Vers la génération de modèles de sûreté de fonctionnement

Xavier Dumas*, Claire Pagetti*, Laurent Sagaspe*, Pierre Bieber*, Philippe Dhaussy**

*ONERA-CERT - 2 av. E. Belin 31055 Toulouse
nom@cert.fr
http://www.cert.fr/
**ENSIETA - DTN - 2 rue F. Verny 29806 Brest
dhaussy@ensieta.fr
http://www.ensieta.fr/dtn/index.php

Résumé. La conception et le développement de systèmes embarqués critiques sont assujettis à la fois à des objectifs économiques mais également au respect des normes de sécurité. Dès lors, la qualité des analyses de sûreté de fonctionnement et des interactions entre les experts de sûreté de fonctionnement et les équipes de développement est primordiale. Partant du constat que les échanges entre ces équipes ne sont pas encore suffisamment automatisés, nous proposons des techniques de génération automatique de modèles de sûreté de fonctionnement à partir de spécifications exprimées sous forme de modèle. L'algorithme générique proposé a été implanté par un code de transformation de modèles AADL en AltaRica et une expérimentation a été réalisée sur une spécification d'un asservissement de gouverne avionique.

1 Introduction

Contexte. La conception et le développement de systèmes embarqués critiques sont assujettis à la fois à des objectifs économiques, telle la réduction des coûts et du temps de développement, mais également au respect des normes de sécurité. Dans le contexte aéronautique, par exemple, ces contraintes sont amplifiées puisque le processus de développement doit répondre à une certaine fiabilité pour passer l'étape de certification. De ce fait, le cycle de développement est soumis à davantage de validation et de vérification ainsi qu'à une plus grande traçabilité. Dès lors, les activités d'évaluation liées à la *sûreté de fonctionnement* [1], où l'on établit le niveau de confiance justifié qu'il est possible d'attribuer à un système lorsqu'il est utilisé correctement, occupent une place prépondérante. Une représentation schématique d'un cycle de développement d'un système critique est donnée dans la figure 1. Ce cycle est le résultat de l'imbrication d'un cycle de développement que l'on pourrait qualifier de *classique* et d'un cycle de sûreté de fonctionnement.

La première étape est l'analyse des besoins qui permet ensuite de définir les grandes fonctionnalités attendues du système. Ces fonctions de haut niveau sont étudiées d'un point de

[1] Pour rappel, selon [Laprie (1989)], la sûreté de fonctionnement englobe outre la sécurité innocuité -non occurrence de défaillance à caractère catastrophique -, la disponibilité, la maintenabilité, la fiabilité, l'intégrité et la confidentialité.

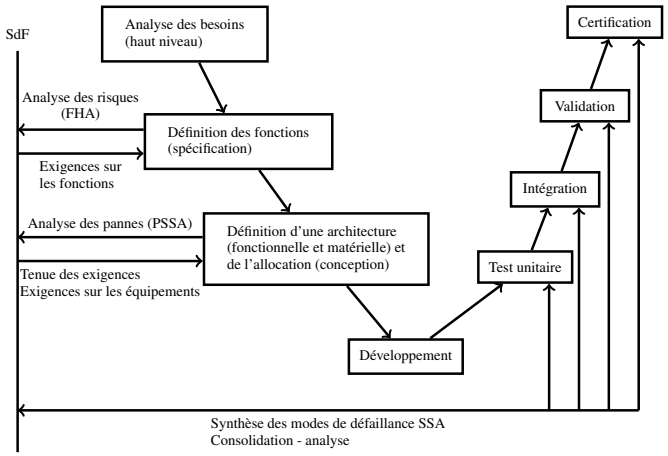

FIG. 1 – *Cycle de développement*

vue des risques associés, cette première analyse de sûreté de fonctionnement s'appelle la FHA (pour Functional Hazard Analysis). Généralement, les équipes de développement et de sûreté de fonctionnement sont dissociées, ce qui donne une importance majeure aux moyens d'interaction entre ces équipes. L'analyse des risques produit en sortie un certain nombre d'exigences que devra satisfaire le système pour répondre à des objectifs de sécurité. Les fonctions sont ensuite dérivées, en tenant compte des exigences issues de la FHA, pour proposer un découpage fonctionnel (appelé *architecture fonctionnelle*) et une architecture support préliminaire (appelée *architecture matérielle*). L'*allocation* consiste à placer les différentes fonctions sur le support matériel. L'ensemble forme une architecture préliminaire du système. Cette phase est primordiale car à partir de ses sorties, les développements des logiciels et des équipements seront lancés. Le travail présenté dans la suite se situe précisément à cette étape du développement, et plus particulièrement dans la validation que les choix préliminaires sont corrects. Lors du raffinement des fonctions et des allocations, des études de *safety* (pour sécurité innocuité) sont menées afin de valider le respect des exigences issues de la FHA sous réserve d'hypothèse sur les ressources. Cette étape s'appelle la PSSA (pour Preliminary System Safety Assessment). Si l'allocation proposée ne peut répondre aux exigences, alors les architectures et l'allocation sont rejetées, les concepteurs doivent alors reconsidérer le raffinement des fonctions et le choix du support afin de proposer une nouvelle solution compatible avec les exigences. Trouver une solution peut donc se faire à la suite d'un certain nombre d'itérations entre les équipes de développement et de sûreté de fonctionnement. Une fois obtenu l'accord des deux parties, le développement proprement dit commence. Suit ensuite la remontée du cycle avec tous les tests nécessaires à la mise sur le marché avec des consolidations de la part de l'équipe de sûreté de fonctionnement sous forme de SSA (pour System Safety Assessment).

Objectifs et contribution. Une des phases primordiales et décisives du développement d'un système hautement critique est la phase de définition des architectures fonctionnelles et matérielles, et de l'allocation associée respectant les contraintes de sûreté de fonctionnement. Les

pratiques actuelles en terme d'analyse de sécurité innocuité [Laprie et al. (1995)] sont relativement matures et outillées, on peut notamment citer les analyses par arbres de défaillance, la simulation et en particulier la simulation de Monte Carlo, ou encore les modèles de Markov. En revanche, la phase de modélisation du système pour ces analyses manque encore d'automatisation. En effet, les experts en sûreté de fonctionnement produisent leurs modèles à partir de documents, souvent textuels et non exploitables tels quels par un outil, et de leur compétence métier.

L'objectif est de proposer un atelier de spécification / modélisation unifié pour les concepteurs et de simplifier les étapes de modélisation de sûreté de fonctionnement. La première étape décrite dans ce papier est de proposer une méthodologie permettant la génération de modèle de sûreté de fonctionnement. L'idée est la suivante : si les spécifications sont exprimées dans un formalisme à base de modèles, ce qui tend à être le cas avec la prise en compte progressive de l'approche "Ingénierie Dirigée par les Modèles" [Favre et al. (2006)] (IDM), alors il est possible de générer un modèle partiel de sûreté de fonctionnement. Un des avantages de l'approche est de partir d'un modèle commun à la fois pour le développement -en effet, il existe des générateurs de code à partir de modèle - et pour les analyses -notamment de sûreté de fonctionnement. Sous réserve que les transformations soient correctes, le modèle d'analyse est conforme à la spécification étudiée. Le choix d'une transformation de modèles plutôt qu'une compilation classique repose sur l'idée de mettre en place un atelier unifié ce qui sous tend que plusieurs formalismes seront pris en entrée, comme du Simulink ou du Scade. Les règles de transformation seront relativement proches et l'adaptation du code sera plus simple. De plus, il est prévu de coder des transformations inverses, comme de l'AltaRica vers du AADL. L'IDM permet la réutilisation de briques de base d'une transformation à l'autre.

Pour mettre en œuvre cette méthodologie, nous avons opté pour les langages AADL [Feiler et al. (2006)] (pour Architecture Analysis & Design Language) et AltaRica, mais nous aurions aussi bien pu porter notre choix sur d'autres formalismes similaires. Le langage AADL est adapté pour décrire des architectures fonctionnelles et logicielles. Le langage AltaRica [Arnold et al. (2000)] est un langage formel dédié à des modélisations orientées sûreté de fonctionnement. La traduction est implantée sous forme de transformation de modèles [Favre et al. (2006)] et l'outil utilisé est KerMeta [Muller et al. (2005)], il aurait également été possible d'utiliser d'autres logiciels comme ATL par exemple.

Travaux connexes. Les travaux d'Ana-Elena Rugina [Rugina et al. (2006)] sont proches des problématiques décrites dans l'introduction, mais les choix méthodologiques sont différents. Le langage source est le langage AADL étendu avec l'annexe *AADL Error Model Annex* [SAE-AS5506/1 (2006)]. Cette annexe permet de décrire des modèles d'erreur permettant ainsi la modélisation du comportement en présence de fautes. Les modèles cible sont les réseaux de Petri, en effet plusieurs méthodes d'évaluation de sûreté de fonctionnement sur des réseaux de Petri sont disponibles [Betous-Almeida et Kanoun (2004); Kanoun et Borrel (1996); Bondavalli et al. (1999)]. La démarche générale est représentée dans la figure 2 : elle consiste à créer un modèle erreur du comportement de chaque composant en présence de ses propres fautes et éventuellement de ses réparations (sans tenir compte de son environnement). Ensuite par un ensemble de règles de transformation itératives, l'auteur construit le modèle de Réseaux de Petri permettant les analyses.

Notre approche, mise en correspondance dans la figure 3, prend en entrée des modèles AADL standard avec des informations sur les dépendances de données (exprimées dans le champ *Flow Path*) et des informations sur le type de composant (par exemple s'il s'agit d'un voteur ou d'un composant logiciel). Ce langage est encore restreint et les modèles générés, hormis les composants pris dans la librairie, ont tous des comportements par défaut similaires alors qu'il serait possible, avec l'annexe d'erreur par exemple, de réaliser une description plus fine. L'avantage de notre approche est la simplicité de l'algorithme : nous produisons un modèle AltaRica en parcourant le modèle AADL une fois ce qui n'est pas le cas de [Rugina et al. (2006)] où les transitions déclenchées par des propagations sont identifiées progressivement par itération. Le langage AltaRica propose des constructions qui permettent d'éviter ces itérations, notamment grâce aux flux partagés. Cet avantage sera maintenu lorsque nous étendrons les modèles d'entrée avec l'annexe erreur.

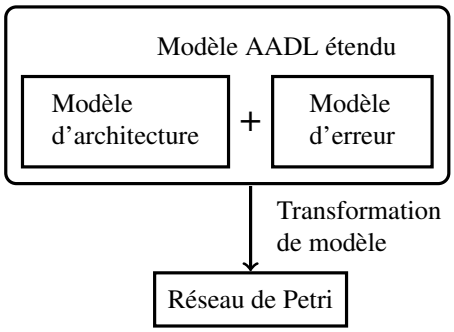

FIG. 2 – *Transformation de modèles ADDL avec un modèle d'erreur en réseaux de Petri*

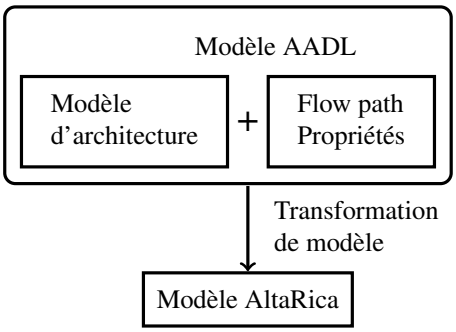

FIG. 3 – *Transformation de modèles ADDL en AltaRica*

Plan Le document est organisé de la sorte : dans la section 2, nous présentons formellement ce en quoi consiste la définition d'une spécification sous forme de système et l'application en AADL. L'algorithme est défini dans la section 3 et son implantation dans la section suivante. Enfin nous concluons en section 5.

2 Qu'est-ce qu'une spécification sous forme de modèle

Les techniques de transformation que nous proposons se basent sous l'hypothèse que la spécification du système en cours de développement est exprimée sous forme d'un modèle. Dans cette partie, nous explicitons une façon simple et relativement standard de spécifier un système par un modèle.

2.1 Modèle d'un système en phase préliminaire

Les modèles des fonctions, des ressources de calcul et de transport peuvent être représentés par des graphes orientés. L'allocation des fonctions sur les ressources est ensuite codée par une fonction de placement.

Architecture fonctionnelle. Les grandes fonctions du système sont raffinées sous forme de sous-fonctions interagissantes organisées selon les échanges des données. On représente cet ensemble organisé par une *architecture fonctionnelle*. La partie gauche de la figure 4 présente une architecture fonctionnelle constituée de quatre fonctions. Formellement, l'architecture fonctionnelle est constituée d'un graphe $(\mathcal{F}, \mathcal{C}_\mathcal{F})$ orienté pour lequel l'ensemble des noeuds $\mathcal{F} = \{f_1, ..., f_m\}$ désigne les fonctions et l'ensemble des arcs, $\mathcal{C}_\mathcal{F} \subseteq \mathcal{F} \times \mathcal{F}$, représente les échanges de données entre les fonctions. Un arc $(f_i, f_j) \in \mathcal{C}_\mathcal{F}$ modélise un flux de données f_i vers f_j. Ainsi, la fonction f_1 envoie une ou plusieurs données à la fonction f_2.

FIG. 4 – *Architectures fonctionnelle et matérielle*

Architecture matérielle. L'*architecture matérielle* du système est constituée d'un ensemble de ressources comme des calculateurs, des mémoires, des bus de communication ou des alimentations. Formellement, l'architecture matérielle est constituée d'un graphe $(\mathcal{R}, \mathcal{C}_\mathcal{R})$ orienté pour lequel l'ensemble des noeuds $\mathcal{R} = \{r_1, ..., r_k, ..., r_n\}$ désigne l'ensemble de n ressources connectées entre elles par des liaisons représentées par les arcs, $\mathcal{C}_\mathcal{R} \subseteq \mathcal{R} \times \mathcal{R}$. Dans la figure 4 (partie droite), trois ressources R_i sont connectées les unes aux autres et alimentées par une ressource A_1.

Allocation. La dernière étape de conception consiste à relier ces deux architectures et à porter les fonctions sur les ressources : l'allocation. Dans les exemples décrits ci-dessus, une affectation possible est de mettre les fonctions f_1 et f_3 sur la ressource R_1, f_2 sur R_2 et f_4 sur R_3. Formellement, une allocation peut être considérée comme une fonction $alloc : \mathcal{F} \rightarrow \mathcal{R}$ qui à chaque fonction f de $\mathcal{F}$ associe la ressource r de $\mathcal{R}$ qui l'exécute. Pour que l'allocation soit correcte il faut que les moyens de communication entre les ressources permettent le flot de données entre les fonctions : $alloc(f_i) = r_k \wedge alloc(f_j) = r_l \wedge (f_i, f_j) \in \mathcal{C}_f \Rightarrow (r_k, r_l) \in \mathcal{C}_r \vee r_k = r_l$.

2.2 Codage d'une spécification en AADL

AADL [Feiler et al. (2006)] est un langage normalisé par le SAE (Society of Automotive Engineers) de description d'architectures orienté systèmes embarqués temps réel. Une première version stable, AADL 1.0, a été publiée en novembre 2004. La modélisation d'un système peut s'effectuer de manière graphique, arborescente ou bien textuelle. AADL permet de décrire des composants logiciels et matériels de manière hiérarchique. Des outils sont également disponibles pour analyser le comportement et les performances du système.

Présentation d'AADL Pour une description détaillée et minutieuse, le lecteur se référera aux différents manuels AADL se trouvant sur les pages Web http://www.aadl.info/ ou à la thèse

de T. Vergnaud [Vergnaud (2006)]. Nous décrivons néanmoins les concepts fondamentaux en nous basant sur l'exemple de la figure 5. Les composants AADL sont définis en deux parties : l'interface (*component type*) et les implantations (*component implementation*). Un type peut posséder plusieurs implantations, une implantation possède un type unique et peut être une implantation étendue d'une autre implantation.

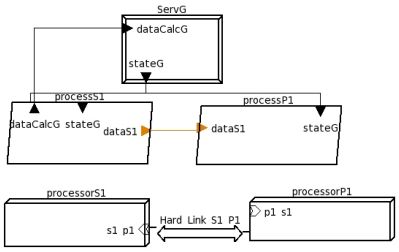

FIG. 5 – *Exemple d'un système AADL*

Le langage AADL se prête totalement à la modélisation de systèmes selon le paradigme décrit dans la section précédente. En effet, AADL propose un découpage précis entre ce qui est de l'ordre du fonctionnel, ou logiciel, et de l'ordre du matériel. Dans la figure 5, les deux parallélogrammes supérieurs décrivent l'architecture fonctionnelle et la partie inférieure l'architecture matérielle. Plusieurs composants logiciels et matériels sont prédéfinis, ainsi les parallélogrammes sont des *processus* P_i, c'est-à-dire des fonctions, dont le comportement est le résultat de l'ordonnancement des tâches temps réel internes représentées par des *thread* Th_i (non représentés dans la figure). La granularité fonctionnelle des analyses de sûreté de fonctionnement à l'étape d'architecture préliminaire se situe au niveau du processus, c'est-à-dire que les fonctions f_i décrites dans la section 2.1 correspondent à des processus AADL. De ce fait, nous ne tiendrons donc pas compte du comportement temps réel interne, ni des structures de données ou des appels de sous-programmes qui peuvent être représentés en AADL. Les processus communiquent entre eux par l'échange de données qui sont envoyées par des *ports de connexion* du composant, ces flux sont point à point mais pourront ensuite être physiquement supportés par une connexion multicast par exemple. Les ports sont déclarés dans l'interface dans le champ (*features*) avec un attribut de direction (*in* pour entrant, *out* pour sortant ou *in out* pour bidirectionnel) et sont typés par les types prédéfinis : *data port* correspond au transport d'une donnée, *event port* correspond à des communications événementielles, ainsi la réception d'événement particulier peut déclencher l'exécution d'un thread, et l'envoi de différents événements peut se faire par file d'attente, et *data event port* est la synthèse des deux types précédents. On suppose dans la suite que les processus échangent leurs données via des *dataPort*.

Dans la partie inférieure de la figure 5, les deux parallélépipèdes $ProcessorS_1$ et $ProcessorP_1$ sont des composants prédéfinis de type processeurs, ils communiquent via un bus *Hard Link S1 P1*. Les connexions depuis les composants matériels se font par des ports de connexion appelés *bus access*. Les analyses de sûreté de fonctionnement à l'étape d'architecture préliminaire ne prennent pas en compte les mémoires internes, nous les excluons dans la suite. Ainsi un composant processeur sera supposé correct s'il a accès à sa mémoire interne et que celle-ci fonctionne

également. Un autre type de composant prédéfini est le *device* (pour dispositif). Il est utilisé pour représenter des éléments dont le comportement interne n'est pas connu ou ne peut être décrit sous la forme d'un bus ou d'un CPU, comme par exemple des capteurs, des gouvernes ou des alimentations. Un device $serv_G$ est relié aux processus P_1 et S_1, on peut supposer que $serv_G$ est une servocommande qui envoie régulièrement son état aux processus et attend des ordres.

Le processus P_1 (resp. S_1) est hébergé sur le processeur $ProcesseurP_1$ (resp. $ProcesseurS_1$), cette relation est équivalente à l'allocation définie précédemment. Le système complet est représenté par un rectangle aux bords arrondis. La correspondance entre les données entrantes et les données sortantes se fait au niveau système. Ce système peut ensuite être mis en interaction avec d'autres composants ou systèmes.

Information de flux Les flux AADL ne correspondent pas à une construction réelle mais à un moyen de représenter la circulation des données à travers l'architecture afin de faciliter les analyses sur le système. Ce qui est pertinent pour une analyse de sécurité innocuité est de tracer les propagations des défaillances, les dépendances de données doivent donc être enregistrées dans le modèle de spécification. L'idée est la suivante : considérons tout d'abord le cas d'une fonction sans protection particulière qui a deux entrées et qui calcule une sortie en fonction des entrées, si la valeur de l'une des deux entrées est fausse à cause d'erreurs antérieures, alors la sortie calculée par la fonction sera elle aussi erronée. L'erreur sera propagée. En revanche, si la fonction est un voteur 2/3, elle prend en entrée 3 valeurs et tant qu'au moins 2 entrées s'accordent, elle donnera une sortie cohérente. Evidemment si ces 2 entrées sont erronées, la sortie sera également erronée mais la probabilité d'avoir 2 entrées fausses est plus faible que d'en avoir une seule mauvaise. AADL permet de définir le cheminement interne des données par le biais des *flow path*, il s'agit de tronçons de flux. Cette dépendance de données sera ensuite traduite en propagation d'erreur. Concrètement, la déclaration suivante décrit la dépendance directe entre la donnée *data_S1* et la donnée *stateG* du processus S1.

```
flows
  stateG_implies_dataS1:
  flow path stateG -> dataS1;
```

Information de propriété De même, des attributs peuvent être ajoutés dans le champ *properties* d'un processus. Des propriétés prédéfinies peuvent être utilisées de même que de nouveaux types. Dans l'exemple ci-dessous, le type *dependability_properties* a été défini et regroupe trois types d'information : les modes de défaillance associés, ici il n'y a qu'un mode la perte, le taux de perte maximum requis, ici inférieur à 10^{-7}/FH, et le type de composant, ici un voteur.

```
properties
   dependability_properties::failure_mode => loss
   dependability_properties::law => exp 1e-7
   dependability_properties::type => voteur
```

2.3 Organisation à des fins de traduction

En résumé, pour une première traduction, nous considérons un sous-ensemble des descriptions AADL telles que : les processus (resp. les processeurs) ne possèdent que des *dataPort* (resp. *bus access*), les processeurs ne contiennent pas de sous-composant, il n'y a pas de groupe de port.

Formellement, un système modélisé en AADL est une arborescence d'instances de composant interconnectées. Une instance d'un composant élémentaire AADL, i.e. une implantation d'un type, peut être vue comme un uplet $\mathcal{C} = \langle F^{in}, F^{out}, P, K \rangle$ où

- F^{in} et F^{out} sont les ports de connexion entrants et sortants du composant. On suppose que $F^{in} = D^{in} \cup A^{in}$ (resp. $F^{out} = D^{out} \cup A^{out}$) où D^{in} sont les ports de données et A^{in} sont les accès au bus. On note que les processus n'ont que des D, les processeurs et les bus des A, et les dispositifs les deux types ;
- $P \subseteq D^{in} \times D^{out}$ définit les dépendances des données spécifiées dans le *flow path* ;
- $K = (M, T, Ty)$ est l'ensemble des informations de sûreté de fonctionnement spécifiées dans les propriétés où M est l'ensemble de modes de défaillance, T est l'ensemble des taux de défaillance requis pour chaque mode de défaillance et Ty est le type de composant (prédéfini ou non). Les composants prédéfinis sont les voteurs, les comparateurs, les consolideurs, les contacteurs, les sélecteurs et les détecteurs.

Une instance d'un système est un uplet $\mathcal{S} = \langle F^{in}, F^{out}, \mathcal{S}_1, \ldots, \mathcal{S}_n, A, R \rangle$ où F^{in} et F^{out} sont les ports de connexion entrants et sortants du système, $\mathcal{S}_i$ sont les sous-systèmes de $\mathcal{S}$, A est l'allocation des composants logiciels sur les composants matériels et R est la connexion entre les différents ports des $\mathcal{S}_i$.

3 Algorithme de traduction

L'algorithme proposé est la transcription des méthodes classiques mises en œuvre par les experts de sécurité innocuité pour évaluer la sécurité d'un système. Assurer un niveau de confiance dans un système est équivalent à assurer la confiance dans les fonctions vitales du système. Par exemple, pour un aéronef en vol, la sécurité doit assurer que la fonction d'asservissement d'une gouverne est fiable et que la perte de la fonction, i.e. la non exécution de la fonction, a un taux d'occurrence inférieur à 10^{-9} par heure de vol.

Pour ce faire il faut analyser les composants fonctionnels, les f_i de la section 2.1. La première étape consiste à déterminer les modes de défaillance possible de chaque composant (fonctionnel et matériel) : la perte (la fonction ne produit pas de résultat), le fonctionnement erroné (la fonction s'exécute toujours mais produit des résultats qui sont différents de ce que la fonction est supposée calculer) ou le fonctionnement intempestif (la fonction produit un résultat alors qu'elle n'est pas supposée fournir de résultat). Ainsi, les différents états possibles d'un composant élémentaire AADL (processus, processeur, bus, device) sont décrits dans le champ *property*. La traduction en AltaRica d'un composant AADL est un automate avec pour états les différents modes de défaillance associés ainsi que l'état nominal, c'est-à-dire celui où tout est normal.

La deuxième étape est de décrire la propagation des erreurs. Si une fonction entre dans un mode de défaillance, qu'en est-il des fonctions qui utilisent les sorties de la fonction défaillante ? Si aucune protection n'a été apportée, comme une détection de la cohérence des

données, une fonction consommant une donnée erronée à de fortes chance de passer également dans ce même mode de défaillance. Nous considérons que le défaillances se propagent à travers les données échangées par les constituants du modèle AADL. Les dépendances entre les données sont décrites en AADL par les ports de connexion et les *flow path*.

Les fonctions entrent également dans un mode de défaillance à cause des défaillances des ressources matérielles qui les supportent. Si l'architecture support est explicitement décrite, il faut donc relier la défaillance physique d'une ressource avec la défaillance correspondante des fonctions supportées par cette ressource.

Dans cette section, nous présentons AltaRica et la formalisation de l'algorithme de traduction. Le preuve de correction de cet algorithme est difficile à établir. En effet, les formalismes AADL et Altarica sont difficilement comparables. En particulier, le comportement des noeuds Altarica qui permet de décrire précisément la propagation des défaillances n'a pas de contrepartie en AADL de base. Aussi, la validation du modèle Altarica produit doit être assurée par l'analyste de sécurité-innocuité, par exemple, en jouant un ensemble de scénarios de défaillances avec le simulateur interactif associé à Altarica.

3.1 Présentation d'AltaRica

Le langage AltaRica [Arnold et al. (2000)], basé sur le paradigme des automates à contraintes, a été développé au LaBRI depuis une dizaine d'années en collaboration avec des partenaires industriels dans le but de réaliser un atelier logiciel pour la description fonctionnelle et dysfonctionnelle de systèmes et d'installations industrielles. Plusieurs outils d'analyse de modèles AltaRica sont aujourd'hui disponibles tels que MecV [Vincent (2003)], Cecilia OCAS (Dassault System), AltaRica Workshop (Arboost Technology) ou Simfia (Apsys). AltaRica a été utilisé dans plusieurs projets et la faisabilité des analyses de sûreté de fonctionnement à partir de modèles AltaRica a été illustrée dans [Bieber et al. (2004)].

```
node fonction
 state s:{correct,lost};
 flow
  O:{correct,lost}:out;
  I:{correct,lost}:in;
 event
  fail_lost;
 trans
  s = correct |- fail_lost -> s:=lost;
 assert
  O = case {s = correct : I,
            else s};
 init s:=correct;
edon
```

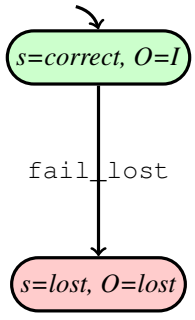

FIG. 6 – *Code AltaRica et sémantique associée*

AltaRica permet de décrire hiérarchiquement un système grâce à une organisation de nœuds. Un exemple de nœud est donné dans la partie gauche de la figure 6. Un nœud est défini par

son nom (ici *fonction*), des variables internes seulement visibles dans ce nœud introduits par la clause *state* (dans l'exemple il n'y a qu'une variable s), des variables externes appelées *flow* visibles par d'autres nœuds (dans l'exemple il y a O qui est émise et I qui est lue par le nœud), des transitions (gardes, événements, affectations) décrivant après le mot clé *trans* les différents comportements du composant (dans l'exemple, une seule transition est décrite spécifiant que si s a pour valeur *correct* et que l'événement *fail_lost* survient, alors s prend pour valeur *lost*), des contraintes définissant les configurations autorisées entre les variables. Ces assertions mettent principalement en relation les variables de flux avec les variables d'état. Dans l'exemple, la variable de flux O prend pour valeur *correct* si s et I sont corrects et *lost* sinon, ce qui traduit bien la notion d'erreur. Enfin, une valeur d'initialisation peut être déclarée. On suppose dans l'exemple que le fonctionnement est *correct* initialement. La sémantique d'un nœud AltaRica est donnée par un *système de transition interfacé* qui est un système de transition étendu avec des variables externes. La sémantique du nœud *fonction* est représentée dans la partie droite de la figure 6.

Formellement un nœud AltaRica [Rauzy (2002)], sans hiérarchie, est défini par un automate de mode $\mathcal{M} = \langle D, S, F^{in}, F^{out}, dom, \Sigma, \delta, \sigma, I \rangle$ avec :

- D est un domaine fini de valeurs des variables,
- S, F^{in} et F^{out} sont des ensembles finis de variables appelées respectivement variable d'état, variable de flux d'entrée et de sortie. Ces ensembles sont tous 2 à 2 disjoints. Dans la suite, on note $V = S \cup F^{in} \cup F^{out}$,
- $dom : V \rightarrow 2^D$ est une fonction qui associe à une variable son domaine telle que $\forall v \in V, dom(v) \neq \emptyset$,
- Σ est un ensemble fini d'événements,
- δ est une fonction partielle appelée *transition* : $dom(S) \times dom(F^{in}) \times \Sigma \rightarrow dom(S)$. On appelle $dom(S) \times dom(F^{in})$ la garde de la transition,
- σ est une fonction totale appelée *assertion* : $dom(S) \times dom(F^{in}) \rightarrow dom(F^{out})$,
- $I \subset dom(S)$ décrit les conditions initiales.

Un nœud AltaRica quelconque est un uplet $\mathcal{N} = \langle \mathcal{N}_0, \mathcal{N}_1 \ldots, \mathcal{N}_n, V \rangle$ avec $\mathcal{N}_0$ est le comportement local du nœud ; $\mathcal{N}_i$ sont les sous-nœuds de $\mathcal{N}$ et V est un ensemble de vecteurs de synchronisation. En fait, $\mathcal{N}$ est un automate de mode $\langle D, S, F^{in}, F^{out}, dom, \Sigma, \delta, \sigma, I \rangle$ où $S = \cup S_i$, $F^{in} = \cup F_i^{in}$ et $F^{out} = \cup F_i^{out}$. La relation de transition δ est le relèvement des relations de transition : soit $e = (e_1, \ldots, e_n) \in V$, si $T_i = \delta_i(S_i, I_i, e_i)$ alors $\delta(S_1, \ldots, S_n, I_1, \ldots, I_n, e) = \langle T_1, \ldots, T_n \rangle$. L'assertion est l'intersection des relèvements de toutes les assertions σ_i.

3.2 Algorithme

Un composant AADL, suivant les recommandations de la partie 2.2, peut être de deux natures d'un point de vue sûreté de fonctionnement :

- soit le champ *dependability_properties : :type* est renseigné auquel cas la traduction consiste à utiliser une librairie de composants AltaRica et générer selon le type (par exemple voteur ou comparateur). Un voteur 2/3 par exemple, prend trois valeurs en entrée et transmet la valeur sur laquelle au moins deux entrées s'accordent ;
- soit le champ n'est pas renseigné et la traduction produit un nœud similaire à celui de la figure 6 lorsqu'il n'y a qu'un mode défaillance correspondant à la perte, i.e. *dependability_properties : :failure_mode => loss*.

Dans la suite, nous ne détaillerons pas la librairie de composants et nous nous focalisons sur la traduction générique. Le nom des variables de flux varie selon les ports de connexion du composant et les formules de l'assertion du composant dépendent des *flow path*. La hiérarchie AADL est ensuite conservée dans le modèle résultat. En effet, tout composant d'un système AADL est traduit en un composant AltaRica et est déclaré comme un sous-nœud du nœud associé au système. La fonction d'allocation est implantée par des assertions et des vecteurs de synchronisation du système.

Traduction d'un composant AADL Tout composant $\mathcal{C} = \langle F^{in}, F^{out}, P, K\rangle$ avec $K.Ty = \emptyset$ est traduit en un nœud AltaRica $\mathcal{M} = \langle D, S, F^{in}, F^{out} \cup \{ST\}, dom, \Sigma, \delta, \sigma, I\rangle$ tel que :

- l'unique variable d'état s prend ses valeurs dans $\{\texttt{correct}\} \cup K.M$, elle a la valeur `correct` lorsque le composant fonctionne normalement sinon sa valeur est égales au nom d'un des modes de défaillance déclarés dans AADL. S'il n'y a qu'un mode de perte, s prendra deux valeurs `correct` et `lost` lorsqu'il est en panne ;
- les ports de connexion F^{in} et F^{out} forment l'interface du composant AADL et par transposition l'interface du composant AltaRica. On ajoute dans les flux sortants la variable ST qui symbolise l'état interne du composant AltaRica ;
- $dom(f) = \{\texttt{correct}\} \cup K.M, \forall f \in V$;
- une défaillance est produite par les événements menant à un mode de défaillance. Ainsi, s'il n'y a que la perte $\Sigma = \{\texttt{fail\_lost}\}$, s'il y a plusieurs modes, il y a tous les événements $\Sigma = \{\texttt{fail\_evt} \mid evt \in K.M\}$,
- les transitions considérées en terme de sûreté de fonctionnement sont celles qui mènent dans les modes de défaillance. S'il n'y a que la perte du composant, il n'y aura qu'une unique transition δ : $s = \textit{correct} \vdash \texttt{fail\_lost} \rightarrow s := \textit{lost}$; sinon il faut ajouter une transition similaire pour chaque mode de défaillance de $K.M$.
- pour toute variable de flow $\textit{out}_j \in F^{out}$, on considère l'ensemble $\{in_i \mid (in_i, \textit{out}_j) \in P\}$ des variables de flux entrantes dont dépend la variable $\textit{out}_j$. Alors, si l'on ne considère que la perte du composant, l'assertion associée à $\textit{out}_j$ est la suivante $\textit{out}_j =$ (si $s = \textit{correct}$ et $\wedge_{i \in K_j} in_i = \textit{correct}$ alors *correct* sinon *lost*) ;
- enfin, on considère que le comportement initial du composant est correct $I(s) = \texttt{correct}$.

Traduction d'un système Soit un système AADL $\mathcal{S} = \langle F^{in}, F^{out}, \mathcal{S}_1, \ldots, \mathcal{S}_n, A, R\rangle$, $\mathcal{S}$ est traduit en un nœud AltaRica $\mathcal{N} = \langle \mathcal{N}_0, \mathcal{N}_1 \ldots, \mathcal{N}_n, V\rangle$ avec

- chaque sous-système $\mathcal{S}_i$ est traduit en un nœud AltaRica $\mathcal{N}_i$ (algorithme récursif) ;
- $\mathcal{N}_0$ a pour variables de flux F^{in} et F^{out} ;
- l'assertion σ est équivalente à R (la connexion entre ports) ;
- le vecteur de synchronisation est donné par la relation d'allocation A : si un processus $\mathcal{S}_i$ est sur un processeur $\mathcal{S}_j$ alors $(\texttt{fail\_evt}_i, \texttt{fail\_evt}_j) \in V$ pour tout `evt dans` $K.M$.

4 Une implantation de l'algorithme

L'approche que nous avons mise en œuvre dans ce travail est basée sur l'approche Ingénierie Dirigée par les Modèles (IDM). Les méta modèles des langages AADL et AltaRica sont

exploités pour la conception de règles de transformation. Ces règles ont pour objet de traduire des modèles AADL, conformes au méta modèle AADL en des modèles AltaRica, conformes au méta modèle AltaRica. Les règles de transformation font appel aux constructions ou méta entités des meta modèles source et cible. Ces règles sont exprimées dans le langage KerMeta, développé à l'IRISA par l'équipe Triskell depuis 2004. Elles implantent l'algorithme décrit dans la section précédente avec pour restriction : un seul mode de défaillance est considéré pour tous les composants, à savoir la perte. L'ensemble des sources se trouvent en libre accès sur la page http://www.cert.fr/anglais/deri/pagetti/xavier/index.html.

Présentation de KerMeta Le logiciel choisi est KerMeta [Muller et al. (2005)], langage de méta-modélisation exécutable disponible en Open-source. La figure 7 décrit le fonctionnement général d'une transformation de modèle. Un méta modèle MM A et un modèle A, conforme à ce méta modèle, sont chargés. Le modèle A doit être conforme aux règles syntaxiques définies dans le méta-modèle. Le programme écrit en KerMeta implante des règles de transformation du modèle A en un modèle conforme au méta modèle MM B. Il génère le modèle B qui est conforme, par construction, à la structure du méta-modèle MM B.

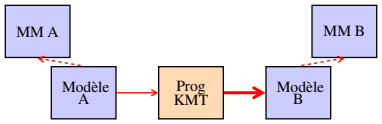

FIG. 7 – *Manipulation de modèles avec KerMeta*

Dans notre cas, le méta-modèle MM A correspond à la définition de la grammaire abstraite du langage AADL et le méta-modèle MM B à la définition de la grammaire abstraite du langage AltaRica. La méta-modélisation est une technique de définition des concepts syntaxiques, à peu près équivalente à la définition de la syntaxe d'un langage sous forme de BNF. Le méta-modèle de AADL est celui trouvé sur le site officiel. Le méta-modèle d'AltaRica a été généré directement à partir de la BNF du langage. Les règles de transformation ont été codées en KerMeta. Tous ces éléments sont téléchargeables sur la page précédemment mentionnée.

Exemple d'une règle de transformation La figure 8 illustre la clause assert contenant les spécifications et dépendances si il y en a (l.2, l.3), entre les variables de flux. Ces dépendances sont converties en relation ternaire représentées par des *case* en AltaRica dans la classe *Case* du méta-modèle (l.8 à l.12). Cette information est finalement ajoutée dans le nœud correspondant (l.13).

Résultats sur une étude de cas Ce programme a été appliqué sur une spécification simplifiée de l'asservissement d'une gouverne de direction, appelée le *rudder*, d'un aéronef. L'asservissement consiste à commander la gouverne en respectant la structure de l'avion et les ordres pilote. Ces derniers sont convertis en signaux électriques puis envoyés à des calculateurs qui déterminent le déplacement du *rudder*. Une fois ces calculs effectués, les informations sont transmises aux servocommandes munies de vérins qui vont déplacer le *rudder*. Le système est composé de :

```
//gestion des flows AADL (spécifier les chemins des ports)
if (!n.flowSpecs.isVoid) then
  if (!n.flowSpecs.flowPathSpec.isVoid) then
    n.flowSpecs.flowPathSpec.each
      {o|
     //les flows AADL seront représentés par la
         //relation ternaire case
         var cs:Declaration::Case init Declaration::Case.new
         cs.out := o.dst.name
         cs.in := o.src.name
         cs.~var:=CONST_STATE
         cs.~else:=cs.~var
         testNode.assert2.add(cs)
      } end
end  [...]
```

FIG. 8 – *Traduction des chemins de flows*

– 3 calculateurs primaires (*P1, P2, P3*) qui calculent les ordres pour les servocommandes,
– 1 calculateur secondaire actif lorsque les 3 calculateurs primaires sont défaillants (*S1*) ;
– plusieurs bus électriques et liens de communication relient les composants,
– le *rudder* actionné par les vérins de 3 servocommandes.

Un modèle complet du rudder a été réalisé en AADL et est disponible sur la page web notifiée. La version graphique est donnée dans la figure 9.

Le modèle du rudder est ensuite transformé en un modèle AltaRica, lui-même traduit en un fichier texte pour pouvoir utiliser les outils du LaBRI ou en fichier OCAS. Nous avons fait des simulations et études de propagation de pannes sur le modèle OCAS généré.

Propriétés vérifiées L'outil OCAS permet un certain nombre d'analyses sur des modèles AltaRica. La première d'entre elles est la simulation du modèle. Il est possible de tirer les événements menant aux modes de défaillance. L'interface graphique permet l'utilisation de code couleur, par exemple les liens entre composants sont verts pour indiquer que les données qui transitent sont correctes. Une fois le déclenchement d'un événement de panne, des liens deviennent rouge indiquant immédiatement les composants impactés par l'erreur.

Si le modèle AltaRica est statique, c'est-à-dire qu'il n'y a pas de réparation, il est possible de générer automatiquement des arbres de défaillances et la probabilité d'occurrence d'événements redoutés. Ces analyses sont les plus courantes dans le monde industriel.

Il est également possible de vérifier des propriétés qualitatives à base de model-checking comme par exemple *il n'existe aucune combinaison de N pannes menant à la perte du système*. Dans le cas du rudder, avec pour unique mode de défaillance la perte, le système est constitué de 16 nœuds AltaRica ayant 2 états et il faut au moins 4 pannes pour perdre le contrôle de la gouverne.

5 Conclusion

Ce travail est la première étape vers le développement d'un outil de génération automatique de modèles de sécurité innocuité. Nous pensons que cette technique peut contribuer à une meilleure utilisation du langage AltaRica dans les processus d'analyse industriels. Nous avons présenté dans cet article un algorithme de traduction relativement générique qui s'applique sur des modèles AADL. Mais il pourrait tout aussi bien s'adapter sur tout autre formalisme qui

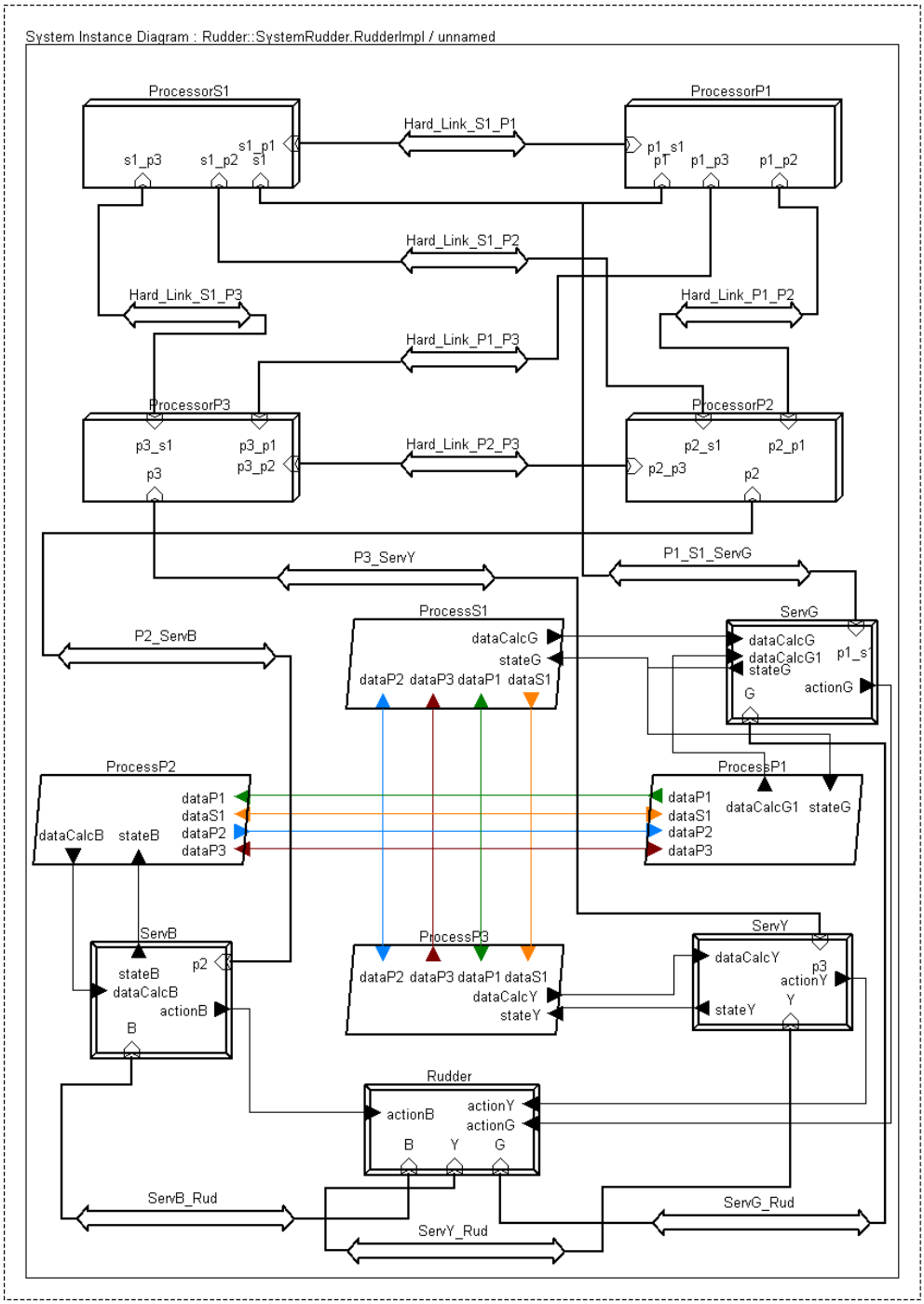

FIG. 9 – *Modélisation du Rudder en AADL*

permet de décorréler les architectures fonctionnelles et matérielles. L'algorithme, et donc son implantation, ne prend en entrée qu'une partie du méta modèle d'AADL et doit être étendu. Une attention particulière sera portée sur les modes qui permettent de modéliser des comportements intéressants, comme par exemple des politiques d'engagement qui n'ont pas été prises en compte à l'heure actuelle.

Un deuxième travail à mener concerne la gestion des modifications des spécifications d'entrée. En effet, l'expert sécurité peut enrichir le modèle AltaRica généré et dès qu'une modification sur la spécification est opérée, un nouveau modèle AltaRica est généré. Dès lors, la question se pose de conserver ou importer les enrichissements réalisés sur le modèle précédent.

Références

Arnold, A., A. Griffault, G. Point, et A. Rauzy (2000). The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae 40*, 109–124.

Betous-Almeida, C. et K. Kanoun (2004). Construction and stepwise refinement of dependability models. *Performance Evaluation 56*(1-4), 277–306.

Bieber, P., C. Bougnol, C. Castel, J.-P. Heckmann, C. Kehren, S. Metge, et C. Seguin (2004). Safety assessment with altarica. In R. Jacquart (Ed.), *IFIP Congress Topical Sessions*, pp. 505–510. Kluwer.

Bondavalli, A., I. Mura, et K. Trivedi (1999). Dependability modelling and sensitivity analysis of scheduled maintenance systems. *3rd European Dependable Computing Conference (EDCC-3)*, 7–23.

Favre, J.-M., J. Estublier, et M. Blay-Fornarino (2006). *L'ingénierie dirigée par les modèles. Au-delà du MDA (Traité IC2, série Informatique et Systèmes d'Information)*. Lavoisier.

Feiler, P., D. P. Gluch, et J. J. Hudak (2006). The Architecture Analysis and Design Language (AADL) : An introduction. Technical report, Society of Automotive Engineers (SAE).

Kanoun, K. et M. Borrel (1996). Dependability of Fault-Tolerant Systems-Explicit Modeling of the Interactions between Hardware and Software Components. *Proc. IEEE International Computer Performance & Dependability Symp.(IPDS'96) 49*, 252–61.

Laprie, J. (1989). Dependability : a unifying concept for reliable computing and fault tolerance. In T. Anderson (Ed.), *Dependability of Resilient Computers*, Chapter 1, pp. 1–28. Oxford, UK : Blackwell Scientific Publications.

Laprie, J.-C., J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac, et P. Thévenod (1995). *Guide de la sûreté de fonctionnement*. Cépaduès-Editions.

Muller, P. A., F. Fleurey, et J. M. Jézéquel (2005). Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML 2005*.

Rauzy, A. (2002). Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety 78*, 1–12.

Rugina, A.-E., K. Kanoun, et M. Kaâniche (2006). Modélisation de la sûreté de fonctionnement de système à partir du langage aadl. *Rapport LAAS*.

SAE-AS5506/1 (2006). Architecture analysis and design language annex volume 1. *Error Model Annex*.

Vergnaud, T. (2006). *Modélisation de systèmes temps-réels répartis embarqués pour la génération automatique d'applications formellement vérifiées*. Ph. D. thesis, Ecole Nationale Supérieure des Télécommunications.

Vincent, A. (2003). *Conception et réalisation d'un vérificateur de modèles AltaRica*. Phd thesis, LaBRI, University of Bordeaux 1.

Summary

The development of highly critical embedded systems is accompanied with careful safety analyses. These analyses are performed on a *safety model* which is often constructed *by hand*. The purpose of the paper is to propose a first step in the development of a unified toolbox for specifying, modeling and analysing a system. Assuming that the specifications are written in a model, for instance an AADL model, then we describe an algorithm that generates the safety model in a formal language called AltaRica. This algorithm has been partially implemented in the KerMeta framework.