalide mais un checksum correct. Confirme que ce scénario d'erreur protocolaire est représenté.** |
| **Utilité FSM→C** | **Guide les tests de validation protocolaire : indique qu'il faut tester avec des trames syntaxiquement invalides mais avec checksum correct. Valide la couverture des erreurs de protocole de haut niveau.** |

| **Propriété** | **P13** |
|---|---|
| **Formule** | **(SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m != cs_ref_m) --> SLAVE_BITBUS.ERROR_IN_S** |
| **Fonction** | **Vérifie que lorsque l'esclave reçoit une trame à la fois invalide en format ET avec un checksum incorrect, il passe systématiquement à** |

| | |
|---|---|
| | **l'état d'erreur. Garantit la gestion stricte des erreurs multiples.** |
| **Utilité FSM→C** | **Spécifie le traitement des erreurs multiples : en présence d'erreurs combinées, transition directe vers ERROR_IN_S. Simplifie le code en évitant de traiter séparément les erreurs multiples. Code : if (!checksum_valid \|\| !format_valid) { state = ERROR_IN_S; } avec priorité au checksum.** |

| **Propriété** | **P14** |
|---|---|
| **Formule** | **E<> (SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m != cs_ref_m)** |
| **Fonction** | **Vérifie l'atteignabilité de l'état où l'esclave reçoit une trame avec erreurs multiples (format et checksum). Confirme la représentation complète des scénarios d'erreur.** |
| **Utilité FSM→C** | **Justifie les tests de pire cas : valide qu'on peut tester avec des trames présentant plusieurs types d'erreurs simultanées. Guide l'implémentation de tests de robustesse extrême.** |

| **Propriété** | **P15** |
|---|---|
| **Formule** | **(MASTER_BITBUS.RECEIVE_VALID_FRM and cs_frame_s == cs_ref_s) --> MASTER_BITBUS.PROCESS_BITBUS** |

| | |
|---|---|
| **Fonction** | **Vérifie que lorsque le maître reçoit une trame valide (format correct) avec un checksum correct, il passe systématiquement à l'état de traitement. Garantit le traitement approprié des trames conformes au protocole.** |
| **Utilité FSM→C** | **Confirme la logique de validation complète côté maître : checksum ET format doivent être validés avant traitement. Structure le code de réception avec validation en deux étapes. Code : if (checksum_valid(frame) && format_valid(frame)) { state = PROCESS_BITBUS; } sinon gestion d'erreur.** |

| **Propriété** | **P16** |
|---|---|
| **Formule** | **E<> (MASTER_BITBUS.RECEIVE_DATA_FRAME and cs_frame_s == cs_ref_s)** |
| **Fonction** | **Vérifie l'atteignabilité de l'état où le maître reçoit une trame de données valide avec checksum correct. Confirme que la communication esclave→maître peut s'effectuer correctement.** |
| **Utilité FSM→C** | **Valide la bidirectionnalité du code : confirme que le chemin de réception maître est fonctionnel. Guide l'implémentation de tests de communication bidirectionnelle.** |

### D.3 Contraintes Temporelles

| **Propriété** | **P17** |
|---|---|

| | |
|---|---|
| **Formule** | **A[] (MASTER_BITBUS.SEND_PROCESS imply MASTER_BITBUS.x == T_Pol)** |
| **Fonction** | **Vérifie que chaque fois que le maître atteint l'état SEND_PROCESS, son horloge x est exactement égale à T_Pol (période de polling). Garantit le respect strict du timing cyclique de scrutation défini par le protocole industriel.** |
| **Utilité FSM→C** | **Impose un timing périodique strict : le code C doit implémenter un timer cyclique précis (x). L'envoi ne peut se faire qu'à x == T_Pol exactement. Structure : if(timer_x >= T_Pol); state = SEND_PROCESS;;. Justifie l'utilisation de timers haute précision pour garantir la périodicité.** |

| **Propriété** | **P18** |
|---|---|
| **Formule** | **(MASTER_BITBUS.WAIT_BITBUS_RESP and MASTER_BITBUS.x == MASTER_BITBUS.T_out_M) --> (MASTER_BITBUS.PROCESS_BITBUS or MASTER_BITBUS.SDLC_CONNECTION_REQUEST)** |
| **Fonction** | **Vérifie que lorsque le maître attend une réponse Bitbus et que son timeout T_out_M expire, il transite soit vers le traitement (si réponse reçue à temps), soit vers une nouvelle demande de connexion SDLC. Garantit qu'aucun timeout ne laisse le système dans un état d'attente indéfinie.** |
| **Utilité FSM→C** | **Spécifie la gestion du timeout maître : à l'expiration de T_out_M, deux transitions possibles selon si une réponse a été reçue. Code : if (timer_x >= T_out_M) { if (response_received) state = PROCESS_BITBUS; else state = SDLC_CONNECTION_REQUEST; }. Évite les attentes infinies et garantit une récupération déterministe.** |

| **Propriété** | **P19** |
|---|---|

| | |
|---|---|
| **Formule** | **E<> (MASTER_BITBUS.WAIT_BITBUS_RESP and MASTER_BITBUS.x == MASTER_BITBUS.T_out_M)** |
| **Fonction** | **Vérifie l'atteignabilité de l'état où le maître atteint son timeout pendant l'attente d'une réponse. Confirme que le scénario de timeout maître est modélisé et atteignable.** |
| **Utilité FSM→C** | **Guide les tests de timeout : valide qu'on peut simuler un timeout en ne fournissant pas de réponse dans le délai T_out_M. Justifie les tests de non-réponse et de déconnexion esclave.** |

| **Propriété** | **P20** |
|---|---|
| **Formule** | **(SLAVE_BITBUS.y >= SLAVE_BITBUS.T_out_S) --> SLAVE_BITBUS.T_OUT_EXCEEDED_S** |
| **Fonction** | **Vérifie que lorsque l'horloge de l'esclave y atteint ou dépasse le timeout T_out_S, l'esclave passe systématiquement à l'état T_OUT_EXCEEDED_S. Garantit la détection et le traitement du timeous côté esclave.** |
| **Utilité FSM→C** | **Définit le timeout esclave : transition inconditionnelle vers T_OUT_EXCEEDED_S quand y >= T_out_S. Code : if (timer_y >= T_out_S) { state = T_OUT_EXCEEDED_S; }. Pas besoin de logique complexe de récupération à ce niveau, juste détection et transition.** |

| Propriété | P21 |
|---|---|
| Formule | E<> (SLAVE_BITBUS.y >= SLAVE_BITBUS.T_out_S) |
| Fonction | Vérifie l'atteignabilité de l'état où l'esclave atteint son timeout. Confirme que le scénario de timeout esclave est représenté dans le modèle. |
| Utilité FSM→C | Guide les tests de timeout esclave : valide qu'on peut simuler une absence de communication du maître. |

| Propriété | P22 |
|---|---|
| Formule | A[] (SLAVE_BITBUS.SEND_LINK_RESP imply SLAVE_BITBUS.y == T_rep) |
| Fonction | Vérifie que chaque fois que l'esclave envoie une réponse de liaison (LINK_RESP), son horloge y est exactement égale au délai de réponse T_rep. Garantit le respect du timing de réponse pour les demandes de liaison. |
| Utilité FSM→C | Impose un délai de réponse fixe pour LINK_RESP : l'esclave doit attendre exactement T_rep avant d'envoyer. Code : wait_until(timer_y == T_rep); send_link_response(); timer_y = 0;. Garantit un comportement temporel prévisible et conforme aux spécifications industrielles. |

| Propriété | P23 |
|---|---|

| | |
|---|---|
| **Formule** | **A[] (SLAVE_BITBUS.SEND_UNLINK_RESP imply SLAVE_BITBUS.y == T_rep)** |
| **Fonction** | **Vérifie que chaque fois que l'esclave envoie une réponse de déliaison (UNLINK_RESP), son horloge y est exactement égale à T_rep. Garantit le respect du timing de réponse pour les demandes de déliaison.** |
| **Utilité FSM→C** | **Unifie le timing de toutes les réponses : UNLINK_RESP utilise le même délai T_rep que LINK_RESP. Simplifie l'implémentation en utilisant une fonction générique de délai de réponse pour tous les types de messages. Code réutilisable : response_delay(T_rep); send_unlink_response();** |

| **Propriété** | **P24** |
|---|---|
| **Formule** | **A[] (SLAVE_BITBUS.SEND_DATA_RESPONSE imply SLAVE_BITBUS.y == T_rep)** |
| **Fonction** | **Vérifie que chaque fois que l'esclave envoie une réponse alphanumérique, son horloge y est exactement égale à T_rep. Garantit le respect du timing de réponse pour toutes les requêtes de données.** |
| **Utilité FSM→C** | **Confirme l'uniformité temporelle : toutes les réponses esclaves (LINK, UNLINK, DATA) utilisent T_rep. Permet d'implémenter une seule fonction de gestion de timing pour toutes les réponses, réduisant la complexité du code et les risques d'erreur.** |

### D.4 Propriétés de Vivacité

| Propriété | P25 |
|---|---|
| Formule | MASTER_BITBUS.PROCESS_BITBUS --> MASTER_BITBUS.SEND_PROCESS |
| Fonction | Vérifie que chaque fois que le maître atteint l'état de traitement PROCESS_BITBUS, il finira par atteindre l'état SEND_PROCESS. Garantit l'absence de livelock et assure la progression du cycle de communication sans blocage permanent dans l'état de traitement. |
| Utilité FSM→C | Garantit la progression du code : l'état PROCESS_BITBUS ne peut être un état terminal. L'implémentation C doit assurer qu'après le traitement, il y a toujours une transition vers SEND_PROCESS. Évite les boucles infinies dans le traitement. Structure : process_data(); state = SEND_PROCESS; avec interdiction de rester indéfiniment en PROCESS_BITBUS. |

| Propriété | P26 |
|---|---|
| Formule | MASTER_BITBUS.WAIT_FOR_CONNECTION and SLAVE_SDLC.ACK --> MASTER_BITBUS.PROCESS_BITBUS |
| Fonction | Vérifie que lorsque le maître attend une connexion et que l'esclave SDLC envoie un accusé de réception (ACK), le maître finira par atteindre l'état PROCESS_BITBUS. Garantit la reprise correcte du protocole Bitbus après établissement de la connexion SDLC. |
| Utilité FSM→C | Spécifie la synchronisation inter-couches : après réception d'un ACK SDLC, transition vers PROCESS_BITBUS pour reprendre Bitbus. Code : if (state == WAIT_FOR_CONNECTION && sdlc_ack_received()) { state = PROCESS_BITBUS; }. Assure la cohérence entre les couches SDLC et Bitbus. |

### D.5 Bornes sur les ressources

**P27**

**Propriété :**

| Propriété | P27 |
|---|---|
| Formule | A[] (MASTER_BITBUS.attempt_bb <= 2) |
| Fonction | Vérifie que le compteur de tentatives Bitbus (attempt_bb) ne dépasse jamais 2. Garantit une borne supérieure sur le nombre de retransmissions pour éviter les boucles infinies de retry et assurer une utilisation déterministe des ressources. |
| Utilité FSM→C | Borne les retries Bitbus : le code C doit implémenter un compteur attempt_bb avec vérification stricte <= 2. Code : if (++attempt_bb > 2) { abandon_and_reset(); } else { retry_transmission(); }. Garantit une terminaison déterministe et évite l'épuisement de ressources (CPU, bande passante) en cas de défaillance persistante. |

| Propriété | P28 |
|---|---|
| Formule | A[] (MASTER_BITBUS.attempt_link <= 3) |
| Fonction | Vérifie que le compteur de tentatives de liaison (attempt_link) ne dépasse jamais 3. Garantit une borne supérieure sur le nombre de tentatives de connexion pour éviter les retries infinis en cas de défaillance persistante de l'esclave. |
| Utilité FSM→C | Borne les tentatives de liaison : maximum 3 essais de LINK. Code : if (++attempt_link > 3) { report_link_failure(); state = ERROR_STATE; } else { resend_link_request(); }. Permet de détecter rapidement une défaillance esclave et d'alerter le système superviseur. |

| Propriété | P29 |
|---|---|
| Formule | A[] (MASTER_BITBUS.attempt_unlink <= 3) |
| Fonction | Vérifie que le compteur de tentatives de déliaison (attempt_unlink) ne dépasse jamais 3. Garantit une borne supérieure sur le nombre de tentatives de déconnexion, assurant une terminaison déterministe même en cas de non-réponse de l'esclave. |
| Utilité FSM→C | Borne les tentatives de déliaison : maximum 3 essais d'UNLINK. Code : if (++attempt_unlink > 3) { force_disconnect(); } else { resend_unlink_request(); }. Permet une déconnexion forcée après 3 échecs, évitant que le maître reste indéfiniment en attente de confirmation de déliaison. |

**D.6 Séquencement avec la couche SDLC**

| Propriété | P30 |
|---|---|
| Formule | A[] (MASTER_BITBUS.SEND_LINK_REQ imply MASTER_SDLC.slave_state_in_m == NRM) |
| Fonction | Vérifie que chaque fois que le maître Bitbus envoie une requête de liaison, l'état SDLC de l'esclave vu par le maître est NRM (Normal Response Mode). Garantit que les opérations de liaison Bitbus ne sont initiées que lorsque la couche SDLC est dans un état stable et opérationnel. |
| Utilité FSM→C | Impose une précondition SDLC : avant d'envoyer LINK_REQ, vérifier que slave_state == NRM. Code : if (slave_sdlc_state != NRM) { wait_for_sdlc_ready(); } send_link_request();. Assure la cohérence des couches protocoles et évite d'envoyer des requêtes Bitbus sur une liaison SDLC non établie. |

| Propriété | P31 |
| --- | --- |
| Formule | A[] (MASTER_BITBUS.SEND_ALPHANUM_M imply MASTER_SDLC.slave_state_in_m == NRM) |
| Fonction | Vérifie que chaque fois que le maître envoie des données alphanumériques, l'état SDLC est NRM. Garantit que les échanges de données applicatives ne se produisent que sur une connexion SDLC établie et stable. |
| Utilité FSM→C | Précondition pour l'envoi de données : vérifier slave_state == NRM avant chaque transmission de données. Code : assert(slave_sdlc_state == NRM); send_alphanum_data();. Prévient les pertes de données en s'assurant que la couche liaison est opérationnelle. |

| Propriété | P32 |
| --- | --- |
| Formule | A[] (MASTER_BITBUS.SEND_UNLINK_REQ imply MASTER_SDLC.slave_state_in_m == NRM) |
| Fonction | Vérifie que chaque fois que le maître envoie une requête de déliaison, l'état SDLC est NRM. Garantit que les opérations de déliaison sont effectuées dans le contexte d'une connexion SDLC active. |
| Utilité FSM→C | Précondition pour la déliaison : SDLC doit être en NRM. Code : if (slave_sdlc_state == NRM) { send_unlink_request(); } else { log_error("SDLC not ready"); }. Assure une terminaison propre de session sur une liaison active. |

| Propriété | P33 |
|---|---|
| Formule | A[] (MASTER_BITBUS.SEND_ALPHANUM_M imply !SLAVE_NO_LINKED) |
| Fonction | Vérifie que chaque fois que le maître envoie des données alphanumériques, l'esclave est lié (SLAVE_NO_LINKED == 0). Garantit qu'aucun échange de données applicatives ne se produit sans liaison Bitbus préalable établie. |
| Utilité FSM→C | Double précondition : SDLC en NRM ET Bitbus lié. Code : if (slave_sdlc_state == NRM && slave_linked) { send_data(); } else { error_not_linked(); }. Renforce la sécurité en vérifiant les deux couches protocoles avant transmission de données. |

| Propriété | P34 |
|---|---|
| Formule | A[] (MASTER_BITBUS.SEND_UNLINK_REQ imply !SLAVE_NO_LINKED) |
| Fonction | Vérifie que chaque fois que le maître envoie une requête de déliaison, l'esclave est déjà lié. Garantit la cohérence logique en interdisant les tentatives de déliaison d'une connexion inexistante. |
| Utilité FSM→C | Évite les UNLINK invalides : ne délier que si déjà lié. Code : if (!slave_linked) { log_error("Already unlinked"); return; } send_unlink_request();. Prévient les erreurs de logique et les états incohérents. |

### D.7 Gestion des erreurs et reset

| Propriété | P35 |
|---|---|
| Formule | MASTER_BITBUS.ERROR_IN_M --> MASTER_BITBUS.SDLC_CONNECTION_REQUEST |
| Fonction | Vérifie que chaque fois que le maître détecte une erreur (ERROR_IN_M), il finira par atteindre l'état de reset (SDLC_CONNECTION_REQUEST). Garantit que toute erreur détectée côté maître conduit à une réinitialisation contrôlée du protocole. |
| Utilité FSM→C | Stratégie de récupération d'erreur : toute erreur maître déclenche un reset complet vers SDLC_CONNECTION_REQUEST. Code : if (error_detected) { cleanup_state(); state = SDLC_CONNECTION_REQUEST; }. Simplifie la gestion d'erreur en utilisant une stratégie de reset globale plutôt que des récupérations locales complexes. |

| Propriété | P36 |
|---|---|
| Formule | SLAVE_BITBUS.ERROR_IN_S --> MASTER_BITBUS.SDLC_CONNECTION_REQUEST |
| Fonction | Vérifie que chaque fois que l'esclave détecte une erreur (ERROR_IN_S), le maître finira par atteindre l'état de reset. Garantit la propagation des erreurs esclave vers le maître et la synchronisation de la récupération d'erreur. |
| Utilité FSM→C | Synchronisation maître-esclave en erreur : une erreur esclave doit être signalée au maître qui initie le reset. Implique un mécanisme de signalisation d'erreur (flag, message spécial). Code maître : if (slave_error_received) { state = SDLC_CONNECTION_REQUEST; }. Assure une récupération coordonnée des deux parties. |

| Propriété | P37 |
|---|---|
| Formule | A[] (MASTER_BITBUS.ERROR_IN_M imply MASTER_SDLC.slave_state_in_m == NRM) |
| Fonction | Vérifie que chaque fois que le maître est en état d'erreur, l'état SDLC sous-jacent est NRM. Garantit que les erreurs Bitbus se produisent uniquement dans le contexte d'une connexion SDLC établie, excluant les erreurs dues à une couche liaison défaillante. |
| Utilité FSM→C | Distingue erreurs Bitbus vs SDLC : une erreur Bitbus survient sur SDLC opérationnel. Si SDLC n'est pas en NRM, l'erreur n'est pas Bitbus mais SDLC. Guide le diagnostic : if (error && sdlc_state != NRM) { root_cause = SDLC_LAYER; } else { root_cause = BITBUS_LAYER; }. |

| Propriété | P38 |
|---|---|
| Formule | A[] (SLAVE_BITBUS.ERROR_IN_S imply SLAVE_SDLC.slave_state == NRM) |
| Fonction | Vérifie que chaque fois que l'esclave est en état d'erreur, l'état SDLC de l'esclave est NRM. Garantit la même cohérence côté esclave : les erreurs applicatives sont détectées sur une couche liaison fonctionnelle. |
| Utilité FSM→C | Même principe côté esclave : erreur Bitbus seulement si SDLC en NRM. Simplifie le diagnostic d'erreur et la localisation des défaillances dans l'architecture en couches. |

| Propriété | P39 |
|---|---|
| Formule | E<> MASTER_BITBUS.PROC_SEND_LINK_REQ |
| Fonction | Vérifie l'atteignabilité de l'état où le maître traite l'envoi d'une requête de liaison. Confirme que la phase de liaison est effectivement accessible dans le modèle, validant la complétude du scénario d'établissement de connexion. |
| Utilité FSM→C | Valide la complétude du code de liaison : confirme que le chemin vers PROC_SEND_LINK_REQ existe. Justifie les tests d'établissement de connexion et garantit que cette fonctionnalité est implémentée et accessible. |

**D.8 Atteignabilité des États Clés**

| Propriété | P40 |
|---|---|
| Formule | E<> MASTER_BITBUS.PROC_SEND_UNLINK_REQ |
| Fonction | Vérifie l'atteignabilité de l'état où le maître traite l'envoi d'une requête de déliaison. Confirme que la phase de terminaison gracieuse de connexion est modélisée et accessible. |
| Utilité FSM→C | Valide la complétude du code de déliaison : confirme que le chemin de déconnexion gracieuse est implémenté. Guide les tests de terminaison de session et de cleanup de ressources. |

| Propriété | P41 |
|---|---|
| Formule | E<> MASTER_BITBUS.PROC_SEND_ALPHANUM_M |
| Fonction | Vérifie l'atteignabilité de l'état où le maître traite l'envoi de données alphanumériques. Confirme que la phase d'échange de données applicatives est accessible, validant le scénario nominal de communication. |
| Utilité FSM→C | Valide le chemin nominal complet : depuis l'initialisation jusqu'à l'envoi de données applicatives. Confirme que l'implémentation permet d'atteindre l'objectif principal du protocole (transfert de données). |

| Propriété | P42 |
|---|---|
| Formule | E<> SLAVE_BITBUS.SEND_LINK_RESP |
| Fonction | Vérifie l'atteignabilité de l'état où l'esclave envoie une réponse de liaison. Confirme que l'esclave peut répondre aux requêtes de liaison, validant le comportement réactif côté esclave pour l'établissement de connexion. |
| Utilité FSM→C | Valide la réactivité de l'esclave : confirme que l'esclave peut traiter et répondre aux LINK_REQ. Guide les tests de handshake bidirectionnel et de synchronisation maître-esclave. |

| Propriété | P43 |
|---|---|

| | |
|---|---|
| **Formule** | **E<> SLAVE_BITBUS.SEND_UNLINK_RESP** |
| **Fonction** | **Vérifie l'atteignabilité de l'état où l'esclave envoie une réponse de déliaison. Confirme que l'esclave peut participer à la terminaison gracieuse de connexion.** |
| **Utilité FSM→C** | **Valide le cleanup bidirectionnel : confirme que l'esclave participe activement à la terminaison. Guide les tests de déconnexion propre et de libération de ressources côté esclave.** |

| **Propriété** | **P44** |
|---|---|
| **Formule** | **E<> SLAVE_BITBUS.SEND_ALPHANUM_S** |
| **Fonction** | **Vérifie l'atteignabilité de l'état où l'esclave envoie des données alphanumériques. Confirme que l'esclave peut émettre des réponses de données, validant la communication bidirectionnelle complète.** |
| **Utilité FSM→C** | **Valide la communication bidirectionnelle : confirme que l'esclave peut non seulement recevoir mais aussi envoyer des données. Justifie les tests de communication full-duplex et de transfert de données dans les deux sens.** |

### D.9 Cohérence Logique, Exclusion Mutuelle et Traçabilité

| Propriété | P45 |
| --- | --- |
| Formule | A[] (MASTER_BITBUS.frame_sent_m == data_response_m imply SLAVE_NO_LINKED == 0) |
| Fonction | Vérifie que chaque fois qu'une trame de type réponse de données est envoyée, l'esclave est lié (SLAVE_NO_LINKED == 0). Garantit que les réponses de données ne sont émises que dans le contexte d'une session Bitbus établie. |
| Utilité FSM→C | Invariant de session : ne jamais envoyer de données sans liaison active. Code : assert(slave_linked == true); send_data_response();. Prévient les erreurs de logique où des données seraient envoyées hors contexte de session. |

| Propriété | P46 |
| --- | --- |
| Formule | A[] (MASTER_BITBUS.SEND_LINK_REQ imply SLAVE_NO_LINKED == 1) |
| Fonction | Vérifie que chaque fois que le maître envoie une requête de liaison, l'esclave n'est pas lié (SLAVE_NO_LINKED == 1). Garantit la cohérence logique en interdisant les demandes de liaison redondantes sur une session déjà établie. |
| Utilité FSM→C | Évite les LINK redondants : ne lier que si pas déjà lié. Code : if (slave_linked) { log_error("Already linked"); return; } send_link_request();. Prévient les incohérences d'état et optimise en évitant des opérations inutiles. |

| Propriété | P47 |
| --- | --- |
| Formule | A[] (MASTER_BITBUS.SEND_UNLINK_REQ imply SLAVE_NO_LINKED == 0) |
| Fonction | Vérifie que chaque fois que le maître envoie une requête de déliaison, l'esclave est lié. Garantit qu'on ne tente de délier que des connexions existantes, maintenant la cohérence de l'état de session. |
| Utilité FSM→C | Cohérence déliaison : ne délier que si lié (voir P34). Renforcement avec précondition stricte dans le code. |

| Propriété | P48 |
| --- | --- |
| Formule | MASTER_BITBUS.ERROR_IN_M --> MASTER_BITBUS.PROCESS_BITBUS |
| Fonction | Vérifie que chaque fois que le maître détecte une erreur, il finira par revenir à l'état de traitement PROCESS_BITBUS. Garantit qu'après la gestion d'erreur et le reset, le système retourne à un état opérationnel stable permettant la reprise du protocole. |
| Utilité FSM→C | Garantit la récupération : après erreur et reset, retour à un état opérationnel (PROCESS_BITBUS). Assure que le code ne reste pas bloqué en état d'erreur permanent. Structure de récupération : error_handler() → reset() → state = PROCESS_BITBUS; resume_normal_operation(); |

| Propriété | P49 |
| --- | --- |

| | |
|---|---|
| **Formule** | **A[] not (MASTER_BITBUS.SEND_LINK_REQ and MASTER_BITBUS.SEND_UNLINK_REQ)** |
| **Fonction** | **Vérifie l'exclusion mutuelle stricte entre l'envoi de requêtes de liaison et de déliaison. Garantit qu'aucun état du système ne permet simultanément ces deux opérations contradictoires.** |
| **Utilité FSM→C** | **Exclusion mutuelle LINK/UNLINK : implémentation via états mutuellement exclusifs. Code : enum State { SEND_LINK_REQ, SEND_UNLINK_REQ, ... }; avec une seule variable d'état, garantit naturellement l'exclusion mutuelle. Pas besoin de verrous supplémentaires.** |

| **Propriété** | **P50** |
|---|---|
| **Formule** | **A[] not (MASTER_BITBUS.SEND_ALPHANUM_M and MASTER_BITBUS.SEND_LINK_REQ)** |
| **Fonction** | **Vérifie l'exclusion mutuelle entre l'envoi de données alphanumériques et de requêtes de liaison. Garantit que les phases de liaison et de transfert de données ne se chevauchent pas.** |
| **Utilité FSM→C** | **Séquencement LINK puis DATA : jamais simultanés. Implémentation séquentielle stricte : compléter LINK avant d'autoriser DATA. Code : state = SEND_LINK_REQ; wait_link_complete(); state = SEND_ALPHANUM_M;** |

| **Propriété** | **P51** |
|---|---|

| | |
|---|---|
| **Formule** | **A[] not (MASTER_BITBUS.SEND_ALPHANUM_M and MASTER_BITBUS.SEND_UNLINK_REQ)** |
| **Fonction** | **Vérifie l'exclusion mutuelle entre l'envoi de données alphanumériques et de requêtes de déliaison. Garantit que les transferts de données et la terminaison de session sont séquentiels et non concurrents.** |
| **Utilité FSM→C** | **Séquencement DATA puis UNLINK : compléter les transferts avant déliaison. Code : finish_data_transfer(); state = SEND_UNLINK_REQ;. Évite la perte de données en transit lors de la déconnexion.** |

| **Propriété** | **P52** |
|---|---|
| **Formule** | **A[] not ((SLAVE_BITBUS.SEND_LINK_RESP and SLAVE_BITBUS.SEND_UNLINK_RESP) or (SLAVE_BITBUS.SEND_LINK_RESP and SLAVE_BITBUS.SEND_DATA_RESPONSE) or (SLAVE_BITBUS.SEND_UNLINK_RESP and SLAVE_BITBUS.SEND_DATA_RESPONSE))** |
| **Fonction** | **Vérifie l'exclusion mutuelle complète entre les trois types de réponses de l'esclave (liaison, déliaison, données). Garantit que l'esclave n'envoie qu'un seul type de réponse à la fois, assurant la cohérence et la non-ambiguïté des échanges.** |
| **Utilité FSM→C** | **Exclusion mutuelle ternaire : un seul type de réponse à la fois. Implémentation : enum ResponseType { LINK_RESP, UNLINK_RESP, DATA_RESP }; ResponseType current_response;. Simplifie le code de réception côté maître qui n'a qu'un seul type de réponse à traiter par cycle.** |

| Propriété | P53 |
|---|---|
| Formule | **MASTER_BITBUS.SEND_LINK_REQ --> (MASTER_BITBUS.RECEIVE_LINK_RESP or MASTER_BITBUS.ERROR_IN_M or MASTER_BITBUS.SDLC_CONNECTION_REQUEST)** |
| Fonction | **Vérifie la traçabilité requête-réponse pour la liaison : après l'envoi d'une requête de liaison, le système atteindra soit la réception d'une réponse, soit un état d'erreur, soit une nouvelle demande de connexion SDLC. Garantit qu'aucune requête de liaison ne reste sans suite définie.** |
| Utilité FSM→C | **Garantit un futur déterministe après LINK_REQ : trois issues possibles clairement définies. Guide l'implémentation du timeout et de la gestion d'erreur : après SEND_LINK_REQ, attendre RECEIVE_LINK_RESP avec timeout qui mène à ERROR_IN_M ou SDLC_CONNECTION_REQUEST. Code : send_link_req(); wait_response_or_timeout(); handle_outcome();** |

| Propriété | P54 |
|---|---|
| Formule | MASTER_BITBUS.SEND_UNLINK_REQ --> (MASTER_BITBUS.RECEIVE_UNLINK_RESP or MASTER_BITBUS.SDLC_CONNECTION_REQUEST) |
| Fonction | Vérifie la traçabilité requête-réponse pour la déliaison : après l'envoi d'une requête de déliaison, le système atteindra soit la réception d'une réponse de déliaison, soit une nouvelle demande de connexion SDLC. Garantit qu'aucune requête de déliaison ne laisse le système dans un état indéfini. |
| Utilité FSM→C | Garantit deux issues après UNLINK_REQ : réponse ou reset. Implémentation : send_unlink_req(); wait_response_or_timeout(); if (timeout) { state = SDLC_CONNECTION_REQUEST; } else { state = RECEIVE_UNLINK_RESP; }. Assure une terminaison propre même sans confirmation de l'esclave. |

**Conclusion**

Les cinquante-quatre propriétés formelles présentées dans cette annexe constituent un cadre structuré de vérification du protocole Bitbus. Chacune d'elles est associée à une interprétation opérationnelle permettant d'expliciter son impact sur l'implémentation en C embarqué, établissant ainsi un lien méthodologique entre la vérification par model checking et le développement logiciel.

L'analyse systématique de ces propriétés ne se limite pas à une validation théorique du modèle. Elle permet d'identifier explicitement les contraintes fonctionnelles et temporelles qui doivent être respectées lors de l'implémentation, notamment en matière de gestion des temporisations, des mécanismes de reprise, des transitions d'états exclusives et du traitement des situations d'erreur.

L'ensemble s'inscrit dans une démarche de développement maîtrisée des systèmes critiques, où les assertions issues du modèle formel servent de référence pour structurer et contraindre l'implémentation logicielle. Cette continuité contribue à renforcer la traçabilité entre spécification, vérification formelle et réalisation logicielle.