Université Fédérale
Toulouse Midi-Pyrénées

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace

Présentée et soutenue par :
Hamza BOURBOUH

le jeudi 16 mars 2023

Titre :

Compilation et vérification formelle de modèles à base d'automates et de flots de données pour les systèmes critiques

Static analyses and model checking of mixed data-flow/control-flow models for critical systems

---

École doctorale et discipline ou spécialité :

ED MITT : Informatique et Télécommunications

Unité de recherche :

Équipe d'accueil ISAE-ONERA MOIS

Directeur(s) de Thèse :

M. Pierre-Loïc GAROCHE (directeur de thèse)

Jury :

M. Eric FERON Professeur KAUST, Arabie Saoudite - Président
M. Guillaume BRAT Chercheur NASA, Etats-Unis - Co-encadrant
M. Benoit COMBEMALE Professeur Université de Rennes 1 - Rapporteur
M. Pierre-Loïc GAROCHE Professeur ENAC - Directeur de thèse
Mme Laure GONNORD Professeure Université de Grenoble - Rapporteure
Mme Yassamine SELADJI Maître de conférences Université Abou Bekr Belkaid, Algérie - Examinatrice
M. Xavier THIRIOUX Professeur ISAE-SUPAERO - Examinateur

Acknowledgements

I would like to thank my advisor, Pr. Pierre-Loïc Garoche, for his guidance, support, and patience throughout this work. I would also like to thank Dr. Guillaume Brat for his great management and support and my colleagues at NASA Ames: Dr. Anastasia Mavridou, Mr. Maxime Arthaud, Dr. Irfan Sljivo, and Dr. Mohammad Hejase for their help and feedback. I would like to thank Dr. Temesghen Kahsai for his supervision during my internship that introduced me to this Ph.D. topic. I would like to thank Pr. Christophe Garion and Pr. Xavier Thirioux for the all these years working closely on several papers. I would like to thank my beloved wife, my parents, and family for their continuous support. This work was partially supported by the ANR-INSE-2012 CAFEIN project, CNES project No. R-S16/BS-0004-045 and NASA Contract No. NNX14AI09G.

Résumé — Ce travail de thèse porte sur la vérification de modèles Simulink/Stateflow par les méthodes formelles. L'objectif est de permettre la vérification des modèles Simulink par rapport aux propriétés formelles qui représentent les exigences du système. Nous avons proposé une traduction bidirectionnelle de Simulink/Stateflow vers Lustre, un langage synchrone avec une sémantique formelle bien définie. Le principal résultat de ce travail de thèse est la boîte à outils CoCoSim: un projet open source pour spécifier et vérifier les exigences définies par l'utilisateur sur les modèles Simulink. La boîte à outils a été conçue pour faciliter les activités de vérification et de validation (V&V) des modèles Simulink. De plus, la boîte à outils est hautement automatisée et possède une architecture personnalisable et configurable qui permet d'intégrer d'autres techniques pour augmenter l'évolutivité. La boîte à outils a également été intégrée avec d'autres outils de vérification pour accroître son applicabilité.

Mots clés : Conception basée sur des modèles, Systèmes cyber-physiques, Vérification logicielle, Méthodes formelles, Simulink, Stateflow, Lustre, Vérification et validation.

Abstract — This Ph.D. work is focused on the verification of Simulink/Stateflow models by means of formal methods. The objective is to enable the verification of Simulink models with respect to formal properties that represent system requirements. We proposed a bidirectional translation from Simulink/Stateflow to Lustre, a synchronous language with well-defined formal semantics. The primary outcome of this Ph.D. work is the CoCoSim toolbox: an open-source framework for specifying and verifying user-defined requirements on Simulink models. The toolbox has been designed to ease verification and validation (V&V) activities for Simulink models. In addition, the toolbox is highly automated and has a customizable and configurable architecture that allows other techniques to be integrated to increase scalability. The toolbox has also been integrated with other verification tools to increase its applicability.

Keywords: Model-based design, Cyber-physical systems, Software verification, Formal methods, Simulink, Stateflow, Lustre, Verification and validation.

Robust Software Engineering, NASA Ames Research Center, CA, U.S.A

Contents

List of Figures

List of Tables

Table of Acronyms

CPS	*Continuation-Passing Style*
IC3	*Incremental Construction of Inductive Clauses for Indubitable Correctness*
MDD	*Model-Driven Development*
PDR	*Property Directed Reachability*
SMT	*Satisfiability Modulo Theories*

Résumé

Introduction

Ce travail de doctorat porte sur la vérification formelle des modèles Simulink, qui sont largement utilisés dans la conception de systèmes critiques tels que les systèmes de contrôle de vol. La vérification formelle est un processus qui consiste à prouver qu'un système répond à ses exigences et spécifications en utilisant des méthodes mathématiques. La vérification formelle des modèles Simulink est nécessaire car ces modèles peuvent être complexes et sujets à des erreurs, et les conséquences de ces erreurs peuvent être graves dans les systèmes critiques pour la sûreté.

Le principal résultat de ce travail est la toolbox CoCoSim, un framework open-source pour spécifier et vérifier les exigences définies par l'utilisateur sur les modèles Simulink. La toolbox CoCoSim est conçue pour faciliter les activités de vérification et de validation (V&V) des modèles Simulink, notamment la validation de la compilation et la connexion d'outils externes. Elle fournit une plateforme permettant aux utilisateurs de spécifier leurs exigences dans un langage formel, puis de vérifier automatiquement ces exigences par rapport au modèle Simulink.

La motivation de ce travail est de permettre l'application de prototypes de niveau recherche à des modèles réels conçus par des ingénieurs de contrôle dans Simulink, comblant ainsi le fossé entre la recherche universitaire et l'utilisation industrielle. CoCoSim assure la traduction d'un large sous-ensemble de blocs discrets Simulink dans Lustre et fournit des moyens d'afficher et de comprendre les résultats des analyses dans MATLAB Simulink. Nous avons appliqué CoCoSim à de nombreuses études de cas de taille moyenne. Nous pensons qu'il est suffisamment puissant pour de véritables modèles Simulink industriels. Cependant, nous souhaitons qu'il s'agisse d'une boîte à outils à code source ouvert afin que la communauté puisse concevoir et démontrer des analyses encore plus puissantes (extensibilité ou applicabilité). La motivation est d'illustrer et de soutenir l'application pratique de méthodes formelles sur des modèles Simulink.

Le rapport commence par une discussion du contexte et des travaux connexes, notamment la représentation formelle des modèles Simulink/Stateflow et la spécification formelle des propriétés. Le chapitre 2 du rapport fournit une compilation bidirectionnelle de Simulink/Stateflow vers Lustre, qui est un langage de flot de données synchrone ayant une base sémantique formelle. Cette compilation est essentielle pour permettre à la toolbox CoCoSim de raisonner sur les modèles Simulink de manière formelle. Le chapitre 3 décrit en détail la toolbox CoCoSim, notamment son architecture, ses caractéristiques et ses capacités. La toolbox comprend une variété de modules pour la compilation, la vérification et la validation, et elle prend en charge plusieurs solveurs pour générer des conditions de vérification et des obligations de preuve.

Enfin, le rapport se termine par quelques cas d'utilisation démontrant les capacités de CoCoSim. Ces cas d'utilisation illustrent comment la toolbox peut être utilisée pour spécifier et vérifier une variété de propriétés sur les modèles Simulink, y compris les propriétés de sûreté, la

correction fonctionnelle et les exigences de temps. Dans l'ensemble, le travail présenté dans cette thèse de doctorat apporte une contribution importante au domaine de la vérification formelle des modèles Simulink et fournit un outil précieux pour assurer la sûreté et la correction des systèmes critiques.

Chapitre 1: Contexte et travaux connexes

Ce chapitre fournit une description de Simulink, Stateflow et Lustre. Il aborde ensuite les travaux connexes concernant la sémantique formelle des modèles Simulink et Stateflow et la spécification formelle des propriétés.

Simulink

Simulink [97], développé par MathWorks, est un langage de programmation graphique permettant de modéliser des systèmes dynamiques, y compris des systèmes à temps discret, c'est-à-dire des systèmes de flux de données synchrones. Simulink a gagné en popularité dans le développement de systèmes embarqués critiques. Il prend en charge la conception et la simulation de systèmes complexes avant de générer automatiquement un code C intégré. Un modèle Simulink consiste en un ensemble de blocs reliés par des signaux qui peuvent être organisés en modèles hiérarchiques. Par exemple, la figure 1 illustre un exemple de chronomètre qui mesure le temps écoulé entre l'activation et la désactivation. Deux signaux externes contrôlent le chronomètre : un signal de basculement pour activer le chronomètre appelé "toggle" et un signal de réinitialisation pour remettre le compteur à zéro appelé "reset".

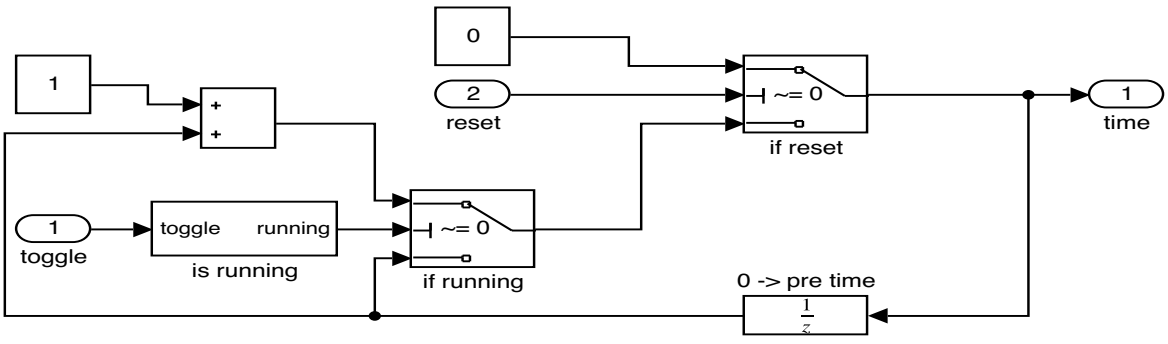


Figure 1: Exemple de chronomètre dans Simulink.

Simulink est largement utilisé pour concevoir et simuler des systèmes de contrôle, des systèmes de communication, des algorithmes de traitement du signal et d'autres systèmes dynamiques. Il permet aux utilisateurs de créer des modèles à l'aide d'une interface glisser-déposer, avec une bibliothèque de blocs prédéfinis pour la modélisation de composants courants tels que les intégrateurs, les filtres et les boucles de rétroaction. Les utilisateurs peuvent également créer leurs propres blocs personnalisés à l'aide du code MATLAB, ce qui permet une plus grande flexibilité dans la modélisation de systèmes complexes.

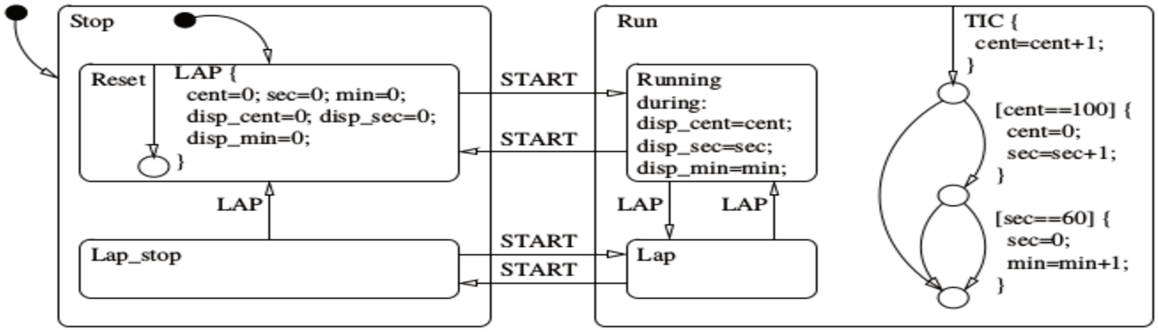


Figure 2: Chronomètre STATEFLOW modèle.

Stateflow

STATEFLOW [96] est une toolbox développée par MathWorks Inc. qui étend les fonctionnalités de SIMULINK. STATEFLOW fournit un environnement de modélisation graphique pour la conception et la simulation de logiques de contrôle et de machines à états complexes. Il permet aux utilisateurs de créer des modèles de systèmes pilotés par événements à l'aide de diagrammes d'états, qui sont une forme de machine à états finis.

STATEFLOW permet la création de modèles incluant des événements discrets, une logique de décision et des transitions conditionnelles, qui peuvent être utilisés pour modéliser des systèmes de contrôle complexes. Il fournit un éditeur graphique pour créer des diagrammes d'état et prend en charge une variété de types de données et d'expressions pour spécifier la logique et les conditions. STATEFLOW permet également aux utilisateurs de définir des fonctions et des types de données personnalisés dans MATLAB.

STATEFLOW est utilisé dans de nombreux domaines, notamment l'automobile, l'aérospatiale et l'automatisation industrielle. Il est souvent utilisé en conjonction avec SIMULINK, qui fournit une plateforme pour la modélisation et la simulation de systèmes dynamiques. Ensemble, SIMULINK et STATEFLOW fournissent un ensemble complet d'outils pour la conception, la simulation et l'analyse de systèmes de contrôle complexes.

La figure 2 montre le modèle de chronomètre qui capture le comportement de base d'un chronomètre. Ce système interagit avec son environnement par le biais de trois signaux : START, LAP et TIC. TIC modélise l'incrément de temps du système. Les événements START et LAP modélisent l'interaction de l'utilisateur avec le dispositif. Il existe deux états de niveau supérieur, Run et Stop, et quatre états internes, Reset, Lap stop, Running et Lap. L'appareil est initialisé en mode arrêt et passe de l'état arrêt à l'état marche. Lorsqu'il est en état de marche, il peut être mis en pause à l'aide du signal LAP. Chaque signal TIC reçu dans l'état Run incrémente le timer, en effectuant des effets secondaires sur les variables internes du timer (c.-à-d. cent, sec, min). Ce comportement spécifique est modélisé par un ensemble de transitions utilisant uniquement des jonctions et commençant par une transition interne. Ce modèle est intéressant car il s'appuie sur plusieurs constructions de STATEFLOW: les transitions internes et externes, les jonctions et les états hiérarchiques.

SIMULINK/STATEFLOW sont également largement utilisés pour la génération de code, qui permet de traduire automatiquement les modèles en code exécutable dans des langages de programmation tels que C ou C++. Cela peut être particulièrement utile dans les applications

critiques pour la sûreté, où la fiabilité et la correction du logiciel sont de la plus haute importance. Les sections 1.1.1 et 1.1.2 fournissent de plus amples informations sur SIMULINK et STATEFLOW.

Lustre

LUSTRE [29] est un langage de programmation de flux de données synchrone utilisé pour la conception de systèmes réactifs, y compris les systèmes en temps réel et les systèmes critiques de sûreté. Il a été initialement développé par Paul Caspi et son équipe au laboratoire Verimag de Grenoble, en France, dans les années 1980. LUSTRE se caractérise par sa sémantique d'exécution synchrone, ce qui signifie que tous les composants du système sont mis à jour simultanément à des points discrets dans le temps, par exemple des fronts montants d'un singal booleen jouant le role d'horloge. Cela permet une analyse et une vérification formelles des programmes LUSTRE, garantissant qu'ils respectent certaines propriétés de sûreté. LUSTRE a été utilisé dans de nombreux domaines, notamment l'avionique, l'automobile et les systèmes de contrôle industriels. Il a également influencé le développement d'autres langages de flot de données synchrones, tels que Signal et Esterel. La section 1.1.3 fournit davantage d'informations sur LUSTRE.

```
1  node count (tick : bool) returns (time : int);
2  let
3    time = 0 -> pre time + 1;
4  tel
5
6  node stopwatch (tick, toggle, reset : bool) returns (time : int);
7  var running : bool clock;
8  let
9    running = ((false -> pre running) <> toggle) or reset;
10   time = merge running (true  -> count(tick when running) every reset)
11                        (false -> (0 -> pre time) when not running);
12 tel
```

Listing 1: L'exemple du chronomètre avec des horloges en Lustre

Notre boîte à outils COCOSIM utilise LUSTRE [29] comme langage intermédiaire, car sa sémantique est bien définie et a été étudiée dans la recherche pendant de nombreuses années. De plus, LUSTRE est bien adapté à la vérification formelle, et de nombreux vérificateurs de modèles acceptant LUSTRE en entrée ont été développés en utilisant différentes techniques de vérification telles que Kind2 [32], Zustre [52], et JKind [59].

Le sous-ensemble discret de SIMULINK a été traduit en divers langages d'entrée à des fins de vérification de modèle, tels que NuSMV [102], LUSTRE [131] et les automates hybrides [123]. Tripakis et al. (2005) ont réalisé le premier travail de traduction de SIMULINK vers LUSTRE [131]. Nous partageons le même algorithme de génération récursive d'un nœud LUSTRE pour chaque sous-système, mais nous différons dans la manière de traduire certains blocs et de gérer les systèmes multipériodiques. Malheureusement, il est impossible de comparer l'outil car son binaire n'est pas maintenu et ne supporte pas les nouvelles versions des modèles SIMULINK. Zhou et Kumar (2012) ont proposé une approche récursive pour traduire une classe de diagrammes SIMULINK en automates finis étendus aux entrées/sorties (I/O-EFA) [139], la représentation formelle du modèle est aplatie et perd la hiérarchie et l'architecture du modèle SIMULINK.

CoCoSim traduit le modèle Simulink de manière modulaire. Chaque sous-système Simulink est traduit en un nœud Lustre. Cette modularité permet à l'outil d'effectuer la vérification par composition, une technique qui nous aide à passer à l'échelle avec de grands modèles. Meenakshi et al. (2006) [102] analysent la représentation textuelle du modèle et utilisent le langage d'entrée NuSMV comme langage formel intermédiaire pour la vérification à l'aide d'un vérificateur de modèle symbolique ; les types de données pris en charge par le langage sont limités aux booléens et aux entiers bornés. La validation de la traduction de Simulink au langage d'entrée NuSMV n'a pas été abordée.

Au fil des ans, plusieurs approches ont été proposées pour analyser les diagrammes de Stateflow. Ces approches manquent souvent d'une ou de plusieurs des caractéristiques souhaitées suivantes : (i) une sémantique formelle convaincante ; (ii) une compilation fidèle qui préserve la structure hiérarchique du modèle Stateflow et enfin, (iii) un moteur d'analyse entièrement automatisé ; Notre travail vise à fournir un cadre pour répondre de manière adéquate à tous ces points. Notre approche est basée sur une série d'articles de Hamon [78, 77, 79] fournissant une sémantique opérationnelle et dénotationnelle pour Stateflow, et concevant des interprètes pour Stateflow. Notre contribution est détaillée dans la section 2.3.

De plus, tous les travaux antérieurs sont soit propriétaires, soit non maintenus, ce qui pose des problèmes d'adaptation aux nouvelles versions de Simulink. Nous avons relevé ces deux défis en faisant de CoCoSim une architecture ouverte, de sorte que chacun puisse utiliser notre outil, y contribuer ou le personnaliser. En outre, notre traducteur CoCoSim utilise les API Simulink pour accéder et calculer toutes les informations de modèle nécessaires à la traduction au lieu d'analyser le format textuel du modèle. Par conséquent, les nouvelles versions de Simulink sont moins susceptibles d'avoir un impact négatif sur le traducteur. En outre, le traducteur est modulaire et les nouveaux blocs Simulink peuvent être facilement pris en charge et intégrés.

Un autre défi est le manque d'outils permettant la vérification automatique des propriétés Simulink/Stateflow spécifiées en tant qu'observateurs synchrones. Les observateurs synchrones sont un formalisme utilisé pour spécifier les propriétés temporelles des systèmes synchrones, tels que les modèles Simulink et Stateflow. Ces propriétés sont typiquement exprimées dans une logique telle que la logique temporelle ou la logique temporelle linéaire et peuvent être utilisées pour vérifier qu'un système répond à certaines exigences de sûreté et de correction. Dans le cas des langages et modèles synchrones, de nombreux travaux [37, 92, 135] préconisent l'utilisation d'exigences liées aux composants. Cependant, il est à noter que la définition de tels contrats ou leur raisonnement reste un défi. Les contrats formalisés peuvent être utilisés pour de nombreuses applications : oracles de test, synthèse de test, synthèse réactive [84, 86], raisonnement compositionnel ou validation de contrats individuels. CoCoSpec [31, 33] est un langage de spécification pour Lustre [73] et a été étendu aux modèles Simulink.

Cependant, l'absence d'outils permettant la vérification automatique des observateurs synchrones dans les modèles Simulink et Stateflow rend difficile l'application de ces méthodes formelles dans la pratique. Bien qu'il existe quelques outils pour vérifier les propriétés temporelles dans les modèles Simulink et Stateflow, ces outils ne sont pas largement utilisés et ne couvrent pas nécessairement toutes les caractéristiques des modèles.

En résumé, l'absence d'une base sémantique formelle et d'outils pour soutenir la vérification

automatique des modèles Simulink et Stateflow rend difficile de garantir leur sûreté et leur exactitude dans la pratique. Des recherches supplémentaires sont nécessaires pour développer des méthodes et des outils formels largement adoptés et pouvant être utilisés efficacement pour vérifier ces systèmes complexes.

Chapitre 2: Compilation bidirectionnelle des modèles Simulink/Stateflow vers Lustre

Ce chapitre décrit une contribution importante au domaine de la vérification formelle des modèles Simulink/Stateflow: une compilation bidirectionnelle entre Simulink et le langage de programmation Lustre. Cette compilation permet la vérification formelle des modèles Simulink/Stateflow, ce qui est crucial pour garantir la sûreté et la correction des systèmes critiques.

La première partie du chapitre décrit la compilation sémantique des modèles Simulink vers Lustre. Le programme Lustre résultant préserve le comportement fonctionnel du modèle Simulink original, tout en fournissant une sémantique formelle qui permet de raisonner sur son comportement. L'algorithme est expliqué dans la Section 2.2.1, et les blocs qui nécessitent plus d'attention pour capturer correctement leur sémantique sont expliqués dans les Sections 2.2.4, 2.2.5, et 2.2.6. Dans le chapitre 3, nous présentons CoCoSim où cet algorithme est implémenté, et où la sémantique de plus d'une centaine de blocs Simulink est définie comme des fonctions MATLAB produisant du code Lustre.

La deuxième partie du chapitre présente la sémantique dénotationnelle des modèles Stateflow que nous avons publiée dans [17]. Nous proposons un processus de compilation utilisant une sémantique dénotationnelle de type CPS (continuation-passing style). Notre technique de compilation préserve le comportement structurel et modal du système. L'approche globale est implémentée et intégrée dans notre toolbox open-source CoCoSim (voir chapitre 3). Nous présentons également des évaluations expérimentales préliminaires dans le chapitre 3, illustrant l'efficacité de notre approche dans la génération de code et la vérification de sûreté des modèles Stateflow à l'échelle industrielle.

Enfin, le chapitre montre également comment recompiler le code Lustre en modèles Simulink. Il s'agit d'une étape cruciale pour les applications pratiques de la vérification formelle, car elle permet de réintégrer les résultats de l'analyse formelle dans le modèle Simulink. La compilation de Lustre à Simulink implique la mise en correspondance de constructions Lustre avec des blocs Simulink, puis la connexion de ces blocs pour créer un modèle Simulink qui préserve le comportement du programme Lustre original. L'approche a été validée en démontrant l'équivalence comportementale entre certains modèles compilés. Les applications sont nombreuses, de la validation du framework à l'assistance à la spécification formelle ou à la production de preuves exécutables en tant qu'observateurs synchrones. Il est maintenant intégré dans la toolbox CoCoSim et est suffisamment mature pour être utilisé automatiquement pour fournir un retour d'information au niveau du modèle. Les travaux futurs comprennent l'extension du langage d'entrée pour permettre l'utilisation de fonctions définies en externe, comme le code C, et la manipulation de types de données machine (par exemple, int8, uint8, int16, uint16).

Dans l'ensemble, la compilation bidirectionnelle entre Simulink/Stateflow et Lustre fournit un outil important pour la vérification formelle des modèles Simulink, permettant de vérifier les propriétés de sûreté et de correction des systèmes critiques. Le processus de compilation consiste à convertir les modèles Simulink en un langage formel avec une sémantique bien définie, puis à renvoyer les résultats au modèle Simulink d'origine, ce qui permet de mettre en relation le monde de la vérification formelle et la conception de systèmes pratiques.

Chapitre 3: CoCoSim: Une boîte à outils pour faciliter les activités de V&V

CoCoSim est une toolbox open-source qui fournit une base sémantique formelle pour un sous-ensemble bien défini de blocs Simulink/Stateflow. Elle permet la vérification des modèles Simulink/Stateflow à l'aide de méthodes formelles et la génération de code. CoCoSim est structuré comme un compilateur qui itère sur les blocs Simulink en utilisant l'API MATLAB et produit des nœuds Lustre équivalents. Les contrats, qui sont des observateurs synchrones attachés aux sous-systèmes Simulink, peuvent être utilisés pour dénoter les éléments du contrat tels que les hypothèses et les garanties.

Le framework CoCoSim est conçu pour prendre en charge l'analyse des systèmes Simulink critiques en matière de sûreté. La validité des nœuds Lustre générés par CoCoSim peut être vérifiée à l'aide d'un contrôle de modèle basé sur SMT. CoCoSim est un framework hautement automatisé qui prend en charge différentes techniques de vérification et peut s'adapter à de grands modèles. La chaîne d'outils a été utilisée pour vérifier les propriétés d'une série de modèles, notamment un système de contrôle des micro-ondes, le système d'amarrage de la navette spatiale et un système de pompe à perfusion analgésique contrôlée par le patient. La compilation de Simulink à Lustre a été validée à l'aide de tests et de vérifications d'équivalence, montrant que la compilation est saine.

Le framework est conçu pour fournir une intégration étroite et une boucle de rétroaction entre les exigences de haut niveau et la vérification et la validation des modèles ou du code par rapport à ces exigences. Il relie les exigences formelles aux modèles Simulink pour la vérification, et les résultats de la vérification aux exigences.

CoCoSim fonctionne comme un compilateur, transformant les blocs Simulink en nœuds Lustre. Il permet ensuite de compiler le modèle Lustre en code C ou de l'analyser à l'aide de vérificateurs de modèles Lustre tels que Kind2 ou Zustre. CoCoSim assure la traçabilité entre les vérificateurs de modèles et les modèles Simulink, permettant ainsi de fournir des informations aux modèles. Ce framework automatisé prend en charge la vérification et la génération de code pour les modèles Simulink/Stateflow.

Moteur de prétraitement des blocs

Le moteur de prétraitement de CoCoSim simplifie les modèles Simulink en remplaçant certains blocs prédéfinis par des blocs de base équivalents. Les bibliothèques de prétraitement, basées sur des fonctions MATLAB, transforment le modèle en utilisant l'API Simulink. CoCoSim permet

également de vérifier l'équivalence entre le modèle original et le modèle simplifié. L'objectif principal de ce prétraitement est de réduire le nombre de blocs à traduire en Lustre en les remplaçant par des blocs plus simples.

Une représentation interne des modèles Simulink

Après le prétraitement des blocs Simulink, nous générons une structure de données contenant toutes les informations nécessaires sur le modèle, les blocs et les paramètres de configuration. Cette représentation présente deux avantages. Premièrement, elle nous permet d'obtenir toutes les informations sur le modèle en une seule fois en utilisant l'API Simulink, évitant ainsi de parcourir le format textuel du modèle Simulink qui peut changer à chaque nouvelle version. Deuxièmement, cette représentation peut être exportée au format JSON, ce qui facilite l'intégration de compilateurs externes avec la boîte à outils.

Traduction sémantique des diagrammes Simulink/Stateflow

Les blocs Simulink pris en charge par CoCoSim incluent plus de 100 blocs couramment utilisés, qui sont soit transformés en blocs plus simples, soit traduits directement en Lustre. Le traducteur, développé en MATLAB avec un modèle de conception de visiteur, permet d'ajouter facilement de nouvelles traductions ou de personnaliser les traductions existantes. En ce qui concerne les blocs Stateflow, CoCoSim prend en charge les constructions les plus connues, notamment les états exclusifs et parallèles, les actions d'état (entrée, pendant et sortie), ainsi que les transitions avec différentes combinaisons d'événements, conditions et actions.

Backends de CoCoSim

Les analyses de CoCoSim sont effectuées sur le code Lustre généré et les résultats sont renvoyés à Simulink. Les outils connectés sont open source et disponibles gratuitement telles que Kind2 [32], Zustre [52], et JKind [59].

Évaluation expérimentale de la correction de la traduction

Nous avons testé CoCoSim sur de nombreux modèles Simulink générés automatiquement. Tous les composants de l'architecture de CoCoSim ont été testés individuellement, y compris chaque bibliothèque de l'étape de prétraitement, qui a été validée à l'aide de Simulink Design Verifier, comme expliqué dans la Section 3.2.1.

Pour valider notre traduction, nous avons généré automatiquement un ensemble de tests de régression basés sur les blocs atomiques que nous prenons en charge. Nous avons inclus les variations les plus courantes des paramètres de chaque bloc, y compris les types de données, les dimensions des entrées et les blocs exécutés de manière conditionnelle. Au total, nous avons créé environ 2000 tests comprenant en moyenne 16 variations par bloc.

Chapitre 4: Cas d'utilisation

CoCoSim est une toolbox conçue pour faciliter la vérification des modèles Simulink/Stateflow, qui sont couramment utilisés dans la conception de systèmes critiques pour la sûreté. La vérification formelle de tels systèmes est une tâche complexe qui implique de multiples outils et méthodes. Dans ce contexte, CoCoSim a été appliqué à deux cas d'utilisation pour démontrer son applicabilité dans des scénarios du monde réel.

Le premier cas d'utilisation concerne les dix défis cyber-physiques de Lockheed Martin, qui sont un ensemble de modèles Simulink industriels et d'exigences fonctionnelles en langage naturel développées par des experts du domaine. CoCoSim a été utilisé conjointement avec FRET (Formal Requirements Elicitation Tool) pour effectuer une analyse de bout en bout des défis. Le processus a commencé par l'élicitation et la formalisation des exigences, suivies par l'attachement des exigences aux modèles Simulink/Stateflow et leur compilation en Lustre à l'aide de CoCoSim. Enfin, le modèle Lustre a été analysé et vérifié à l'aide de différentes techniques de vérification. Ce cas d'utilisation a démontré l'efficacité de CoCoSim dans la vérification de systèmes complexes et sa capacité à s'intégrer à d'autres outils.

Le deuxième cas d'utilisation concerne la vérification du système de navigation d'un rover. Dans ce cas d'utilisation, différentes méthodes de vérification ont été intégrées, y compris CoCoSim, FRET, AdvoCATE, et Event-B. Un *Assurance case* a été utilisé comme point d'intégration pour maintenir des liens formels cohérents entre les processus de développement et d'assurance. Ce cas d'utilisation a démontré les avantages de l'utilisation d'un *Assurance case* pour intégrer différents outils de vérification et maintenir une traçabilité claire des résultats de vérification.

Dans l'ensemble, ces cas d'utilisation démontrent l'efficacité de CoCoSim dans la vérification de systèmes complexes et la capacité d'intégration avec d'autres outils et framework. L'utilisation d'un *Assurance case* comme point d'intégration offre des avantages supplémentaires en maintenant une traçabilité claire des résultats de la vérification.

Le cadre intégré FRET-CoCoSim

La Figure 3 présente le flux de notre cadre d'analyse. Voici les étapes clés :

Étape 0 : L'utilisateur rédige et affine les exigences dans fretish, en se basant sur les explications sémantiques et les capacités de simulation de FRET.

Étape 1 : Les exigences sont traduites en formules Pure Past-Time / Future-Time Metric LTL (pmLTL / fmLTL).

Étape 2 : Les données du modèle en cours d'analyse sont utilisées pour établir une correspondance entre les propositions des exigences et les signaux Simulink.

Étape 3 : Les formules pmLTL / fmLTL et la correspondance architecturale sont utilisées pour générer des moniteurs CoCoSpec et des données de traçabilité.

Étape 4 : Les moniteurs CoCoSpec générés, les données de traçabilité et le modèle Simulink

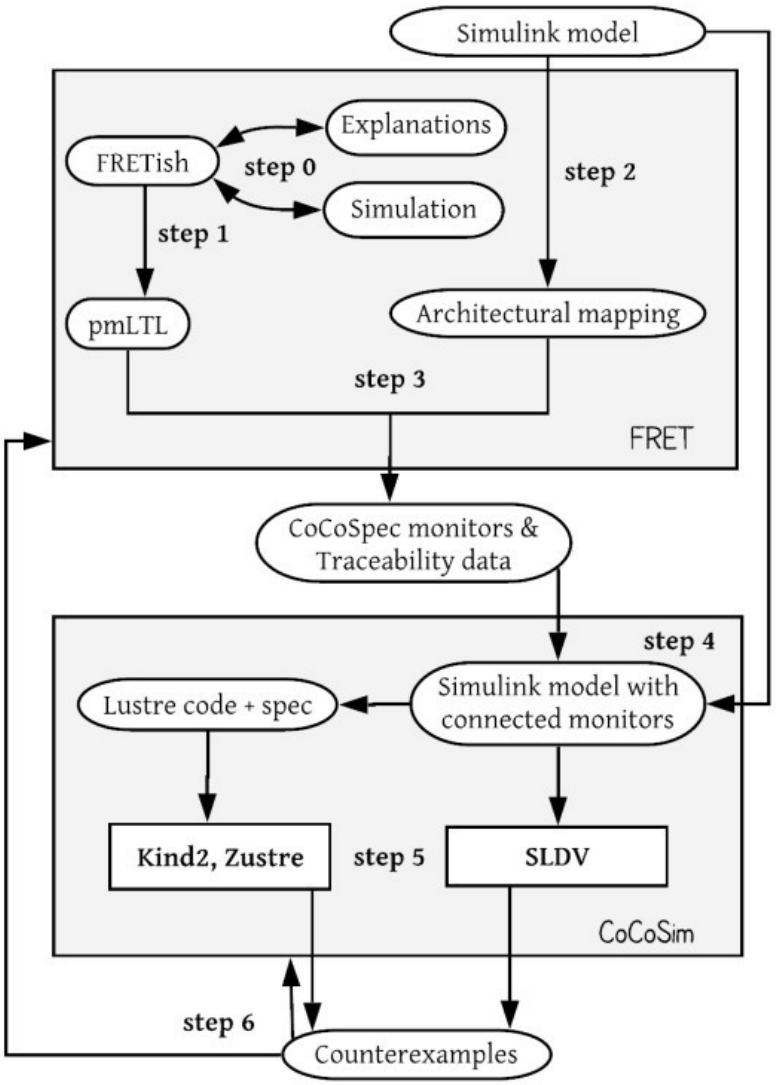


Figure 3: Cadre d'analyse des besoins.

sont importés dans CoCoSim [15]. CoCoSim génère des moniteurs Simulink et produit du code Lustre pour le modèle complet (modèle initial plus moniteurs attachés).

Étape 5 : Le modèle cible est analysé à l'aide d'outils de vérification basés sur Simulink (comme Simulink Design Verifier) et basés sur Lustre (comme Kind2 et Zustre).

Étape 6 : Les contre-exemples produits par l'analyse peuvent être retracés jusqu'à CoCoSim ou FRET.

Analyse - Cas d'utilisation et exigences sélectionnés

Notre étude de cas implique les tâches suivantes : solliciter les exigences dans fretish, établir la correspondance entre les variables fretish et les variables du modèle, effectuer une analyse,

interpréter les contre-exemples au niveau des exigences et interpréter les exigences au niveau du modèle. Trois chercheurs ont participé à l'étude : un ingénieur en contrôle en tant qu'expert du domaine, un expert en exigences et un expert en vérification. Les tâches 1 et 2 ont été réalisées en collaboration par les experts en exigences et en domaine. L'expert en vérification a effectué les tâches 3 et 5, tandis que l'expert en exigences a réalisé la tâche 4.

Leçons apprises

Nous avons réussi à capturer 69 des 74 exigences LMCPS dans FRET. Cependant, certaines exigences ont posé des problèmes, notamment celles contenant des conditions temporelles ou faisant référence à des valeurs précédentes de variables. Nous avons contourné ces limitations en utilisant des variables internes/auxiliaires, mais une solution directe dans fretish serait préférable.

Nous avons généré des spécifications et des données de traçabilité pour tous les défis LMCPS, ce qui nous a permis de créer automatiquement des moniteurs pour les modèles Simulink. La capacité d'interpréter et de tracer des contre-exemples aux niveaux du modèle et des exigences s'est révélée très utile pour détecter les conflits et comprendre les problèmes. Le simulateur FRET a également été précieux pour l'analyse des contre-exemples. Pour gérer la complexité, nous avons effectué des analyses modulaires sur les exigences locales plutôt que globales. Notre approche de mappage architectural nous a permis de déployer des spécifications CoCoSpec à différents niveaux de comportement du modèle, ce qui s'est révélé essentiel pour les modèles complexes.

Navigation du Rover

Dans Section 4.2, nous proposons une méthodologie détaillée pour l'étude de cas du Rover d'Inspection, qui comprend une sélection spécifique d'outils. L'outil principal utilisé dans les étapes conceptuelles, de conception et d'assurance au niveau du système est AdvoCATE. Les outils FRET, CoCoSim et Event-B sont utilisés pour l'application des méthodes formelles. Dans la phase d'analyse, nous effectuons une analyse compositionnelle au niveau du système en utilisant CoCoSim et Kind2, ainsi qu'une vérification des composants par rapport au modèle système avec les outils Event-B et Kind2.

Chapitre 5: Conclusion et perspectives

Cette thèse de doctorat a abordé la vérification formelle des modèles Simulink/Stateflow en quatre chapitres.

Le premier chapitre donnait un aperçu général du problème et discutait des travaux connexes dans le domaine de la vérification formelle des modèles Simulink/Stateflow. Dans le deuxième chapitre, nous avons proposé une approche de traduction des modèles Simulink/Stateflow vers le langage formel Lustre. Nous avons également développé une compilation inverse des modèles Lustre vers CoCoSim, permettant de valider l'outil CoCoSim et de relier les exigences à l'analyse des modèles Simulink.

La mise en œuvre de cette approche s'est concrétisée dans la boîte à outils CoCoSim présentée dans le troisième chapitre. CoCoSim vise à faciliter les activités de vérification et de validation des modèles Simulink. La boîte à outils est hautement automatisée et dispose d'une architecture personnalisable et configurable pour intégrer d'autres techniques et améliorer la scalabilité.

Le quatrième chapitre a illustré l'utilisation de CoCoSim à travers plusieurs études de cas industriels et académiques, dont les défis CyberPhysical de Lockheed Martin et l'étude de cas du rover de navigation. Les résultats montrent que CoCoSim peut vérifier efficacement les modèles Simulink, notamment grâce à la vérification compositionnelle. Les méthodes formelles, telles que la vérification de modèles, peuvent être limitées pour les modèles volumineux, non linéaires et nécessitant des calculs numériques intensifs.

Plusieurs pistes de recherche future sont envisageables, notamment l'amélioration de la boîte à outils pour prendre en charge davantage de fonctionnalités de Simulink/Stateflow, la vérification des filtres numériques et des contrôleurs, l'intégration de la précision des nombres flottants et la compilation précise des calculs numériques, ainsi que l'étude de l'utilisation de CoCoSim dans des études de cas industriels plus poussés.

Introduction

In the development cycle of critical systems, performing early verification and validation (V&V) can help reduce the cost and time of detecting and fixing errors. Thus, performing V&V at the design level helps eliminate potential problems before the software is fully implemented. Our objective is to enable the verification of Simulink (a graphical dataflow modeling language widely used in the design of flight control systems) models with respect to formal properties that represent system requirements. The primary outcome of this Ph.D. work is the CoCoSim toolbox: an open-source framework for specifying and verifying user-defined requirements on Simulink models. The open architecture of the tool enables the integration of multiple analyses (ours and promising ones of the research community, for instance) to truly enable the application of formal verification methods on Simulink/Stateflow models.

Model Driven Development (MDD) is a widely used technique in developing Cyber-physical systems, specifically for embedded systems. Using MDD techniques helps refine High-Level Requirements down to the embedded code while having an executable model at different stages; this enables applying the V&V technique in all development stages of the system. Moreover, early error detection at the design level helps reduce the time and cost of fixing errors, which is usually very high when left to the late stages of system development. Some of the programming languages that support MDD are synchronous languages. This is because they aim to specify the behavior of synchronous reactive systems. This model of computation describes programs that intend to be run forever, repeating the exact computation regularly, at fixed time steps.

Among the varieties of synchronous languages, we focus mainly on two of them. First, Simulink/Stateflow[1] from MathWorks is a *de facto* standard in the industry; it is supported by powerful MATLAB toolboxes, providing a large set of advanced mathematical functions. One of the most vital points of Simulink is the capability of specifying both continuous and discrete components in the same model. Eventually, continuous components will be discretized before being executed on the final embedded platform, they are essential ingredients of the development process of controllers. The Simulink framework also provides simulation engines enabling the execution of model traces combining discrete components with continuous ones specified with Ordinary Differential Equations (ODEs). In addition, the discrete subset of Simulink can be used to automatically produce embedded code, thus accelerating code development while minimizing the introduction of errors during the design.

The second industrial language is ANSYS SCADE[2] and its associated academic companion Lustre [29]. While SCADE resembles the discrete subset of Simulink with similar graphical objects to describe model components, Lustre is a textual, yet equivalent, language.

Simulink and SCADE are used as MDD tools in developing critical systems in many safety-critical areas, such as automotive and avionics industries, where errors are not allowed since they can have catastrophic consequences. Simulink and SCADE offer code generation capabilities and means to perform verification and validation (V&V). On the one hand, exhaustive testing of such complex systems is never possible. On the other hand, formal verification techniques

[1] https://www.mathworks.com/products/simulink.html
[2] http://www.esterel-technologies.com/products/scade-suite/

are complete (they perform exhaustive verification) as they show the correctness of a model for all possible inputs. Therefore using formal verification techniques is highly desirable.

At a research level, the topics of certified compilation or formal verification of dataflow languages are very active. Nevertheless, integrating academic research ideas into commercial frameworks such as Simulink and SCADE is quite tricky. Therefore, it is more common to develop methods and tools on well-defined and publicly available languages such as Lustre.

The motivation behind this work is to enable the application of research-level prototypes to actual models designed by control engineers in Simulink, thus bridging the gap between academic research and industrial use. CoCoSim addresses the translation of a large subset of Simulink discrete blocks into Lustre and provides means to display and understand the results of the analyses within MATLAB Simulink. We have applied CoCoSim to many case studies of moderate sizes. We believe that it is powerful enough for genuine industrial Simulink models. However, we want it to be an open-source toolbox so that the community can devise and demonstrate even more powerful (in terms of scalability or applicability) analyses. The motivation is to illustrate and support the practical application of formal methods on Simulink models.

This Ph.D. work has resulted in the following publications:

Conference papers:

1. Hamza Bourbouh et al. "Automated analysis of Stateflow models". In: *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Ed. by Thomas Eiter and David Sands. Vol. 46. EPiC Series in Computing. EasyChair, 2017, pp. 144–161

2. Hamza Bourbouh et al. "CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models". In: *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. 2020

3. Hamza Bourbouh et al. "Integrating Formal Verification and Assurance: An Inspection Rover Case Study". In: *NASA Formal Methods*. Ed. by Aaron Dutle et al. Cham: Springer International Publishing, 2021, pp. 53–71

4. Hamza Bourbouh, Guillaume Brat, and Pierre-Loïc Garoche. "CoCoSim: an automated analysis framework for Simulink/Stateflow". In: *Model Based Space Systems and Software Engineering-European Space Agency Workshop (MBSE 2020)*. 2020

5. Anastasia Mavridou et al. "Bridging the Gap Between Requirements and Simulink Model Analysis". In: *Joint 26th International Conference on Requirements Engineering: Foundation for Software Quality Workshops, Doctoral Symposium, Live Studies Track, and Poster Track*. Pise, Italy, Mar. 2020

6. A. Mavridou et al. "The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained". In: *International Requirements Engineering Conference (RE)*. IEEE, 2020, pp. 300–310

Journals:

7. Hamza Bourbouh et al. “From Lustre to Simulink: Reverse Compilation for Embedded Systems Applications”. In: *ACM Transactions on Cyber-Physical Systems* 5.3 (2021), pp. 1–20

Technical reports:

8. Hamza Bourbouh et al. *Integration and Evaluation of the Advocate, FRET, CoCoSim, and Event-B Tools on the Inspection Rover Case Study.* Tech. rep. 2020

9. Anastasia Mavridou et al. *Evaluation of the FRET and CoCoSim tools on the Ten Lockheed Martin Cyber-Physical Challenge Problems.* Tech. rep. 84 pages. NASA, Aug. 2019

The report begins with a discussion of the context and related work, including formal representation of Simulink/Stateflow models and formal specification of properties. A bidirectional translation from Simulink/Stateflow is provided in Chapter 2. Chapter 3 describes the CoCoSim toolbox, designed to ease verification and validation (V&V) activities for Simulink models, including translation validation and connecting external tools. Finally, the report ends with some use cases demonstrating the capabilities of CoCoSim.

Chapter 1

Context and related works

Contents

This chapter briefly introduces some concepts and definitions necessary to understand the following chapters. In addition, some related works are cited here and in the following chapters.

1.1 Background

1.1.1 Simulink

Simulink [97], developed by MathWorks, is a graphical programming language for modeling dynamical systems, including discrete-time ones, i.e., synchronous dataflow systems. Simulink has gained popularity in critical embedded systems development. It supports the design and simulation of complex systems before automatically generating embedded C code. A Simulink model consists of a set of blocks connected by signals that can be organized as hierarchical models. For example, Fig. 1.1 illustrates a stopwatch example that measures the time elapsed between activation and deactivation. Two external signals control the stopwatch: a `toggle` signal to toggle the activation of the stopwatch and a `reset` signal to reset the counter.

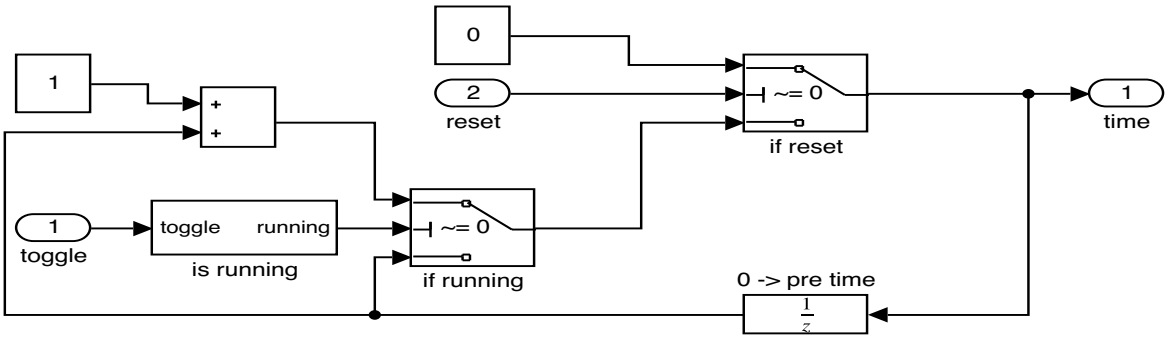


Figure 1.1: Stopwatch example in Simulink.

Simulink has a rich library of blocks and supports continuous and discrete solvers in its simulation engine. Furthermore, blocks can run on different sample times (multi-periodic) or on one global sample time (mono-periodic). On the other hand, Simulink lacks formally published reference semantics for its block library, making formal analysis of such models difficult.

In a Simulink model, the user can use blocks, signals, and annotations. Each Simulink block implements a given mathematical function or a stateful system specification and has a specific number of inputs and outputs connected with other blocks using Simulink signals. These blocks are either basic blocks from Simulink library (e.g., *Sum*, *UnitDelay*, *Gain*) or a grouping of several blocks into a *Subsystem* block. Simulink diagrams are hierarchical and graphically organized using subsystems. There are two types of subsystems; *Virtual Subsystem* is flattened during the simulation and only used to organize the model at design time. *Atomic Subsystem* is executed as an atomic unit in the final code.

A discrete Simulink model runs on a fixed time step defined with a period T and initial offset T_0. A Simulink model can be mono-periodic, where all blocks run with the same period, or multi-periodic, where blocks can run on different periods. A block is executed, and its outputs are updated only when certain execution conditions are satisfied. If these conditions are not met, the block is not executed, and its output signals hold their values. Matlab/Simulink

defines three types of execution conditions and blocks:

- Unconditional blocks or subsystems: The block is set with a sample time $D = (T^B, T_0^B)$ with period T^B and initial offset T_0^B and it is updated only at times $kT^B + T_0^B$ for $k \in \mathbb{N}$, whereas it remains constant during the intervals $[kT^B + T_0^B, (k+1)T^B + T_0^B)$.
- Conditionally executed subsystems: such as *Enabled/Triggered Subsystem*s, are subsystems that are conditionally executable when a certain guard condition over certain variables, called control inputs, holds. Furthermore, the data of the system-block may be reset when the guard condition holds, Moreover, the outputs may be reset when the guard condition is not valid anymore.
- Logically-executed subsystems: the Subsystem is executed one or more times at the current time step when a signal from a control block enables it. Examples of logically-executed subsystems are *If Action Subsystem*, *Switch Case Action Subsystem*, and *For Each Subsystem*.

In the following, we rely on Zhou and Kumar [139] formal representation of Simulink diagrams and Simulink blocks. This formal representation is used later in Chapter 2 to provide a formal translation from Simulink models to Lustre. Even though Zhou and Kumar's representation can represent both continuous-time and discrete-time blocks, we are only interested in the latter.

1.1.1.1 Unconditional atomic blocks

Simulink provides a library of basic blocks that can be composed into more complex Simulink diagrams. Such blocks are referred to as atomic-blocks.

Based on the formalization of Zhou and Kumar (2012) [139] an atomic-block can be defined as follows:

Definition 1.1 *An atomic Simulink block B can be represented as a tuple $(\mathcal{U}^B, \mathcal{Y}^B, \mathcal{D}^B, \mathcal{D}_0^B, \{(G_i^B, f_i^B, h_i^B)\}_{i=1}^{q^B}, (T^B, T_0^B))$, where*

- $\mathcal{U}^B = \mathcal{U}_1^B \times \ldots \times \mathcal{U}_{m^B}^B$ *is the set of typed inputs,*
- $\mathcal{Y}^B = \mathcal{Y}_1^B \times \ldots \times \mathcal{Y}_{p^B}^B$ *is the set of typed outputs,*
- $\mathcal{D}^B = \mathcal{D}_1^B \times \ldots \times \mathcal{D}_{n^B}^B$ *is the set of typed data,*
- $\mathcal{D}_0^B \subseteq \mathcal{D}^B$ *is the set of initial data conditions,*
- $\{(G_i^B, f_i^B, h_i^B)\}_{i=1}^{q^B}$ *is a set of triples, where*
  - $G_i^B \subseteq \mathcal{D}^B \times \mathcal{U}^B$ *is a predicate representing an enabling guard, such that* $\vee_{i=1}^{q^B} G_i = True$.
  - $f_i^B : \mathcal{D}^B \times \mathcal{U}^B \to \mathcal{D}^B$ *is a data-update function,*
  - $h_i^B : \mathcal{D}^B \times \mathcal{U}^B \to \mathcal{Y}^B$ *is an output-assignment function.*
- T^B *is the sample period, and* T_0^B *is an offset (it is assumed zero by default if unspecified).*

The kth sampling time occurs at $kT^B + T_0^B$. The value of the input signal at the kth sampling time is denoted as $u(k) = (u_1(k), \ldots, u_{m^B}(k)) \in \mathcal{U}^B$, and similarly for other signals. At the kth sampling time, if the data $d(k)$ and the input $u(k)$ are such that $G_i^B(d(k), u(k))$ holds, the next data, $d(k+1) = f_i^B(d(k), u(k))$, is computed, and the output value is assigned to $y(k) = h_i^B(d(k), u(k))$.

(Zhou and Kumar, 2012 [139]).

Let us illustrate Definition 1.1 by defining the atomic-blocks used in the stopwatch SIMULINK model of figure 1.1.

Example 1.1
The stopwatch SIMULINK model defined in figure 1.1 uses several atomic-blocks, such as: Constant, Inport, Outport, Sum, Switch, and Unit Delay.
We consider that all SIMULINK blocks of the stopwatch model run on the same sample time as the model itself named $(T^{Stopwatch}, T_0^{Stopwatch})$.

A Constant block with value v models the relation $y(k) = v$, where y is its output and k represents the kth sampling time. Furthermore, it is a stateless block, so no data or data-update functions exist. The Constant block can be represented as:

$$(-, y, -, -, \{(-, -, y(k) = v)\}, (T^{Constant}, T_0^{Constant})),$$

where $T^{Constant}$ is the sample-period and $T_0^{Constant}$ is an offset. In the case of a Constant block, the sample time specifies the period between which the block output may change during simulation (for example, due to tuning the Constant value parameter). In the case of formal verification, We assume that the output of the block will never change (i.e., $T^{Constant} = infinity$ and $T_0^{Constant} = 0$).

Inport blocks link signals from outside a system into the system. It models the relationship $y(k) = u(k)$, where y is its output, u is its input provided outside the system, and k represents the kth sampling time. Sample time $(T^{Inport}, T_0^{Inport})$ can be used to specify the discrete interval between sample time hits. In this example, we assume the externally linked signal will run on the same sample time as the model. Therefore, the block can be represented as:

$$(u, y, -, -, \{(-, -, y(k) = u(k))\}, (T^{Stopwatch}, T_0^{Stopwatch})),$$

Outport blocks link signals from a system to a destination outside the system. They can connect signals flowing from a subsystem to other parts of the model. They can also supply external outputs at the top level of a model hierarchy. It can be represented similarly to the Inport block above:

$$(u, y, -, -, \{(-, -, y(k) = u(k))\}, (T^{Stopwatch}, T_0^{Stopwatch})),$$

Where u is its input and y is its virtual output linked to the external system.

The Sum block performs addition or subtraction on its inputs. Depending on its configuration, the block can add or subtract scalar, vector, or matrix inputs. It can also collapse the

elements of a signal and perform a summation.
In this specific example of the stopwatch model, the Sum block accepts two scalar inputs $u1$ *and* $u2$*, and generates its output* y *where* $y(k) = u1(k) + u2(k)$ *and* k *represents the* k*th sampling time. Specifying sample time is not recommended for the Sum block because it causes sample rate transition implicitly introduced by* SIMULINK *to change the block's algorithm. This mixing can often lead to ambiguity and confusion in* SIMULINK *models. The Sum block in the stopwatch model can be represented as:*

$$((u1, u2), y, -, -, \{(-, -, y(k) = u1(k) + u2(k))\}, -),$$

The Unit Delay block delays its input by one step. It models the relations, $d(k+1) = u(k)$ *with* $d(0) = d_0$*, and* $y(k) = d(k)$*, where* u *is its input,* d *is its data,* y *is its output,* d_0 *is the initial data condition, and* k *represents the* k*th sampling time. Thus, the Unit Delay block can be represented as:*

$$(u, y, d, d_0, \{(-, d(k+1) = u(k), y(k) = d(k))\}, (T^{Stopwatch}, T_0^{Stopwatch})),$$

The Switch block combines multiple signals into a single stream. It passes through the first or third input signal based on the value of the second input. The Switch block does not have a Sample time parameter.
The Switch block used in the stopwatch model can be represented as:

$$((u1, u2, u3), y, -, -, \{(u2 == True, -, y(k) = u1(k)), (u2 == False, -, y(k) = u3(k))\}, -),$$

Zhou and Kumar (2012) introduced the following concepts for the computation of an atomic block over sample times.

Definition 1.2 *Given an atomic-block* B *and an input* $u \in \mathcal{U}^B$*, we call the computation of the corresponding output* $y \in \mathcal{Y}^B$ *a* ***step of*** B ***over*** u*.* y *is called the* ***output of a step of*** B ***over*** u*. Given an input sequence* $\{u(k)\}_{k=0}^K$*, a* ***step-trajectory*** *of* B *over* $\{u(k)\}_{k=0}^K$ *is a sequence of steps of* B*, where the* k*th step* $(0 \leq k \leq K)$ *in the sequence is over the input* $u(k)$*. Letting* $y(k)(0 \leq k \leq K)$ *denote the output of* B *over* $u(k)$*,* $\{y(k)\}_{k=0}^K$ *is called the* ***output of step-trajectory*** *of* B *over* $\{u(k)\}_{k=0}^K$*. (Zhou and Kumar, 2012 [139]).*

1.1.1.2 System-Blocks: Simulink Diagrams

A Simulink diagram, also known as a system-block, may be produced by recursively connecting atomic-blocks and other simpler system-blocks. A block's output can be linked to the input of different blocks, including its own. Zhou and Kumar (2012) define such connection as the following:

The connections over a set of system-blocks $\mathcal{B}$ can be represented using a relation

$C \subseteq (\mathcal{B} \times N)^2$, where N denotes the set of port numbers. A connection $c = ((B_1, i), (B_2, j)) \in C$ connects the output port i of system-block B_1 to an input port j of system-block B_2. The "C-connected $\mathcal{B}$ system" thus formed is denoted $\mathcal{B}/C$. Note a possible choice for connections is the "null-connection", and $\{B\}/\varnothing = B$. (Zhou and Kumar, 2012 [139]).

An unconditional subsystem is a "C-connected $\mathcal{B}$ system", denoted $\mathcal{B}/C$ where $\mathcal{B}$ is a set of system-blocks and $C \subseteq (\mathcal{B} \times N)^2$ is a set of interconnections. Grouping blocks into subsystems help in the formation of a hierarchical block diagram, with a subsystem block on one layer and the blocks that comprise the Subsystem on another.

Conditionally and logically executed subsystems are system-blocks that can be made conditionally executable when a certain guard condition over certain variables, called control inputs, holds. Furthermore, the data of the system-block may be reset when the Subsystem is re-enabled, and the outputs may be reset when the guard condition is disabled. Zhou and Kumar (2012) define such system-block as the following:

Definition 1.3 *Given a system-block B, a* conditioning over B *is 5-tuple* $\theta := (\mathcal{U}^\theta, G^\theta, f^\theta, h^\theta, (T^\theta, T_0^\theta))$, *where*

- $\mathcal{U}^\theta = \mathcal{U}_1^\theta \times \ldots \times \mathcal{U}_{m^B}^\theta$ *is the set of conditioning-inputs (also called control-inputs),*
- $G^\theta \subseteq \mathcal{U}^\theta$ *is a condition (predicate) over* $\mathcal{U}^\theta$,
- $f^\theta : \mathcal{D}^B \to \mathcal{D}^B$ *is a data-resetting function.*
- $h^\theta : \mathcal{Y}^B \to \mathcal{Y}^B$ *is an output-resetting function, and*
- T^θ *is a sample period,* T_0^θ *is an offset.*

When G^θ holds, B computes; otherwise, h^θ assigns the output. Also, when G^θ becomes true, the first computation of B is preceded by a data update by f^θ. The "θ-conditioned B" system thus formed is denoted $B \Downarrow \theta$. The conditioning can be implemented by placing a system-block inside a certain Subsystem *block (of Simulink Library) which can be configured to specify the conditioning parameters. Note a possible choice for conditioning is "null-conditioning", denoted $\perp := (-, True, id, id, \infty)$, in which case $B \Downarrow \perp = B$. (Here, id denotes the identity function.)*

(Zhou and Kumar, 2012 [139]).

Based on the previous definitions of atomic-blocks, "C-connected set of system-blocks $\mathcal{B}$," and "θ-conditioned system-block B." Zhou and Kumar formally define the class of SIMULINK diagrams (also referred to as system-blocks) as the following:

Definition 1.4 *A class of SIMULINK diagrams (also called system-blocks) is recursively defined as follows:*

1. *B is an atomic-block, then B is a system-block.*
2. *$\mathcal{B}$ is a set of system-blocks, $C \subseteq (\mathcal{B} \times N)^2$ is a set of interconnections, then "C-connected $\mathcal{B}$", denoted $\mathcal{B}/C$, is a system-block.*

3. *B is a system-block and θ is a conditioning over B, then θ-conditioned B, denoted $B \Downarrow \theta$, is a system-block.*

(Zhou and Kumar, 2012 [139]).

Remark 1.1 *For a system-block $\overline{B} := \mathcal{B}/C$, we have:*

- *inputs $\mathcal{U}^{\overline{B}} = \prod_{\nexists((.,.),(B,i)))\in C} \mathcal{U}_i^B$,*
- *outputs $\mathcal{Y}^{\overline{B}} = \prod_{B\in\mathcal{B}} \mathcal{Y}^B$,*
- *data $D^{\overline{B}} = \prod_{B\in\mathcal{B}} D^B$,*
- *initial data $D_0^{\overline{B}} = \prod_{B\in\mathcal{B}} D_0^B$,*
- *sample-period $T^{\overline{B}}$ and offset $T_0^{\overline{B}}$ are ensuring that each computation of each system-block $B \in \mathcal{B}$ coincides with some computation of the connected system-block $\overline{B}$ (i.e., no computation of any system-block is missed).*

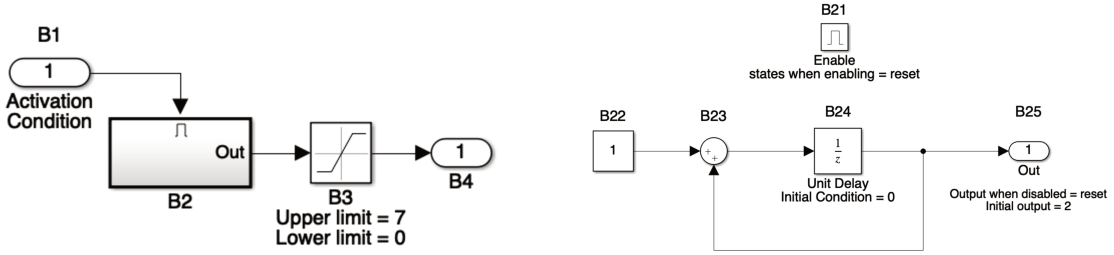
(Zhou and Kumar, 2012 [139]).

Remark 1.2 *For a system-block $\overline{B} := \mathcal{B} \Downarrow \theta$, we have:*

- *inputs $\mathcal{U}^{\overline{B}} = \mathcal{U}^B \times \mathcal{U}^\theta$,*
- *outputs $\mathcal{Y}^{\overline{B}} = \mathcal{Y}^B$,*
- *data $D^{\overline{B}} = D^B$,*
- *initial data $D_0^{\overline{B}} = D_0^B$,*
- *sample-period $T^{\overline{B}} = \begin{cases} T^\theta & \text{when } T^\theta \text{ specified} \\ T^B & \text{otherwise} \end{cases}$,*

  and offset $T_0^{\overline{B}} = \begin{cases} T_0^\theta & \text{when } T^\theta \text{ specified} \\ T_0^B & \text{otherwise} \end{cases}$.

  Note by the Simulink grammar, for a system-block $\overline{B} := \mathcal{B} \Downarrow \theta$, T^θ is either specified and in which case $T^{\overline{B}}$ is inherited to be T^θ, or is unspecified and in which case $T^{\overline{B}}$ is inherited to be T^B. Similarly for $T_0^{\overline{B}}$. (Zhou and Kumar, 2012 [139]).



(a) The root level of the Simulink diagram.

(b) The content of the enabled Subsystem B2.

Figure 1.2: Example of an enabled counter in Simulink.

Example 1.2
Fig. 1.2 illustrates an example of a Simulink *diagram* B *of a counter. The counter is defined inside the* Enabled Subsystem $B2$*, the latter is controlled by* Inport *block* $B1$*. The* $B2$ *Subsystem is executed only when the control input from* $B1$ *is positive. The counter is incremented by one while* $B2$ *is enabled. The* Unit Delay *block* $B24$ *is the only stateful block inside* $B2$*. When* $B2$ *is disabled, the memories of the* Unit Delay *block* $B24$ *will be reset in the next activation since the parameter "states when enabling" of* $B21$ *is set to "reset." Also, the* Outport *block* $B25$ *is set to "reset" when disabled; therefore, the counter will be set to the initial output of* $B25$ *when it is disabled, and it will reset its memories (i.e.,* Unit Delay *memories) when restarted. The* Saturation *block* $B3$ *limits the value of the counter to* $[0, 7]$*. The* Enable Port *block* $B21$ *is used to configure the* Enabled Subsystem $B2$ *parameters such as "states when enabling" and "Sample time". We assume all blocks are running with the same sample period* $T = 1$*.* B *conforms to the* Simulink *diagram class stated in Definition 1.4:* $B = \{B1, B2, B3, B4\}/C1$*, where:*

- $C1 = ((B1, .), (B2, .)), ((B2, .), (B3, .)), ((B3, .), (B4, .))$,
- $B2 = (\{B22, B23, B24, B25\}/C2) \Downarrow \theta$*, where*
  - $C2 = ((B22, .), (B23, .)), ((B23, .), (B24, .)), ((B24, .), (B23, .)), ((B24, .), (B25, .))$,
  - $\theta = (\mathcal{U}^\theta, u^\theta(k) > 0, d(k) = d_0, (y_{22}(k), y_{23}(k), y_{24}(k), y_{25}(k)) = (-, -, -, y_{25_0}), -)$
- $B1, B22, B23, B24, B25, B3, B4$ *are atomic-blocks*

1.1.2 Stateflow

The widespread deployment of cyber-physical systems in safety-critical scenarios like automotive, avionics, and medical devices, has made formal and automated analysis of such systems necessary. This is witnessed by the sheer number of comprehensive approaches proposed in the verification community. Stateflow [96] is a widely used modeling framework for embedded and cyber-physical systems where control software interacts with physical processes. Specifically, Stateflow is a toolbox developed by MathWorks Inc. that extends Simulink with an environment for modeling and simulating reactive systems. A Stateflow diagram can be included in a Simulink model as one of the blocks interacting with other Simulink components using input and output signals. Stateflow is a highly complex language with no formal semantics provided as a reference by the tool provider. Its semantics is only described through examples on the MathWorks website [96] without any formal definition. Hamon and Rushby have provided operational and denotational semantics for Stateflow [77, 78, 79]. A Stateflow diagram has a hierarchical structure, which can be either arranged in *parallel* in which all states become active whenever the diagram is activated; or *sequentially*, in which states are connected with transitions and only one of them can be active. We use a syntactic representation of Stateflow models described by the grammar presented in Table 1.1.

Syntax. A program $(s, src_{i\in[1\ldots n]})$ is composed of state definitions $s : sd$ and junctions $j : T$, with a main node s. A single state

$$
\begin{array}{rcl}
P & ::= & (s, [src_0, \ldots, src_n]) \\
src_i & ::= & s : sd \mid j : T \\
sd & ::= & ((a_e, a_d, a_x), T_o, T_i, C) \\
C & ::= & Or\,(T, [s_0, \ldots, s_n]) \\
 & & \mid And\,([s_0, \ldots, s_n]) \\
t & ::= & (e, c, (a_c, a_t), d) \\
T & ::= & \varnothing \mid t.T \\
d & ::= & p \mid j \\
p & ::= & \varnothing \mid s.p
\end{array}
$$

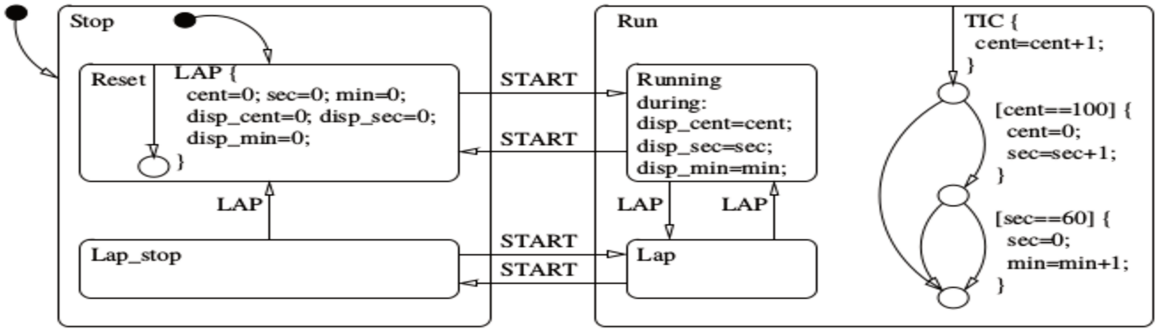


Figure 1.3: Stopwatch STATEFLOW model.

$$
\begin{array}{rl}
\mathit{main.run.running}: & ((\varnothing_a, \texttt{disp} = (\texttt{cent}, \texttt{sec}, \texttt{min}), \varnothing_a)\,, \\
& [(\mathit{START}, \mathit{true}, \varnothing_a, \varnothing_a, P\ \mathit{main.stop.reset})\,; \\
& (\mathit{LAP}, \mathit{true}, \varnothing_a, \varnothing_a, P\ \mathit{main.run.lap})], [], \mathit{Or}\,([],)) \\
\mathit{main.run.lap}: & ((\varnothing_a, \varnothing_a, \varnothing_a)\,, \\
& [(\mathit{START}, \mathit{true}, \varnothing_a, \varnothing_a, P\ \mathit{main.stop.lap\_stop})\,; \\
& (\mathit{LAP}, \mathit{true}, \varnothing_a, \varnothing_a, P\ \mathit{main.run.running})], [], \mathit{Or}\,([],)) \\
\mathit{main.run}: & ((\varnothing_a, \varnothing_a, \varnothing_a)\,, [], \\
& [(\mathit{TIC}, \mathit{true}, \texttt{cent} = \texttt{cent} + 1, \varnothing_a, J\ j1)], \mathit{Or}\,([], \{\mathit{running};\ \mathit{lap}\})) \\
j1: & [(\mathit{noevent}, \texttt{cent} == 100, \texttt{cont} = 0;\ \texttt{sec} = \texttt{sec} + 1, \varnothing_a, J\ j2)\,; \\
& (\mathit{noevent}, \texttt{cent!} = 100, \varnothing_a, \varnothing_a, J\ j3)] \\
j2: & [(\mathit{noevent}, \texttt{sec} == 60, \texttt{min} = \texttt{min} + 1, \varnothing_a, P\ \mathit{main.run})\,; \\
& (\mathit{noevent}, \texttt{sec!} = 60, \varnothing_a, \varnothing_a, J\ j3)] \\
j3: & []
\end{array}
$$

Figure 1.4: Encoding of the Stopwatch STATEFLOW model using the syntax from Table 1.1.

is defined by the *entry*, *during* and *exit* actions (a_e, a_d, a_x), *outer* and *inner* transitions T_o and T_i, as well as component content C. A component content C is either an $Or(T, sl)$ state with initializing transitions T and substates sl, or an $And(sl)$ state where all sl substates are run in parallel. A transition list is an ordered sequence of transitions. Each transition is associated with an event, a condition, side effect condition actions a_c, transition actions a_t and a destination d. Destinations could be either states or junctions, leading to further transitions.

Modeling Stopwatch example. Let us look at a concrete example that illustrates different STATEFLOW constructs. Figure 1.3 shows the `Stopwatch` model capturing a basic behavior of a stopwatch. This system interacts with its environment via three signals: *START*, *LAP* and *TIC*. *TIC* models the time increment of the system. *START* and *LAP* events model the user interaction with the device. There are two top-level states *Run* and *Stop*, and 4 inner states *Reset*, *Lap_stop*, *Running* and *Lap*. The device is initialized in *Stop* mode and switches back and forth between *Stop* and *Run*. When in *Running* state, it can be paused using the *LAP* signal. Each *TIC* signal received when in *Run* state increments the timer, performing side effects on the internal timer variables (i.e., *cent, sec, min*). This specific behavior is modeled by a set of

transitions using only junctions and starting with an inner transition. This model is interesting since it relies on multiple constructs of STATEFLOW: *inner* and *outer* transitions, *junctions* and hierarchical states.

Figure 1.4 describes the Stopwatch model (Fig. 1.3) using the syntax from Table 1.1. Transitions from a state s to another s' are defined as outer T_o and inner transitions T_i of node s, depending on the target node. The transitions associated with a node content of type *Or* describe the initialization transitions when entering the node. Junctions enable the definition of transitions combining multiple conditions. The semantics of transitions is far from trivial: a transition path is evaluated one segment (atomic transition) after the other, performing side effects on the state via *condition actions* a_c on the go, while *transition actions* a_t are only performed when all conditions over the path are satisfied, and the path reaches a state, not a junction. If a given path is eventually fireable, the exit actions of the original node are performed, then the transition actions, to conclude with entry actions of the target node.

1.1.3 Lustre

Lustre [29] is a synchronous language for modeling systems of synchronous reactive components.

A LUSTRE program L is a collection of nodes $N_0, N_1, \ldots, N_m$. The nodes satisfy the grammar described in Figure 1.5 in which *td* denotes type constructors, including enumerated types, and value v either constants of enumerated types C or primitive constants such as integers i or reals r. Each node is declared by the grammar construct d of Figure 1.5 and is represented by the following tuple:

$$N_i = (\mathcal{I}_i, \mathcal{O}_i, \mathcal{L}_i, Eqs_i)$$

Where $\mathcal{I}_i, \mathcal{O}_i$, and $\mathcal{L}_i$ are sets of typed input, output, and local variables. Eqs_i represents the set of stream definitions defined as:

$$Eqs_i = \left\{ (v_i^j)_{1 \leq j \leq nb_i} = expr_i \right\}_{i \in \{0,\ldots,|Eqs_i|-1\}}$$

where $nb_i \in \mathbb{N}^*$ denotes the number of output variables defined by the expression $expr_i$, $v_i^j \in \mathcal{O}_i \cup \mathcal{L}_i$ and $expr_i$ is an expression where $Vars(expr_i) \subseteq \mathcal{O}_i \cup \mathcal{I}_i \cup \mathcal{L}_i$. $Vars(expr_i)$ is the set of variables in $expr_i$; and expressions $expr_i$ are arbitrary LUSTRE expressions, as presented in Figure 1.5 by constructor e, including node calls $N_j(u_1, \ldots, u_n)$.

LUSTRE code consists of a set of nodes transforming streams of input values into streams of output values. LUSTRE models are synchronous in the sense that the processing time of each component is neglected, and communication is assumed to be instantaneous [11]. A notion of symbolic "abstract" universal clock is used to model system progress. More materials on LUSTRE semantics can be found in [24, 29, 74].

Let us illustrate LUSTRE syntax on a possible model for the stopwatch example. The code is presented in Listing 1.1.

```
1 node count (tick : bool) returns (time : int);
2 let
```

$$
\begin{array}{rcl}
td & ::= & \textbf{type}\ enum\_ident\ = enum\ \{\ C_1, \ldots, C_n\} \\
bt & ::= & real\ \mid\ bool\ \mid\ int\ \mid\ enum\_ident \\
d & ::= & \textbf{node}\ f\ (p)\ \textbf{returns}\ (p);\ \ \textbf{vars}\ p\ \textbf{let}\ D\ \textbf{tel} \\
p & ::= & x : bt;\ ...;\ x : bt \\
D & ::= & pat = e;\ D\ \mid\ pat = e; \\
pat & ::= & x\ \mid\ (pat, ..., pat) \\
e & ::= & v\ \mid\ x\ \mid\ (e, ..., e)\ \mid\ e \rightarrow e\ \mid\ op(e, ..., e)\ \mid\ \textbf{pre}\ e \\
 & & \mid\ f(e, ..., e)\ \mid\ f(e, ..., e)\ \textbf{every}\ e \\
 & & \mid\ e\ \textbf{when}\ C(x)\ \mid\ \textbf{merge}\ x\ (C \rightarrow e)...(C \rightarrow e) \\
 & & \mid\ \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
v & ::= & C :: enum\_ident\ \mid\ i :: int\ \mid\ r :: real\ \mid\ true :: bool\ \mid\ false :: bool
\end{array}
$$

Figure 1.5: A subset of LUSTRE syntax

```
3   time = 0 -> pre time + 1;
4 tel
5
6 node stopwatch (tick, toggle, reset : bool) returns (time : int);
7 var running : bool clock;
8 let
9   running = ((false -> pre running) <> toggle) or reset;
10  time = merge running (true  -> count(tick when running) every reset)
11                       (false -> (0 -> pre time) when not running);
12 tel
```

Listing 1.1: The stopwatch example with clocks

Lines 1 to 4 of Listing 1.1 define a `count` node returning an integer stream representing the sequence of natural numbers. Primitive types like `bool`, `int`, or `real` are available. Note that, in general, a node may declare several output streams.

Line 6 declares a node named `stopwatch` that takes three boolean streams as parameters, namely `tick`, `toggle` and `reset` and declares a single integer stream as output, namely `time`.

In LUSTRE, a node is defined by a set of *stream equations* with possible local variables denoting internal flows. Stream equations are defined between the `let` and `tel` keywords. For instance, line 7 declares `running` as a local boolean flow.

When defining equations, regular arithmetic and comparison operators are lifted to streams and are evaluated at each time step. For instance, line 9 of Listing 1.1 defines the stream `running` as a disjunction of the `reset` input stream and the result of comparing two boolean streams: `false -> pre running`, a LUSTRE expression, and `toggle`, one of the input streams of the node. The temporal operator `pre`, for *previous*, enables a limited form of memory, allowing to read the value of a stream at the previous instant. The arrow operator allows to build a stream `c -> e` as the expression `e` while specifying the first value `c`. Therefore, the expression `false -> pre running` denotes a boolean stream whose first value is `false` and whose next values are the previous values of the `running` stream.

A node that relies on these constructs is considered as *stateful*; the values of the memories define its internal state. Without these temporal operators, nodes act as mathematical

functions.

Another specific construct of LUSTRE is the definition of clocks and clocked expressions. Clocks are defined as enumerated types, the simplest ones being boolean clocks. Expressions can then be clocked for such clock values. For instance, let us consider the expression `e when c` where `c` is a boolean clock, then the expression `e when c` is not defined when variable `c` is false.

Let us now explain the expressions in lines 10 and 11:

- `count(tick when running)` is a call to node `count` with argument `tick` clocked on the clock `running`. Therefore, this expression has no value when `running` is false. Notice that the `tick` parameter of the node `count` is unused in the definition: it is only used for clocking.
- `count(tick when running) every reset` is the previous call to node `count` completed with a reset expression `every reset`. This expression specifies that if boolean `reset` is true, then the call to `count` reinitializes the node to its initial state, and therefore the `time` stream to `0`. The local stream `running` is true whenever `reset` is true, therefore the node `count` is always executed when `reset` is true and the arrow operator will reset to its initial value 0.
- `(0 -> pre time) when not running` is an integer stream. It starts with value `0` and then uses the previous value of `time`. This expression is clocked on the negation of the `running` clock. Notice that it is defined iff the previous expression is not defined.
- lines 10 and 11 define a `merge` expression. It is used to create a flow clocked on a particular clock using expressions clocked on sub-clocks of this particular clock. Here, this means that:
  - `time` will be clocked on the base clock.
  - when `running` is true, the expression `count(tick when running) every reset` is used to define `time`. This expression must be clocked on `running`, which is trivially the case here, but we may have used an external expression clocked on `running` for instance.
  - when `running` is false, the expression `(0 -> pre time) when not running` is used to define `time` and is also trivially clocked on the negation of `running`.

Table 1.2 presents an example of an evaluation of several streams from the `stopwatch` node: first `running` is `false`, then becomes `true` when the `toggle` is `true` and becomes `false` when the `toggle` is `true` again (simulating the operator pressing the toggle button of the stopwatch). The `reset` parameter is always considered false in this example.

Finally, expressions associated with each clock case have to be clocked appropriately and the clocking phase of the compiler allows checking the consistency of clock definitions and uses, as would do a typing compilation phase.

Nodes and calls form a hierarchy of nodes comparable to the notion of *subsystems* in SIMULINK. Type and clock inferences guarantee at compile time that expressions and functions call respect their type constraints and properly rely on previous values to build current

`toggle`	false	true	false	false	false	false	true	false	false
`running`	false	true	true	true	true	true	false	false	false
`count(tick when running)`		1	2	3	4	5			
`(0 -> pre(time)) when not running`	0						5	5	5
`time`	0	1	2	3	4	5	5	5	5

Table 1.2: Evolution of expressions and variables using a clock in the stopwatch example

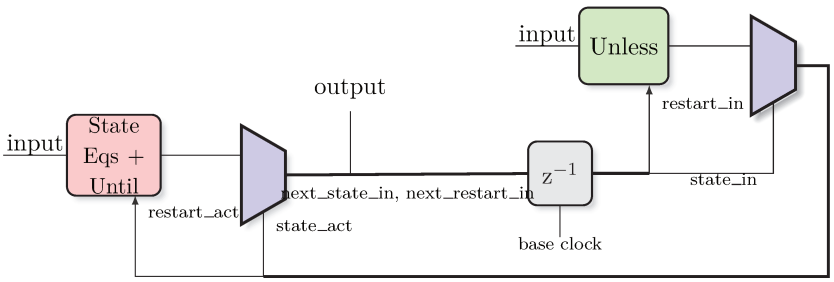


Figure 1.6: Automaton as a pure dataflow.

ones. For example, consider the following equations `x = f(y); y = g(x);`. These two equations create a causality problem and produce an *algebraic loop error*. However, notice that the same definitions with either `f(pre y)` or `g(pre x)` would be typable and accepted by the compiler.

Automata are supported since Lustre V6 [40, 39]. The overall behavior of an automaton is pictured in Fig. 1.6. An automaton consists of states, each with its own set of equations and possibly local variables. At each instant, two pairs of variables are computed: a putative *state_in* and an actual state *state_act* and also, for both states, two booleans *restart_in* and *restart_act*, that tell whether their respective state equations should be reset before execution. The actual state is obtained via an immediate (`unless`) transition from the putative state, whereas the next putative state is obtained via a normal (`until`) transition from the actual state. Only the actual state equations are executed at each instant. Finally, the `restart/resume` keyword switches drive a state reset function. A complete presentation of these constructs is given in [62].

1.2 Formal semantics of Simulink/Stateflow models

In this work, we limited ourselves to the Discrete Simulink blocks analysis. Our toolbox CoCoSim uses Lustre [29] as its intermediate language since its semantics are well-defined and have been studied in academia for many years. Moreover, Lustre is well suited for formal verification, and many model checkers accepting Lustre as an input have been developed using different verification techniques such as Kind2[32], Zustre[52], and JKind[59].

1.2.1 Discrete-time Simulink Semantic

The discrete subset of Simulink has been translated to various input languages for model checking purposes, such as NuSMV [102], Lustre [131], and hybrid automata [123]. Tripakis et al. (2005) provided the first work on translating Simulink to Lustre [131]. We share the same algorithm of recursively generating a Lustre node for each Subsystem, but we differ in how we translate some blocks and handle multi-periodic systems. Unfortunately, comparing the tool is impossible since its binary is not maintained and does not support newer versions of Simulink models. Zhou and Kumar (2012) proposed a recursive approach for translating a class of Simulink diagrams to input/output-extended finite automata (I/O-EFA) [139], the formal representation of the model is flattened and loses the hierarchy and architecture of the Simulink model. CoCoSim translates the Simulink model in a modular way. Each Simulink Subsystem is translated into a Lustre node. This modularity allows the tool to perform verification compositionally, a technique that helps us scale with large models. Meenakshi et al. (2006) [102] parse the textual representation of the model and use NuSMV input language as an intermediate formal language for verification using a symbolic model checker; the data types supported by the language are limited to Booleans and bounded integers. Validation of the translation from Simulink to NuSMV input language was not discussed.

Moreover, all of the previous works are either proprietary or not maintained, which causes problems in adapting to new Simulink versions. We addressed those two challenges by making CoCoSim an open architecture, so people can use, contribute to, or customize our tool. Furthermore, our CoCoSim translator uses the Simulink APIs to access and compute all model information needed for the translation instead of parsing the textual format of the model. Therefore, new Simulink releases are less likely to impact the translator negatively. In addition, the translator is modular, and new Simulink blocks can be easily supported and integrated.

1.2.2 Stateflow Semantic

Over the years, several approaches have been proposed to analyze Stateflow diagrams. Such approaches often lack one or many of the following desired features: (i) convincing formal semantics; (ii) a faithful compilation that preserves the hierarchical structure of the Stateflow model and last but not least, (iii) fully automated analysis engine; Our work aims to provide a framework for adequately addressing all those points. Our approach is based on a series of papers by Hamon [78, 77, 79] providing operational and denotational semantics for Stateflow, and designing interpreters for Stateflow. Our contribution is detailed in Section 2.3.

Despite its lack of formal semantics, Stateflow is widely used in the industry both for modeling purposes and code generation [107]. It has been studied for formal modeling and verification by numerous approaches, e.g., targeting automata [14, 91, 137], hybrid automata [7], process calculi such as CPS [35, 140], transition systems [107], tabular expressions [125], or the synchronous language Lustre [119]. As a general remark, each of these approaches restricts the considered language, e.g., on events, inner transitions, or junctions, and synthesizes an encoding in the target language, supported by verification tools or test case generation, independently of the code generated from the same model.

An approach related to ours is the work described by Scaife et al. (2004) in [119]. This approach translates a STATEFLOW model into LUSTRE dataflow language [29]. Since LUSTRE-like languages such as Scade [129] are used in the industry to generate embedded code, the approach is compatible with compilation and verification. The approach is, however, very limited concerning the full STATEFLOW semantics. A tool, `sf2lus`, is provided and performs the translation for the considered subset. This approach differs from ours in several ways: (i) Our translation keeps state machines' structure and, by consequence, makes it easy to read and trace state information; (ii) sf2lus mishandles events at the same time, which makes the behavior unsound wrt STATEFLOW events semantics. Events occur (and wake the chart) in an ascending order based on their port numbers once at a time. Such a chart can then be executed multiple times in the same time step; (iii) `sf2lus` does not support newer versions of STATEFLOW; and last (iv) our tool is developed in Matlab and is directly integrated into the SIMULINK/STATEFLOW environment, which allows, among other features, the use of the Matlab simulation tool to investigate failed properties further.

Another line of related works is developed in a series of papers by Hamon [77, 78] on which we based our work. In these papers, formal semantics for STATEFLOW is provided, either in operational or denotational flavors. Whalen (2010) [136] starts from Hamon's denotational semantics to define a structural operational semantics for three Statecharts languages, including STATEFLOW, but does not provide a compilation schema or analysis techniques. Tiwari (2002) [130] uses a translation of a fragment of STATEFLOW models to pushdown systems to generate and check invariants. To the best of our knowledge, until now, these are the only formal reference semantics available for STATEFLOW.

Last, SIMULINK Design Verifier (SLDV) is a toolbox provided by Mathworks to perform an automated formal analysis of SIMULINK/STATEFLOW models. However, SLDV is a commercially distributed tool; therefore, details on implementation and functionality are unavailable to the public.

1.3 Formal specification of properties

Programming languages could be fitted with specification languages, e.g., ACSL [10] for C or SPARK [3] for ADA. In the case of synchronous languages and models, multiple works [37, 92, 135] advocate for the use of component-attached requirements. However, notice that the actual definition of such contracts or their reasoning is still challenging. Formalized contracts can be used for many applications: test oracles, test synthesis, reactive synthesis [84, 86], compositional reasoning, or validation of individual contracts. CoCoSpec [31, 33] is a specification language for LUSTRE [73] and has been extended to SIMULINK models.

In [81], "An Axiomatic Basis for Computer Programming", HOARE defines deductive reasoning to validate code-level annotations. This paper introduces the concept of HOARE triple $\{Pre\}code\{Post\}$ as a way to express the semantics of a piece of code by specifying the post-conditions ($Post$) that are guaranteed after the execution of the code, assuming that a set of preconditions (Pre) was satisfied. HOARE supports a vision in which this axiomatic semantics is used as the "ultimately definitive specification of the meaning of the language [...], leaving certain aspects undefined. [...] Axioms enable the language designer to express its general

intentions simply and directly, without the mass of detail usually accompanying algorithmic descriptions." When this pair $(Pre, Post)$ is associated with a function, it can be interpreted as a function contract. In more general use of formal specification, the local reasoning about the function makes the assumption Pre, but the precondition has to be guaranteed when this function is called. Otherwise, the function is not fully specified, and its behavior is not defined.

This idea has been extended naturally to synchronous dataflow languages with the concept of synchronous observer [75, 134, 117]. A synchronous observer encodes a predicate corresponding to the postcondition of the Hoare triple. However, since the semantics are not expressed over values but streams, the principle of the Hoare triple has to be lifted to sequences of values.

$$\{Pre(state, inputs)\} node(in, out) \{Post(state, state', in, out)\}$$

means

$$\square \left(\bigwedge \begin{array}{l} \mathcal{H}(Pre(state, input)) \\ node(state, state', in, out) \end{array} \Rightarrow Post(state, state', in, out) \right).$$

with $\mathcal{H}(p) \triangleq$ { p has held since beginning }. The operator $\mathcal{H}$ can be defined in Lustre with the node `Sofar`:

```
node Sofar (in: bool) returns (out: bool);
let
  out = in -> pre out and in;
tel
```

Such a synchronous contract is active when, at a given time step, all the inputs and internal states, up to now, have satisfied the precondition. Moreover, it is valid if then the postcondition always applies.

In Lustre, recent works [30] proposed a dedicated language to annotate Lustre model with a rich specification. Figure 1.7 gives an example. The node `ml` represents the mode logic of an aircraft controller, deciding whether the autopilot is active or not. Its specification is described in a `contract`. This contract can bind new variables but, more importantly, can specify the precondition `altitude >= 0.0` for that contract. Two main postconditions are expressed as well as four different modes. Each of these modes is guarded by some conditions in the `require` expressions, while a conditional postcondition `ensure` is specified. Last, in the actual Lustre node, the contract is declared.

In other words, the synchronous observer acts as a description of axiomatic semantics for a synchronous model. The observer is defined in the same language as the model and corresponds to a set of boolean streams. If the property is valid, the output flow encoding the property should remain `true` during the program's execution.

For instance, a synchronous observer on the stopwatch example (Sec. 1.1.3) would be the assertion that the stream `time` always has a non-negative value. Listing 1.2 illustrates a simple CoCoSpec contract using the stopwatch node introduced earlier and specifying that the stream `time` always has a non-negative value given the assumption that `toggle` and `reset` cannot be pressed at the same time.

```
contract ml ( altRequest, fpaRequest, deactivate : bool ; altitude, targetAlt : real )
    returns ( altEngaged, fpaEngaged : bool ) ;
let
  var altRequested = switch(altRequest, deactivate) ;
  var fpaRequested = switch(fpaRequest, deactivate) ;
  var smallGap = abs(altitude - targetAlt) < 200.0 ;
  assume altitude >= 0.0 ;
  guarantee targetAlt >= 0.0 ;
  guarantee not altEngaged or not fpaEngaged ;
  mode guide210Alt ( require smallGap ; require altRequested; ensure altEngaged ; ) ;
  mode guide210FPA ( require smallGap ; require fpaRequested ;
                     require not altRequested; ensure fpaEngaged; ) ;
  mode guide180 ( require not smallGap ; require fpaRequested; ensure fpaEngaged; ) ;
  mode guide170 ( require not smallGap ; require altRequested ;
                  require not fpaRequested; ensure altEngaged ; ) ;
tel

node ml ( altRequest, fpaRequest, deactivate : bool ; altitude, targetAlt : real )
returns ( altEngaged, fpaEngaged : bool );
(*@contract import
     mlSpec ( altRequest, fpaRequest, deactivate : bool ; altitude, targetAlt : real )
     returns ( altEngaged, fpaEngaged : bool ); *)
let ... tel
```

Figure 1.7: Listing of CoCoSpec contracts at Lustre level using modes.

```
1 contract stopwatchSpec (toggle, reset : bool) returns (time : int);
2 let
3   -- we can assume that the two buttons are never pressed together
4   assume not (toggle and reset);
5   -- the elapsed time is always non-negative
6   guarantee time >= 0;
7 tel
```

Listing 1.2: The stopwatch CoCoSpec contract example

1.4 Conclusion

This chapter started with a background on Simulink, Stateflow, and Lustre, and then it discussed the context and related works concerning formal semantics of Simulink and Stateflow models and the formal specification of properties. It can be concluded that due to the lack of a formal semantic foundation, it is challenging to verify Simulink and Stateflow models automatically. While there have been efforts to develop formal semantics for Simulink and Stateflow, these semantics do not cover all the features of the models and can be challenging to use. Furthermore, there is also a lack of tools to support the automatic verification of Simulink/Stateflow properties specified as synchronous observers.

Chapter 2

Bidirectional translation of Simulink/Stateflow models to Lustre

Contents

This chapter presents our contribution to providing a bidirectional translation between Simulink and the Lustre programming language. We first describe the semantics translation of Simulink models to Lustre. We then describe our denotational semantics for Stateflow. Finally, we also show how to compile Lustre code back to Simulink models.

The work on denotational semantics for Stateflow (Section 2.3) has been published in [17]. In addition, the work on reverse compilation of Lustre to Simulink (Section 2.4) has been published in [19].

2.1 Introduction

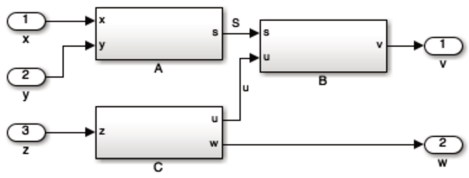
Let us first outline the specificities of synchronous languages and draw some parallels between constructs.

Synchronous languages aim at specifying the behavior of synchronous reactive systems. This computation model describes programs intended to be run forever, repeating the exact computation regularly, at fixed time steps. Usually, the time step length is not an element of the program itself but more like a meta-information required for further developments such as scheduling multiple processes. Synchronous models assume that the processing time of functions is immediate and communication is instantaneous [12]. These strong hypotheses allow us to separate the concerns: on the one hand, a functional description of the computation – the model –, and, on the other hand, physical constraints: the evaluation of the function body has to meet the deadlines. For example, if the program is expected to be executed 100 times a second, i.e., at 100Hz, then the evaluation of the function should be performed in less than 10 ms. This constraint is the subject of dedicated analyses such as the computation of worst-case execution time or the calculation of bounds over network delays. It can also impact hardware development, providing requirements in terms of computational power. That said, the model can focus on functional behavior and the computation performed at each time step, regardless of these time step lengths.

This work aims to facilitate knowledge transfer between designers using different modeling languages. For example, a designer familiar with Simulink can use our translation to generate Lustre code from a Simulink model. This code can then be verified using existing verification tools. Conversely, a designer familiar with Lustre can use our translation to generate a Simulink model from Lustre code. This model can then be simulated using existing simulation tools, such as Simulink, or be integrated within a large Simulink model.

2.2 Semantic translation of Simulink to Lustre

Simulink and Lustre are both synchronous data flow languages and share many similarities. For instance, Simulink Subsystems and Lustre nodes are transforming streams of input values into streams of output values. In Lustre, a node is defined by a set of stream equations with possible local variables denoting internal flows. This is equivalent to a Simulink Subsystem with a set of blocks connected to each other, each block output represents an internal flow that can be consumed by other blocks' inputs. The Simulink block algorithm is mapped to a Lustre



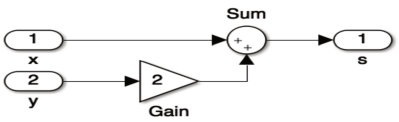

```
node A(x,y: real) returns(s: real); let ... tel
node B(s,u: real) returns(v: real); let ... tel
node C(z: real) returns(u,w: real); let ... tel
node Root(x,y,z: real) returns(v,w: real);
  var s, u: real;
 let
    s     = A(x,y);
    (u,w) = C(z);
    v     = B(s,u);
 tel
```

Figure 2.1: Simulink diagram to Lustre Nodes.

stream equation, and a signal connecting two Simulink blocks is mapped to a local Lustre variable. Hence, the translation of Simulink models to Lustre should be relatively easy and straightforward, and the only challenge will be to formally define Simulink blocks' semantics since it is not provided by the Simulink tool. The Algorithm is explained in Section 2.2.1, and the blocks that need more attention to capture their semantics correctly are explained in Sections 2.2.4, 2.2.6, and 2.2.5. In Chapter3, we introduce CoCoSim where this Algorithm is implemented and where more than a hundred Simulink blocks' semantics are defined as MATLAB functions producing Lustre code.

2.2.1 The translation algorithm

Algorithm 1 defines a procedure *SLX2Lus* that takes a system-block with content (i.e., a "C-connected $\mathcal{B}$ system") and produces the list of nodes defined by that block. The procedure *SLX2Lus* starts by compiling the system-block $\overline{B} = \{B_1, \ldots, B_n\}/C$, where $\mathcal{U}^{\overline{B}}$ and $\mathcal{Y}^{\overline{B}}$ are typed inputs and outputs respectively, into a Lustre node $N = (\mathcal{I}^N, \mathcal{O}^N, \mathcal{L}^N, Eqs^N)$. It creates first the inputs $\mathcal{I}^N$ (resp. outputs $\mathcal{O}^N$) based on $\mathcal{U}^{\overline{B}}$ (resp. $\mathcal{Y}^{\overline{B}}$) then goes over the content of the model to generate a set of local Lustre equations Eqs_i^N defining the block execution for each block inside $\overline{B}$. Each local equation creates local variables that are added to the node variables $\mathcal{L}^N$. The procedure *SLX2Lus* is recursively applied on each block inside $\overline{B}$ that has a content (e.g., Subsystem), resulting in a new list of nodes that will be added to the main list of nodes L. Fig. 2.1 and 2.2 illustrate an example of such a generation. The procedures *mapSLX2LUSVars* and *getBlockEquation* are explained in Sections 2.2.2 and 2.2.3. Procedure *IsSubsystem* checks if a block has content, and procedure *IsCondExecSubsystem* checks if the block is a conditionally executed Subsystem (i.e., θ-conditioned B_i).




```
Gain = 2.0 * y;
Sum = x + Gain;
s = Sum;
```

Figure 2.2: Simulink atomic blocks to Lustre code example.

Algorithm 1 SIMULINK to LUSTRE algorithm

1: **procedure** SLX2LUS($\overline{B}$) ▷ produce LUSTRE code of system-block $\overline{B}$
2: let $\overline{B} = \{B_1, \ldots, B_n\}/C$
3: let $\mathcal{U}^{\overline{B}} = \prod_{\nexists((.,.),(B,i)))\in C} \mathcal{U}_i^B$
4: let $\mathcal{Y}^{\overline{B}} = \prod_{B\in\mathcal{B}} \mathcal{Y}^B$
5: $L \leftarrow []$ ▷ initialize set of LUSTRE nodes
6: $\mathcal{I}^N \leftarrow mapSLX2LUSVars(\mathcal{U}^{\overline{B}})$ ▷ Create LUSTRE node Inputs
7: $\mathcal{O}^N \leftarrow mapSLX2LUSVars(\mathcal{Y}^{\overline{B}})$ ▷ Create LUSTRE node outputs
8: $\mathcal{L}^N \leftarrow []$ ▷ Initialize LUSTRE node variables list
9: $Eqs^N \leftarrow []$ ▷ Initialize LUSTRE node stream definitions list
10: **for** each B_i in $\{B_1, \ldots, B_n\}$ **do** ▷ Go over blocks of current Subsystem $\overline{B}$
11: $expr_i,\ v_i \leftarrow getBlockEquation(B_i)$ ▷ get block equations
12: $Eqs^N.add(expr_i)$ ▷ add block equations to the node body
13: $\mathcal{L}^N.add(v_i)$ ▷ add local variables to the variables list
14: **if** $IsSubsystem(B_i)$ **then**
15: $L_i \leftarrow SLX2Lus(B_i)$ ▷ call SLX2Lus recursively on B_i
16: $L.add(L_i)$ ▷ add generated nodes to the current list of nodes
17: **end if**
18: **if** $IsCondExecSubsystem(B_i)$ **then** ▷ if B_i is conditionally executed Subsystem
19: $N_i \leftarrow getNodeWrapper(B_i)$ ▷ Add the wrapper node that control the execution
20: ▷ of the Subsystem B_i.
21: $L.add(N_i)$ ▷ add wrapper node to the current list of nodes
22: **end if**
23: **end for**
24: $N \leftarrow (\mathcal{I}^N, \mathcal{O}^N, \mathcal{L}^N, Eqs^N)$ ▷ create the LUSTRE node corresponding to system-block $\overline{B}$
25: $L.add(N)$ ▷ add node to the list of nodes
26: **return** L ▷ LUSTRE program L
27: **end procedure**

2.2.2 Mapping Simulink data types to Lustre data types

In Algorithm 1, procedure *mapSLX2LUSVars* maps Simulinks signals to Lustre variables. Each Simulink signal is defined by a base type and a dimension. The supported Simulink data types are *double*, *single*, machine integers (e.g., *int8*, *uint8*), *boolean*, *enumeration*, and *Simulink Bus*.
Lustre data types are *real* for rational numbers, *int* for Integers, *bool* for booleans, and *enum* for enumerations.
Procedure *mapSLX2LUSVars* maps *double* and *single* to *real*, *Boolean* to *bool*, machine integers to *int*, *enumeration* to Lustre *enum*, and the *Simulink Bus* is inlined into its base types.

A set of Lustre libraries (`int_to_int8`, `int_to_uint8`, ...) are defined to add constraints on Lustre Integers.

Simulink provides good support for multidimensional array operations, whereas Lustre syntax used by CoCoSim backends (e.g., Kind2, LustreC) has a more limited support. In particular, partial assignment of an array and other operations such as concatenation, reshaping, and permutation. For this reason, we decided to inline all multidimensional arrays column-wise following MATLAB convention. For instance, a Simulink signal of type *double* and dimension $[2, 3]$ will be mapped to 6 Lustre variables of type *real*.

2.2.3 Defining the Lustre equations of a block

In Algorithm 1, procedure *getBlockEquation* defines the Lustre equation that expresses the execution of a given Simulink block. Since Lustre and Simulink are both data flow synchronous languages, the mapping between the Simulink block execution and its Lustre code is straightforward. For example, in Section 1.1.1.1, an atomic Simulink block B can be represented as a tuple:

$$(\mathcal{U}^B, \mathcal{Y}^B, \mathcal{D}^B, \mathcal{D}_0^B, \{(G_i^B, f_i^B, h_i^B)\}_{i=1}^{q^B}, (T^B, T_0^B))$$

The kth sampling time occurs at $kT^B + T_0^B$. The value of the input signal at the kth sampling time is denoted as $u(k) = (u_1(k), \ldots, u_{m^B}(k)) \in \mathcal{U}^B$, and similarly for other signals. At the kth sampling time, if the data $d(k)$ and the input $u(k)$ are such that $G_i^B(d(k), u(k))$ holds, the next data, $d(k+1) = f_i^B(d(k), u(k))$, is computed, and the output value is assigned to $y(k) = h_i^B(d(k), u(k))$.

Let us simplify this representation with a single data-update function and a single output-assignment function as follows:

$$d(k+1) = F^B(d(k), u(k)) = \begin{cases} f_1^B(d(k), u(k)) & G_1^B(d(k), u(k)) \\ \ldots & \\ f_{q^B}^B(d(k), u(k)) & G_{q^B}^B(d(k), u(k)) \end{cases}$$

$$y(k) = H^B(d(k), u(k)) = \begin{cases} h_1^B(d(k), u(k)) & G_1^B(d(k), u(k)) \\ \dots & \\ h_{q^B}^B(d(k), u(k)) & G_{q^B}^B(d(k), u(k)) \end{cases}$$

Therefore, the semantics of an atomic block B of how the states are updated and outputs are assigned by:

$$[\![B]\!] : \mathbb{N} \times \mathcal{D}^B \times \mathcal{U}^B \to \mathcal{D}^B \times \mathcal{Y}^B$$
$$(k, d(k), u(k)) \mapsto (d(k+1), y(k)) = (F^B(d(k), u(k)), H^B(d(k), u(k)))$$

For such a block, the algorithm generates two LUSTRE equations, one that updates the block outputs and the second one updating the subsequent block data. Both equations are defined based on the block semantic. The algorithm starts by mapping block B inputs, outputs, and data to their equivalent LUSTRE variables. Let v_d, v_u and v_y be the mapped LUSTRE variables of data $(d(k+1))$, inputs $(u(k))$ and outputs $(y(k))$ of the block, respectively, using *mapSLX2LUSVars* procedure. Any occurrence of $u(k)$, $y(k)$ and $d(k+1)$ is mapped to the current value of LUSTRE streams v_u, v_y and v_d respectively. The value of $d(k)$ is the previous computed data and is mapped to $d_0 \to pre(v_d)$ in LUSTRE where $d(0)$ is the initial data condition defined in the block parameters and d_0 is its mapped LUSTRE value.

For instance, if F^B and H^B are defined as follows:

$$d(k+1) = F^B(d(k), u(k)) = \alpha_1 * d(k) + \beta_1 * u(k)$$
$$y(k) = H^B(d(k), u(k)) = \alpha_2 * d(k) + \beta_2 * u(k)$$

Where d(0) is the initial data provided by the user. The previous SIMULINK equations will then be mapped to :

$$v_d = F^L(d_0 \to pre(v_d), v_u) = \alpha_1 * (d_0 \to pre(v_d)) + \beta_1 * v_u$$
$$v_y = H^L(d_0 \to pre(v_d), v_u) = \alpha_2 * (d_0 \to pre(v_d)) + \beta_2 * v_u.$$

Example 2.1
Let us provide LUSTRE equivalents of Simulink blocks in the stopwatch example in Fig. 1.1.

The Unit Delay block used in Fig. 1.1 named "0→pre time" *delays its input by one step. Its data-update function and output-assignment function can be defined by:*

$$[\![UnitDelay]\!] : \mathbb{N} \times \mathcal{D}^{UnitDelay} \times \mathcal{U}^{UnitDelay} \to \mathcal{D}^{UnitDelay} \times \mathcal{Y}^{UnitDelay}$$
$$(k, d(k), u(k)) \mapsto (d(k+1), y(k)) = (u(k), d(k))$$

Where $d(0) = d_0$ is the initial data condition. In other words, the current data is assigned to the current output, whereas the current input is assigned to the data of the next step.

Let `pre_time_data`, `pre_time_input` *and* `pre_time_output` *be the mapped LUSTRE variables of data, inputs and outputs of the block respectively using mapSLX2LUSVars procedure.*

`d0` *is the initial data condition. A direct translation of the Unit Delay block will produce the following* LUSTRE *code:*

```
pre_time_data =  pre_time_input;
pre_time_output = d0 -> pre(pre_time_data);
```

We could then merge these two equations in a single LUSTRE *equation by substituting* `pre_time_data` *by its definition, and we end up with the following equation:*

```
pre_time_output = d0 -> pre(pre_time_input);
```

Which corresponds to a Unit Delay over data flow `pre_time_input` *with initial value* `d0`*.*

For a SIMULINK *Constant block with a value* 0*, we do not need to provide its data or data-updating function since it is a stateless block:*

$$[\![Constant0]\!] : \mathbb{N} \times \mathcal{U}^{Constant0} \rightarrow \mathcal{Y}^{Constant0}$$
$$(k, u(k)) \mapsto (y(k) = 0)$$

The LUSTRE *equivalent equation would be:*

```
constant0_output = 0;
```

The Switch block used in the stopwatch model is represented as:

$$[\![Switch]\!] : \mathbb{N} \times \mathcal{U}^{Switch} \rightarrow \mathcal{Y}^{Switch}$$
$$(k, (u_1(k), u_2(k), u_3(k))) \mapsto \left(y(k) = \begin{cases} u_1(k) & when\ u_2(k) = True \\ u_3(k) & Otherwise \end{cases} \right)$$

The LUSTRE *equivalent equation would be:*

```
switch_output = if (switch_input2 = true) then switch_input1 else switch_input3;
```

The Sum block is represented as:

$$[\![Sum]\!] : \mathbb{N} \times \mathcal{U}^{Sum} \rightarrow \mathcal{Y}^{Sum}$$
$$(k, (u_1(k), u_2(k))) \mapsto (y(k) = u_1(k) + u_2(k))$$

The LUSTRE *equivalent equation would be:*

```
sum_output = sum_input1 + sum_input2;
```

For Subsystems calls, the LUSTRE *equation is a simple call to the* LUSTRE *node defining that Subsystem.*

The LUSTRE *equation for the* "is running" SIMULINK *subsystem execution would be:*

```
is_running_output = is_running(is_running_input);
```

Where `is_running` *is the* LUSTRE *node defining the* SIMULINK *subsystem* "is running"*.*

The Algorithm 1 applied on the SIMULINK *model of stopwatch of Fig. 1.1 would produce the following* LUSTRE *code:*

```
node stopwatch(toggle: bool; reset: bool) returns (time: int)
var
constant0 : int;
constant1: int;
sum_output: int;
is_running_output: bool;
if_running_output: int;
if_reset_output: int;
pre_time: int;
let
    constant0 = 0;
    constant1 = 1;
    sum_output = 1 + pre_time;
    is_running_output = is_running(toggle);
    if_running_output = if is_running_output then sum_output else pre_time;
    if_reset = if reset then constant0 else if_running_output;
    pre_time = 0 -> pre time;
    time = if_reset;
tel
```

The above LUSTRE generation is similar to Tripakis et al. (2005) work [131]. We share the same algorithm of recursively generating a LUSTRE node for each Subsystem, but we differ in how we translate conditionally executed subsystem and handle multi-periodic systems. In the following Sections, we go into more detail about how we handle particular SIMULINK blocks and multi-periodic systems.

2.2.4 Handling Conditionally Executed Subsystem

Conditionally executed Subsystem, such as enabled/triggered subsystems, are a type of subsystems that are conditionally executable when a specific guard condition over certain variables, called control inputs, holds. Furthermore, the data inside the system-block may be reset when the guard condition holds, while the outputs may be reset when the guard condition is violated.

We differentiate between different types of Conditionally Executed Subsystem: *Enabled Subsystem*, *Triggered Subsystem*, and *Enabled and Triggered Subsystem*. Given a system-block B, a *conditioning over* B is 5-tuple $\theta := (\mathcal{U}^\theta, G^\theta, f^\theta, h^\theta, (T^\theta, T_0^\theta))$ as defined in Definition 1.3. In the following, we give the semantics for each type:

Enabled Subsystem: Subsystem whose execution is enabled by external input. B is executed when $G^\theta(u^\theta(k)) = any(u^\theta(k) > 0.0)$ holds. The control signal can be either a scalar or a vector. If a scalar value is greater than zero, the Subsystem executes. Likewise, if any one of the vector element values is greater than zero, the Subsystem executes. Further, when a Subsystem block is disabled, block states for the blocks within the Subsystem can be held or reset:

- **held**: Hold block states at their previous values.

$$f^\theta : d^B(k+1) = d^B(k)$$

- **reset**: Reset block states to their initial conditions (zero if not defined).

$$f^\theta : d^B(k+1) = d^B(0)$$

Each *Outport* O in block B has its own settings when the Subsystem is disabled. The parameter "Output when disabled" of each Outport can be set to "reset" or "held" in case of a reset, an initial output is required:

- **held**: Output is held when the Subsystem is disabled.

$$h^\theta : y^O(k) = y^O(k-1)$$

- **reset**: Output is reset to the value given by the Initial output when the Subsystem is disabled.

$$h^\theta : y^O(k) = y^O(0)$$

Triggered Subsystem: Subsystem whose execution is triggered by external input. B is executed the same way described above in the Enabled Subsystem case, and the only difference is the definition of the guard G^θ that depends on the user setting. There are three commonly used settings of how the control signal triggers execution:

- **rising**: Trigger the execution of the Subsystem when the control signal rises from a negative or zero value to a positive value.

$$G^\theta_{rising}(u^\theta(k)) = (u^\theta(k-1) \leq 0) \ and \ (u^\theta(k) > 0)$$

- **falling**: Trigger the execution of the Subsystem when the control signal falls from a positive or zero value to a negative value.

$$G^\theta_{falling}(u^\theta(k)) = (u^\theta(k-1) \geq 0) \ and \ (u^\theta(k) < 0)$$

- **either**: Trigger the execution of the Subsystem with either a rising or falling control signal.

$$G^\theta_{either}(u^\theta(k)) = G^\theta_{falling}(u^\theta(k)) \ or \ G^\theta_{rising}(u^\theta(k))$$

For first execution ($k = 0$), $G^\theta(u(k))$ holds.

Enabled and Triggered Subsystem: A Subsystem that contains both an Enable port block and a Trigger port block. When a trigger signal rises or falls through zero, the enable input port

is checked to evaluate the enable control signal. If its value is greater than zero, the subsystem is executed. When both inputs are vectors, the subsystem executes if at least one element of each vector is nonzero.

Conditionally Executed Subsystem translation

We give the translation method of the Enabled Subsystem, the other types follow the same approach. Given a conditionally executed Subsystem $B \Downarrow \theta$ with a conditioning $\theta := (\mathcal{U}^\theta, G^\theta, f^\theta, h^\theta, -)$.

The following LUSTRE node is wrapping the execution of Subsystem B generated by *getNodeWrapper* from Algorithm 1:

```
node B_wrapper(u, u_theta: real)
returns(y:real);
let
    --Compute previous outputs at every step
    pre_y = 0.0 → pre y;
    automaton enabled_subsystem
    state Active:
    unless (not G_L^θ(u_theta)) resume Inactive
    let
      y = B(u);
    tel

    state Inactive:
    (* The choice of restart or resume is based on if B is configured to "reset" or "held" its
        memories when it is disabled.*)
    unless (G_L^θ(u_theta)) restart/resume Active
    let
    -- h_L^θ returns the initial condition ("reset") or the previous value of y ("held").
      y = h_L^θ(pre_y);
    tel
tel
```

Where u is the input, y is the output, and u_theta is the "enable control signal".

h_L^θ returns for each block output its initial condition when "output when disabled" is set to "reset" or the previous value of the output when "output when disabled" is set to "held."

G_L^θ is the guard that control the execution of the block B, in the case of Enabled Subsystem $G_L^\theta(u\_theta) = (u\_theta > 0.0)$.

The wrapper node computes the previous outputs at each step. Then it defines an automaton with two states. The initial state is called *Active*, where B is executed, and the *Inactive* state, where B is disabled. When in the *Inactive* state, the outputs hold their memories or reset to their initial condition based on their setting "output when disabled" value. Once the automaton is in the *Active* state and the control guard $G_L^\theta(u\_theta)$ does not hold, the automaton switches to the *Inactive* state and execute h_L^θ. The automaton switches from *Inactive* to *Active* when the control guard $G_L^\theta(u\_theta)$ holds. Prior to executing B, f_L^θ is executed by resuming the execution of state *Active* (`resume Active`), which means holding the previous memories, otherwise, restarting the state execution (`restart Active`) which reset all memories inside the state *Active* including

the call to B.

Limitation of this translation: The LUSTRE automaton captures the semantics of Enabled Subsystem concisely: the Subsystem is enabled (state Active) or disabled (state Inactive). When restarting the execution of B, we reset the memories (using the restart keyword) or held them (resume keyword). In LUSTRE, the modularity is respected; when resetting memories, all nodes called are reset, as also the nested nodes. Therefore, if B calls another node N with memories, N will be reset too. In Simulink, for nested subsystems whose Enable blocks have different parameter settings, the settings for the child subsystem override the settings inherited from the parent subsystem. Therefore, our translation does not support nested conditionally executed subsystem with different settings; only the same settings are allowed. A solution to this issue could be to extend all nodes' inputs with a reset input that resets memories individually.

2.2.5 Handling Resettable Subsystem

A resettable Subsystem is a Subsystem whose block states are reset with an external trigger. A resettable subsystem executes at every step but conditionally resets the states of blocks within it when a trigger event occurs at the reset port.

LUSTRE supports resetting the internal state of a node to its initial state by using the construct `every`. Writing:

```
y = B(u) every c;
```

Makes a call to node B with argument u, and every time the Boolean stream c is true, the internal state of the node is reset to its initial value. Condition c is a guard over the reset signal.

Similar to the limitation over conditionally executed subsystem, nested subsystems' settings override the settings inherited from the parent subsystem. Resetting memories for individual subsystems could be a solution but breaks the modularity of the translation. We detect such cases and report them back to the user as unsupported.

2.2.6 Handling multi-periodic systems

Our approach to dealing with multi-periodic systems is highlighted in Fig. 2.3.

First, we need to define the semantics of multi-periodic systems in SIMULINK and connect it to the one of synchronous programs. In synchronous languages, execution time is neglected while each computation is performed repetitively, e.g., every *ts* seconds. In SIMULINK, most discrete subsets of blocks are fitted with synchronous semantics, but the case of multi-periodic systems is more complex and requires an analysis of the internal semantics.

Once the semantics has been defined, the second contribution is the extension of the above LUSTRE generation to encode the multi-periodic communication with classical LUSTRE over- and sub-sample operators (left-hand side of Fig. 2.3). This amounts to expressing the whole

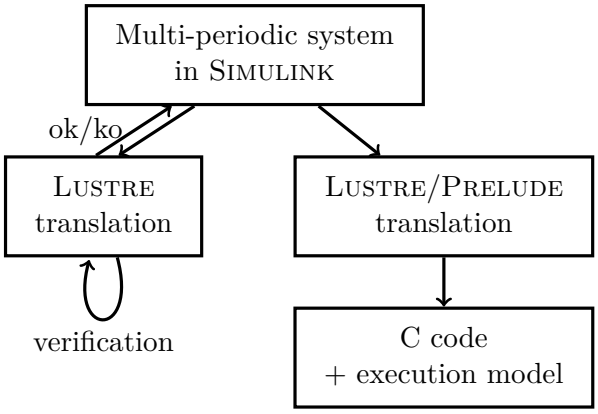


Figure 2.3: Code generation of multi-periodic systems in SIMULINK

system on a base clock. This LUSTRE model is then used to perform formal analysis using SMT-based model-checking. While required to analyze the entire system properly, this encoding is not efficient for execution.

Once the verification is valid, the last step is efficient code generation (right-hand side of Fig. 2.3). Each synchronous component is translated as a LUSTRE model, which will eventually be compiled into C code, while aggregating nodes, mixing different clocks, or execution rates are expressed as PRELUDE programs. PRELUDE [105] is a synchronous language that has been defined to program multi-periodic applications. From a PRELUDE program, the compiler generates a set of classical real-time tasks, and many *predictable* implementations have been proposed for multi- and many-core architectures [109].

Both SIMULINK and LUSTRE allow the definition of data flows that run on a different clock. An atomic block B with a sample time (T^B, T_0^B) with period T^B and initial phase T_0^B is updated only at times $k * T^B + T_0^B$ for $k \in \mathbb{N}$, whereas, it remains constant during the intervals $[k * T^B + T_0^B, (k+1) * T^B + T_0^B)$ for SIMULINK and undefined for LUSTRE.

To characterize $(T^{\overline{B}}, T_0^{\overline{B}})$ for a system-block $\overline{B} := \mathcal{B}/C$, we can express periods and offsets without loss of generality as rational numbers $p/q \in \mathbb{Q}$ where $p \wedge q = 1$. Let us consider n blocks B_i, each associated to the period and offset $T_i, T_{0_i} \in \mathbb{Q}$. Let $T_i^n, T_i^d, T_{0_i}^n, T_{0_i}^d \in \mathbb{N}$ be such that $T_i = T_i^n / T_i^d$ and $T_{0_i} = T_{0_i}^n / T_{0_i}^d$. We have two cases:

- If all offsets are equal (i.e., $T_{0_i} = T_{0_j}$). In this case, $T_0^{\overline{B}} = T_{0_i}$, and $T^{\overline{B}} = T^n / T^d$ where $T^d = lcm\left(\{T_i^d\}_i\right)$ and $T^n = gcd\left(\{T_i \times T^d\}_i\right)$.
- Otherwise, $T_0^{\overline{B}} = 0$, and $T^{\overline{B}} = T^n / T^d$ where $T^d = lcm\left(\{T_i^d, T_{0_i}^d\}_i\right)$ and $T^n = gcd\left(\{T_i \times T^d, T_{0_i}^n \times T^d\}_i\right)$.

Let us remark that since T^d is defined as the least common multiplier of all denominators of periods and offsets, the terms in the gcd expression of T^n are all integers.

Example 2.2
Figure 2.4 is a simple example of SIMULINK diagram B running on different sample times D1 = (1s, 0s) and D2 = (2s, 0s). The Inport In1 is running on D1. The Sum block adds two signals

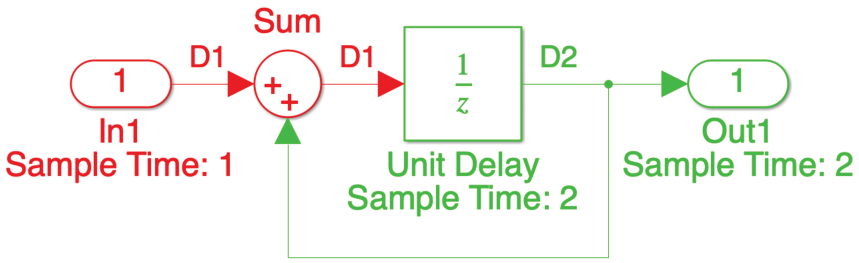
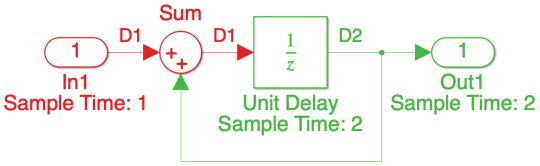
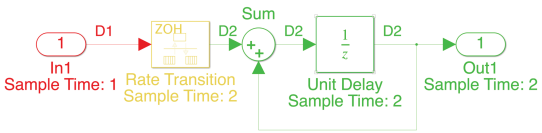


Figure 2.4: A multi-periodic SIMULINK diagram.



(a) Example of a multi-periodic SIMULINK model.



(b) Introducing explicit *Rate Transition* block in the SIMULINK model of Fig.2.5a

Figure 2.5: SIMULINK blocks clock interface before and after introducing *Rate Transition*.

running on different periods, D1 and D2. The Unit Delay delays its input signal by one step but updates its output every 2 seconds (sample time D2).

B conforms to the SIMULINK diagram class stated in Definition 1.4: $B = \{B^{Inport}, B^{Sum}, B^{UnitDelay}, B^{Outport}\}/C$ *and* C *is omitted.* $\mathcal{U}^B = \mathcal{U}^{Inport}$, $D^B = D^{UnitDelay}$, $D_0^B = D_0^{UnitDelay}$, $T^B = gcd(1,2) = 1$, *and* $T_0^B = 0$.

The model's output is incremented by In1 every 2 seconds because the UnitDelay block updates its outputs every two seconds and because Sum is a stateless block that adds the values of its inputs at the same time step. The behavior can be represented as:

t	0	1	2	3	4	5
In1	1	1	1	1	1	1
Out1	0	0	1	1	2	2

The model described in Fig.2.5a will be rejected by LUSTRE as blocks clock signature are not respected. In LUSTRE, we cannot add two flows running on different clocks. By default, SIMULINK introduces an implicit *Rate Transition* block between blocks running on different sample times (e.g., between *Sum* and *Unit Delay* in Fig.2.5a). The user can force SIMULINK to reject models with unspecified data transfers between rates. In the case of the previous SIMULINK example, we get the following error when we set "Multi-task (or Single task) rate transition" to "error": *The sample time 1 of 'example/Sum' at input port 2 is different from the sample time 2 of 'example/Unit Delay' at output port 1.*

A solution to this problem is to let the user specify the correct data transfer block explicitly (e.g., *Rate Transition* block). Fig. 2.5b illustrates a solution to the previous error by inserting a *Rate Transition* block between inport *In1* and *Sum* block. Thanks to these introduced *Rate Transition* blocks, the boundaries between blocks that run on different periods become explicit

and limit the reasoning about clocks to these data transfer blocks.

The *Rate Transition* block (RTB) has two block parameters that control its execution: **Ensure data integrity** and **Ensure deterministic data transfer**. When the first parameter is checked, it ensures data integrity when the block transfers data. A data integrity problem exists when the input to a block changes during the execution of that block. For instance, a faster block supplies the input to a slower block. In a protected data transfer, the output of the faster block is held until the slower block finishes executing. The second parameter enforces a deterministic data transfer where the data transfer timing is entirely predictable, as determined by the sample rates of the blocks. The timing of a non-deterministic data transfer depends on the availability of data, the sample rates of the blocks, and the time at which the receiving block begins to execute relative to the driving block.

We only allow rate transition blocks that ensure data integrity and determinism. With this restriction, there exist two main types of data transfer:

- ZOH: the Zero-Order-Hold is a deterministic RTB that implements direct communications from fast to slow blocks at harmonic periods. $\exists\, n \in \mathbb{N},\ T^{in} = T^{out}/n \ and\ T_0^{in} = T_0^{out} = 0.$
- $1/Z$: Acts as a unit delay that implements delayed deterministic communications from slow to fast block at harmonic periods. $\exists\, n \in \mathbb{N},\ T^{in} = T^{out} * n \ and\ T_0^{in} = T_0^{out} = 0.$

Translation in Lustre The first Simulink multi-periodic translation technique [131] produces a pure Lustre specification. The first step translates the sample time in Lustre as the sub-sampled clock of the common base clock. The idea is to compute the common base clock as explained above and express each couple (period, offset) relatively to it. For the example in Fig. 2.5b, one solution to encode D1 and D2 could be:

```
D1 = true;
D2 = true -> not pre(D2);
```

D1 was equal to the common base clock, whereas D2 was twice as slow. So then, alternating true and false in flow D2 will lead to true every two times. However, using `pre` will become unpractical when the clocks become complex. Instead, we use some counters and have a similar approach as the one in Prelude.

```
D1 = make_clock(1,0);
D2 = make_clock(2,0);
```

Here `make_clock` is a Lustre node that generates a Boolean clock that is true at the logical instants $k * \text{period} + \text{offset}$ with $k \in \mathbb{N}$ and false otherwise.

```
node make_clock(period, offset : int)
returns(clk : bool)
var count: int;
let
 count = ((period - offset) -> (pre(count) + 1)) mod period;
 clk = (count = 0);
tel
```

The second step combines different sample times. The idea is to bring back a flow on the common base clock by holding its values when the input clock is undefined and then sub-sampling it again with the output clock. Let us detail the translation for the two rate transition blocks, ZOH and 1/z, supported by our translation. Let (inTs, inTsOffset) (resp. (outTs, outTsOffset)) be the sample time of the RTB input port called RTB_U (resp. output port called RTB_Y) relative to the common base clock. We thus have:

```
C_in = make_clock(inTs, inTsOffset);
C_out = make_clock(outTs, outTsOffset);
```

From Fast to slow: outTs > inTs, ZOH block The communication is direct, and the output port RTB_Y takes the value of the input port RTB_U on its sample time:

```
RTB_tmp =
  merge C_in
  (true → RTB_U)
  (false → (dft -> pre RTB_tmp) when not C_in);
RTB_Y = RTB_tmp when C_out;
```

From slow to fast: outTs < inTs, 1/z block The block behaves as a *Unit Delay*. We first compute the previous value of the input signal, then compute the values in the base clock by keeping the last values when it is undefined and finally sample the signal to the output clock.

```
RTB_tmp =
 merge C_in
 (true → (dft -> pre RTB_U))
 (false → (dft -> pre RTB_tmp) when not C_in);
RTB_Y = RTB_tmp when C_out;
```

Example 2.3
The example of Fig. 2.5b is translated as follows:

```
In1_on_cc = merge D1 (true → In1) (false → (0.0 -> pre In1_on_cc) when not D1);
RateTransition = In1_on_cc when D2;
Sum = RateTransition + UnitDelay;
UnitDelay = 0.0 -> pre Sum;
Out1 = UnitDelay;
```

Example 2.4
Let us consider Fig. 2.6 to illustrate the translation of a few examples of Rate Transition (RTB) block (ZOH and 1/z).

In Table. 2.1, we give a few different settings of RTB block and their translation in Lustre *and* Prelude. *We set the Counter Subsystem with a* Simulink *clock of* $(2s, 0s)$*; that is, the counter is incremented by one every 2 seconds. The common base clock is* $(1s, 0s)$.

Translation in Prelude Prelude [105] is a synchronous language that has been defined to program multi-periodic applications. The language considers *imported nodes* that can be programmed in C or Lustre. An example is given below where a node *Sum* has two inputs, one output, and a worst-case execution time (WCET) of one logical time.

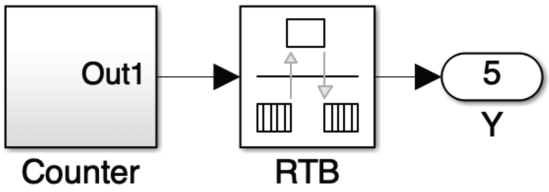


Figure 2.6: Sampled counter

```
imported node Sum (v1, v2 :real)
returns (v :real) wcet 1;
```

PRELUDE uses two sampling operators, an acceleration operator `*^n` that will divide the period by n, and a deceleration operator `/^n` that will multiply the period by n. In the COCOSIM framework, the code generated from SIMULINK will be composed of LUSTRE nodes for mono-periodic (sub-) systems and PRELUDE when several rates are handled. Thus, in that case, LUSTRE nodes are assembled in a PRELUDE program, which details the real-time constraints of the system and the semantics of the communications between the LUSTRE nodes. This assembly is not done directly in LUSTRE because communications relate to nodes executing at different periodic rates, and PRELUDE is better suited to the specification and efficient compilation.

```
node assembly(In1: real rate(10,0))
 returns (Out1_1 :real)
var
  Sum_1_1 :real;
  UnitDelay_1_1 :real ;
 let
  Sum_1_1 = Sum(0.0 fby UnitDelay_1_1, In1/^ 2);
  UnitDelay_1_1 = UnitDelay(Sum_1_1);
  Out1_1 =  UnitDelay_1_1;
tel
```

The example corresponds to Fig. 2.5b. The *assembly* node has one input *In1* with a clock $(10, 0)$ meaning that *In1* has a period of 10 and an offset of 0. It is up to the user to link the logical clock with the physical one. Since *In1* is running at D1=(1s,0s), the logical clock is at $100ms$.

The imported node *Sum* consumes the flows `In1/^2` and `(0.0 fby UnitDelay_1_1)`. Nodes are synchronous, thus, the two inputs and the output must have the same clock. As the clock of `In1` is $(10, 0)$ by definition, the flow `In1/^2` has $(20, 0)$ as clock. The operator `/^2` is a deceleration by 2.

The operator `*^2` is an acceleration that will divide the period by 2. It is used for the $1/z$ block, as shown in Table 2.1.

2.2.7 Limitations of the translation

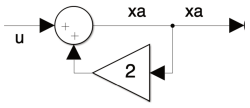
Algebraic Loop: An algebraic loop occurs when there is a circular dependency of block outputs and inputs in the same time step. Figure 2.7 illustrates a simple example of an algebraic loop

t	0	1	2	3	4	5	6	7
Counter: Sample Time [2s, 0]	0	0	1	1	2	2	3	3
RTB: Determinism:ON; Integrity:ON; Sample time [4s,0] => type ZOH								
SIMULINK: Y	0	0	0	0	2	2	2	2
LUSTRE:								
$C\_4\_0 = make\_clock(4, 0);$	true	false	false	false	true	false	false	false
$Counter = Counter\_SS();$	0	-	1	-	2	-	3	-
$Y = Counter\ when\ C\_4\_0;$	0	-	-	-	2	-	-	-
PRELUDE:								
$Y = Counter/\widehat{}\ 2;$	0	-	-	-	2	-	-	-
RTB: Determinism:ON; Integrity:ON; Sample time [1s,0] => type 1/z								
SIMULINK: Y	0	0	0	0	1	1	2	2
LUSTRE:								
$C\_2\_0 = make\_clock(2, 0);$	true	false	true	false	true	false	true	false
$Counter = Counter\_SS();$	0	-	1	-	2	-	3	-
$Y = merge\ C\_2\_0$ $(true \rightarrow (0.0- > pre\ Counter))$ $(false \rightarrow (0.0- > pre\ Y)\ when\ not\ C\_2\_0);$	0	0	0	0	1	1	2	2
PRELUDE:								
$Y = (0.0\ fby\ Counter) *\widehat{}\ 2;$	0	0	0	0	1	1	2	2

Table 2.1: Translation to LUSTRE and PRELUDE of different settings of block RTB in Fig. 2.6. *Counter_SS* is the translation of *Counter* Subsystem in LUSTRE.

accepted by SIMULINK. The Sum block is an algebraic variable x_a that is constrained to be equal to the first input u plus $2 * x_a$ (i.e. $x_a = u + 2 * x_a$ or $x_a = -u$).

In SIMULINK models, algebraic loops are algebraic constraints. SIMULINK solves these algebraic equations numerically for x_a at each step of simulation using the ODE (Ordinary Differential Equation) solver. On the other hand, LUSTRE forbids such constructs, and no cyclic dependency is allowed, therefore, models with algebraic loops are detected and not supported by the compiler.



(a) Example of an algebraic loop accepted by SIMULINK.

```
xa = u + 2*xa;
```

(b) The generated LUSTRE that will be rejected because of the circular dependency.

Figure 2.7: A simple example of an algebraic loop.

2.3 Denotational semantics of Stateflow

STATEFLOW is a widely used modeling framework for embedded and cyber-physical systems where control software interacts with physical processes. In this Section, we present denotational semantics for STATEFLOW models we published in [17]. We propose a compilation process using continuation-passing style (*CPS*) denotational semantics. Our compilation technique preserves the structural and modal behavior of the system. The overall approach is implemented and integrated into our open source toolbox COCOSIM (See Chapter 3). We also present preliminary experimental evaluations in Chapter 3, illustrating our approach's effectiveness in code generation and safety verification of industrial scale STATEFLOW models.

A CPS semantics for STATEFLOW. The starting point of our approach is the expression of the denotational semantics of STATEFLOW [77] as a pure *continuation-passing style* (CPS) denotational semantics. CPS was proposed in the 70s by Plotkins [108] for λ-calculus call-by-value semantics and later developed for efficient compilation means, for example, in the long line of Danvy's works, e.g., [90] and Appel's book [8]. As recalled by Danvy, CPS terms can be expressed yet enjoy several valuable properties, e.g., "offering a good format for compilation and optimization". The following equations define Plotkin's call-by-value CPS rules:

$$\begin{array}{rcl} [\![x]\!]\,\kappa & = & \kappa\,x \\ [\![\lambda\,x.e]\!]\,\kappa & = & \kappa\,(\lambda\,x\cdot\lambda\,k\cdot[\![e]\!]\,k) \\ [\![e_0\,e_1]\!]\,\kappa & = & [\![e_0]\!]\,(\lambda\,v_0.[\![e_1]\!]\,(\lambda\,v_1\cdot v_0\;v_1\;\kappa)) \end{array}$$

The key idea is to associate to each function an additional argument, the explicit continuation $\kappa : t \to t$. This continuation is an endomorphic map over t values on which control is explicitly modeled: function calls, intermediate values, evaluation order, etc. In compilation, CPS β-reduction amounts to characterize a global continuation, which, when evaluated, produces the generated code.

Our work makes the following contributions:

- We adapted Hamon's denotational semantics [77] to pure CPS semantics, solving some of its flaws (see §2.3.2).
- We instantiated such CPS semantics for different uses, such as a *model interpreter* and a *code generator* for STATEFLOW models. Such a framework has several advantages, including a formal semantics of STATEFLOW that preserves the hierarchical structure of the model.
- We implemented the general CPS semantics and the proposed instantiations in OCaml: an interpreter and a code generator both for imperative code and LUSTRE automaton.
- The LUSTRE automaton code generator from STATEFLOW has been implemented and integrated into COCOSIM [15] – an automated analysis framework for SIMULINK models. COCOSIM, among other features, provides an intuitive user interface that facilitates the modeling of safety properties, code generation, verification, and graphical debugging of failed properties.

- We used CoCoSim fitted with this new capability to address Stateflow model compilation and verification on a set of industrial scale benchmarks and have experimentally evaluated our approach using two evaluation scales: (i) does the tool generate faithful code (wrt. the intended Stateflow semantics)? Moreover, (ii) is the tool able to verify safety properties efficiently? In Section 3.3.1 we provide evidence that answers positively both questions.

2.3.1 CPS Denotational Semantics for Stateflow

Motivating Continuation-passing style. The denotational semantics presented in [77] relies on continuations to model the actions of path computation. Indeed, actions associated with an atomic transition (a segment) are performed immediately for condition actions and eventually for transition actions. Success and fail continuations allow capturing this complex behavior in functions, representing side effects as denotations. However, values manipulated by this denotational semantics were explicitly first order and represented by environments ρ of type *Env*: $\rho ::= \{x_0 : v_0, \ldots, x_n : v_n, s_0 : b_0, \ldots, s_k : b_k\}$. These environments represented both the values of variables x_i and the active status of states s_i.

The encoding of [77] could be significantly improved by rearranging arguments to push environments in the rightmost position and defining a pure continuation-passing style (CPS) denotational semantics and point-free wrt environments. Indeed, in some situations, the author would drop continuations and evaluate explicit intermediate environments.

The following definitions characterize our CPS denotational semantics for Stateflow, following precisely the semantics of [77] while solving its flaws. The semantics is a higher order and environments are never made explicit: the evaluation of a model component acts as a transformer.

Conditions. Transitions in Stateflow are computed based on the current environment and an active event, evaluating conditions. Without loss of generality, we assume that the event is part of the environment and is not made explicit in the rules. An active event `e` could be checked as a regular condition using the predicate `event(e)`. We recall that the environment contains both variables mapped to values and the active status of Stateflow states. To clarify the expression, we made explicit the check whether a state characterized by p is active using the predicate `active(p)`.

Actions. The critical ingredient of transition computation in Stateflow is the sequence of actions applied to the current environment, updating values of variables and changing active and inactive states. Actions act as transformers and are the values manipulated by our CPS denotational semantics. We denote by *Den* this transformer type.

Basic action constructors are left free here but typically express some imperative assignment of an expression to an environment variable. In addition, we introduce the actions `open p` and `close p` which switches the Boolean status of state p to `true` or `false` respectively. To generalize the approach, we express disjunctions as actions using the constructor

$\mathcal{I}te(condition, Den, Den)$.

A primitive action (assignment or open/close action) semantics, i.e., its interpretation as a transformer, is defined using the function $\mathcal{A}[\![\cdot]\!] : action \to Den$. (Actions) transformers can be combined using the operator $\gg$: $Den \to Den \to Den$, which is associative: $a_1 \gg a_2 \gg a_3$ means that action a_1 is performed before action a_2 followed by a_3. Last, the default action, identity, is denoted $\mathcal{I}d$.

Denotational semantics as a functions map. Semantics functions are associated with state names and a global continuation environment θ of type *KEnv* is defined as follows:

$$\begin{array}{lll} \theta & ::= & \{\, p_0 : (\mathcal{S}[\![p_0 : sd_0]\!]^e_m\, \theta, \mathcal{S}[\![p_0 : sd_0]\!]^d\, \theta, \mathcal{S}[\![p_0 : sd_0]\!]^x_m\, \theta) \\ & & \quad \ldots \\ & & \quad p_n : (\mathcal{S}[\![p_n : sd_n]\!]^e\, \theta, \mathcal{S}[\![p_n : sd_n]\!]^d\, \theta, \mathcal{S}[\![p_n : sd_n]\!]^x\, \theta) \\ & & \quad j_0 : \mathcal{T}[\![T_0]\!]\theta, \ldots, j_k : \mathcal{T}[\![T_k]\!]\theta\} \end{array}$$

Functions $\mathcal{S}[\![p_0 : sd_0]\!]^e_m$, $\mathcal{S}[\![p_0 : sd_0]\!]^d$ and $\mathcal{S}[\![p_0 : sd_0]\!]^x_m$ denote, respectively, the semantics of a state when entering it, executing it, or exiting it. Note that entry and exit actions are parametrized by a mode $m \in Mode = L \mid S$. This mode, either loose (L) or strict (S), captures the difference between inner and outer transitions for entry and exit actions. Junctions are associated with the transition list semantics function $\mathcal{T}$. The θ map captures the semantics of all components of the STATEFLOW model and is typically provided as an argument of denotations.

Transitions semantics. STATEFLOW semantics is rather complex. A STATEFLOW transition amounts to evaluate a sequence of atomic transitions. Depending on some dynamic conditions, each atomic transition may be eventually fired or not. In all cases, it will impact the environment through side effects (condition actions). We introduce three continuations: `success` of type $k^+ ::= Den$ modeling a fired transition, a `fail` continuations of type $k^- ::= Den$ modeling an unfired one and a third case $\texttt{fail}^{glob}$ of type k^- capturing complex executions in which a series of junctions ends in a terminal junction. In case of a transition leading to another state, some entry or exit actions may be performed[1]. They are captured by the `wrapper` continuation of type $w ::= p \to Den \to Den$.

The evaluation of a destination path $\mathcal{D}[\![.]\!]$, which is a state amounts to apply the wrapper on the success continuation. Otherwise, when the destination is a junction, the transition list semantics is evaluated with the same continuations.

$\mathcal{D}[\![p]\!]\,(\theta : KEnv)\,(wrap : w)\,(success : k^+)\,(fail\, fail^{glob} : k^-) : Den = wrap\; p\; success$
$\mathcal{D}[\![j]\!]\,\theta\; wrap\; success\; fail\; fail^{glob} = \theta^{\mathsf{j}}(j)\; wrap\; success\; fail\; fail^{glob}$

Atomic transition semantics $\tau[\![.]\!]$ introduces an $\mathcal{I}te$ action: in case of an unfeasible condition, the (regular) fail continuation is used, otherwise, a new success continuation is built, evaluating the transition actions of the atomic transition. The action associated to the then-branch combines the condition actions followed by the new destination evaluation and relies on the newly defined success continuation.

[1]Note that those actions are not performed for a transition ending in a terminal junction.

$$\tau[\![(e_t, c, (a_c, a_t), d)]\!]\,(\theta : KEnv)\,(wrapper : w)\,(success : k^+)\,(fail\ fail^{glob} : k^-) : Den =$$
$$\quad \mathcal{I}te(event(e_t) \wedge c,$$
$$\qquad (\texttt{let}\ success' = success \gg (\mathcal{A}[\![a_t]\!])\ \texttt{in}$$
$$\qquad\ (\mathcal{A}[\![a_c]\!]) \gg (\mathcal{D}[\![d]\!]\,\theta\ wrapper\ success'\ fail\ fail^{glob})),$$
$$\qquad fail)$$

Evaluation of a list of transitions $\mathcal{T}[\![.]\!]$ performs a left-to-right traversal of the list: the unfeasibility of the head transition leads to the evaluation of the next, involving the definition of fail continuations. As a special case, when provided with an empty list of transitions, it always evaluates to the global fail continuation.

$$\mathcal{T}[\![\varnothing]\!]\,(\theta : KEnv)\,(wrapper : w)\,(success : k^+)\,(fail\ fail^{glob} : k^-) : Den = fail^{glob}$$
$$\mathcal{T}[\![t.T]\!]\,\theta\ wrapper\ success\ fail\ fail^{glob} =$$
$$\quad \texttt{let}\ fail' = \mathcal{T}[\![T]\!]\,\theta\ wrapper\ success\ fail\ fail^{glob}\ \texttt{in}$$
$$\quad \tau[\![t]\!]\,\theta\ wrapper\ success\ fail'\ fail^{glob}$$

State semantics. State semantics involves the opening and closing actions of states. We introduce wrapper functions dedicated to inner and outer transitions and entering states.

Wrapper considers a source and destination paths, p_s and p_d, and identifies the common prefix p of both paths. Depending on the context (inner or outer transition), it will compose, in order, the exit actions of remaining p_s, the transition actions continuation, and the entering actions of the remaining p_d. Outer transitions involve loose state semantics, while inner transitions involve strict ones. A specific wrapper $\texttt{open\_path}^{\mathbf{v}}$ is introduced to enter substates of a path.

$$\texttt{open\_path}^{\mathbf{v}}\ (\theta : KEnv)\,(p : Path)\,(p_s : Path)\,(p_d : Path) : w =$$
$$\texttt{if}\ \texttt{hd}(p_s) = \texttt{hd}(p_d) \wedge \texttt{hd}(p_s) \neq \varnothing\ \texttt{then}$$
$$\quad \texttt{open\_path}^{v}\ \theta\ p.\texttt{hd}(p_s)\,\texttt{tl}(p_s)\,\texttt{tl}(p_d)$$
$$\texttt{else match}\ \mathbf{v}\ \texttt{with}$$
$$\quad \mathbf{o}\ \texttt{->}\ \lambda\, den.\theta^{\mathbf{x}}_L(p.\texttt{hd}(p_s)) \gg den \gg \theta^{\mathbf{e}}_L(p.\texttt{hd}(p_d))\,\texttt{tl}(p_d)$$
$$\quad \mathbf{i}\ \texttt{->}\ \lambda\, den.\theta^{\mathbf{x}}_S(p.\texttt{hd}(p_s)) \gg den \gg \theta^{\mathbf{e}}_S(p.\texttt{hd}(p_d))\,\texttt{tl}(p_d)$$
$$\quad \mathbf{e}\ \texttt{->}\ \lambda\, den.den \gg \theta^{\mathbf{e}}_L(p.\texttt{hd}(p_d))\,\texttt{tl}(p_d)$$

This definition assumes that *hd* and *tl* functions, returning the head and tail of a list, are extended to handle empty lists, i.e., $\texttt{hd}\,\varnothing = \varnothing$ and $\texttt{tl}\,\varnothing = \varnothing$.

We now define the core semantics functions of states, $\mathcal{S}[\![\cdot]\!]^{\{\mathbf{e},\mathbf{d},\mathbf{x}\}}_m$. Note that the mode parameter m is provided as an index. Entering or exiting a path executes the entry and exit actions of all states in the path, respectively. Depending on the outer or inner status of a transition, the entry or exit actions of the root node shall or shall not be evaluated. The following definitions capture Stateflow semantics, handling specificities such as transitions from a node to a child or parent.

$$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{e}}_S\,(\theta : KEnv)\,(\varnothing : Path) : Den = (\mathcal{C}[\![C]\!]^{\mathbf{e}}\,p\,\theta)$$
$$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{e}}_S\,\theta\ s.p_d = (\theta^{\mathbf{e}}_L(p.s)\ p_d)$$
$$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{x}}_S\,(\theta : KEnv) : Den = (\mathcal{C}[\![C]\!]^{\mathbf{x}}\,p\,\theta)$$

$$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{e}}_L\,\theta\,\varnothing = (\mathcal{A}[\![a_e]\!]\,\theta) \gg (\mathcal{A}[\![\texttt{open}\ p]\!]) \gg (\mathcal{C}[\![C]\!]^{\mathbf{e}}\,p\,\theta)$$
$$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{e}}_L\,\theta\ s.p_d = (\mathcal{A}[\![a_e]\!]\,\theta) \gg (\mathcal{A}[\![\texttt{open}\ p]\!]) \gg (\theta^{\mathbf{e}}_L(p.s)\ p_d)$$
$$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{x}}_L\,\theta = (\mathcal{C}[\![C]\!]^{\mathbf{x}}\,p\,\theta) \gg (\mathcal{A}[\![a_x]\!]\,\theta) \gg (\mathcal{A}[\![\texttt{close}\ p]\!])$$

The definition for during actions is the following. First outer transitions are evaluated. If

none succeeds, the during action of the node is computed. Then, inner transitions apply. If no transition can be fired at all, the components of the node are computed.

$$
\begin{array}{l}
\mathcal{S}[\![p : ((a_e, a_d, a_x), T_o, T_i, C)]\!]^{\mathbf{d}}\,(\theta : KEnv) : Den = \\
\quad \texttt{let}\ \ wrapper_i = \texttt{open\_path}^{\mathbf{i}}\,\varnothing\ p\ \texttt{in} \\
\quad \texttt{let}\ \ wrapper_o = \texttt{open\_path}^{\mathbf{o}}\,\varnothing\ p\ \texttt{in} \\
\quad \texttt{let}\ \ fail_o = \\
\qquad \texttt{let}\ \ fail_i = \mathcal{C}[\![C]\!]^{\mathbf{d}}\,p\,\theta\ \texttt{in} \\
\qquad\quad (\mathcal{A}[\![a_d]\!]\,\theta) \gg (\mathcal{T}[\![T_i]\!]\,\theta\ wrapper_i\,\mathcal{I}d\,fail_i\,fail_i)\ \ \texttt{in} \\
\quad \mathcal{T}[\![T_o]\!]\,\theta\ wrapper_o\,\mathcal{I}d\,fail_o\,fail_o
\end{array}
$$

Component semantics definitions follow. Entry transitions associated with an *Or* node initiate the component and shall not fail (the $\bot$ value is unreachable). Execution or exiting of an *Or* component applies to the active element. Parallel states rely on `fold_right` to ensure proper function compositions.

$$
\begin{array}{l}
\mathcal{C}[\![Or(T, \varnothing)]\!]^{\mathbf{e}}\,p\,\theta = \mathcal{I}d \\
\mathcal{C}[\![Or(\varnothing, [s_0])]\!]^{\mathbf{e}}\,p\,\theta = \texttt{open\_path}^{\mathbf{e}}\,p\,\varnothing\,s_0\,\mathcal{I}d \\
\mathcal{C}[\![Or(T, S)]\!]^{\mathbf{e}}\,p\,\theta = \mathcal{T}[\![T]\!]\,\theta\,(\texttt{open\_path}^{\mathbf{e}}\,\varnothing\,p)\,\mathcal{I}d\,\bot\,\bot \\
\mathcal{C}[\![Or(T, \varnothing)]\!]^{\mathbf{d}}\,p\,\theta = \mathcal{I}d \\
\mathcal{C}[\![Or(T, x.S)]\!]^{\mathbf{d}}\,p\,\theta = \mathcal{I}te(\texttt{active}(p.x), \theta^{\mathbf{d}}(p.x), \mathcal{C}[\![Or(T, S)]\!]^{\mathbf{d}}\,p\,\theta) \\
\mathcal{C}[\![Or(T, \varnothing)]\!]^{\mathbf{x}}\,p\,\theta = \mathcal{I}d \\
\mathcal{C}[\![Or(T, x.S)]\!]^{\mathbf{x}}\,p\,\theta = \mathcal{I}te(\texttt{active}(p.x), \theta_L^{\mathbf{x}}(p.x), \mathcal{C}[\![Or(T, S)]\!]^{\mathbf{x}}\,p\,\theta) \\
\mathcal{C}[\![And(S)]\!]^{\mathbf{e}}\,p\,\theta = \texttt{fold\_right}\ \ (\lambda\,x.\,\lambda\,res.\theta_L^{\mathbf{e}}(p.x)\,\varnothing \gg res)\,S\,\mathcal{I}d \\
\mathcal{C}[\![And(S)]\!]^{\mathbf{d}}\,p\,\theta = \texttt{fold\_right}\ \ (\lambda\,x.\,\lambda\,res.\theta^{\mathbf{d}}(p.x) \gg res)\,S\,\mathcal{I}d \\
\mathcal{C}[\![And(S)]\!]^{\mathbf{x}}\,p\,\theta = \texttt{fold\_right}\ \ (\lambda\,x.\,\lambda\,res.\theta_L^{\mathbf{x}}(p.x) \gg res)\,S\,\mathcal{I}d
\end{array}
$$

Program semantics. The evaluation of the main program produces a transformer:

$$\mathcal{P}[\![(s, Srcs]\!] : Den = \mathcal{I}te(\texttt{active}(s), \theta^{\mathbf{d}}(s), \theta_L^{\mathbf{e}}(s)\,\varnothing)$$

Where θ is built using *Srcs* and the initial environment assumes all states are inactive, including the main one, s.

2.3.2 Comparison with Hamon's denotational semantics

The previous definitions are directly extracted from [77] but were modified to solve minor soundness flaws and to be compatible with the pure CPS semantics we designed. Without developing much about the soundness flaws[2], let us highlight the main differences in the semantics definitions:

- we adapted the rule to match our understanding of nontrivial Stateflow constructs, as exhibited by the current Stateflow simulation engine. For example, sequences of actions performed when leaving a state and entering another one follow a specific order: [77] the use of success and fail continuations were improperly combined. Our encoding introduced a new argument `wrapper` used in $\mathcal{D}$, τ and $\mathcal{T}$. Open and close actions bind dedicated wrappers that reorder actions. Our version follows current Stateflow behavior.

[2]In fairness to this work, it is somehow tricky to figure out what the undocumented semantics of Stateflow circa 2005 may look like and to what extent it was well specified. Notice also that [136] corrects some flaws in Hamon's denotational semantics.

$$
\begin{aligned}
\mathcal{A}[\![\texttt{open}\, p]\!](\rho) &= \rho\,[p \mapsto \texttt{true}] \\
\mathcal{A}[\![\texttt{close}\, p]\!](\rho) &= \rho\,[p \mapsto \texttt{false}] \\
\mathcal{A}[\![v = expr]\!](\rho) &= \rho\,[v \mapsto [\![expr]\!]_\rho] \\
\mathcal{I}te(cond, T, E)(\rho) &= \texttt{if}\ [\![cond]\!]_\rho\ \texttt{then}\ T(\rho) \\
&\quad\ \texttt{else}\ E(\rho) \\
(D_1 \gg D_2)(\rho) &= D_2 \circ D_1(\rho) \\
\mathcal{I}d(\rho) &= \rho \\
\bot &= \texttt{assert}\,\texttt{false}
\end{aligned}
$$

(a) Interpreter

$$
\begin{aligned}
\mathcal{A}[\![\texttt{open}\, p]\!] &= p = \texttt{true} \\
\mathcal{A}[\![\texttt{close}\, p]\!] &= p = \texttt{false} \\
\mathcal{A}[\![v = expr]\!] &= v = expr \\
\mathcal{I}te(cond, T, E) &= \texttt{if}\ cond\ \texttt{then}\ T \\
&\quad\ \texttt{else}\ E \\
(D_1 \gg D_2) &= D_1\,;\ D_2 \\
\mathcal{I}d &= \texttt{nop} \\
\bot &= \texttt{assert}\,\texttt{false}
\end{aligned}
$$

(b) Code Generator

Figure 2.8: Instantiations

- regarding CPS, as explained at the beginning of the Section, the ρ argument is abstracted away and gives rise to point-free semantics. Moreover, as found in Hamon's work, every dynamic access to the environment is removed. For instance, dynamic if-then-else statements are lifted to a dedicated constructor to postpone their execution; similarly, action evaluation is constantly introduced within computed continuations and never evaluated directly (see, e.g., $\tau[\![\cdot]\!]$ and $S[\![\cdot]\!]^{\mathbf{d}}$). This brings far more flexibility in the purpose and design of semantics functions and allows, for instance, to define interpreters, code generators, and source-to-source transformations.

2.3.3 Modular code generation for Stateflow

The formal semantics presented in the previous Section can be instantiated with appropriate definitions for the primitive elements of the denotational semantics: $\mathcal{A}[\![\cdot]\!]$, $\mathcal{I}te(\cdot,\cdot,\cdot)$, $\gg$, $\bot$ and $\mathcal{I}d$.

We present here different settings for the instantiation either as an interpreter or as a code generator. Section 2.3.4 will address our main goal: generate Lustre automata from Stateflow model while preserving the hierarchical structure model.

Interpreter instantiation

The denotational semantics of [77] can be obtained where transformers modify environments: $Den = Env \to Env$. Figure 2.8a details the associated definitions. An environment ρ defines a map from variables to values, including the active status of states. $\rho[v \to c]$ represents the substitution of a variable v to value c in environment ρ. We assume that $[\![expr]\!]_\rho$ represents the evaluation of expression *expr* in ρ, with a Boolean interpretation when evaluating a condition expression. The bottom construct throws an exception but should not happen for well constructed models. We recall that events are part of the environment and are accessible through predicate $active(e)$ using in conditional expressions. Such an instantiation provides a simulator for the model: when provided with an initial environment, it computes the successor environment.

Code generator instantiation

One can also synthesize imperative code by synthesizing an abstract syntax tree while evaluating transformers. *Den* denotes here an abstract syntax tree (AST):

$$\begin{array}{rcl} \mathit{Den} & ::= & \mathit{Den}; \mathit{Den} \\ & | & \texttt{if}\ \mathit{cond}\ \texttt{then}\ \mathit{Den}\ \texttt{else}\ \mathit{Den} \\ & | & v = \mathit{expr}\ \mid\ \texttt{nop}\ \mid\ \texttt{assert}\,\texttt{false}. \end{array}$$

Figure 2.8b provides such simple instantiation. Applying the code generator on the Stopwatch example from Fig. 1.3 generates a relatively large program: about 800 LOC, for an overall number of 220 actions and 135 conditions, nested up to depth 13.

Preserving hierarchical structure

Stateflow semantics is global since the environment is shared among all states. However, the transitions are attached locally to states. We can preserve this hierarchical structure by associating a procedure to each state execution denotation: each call to the denotation $\theta^{\mathbf{e}}(p)$, $\theta^{\mathbf{d}}(p)$ or $\theta^{\mathbf{x}}(p)$ could be respectively substituted by a call to a procedure `thetae_p`, `thetad_p` or `thetax_p` instead of executing $\mathcal{S}[\![p : sd]\!]^{\mathbf{e},\mathbf{d},\mathbf{x}}\,\theta$. This is possible since all arguments of these semantics functions are static (paths, modes, etc.).

For a program $(s, srcs)$, the total code generation is then performed state by state, generating procedures `thetad_p` for each state p declared in program sources *srcs*. The main procedure is the one associated with state s. For the Stopwatch example in Fig. 1.3, it generates 7 procedures, for an overall number of about 270 LOC, 100 actions, and 55 conditions, nested up to depth 7. We have implemented an interpreter and a code generator instantiation of the CPS denotational semantics in OCaml. The code can be found in [41].

In our approach, modularity is itself modular as we can choose either to turn every semantics function into a procedure or on the contrary to inline its results. In this respect, turning junction-related semantics function $\theta^{\mathbf{j}}(j)$, which amounts to computing $\mathcal{T}[\![j : T]\!]\,\theta$ into a procedure helps in factorizing out common prefixes of transition sequences, provided one can defunctionalize [45] its arguments *wrapper*, *success*, and *fail*, expressed as first-order values.

For Stateflow models with complex transition sequences between junctions, this would greatly help factorize common junctions occurring in many paths, avoiding combinatorial blow-ups. However, this is left for future work.

2.3.4 Stateflow models as Lustre automata

This Section describes the compilation of Stateflow models into Lustre automata. Automata are supported since Lustre V6 [40, 39]. The overall behavior of an automaton is pictured in Fig. 2.9. An automaton consists of states, each with its own set of equations and possibly local variables. At each instant, two pairs of variables are computed: a putative *state_in* and an actual state *state_act* and also, for both states, two booleans *restart_in* and *restart_act*,

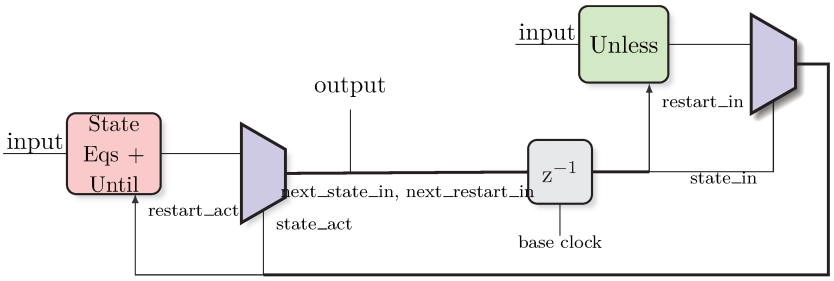


Figure 2.9: Automaton as a pure dataflow.

that tell whether their respective state equations should be reset before execution. The actual state is obtained via an immediate (`unless`) transition from the putative state, whereas the next putative state is obtained via a normal (`until`) transition from the actual state. Only the actual state equations are executed at each instant. Finally, a state reset function is driven by the `restart/resume` keyword switches. A complete presentation of these constructs is given in [62].

We can generate a Lustre AST mimicking the execution of the previous imperative code. The translation is also modular: a Lustre node is built for each component, and each condition within a component is turned into an automaton. The use of intermediate local variables in the Lustre source makes the control-flow explicit. The denotation is defined as $Den = (Name \rightarrow Name \rightarrow LustreAST)$, taking two names *in* and *out* standing for input and output variables and producing a piece of code, assigning output from input. Lustre automata encode the exact semantics of the imperative conditional statements, whereas standard Lustre conditional is a strict operator which does not suit our needs. This instantiation is presented in Figure 2.10, assuming a supplementary node call action `call` p. We denote by $\widetilde{in}$ and $\widetilde{out}$ the sequence of variables present in the Stateflow environment, prefixed by names *in* and *out* respectively. Similarly, $\widetilde{T_{in}}$ and $\widetilde{T_{out}}$ denote their respective types.

Note that the action transformer function generates a set of Lustre flow definitions. Its evaluation combines all atomic transitions of Stateflow of an end-to-end transition into a single Lustre automaton state. No intermediate step is introduced. For the moment, the language of actions is limited to basic Lustre flow definitions but is left free in the general semantics of [79].

Main compilation schema. We have implemented this compilation as a component of the CoCoSim tool framework. To illustrate how the compiler works, let us consider a simple example. Figure 2.11 presents a simple transition between two states A and B. Each state is compiled into a corresponding Lustre automaton state (suffixed with _IDL). As Lustre only allows computations in states, the transition between A and B is compiled into a Lustre state that executes the transition actions. A_EXIT_B_ENTRY thus represents the transition in our example and contains all the actions to be executed: condition actions, exiting state A, transition actions, and entering state B. id indicates the active state at the beginning of each clock. "Set B to active" means "update id to 2".

The Lustre code associated with the previous transition A→ B is presented in the exact Figure. Nested Ite constructs synthesized by our CPS semantics have been merged into a single

$$
\begin{array}{rl}
\mathcal{A}[\![\texttt{open}\ p]\!]\ in\ out := & \widetilde{out} = \widetilde{in}[in\_p \mapsto \texttt{true}] \\
\mathcal{A}[\![\texttt{close}\ p]\!]\ in\ out := & \widetilde{out} = \widetilde{in}[in\_p \mapsto \texttt{false}] \\
\mathcal{A}[\![v = expr]\!]\ in\ out = & \widetilde{out} = \widetilde{in}[in\_v \mapsto [\![expr]\!]_{in}] \\
\mathcal{A}[\![\texttt{call}\ p]\!]\ in\ out := & \widetilde{out} = \texttt{thetad\_p}(\widetilde{in}) \\
(L_1 \gg L_2)\ in\ out := & (L_1\ in\ name_{uid})\,; \\
& (L_2\ name_{uid}\ out) \\
\mathcal{I}d\ in\ out := & \widetilde{out} = \widetilde{in} \\
\bot\ in\ out := & \texttt{assert false}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}te(cond, T, E)\ in\ out := \\
\ \texttt{automaton}\ name_{uid} \\
\ \texttt{state Cond :} \\
\ \ \texttt{unless}\ [\![\neg cond]\!]_{in}\ \texttt{restart}\ NotCond \\
\ \ \texttt{let}\ (T\ in\ out);\ \texttt{tel} \\
\ \texttt{state NotCond :} \\
\ \ \texttt{unless}\ [\![cond]\!]_{in}\ \texttt{restart}\ Cond \\
\ \ \texttt{let}\ (E\ in\ out);\ \texttt{tel}
\end{array}
$$

$$
\begin{array}{l}
\texttt{node thetad\_p}\ (\widetilde{in} : \widetilde{T_{in}})\ \texttt{returns}\ (\widetilde{out} : \widetilde{T_{out}}) \\
\qquad \texttt{let}\ (\mathcal{S}^{\mathbf{d}}[\![p]\!]\ in\ out);\ \texttt{tel}
\end{array}
$$

Figure 2.10: Lustre instantiation

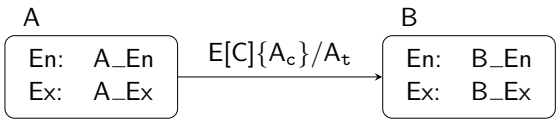
automaton with more states to simplify the presentation.

2.4 Compilation of Lustre code to Simulink models

Few works address the automatic synthesis of MATLAB Simulink annotations from lower-level models or code. In this Section, we present a compilation process from Lustre models to genuine MATLAB Simulink, without the need to rely on external C functions or MATLAB functions. This translation is based on the modular compilation of Lustre to imperative code and preserves the hierarchy of the input Lustre model within the generated Simulink one. We implemented the approach and used it to validate a compilation tool-chain, mapping Simulink to Lustre and then C, thanks to equivalence testing and checking. This backward-compilation from Lustre to Simulink also provides the ability to produce automatically Simulink components modeling specification, proof arguments, or test cases coverage criteria.

Let us focus on safety-critical controller software and systems. In most cases, such systems are designed and implemented as the composition of several reactive components, each performing a specific and relatively simple function. In the aerospace domain, the certification regulation DO178C [1] specifies the different steps of software development and emphasizes the need to *specify requirements* and *verify the validity* of intermediate models or code with respect to their requirements. Therefore, the first question that naturally arises is the following: in a Model-Based System Engineering (MBSE) context, how can system designers specify these requirements and verify the validity of their models?

In addition to models, a contract-based design is a leading methodology for developing component-based software. In this paradigm, each component is associated with a contract specifying its input-output behavior in terms of guarantees provided by the component when its environment satisfies certain given assumptions. These assume/guarantee pairs can thus be used to specify *requirements* at the component level. While Hoare first proposed this approach to specify axiomatic semantics of imperative programs [81], it has been later lifted to reactive systems through the notion of *synchronous observers* [31, 52, 75, 76, 117]. When contracts are specified formally for individual components, they can facilitate several development activ-




```
automaton ab
  state CENTER_POINT:
    unless id=1 and E and C restart A_EXIT_B_ENTRY;
    unless id=1 restart A_IDL;
    unless id=2 restart B_IDL;
    let
      outputs = old_outputs;
    tel

  state A_IDL:
    let
      outputs = A_during_action(old_outputs, inputs);
    tel
    until true restart CENTER_POINT;

  state A_EXIT_B_ENTRY:
    let
      -- execute the actions: condition action,
      -- A exit action, transition action, B entry action
    tel
    until true restart CENTER_POINT;

  state B_IDL:
    let
      outputs = B_during_action(old_outputs, inputs);
    tel
    until true restart CENTER_POINT;
```

Figure 2.11: A simple STATEFLOW transition encoding.

ities, such as compositional reasoning during static analysis, step-wise refinement, systematic component reuse, and component-level and integration-level test case generation.

At the model level, writing requirements with synchronous observers can usually be performed in the same language as the model, easing its deployment and the adoption of the approach by engineers. In addition, these requirements may be verified on the model by simulation or other techniques, such as model-checking. Notice that contracts or synchronous observers attached to model components can also be used to specify additional knowledge, such as invariants computed by a first analysis.

For instance, approaches such as FRET [69] or ArgoSim Stimulus[3] ease the formalization of requirements but can hardly be directly linked to SIMULINK models to provide genuine SIMULINK components representing the specification. On the other hand, the FRET tool can generate these requirements in the CoCoSpec specification language [31], an extension of the LUSTRE language to support assume-guarantees contracts. Formal analysis of SIMULINK models is also

[3] www.argosim.com

addressed by providing a formal semantic of the model, allowing either executable embedded code or a model for analysis by formal tools. In Section. 2.2.1, we provided a way to translate Simulink models into an equivalent Lustre model for which contracts are expressed using the CoCoSpec specification language and verified using model-checking techniques with tools such as Kind2 [83, 31].

While translating Simulink models into formal languages such as Lustre allows us to formally analyze such models, bringing back analysis artifacts generated from Lustre in the Simulink model was not done before our work. It needs a translation of Lustre code to Simulink. These artifacts can include invariants generated from SMT solvers that can be used to infer contracts for sub-components, Lustre annotations such as test-case coverage conditions, and auto-generated CoCoSpec contracts from formalized requirements using FRET, for instance. Connecting Lustre artifacts back to the Simulink model make it simpler for engineers using Simulink to adopt the use of formal tools, and no knowledge of formal languages such as Lustre is required. The compilation of Lustre nodes into Simulink subsystems is performed in two steps:

- The first step is to produce a simplified version of the input Lustre model, preserving the hierarchical structure of nodes. This is done using a dedicated backend we implemented in LustreC [52], an open-source Lustre compiler.
- The second step is to submit the previously produced description to a dedicated backend of CoCoSim that creates Simulink objects of the associated hierarchy of components and connects the corresponding ports in the Simulink model.

Translating Lustre to Simulink, is more challenging when dealing with a complex Lustre abstract syntax tree and requires a simplified version of Lustre equations. For instance, while a basic expression `pre e` could be associated with unit delay, its presence as an argument in a complex expression or a node call is more difficult to tackle. Our idea is to use a Lustre compiler to simplify expressions and produce appropriate constructs. We use LustreC, a Lustre compiler implementing the synchronous dataflow languages hierarchical compilation scheme [13, 24]. The LustreC compiler is implemented as a sequence of transformations and could eventually produce an imperative version of the Lustre input model.

2.4.1 From Lustre to normalized Lustre

LustreC compilation is essentially structured in three main phases. LustreC takes as input Lustre models composed of "classic" dataflow nodes, mixed with hierarchical state machines [17, 40, 62]. Therefore, the first phase of the compiler amounts to producing pure dataflow Lustre by introducing new variables for each automaton to represent its states and encoding transitions in automata as clocked expressions and merges of them. The second phase performs normalization, an updated version that will be detailed in the following. The main idea of this normalization phase in LustreC is to introduce new Lustre variables to encode intermediate values as in classic three-address code. Finally, the last phase translates each normalized node into imperative machine code.

$$
\begin{array}{rcl}
td & ::= & \textbf{type } enum\_ident \ = enum \ \{ \ C_1, \ldots, C_n\} \\
bt & ::= & real \ | \ bool \ | \ int \ | \ enum\_ident \\
d & ::= & \textbf{node } f \ (p) \ \textbf{returns } (p); \ \ \textbf{vars } p \ \textbf{let } D \ \textbf{tel} \\
p & ::= & x : bt; \ ...; \ x : bt \\
\tilde{D} & ::= & pat = \tilde{e}; \ D \ | \ pat = \tilde{e}; \\
pat & ::= & x \ | \ (pat, ..., pat) \\
l & ::= & v \ | \ x \\
\tilde{e} & ::= & l \\
 & & | \ true \rightarrow false \\
 & & | \ op(l, ..., l) \ | \ \textbf{pre } l \\
 & & | \ f(l, ..., l) \ \ | \ \ f(l, ..., l) \ \textbf{every } l \\
 & & | \ f(l \ \textbf{when } C(x), ..., l \ \textbf{when } C(x)) \ \ | \ \ f(l \ \textbf{when } C(x), ..., l \ \textbf{when } C(x)) \ \textbf{every } l \\
 & & | \ \textbf{if } l \ \textbf{then } l \ \textbf{else } l \\
 & & | \ \textbf{merge } x \ (C \rightarrow l)...(C \rightarrow l) \\
 & & | \ l \ \textbf{when } C(x) \\
v & ::= & C :: enum\_ident \ | \ i :: int \ | \ r :: real \ | \ true :: bool \ | \ false :: bool
\end{array}
$$

Figure 2.12: The normalized Lustre syntax

Since the previously presented compilation scheme used by the compiler LustreC is reliable and used to produce trustable C code [13, 65], we adapted it to perform our required simplifications on the Lustre code by modifying the existing normalization stage to produce for each Lustre node a normalized node that can be easily compiled into a Simulink construct, preserving the hierarchy of the initial Lustre nodes. While the original normalization of the LustreC tool was the direct implementation of [13], the updated normalization introduces extra variables and associated definitions for all operators or function calls, including primitive operators such as arithmetic or logical operators.

The normalization process transforms a Lustre model defined into the grammar of Figure 1.5 into one of Figure 2.12. It introduces an additional grammar element l denoting a leaf value, i.e., a variable or a constant. Normalization of an expression returns a fresh typed and clocked variable along with a set of newly bound stateful normalized equations and associated new variables. Except for node calls, these normalized equations do not involve nested constructs and correspond to three-address codes for binary operators. The arguments of node calls are constants or variables, except for a particular case where they are all sampled on the same clock optimized in Simulink block generation, as explained in the following section.

Let us illustrate the normalization of the stopwatch example presented in Listing 1.1 page 26. After normalization, the Lustre code presented in Listing 2.1 is generated. The original Lustre expressions are given in the comments.

Listing 2.1 follows the grammar described in Figure 2.12. The nodes `count` and `stopwatch` are normalized in a classic three-address code so each complex expression is decomposed into simple expressions involving new fresh variables or constants. Each stateful node has a boolean variable `is_init` denoting its first time step defined by `true -> false`. The arrow expression `l1→l2` is replaced by a conditional statement `if is_init then l1 else l2`, see variable `time` in node `count` and streams `__stopwatch_7` and `__stopwatch_2` in node `stopwatch`. Node `stopwatch` contains clocked expressions using `when` and `merge` operators, their semantics is

```
1  node count (tick: bool) returns (time: int)
2  var is_init: bool; __count_2, __count_3: int;
3  let
4         -- Norm. of: time = 0 -> pre time + 1;
5         is_init = true -> false;
6         __count_2 = pre time;
7         __count_3 = __count_2 + 1;
8         time = if is_init then 0 else __count_3;
9  tel
10
11 node stopwatch (tick: bool; toggle: bool; reset: bool) returns (time: int)
12 var running: bool clock; is_init, __stopwatch_5, __stopwatch_6, __stopwatch_7: bool;
13 __stopwatch_1, __stopwatch_2 : int;
14 __stopwatch_3: int when not running;
15 __stopwatch_4: int when running;
16 let
17        is_init = true -> false;
18        -- Normalization of: running = ((false -> pre running) <> toggle) or reset;
19        __stopwatch_6 = pre running;
20        __stopwatch_7 = if is_init then false else __stopwatch_6;
21        __stopwatch_5 = __stopwatch_7 <> toggle
22        running = __stopwatch_5 or reset;
23        -- Norm. of: (0 -> pre time) when not running
24        __stopwatch_1 = pre time;
25        __stopwatch_2 = if is_init then 0 else __stopwatch_1;
26        __stopwatch_3 = __stopwatch_2 when not running;
27        -- Norm. of: count(tick when running) every reset
28        __stopwatch_4 = count(tick when running) every reset;
29        -- Norm. of: time = merge running (true  -> count(tick when running) every reset) (false
              -> (0 -> pre time) when not running);
30        time = merge running
31        (true -> __stopwatch_4) (false -> __stopwatch_3);
32 tel
```

Listing 2.1: Normalized Lustre code of the stopwatch example

explained in section 1.1.3. Stream `__stopwatch_4` (respectively, `__stopwatch_3`) is clocked on `running` (respectively `not running`). The expression `(count(tick when running)every reset)` is not further normalized since it respects the grammar rule
$f(l$ **when** $C(x), ..., l$ **when** $C(x))$ **every** l
described in Figure 2.12. The advantage of keeping it unnormalized is explained in the next section.

2.4.2 From normalized Lustre to Simulink

A normalized node is translated to a SIMULINK Subsystem. We first start translating leaf nodes, then finish with the top nodes. All nodes are translated as Subsystems and used as a library of nodes. If a node `g` calls a node `f` then the Subsystem that corresponds to node `f` is instantiated and used inside the Subsystem corresponding to node `g`.

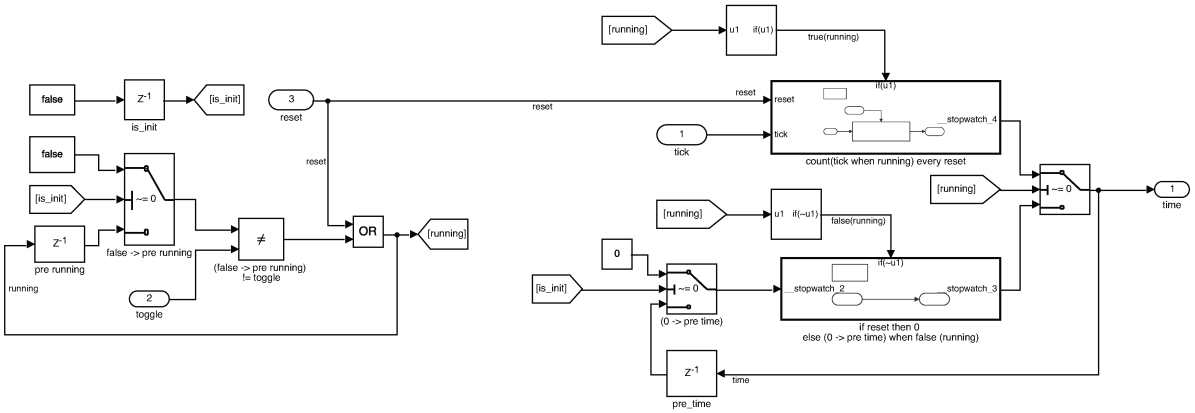


Figure 2.13: The SIMULINK model generated from the LUSTRE example of Listing 2.1.

Each equation definition is mapped to SIMULINK components. Since both SIMULINK and LUSTRE are synchronous dataflow languages, the order of equations is unimportant. When a LUSTRE variable is defined, a *Goto* SIMULINK block is used with the same name as the variable. When the LUSTRE variable is used in an equation, a *From* SIMULINK block is used to read from the signal associated with the same variable. For each LUSTRE variable, there is one *Goto* SIMULINK block that stores the value of the variable and many *From* SIMULINK blocks that read from the *Goto* SIMULINK block.

The generated SIMULINK model of the LUSTRE example of Listing 2.1 is illustrated in Figure 2.13. The user can keep the *Goto* and *From* blocks or remove them and link signals with the same tag to each other. For readability, we kept only the `is_init` and `running` *Gotos*. We note that the generated SIMULINK models are sometimes difficult to read when the LUSTRE source is large. We recommend that engineers always update the LUSTRE files and regenerate the SIMULINK blocks rather than try to edit the generated model directly.

To explain the generated SIMULINK model of Figure 2.13, we will go over the grammar in Figure 2.12 of the normalized LUSTRE code and define the equivalent SIMULINK components. Each case is illustrated in Figures 2.14-2.21.

(a) rule $x = l$, local assignment of a variable or constant to another variable, e.g., `out1 = x` The equation is a simple alias between variables and is modeled in Fig. 2.14. A *From* SIMULINK block is used to read from the signal associated with the variable `x` specified in the tag. A *Goto* SIMULINK block is used to write on the signal associated with the variable `out1` specified in the tag. In the case of an assignment of a constant $x = C$, a *Constant* SIMULINK block is used in place of the *From* SIMULINK block.

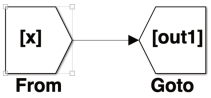


Figure 2.14: Local assignment

(b) rule $x = pre\ l$, state assignment, i.e., a `pre` construct over a variable, e.g., `pre_x = pre x`. Thanks to our modified normalization phase, each `pre` operator argument is aliased to a

new variable, here `pre_x`. In Fig. 2.15, *Unit Delay* acts as a memory, but its initial value, usually specified in SIMULINK, is left unused since, in a valid LUSTRE model, any `pre` is guarded by an arrow construct, preventing its use at the first time step or after a reset.

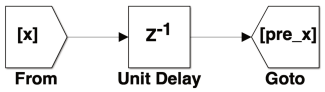


Figure 2.15: `pre` construct

(c) rule $x = true \rightarrow false$, this is the only arrow construct in our normalized LUSTRE, e.g., `is_init = true→false`, and is modeled in Fig. 2.16. The *Unit Delay* block uses its initial condition `true` at the first step and then the previous value of its input for the following steps. Since the input of the *Unit Delay* in Figure 2.16 is the constant `false`, the *Unit Delay* will produce `false` at all times except the initial step defined by `true`. The *Unit Delay* block could be reset to its initial value by an external signal (see Reset block in Fig. 2.22).

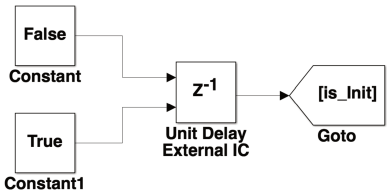


Figure 2.16: Arrow construct

(d) rule $x =$ **if** l **then** l **else** l, conditional statements, e.g., `y = if guard then x1 else x2`. Both `x1` and `x2` run on the same clock; the equation is, therefore, mapped to a switch, as depicted in Fig. 2.17. The *Switch* SIMULINK block uses its second input as a condition. In Fig. 2.17, the condition should be different from zero. The output of the *Switch* SIMULINK block is its first input if the condition is true; otherwise, it is the third input.

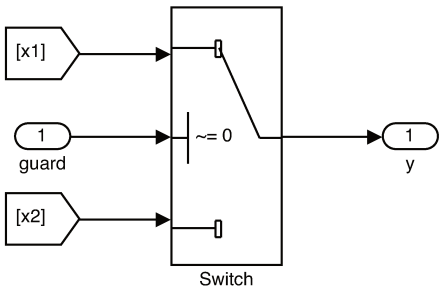
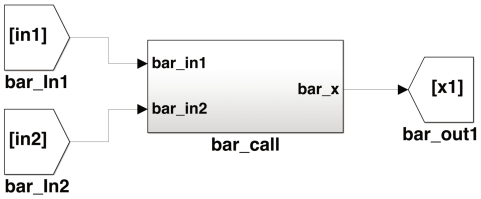
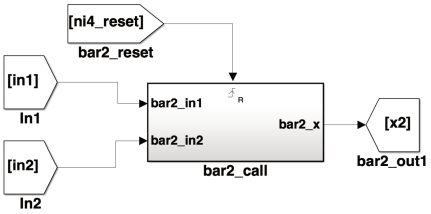


Figure 2.17: Branching construct

(e) rule $x =$ **merge** l $(C \rightarrow l)...(C \rightarrow l)$, merging construct, e.g., `time = merge running (true→__stopwatch_5)(false→__stopwatch_4)`. Since the LUSTRE equations are properly clocked, we can soundly represent the merge as a similar branching construct



(a) Call to a function or node



(b) Call to a stateful node with resettable port

Figure 2.18: Node calls

like in Fig. 2.17 and assume that input expressions `__stopwatch_5` and `__stopwatch_4` are properly clocked (see Listing 2.1).

(f) rules $x = op(l, ..., l)$ or $x = f(l, ..., l)$ where op is a LUSTRE binary or unary operator, and f is a LUSTRE node running on the same base clock as its parent node, e.g., `x = bar(in1, in2)`. This is modeled in Fig. 2.18a where `bar_call` is an instantiation of Subsystem `bar` associated to some LUSTRE node called `bar`. In the case of an operator of the standard library, such as `x = in1 + in2`, the Subsystem `bar_call` would be a basic SIMULINK block, e.g., `Add` block, or `Gain` with scalar -1 for unary minus.

Let us finish with two complex constructs: clocked expressions and resetting nodes.

Clocked expressions Following [13], LUSTRE nodes are considered homogeneous in terms of clocks, i.e., all input and output flows have to be clocked with the same base clock. However, within the node content, internal flows may be clocked on other signal values, either local signals or inputs. Let ck be the base clock of the node. Then, any local clock is defined as a sub clock of this base clock ck. Thus, any expression and equation, including those listed above, are clocked, perhaps implicitly through the application of the clock calculus phase, and subject to the transformations presented here. Clocked expressions are any right-hand side of an equation where the defined variable is assigned a specific clock different from the base clock ck. In the grammar of the normalized LUSTRE defined in Fig. 2.12, the right-hand side expressions that can be clocked are $f(l, ..., l)$ and $f(l, ..., l)$ **every** l if the arguments of node f are clocked on a different clock than the base clock ck, expressions $f(l$ **when** $C(x), ..., l$ **when** $C(x))$ and $f(l$ **when** $C(x), ..., l$ **when** $C(x))$ **every** l, and the expression l **when** $C(x)$. Clocked expressions can be modeled in SIMULINK with an *If Action Subsystem*, a Subsystem whose execution is enabled by an If block. The If block evaluates a logical expression and then, depending on the result of the evaluation, outputs an action signal that enables the execution of the *If Action Subsystem* linked to it. When the latter is not activated, it is configured to either reset its memories (e.g., *Unit Delays* use the initial condition) or hold the previous value. We choose the `Held` feature to ensure that the `If Action Subsystem` maintains its state when not active. For instance, in the expression `count(tick when running)`, the node `count` holds the value of the counter when it is `not running`.

The advantage of not further normalizing the equation `y = f(x when true(c))` is illustrated in Fig. 2.19, the argument `x` do not need to be clocked twice, since node `f` is only

executed when the condition `c` is positive. The equivalent normalized LUSTRE code is `x2 = x when true(c); y = f(x2);`, where both expressions `x when true(c)` and `f(x2)` will be embedded inside two different *If Action Subsystem* with the same condition. The first *If Action Subsystem* is shown in Fig. 2.20, and the second is in Fig. 2.19. This particular construction is also handled since, first, it frequently occurs in our LustreC compilation chain, and second, it exempts us from implementing a *If Action Subsystem* factorization algorithm.

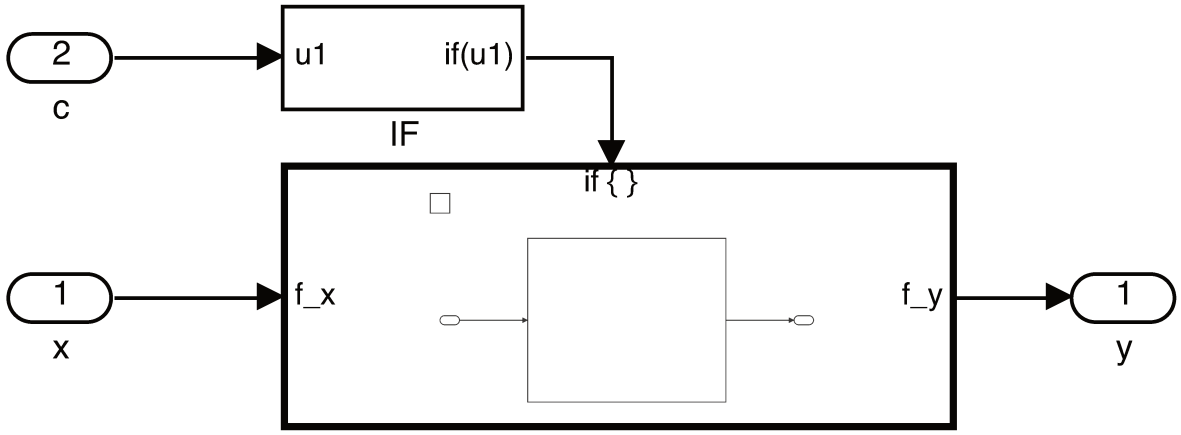


Figure 2.19: Clocked expressions: `f(x when true(c))` or `f(x)` where `x` is clocked on `true(c)`

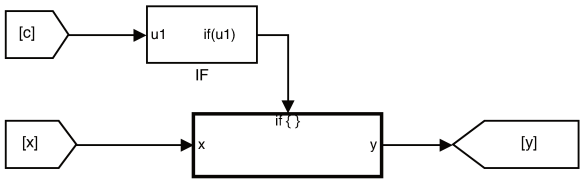


Figure 2.20: Clocked expressions: `x when true(c)`

Note that the expression `f(x) when true(c)` is different since the call to `f(x)` is running on the base clock and then its output is sampled on the clock `c`. The equation `y = f(x)when true (c)` is normalized into `x2 = f(x); y = x2 when true(c);`, the equivalent SIMULINK blocks are shown in Fig. 2.21.

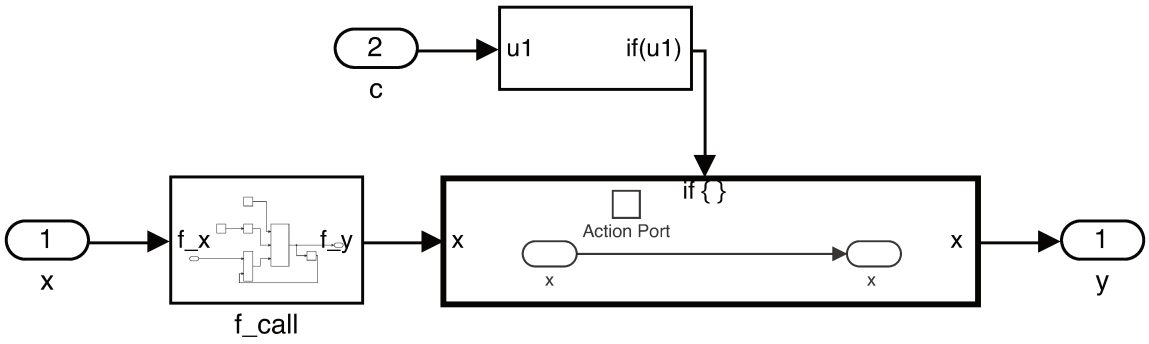
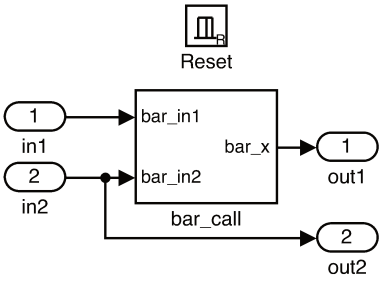


Figure 2.21: Clocked expressions: `f(x) when true(c)`

Stateful nodes and reset In Fig. 2.22, the called node `bar` is stateful: either it contains an arrow function, and typically `pre` expressions, or it calls other stateful nodes.

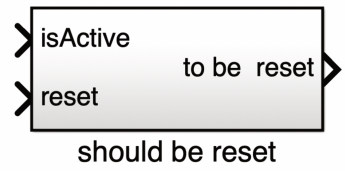



```
node foo (in1, in2: int) returns (out1, out2: int);
var x : int;
let
  x = bar(in1, in2); -- a stateful node
  out1 = x;
  out2 = in2;
tel
```

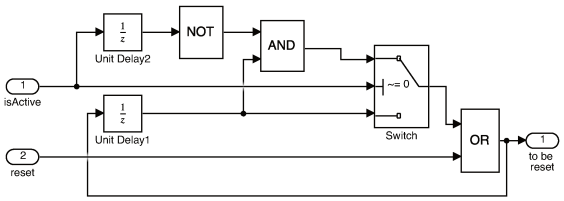
Figure 2.22: Stateful LUSTRE nodes as SIMULINK Resettable Subsystem.

Since `foo` contains a stateful node, it is itself stateful. The definition of the node as a resettable Subsystem, as suggested in Fig. 2.18b, will recursively reset each memory in the node and its children, performing the expected behavior. This produces the SIMULINK diagram presented in Fig. 2.22.

There is, however, a case where this encoding is erroneous. Indeed, in SIMULINK, if a resettable sub-system contains `Action Subsystems` with the `Held` feature, then the reset action is not propagated within this `Action Subsystems`. The only place where we use this `Action Subsystems` in our translation is in the treatment of clocked expressions explained above. Therefore, in the case of a node conditionally reset with an `every cond` and containing sub-expressions defined over sub-clocks, one needs to explicitly propagate the reset signal to these `Action Subsystems`.



(a) The Subsystem keeping track if an `Action Subsystem` should be reset.



(b) The content of the "should be reset" Subsystem

Figure 2.23: Tracking reset state of `Action Subsystem` with `Held` feature.

The generation of Subsystems associated with clocked node calls should then be explicitly extended with reset inputs, adding memory to record the reset status of the node for clocked sub-streams. The Subsystem that keeps track of the reset status is presented in Fig. 2.23. Its output is added as an input of the `Action Subsystem`, the `isActive` input is the condition associated with the `If Action Subsystem` whereas the `reset` input is the reset condition of the parent Subsystem. The output `"to be reset"` is positive when the reset input is positive or the `Action Subsystem` is re-activated and `"to be reset"` was previously active. If the `Action` Subsystem is not active, the previously `"to be reset"` value is kept. For instance, in Table 2.2, we give an example of an execution of the Subsystem presented in Figure 2.23. The `If Action Subsystem` was inactive for the first three steps, and the `reset` signal was active for the second

time step. The `If Action Subsystem` should be reset when it was re-activated at the fourth step, and the `"to be reset"` signal is going to be negative at the fifth step since the `reset` input was not triggered. Thanks to our extensive validation, we are confident that this encoding faithfully addresses this specific case. The validation process is detailed in Section 3.3.2.

`isActive`	false	false	false	true	true
`reset`	false	true	false	false	false
`to be reset`	false	true	true	true	false

Table 2.2: A simulation of the "should be reset" Subsystem of Fig 2.23.

The presented approach enables the translation of Lustre nodes to Simulink subsystems. The proposed algorithm can produce regular subsystems or support the definition of contracts at Simulink level, using boolean flows.

The added value of our approach to alternative approaches such as [92] is the production of basic Simulink subsystems relying only on primitive blocks such as unit delays, merge, and relational and arithmetic operators. It can also address the complete input language of the compiler we used, including clocks and hierarchical definition through multiple Lustre nodes and Lustre automata. The implementation is, however, limited since it does not yet handle machine-level types nor external C functions, while this could technically be implemented since Simulink supports both constructs.

The approach has been validated in large use cases, demonstrating the behavioral equivalence between some compiled models.

The applications are numerous, from validating the framework to supporting formal specification or producing runnable proof evidence as synchronous observers. It is now integrated into the CoCoSim toolbox and is mature enough to be used automatically to provide feedback at the model level.

Future work includes the extension of the input language to enable the use of externally defined functions, such as C code, and the handling of machine data types (e.g., int8, uint8, int16, uint16).

2.5 Conclusion

This chapter describes our contribution to providing a bidirectional translation between Simulink and the Lustre models. We first characterized the semantics translation of Simulink models to Lustre. We then described our denotational semantics for Stateflow. Finally, we also showed how to compile Lustre code to Simulink models. In the next chapter, we will present CoCoSim, a toolbox that uses these translations to ease the V&V activities of Simulink models.

Chapter 3

CoCoSim: A toolbox to ease V&V activities

Contents

It is well-known that the requirements gathering and the validation and verification (V&V) activities are crucial for the success of any software development project. In the context of Simulink models, these activities are even more critical, as the models can be very complex and involve many different stakeholders. In this chapter, we present CoCoSim, a toolbox we developed that aims to ease the requirements gathering and the V&V activities for Simulink models. We also introduce FRET, an open-source tool developed at NASA Ames for writing, understanding, formalizing, and analyzing requirements. Note that FRET is not a contribution of this work, but we collaborated closely with the FRET team to successfully integrate both tools CoCoSim and FRET. The results of this collaboration has been presented in [20, 22, 100, 99, 98]. The main contribution of this chapter is twofold. First, we show how CoCoSim can be used with FRET to formalize requirements as Simulink components, and we describe the benefits of this approach. Second, we present the results of our preliminary evaluation of CoCoSim, which shows that it can correctly translate Simulink models into Lustre synchronous language.

Sections 3.1.2 and 3.1.3 are extracted from [99]. The work about translation validation in Section 3.3.2 is published in [19].

3.1 The need to formalize requirements as Simulink components

Formal verification and simulation are powerful tools to validate requirements against complex systems. However, requirements are developed in the early stages of the software lifecycle and are typically written in ambiguous natural language. There is a gap between such requirements and formal notations that can be used by verification tools and a lack of support for correctly associating requirements with software artifacts for verification. We propose to write requirements in an intuitive, structured natural language with formal semantics and to support formalization and model/code verification as a smooth, well-integrated process. In a research project, inside NASA Ames formal methods team, we have developed an end-to-end, open-source requirements analysis framework that checks Simulink models against requirements written in structured natural language. The framework was built in the Formal Requirements Elicitation Tool (FRET); we use FRET's requirements language named fretish and formalization of fretish requirements in temporal logic. Our proposed framework contributes the following features: 1) automatic extraction of Simulink model information and association of fretish requirements with target model signals and components; 2) translation of temporal logic formulas into synchronous dataflow CoCoSpec specifications as well as Simulink monitors, to be used by verification tools; the correctness of the translation is established through extensive automated testing; 3) interpretation of counterexamples produced by verification tools back at the requirements level. These features support a tight integration and feedback loop between high-level requirements and their analysis. We demonstrate our approach in two case studies in Chapter 4.

The safety-critical industry imposes a strict development process according to which requirements are written in the early phases of the software lifecycle and are refined into models and/or code while keeping track of traceability information. Verification and validation (V&V) activities must ensure that the development process properly preserves these requirements (for example, see [115] and its formal method supplement [116]).

Requirements are typically written in natural language, which is well-known to be ambiguous and not amenable to formal analysis. On the other hand, formal mathematical notations can be used by analysis tools but are unintuitive for developers. Frameworks like Stimulus [82] or FRET (Formal Requirements Elicitation Tool) [70] address this problem by enabling the capture of requirements in restricted natural languages with formal semantics. FRET additionally supports the automated formalization of requirements in temporal logic.

Associating high-level requirements with software artifacts in terms of architectural information, such as components and signals, is necessary to support V&V activities. This is also the case when requirements are formal; for example, the atomic propositions that make up a formula must be connected to variable values or method executions in the target code.

In our work, we presented an end-to-end open-source requirements analysis framework that supports a tight integration and feedback loop between high-level requirements and the V&V of models or code against these requirements. Our framework is built on top of the open-source tool FRET.[1] It connects FRETISH requirements, explained in Section 3.1.2, to SIMULINK models for verification, and verification results back to requirements.

Our framework can be connected to any SIMULINK/LUSTRE V&V tools, although it currently uses our toolbox COCOSIM [15], and the SIMULINK Design Verifier (SLDV). We have applied our framework to a major case study: the Lockheed Martin Cyber-Physical Systems (LMCPS) challenge [53] (see Sec. 4.1), a set of aerospace-inspired examples provided as text documents specifying the requirements along with associated SIMULINK models. Examples range from basic integrators to complex autopilots. We report on our experience using FRET, COCOSIM, and their interconnection to capture and analyze LMCPS requirements.

3.1.1 Synchronous observers to support V&V activities.

In Section 1.3, we defined a synchronous observer as a description of axiomatic semantics for a synchronous model. The observer can be defined in the same language as the model and corresponds to a set of boolean streams. If the property is valid, the output flow encoding the property should remain `true` during the program's execution.

Graphically speaking, a synchronous observer is a subsystem that accesses some internal flows and computes a boolean output. For example, Figure 3.1 performs such computation and verifies that a specific relationship between its two inputs is always valid.

In control theory, we speak about an open-loop property: the property can be expressed over the controller inputs, outputs, or memories without knowledge of the plant semantics. Figure 3.2 presents the association of such a synchronous observer, an open-loop property, attached to a component element.

The content of the observer itself is left free and could be as complex as required, depending on the complexity of the requirement it models. While this notion is expressive enough and is capable of capturing all kinds of requirements, it is sometimes more convenient to refine the specification by expressing hypotheses, i.e., the precondition of the Hoare triples, or modes,

[1] `https://github.com/NASA-SW-VnV/fret`

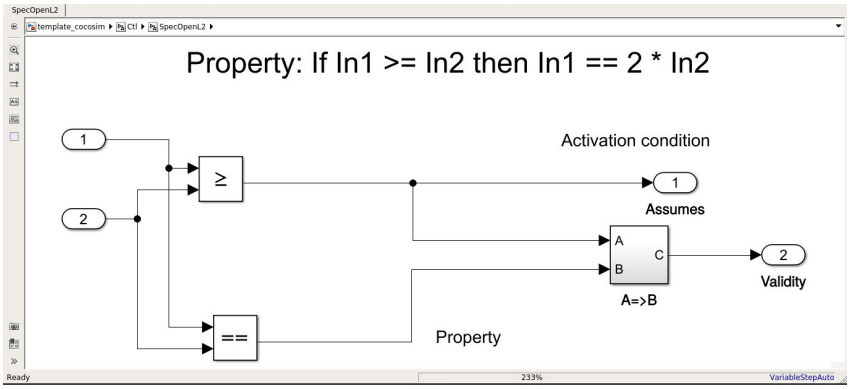


Figure 3.1: A synchronous observer as SIMULINK subsystem.

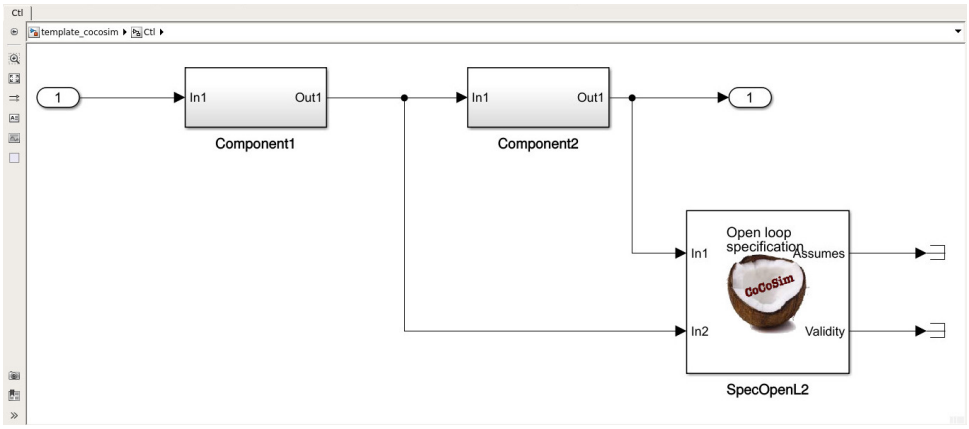


Figure 3.2: Open-loop properties in a synchronous observer

conditional behavior depending on some conditions.

At SIMULINK level dedicated constructs, such as shown in Fig. 3.3, ease the definition of such model-based contracts.

The synchronous observer node's complexity can contain any legal SIMULINK or Scade/LUSTRE content. For example, Figure. 3.4 presents a template to support the expression of closed-loop properties. This observer contains both the plant model and a set of closed and open-loop properties. Within that specification subsystem, observers can access any flows, including the plant' flows.

However, the insertion of the closed-loop specification node within a model is not as convenient as it is for an open-loop property. The open one could be defined only with probes, while the closed one needs, maybe artificially, to reconstruct a feedback loop. This is presented in Figure. 3.5. Note the occurrence of a *specification-based unit delay* to prevent the creation of a spurious algebraic loop.

Once the specification is formalized, as regular SIMULINK components, one can rely on them to support numerous verification and validation activities. Let us look at the example in Figure 3.6 to illustrate these various uses.

This observer only focuses on a very local property: depending on some conditions, the

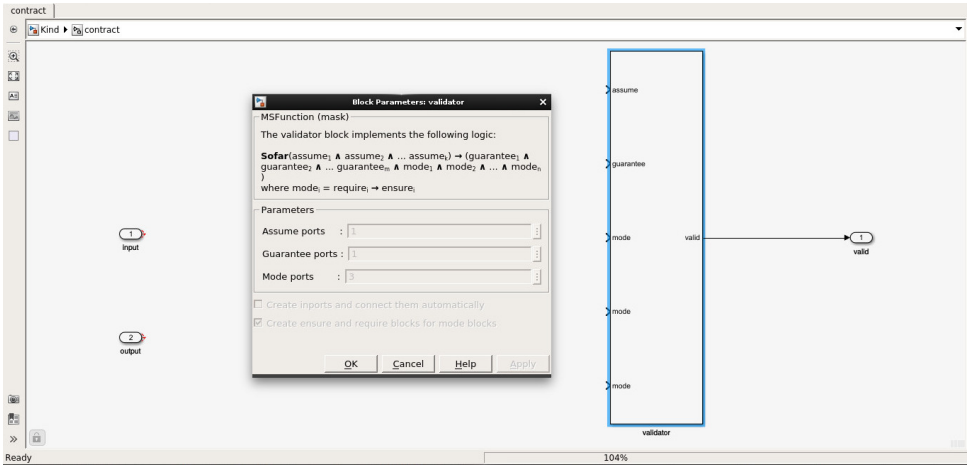


Figure 3.3: Modes as Simulink contracts

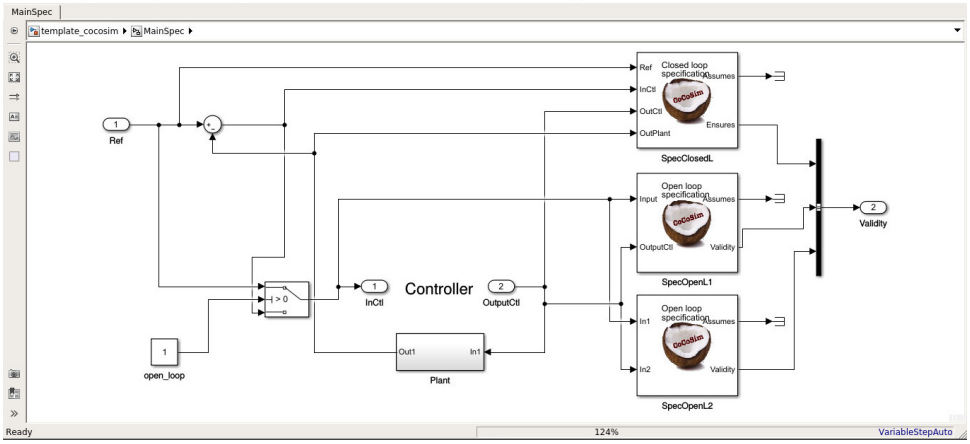


Figure 3.4: Encoding closed-loop properties in an observer

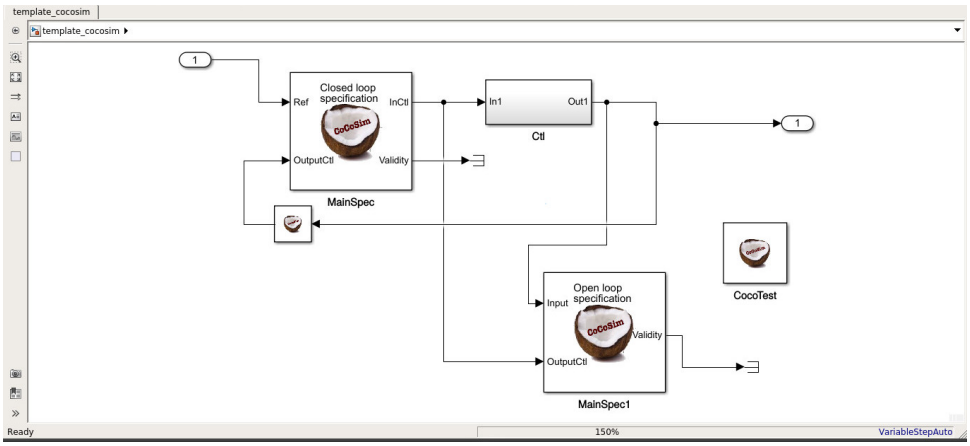


Figure 3.5: Injecting closed-loop observers as model annotations

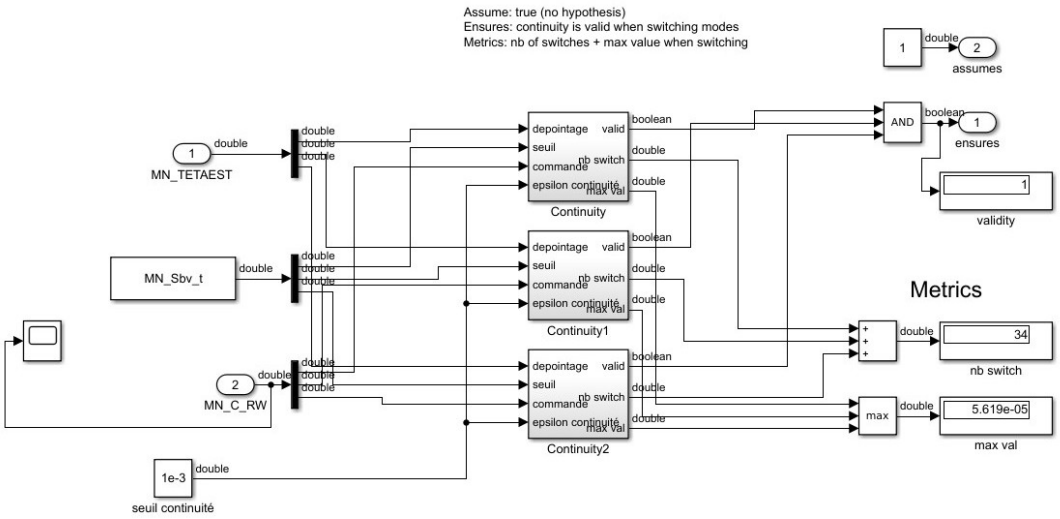


Figure 3.6: Example of a specification

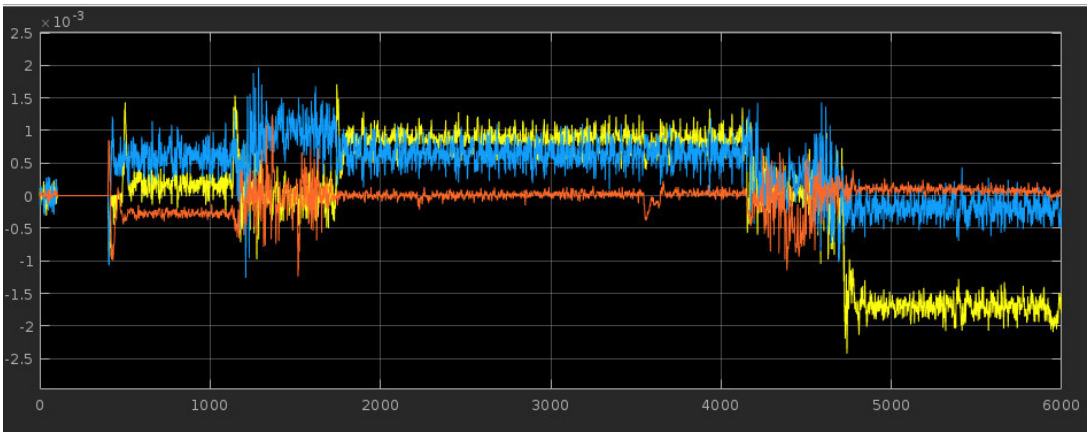


Figure 3.7: Controller simulation.

controller switches between different control laws. This property ensures that the switch is continuous. However, simulations performed on the whole controller leave no opportunity to evaluate the validity of this specific property. Figure 3.7 provides one such run. While one can consider that global behavior is acceptable, it is essential to provide solid arguments for each requirement.

Synthesis of test oracles

Each formalized requirement acts as a test oracle. For example, the synchronous observer defines a predicate. Therefore its boolean output corresponds to the validity of the expressed requirement.

This block is runnable and can be used at various levels. As visible in Figure 3.6 additional elements could be added to the model to visualize the status of the property. In this specific simulation run, the positive value of the output shows that the property was valid during the execution of that single test.

In addition, since our CoCoSim framework can produce C code for Simulink models, the observer itself can be compiled to produce code. This opens the opportunity to produce C code or binaries implementing test oracles.

Computation of metrics regarding test suites

When considering an extensive test suite, it is essential to evaluate the validity of each requirement for each test case but also to measure the coverage of the specification. This is because it can happen, for example, that a test case does not activate a specification. The notion of modes in CoCoSpec is appropriate: one needs to provide figures regarding the evaluation of each mode by a test suite.

Figure 3.6 also provides these elements as internal flows. Each simulation will produce some numerical values denoting the activation of the property or some meaningful values. In this specific case, we compute the number of mode switches, which was 34 in that run, and the maximal value of the discontinuity, which was $5 \cdot 10^{-5}$.

Metrics and coverage of requirements can then be automatized, either at the model level, or at the code level.

Supporting the generation of test cases

Since the property is expressed in the same language as the model, it can be easily expressed in the intermediate language and, eventually, in C. For a particular class of specifications, e.g., blocks limited to boolean and linear integer flows, satisfiability model checkers can search for a sequence of inputs activating a given mode or satisfying a given condition. On the other hand, synthesizing traces containing real/floats values is much more challenging and requires different techniques.

Consistency of specification

Among the possibilities, let us also mention the evaluation or verification of the consistency of the specification. For example, at the contract level, one can ensure that mode constraints are disjunctive or that the mode partitioning is complete, i.e., that their disjunction is always valid.

One can also check the validity of assumes, requires, ensures statements or evaluate whether the expressed predicates are compatible with predicates expressed over sub-components.

We use the CoCoSpec language, presented in Section 1.3, to generate monitors. CoCoSpec [31] is an extension of the synchronous dataflow language Lustre [74]. We recall that CoCoSpec extends Lustre with constructs for the specification of assume-guarantee contracts. Each contract is linked to a node and has access only to that node's input/output streams. The body of a contract contains `assume` (A) and `guarantee` (G) statements as well as `mode` and internal variable declarations. Modes consist of `require` (R) and `ensure` (E) statements. A mode is *active* at time t, if $\bigwedge R = \mathit{true}$ at t. Assumptions and requires are expressions

over input streams, while guarantees and ensures are expressions over input/output streams. A node *satisfies* a contract $C = (A, G')$ if it satisfies $\texttt{H}\ A \Rightarrow G'$, where $G' = G \cup \{R_i \Rightarrow E_i\}$.

The main goal of the open-source CoCoSim framework [15] is to support the analysis of safety-critical Simulink systems. Discrete systems developed in Simulink can be faithfully translated into Lustre [131], the intermediate language of CoCoSim. CoCoSim iterates over Simulink blocks using the Simulink API and produces equivalent Lustre nodes as explained in Section 2.2.1. Different Lustre-based tools can check the validity of the generated Lustre nodes by using SMT-based model checking.

3.1.2 FRET: Formal specification of properties

A fretish requirement contains up to six fields: scope, condition, component*, shall*, timing, and response*, where an asterisk indicates mandatory fields. 'component' specifies the component that the requirement refers to. 'shall' is used to express that the component's behavior must conform to the requirement. 'response' is a Boolean condition that the component's behavior must satisfy. 'scope' specifies intervals where the requirement is enforced. For instance, 'scope' can specify system behavior *before* a mode occurs, *after* a mode ends, or when the system is *in* a mode. The optional 'condition' field is a Boolean expression that triggers the need for a 'response' within the scope. When triggered, the response must occur as specified by field timing, e.g., *immediately*, *always*, *after/for/within N time units*.

Each template is designated by a *template key* with values for fields [*scope*, *condition*, *timing*]. For example, [*in*, *null*, *always*] identifies requirements of the form *In M mode, the software shall always satisfy R*. Condition *null* (as opposed to *regular*) means that the response is triggered at the beginning of each scope interval. The most common key is [*null*, *null*, *always*], i.e., *The software shall always satisfy R*. Scope *null* indicates global scope, meaning the requirement is enforced on the entire execution interval. At the time of this writing, fretish supported eight values for field mode, two values for field condition, and seven values for field timing, for a total of $8 \times 2 \times 7 = 112$ semantic templates. More details on fretish and its semantics are available in [70].

We briefly review the main pmLTL operators (`Y, O, H, S, SI`), which stand for Yesterday, Once, Historically, Since, and Since Inclusive, respectively. `Y` refers to the previous time step, i.e., at any non-initial time, $\texttt{Y}\phi$ is true iff ϕ holds at the previous time step. `O` refers to at least one past time step, i.e., $\texttt{O}\phi$ is true iff ϕ is true at some past time step, including the present time. $\texttt{H}\phi$ is true iff ϕ is always true in the past. $\phi\,\texttt{S}\,\psi$ is true iff ψ holds somewhere at step t in the past and for all steps t' (such that $t' > t$) ϕ is true. Finally, $\phi\,\texttt{SI}\,\psi \equiv \phi\,\texttt{S}\,(\psi\,\&\,\phi)$. Timed modifiers constrain an operator's scope to specific intervals: $O_p\ [l, r]\ \phi$, where $O_p \in \{\texttt{O, H, S, SI}\}$ and $l, r \in \mathbb{N}^+$. For instance, $\texttt{O}\ [l, r]\ \phi$ is true at time t iff ϕ was true in *at least one* of the previous time steps t' such that $t - r \leq t' \leq t - l$. So $\texttt{O}[0, 3]$ restricts the scope of `O` to the interval, including the step where the interval is interpreted and the previous three time steps.

Requirements Language and Temporal Logics. fretish is based on restricted natural-language grammar and allows users to express requirements conveniently. Here is an example requirement in fretish:

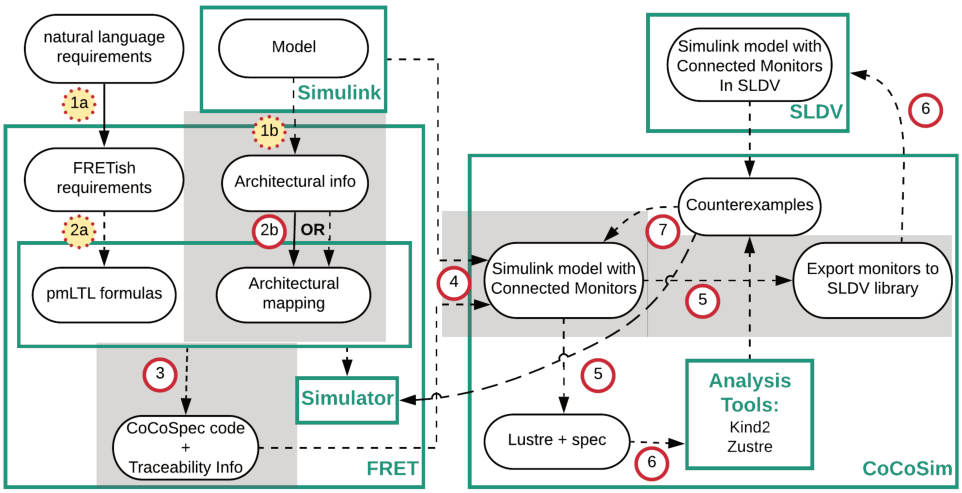


Figure 3.8: FRET-CoCoSim Workflow.

AP-002: In roll_hold mode RollAutopilot shall always satisfy autopilot_engaged & no_other_lateral_mode

For each such requirement, FRET generates a pure Future Time Metric LTL (fmLTL) and a pure Past Time Metric LTL (pmLTL) formalization. The syntax of the generated formulas is compatible with the NuSMV model checker [38]. In this work, we focus on pmLTL, since CoCoSpec assume-guarantee contracts are pairs of past time formula predicates.

3.1.3 FRET-CoCoSim workflow

Figure 3.8 shows the steps of the FRET-CoCoSim workflow. Continuous arrows represent manually-performed steps, whereas steps shown as dashed arrows are performed automatically.

In Step `1a`, requirements are manually written in FRETISH, which are subsequently formalized by FRET (Step `2a`) as pmLTL formulas. Our CoCoSpec code generator COCOGEN (Steps `2b` and `3`) takes pmLTL formulas and model information and generates a CoCoSpec representation. Since COCOGEN is a stand-alone tool does not require FRET, Steps `1a` and `2a` are optional (depicted by dotted circles). We ensure in the workflow that requirements and analysis activities are fully aligned. As a result, 1) Simulink monitors are derived directly from the requirements (and not handcrafted), and 2) analysis results can be traced back not only at the model level but also at the level of requirements so that the requirements engineer can benefit from insights gained through the analysis. We need information bridging the gap between requirement propositions and model signals/architecture. In Step `1b`, this information can be automatically generated from a Simulink model. In Step `2b`, the mapping between the requirements' propositions and the model signals/architecture can be automatically or manually performed. In Step `3`, COCOGEN generates CoCoSpec contracts and traceability information.

The generated files are imported into CoCoSim along with the Simulink model. Then CoCoSim automatically generates a new Simulink model with monitor components generated from the imported contracts connected to the Simulink model using traceability information.

At Step 5, CoCoSim either: 1) generates equivalent Lustre code, which is annotated with the CoCoSpec specification properties that can be subsequently analyzed by the Kind2 and Zustre model checkers (Step 6); or 2) transforms the CoCoSpec monitors into Simulink observers to make the new system analyzable by SLDV. Any counterexample generated during the analysis can be traced back to Simulink or fret for simulation (Step 7).

CoCoSpec Specification Generation

Requirement Examples Next, we provide examples of LMCPS requirements given to us in natural language and show how we wrote them in fretish. Let us consider requirement `[FSM-001]` from the Finite State Machine (FSM) challenge problem, which is an abstraction of an advanced autopilot system with an independent sensor platform. The natural language form of `[FSM-001]` is : "*Exceeding sensor limits shall latch an autopilot pullup when the pilot is not in control (not standby) and the system is supported without failures (not apfail)*". A fretish version of this requirement is:

> **FSM-001**: FSM shall always satisfy (limits & autopilot) ⇒ pullup

where autopilot equals `(! standby & ! apfail & supported)`.

Additionally, let us consider requirement `[AP-004b]`. This challenge problem includes a realistic full six-degree-of-freedom model of the DeHavilland Beaver airplane with an autopilot (AP). The natural language form of `[AP-004b]` is: "*Response to roll step commands shall not exceed 10% overshoot in calm air*".
A fretish version of this requirement is:

> **AP-004b**: in roll_hold mode AP shall always satisfy overshoot ≤ 0.1

Architectural Mapping We need architectural information from the model to generate the CoCoSpec monitors and automatically connect them at the right hierarchical level. For instance, for `[FSM-001]`, we need information about the hierarchical level, i.e., the path, of the model component that corresponds to `FSM`. Additionally, we need information about the ports that form the interface of the model component, i.e., name, port type (e.g., `Inport, Outport`), datatype (e.g., `boolean, double, enum, bus`), dimensions of the port and its width.[2] Our framework provides a mechanism to automatically extract this information from a Simulink model and import it into cocogen.

Once imported, the architectural mapping procedure starts (Step 2b), which includes mapping every component and proposition mentioned in a requirement to a model component and a model port path, respectively. There are two ways to do the architectural mapping: in the fortunate case that the same

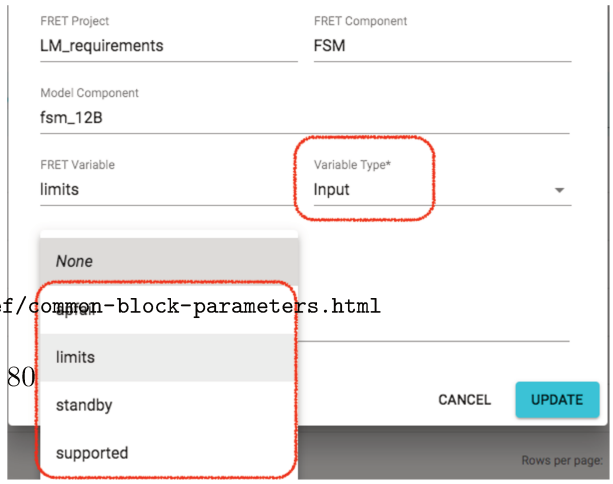


Figure 3.9: Variable mapping.

[2] https://www.mathworks.com/help/simulink/slref/common-block-parameters.html

names are used both in the requirements and in the model, COCOGEN automatically constructs the desired mapping. From our experience, however, this is usually not the case. Different engineers work on requirements and models, and these two parts are hardly ever aligned. For this reason, we provide an easy-to-use user interface in COCOGEN, through which the user can pick the path of the corresponding model component or port from a drop-down menu (see Figure 3.9) and map it with a requirement's component or proposition. Then COCOGEN can automatically identify all the other required information (port types, data types, dimensions, etc.) to generate correct-by-construction monitors and corresponding traceability information. Alternatively, a user may provide the required information manually.

Library of pmLTL Operators in Lustre COCOGEN receives as input a pmLTL formula, which it translates into CoCoSpec code. To facilitate this translation, we have created a library of pmLTL operators in Lustre.

We now present the pmLTL operators `O, H, S, SI`.

```
--Once
node O(X:bool) returns (Y:bool);
let
  Y = X or (false -> pre Y);
tel
```

```
--Historically
node H(X:bool) returns (Y:bool);
let
  Y = X -> (X and (pre Y));
tel
```

```
--Y since X
node S(X,Y: bool) returns (Z:bool);
let
Z = X or (Y and (false -> pre Z));
tel
```

```
--Y since inclusive X
node SI(X,Y: bool) returns (Z:bool);
let
  Z = Y and (X or (false -> pre Z));
tel
```

To support timed modifiers that constrain an operator's scope to a specific interval $[l, r]$, we defined additional Lustre nodes. For instance, for the timed version of `O`, we added the following nodes to the library:

```
--Timed Once: general case
node OT(const L: int; const R: int; X: bool;) returns (Y: bool);
   var D:bool;
let
   D = delay(X,R);
   Y = OTlore(L-R,D);
tel

--Timed Once: less than or equal to N
node OTlore(const N: int; X: bool; ) returns (Y: bool);
    var C:int;
let
    C = if X then 0
        else (-1 -> pre C + (if pre C <0 then 0 else 1));

    Y = 0 <= C and C <= N;
```

```
tel
```

The delay function delays input `X` by `R` time units to define the right bound of the interval in which the valuation of `X` must be checked. Once the input `X` has been delayed by `R` time steps, we can treat the `R` bound as zero and use the `OTlore` (Once Timed less than or equal to) node to check the valuation of `X` in the interval defined by the 0 (current) time step and the left bound `L-R`. `OTlore` is implemented using an integer counter C, which counts the number of time steps that occurred since the last occurrence of property X. If the event has never occurred, the counter keeps its initial value of -1. Below we can see a simple example with $N = 2$ for time steps t=[0..7]:

X	F	F	F	T	F	F	F	F	...
C	-1	-1	-1	0	1	2	3	4	...
Y	F	F	F	T	T	T	F	F	...
t	0	1	2	3	4	5	6	7	...

Other time-constrained operators are defined through `OT` using the usual temporal logic equivalences.

```
-- Timed Historically: general case
node HT(const L: int; const R: int; X: bool;) returns (Y: bool);
let
  Y = not OT(L,R,not X);
tel

-- Timed Since: general case
node ST(const L: int; const R: int; X: bool; Y: bool;)  returns (Z: bool);
let
  Z = S(X, Y) and OT(L,R,X);
tel
```

COCOSPEC Code Generation From the FRETISH version of `[FSM-001]` (Section 3.1.3), FRET generates the following pmLTL formula: `H f`, where `f` is a placeholder for `(limits & autopilot)` $\Rightarrow$ `pullup`. The generated CoCoSpec code uses the data types of the input and output variables provided through the architectural mapping to generate the contract signature.

```
contract FSMSpec(apfail:bool; limits:bool; standby:bool; supported:bool; ) returns (pullup:
    bool; );
let
var autopilot:bool=supported and not apfail and not standby;
guarantee "FSM001" (limits and autopilot) => (pullup);
tel
```

Below we can see part of the generated code for `roll_hold_mode` from requirement `[AP-004b]`. For the complete code, COCOGEN aggregates all requirements that refer to the same mode to generate all `ensure` and `guarantee` statements.

```
var overshoot : real = (roll - step)/step;
mode roll_hold_mode (
    ensure "AP-004b" overshoot <= 0.1;
    );
```

Verifying COCOSPEC Formalizations We assure that the CoCoSpec code generated by our approach captures the intended semantics. We extend the verification framework provided by FRET to check whether the CoCoSpec code of a requirement conforms to the intended FRET semantics. The FRET semantics [70] is compositionally defined based on different valuations of the FRETISH fields scope, condition, and timing. We call *template key* a combination of values of these fields, e.g., [*in*, *null*, *after*] identifies requirements of the following form: *in mode m, the software shall, after 2 seconds, satisfy P*. Our framework uses the following FRET components [70]:

- `Trace_Generator` uses two approaches to produce traces, i.e., example executions. The first approach uses boundary value analysis and equivalence class to define concrete traces that capture interesting relations of template keys. The second approach is based on random trace generation and produces 60000 different random traces in the range [0..12].
- `Oracle` takes a trace and a verification pair $\langle t, \phi \rangle$, where t is a template key and ϕ is its corresponding CoCoSpec formalization and computes the truth value of t on the trace, based on the FRET semantics of t.

For COCOGEN, we have additionally developed the following components:

- `CoCoSpec_Retriever` produces the set of all possible verification pairs $\langle t, \phi \rangle$, that must be checked.
- `CoCoSpec_Evaluator` receives a trace, a verification pair $\langle t, \phi \rangle$, and an expected value e from the `Oracle` and checks whether ϕ evaluates to e on the trace. Essentially, it checks whether the generated CoCoSpec code conforms to the template key semantics for a particular execution trace. For conformance checking, `CoCoSpec_Evaluator` uses the Kind2 model checker.

`Trace_Generator` outputs a trace e. Both ϕ and e are then fed to the Kind2 model checker. We use the *interpreter* Kind2 mode to check conformity, in which Kind2 uses the input valuations from the trace file to evaluate the CoCoSpec formalization at each time step. Since Lustre formalizations are based on past-time formulas, those are evaluated at the end of the trace.

Our verification framework helped us detect discrepancies between the CoCoSpec generated formulas and the intended FRET template key semantics. Despite our deep knowledge of temporal logic operators and their semantics, we detected a problem regarding our definition of the timed once operator in CoCoSpec. Let us consider the generated formalization ϕ that corresponds to the [*null*,*null*,*within*] template key (no scope, no condition, within timing).

```
node test164_3_null_null_within_CoCoSpec (RES: bool) returns (PROP: bool);
  var FTP: bool = true -> false;
  let
    PROP = H((H(not RES)) => OT(1,0,FTP));
  tel
```

In particular, this means that if `RES` has not occurred yet, we are either within 1 step from `FTP` (First Time Point of the trace) or the property is violated. The following discrepancy was reported by our framework over a trace interval [0..12]:

Mode: {[8..11]}; Duration: 2; Response: {[2..4][7..10]}
Discrepancy `null,null,within`: expected: true; Kind2: false.

Since the scope is `null`, ϕ is evaluated throughout the entire trace. Additionally, no condition means that `RES` must occur at time steps 0, 1, or 2. Our initial definition of the (`OT`) operator was non-inclusive of the right bound interval, i.e., $[l, r)$ instead of $[l, r]$, which would omit evaluating ϕ at time step 2.

CoCoSpec contracts to Simulink

An essential step in supporting the formalization of requirements is the capability to add the specification to a model. Most of the tools handling formalized requirements use some formal annotation and formal languages to express these requirements. For instance, the AGREE framework [92] and FRET [69] formalization tool use Lustre and CoCoSpec, respectively, to express requirements. We integrated our work in CoCoSim and connected it with FRET output, automatically translating CoCoSpec contracts generated by FRET to contracts expressed in Simulink and supported by CoCoSim. This work was applied to publicly available industry-provided examples [3] from Lockheed Martin Cyber-Physical Systems (LMCPS) challenges [54, 53] which is a set of aerospace-inspired examples provided as text documents specifying the requirements along with associated Simulink models. Examples range from a basic integrator to complex autopilots. The complete case study and formalized requirements are presented in a detailed technical report [100].

Checking requirements against the Simulink model

CoCoSim attaches CoCoSpec contracts to Simulink subsystems. This process relies heavily on CoCoSim's Lustre-to-Simulink compiler explained in Section 2.4. The first compilation step is performed by LustreC [63], an open-source Lustre compiler, produces information necessary to extract the model structure. The second step transforms the produced structure into Simulink blocks, relying on the Simulink API. Each Lustre node is defined as a Simulink subsystem; hence each node call is transformed into an instance of that subsystem in Simulink. Mathematical operators are translated into equivalent Simulink blocks. The `pre` operator is implemented as a Simulink Unit delay block. CoCoSpec constructs (i.e., assume, guarantee, require and ensure) are also compiled and translated: their equivalent Simulink blocks are provided by a dedicated CoCoSim library [15]. Figure 3.10 illustrates the generated Simulink observer for requirement `[FSM-001]`.

After importing CoCoSpec contracts as Simulink observers, the user may rely on CoCoSim backends to evaluate these contracts; CoCoSim acts as a verification hub, providing easy access to existing solvers. In addition, CoCoSim supports checking the validity of requirements against the Simulink model: transforming the model to an equivalent Lustre model with contracts. Kind2 or Zustre model checkers were then used to check the validity of the properties. In the case of a counterexample, CoCoSim creates a harness model for the user to simulate the trace of the counterexample and, thus, support model or specification debugging.

[3] https://github.com/hbourbouh/lm_challenges

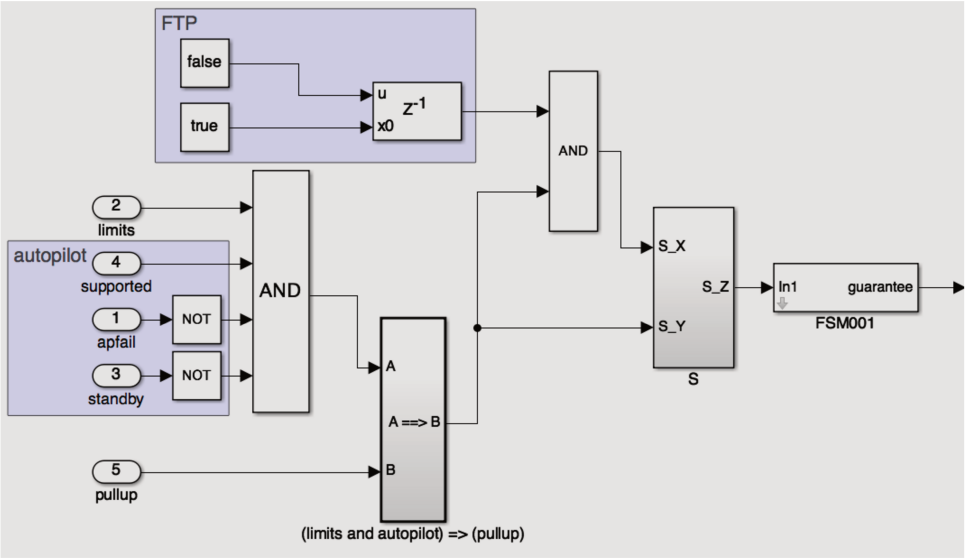


Figure 3.10: The generated Simulink version of requirement `[FSM-001]`.

The contracts can also be transformed into SLDV-compatible verification blocks. SLDV can then be used to prove properties, generate test cases, or evaluate the MC-DC test coverage of a set of tests. The translation technique preserves the structural and modal behavior of the system, allowing us to analyze it compositionally. In addition, CoCoSim carefully handles traceability during model analysis, enabling the simulation of the model's counterexamples.

By translating CoCoSpec contracts into executable Simulink blocks, the latter can be used as monitors for simulation, testing, and runtime verification. Additionally, a mode's required condition (activation) can be used to check whether a mode can be activated or not for a given test suite. Checking the feasibility of a mode activation allows us to perform a form of semantics coverage.

3.2 CoCoSim

CoCoSim [15] is an open-source toolbox for verifying Simulink/Stateflow models. It is integrated within MATLAB as a toolbox and provides easy access to a set of tools. Specification-wise, CoCoSim allows attaching contracts, i.e., synchronous observers, to Simulink subsystems. Contracts are dedicated subsystems relying on boolean dataflows to denote elements of the contract, e.g., assumptions and guarantees. CoCoSim is structured as a compiler and follows a simple schema initially developed for the discrete subset of Simulink [131]. It iterates over blocks using the MATLAB API and produces their equivalent version as Lustre nodes (cf. Algorithm 1 in Section 2.2.1).

Figure 3.11 presents the CoCoSim architecture. In practice, the first preprocessing phase performs model-to-model transformation and replaces some blocks with equivalent but simpler versions. The second phase compiles this simpler version of the Simulink model to Lustre (cf. Section 2.2.1). This modular compilation produces a Lustre node for each Simulink subsystem. Once the Lustre model is obtained, it can be either compiled to C code with

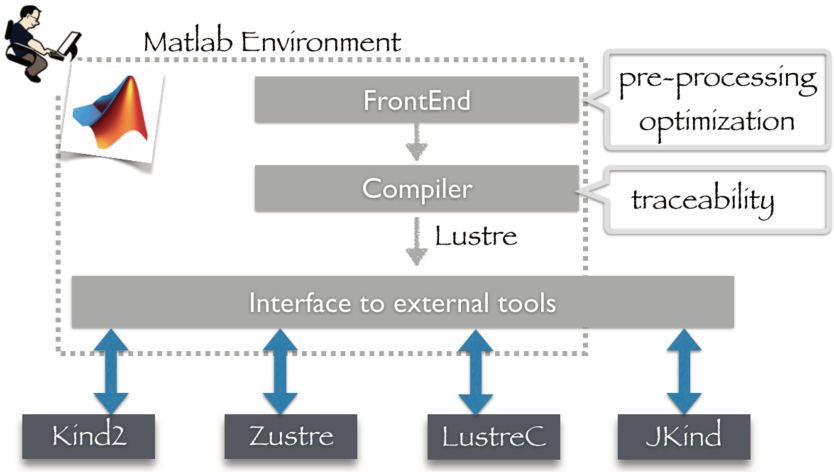


Figure 3.11: The overall architecture of CoCoSim

the LustreC compiler [52] or submitted to Lustre model checkers such as Kind2 [31, 83] or Zustre [62].

CoCoSim carefully addresses traceability issues by manipulating a model along its processing chain. This enables the expression of feedback from model checkers to Simulink models. For example, a counterexample can be replayed at Simulink level using its simulation engine.

In addition, CoCoSim is a highly automated framework for verification and code generation of Simulink/Stateflow models. It consists of an open architecture, allowing the integration of different analyses. It supports different verification techniques to scale to large models. The main objective of CoCoSim is to provide the following:

- A **formal Semantic** for a well-defined subset of Simulink/Stateflow blocks. This formal representation allows the use of formal verification methods and code generation. It also can be used as a semantic reference for other tools.
- A **highly automated** toolchain: all the steps of verification or code generation are automated, beginning with pre-processing the model by transforming some blocks into a set of basic ones, then producing a formal representation of the model in Lustre formalism, and finally calling a verification or code generation tool based on the Lustre code.
- A **Customizable** and **configurable** architecture:

  The translation is customizable and extensible; new blocks can be easily supported.
- Full **traceability** throughout the analysis process: The translation from Simulink to Lustre is modular and preserves the hierarchical structure of the model. In addition, a mapping file between Simulink signals and Lustre variables is generated. This traceability is crucial in reporting analysis results expressed in Lustre back to the user in the context of the Simulink model. For instance, the Counterexamples generated by model checkers are reported back to the Simulink level using a *Signal builder* block to help the user debug the internal Signals values.

- **Scalability** to large models: obtained through the use of various verification techniques and compositional reasoning.

To achieve the above set of objectives, CoCoSim is structured as a compiler, sequencing a series of translation steps eventually leading to either the production of source code or the call of a verification tool. By design, each phase is highly parametrizable through an API and could then be used for different purposes depending on the customization.

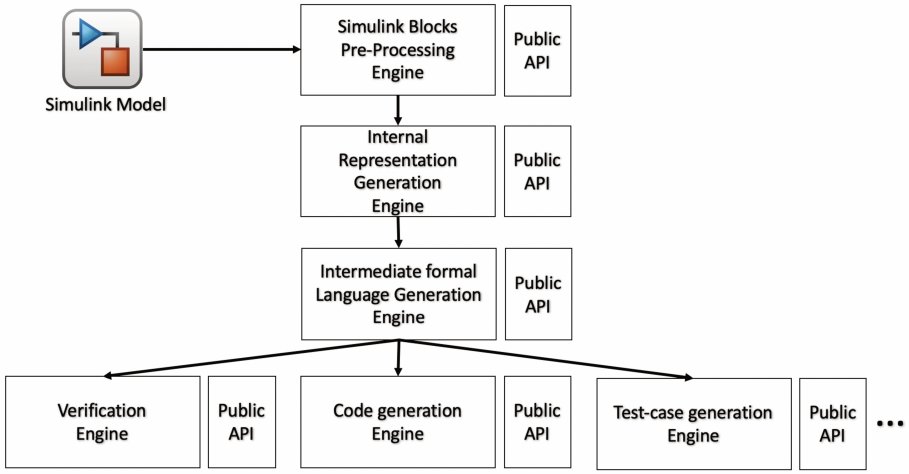


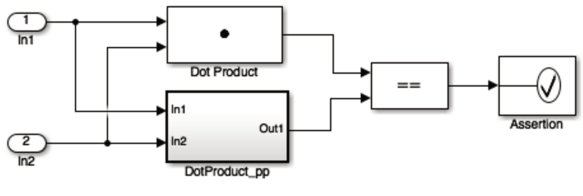
Figure 3.12: CoCoSim framework

Figure 3.12 outlines the translation steps of the tool: First, simplifying the model by substituting complex blocks with a set of basic blocks, then producing an internal representation of the model, and finally generating a Lustre model equivalent to the original Simulink model. Once the Lustre model is generated, external tools can be integrated to do code generation to C or Rust, formal verification of properties, or test-case generation.

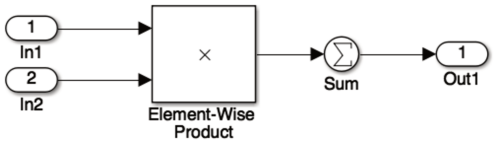
3.2.1 Pre-processing blocks engine

The pre-processing engine is an independent library in CoCoSim that takes a Simulink model as an input and produces a simpler yet equivalent version of that model. In addition, Simulink provides many predefined block libraries (e.g., *Integrator*, *Saturation*, *Dot Product*, *Transfer Function*) that help the user build the model faster. These predefined block libraries can be expressed using basic blocks such as Product and Sum. For instance, Fig. 3.13 illustrates the simplification of the *Dot Product* block by combining an element-wise product of two vectors and the sum of the elements of the resulting vector.

The pre-processing engine consists of a set of libraries performing a simplification of an atomic block. Each library is a MATLAB function defined in a separate file that takes as an input the name of the model; it performs transformations on it using Simulink API functions such as *find_system*, *add_block*, *delete_block*, *get_param* and *set_param*. CoCoSim provides a configuration file where the user can change the order of libraries execution. The order is essential when a block's transformation results in a new block that needs to be handled by



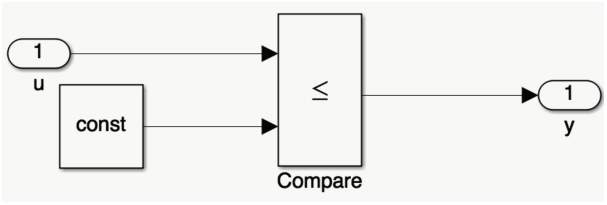
(a) Dot Product block and its substituted version which their equivalence can be checked using SLDV



(b) The content of DotProduct_pp Subsystem in Fig. 3.13a

Figure 3.13: Example of simplifying Dot Product Simulink block.

(a) Compare To Constant block from Simulink library.



(b) Definition of the block 3.14a

Figure 3.14: Example of a non-atomic block and its content.

another library. The user can also define new libraries by adding the MATLAB function file and setting the execution order in the configuration file.

Moreover, CoCoSim provides means of verifying the equivalence between the original model and the pre-processed model. For example, in Fig. 3.13a, we proved the equivalence between *Dot Product* and its simplified version using Simulink Design Verifier (SLDV).

The main objective of this pre-processing step is to reduce the number of blocks that need to be translated to Lustre. Therefore, we mainly pre-process atomic blocks into basic blocks that the translator will support later.

Masked-subsystems: In Simulink, block libraries can be atomic blocks (e.g., *Integrator*, *Saturation*, *Dot Product*, *Transfer Function*) or masked-subsystems with a unique MaskType (e.g., *Compare To Constant*, *Saturation Dynamic*). A Masked-subsystem is a subsystem encapsulated as a block with its own parameters and creates a complex block based on basic ones. For example, Fig. 3.14 illustrates the content of *Compare To Constant*.

CoCoSim considers masked-subsystems blocks as subsystems; therefore, there is no need to pre-process or support them later in the translator; they will be recursively translated as Lustre nodes. The masked-subsystem is then supported if its content is also supported. This approach enables automatic support of a large set of blocks defined as masked-subsystems in the Simulink library.

3.2.2 An internal representation of Simulink models

The next step after pre-processing SIMULINK blocks is the generation of a data structure containing all needed information about the model, its blocks, and configuration parameters. The generation of this representation has two advantages. (1) We use SIMULINK API to get all model information at once (to avoid simulating the model many times), therefore is no need to parse the textual format of SIMULINK model that changes with every new SIMULINK release. (2) The representation can be exported as a JSON file, and external compilers can be integrated with the toolbox based on this representation. For instance, a JAVA compiler was developed by IOWA university based on the exact representation, and it is integrated with COCOSIM as an alternative to our compiler.

3.2.3 Semantic translation of Simulink/Stateflow Diagrams

The main algorithm of our translator is explained in Algorithm 1 in Section 2.2.1.

Supported Simulink blocks COCOSIM supports most frequently used SIMULINK blocks libraries (more than 100 blocks) either by transforming them to simpler blocks (see Sec.3.2.1) or by direct translation to LUSTRE. The translator is developed in MATLAB with a visitor design pattern. New SIMULINK blocks translation can be easily added, and existing SIMULINK blocks translation can be customized or easily modified.

Supported Stateflow blocks Our implementation in COCOSIM supports the most known STATEFLOW constructs such as *states* (Exclusive (OR) and Parallel (AND) states), most used *state actions* (entry, during, and exit actions), *transitions* with the following transition labels: event only, event and condition, condition only, action only, or event and condition and action combined. In addition to these constructs, COCOSIM supports default transitions, inner and outer transitions, self-loop transitions, inter-level transitions, connective junctions, and history junctions. COCOSIM also supports basic data types (int, real, and booleans) and also arrays with static indexes (e.g., [2]). More specific charts [4] are supported such as: flow charts, STATEFLOW functions (graph functions), Hierarchical states, *enter*, *exit*, *send* and *after* operators.

The current version of COCOSIM does not support events emission in actions (state action or transition actions); it supports instead the *send* operator that can replace events emission. We also assume the model does not have an unbound behavior (such as loops in junctions). In addition, transitions with more than one event are not yet supported.

Stateflow Datatypes Our work provides global semantics to STATEFLOW, which we believe is hard enough. We do not address specific details such as how to represent SIMULINK datatypes in the target language LUSTRE. Several simple solutions nevertheless come to mind. One could provide each of these types as an external LUSTRE library, hiding their implementations at the cost of making explicit all the implicit coercion in the SIMULINK semantics. Another solution

[4] `https://github.com/coco-team/regression-test`

would be, for instance, to map every specifically-sized integer type to the unspecified LUSTRE integer type (that is what we currently do) and then annotate the LUSTRE program to take care of sizes when generating C code or Horn clauses.

3.2.4 CoCoSim Backends

All COCOSIM analyses are performed on the generated LUSTRE code, and the results are expressed back at SIMULINK level. We sketch here the features of the connected tools. At the current moment, all tools are open-source and freely available.

3.2.4.1 Formal Verification: SMT-based model-checking.

The model-checking formal method is a process of checking whether a given system satisfies a desired property. In software verification, a system is typically a program or a piece of software, and the desired property is a condition the program should satisfy.

To check whether a program satisfies a given property, the model checker first builds a model of the program. The model is a simplified program representation that captures its essential behavior. The model checker then checks whether the property holds in the model. If the property does not hold in the model, then the program does not satisfy the property.

Once requirements have been expressed as COCOSPEC [30] by using COCOSIM libraries attached to the SIMULINK model, different tools can perform SMT-based model-checking and check their validity. If the property supplied is falsified, COCOSIM provides a means to simulate the counterexample trace in the SIMULINK environment.

- Kind2 [32] is a powerful tool that implements multiple algorithms, including k-induction [122] IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness)/PDR (Property Directed Reachability) [26, 25] as well as on-the-fly invariant generation. These can be performed with various SMT solvers: CVC4, Z3, and Yices.
- Zustre relies on the LustreC modular compiler. The input LUSTRE model is compiled in a Horn encoding [61, 62] describing the transition relation.

  This transition system, along with a similar expression of its requirements, is analyzed with Spacer [88, 87], a PDR algorithm integrated into Z3.
- JKind [59] is a similar model-checker developed at Rockwell Collins.

3.2.4.2 Code generation.

While code generation is not the principal goal of COCOSIM, some of its backend tools provide such a feature. E.g., LustreC [63] is an implementation of the modular compilation scheme [13] used in SCADE. It preserves the hierarchy of the initial model, easing the checking of traceability between LUSTRE and generated C code.

Kind2 [32] is, first of all, a model-checker, but it can also produce Rust code from the provided models.

3.2.4.3 Test cases generation.

LustreT is based on some compilation stages of LustreC [63]. It provides two different methods to perform test generation [64]:

- coverage criteria based reachability using bounded-model checking (BMC).
- generation of mutants and synthesis of discriminating test cases.

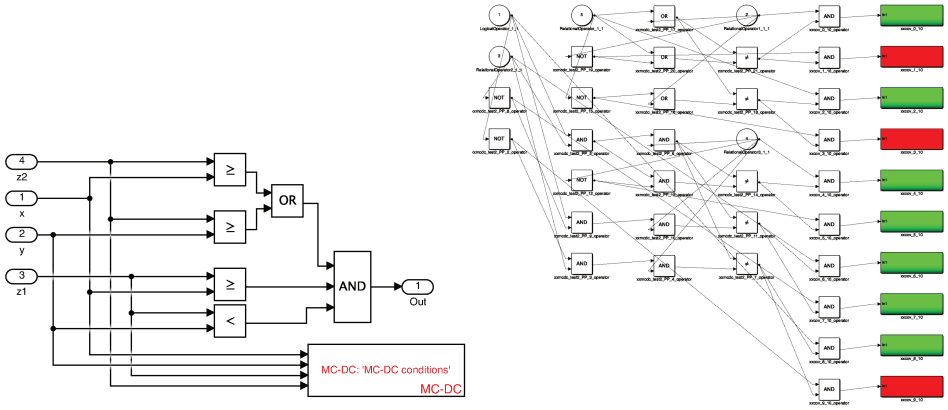
In the first case, coverage criteria such as MC/DC is expressed as a reachability problem. For example, an atom of a boolean predicate has to be true at some point. Then we check the validity of the negation of that property. Model-checker such as Kind2 or Zustre will then perform BMC or, possibly, exhibit a counterexample, a test case activating that specific criterion. An MC/DC criterion will then be mapped to a large set of predicates. The test generation process will populate a set of test cases activating each of these conditions.

The second approach relies on the notion of mutants. Usually, mutants are used to evaluate the quality of a test suite. I.e., generate a set of mutant programs and apply different test suites; a good test suite distinguishes valid programs from mutants. Here the approach is different. After generating mutants, we use the same BMC tools to build a test case that will distinguish them. This does not always succeed since some mutations may be invisible or in dead code. But considering the large set of mutants, this approach is efficient at building test cases.

Listing 3.1 provides an example of generated local MC-DC coverage conditions. In addition, all MC-DC coverage conditions can be added to a subsystem as a synchronous observer (cf. Fig 3.15a). We use it to calculate the coverage of a given test suite by simulation. For example, Fig 3.15 illustrates all 10 MC-DC conditions of the expression *out* in Listing 3.1.

```
1 node top (x, y, z1, z2: int) returns (out: bool)
2 var __cov_1: bool;
3 let
4   out = ((((z2 >= x) or (z2 >= y)) and (z1 >= x)) and (z1 < y));
5   --The following is a special annotation:
6   (*! /coverage/mcdc/:  __cov_1; *)
7   __cov_1 = ((z2 >= y) and (((z2 >= x) or (z2 >= y)) != ((z2 >= x) or (not ((z2 >= y))))));
8 tel
```

Listing 3.1: Example of a Lustre annotation.



(a) MC-DC observer attached to its subsystem.

(b) MC-DC conditions subsystem from inside. Each condition is colored green if it was covered during simulation, red otherwise.

Figure 3.15: MC-DC conditions were generated and annotated to the original Simulink model.

3.2.4.4 Invariant Generation

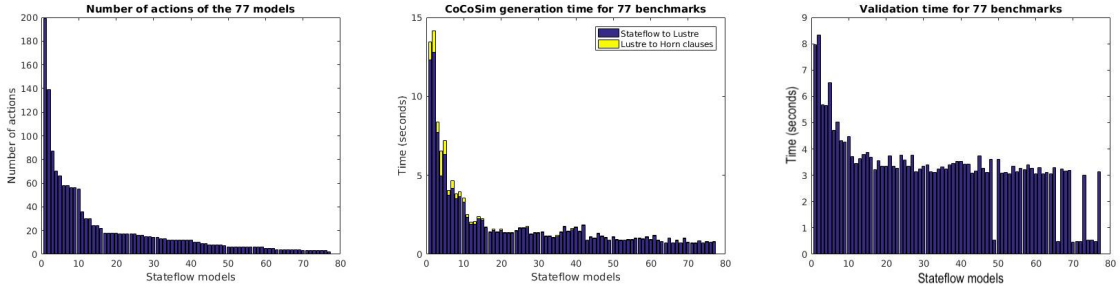
The initial motivation for this work came from using the property-directed reachability-based tool Zustre [61] to analyze synchronous observers associated with Simulink models by translating them to Lustre before analyzing them. The Zustre model checker can provide a counterexample in case of failure and also returns a set of invariants in case of success. However, while traceability information was sufficient to execute the counterexample on the initial Simulink model, the expression of the produced invariant as runnable evidence was impossible. More specifically, the hierarchy-preserving encoding of the Lustre model into the model checker provides a set of local invariants that could be attached to Simulink subsystems in case of success. As an example, Listing 3.2 presents such a generated local invariant (can be read as `(pre(time)>= 0 => time >= 0)`). The Lustre to Simulink translation, described in Section 2.4, allows attaching this property to a subsystem as a synchronous observer.

```
1 node inv(toggle, reset : bool; time:int;) returns (inv:bool);
2 let
3  inv = true ->  (time >= 0 or  not (pre(time) >= 0));
4 tel
```

Listing 3.2: Example of a generated Lustre invariant for Stopwatch example.

3.3 Experimental evaluation of translation correctness

We tested CoCoSim on a large set of automatically generated Simulink models. We also ensured that all components of CoCoSim architecture were tested individually. For instance, every library in CoCoSim pre-processing step is validated individually using Simulink Design Verifier as described in Section 3.2.1.



(a) (b) (c)

Figure 3.16: Runtime and validation experiments of CoCoSim.

The following details how we tested our Simulink and Stateflow to Lustre compilers.

3.3.1 Stateflow experimental evaluation

We have performed experimental evaluations to demonstrate the effectiveness of Stateflow translation to Lustre. Our experiments are carried out in a machine with the following specifications: Intel Core i5, 8Go RAM, 750Go HD, with Ubuntu 16.04. We used CoCoSim v0.1, which runs on Matlab 2014b and up. However, in this experiment, we used Matlab 2016a with Stateflow version 8.7.

The first experiment was performed to illustrate the soundness of our compilation scheme. In particular, we would like to answer the question "*how faithful is the code generated via CoCoSim*?". To answer this question, we have compared the C code generated via CoCoSim with the one generated by Mathworks' Simulink ® *Coder*^{TM} [95]. The latter is a popular product routinely used in different industrial applications for code generation. Specifically, we have used a set of randomly generated test vectors with 100 iterations to validate the code generated by CoCoSim against the one generated by Simulink ® *Coder*^{TM}. Such validation process is given as an option in CoCoSim. This allows a user to validate that the compiled code via CoCoSim conforms at least with the code generated by Simulink *Coder*^{TM}. It also allowed us to demonstrate that other translation tools, such as sf2lus [119], do not respect the full Stateflow semantic compared to ours.

In our experiments, we have used a set of 77 Stateflow models [5] to evaluate the soundness and runtime of the compilation. Fig. 3.16a shows the size of the different models: the number of actions is the sum of all state actions (entry, exit, and during actions), condition, and transition actions in the model. Fig. 3.16b illustrates the amount of time the compiler took to generate first Lustre code and then Horn clauses (used for verification). The models are given in decreasing number of actions. We can easily observe that generation time increases with the number of actions of the models. On average, CoCoSim takes about 1.85s to generate Lustre code. Fig. 3.16c shows the times for the validation script. On average, the validation process takes about 3.35s.

[5] `https://github.com/coco-team/regression-test`

models	# props	# safe	# unsafe	# timeout	safe (time)	unsafe (time)
Microwave	15	15	0	0	65.51	0
NasaDockingApproach	4	3	0	1	360	0
GPCA_System_Monitor	1	1	0	0	0.64	0
GPCA_Logging	1	1	0	0	4.88	0
GPCA_Top_Level_Mode	3	3	0	0	36	0
GPCA_CONFIG	1	0	1	0	0	19.34
GPCA_INFUSION_MGR	7	5	0	2	596.51	0
GPCA_Alarm	8	0	6	2	0	281.12

Figure 3.17: Experimental results of safety verification on a set of use cases.

Safety verification

The second experiment was performed to illustrate our approach's effectiveness in verifying safety properties. We have used three sets of use cases. The first one is a Stateflow model of a microwave that captures the modal behavior of a typical microwave control software [6]. CoCoSim verified 15 properties of this model using the backend solver Zustre [61] as the backend solving engine. The second use case is a Stateflow model that captures the complex behavior of the Space Shuttle when docking with the International Space Station (ISS) [118]. As the shuttle approaches the ISS, it goes through several operational modes related to how the shuttle is to orient itself for capture, docks with the ISS, and captures the ISS docking latch, among several other operating modes. The model describing this behavior is quite intricate and consists of hierarchical and parallel state machines with three levels of hierarchy and multiple parallel state machines, including a total of 64 states. Using CoCoSim, we verified 2 out of 4 safety properties.

The third use case is the Generic Patient Controlled Analgesic infusion pump system. It consists of four main components: *Alarm*, *Infusion*, *Mode* and *Logging*. A more detailed description of the model can be found in [71]. Figure 3.17 summarizes the safety verification experimental evaluation results using CoCoSim.

3.3.2 Simulink translation correctness evaluation

To validate our translation, we first automatically generated a set of regression tests based on the set of atomic blocks we support. Then, we generate the most common variations of the block parameters for each atomic block. The variations include the datatypes and dimensions of the inports, the block parameters, and the block inside a conditionally executed subsystem. We generated around 2000 tests for an average of 16 variations per block.

Since Simulink is not formally defined, we cannot prove the translation is sound. So instead, we used two techniques, equivalence testing and equivalence checking, explained below.

[6] Rockwell Collins developed this model

	#loc	#nodes	#vars	Simulink Design Verifier	Lustre Model-Checking Monolithic	Lustre Model-Checking Compositional
89 benchmarks [L]	289181	160	43001	87 S + 2 U	87 S + 2 U	152 S + 8U
TCM [S]	2040	91	1239	U	U	79 S + 12 U
ROSACE [S]	926	94	520	U	U	87 S + 7 U
FADEC [S]	68	1	46	U	S	S
AOCS [S]	3649	93	3390	U	S	79 S + 14 U

S: Safe (proven valid), **U**: Unknown, i.e., unable to conclude with model checkers; to be evaluated through tests. [L] use cases were initially defined in Lustre, [S] in Simulink. Note the larger set of cases in the last column since it considers all subnodes as intermediate challenges.

Table 3.1: Experiments on Simulink/Lustre Equivalence checking.

Equivalence testing

Since the Lustre model can be compiled to C (and then executed as a binary) and Simulink model can be simulated, we can perform equivalence testing; for generated random inputs, we compare Simulink simulation outputs against Lustre outputs. Moreover, the translation from Simulink to Lustre preserves the hierarchy, thus enabling the evaluation of possible discrepancies in the outputs individually at the node/subsystem level. When the main model is invalid, testing the sub-components helps track which subsystem or node is the leading cause of the error. Because of the difference in the precision of floating point values between Simulink and C code generated from Lustre, signals are compared with respect to a small threshold (by default 10^{-15}). We automated this validation, which runs daily on the generated regression tests.

Equivalence checking

In this technique, we are using our compiler from Lustre to Simulink that shares no code nor algorithm with our translation Lustre to Simulink.

Our approach was evaluated on a set of case studies, from small benchmarks taken from our regression test suite to industrial ones, using equivalence checking, which will be defined in the following.

Let us denote by $\mathcal{L}$ and $\mathcal{S}$ the sets of Lustre and Simulink models. Model semantics is denoted by $[\![\cdot]\!]_L$ for $L \in \mathcal{L}$ and $[\![\cdot]\!]_S$ for $S \in \mathcal{S}$. Each function is a (possibly stateful) map from a set of n typed input flows to m typed output flows, e.g., $[\![L]\!]_L : \mathcal{T}^n \rightarrow \mathcal{T}^m$.

The equivalence checking aims to verify that two models in the same language are equivalent. For Lustre models $L_1, L_2 \in \mathcal{L}$, we say that L_1 is equivalent to L_2, denoted by $L_1 \equiv_L L_2$ iff for any input flow $i \in \mathcal{T}^n$, $[\![L_1]\!]_L(i) = [\![L_2]\!]_L(i)$. The property $\equiv_S$ is defined similarly on Simulink models. Behavioral equivalence can be evaluated either at the Simulink level with Simulink Design Verifier or at the Lustre level with the Kind2 model checker [31, 83]. It can also be assessed through tests when formal tools cannot conclude.

One can consider the compilation process of the CoCoSim toolbox from Simulink to Lus-

TRE as a map $S2L : \mathcal{S} \to \mathcal{L}$. Similarly, the algorithm proposed in Section 2.4 characterizes a map $L2S : \mathcal{L} \to \mathcal{S}$. We assume that $S2L$ is sound, and we specifically target the validation of $L2S$. Therefore, we consider the following verification challenges:

$$\text{for all model } S \in \mathcal{S}, \quad S \equiv_S L2S \circ S2L(S) \tag{3.1}$$

$$\text{for all model } L \in \mathcal{L}, \quad L \equiv_L S2L \circ L2S(L) \tag{3.2}$$

In both cases, there is a single call to $L2S$, and the function $S2L$, implemented in the CoCoSim toolbox, shares no code or algorithm with the LustreC tool that is used to implement the function $L2S$.

The results are presented in table 3.1. The first three columns give metrics about the size of the models. Simulink Design Verifier was applied to the top level of the system. Lustre equivalence checking, cf. eq. (3.2), using Kind2, has been used both on the main node and modularly, considering each sub-node as a verification target.

Concerning the 89 Lustre benchmarks from our regression suite (see the first line of table 3.1), 85 benchmarks contain only one large top node with 1600 code lines and 465 variables on average. Since there is a single top node in these benchmarks, applying compositional verification does not improve the results. Table 3.1 shows that model-checking using a global (top-level) encoding both in Simulink Design Verifier and Lustre was able to prove 87 benchmarks but unable to prove the validity of two remaining benchmarks.

We applied compositional verification on the two remaining benchmarks that were hard to prove due to their complexity. For the first benchmark, 9 out of 16 nodes were proved safe, and one node was confirmed safe by k-induction[122, 83]. Similar results were obtained on the second benchmark. Again, nodes that were hard to validate were hierarchical state machines expressed in Lustre. Lustre automata are compiled into pure dataflow equations, encoding transitions as clocked expressions, which explains that the final Lustre code is more complicated than the original model. All unproven nodes were validated by equivalence testing.

The approach was also applied to 4 industrial Simulink benchmarks: the NASA Transport Class Model (TCM) [27], the ROSACE use case [106], a Full Authority Digital Engine Control (FADEC), and a CNES Attitude and Orbital Control System (AOCS). All these benchmarks were analyzed using Eq. (3.1).

Table 3.1 shows the effectiveness of compositional verification compared to monolithic verification. Simulink Design Verifier was unable to prove any model globally. This can be explained by using nonlinear arithmetic operators, which solvers hardly analyze, and the model's size. We could prove two of the four models using model-checking on the Lustre global encoding. Compositional verification in Lustre shows better performance: more than 85% of nodes are proven safe. Equivalence testing was applied to the unproven nodes. For the AOCS case study, the fact that we were able to prove the system with Lustre globally but unable to verify all the corresponding nodes can be explained by eliminating some behaviors challenging to prove for particular nodes when considering the global system.

3.4 Conclusion

In this chapter, we have presented CoCoSim, a toolbox that aims to ease the requirements gathering and the V&V activities for Simulink models. We have shown how CoCoSim can be used with FRET to formalize requirements as Simulink components, and we have described the benefits of this approach. We have also presented the results of our preliminary evaluation of CoCoSim, which shows that it can correctly translate Simulink models to Lustre synchronous language.

Chapter 4

Use Cases

Contents

This chapter presents a case study on applying CoCoSim with different formal verification frameworks on two use cases: the ten Lockheed Martin Cyber-Physical challenges and a navigation rover. In the first use case, we report on our experience using FRET and CoCoSim to perform an end-to-end analysis of the challenges, from requirements elicitation and formalization to model analysis and verification. We categorize recurring patterns in the formalization of the requirements and discuss the strengths and weaknesses of our verification approach. In the second use case, we demonstrate the integration of different verification frameworks for the verification of the navigation system of a rover. We demonstrate that using an assurance case as the point of integration provides benefits in maintaining coherent formal links across development and assurance processes for many systems. Our work on both use cases has been published in several papers and technical reports [22, 100, 20, 99, 98].

4.1 The ten Lockheed Martin Cyber-Physical challenges

Capturing and analyzing Cyber-Physical Systems (CPS) requirements can be challenging since CPS models typically involve time-varying and real-valued variables, physical system dynamics, or even adaptive behavior. MATLAB®/Simulink ® [124] is a widely-used framework in the industry; in particular, more than 60% of engineers use Simulink for the development and simulation of CPS [138, 104]. In this use case, we report on the application of NASA Ames tools to perform an end-to-end analysis of the ten Lockheed Martin Challenge Problems (LMCPS). LMCPS is a set of industrial Simulink model benchmarks and natural language functional requirements developed by domain experts [54, 53]. End-to-end analysis means we start with requirements elicitation, formalization, and analysis and proceed with model analysis against formalized requirements. Such analyses may result in updates of requirements and/or models. Our framework, which integrates the tools FRET and CoCoSim, as explained in Section 3.1.3, is used to 1) elicit, explain, and formalize the semantics of the given natural language requirements; 2) generate verification code and monitors that can be automatically attached to the Simulink models; 3) perform verification by using SMT-based model checkers. FRET and CoCoSim are open source and can be used by other researchers and practitioners to replicate our case study. We provide a categorization of recurring patterns in the formalization of the requirements and discuss the strengths and weaknesses of our automated verification approach.

End-to-end analysis of CPS models is challenging. Requirements are typically written in natural language, ambiguous, and not readily amenable to formal analysis. Moreover, CPS models typically involve time-varying and real-valued variables, physical system dynamics, or even adaptive behavior. Formal languages supported by analysis tools may not be expressive enough to capture requirements for such systems. Finally, analysis tools may not be able to handle the complexity and scale of CPS models.

We were able to capture and analyze the majority of the LMCPS requirements with our framework. Our main findings can be summarized as follows:

Language and Logics. CPS requirements involve timing, so it is crucial to handle this aspect in requirements elicitation tools. The explanations produced by FRET were instrumental in ensuring that the FRETISH requirements captured our intended semantics. Even though FRETISH is aimed at being intuitive, it was not always straightforward to turn natural language

LMCPS requirements into FRETISH. However, most LMCPS requirements fall within a small number of patterns, an issue we have observed in other studies within our organization. As a result, our logic could not capture some aspects of the system, particularly those related to delay blocks, which are heavily used in the models. We were able to shortcut this problem by exposing internal model variables at the requirements level, but a FRETISH-level solution would be desirable.

Formalization and verification Code. FRET formalizations are compact for most of the requirements of the LMCPS challenge because they are optimized for many of the recurrent patterns. The automated production of verification code was a very smooth process. Automating the process of generating verification code from requirements has been extremely valuable since it reduces the sources of discrepancy and errors in the various artifacts.

Connecting Requirements to Models for Analysis. Requirements capture should not depend on the existence of a model. Different members of our team worked on requirements capture and SIMULINK model analysis. We, therefore, found the capability of importing SIMULINK models in FRET a great help during the step of connecting requirements with their targeted models. The most important feature of our integrated framework has been the capability to preserve the component structure of the LMCPS systems and use it to perform analysis in a modular fashion. It has been instrumental in achieving scalability. Our analysis exposed issues including requirement ambiguities, undefined parts in the models, and minor bugs in the checkers invoked by CoCoSim.

All details of our case study are available at [100], and our tools can be obtained, open source, at `https://github.com/NASA-SW-VnV/`.

4.1.1 Lockheed martin cyber-physical systems (LMCPS) challenge problems

The 10 Cyber-Physical V&V Challenges [53] were created by Lockheed Martin Aeronautics to evaluate and improve the state-of-the-art in formal method toolsets. Each challenge problem includes: 1) documentation that contains a high-level description and a set of requirements written in plain English; 2) a SIMULINK model; 3) a set of parameters (in .mat format) for simulating the model.

The challenges were first presented in the 2016 Safe and Secure Systems and Software Symposium (S5) [53]. They consist of a set of problems inspired by flight control and vehicle management systems, which are representative of flight-critical systems. They are publicly available[1] and as such, they provide an excellent basis for discussion and comparison of approaches across the research community.

Challenge	Description	NoB	Block Types	Template Keys
Triplex Signal Monitor (TSM)	A redundancy management system that prevents errors from propagating past the input of an airborne application.	479	Non-linear (*Switch*), Vectors and Matrices	[*null*, *null*, *always*]

[1] `https://github.com/hbourbouh/lm_challenges`

Finite State Machine (FSM)	An abstraction of an advanced autopilot system interacting with an independent sensor platform to ensure a safe automatic operation in the vicinity of hazardous obstacles.	279	Non-linear (*Switch*)	[*null*, *null*, *always*] [*null*, *regular*, *immediately*]
Tustin Integrator (TUI)	A flight software utility for computing the integration of a signal.	45	Non-linear (*Switch*, *Saturation*), Vectors and Matrices	[*null*, *null*, *always*]
Control Loop Regulators (REG)	A regulators inner loop architecture that is commonly used in many feedback control applications.	271	Non-linear (*Switch*, *Saturation*), Vectors and Matrices, Continuous-time	[*null*, *null*, *always*]
Nonlinear Guidance Algorithm (NLG)	A nonlinear guidance algorithm that generates commands in order to guide an Unmanned Aerial Vehicle (UAV) to follow a moving target respecting a specific safety distance.	355	Non-linear (*Sqrt*, *Switch*), Vectors and Matrices	[*null*, *null*, *always*] [*null*, *regular*, *always*]
Feedforward Cascade Connectivity Neural Network (NN)	A two-input single-output predictor neural network with two hidden layers arranged in a feedforward architecture.	699	Non-linear (*Saturation*), Vectors and Matrices	[*null*, *null*, *always*] [*null*, *regular*, *for*]
Abstraction of a Control Allocator - Effector Blender (EB)	A control allocation method, which enables the calculation of the optimal effector (surface) configuration for a vehicle, given a control minimization effort problem.	75	Non-linear (*Switch*), Vectors and Matrices	[*null*, *null*, *always*]
6DoF with DeHavilland Beaver Autopilot (AP)	A full, realistic full six-degree-of-freedom simulation of the DeHavilland Beaver airplane with autopilot.	1357	Non-linear (*Switch*, *Sqrt*, *Abs*, *MinMax*, *Saturation*), Non-algebraic (*Trigonometric*), Vectors and Matrices, Continuous-time	[*null*, *null*, *always*] [*in*, *null*, *always*] [*in*, *null*, *immediately*] [*in*, *regular*, *always*] [*null*, *regular*, *immediately*]

System-Wide Integrity Monitor (SWIM)	A safety algorithm for monitoring airspeed in the SWIM (System Wide Integrity Monitor) suite in order to provide a warning to an operator when the vehicle speed is approaching a boundary where an evasive fly-up maneuver cannot be achieved.	141	Non-linear (*Switch*, *Sqrt*), Vectors and Matrices	[*null*, *null*, *always*]
Euler Transformation (EUL)	A component that creates a Rotation Matrix describing a rotation about the z-axis, y-axis, and finally x-axis of an Inertial frame in Euclidean space.	97	Non-linear (*Switch*), Non-algebraic (*Trigonometric*), Vectors and Matrices	[*null*, *null*, *always*]

Table 4.1: Summary of LMCPS Challenges (NoB stands for Number of Blocks)

Although in most cases the specified requirements look relatively straightforward, a closer study revealed many questions regarding their precise meaning. Additionally, even though the Simulink models of the challenges were built with commonly used blocks, their analysis has proven to be challenging. Table 4.1 summarizes the ten LMCPS challenges. For each challenge, it includes a brief description, the number of Simulink blocks in the models, the types of blocks that challenged our analysis, and the 7 fret template keys that we used to formalize the requirements of each challenge.

The LMCPS requirements and models were developed to represent challenges that are typical of CPS systems. The inputs and outputs of CPS systems are modeled through signals, which are functions over time. Most of the LMCPS models are highly numeric and often exhibit non-linear behavior. Next, we present elements of LMCPS that proved challenging for the analysis of the requirements. Section 4.1.3 discusses how we handled these challenges.

Vectors and Matrices. LMCPS challenges manipulate signals with multiple dimensions. The use of multi-dimensional signals and matrices is common in CPS Simulink models since control systems are often defined as the composition of linear systems.

Non-Linear and Non-Algebraic Blocks. Trigonometric functions, exponential functions, and the logarithm are typically not supported by SMT solvers. Two of the LMCPS challenges, i.e., AP and EUL, use the *Trigonometric Function* Simulink block to perform common trigonometric functions. The square root, i.e., *sqrt* Simulink block, used in the NLG, AP, and SWIM challenges is usually not well-handled by SMT solvers. Other non-linear Simulink blocks causing discontinuity that are challenging for analysis are *Abs*, *MinMax*, *Switch*, and *Saturation*.

Continuous time blocks. Such blocks can be almost arbitrarily mixed with sampled blocks in Simulink. Thus another challenge comes from the fact that CPS models often contain mixes of continuous and discrete parts.

Complex requirement formalizations. As shown in Table 4.1, we used 7 distinct semantic template keys to express the LMCPS requirements. The formalization that corresponds

Table 4.2: Semantic Template Formalizations. FTP (First Time Point) stands for $\neg\, Y$ TRUE

Template Key	**Past-time Temporal Logic**
[*null*, *null*, *always*]	$H\psi$
[*null*, *regular*, *immediately*]	$H(\phi \wedge ((Y\neg\phi) \vee \text{FTP})) \Rightarrow \psi)$
[*null*, *regular*, *always*]	$H(H(\neg\phi) \vee (\psi S(\psi \wedge (\phi \wedge ((Y\neg\phi) \vee \text{FTP})))))$
[*null*, *null*, *for*]	$H((O[\leq \mathit{duration}]\text{FTP}) \Rightarrow \psi$
[*in*, *null*, *always*]	$H(\text{mode} \Rightarrow \psi)$
[*in*, *null*, *immediately*]	$H((\text{mode} \wedge (\text{FTP} \vee (Y(\neg\text{mode})))) \Rightarrow \psi)$
[*in*, *regular*, *always*]	$((H((((\neg \mathit{mode}) \wedge (Y\mathit{mode})) \wedge (Y\text{TRUE})) \Rightarrow$ $(Y((((\neg\phi)S((\neg\phi) \wedge (\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))) \vee (\psi S$ $(\psi \wedge (\phi \wedge ((Y(\neg\phi)) \vee (\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))))))S$ $((((\neg\phi)S((\neg\phi) \wedge (\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))) \vee$ $(\psi S(\psi \wedge (\phi \wedge ((Y(\neg\phi)) \vee (\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))))) \wedge$ $(\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))))) \wedge$ $(((\neg((\neg \mathit{mode}) \wedge (Y\mathit{mode})))S$ $((\neg((\neg \mathit{mode}) \wedge (Y\mathit{mode}))) \wedge (\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))) \Rightarrow$ $((((\neg\phi)S((\neg\phi) \wedge (\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))) \vee$ $(\psi S(\psi \wedge (\phi \wedge ((Y(\neg\phi)) \vee (\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))))))S$ $((((\neg\phi)S((\neg\phi) \wedge (\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))) \vee$ $(\psi S(\psi \wedge (\phi \wedge ((Y(\neg\phi)) \vee (\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))))) \wedge$ $(\mathit{mode} \wedge (\text{FTP} \vee (Y(\neg \mathit{mode})))))))$

to each template key is shown in Table 4.2. For certain template keys, e.g., [*in*, *regular*, *always*] the formalization is complex and potentially challenging for analysis tools.

4.1.2 The FRET-CoCoSim integrated framework

Figure 4.1 illustrates the flow of our framework. In the elicitation loop – **Step 0** – the user writes and refines requirements in FRETISH based on the semantic explanations and simulation capabilities supported by FRET. Once the user is satisfied with the requirement semantics, the FRETISH requirements are translated in **Step 1** into pure Past-Time / Future-Time Metric LTL (pmLTL/fmLTL) formulas. In **Step 2**, data from the model under analysis is used to produce an architectural mapping between requirement propositions and SIMULINK signals. In **Step 3**, the pmLTL formulas and the architectural mapping are used to generate CoCoSpec monitors and traceability data. In **Step 4**, CoCoSim [15] imports the generated CoCoSpec monitors and traceability data, along with the SIMULINK model. CoCoSim then produces SIMULINK monitors, attaches them to the model, and produces LUSTRE code for the complete model (initial model plus attached monitors). CoCoSim can thus drive both SIMULINK-based (e.g., SIMULINK Design Verifier (SLDV)) and LUSTRE-based (e.g., Kind2 [32], Zustre) verification tools to analyze the target model in **Step 5**. Counterexamples produced by the analysis can be traced back to CoCoSim or FRET (**Step 6**).

We illustrate the entire process through the following requirement from the 6 Degree Of Freedom Dehavilland Beaver Autopilot (AP) LMCPS challenge.

[AP-003c] Natural Language: *The roll hold reference shall be set to 30 degrees in the*

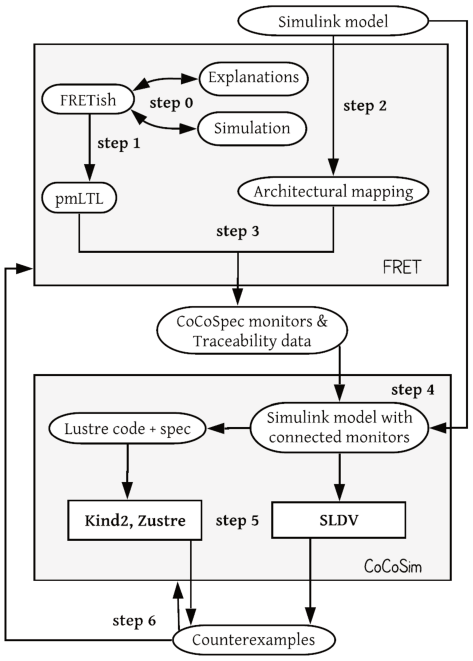


Figure 4.1: Requirement analysis framework

same direction as the actual roll angle if the actual roll angle is greater than 30 degrees at the time of roll hold mode engagement.

Step 0: Elicitation. Understanding the above natural language requirement and making it precise is not straightforward. We first identify the variables involved. By reading the first part of the requirement: *'The roll hold reference shall be set to 30 degrees in the same direction as the actual roll angle'* we identify two variables: `roll_hold_reference` and `roll_angle`, and express this part of the requirement as `roll_hold_reference = 30 * sign(roll_angle)`, where function `sign` returns the sign, e.g., -1 or 1, of the `roll_angle` in order to determine its direction. For the second part of the requirement, i.e., *if the actual roll angle is greater than 30 degrees at the time of roll hold mode engagement*, we identify variables: `roll_angle` and `roll_hold_mode_engagement`. Our first attempt at FRETISH for [**AP-003c**] was the following: [**AP-003c-v1**]:
If abs(roll_angle) > 30 & roll_hold_mode_engagement Autopilot shall immediately satisfy roll_hold_reference = 30 * sign(roll_angle), where function `abs` returns the absolute value of `roll_angle`.

FRET displays requirement semantics in a variety of forms: English descriptions, diagrammatic representations, and logic formulas (metric temporal logics with pure future-time / pure past-time operators). Figure 4.2 shows the English and diagrammatic descriptions generated by FRET for [**AP-003c-v1**]. *TC* denotes a triggering condition.

Since the scope is not specified, the requirement is 'enforced' in the interval defined by the entire execution, i.e., the beginning of time to the end of the execution. Notice that the condition of the requirement, i.e., If abs(roll_angle) > 30 & roll_hold_mode_engagement, is a 'trigger': the requirement is only enforced at time points where the condition becomes true from

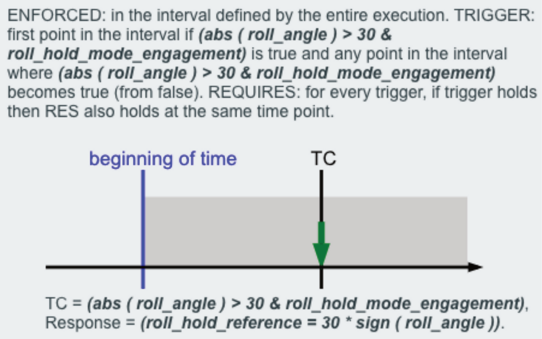


Figure 4.2: FRET semantics for requirement [**AP-003c-v1**]

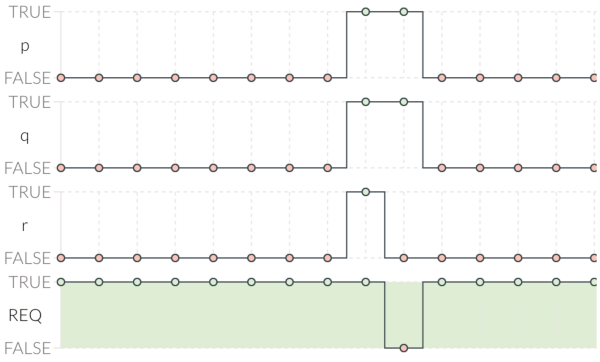


Figure 4.3: Simulating requirement [**AP-003c-v1**]

false, or at the first point of the interval if the condition holds there. Timing "immediately" states that the response should hold simultaneously with each trigger point.

We additionally use the FRET simulator, which allows users to interactively set values of requirement variables over a time interval and observe the consequences on the value of the requirement's formulas. Let us abbreviate abs(roll_angle) > 30 as 'p', roll_hold_mode_engagement as 'q', and roll_hold_reference = 30 * sign(roll_angle) as 'r'. Figure 4.3 shows the FRET simulator. We can see that even when condition ('p' and 'q') holds for two consecutive points, the response is only required to hold at the first point (where the condition becomes true), for the requirement (REQ) to hold. The green color of the last row (REQ) indicates that the requirement holds with the selected valuation of variables (it would be red otherwise).

Having thus used the aids that FRET provides for understanding FRETISH requirements, we realized that [**AP-003c-v1**] did not capture our intentions. Instead of a trigger, we wanted the response to hold every time the condition is true and not only when it becomes true from false. Thus, we rewrote the requirement as follows:
[**AP-003c-v2**]: Autopilot shall always satisfy (abs(roll_angle) > 30 & roll_hold_mode_engagement) ⇒ roll_hold_reference = 30 * sign(roll_angle).

The response is now expressed as an implication. The requirement is no longer true in the scenario of Figure 4.3, as indicated by the red color of the last row (REQ) in Figure 4.4. Version 2 already seems closer to the intended semantics of the initial requirement, however, there is a temporal sub-property that we have not expressed yet — it is hidden in the `roll_hold_mode_engagement` variable. To elicit the full meaning of the requirement, it is ideal

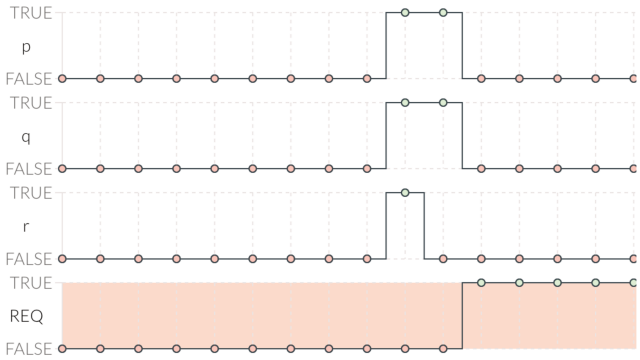


Figure 4.4: Simulating requirement [**AP-003c-v2**]

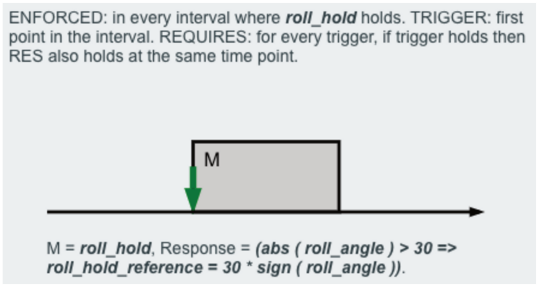


Figure 4.5: FRET semantics for requirement [**AP-003c-v3**]

to express explicitly all sub-properties through the FRETISH grammar. The `roll_hold_mode_engagement` variable captures the fact that there is a mode of operation, i.e., the `roll_hold` mode, and the scope of our requirement only applies to the intervals in which `roll_hold` mode is true. The `roll_hold_reference` must be set to 30 degrees in the direction of the roll angle at the time of roll hold mode engagement, in other words, immediately upon entering the `roll_hold` mode. We unfold this sub-property within the FRETISH requirement as follows:
[**AP-003c-v3**]: when in roll_hold mode Autopilot shall immediately satisfy (abs(roll_angle) > 30) ⇒ roll_hold_reference = 30 * sign(roll_angle).

The FRET semantics for requirement [**AP-003c-v3**] is shown in Figure 4.5. The requirement is 'enforced' in every interval where `roll_hold` holds. The 'trigger' defines the first point in the mode interval: when `roll_hold` becomes true from false. The response is 'required' to hold at that same point.

Step 1: Formalization. The pmLTL formula that FRET generates for [**AP-003c-v3**] is an instantiation of the [*in*, *null*, *immediately*] template key in Table 4.2:
$(H(roll\_hold\ \&\ (FTP \mid (Y(!\ roll\_hold))) \Rightarrow (abs(roll\_angle) > 30 \Rightarrow roll\_hold\_reference = 30 * sign(roll\_angle))))$,
where FTP is a predicate that holds at the First Time Point of an execution (equivalent to $\neg\ Y$ TRUE).

Step 2: Architectural mapping. To generate monitors and automatically attach them at the right hierarchical level of the model, we need architectural data from the model. For instance, for [**AP-003c-v3**], we need the path, in the model hierarchy, of the `Autopilot` component

Table 4.3: FRET to Model Variables mapping for Autopilot (abbr. ap_12BAdapted/GlobalScope by global)

FRET name	Model path
roll_angle	global/Autopilot/Roll_Autopilot/Phi
roll_hold_reference	global/Autopilot/Roll_Autopilot/PhiRef_cmd
roll_hold	global/Autopilot/Roll_Autopilot/RollHold

mentioned in FRETISH. Additionally, we need information about the signals of the component, e.g., name, type (e.g., `input, output`), datatype (e.g., `boolean, double, bus`) that correspond to the variables mentioned in [**AP-003c-v3**]. Our framework provides a mechanism to automatically extract the required data from a SIMULINK model. The mapping of the variables used in [**AP-003c-v3**] is shown in Table 4.3.

Steps 3 & 4: Generation of analysis code and Simulink monitors. To translate [**AP-003c-v3**] into LUSTRE code, the pmLTL formula generated by FRET gets translated into the following COCOSPEC code, used by COCOSIM for analysis:

```
-- AP-003c-v3 requirement in CoCoSpec
guarantee H((roll_hold and (FTP or (pre (not roll_hold))))
            => abs(roll_angle) > 30 =>
            roll_hold_reference = 30 * sign(roll_angle))
```

The COCOSPEC code then gets compiled into a SIMULINK monitor block, which is attached to the original model.

Steps 5 & 6: Analysis and counterexample generation. Requirement [**AP-003c-v3**] was shown to be invalid by the Kind2 model checker. Kind2 returned the counterexample shown in Table 4.4, which shows that at the time of roll hold mode engagement, (when T = 0.025, `roll_hold` becomes true and the absolute value of `roll_angle` is greater than 30), `roll_hold_reference` is not set to 30 degrees in the same direction as the `roll_angle`, i.e., `roll_hold_reference` is not equal to -30. Instead we noticed that at the time of roll hold engagement, `roll_hold_reference` is equal to 0.0, which is the value of `roll_angle` at the previous step. Based on this counterexample we modified the requirement as follows:

[**AP-003c-v4**]: when in roll_hold mode Autopilot shall immediately satisfy (abs(roll_angle) > 30) ⇒ roll_hold_reference = previous(roll_angle), where previous is a function that returns the value of roll_angle at the previous time step. Requirement [**AP-003c-v4**] was proven valid.

To understand why the output is based on the previous value of `roll_angle`, we looked at the SIMULINK model. Figure 4.6 shows the SIMULINK model responsible for returning the `roll_angle`, where `u` is the roll angle and `E` is the condition 'not engaged in roll hold mode'. If `E` is true, output `y` is equal to the previous value of `roll_angle`. Once the roll hold mode becomes active (`E` is false), the value of output `y` is equal to `pre y`; `y` holds this value while roll hold mode is active. Thus, the component holds the value of the roll angle just before the activation of roll hold mode and not "at the time of activation". Thus, we believe that [**AP-003c-v3**] is not satisfied due to an incomplete/erroneous model.

Note that we could not express the temporal sub-property `previous roll_angle` of [**AP-**

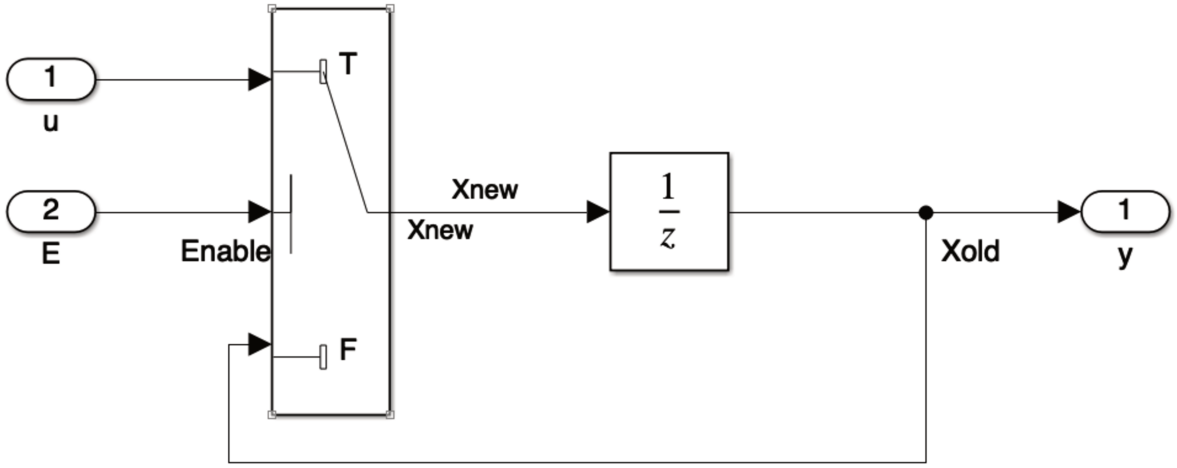


Figure 4.6: SIMULINK model for requirement [**AP-003c-v4**]

Table 4.4: Counterexample of requirement [**AP-003c-v3**]

Inputs	**T = 0**	**T = 0.025**
roll_angle	0.0	-90.0
roll_hold	false	true
Outputs		
roll_hold_reference	3.0	0.0

003c-v4] directly through the FRETISH language and thus, we expressed it through an internal variable, which we defined directly in COCOSPEC.

4.1.2.1 Reliable tool integration

A key feature in the integration of FRET and COCOSIM is the capability to generate COCOSPEC code from pmLTL logic formulas. Our translation keeps the structure of the original formulas and is based on defining pmLTL operators in COCOSPEC. This process is described in [99] for untimed operators. To support timed modifiers that constrain an operator's scope to a specific interval $[l, r]$, we have extended the process described in [99] and created a library of timed operators in LUSTRE. For instance for the timed version of `O`, we added the following nodes to the library:

```
--Timed Once: general case
node OT(const L: int; const R: int; X: bool;)
     returns (Y: bool);
   var D:bool;
let
   D = delay(X,R);
   Y = OTlore(L-R,D);
tel

--Timed Once: less than or equal to N
node OTlore(const N: int; X: bool; ) returns (Y: bool);
```

```
    var C:int;
let
    C = if X then 0
        else (-1 -> pre C + (if pre C <0 then 0 else 1));

    Y = 0 <= C and C <= N;
tel
```

The delay function delays input `X` by `R` time units to define the right bound of the interval in which the valuation of `X` must be checked. Once the input `X` has been delayed by `R` time steps, we can treat the `R` bound as zero and use the `OTlore` (Once Timed less than or equal to) node to check the valuation of `X` in the interval defined by the 0 (current) time step and the left bound `L-R`. `OTlore` is implemented using an integer counter C, which counts the number of time steps that occurred since the last occurrence of property X. If the event has never occurred, the counter keeps its initial value of -1. Other time-constrained operators are defined through `OT` using the usual temporal logic equivalences.

To provide assurance that the CoCoSpec code generated is correct, we extend FRET's formula verification framework to also handle CoCoSpec code. The framework presented in [70] automatically generates test cases each consisting of an execution trace t, a template key k, and an expected truth value e, reflecting the semantics of k applied to t. Formulas corresponding to k are then evaluated on t using a model checker, to ensure that the result agrees with e. In our extension for CoCoSpec, this step uses the Kind2 model checker. Our verification framework helped us detect and correct discrepancies between the CoCoSpec generated formulas and the intended FRET template key semantics.

4.1.3 Analysis - Selected use cases and requirements

Our case study encompasses the following tasks: 1) eliciting requirements in FRETISH; 2) making the mapping between FRETISH variables and model variables; 3) performing analysis; 4) interpreting counterexamples at the requirements level; 5) interpreting requirements at the model level. Three researchers were involved: 1) a control engineer considered the domain expert, 2) a requirements expert, and 3) a verification expert. Tasks 1 and 2 were performed together by the requirements and domain experts. The verification expert performed tasks 3 and 5. Finally, task 4 was performed by the requirements expert.

The verification results are summarized in Table 4.7. The analysis was carried out on a MacBook Pro with 3.1 GHz Intel Core i7 and 16 GB Memory, an R2019b MATLAB/Simulink, and a v1.1.0 Kind2. Kind2 was configured to time out after 2 hours. This Section highlights the analysis results of a subset of the LMCPS challenges and discusses how we approached the challenging elements presented in Section 4.1.1.

4.1.3.1 Requirements and Verification

We focus on the following LMCPS components: FSM, TUI, NN, and AP. We include the FSM and TUI challenges because they exhibit cases of unrealizable requirements and a requirement that we could not express directly in FRETISH. The NN challenge describes a machine learning

Table 4.5: Counterexample of requirement [FSM-003]

Inputs	**T = 0**
standby	true
supported	true
good	true
state	ap_transition_state
Outputs	
STATE	ap_standby_state

model. Verification of models that are inferred by machine learning techniques is currently considered an open area for research. Finally, AP is the most complex of the LMCPS challenges in terms of the number and types of blocks used, and its FRETISH requirements involve a variety of template keys.

FSM represents an abstraction of an advanced autopilot system interacting with an independent sensor platform to ensure a safe automatic operation in the vicinity of hazardous obstacles. The autopilot system, tightly integrated with the vehicle flight control computer, is responsible for commanding a safety maneuver in the event of a hazard. The sensor is the reporting Agent to the autopilot with observability of imminent danger.

All FSM requirement examples were written in FRETISH using the [*null*, *null*, *always*] semantic key pattern. Let us look into the following FSM requirements:

[FSM-002] Natural Language: *The autopilot shall change states from TRANSITION to STANDBY when the pilot is in control (standby).*
[FSM-002] fretish: FSM shall always satisfy (standby & state = ap_transition_state) => STATE = ap_standby_state.

[FSM-003] Natural Language: *The autopilot shall change states from TRANSITION to NOMINAL when the system is supported, and sensor data is good.*
[FSM-003] fretish: FSM shall always satisfy (state = ap_transition_state & good & supported) => STATE = ap_nominal_state.

The valuations ap_transition_state, ap_standby_state, ap_nominal_state of the state and STATE variables represent the *TRANSITION, STANDBY,* and *NOMINAL* states of the autopilot. Requirement **[FSM-002]** was shown to be valid. However, when checking requirement **[FSM-003]**, analysis returned the counterexample shown in Table 4.5. It is interesting to note that the valuation of the input variables of the counterexample satisfies the preconditions of both **[FSM-002]** and **[FSM-003]**. While these requirements are not mutually exclusive, their expected responses are conflicting, which makes them unrealizable [60]. If we form a weaker property, i.e., strengthen the precondition as follows (state = ap_transition_state & good & supported & !standby) (notice the addition of !standby), then **[FSM-002]** and **[FSM-003]** become mutually exclusive and requirement **[FSM-003]** is proven valid.

Note that non-mutually-exclusive requirements are not necessarily problematic since requirements are often complementing each other to make up a system's specification. For example, we found several pairs of requirements that were not mutually exclusive in the LMCPS challenge.

TUI represents a flight software utility for computing the integration of a signal. The algorithm executed by the utility bounds the allowable integration range with a position limiter. The integrator is in normal operation when it is not in reset mode, and the output is within the specified limits.

[**TUI-004**] **Natural Language:** *After 10 seconds of computation at an execution frequency of 10 Hz, the output should equal 10 within a +/-0.1 tolerance for a constant input (xin = 1.0), and the sample delta time T = 0.1 seconds when in the normal mode of operation.*

This requirement could not be expressed directly in FRETISH. The "*After 10 seconds of computation at an execution frequency of 10 Hz*" part of the requirement constitutes a condition that must persist for a time duration (10 seconds). Such conditions are not yet supported in FRETISH.

NN is a two-input, single-output, two-hidden-layer feed-forward nonlinear neural network. Neural networks of this form are common utilities in modeling and simulation for capturing complex numerical dependencies. In this challenge, a single variable, z, is computed based on two independent parameters, x and y. This challenge comes with a truth table in the form of a Matlab matrix file with reference values xt, yt, and zt. The NN specification file consisted of four requirements. We show below the FRETISH version of requirement [**NN-004**], for which we used the for 200 sec metric timing, which resulted in CoCoSpec code that uses the *once timed* (OT) metric LTL operator.

[**NN-004**]: NN shall for 200 sec satisfy (x = xt & y = yt => AbsoluteErrorZtMinusZ $\leq$ 0.01).

Below is the generated CoCoSpec code for [**NN-004**]:

```
var AbsoluteErrorZtMinusZ: real = if (zt-z) > 0.0 then zt - z else z - zt;

guarantee "NN004" OT(200,0,FTP) => ( x = xt and y = yt => AbsoluteErrorZtMinusZ <= 0.01 );
```

Kind2 and SLDV did not return an answer for any of the four NN requirements.

AP AP represents a complete system, and, as shown in Table 4.1, it contains SIMULINK blocks that involve non-linearities, non-algebraic math, and manipulation of matrices. AP is a full six-degree-of-freedom simulation of a single-engined high-wing propeller-driven airplane with autopilot. A six-degree-of-freedom simulation enables movement and rotation in three-dimensional space. The AP model and requirements also capture the plant mode of the airplane, i.e., the physical model and environmental aspects such as wind that influence the airplane's motion. AP requirements define the required behavior of the model in terms of changes in the three perpendicular position axes (forward/backward, left/right, up/down) combined with changes through rotation (yaw, pitch, and roll).

In the AP and FSM challenges, we performed modular analysis. The specification generation mechanism of FRET provides specifications at any desired level and complete traceability information regarding where the corresponding monitor should be deployed in the model. Additionally, the CoCoSim compiler preserves the model's hierarchy, allowing analysis at different

Table 4.6: AP Analysis Results with Kind2

Reqs	Scope	Kind2 Result	Kind2 Time
[AP-000]	Global	Unsupported	
[AP-001]	Roll AP	Valid	< 1 sec
[AP-002]	Roll AP	Valid	< 1 sec
[AP-003a]	Roll AP	Invalid	< 1 sec
[AP-003b]	Roll AP	Invalid	< 1 sec
[AP-003c]	Roll AP	Invalid	< 1 sec
[AP-003d]	Roll AP	Valid	< 1 sec
[AP-004]	Global	Unsupported	
[AP-005]	Global	Unsupported	
[AP-006]	Global	Unsupported	
[AP-007]	Roll AP	Valid	< 1 sec
[AP-008]	Roll AP	Valid	< 1 sec
[AP-010]	Global	Unsupported	
Total running time		CoCoSim: 40.589s	

levels.

The AP model is a closed-loop system that contains an *algebraic loop* involving all top-level components. An algebraic loop occurs when there is a circular dependency of signals/ variables (block outputs and inputs) in the same time step. Lustre forbids such constructs; no circular dependencies are allowed. Strangely though, Kind2 was not able to detect the algebraic loop. We contacted a Kind2 developer and confirmed that there is a bug in the algebraic loop detection algorithm. Once the bug was fixed, top-level analysis was not possible with Kind2. Simulink, on the other hand, treats algebraic loops as algebraic constraints, which it solves numerically using the ODE (Ordinary Differential Equation) solver. However, this is not considered a good programming practice since the simulator engine defines the behavior.

As shown in Table 4.6[2], we were able to get results only for the requirements that were analyzed against a sub-component (local scope, e.g., Roll AP). Requirements that were analyzed against the top AP component (global scope), e.g., **[AP-000]**, were either unsupported or undecided. In particular, analysis with Kind2 was not supported due to the algebraic loop, while analysis with SLDV was undecided.

4.1.3.2 Challenges

Vectors and Matrices SMT solvers do not typically support multi-dimensional signals as native objects. Signals and matrices need to be split into individual scalar variables making analysis harder depending on the operations that need to be performed. For instance, the EB

[2]We also used SLDV but the MathWorks license prevents publication of empirical results compared with SLDV, so we omit the SLDV results from Table 4.6.

and AP challenges use blocks that compute the inverse of matrices. At the same time, AP also manipulates quaternions with some advanced quaternion operations (e.g., *Quaternion Modulus*, *Quaternion Norm* and *Quaternion Normalize*).

We experienced the same problem at the level of requirements. To generate specifications that can be used for analysis by SMT solvers, we had to encode and expand matrix operations as non-matrix formulas. For instance, requirement **[EUL-001]** describes the computation of the DCM321 matrix as the product of the 3x3 Euler_Roll_Rotation and the 3x3 Euler_Pitch_Rotation matrices. To perform analysis on this requirement, we specified it in FRETISH by first decomposing it into nine sub-requirements, i.e., one for each element of the DCM321 matrix. Such decomposition naturally produces a more extensive specification than the original natural language requirement.

Non-Linear and Non-Algebraic Blocks Trigonometric functions, exponential functions, and the logarithm are typically not supported by SMT solvers. To perform meaningful analysis on LMCPS models containing trigonometric and square root blocks, we abstracted their behavior by providing a surrogate version, which is a sound abstraction. For instance, instead of block *sqrt*, which defines the signal $x = sqrt(y)$, we encoded properties $x * x = y \wedge x \geq 0$. Similarly, we provided bounds for the values depending on the input range for trigonometric functions.

Continuous time blocks Our analyses are based on the synchronous dataflow model and can only address discrete-time components. Thanks to the modular feature of our analyses, requirements associated with discrete-time components can be adequately addressed. However, in the case of continuous-time components (defined using continuous blocks such as Integrators, Transfer functions, or State space blocks), we first replace them with their discrete counterparts using SIMULINK discretization functions.

Complex requirement formalizations As shown in Table 4.1, some template keys like [*in*, *regular*, *always*] correspond to really complex formulas. This template key was used for the specification of requirements **[AP-004a]** and **[AP-010a]** of the AP challenge. These requirements were defined on the top-level component of the model, and thus, they could not be analyzed by Kind2 (due to the algebraic loop) nor by SLDV, which returned undecided. Note that, however, even simpler formalizations, such as the ones that correspond to the [*null*, *null*, *always*] key template, could not be analyzed globally. We also tried to verify specifications that correspond to [*in*, *regular*, *always*] at a local level, and interestingly, we were able to analyze them, which shows that modular verification can be effective even for complex specifications.

4.1.4 Lessons Learned

The application of our framework to an externally-provided and challenging system has been very informative. We summarize our experience and lessons learned below.

a) Can LMCPS requirements be captured in fret? We captured 69 out of 74 LMCPS

Table 4.7: LMCPS verification results. N_R: #requirements, N_F: #formalized requirements, N_A: #requirements analyzed by Kind2. Analysis results categorized by **V**alid/**IN**valid/**UN**decided. Timeout (TO) was set to 2 hours.

Name	N_R	N_F	N_A	**Kind2 V/IN/UN**	**Kind2 t(s)**
TSM	6	6	6	5/1/0	37.7
FSM	13	13	13	7/6/0	141.1
TUI	4	3	3	2/1/0	19.2
REG	10	10	10	1/5/4	TO
NLG	7	7	7	0/0/7	TO
NN	4	4	4	0/0/4	TO
EB	5	3	3	0/0/3	TO
AP	14	13	8	5/3/0	40.6
SWIM	3	3	3	2/1/0	25
EUL	8	7	7	1/6/0	43
Total	74	69	64	23/23/18	

requirements in FRET. As mentioned, we could not formalize requirement [**TUI-004**] that contains a temporal condition. Additionally, several requirements refer to the previous value of a variable (e.g., see [**AP-003c-v4**]), defined in pmLTL with the `Y` operator, in LUSTRE with the `pre` operator, and in SIMULINK as a delay block. Currently, FRETISH cannot express the "previous value of a variable." To shortcut this limitation, we used internal/auxiliary variables defined at the CoCoSpec level, but a FRETISH solution would be desirable. Additionally, we did not capture the following requirements: 1) [**AP-009**] since it was out of scope of the SIMULINK model; 2) [**EB-003**] since it is trivial; 3) [**EUL-005**] and [**EB-005**] were unclear: we were not able to interpret their meaning even with the help of the domain expert.

b) Is fretish intuitive? FRET provides 112 semantic template keys, of which we used only 7. Among these template keys, the [*null*, *null*, *always*] key was used the most (75% of the formalizations). We have had a similar experience with a NASA mission that we collaborate with: their requirements fall into recurring patterns. We are currently extending FRET with the capability to define typical requirement patterns within a domain or project and allow users to import and customize patterns within the editor. This extension will make requirements capture a more natural and intuitive process.

c) Are FRET explanations useful? We extensively relied on the semantic descriptions and diagrammatic representations, as well as the FRET simulation capabilities, to understand the meaning of requirements throughout the LMCPS study. It helped us identify and understand several semantic nuances of the FRETISH fields. Using modes and condition fields as first-level constructs of the FRETISH language was particularly useful. As shown in the elicitation phase of [**AP-003c**], unfolding the roll_hold_engagement sub-property through the FRETISH mode field allowed us to elicit the full meaning of the requirement (see [**AP-003c-v2**] and [**AP-003c-v3**] requirements in Section 4.1.2).

d) How effective is the FRET-CoCoSim integration? We could generate specifications and traceability data for all LMCPS challenges. They were used to automatically generate and attach monitors on the SIMULINK models (through CoCoSim). We found the ability to interpret and trace counterexamples at the model and requirement levels, particularly useful.

Counterexamples sometimes uncovered conflicting requirements, and we did not need the model to understand the problem (see Section 4.1.3). The FRET simulator was particularly useful in understanding these counterexamples. In other cases, counterexamples needed to be interpreted at the model level since they exposed behaviors in the model that violated the requirements.

e) How did we deal with model and specification complexity? To deal with complexity, we performed modular analysis whenever possible, i.e., for non-system requirements (requirements that could be applied locally). Our architectural mapping approach allows us to deploy CoCoSpec specifications at different levels of the model behavior. This mapping is essential for complex models where verification does not scale for global scopes. We applied modular verification to 20 out of the 69 requirements. For instance, in the FSM challenge problem, we generated three contracts deployed at three different hierarchical levels of the model. Similarly, in the AP challenge, we generated two separate contracts; one that we deployed at the top level component of the model and one that we deployed at a sub-component level. As a result, we were able to analyze all properties that were specified locally but none of the properties that were specified globally.

f) Which types of property reasoning/checking did we find helpful? Having a tight integration between requirement and verification activities allows us to use different approaches to interpret violated properties. In particular, we found the combination of reasoning at the level of requirements and counterexample simulation at the model level beneficial. When a property was invalid, we tried to understand the reason; i.e., is it because of a faulty requirement or a faulty model? Since in most cases, our formalized requirements were invariants of the form `H` $(A \Rightarrow B)$, we used two approaches: 1) check a weaker property, e.g., by strengthening the preconditions, i.e., $A' \subset A$ and check whether the invariant `H` $(A' \Rightarrow B)$ is satisfied, and 2) check feasibility of B with bounded model checking, i.e., `H` $(\neg B)$, In this case, the model checker returns counterexamples that could help construct stronger preconditions for B to be satisfied. Our case study showed that using these approaches was helpful for reasoning about violated properties. Furthermore, the simulation of counterexamples helped identify weaker properties and produce meaningful reasoning scenarios.

Additionally, we used CoCoSpec modes to perform vacuity checking [89]. A CoCoSpec mode has preconditions that describe the activation of the mode (`Requires`) and actual conditions to be checked (`Ensures`) of the form `H` (`R` $\Rightarrow$ `E`). Our case study showed that it is interesting to check whether the activation of a mode `R` is reachable. If not, the property is trivially true. So, in terms of analysis, showing that `R` is reachable allows us to understand better whether the property is meaningful for the current model. For instance, we discovered that one of the modes was never reachable in the AP challenge.

g) Were the abstraction techniques useful? We used abstractions of non-linear functions, e.g., trigonometric functions and the *sqrt* function, to perform analysis with the Kind2 model checker. This abstraction proved helpful for three challenges: REG, SWIM, and EUL. For instance, we proved two more requirements in the SWIM case study using a square root abstraction. In other cases, for instance, the abstractions would generate nonsensical counterexamples in the EUL challenge. For example, when we abstracted the *cosine* function with the interval [-1,1], we got the following nonsensical counterexample: $cos(0) = 0$.

4.1.5 Related Work

Section 3.1.2 and our published paper [99] describes the technical parts of the FRET-CoCoSim integration: 1) the FRET interface through which the architectural mapping can be performed, 2) the library of (non-metric) pmLTL operators that we defined, 3) the generation of Simulink monitors through CoCoSim.

The components of the LMCPS challenge have also been analyzed in [104]. However, that work focused on comparing the efficacy of verification tools for model testing versus model checking (the latter using QVtest from QRA Corp). There appear to be several differences in our formalization of requirements versus [104]. For example, in their companion material (reference 2 of [104]), they formalize **[AP-003c]**, which they call R1.3, as the invariant $G\{0, T\}(\text{Phi} > 30 \Rightarrow \text{PhiRef} = 30)$. We believe this misrepresents the part of the natural-language requirement that mentions that the roll angle Phi should be considered at the time of roll-hold engagement, as in our **[AP-003c_v3]**. The capability to explore the exact meaning of the requirements written in FRETISH through provided explanations gives us confidence in our requirements capture. Moreover, our framework formalizes requirements automatically, sparing users the error-prone effort of producing complex formulas for elaborate template keys.

Like FRET, the SpeAR [56], ASSERT™ [42], STIMULUS [82], RERD [127], and EARS-CTRL [93] tools provide natural language like formal languages to express requirements and properties. The ARSENAL tool [67] attempts to formalize general natural language requirements, as opposed to FRET and the others mentioned where a constrained natural language like formal language is used to express requirements. Except for STIMULUS, they do not appear to handle metric time, so they would not be able to express some of the neural network properties. EARS-CTRL aims to synthesize controllers, whereas formalizations in this case study are used with CoCoSim to verify the Simulink models against requirements. SpeAR and ASSERT can perform semantics checks on requirements, such as consistency and entailment, producing counterexamples when such checks are violated. Checking for requirements realizability is in our plans. ASSERT generates test cases, and STIMULUS simulates sets of requirements. However, none of the tools automatically synthesize monitors so that models can be model-checked against requirements.

The LMCPS benchmark provides a valuable case study to evaluate requirements elicitation and analysis tools. We found that using an end-to-end automatic framework significantly simplifies requirements elicitation and model analysis. Requirements formalization can quickly become complex, and writing complex formulas by hand or translating them into other logic can be challenging and error-prone. Eliciting requirements with unambiguous and as-intended semantics is not an easy task. Explanations and interactive exploration of written requirements are great tools for facilitating this task.

The ultimate purpose of formal requirements is to enable analysis. Requirements of CPSs can be complex to analyze, so it is vital to provide modular analysis techniques to achieve scalability. Space projects at NASA Ames are currently starting to use our framework, and we have already received valuable feedback. For example, desired are customizable requirement patterns and the ability to express that a condition persists until some event. The advantage of a close collaboration with mission scientists during the requirements development will allow us to evaluate further and improve the usability of our framework.

4.2 Navigation Rover

The complexity and flexibility of autonomous robotic systems necessitate various verification tools, which presents new challenges for design verification and assurance approaches. Combining the different formal verification tools while maintaining sufficient formal coherence to provide compelling assurance evidence is complex and often abandoned for less formal approaches. This case study demonstrates how various formal techniques can be combined to develop a justifiable assurance case. We use the AdvoCATE assurance case tool to guide our analyses and integrate the artifacts from different sources: FRET, CoCoSim, and Event-B. While we present our methodology as applied to a specific Inspection Rover case study, we believe this combination benefits in maintaining coherent formal links across development and assurance processes for a wide range of autonomous robotic systems.

The adoption of formal methods in the industry has been slower than their development and adoption in research. One of the main pitfalls is the difficulty in integrating the results from formal methods with non-formal parts of the system development process. A central stumbling block is the formalization of the (informal) natural language descriptions needed to perform the formal analysis and the analysis and interpretation of the formal verification results.

The integrated formal methods approach relies on various tools cooperating to ease the burden of formal methods at various phases of system development. It often involves facilitating the use of one tool/formalism from within another (e.g., Event-B‖CSP [120]), the development of tool/formalism that incorporates multiple others (e.g. Why3 [57]), or the construction of systematic translations between tools/formalizms (e.g. EventB2JML [111]). Recent work argues that, for autonomous robotic systems, the use of multiple formal and non-formal verification techniques is both beneficial and necessary to ensure that such systems behave correctly [55, 94]. Notably, the usually modular nature of robotic systems makes them more amenable to an integrated verification approach than monolithic systems [28]. The inherent modularity in robotic systems usually stems from using node-based middleware such as the Robot Operating System (ROS) [110]. However, other middlewares, such as NASA's core Flight System (cFS) [101], also support the development of similarly complex, modular systems.

In this use case, we study the support for integrating formal verification results at both system- and component-level in the design, implementation, and assurance of a critical system, namely, an autonomous rover undertaking an inspection mission. In contrast to usual approaches to integrating formal methods, such as those described above, we use an assurance case as the point of integration rather than building bespoke tools or defining mathematical translations between specific formal methods. This way, we harness the benefits of an integrated approach to verification without the usual overheads. Specifically, we use AdvoCATE [49] to perform safety engineering and assurance, FRET [68] to elicit and formalize requirements, and CoCoSim [15] with Kind2 to perform compositional verification of the system-level requirements. Further, we use Event-B [2] and Kind2 for the component-level formal verification. AdvoCATE facilitates the integration of the artifacts/evidence produced from these tools.

In summary, we contribute an inspection rover case study that demonstrates:

- how these tools can be linked via an argument in an assurance case.

- the benefit of using distinct tools due to their limitations (e.g., Kind2 would time out on specific properties verified in Event-B).
- how developing with formal methods in mind from the outset can influence the system's design, making it more amenable to formal verification.

Assurance Case Automation Toolset (AdvoCATE) [49] supports the development and management of assurance cases, composed of all of the assurance artifacts created during system development. AdvoCATE is built with a formal basis to enable automation where all assurance artifacts can be defined and formally related. Some artifacts can be created directly in AdvoCATE (e.g., hazard log, bow tie diagrams), while others, such as formal verification results, can be imported. AdvoCATE uses the Goal Structuring Notation (GSN) [72] to document assurance cases as arguments.

4.2.1 Assurance-based Formal Methods Integration

The objective of this work is *to study the integration of formal verification results via the development of an assurance case, as applied to a robotic system, using a tool palette that includes the three NASA Ames tools* FRET, CoCoSim, *and AdvoCATE, as well as Event-B.* To this end, we provide a step-by-step methodology that builds on existing NASA guidelines [51, 85] that can be used to design and develop mission-critical systems. In particular, existing guidelines [85] suggest the following phases: 1) characterization; 2) modeling; 3) specification; 4) analysis; and 5) documentation. Each phase consists of constituent processes, and the overall process is iterative rather than sequential.

Our methodology focuses on applying formal methods and connects them to parts of a greater system safety assurance methodology [48] needed to perform and assure the application of formal methods. Our methodology is guided by the need to devise a detailed assurance case that integrates verification results from several tools. The steps that we followed are the following:

Step 0: Characterize the initial system.
Step 1: Create an initial system model.
Step 2: Perform preliminary hazard analysis.
Step 3: Define mitigations and safety requirements.
Step 4: Refine system model according to mitigations.
Step 5: Formalize requirements and create formal specification(s).
Step 6: Perform verification and simulation at the system- and component levels.
Step 7: Document verification results and build safety case.

Fig. 4.7 presents a detailed view of our methodology instantiated with the selected tools for the Inspection Rover case study. The upper part of Fig. 4.7 shows the system-level concept, design, and assurance steps mainly performed by the AdvoCATE tool. In contrast, the lower part shows the formal methods application steps performed by the FRET, CoCoSim, and Event-B tools. In the analysis phase (step 6), we perform two types of analysis. First, we use CoCoSim to perform *compositional system-level* analysis with Kind2. We also perform verification at *the component level* against the system model using the Atelier-B and Kind-2 tools. Atelier-B

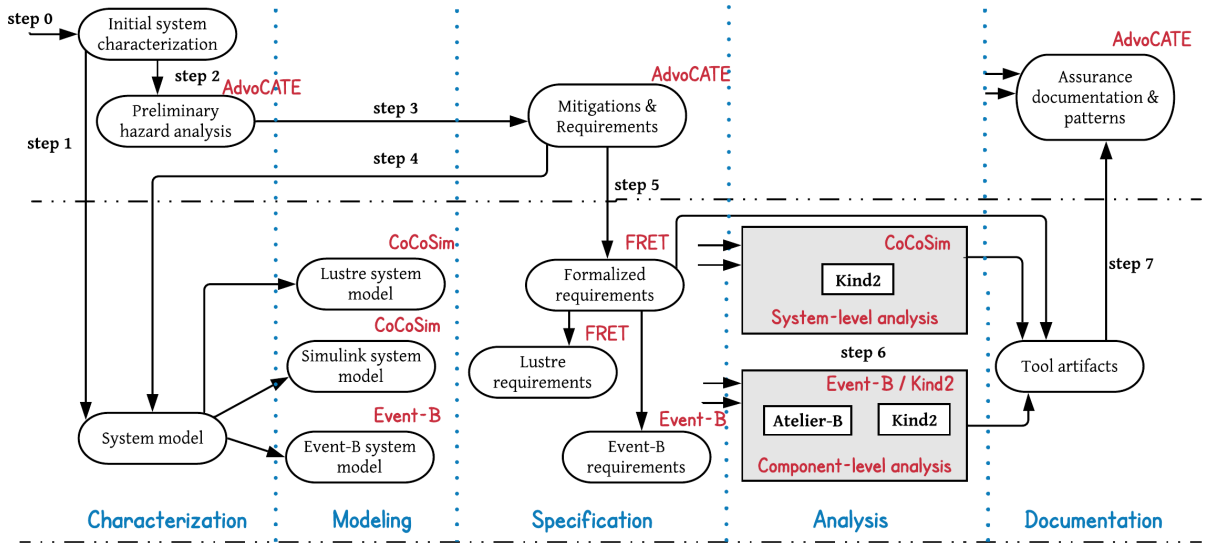


Figure 4.7: Our methodology for integrating verification results via an assurance case instantiated with the selected tools for the Inspection Rover case study. The incoming arrows without a source represent all relevant artifacts from previous phases. For system-level analysis, these comprise the LUSTRE requirements and the SIMULINK system model, while for component-level analysis, these comprise the LUSTRE and Event-B system models and requirements. In the documentation phase, we input all artifacts.

is supported by Rodin platform for verifying Event-B models. Finally, in the documentation phase, we use AdvoCATE to integrate the evidence produced by the tools within the assurance case.

Over the years, we have worked with various formal approaches to the assurance of safety-critical systems. This study explores how such approaches can work together and be integrated within the development process of an autonomous system. With this aim, we developed a case study of a rover system. Our case study is not extracted from an actual mission. Instead, it is developed by iteratively using our expertise in various assurance approaches. The resulting Inspection Rover case study has a reasonable complexity and demonstrates a variety of generic challenges in formal methods techniques and their integration. Most importantly, we make the details of our case study publicly available [21] since we believe it can serve as a good basis for discussing and comparing approaches and tools across the research community.

Four formal methods experts were involved: 1) a safety expert; 2) a requirements expert; 3) a SIMULINK and LUSTRE verification expert; and 4) a verification expert of robotic systems that also served as the domain expert. Step 0 was performed by the domain expert, and step 1 was performed together by the SIMULINK and domain experts. The safety expert performed steps 2 and 3. Steps 4 and 6 were performed by the safety, domain, and SIMULINK experts. Step 5 was performed by the requirements and domain experts, and finally, step 7 was performed mainly by the safety expert with contributions from all other experts.

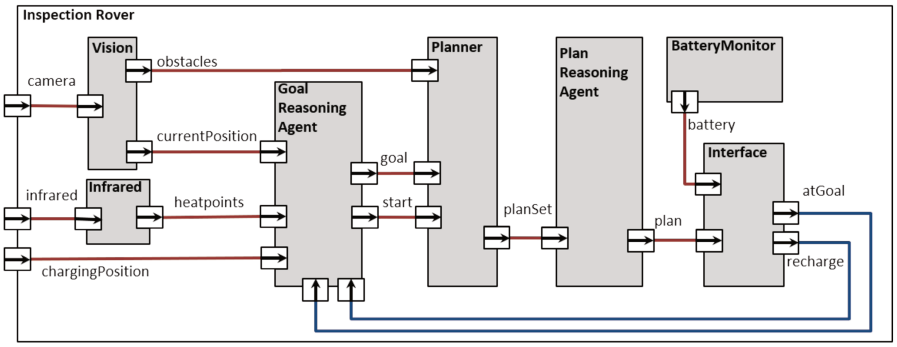


Figure 4.8: Preliminary inspection of rover system architecture.

4.2.2 The Case Study Step-by-Step

Step 0: Characterize Initial System

We performed our case study in the context of the navigation system for an autonomous rover undertaking an inspection mission. The objective of this rover is to explore a square grid of known size and to autonomously navigate to points of interest while avoiding obstacles and recharging as necessary. We assumed that this system would be operated indoors to minimize environmental uncertainty.

Step 1: Create an initial System Model

In step 1, we created the initial system model, which comprises a preliminary architecture of our rover (Fig. 4.8). The rover must navigate to all heat positions on a 2D grid map of known size. The *Vision* system detects obstacles to be avoided. The *Infrared* component identifies grid locations that are hotter than expected. From these heat locations, the autonomous *Goal Reasoning Agent* selects the hottest location as the goal unless the *Battery Monitor* (via the *Interface*) indicates that it must recharge. Next, the *Planner* computes obstacle-free plans for navigating from the current position to the goal. The autonomous *Plan Reasoning Agent* selects the shortest plan. Finally, the *Interface* translates the navigation actions of the plan into the instructions for the hardware components and alerts the *Goal Reasoning Agent* when it reaches the goal or does not have enough battery to execute the chosen plan, so it must recharge.

The initial system model was created in AADL and can be found in [21]. It was also created in Simulink and Event-B and automatically generated in Lustre via CoCoSim. In this work, we used Simulink in two different ways: 1) as an architecture description language, which allowed us to specify the architecture of the rover without providing implementations of low-level components (for compositionally verifying properties using assume-guarantee reasoning); 2) as a behavioral specification language for the implementation of some of the low-level components (for checking properties against component behavior).

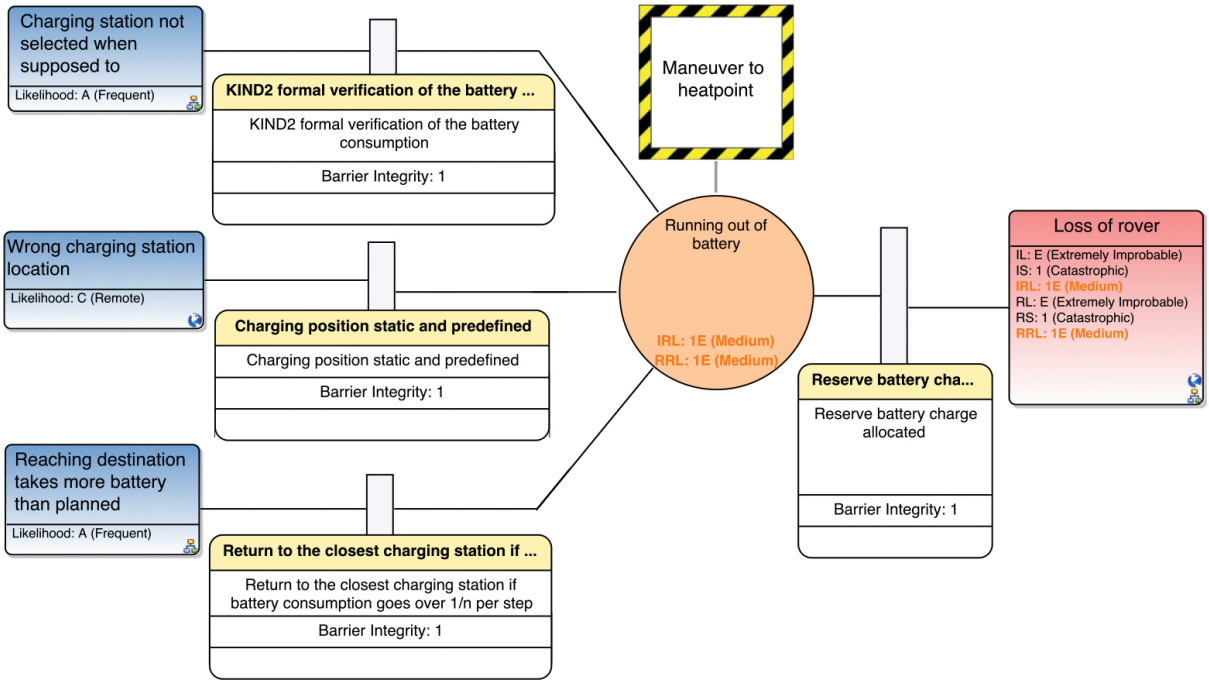


Figure 4.9: Bow Tie Diagram presenting the *running out of battery* hazard (orange circle), its causes (blue rectangles to the left), and consequence (red rectangle to the right).

Step 2: Perform Preliminary Hazard Analysis

To perform the preliminary hazard analysis in AdvoCATE as part of the safety assurance methodology [48], we first defined a functional decomposition of the Inspection Rover based on Fig. 4.8. Then, we performed the traditional hazard analysis (FMEA [128]) in the AdvoCATE hazard log. We identified two top-level hazards: 1) *loss of rover*, and 2) *inspection finished before visiting all of the heatpoints*. In total, we identified 25 hazards, including these two. E.g., we identified the *running out of battery* and *collision with an obstacle* hazard as causes of *loss of rover*. AdvoCATE uses the information from the hazard log to automatically create a safety architecture documented via interconnected Bow Tie Diagrams (BTD) for each hazard [47]. A single BTD shown in Fig. 4.9 details the causes and consequences of the *running out of battery* hazard.

Step 3: Define Mitigations and Safety Requirements

After preliminary hazard analysis, we conducted a risk analysis that qualitatively analyzed the severity and likelihood of the identified hazards to estimate the risk level. From this, we defined mitigations to minimize the risk of those hazards and their consequences. E.g., the *loss of rover* hazard is characterized by catastrophic severity, but its likelihood is calculated based on the events causing it. The combination of the two defines the risk associated with the hazard.

Next, we performed mitigation planning using BTDs. For example, to minimize the risk of *running out of battery* shown in Fig. 4.9: (1) we formally analyzed the navigation system and battery controller, (2) we ensured that the charging station position is predefined so that we can estimate at every point whether we have enough battery to go to recharge, and (3) if the basic

assumptions about battery consumption are violated, then we abort and return to the charging station. Furthermore, besides mitigating the causes to prevent the hazard, we add the recovery barrier between the hazard and the consequence to reduce the severity of the consequence in case the hazard still occurs.

For each of the two top-level hazards, *loss of rover* and *inspection finished before visiting all of the heatpoints*, we define system-level requirements:

[**R1:**] The rover shall not run out of battery.
[**R2:**] The rover shall not collide with an obstacle.
[**R3:**] The rover shall visit all reachable heat points.

The requirements [**R1**] and [**R2**] correspond to the causes of *loss of rover*, while [**R3**] relates to the *inspection finished before visiting all of the heatpoints* hazard. We have decomposed these system-level requirements further into child (component-level) requirements detailing the specific mitigation mechanisms captured in the BTDs. For example, the mitigations from Fig. 4.9 are related to the child requirements of [**R1**], while [**R3**] scopes which heat points should be visited are those that are reachable and have not been visited before. The full list of child requirements for these system-level requirements is presented in [21].

Step 4: Refine System Model According to Mitigations

Some identified mitigations required design modifications resulting in a refined system architecture (Fig. 4.10), reassessed in terms of hazards and mitigations. We present this as a single step for brevity, but there are iterations between these steps in practice. We consider the initial rover position and the charging position as user input. Note that the charging station position is static, and the rover always starts its missions from a predefined initial position.

We modified the original architecture by adding *MapValidator* to check that the initial position, charging position, obstacles, and heat points are mutually exclusive. Furthermore, *MapValidator* checks that the initial position, as recognized by *Vision*, is equal to the predefined *initialPosition*.

Next, we defined the *NavigationSystem* which contains the *ReasoningAgent* and the *Battery_Interface* components. We emphasize these two components as we focus on formally verifying them. We further decompose these components.

The *ReasoningAgent* takes as input the identified and validated obstacle locations, current rover position, heat points, and charger position. It outputs: (1) a plan from the current position to the goal (*plan2D*), (2) a plan from the goal to the charger location (*plan2C*), and (3) the list of *visited* locations. Within the *ReasoningAgent*, the goal reasoning agent (*GRA*) chooses the next goal as either the hottest heat point not yet visited or as the charger if the *recharge* flag is set to true by *Battery_Interface*. The *GRA* updates the *visited* locations.

The *ReasoningAgent* contains *ComputePlan2Charging* and *ComputePlan2Destination* which both have a *Planner* and plan reasoning agent (*PRA*). These return the shortest plan from the goal to the charger (*ComputePlan2Charging*) and the shortest plan from the current position to the goal (*ComputePlan2Destination*).

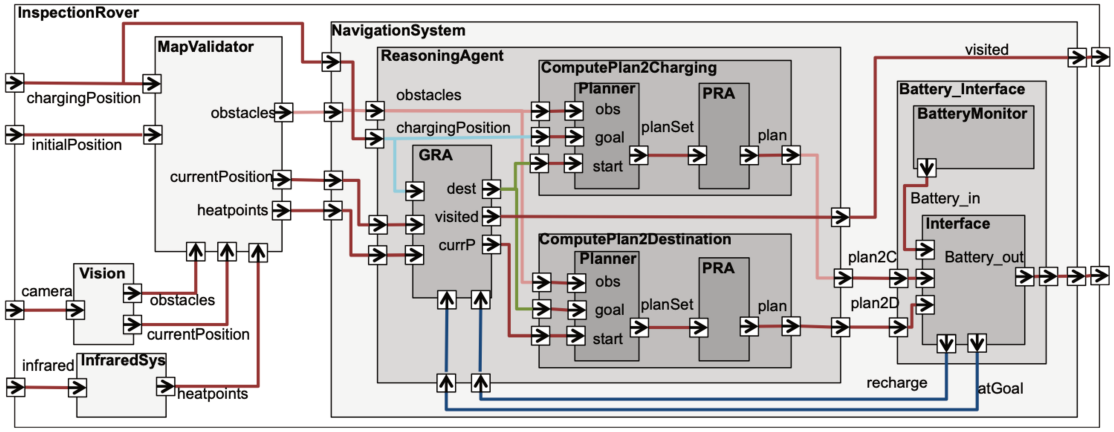


Figure 4.10: Upgraded inspection rover architecture with additional components and data.

Battery_Interface contains a *BatteryMonitor* and a hardware *Interface*. The *Interface* takes the plans from *NavigationSystem* and the battery status from the *BatteryMonitor* as input, and returns two flags indicating whether the rover has reached the goal (*atGoal*) and the status of the battery charge (*recharge*). The *recharge* flag becomes true if the current battery charge is insufficient to follow the plan to the goal (*plan2D*) and return to the charging station (*plan2C*).

If the *recharge* flag is false, then the *Interface* executes the plan and returns *atGoal* as true once it reaches the goal. However, if *recharge* is true, then *atGoal* is set to false. We note that we do not need both of these outputs since we always have *recharge* $\Rightarrow$ *not atGoal*. However, we include both for simplicity. These outputs are fed back to the *ReasoningAgent* that generates the next plan, and this loop executes until all of the heat points have been visited. Note that we assume that *NavigationSystem* and *Interface* have a similar execution frequency.

Step 5: Formalize Requirements and Create Formal Specifications

We manually wrote the requirements in the restricted natural language of FRET, i.e., FRETISH, see Section 3.1.2. For each FRETISH requirement, FRET produces natural language and diagrammatic explanations of its exact meaning and formalizes the requirement in temporal logic. The majority of the requirements that we formalized did not have scope or condition but they did have *always* timing, e.g:

[R1]: Navigation shall always satisfy battery > 0.
Other requirements use the condition field and *immediately* timing, e.g.:

[R1.2]: if recharge GRA shall immediately satisfy goal = chargePosition.
Notice that if recharge is a "trigger": the requirement is only enforced when the condition becomes true from false. The use of 'immediately' states that the response must hold simultaneously with each trigger point. The natural language version of **[R1]** was shown in step 3 above, while for **[R1.2]** it is: *"Charging station shall be selected as the next destination whenever the recharge flag is set to true"*.

Some requirements needed first-order temporal logic, which is not currently supported in FRET. So instead, we used auxiliary variables that we instantiated with quantifiers at the

LUSTRE level. For instance, the natural language requirement **[R3.3]** is *"The hottest heat point that was not visited before shall be the current goal when recharge flag is false"* and was written in FRETISH as follows:

[R3.3]: GRA shall always satisfy if ! recharge then (if forAll_i & i_inGrid then (if ! visited[i] then heatpoints[goal] >= heatpoints[i]))

where forAll_i represents the universal quantification over heat points. Our case study contains 28 requirements, 7 of these required first-order formulae. We were able to write all 28 requirements in FRETISH and formalize them.

FRETISH to verification Code: FRET automatically formalizes requirements in pure future-time (fmLTL) and pure past-time (pmLTL) Linear Temporal Logic. pmLTL formulae exclusively use past-time temporal operators, i.e., `Y, O, H, S`, (Yesterday, Once, Historically, Since, respectively). We used the pmLTL variant since LUSTRE-based analysis tools only accept pmLTL specifications. The automatically generated pmLTL formulae for **[R1]** and **[R1.2]** are:
[R1]: `H(battery>0);`
[R1.2]: `H((recharge & (Y(!recharge) | FTP))⇒(goal=chargePosition));`
where `FTP` means First Time Point of execution (equivalent to ¬ `Y TRUE`). From the pmLTL formulae, we automatically generated LUSTRE-based assume-guarantee contracts that can be directly fed into COCOSIM for verification (the full process is described in Section 3.1.3). For example, below is the generated LUSTRE code for **[R1]**:

```
guarantee "R1" (battery > 0);
```

If requirements were based only on model inputs, e.g., **[R1.2]**, then CoCoGen generates assumptions (instead of guarantees):

```
assume "R1.2" ((recharge and ((pre (not recharge)) or FTP)) => (goal = chargePosition));
```

where `FTP = true → false`. As mentioned earlier, some requirements used first-order logic quantification, such as **[R3.3]** which was generated as follows:

```
guarantee "R3.3" not recharge => (forall (i:int) (0 <= i and i < width) => (not visited[i] =>
    heatpoints[goal] >= heatpoints[i] )));
```

Notice that the `forAll_i` placeholder was replaced by `forall (i:int)`, and `i_inGrid` was replaced by `(0 <= i and i < width)` during generation.

We also specified the requirements in Event-B. Since Event-B does not support temporal logic, we used the FRETISH requirements to guide our modeling. FRETISH was simple enough and more useful as a starting point for formalization than the natural language requirements. E.g., the natural language requirement **[R3.4]** is *"The shortest path to the current goal shall be selected"*. The FRETISH version is: Planner shall always satisfy if (planningCompleted & returnPlan) then (if (forAll_x & x_inPlanSet) then (card(chosenPlan) <= card(x))), where the `card()` function computes the length of a path. The corresponding Event-B invariant was based on the FRETISH version:

$$(planningCompleted = TRUE) \wedge (returnplan = TRUE) \Rightarrow$$
$$(\forall\, x \cdot x \in PlanSet \Rightarrow card(chosenplan) \leq card(x))$$

Similarly, **[R2.5:]** *The calculated path to destination shall not include a location with an obstacle* was defined in Event-B as follows:

$\forall\, p, x \cdot p \in PlanSet \wedge x \in p \Rightarrow x \notin Obs$, where every `PlanSet` element is a set of grid locations.

Step 6: Perform verification and Simulation at System- and Component-Levels

Compositional verification in CoCoSim: Our objective was to attach the component-level child requirements to the relevant component(s) and then, using CoCoSim, compositionally verify the system-level parent requirements. We were unable to model/verify all requirements, e.g., *The current position as recognized by the rover is its current physical position* should be physically tested.

Compositional verification in CoCoSim involves defining a top system node with the associated system-level contract. During verification, the model checker attempts to show that these system-level properties can be successfully derived from the component-level contracts. Using compositional reasoning in CoCoSim, we were able to verify system-level requirements **[R1]** and **[R3]**, defined in step 3 above. However, we could not verify **[R2]**, which involves the *Vision* and the *Planner* components, because there is no CoCoSim model for the *Vision* component.

Compositional verification of **[R1]** was achieved relatively quickly (< 20 secs), as the model checker only had to analyze two components: the *Interface* and *BatteryMonitor* to verify **[R1]**. **[R3]** was more complex since it involved a loop between the *Interface* and *ReasoningAgent*. Kind2 had to carry out a lot of unrolling to adequately assess this property and deal with more complex contracts, including quantifiers and arrays. Thus, we were only able to prove **[R3]** for specific grid widths (minutes for 3×3, hours for 4×4, and larger grids timed out).

Component-Level verification Using Kind2 and Event-B: Previously, we used compositional verification to verify that the system-level parent requirements hold based on the component-level requirements. Here, our objective was to verify that the more detailed specification/implementation of individual components obeys the associated component-level requirements. Recognizing that, for autonomous robotic systems, it is often necessary to use a range of verification techniques for individual components, we used two distinct formal methods here [55, 94]. Specifically, we used Kind2 to verify a simple implementation of the *GRA* and, Event-B to model and verify the *ComputePlan* component.

Specification and verification of the GRA: We constructed a simple Lustre implementation of the *GRA* that we verified using Kind2. Full details can be found in [21]. The *GRA* computes the `start`, `goal` and the `visited` cells. The `start` is initialized as the `currentPosition`, if the goal was reached during the last execution (`atGoal` is true) then the start is the previous

goal (pre_goal), if the recharge flag is true then the start is the previous start position since the rover did not move. The goal is set to chargingPosition if the recharge flag is active. Otherwise, we choose the hottest heat point, computed using the hottestPoint local array that keeps track of the hottest heat point.

Kind2 verified all the properties specified in the specification. Most of the properties were verified in less than 1 second. Due to state space explosion, some properties, e.g., requirement [**R3.3**], were only provable for specific grid sizes. E.g., we verified [**R3.3**] for a grid size up to 4x4.

Specification and verification of ComputePlan using Event-B: Our Event-B model contains three contexts (modeling static aspects) and two machines (modeling dynamic aspects). Event-B supports formal refinement, so our contexts extend one another and our machines indicate refinement steps. Our most primitive context, ctx0, specifies basic details such as the size of the grid, valid grid locations, obstacles, and heat points. We do not explicitly list the elements of these sets since this specification is for a generic planner. This is extended via ctx1, which specifies functions that capture the behavior of the planning component.

The abstract machine, mac0, models a simple search-based planning algorithm that produces a set of plans containing the start and goal. Event-B uses Sets as primitive, so we ensure that these plans, encoded as Sets, can be linearized using the adjacent function specified in ctx1. The refinement, mac1, incorporates a plan reasoning agent and chooses the shortest plan from PlanSet. Another context, ctx2, defines a constant to limit the number of generated plans.

We encoded [**R2.1**], [**R2.4.1**], [**R2.4.3**], [**R2.4.4**], [**R2.5**] and [**R3.4**] in our Event-B model. We could not verify [**R2**] compositionally but its child requirements feature in our Event-B model (e.g. [**R2.5**]). This ensures that the planning components do not accidentally cause the rover to collide with an obstacle. Most of the Event-B proof obligations were proven automatically by Atelier-B in Rodin. Those requiring interactive proof were relatively straightforward.

Event-B was not limited by the state space explosion that caused Kind2 to time out. Instead, we specified more complex component-level properties that would have been difficult to verify for a model-checker. The Event-B model can be found in [21].

Step 7: Document verification Results and Build Safety Case

All of the verification results produced by the tools are part of the safety case constructed in AdvoCATE. Some artifacts were imported automatically into AdvoCATE, while others were added manually. Since this case study did not include a complete system implementation, the safety case we report here is an interim version and contains the current safety assurance status.

The skeleton of the overall argument is generated automatically from the information defined and imported into AdvoCATE, such as hazards, mitigation requirements, formalized requirements, and evidence artifacts. We have extended the skeleton argument based on our specific application and tools. For example, Fig. 4.11 presents an argument fragment about mitigating the *running out of battery* hazard that causes *loss of rover*. Similar arguments exist for other causes of *loss of rover* and the other hazards. For brevity, Fig. 4.11 only contains a fragment of the existing argument. For example, this argument focuses on two aspects: the requirements di-

rectly related to this hazard (right branch) and the causes that lead to the hazard (left branch). Full details can be found in [21].

The goal **G14** focuses on **[R1]** that was verified using CoCoSim. We built a similar argument for each system-level requirement previously verified compositionally with CoCoSim. For each argument, we extended the automatically generated part with a combination of existing argumentation patterns [50, 126] to support application-specific goals (base of Fig. 4.11):

1. the formalization of the natural language requirement is correct (**G3-A1**);
2. the results from CoCoSim are trustworthy (**G4-A1**);
3. the different design representations are consistent (**G5-A1**);
4. the CoCoSim verification result for **[R1]** is valid (**G6-A1**).

To ensure that the different design representations were consistent across the tools, we performed manual reviews where automated consistency validation was unavailable. E.g., we used manual reviews to verify that the design as specified in AdvoCATE was consistent with the Simulink, Kind2, and Event-B models.

The goal **G3-A1** focuses on the correct specification of **[R1]** in fretish and the correct functioning of fret. While we have to verify through a manual review that the natural language requirement is correctly represented in fretish, the correct fret functioning and generation of the corresponding CoCoSim contracts are supported by the automated verification framework of fret.

The goal **G6-A1** focuses on the validity of **[R1]** through analysis with the CoCoSim tool. This part of the argument points out the dependencies to the properties of the other components and implicit assumptions on which these results rely. Finally, to have confidence in the results from CoCoSim, we argued the trustworthiness of CoCoSim with the goal **G4-A1**. Since CoCoSim relies on model transformations and external tools for verification, this correctness must be established. For example, we argued about the correctness of the translation from Simulink to Lustre code that Kind2 uses.

4.2.3 Discussion

Using the given formal verification tools, we verified that the Navigation System would not cause the rover to run out of battery. Due to the specification complexity, we could not verify that a collision will never occur at the system level with CoCoSim. However, we verified with Event-B that the Navigation System would not generate plans that contain obstacles at the component level. Finally, we verified that the rover would visit all the heat points with CoCoSim, but only for a small grid size of 4x4. Verifying the property for more significant grid sizes did not terminate, even after several days of analysis.

This case study showed us that by following our methodology, we could leverage multiple formal tools and use them *complementary*. In this way, we applied formal methods to small, manageable chunks of the system to ease the verification burden and avoid becoming trapped

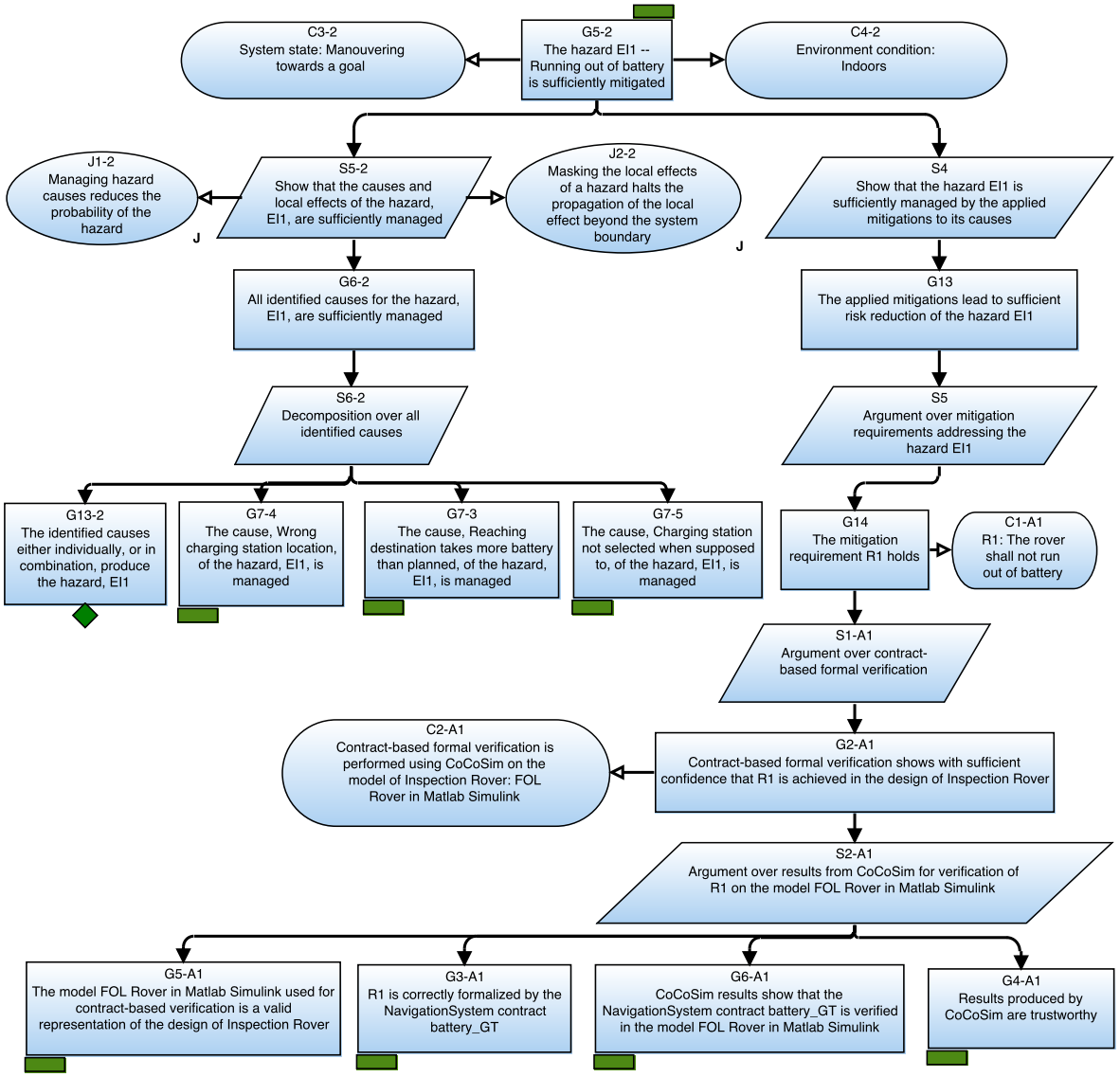


Figure 4.11: The argument-fragment for the *running out of battery* hazard (rectangles represent goals, parallelograms represent strategies, ovals with a 'J' represent justifications, rounded rectangles represent context statements, green rectangles indicate arguments continue elsewhere, green diamonds represent currently undeveloped elements).

by the limitations of any single tool. Using FRET to bridge the gap between the informal and formal steps by formalizing our requirements was particularly useful because it helped us clarify any details implicit in the natural language requirements.

Although the initial natural language requirements looked relatively straightforward in most cases, a closer study revealed many questions regarding their precise meaning. Translating the natural language requirements into FRETISH was not always straightforward. To this end, the semantic explanations and simulation capabilities offered by FRET were instrumental in ensuring that the FRETISH requirements captured our intended semantics. Notice that we could not directly encode first-order logic requirements in FRETISH. We tackled this problem using auxiliary variables as placeholders for the quantifiers at the requirement level, but a FRETISH-level solution is desirable. Finally, we noticed that most of the Inspection Rover requirements follow a small number of patterns, a characteristic that we have observed in other studies within our organization.

The choice of CoCoSim and Kind2 greatly influenced our design decisions. For example, our original design represented cells in the grid as (x, y)-coordinates. However, we subsequently simplified this by using indices to make them easier to represent and reason about in formal tools. Our choice of a compositional verification approach caused us to output specific variables such as the remaining battery power to verify **[R1]** compositionally. Furthermore, we had to adapt the hierarchical structure of the system to accommodate compositional verification. If the choice of formal verification tools is made early in the system development process, the system's design can be more suitable for formal verification using the chosen method(s).

Not all of the formalized requirements were formally verifiable; some described hardware constraints and/or required physical testing. The latter supports the claim that the robotics domain requires formal and informal verification processes [55]. E.g., everything depends on the accuracy of the rover's current position - a property that we could not formally verify in this case. However, we can potentially incorporate run-time analysis by formalizing the requirements to be verified via testing. Specifically, the formalized properties can be used to generate formal run-time monitors to help with fault management during operation. These might help to create recovery barriers in the bow-tie diagrams. In this way, we could include the development of fault management at design time.

Integrating the verification results from the different formal methods in an assurance case required intensive cooperation between the assurance and formal methods experts. The effort required identifying dependencies between different tools, understanding the techniques and the tool implementations, and implicit assumptions on which analyses were run and results interpreted. The activity was exceptionally performed ad hoc. A more systematic approach to gathering assurance information from formal methods applications would be beneficial.

Integrating formal methods often rely on bespoke translations between languages/tools. However, these translations can be difficult and sometimes impossible to formalize/implement correctly. Further, if used in an assurance case, the translations themselves must be assured, as for our translation from FRET to CoCoSim [21]. Although the use of tightly integrated formal methods is desirable, our approach, using an assurance case as the point of integration, incorporates tools for which such systematic translations do not exist by providing arguments demonstrating how to link models in distinct formalisms.

The case study helped us to identify limitations in the tools used (AdvoCATE, FRET, CoCoSim, and Event-B) for robotics applications. It prompted an update to CoCoSim to incorporate unimplemented abstract components. Specifically, CoCoSim now generates Lustre code for these components using the `imported` keyword when no implementation is available. Other limitations include the lack of FRET support for abstract data types, which caused us to edit the FRET-generated CoCoSim contracts manually. There were some difficulties when attempting to automatically import verification artifacts directly from the tools into AdvoCATE, which caused us to insert some details manually.

Our methodology follows the development phases of existing development guidelines [85, 48] and builds on top of them through a set of steps (§4.2.1), which are guided by the need to devise an assurance case that integrates artifacts from different tools. Although in the presented work, we used specific tools, we believe that our methodology can be followed irrespective of the choice of tools.

4.3 Conclusion

In this chapter, we demonstrated that it is possible to perform an end-to-end analysis of CPS models using different frameworks, including ours, CoCoSim. We could automatically generate verification codes, monitors, and traces from requirements. We could also automatically connect requirements to Simulink models and verify the models against the requirements. We have identified some areas for improvement in the used frameworks, mainly related to capturing temporal requirements in FRET and analyzing non-linear functions in CoCoSim.

Additionally, we demonstrated that using different formal verification frameworks in an integrated approach is possible by using an assurance case as a point of integration. This approach allowed us to overcome the limitations of any single tool and take advantage of their complementary features. Overall, we believe that this is a promising approach for designing and verifying critical systems.

Chapter 5

Conclusion & Perspectives

In this Ph.D. thesis, we addressed the problem of formal verification of Simulink/Stateflow models. This work was organized into four chapters.

- The first Chapter provided a general overview of the problem and discussed related work in the context of Simulink/Stateflow formal verification.
- In the second Chapter, we proposed an approach for translating Simulink/Stateflow models to formal languages, specifically to the Lustre synchronous language. We also developed a reverse compilation of Lustre models to CoCoSim. The latter has many applications, from the CoCoSim tool validation, as discussed in Chapter 3, to bridging the gap between requirements and Simulink model analysis.
- The proposed approach is implemented in the CoCoSim toolbox presented in the third Chapter. CoCoSim aims to ease verification and validation activities for Simulink models. The toolbox is highly automated and has customizable and configurable architecture allowing other techniques to be integrated to increase scalability.
- The fourth Chapter demonstrated the use of CoCoSim in some case studies. We have applied CoCoSim to several industrial and academic case studies, including the ten Lockheed Martin CyberPhysical challenges and the navigation rover case study. The results show that CoCoSim can effectively verify Simulink models, especially when using compositional verification. Formal methods, such as model checking, can be limited when dealing with large numerically intensive and non-linear models, hence the need to integrate various verification techniques, as demonstrated in the Rover case study in Section 4.2.

There are many possible directions for future work. One possible direction is to improve the toolbox to support more features of Simulink/Stateflow. For instance, mixing continuous-time discrete-time systems. While Lustre forbids us to express these systems, CoCoSim is generic enough to be able to process these blocks once a suitable formal intermediate language is available. Zelus [23] is a hybrid extension of Lustre, fitted with a code generator providing faithful simulation means. This hybrid model could also be used to validate properties using hybrid model checking [9] and combining guaranteed integration with SMT solvers. Another related approach would be the expression of a Simulink model as a hybrid automaton [103], enabling the use of tools such as SpaceEx [58] or Flow* [36].

The second direction is to address the verification of digital filters and numerical controllers. While current tools could analyze them, they usually perform poorly. Future work targets the integration of (non-linear) invariant generation [4, 5, 112, 113, 114, 121, 132] and their re-validation at the code level [46, 80, 133].

Another direction is to integrate floating point imprecision computation [6, 66] and the accurate compilation of numerical computation [43, 44]. Additionally, it would be interesting to investigate the use of CoCoSim in more industrial case studies.

Bibliography

[1] DO-178C. *Software Considerations in Airborne Systems and Equipment Certification*. 2011 (cit. on p. 60).

[2] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010 (cit. on p. 118).

[3] AdaCore and Altran UK Ltd. *SPARK 2014 Reference Manual*. 2020 (cit. on p. 31).

[4] Assale Adje, Pierre-Loïc Garoche, and Victor Magron. "Property-based Polynomial Invariant Generation using Sums-of-Squares Optimization". In: *Static Analysis - 22st International Symposium, SAS 2015, St Malo, France, 2015. Proceedings*. Ed. by Sandrine Blazy and Thomas Jensen. Vol. 9291. Lecture Notes in Computer Science. Springer, 2015, pp. 235–251 (cit. on p. 133).

[5] Assalé Adjé and Pierre-Loïc Garoche. "Automatic synthesis of k-inductive piecewise quadratic invariants for switched affine control programs". In: *Computer Languages, Systems & Structures (COMLAN)* 47 (2017), pp. 44–61 (cit. on p. 133).

[6] Assalé Adjé, Pierre-Loïc Garoche, and Alexis Werey. "Quadratic Zonotopes - An Extension of Zonotopes to Quadratic Arithmetics". In: *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*. 2015, pp. 127–145 (cit. on p. 134).

[7] Aditya Agrawal, Gyula Simon, and Gabor Karsai. "Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations". In: *Electr. Notes Theor. Comput. Sci.* 109 (2004), pp. 43–56 (cit. on p. 30).

[8] Andrew W. Appel. *Compiling with Continuations (corr. version)*. Cambridge University Press, 2006 (cit. on p. 52).

[9] Kyungmin Bae and Sicun Gao. "Modular SMT-based analysis of nonlinear hybrid systems". In: *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. 2017, pp. 180–187 (cit. on p. 133).

[10] Patrick Baudin et al. *ACSL: ANSI/ISO C Specification Language, version 1.16*. 2020 (cit. on p. 31).

[11] Albert Benveniste and Gérard Berry. *The Synchronous approach to reactive and real-time systems*. Research Report RR-1445. INRIA, 1991 (cit. on p. 26).

[12] Albert Benveniste et al. "The synchronous languages 12 years later". In: *IEEE* 91.1 (2003), pp. 64–83 (cit. on p. 36).

[13] Dariusz Biernacki et al. "Clock-directed modular code generation for synchronous data-flow languages". In: *LCTES'08*. 2008 (cit. on pp. 62, 63, 67, 90).

[14] Pontus Boström and Lionel Morel. "Mode-Automata in Simulink/Stateflow". In: *TUCS Technical Report No 772, September 2006*. 2006 (cit. on p. 30).

[15] Hamza Bourbouh. *CoCoSim – Automated Analysis Framework for Simulink*. `https://github.com/NASA-SW-VnV/CoCoSim` (cit. on pp. 10, 52, 73, 78, 84, 85, 104, 118).

[16] Hamza Bourbouh, Guillaume Brat, and Pierre-Loïc Garoche. “CoCoSim: an automated analysis framework for Simulink/Stateflow”. In: *Model Based Space Systems and Software Engineering-European Space Agency Workshop (MBSE 2020)*. 2020 (cit. on p. 14).

[17] Hamza Bourbouh et al. “Automated analysis of Stateflow models”. In: *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Ed. by Thomas Eiter and David Sands. Vol. 46. EPiC Series in Computing. EasyChair, 2017, pp. 144–161 (cit. on pp. 6, 14, 36, 52, 62).

[18] Hamza Bourbouh et al. “CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models”. In: *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. 2020 (cit. on p. 14).

[19] Hamza Bourbouh et al. “From Lustre to Simulink: Reverse Compilation for Embedded Systems Applications”. In: *ACM Transactions on Cyber-Physical Systems* 5.3 (2021), pp. 1–20 (cit. on pp. 15, 36, 72).

[20] Hamza Bourbouh et al. “Integrating Formal Verification and Assurance: An Inspection Rover Case Study”. In: *NASA Formal Methods*. Ed. by Aaron Dutle et al. Cham: Springer International Publishing, 2021, pp. 53–71 (cit. on pp. 14, 72, 100).

[21] Hamza Bourbouh et al. *Integration and Evaluation of the AdvoCATE, FRET, CoCoSim, and Event-B Tools on the Inspection Rover Case Study*. Tech. rep. TM–2020–5011049, NASA, 2021 (cit. on pp. 120, 121, 123, 126–128, 130).

[22] Hamza Bourbouh et al. *Integration and Evaluation of the Advocate, FRET, CoCoSim, and Event-B Tools on the Inspection Rover Case Study*. Tech. rep. 2020 (cit. on pp. 15, 72, 100).

[23] Timothy Bourke and Marc Pouzet. “Zélus: a synchronous language with ODEs”. In: *Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013, April 8-11, 2013, Philadelphia, PA, USA*. 2013, pp. 113–118 (cit. on p. 133).

[24] Timothy Bourke et al. “A Formally Verified Compiler for Lustre”. In: *PLDI 2017 - 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. Barcelone, Spain, June 2017 (cit. on pp. 26, 62).

[25] Aaron R. Bradley. “IC3 and beyond: Incremental, Inductive Verification”. In: *CAV’12*. 2012 (cit. on p. 90).

[26] Aaron R. Bradley. “SAT-Based Model Checking without Unrolling”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87 (cit. on p. 90).

[27] Guillaume Brat et al. “Verifying the Safety of a Flight-Critical System”. In: *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. Ed. by Nikolaj Bjørner and Frank S. de Boer. Vol. 9109. Lecture Notes in Computer Science. Springer, 2015, pp. 308–324 (cit. on p. 96).

[28] Rafael C Cardoso et al. “Heterogeneous Verification of an Autonomous Curiosity Rover”. In: *NASA Formal Methods Symposium*. Springer. 2020, pp. 353–360 (cit. on p. 118).

[29] Paul Caspi et al. “Lustre: A Declarative Language for Programming Synchronous Systems”. In: *POPL’87*. 1987, pp. 178–188 (cit. on pp. 4, 13, 26, 29, 31).

[30] Adrien Champion et al. “CoCoSpec: A Mode-Aware Contract Language for Reactive Systems”. In: *SEFM'16*. 2016, pp. 347–366 (cit. on pp. 32, 90).

[31] Adrien Champion et al. “CoCoSpec: A Mode-Aware Contract Language for Reactive Systems”. In: *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*. Ed. by Rocco De Nicola and Eva Kühn. Vol. 9763. Lecture Notes in Computer Science. Springer, 2016, pp. 347–366 (cit. on pp. 5, 31, 60–62, 77, 86, 95).

[32] Adrien Champion et al. “The Kind 2 Model Checker”. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 510–517 (cit. on pp. 4, 8, 29, 90, 91, 104).

[33] Adrien Champion et al. “The Kind 2 Model Checker”. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 510–517 (cit. on pp. 5, 31).

[34] Swarat Chaudhuri and Azadeh Farzan, eds. *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016.

[35] Chunqing Chen et al. “Formal modeling and validation of Stateflow diagrams”. In: *International Journal on Software Tools for Technology Transfer* 14.6 (2012), pp. 653–671 (cit. on p. 30).

[36] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. “Flow*: An Analyzer for Non-linear Hybrid Systems”. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 258–263 (cit. on p. 133).

[37] Alessandro Cimatti and Stefano Tonetta. “A Property-Based Proof System for Contract-Based Design”. In: *SEAA'12*. 2012, pp. 21–28 (cit. on pp. 5, 31).

[38] Alessandro Cimatti et al. “Nusmv 2: An opensource tool for symbolic model checking”. In: *International Conference on Computer Aided Verification*. Springer. 2002, pp. 359–364 (cit. on p. 79).

[39] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. “Mixing signals and modes in synchronous data-flow systems”. In: *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*. 2006, pp. 73–82 (cit. on pp. 29, 58).

[40] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. “A conservative extension of synchronous data-flow with state machines”. In: *EMSOFT 2005*. 2005, pp. 173–182 (cit. on pp. 29, 58, 62).

[41] *Continuation-passing style denotational semantics for Stateflow*. `https://github.com/ploc/stateflow_CPS_semantics` (cit. on p. 58).

[42] A. Crapo et al. “Requirements Capture and Analysis in ASSERT^{TM}”. In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. Sept. 2017, pp. 283–291 (cit. on p. 117).

[43] Nasrine Damouche and Matthieu Martel. “Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs”. In: *Automated Formal Methods, AFM@NFM 2017, Moffett Field, CA, USA, May 19-20, 2017*. 2017, pp. 63–76 (cit. on p. 134).

[44] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. “Numerical program optimisation by automatic improvement of the accuracy of computations”. In: *IJIEI* 6.1/2 (2018), pp. 115–145 (cit. on p. 134).

[45] Olivier Danvy. “Defunctionalized Interpreters for Programming Languages”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’08. Victoria, BC, Canada: ACM, 2008, pp. 131–142 (cit. on p. 58).

[46] Guillaume Davy et al. “Ensuring functional correctness of cyber-physical system controllers: from model to code analyses”. In: *Forum on Specification and Design Languages, Special session on Logic and Mathematics Behind Design Automation, FDL’18, TU Munich 10.9-12.9.2018*. Sept. 2018 (cit. on p. 133).

[47] Ewen Denney and Ganesh Pai. “Architecting a safety case for UAS flight operations”. In: *International System Safety Conference*. Vol. 12. 2016 (cit. on p. 122).

[48] Ewen Denney and Ganesh Pai. “Automating the assembly of aviation safety cases”. In: *IEEE Transactions on Reliability* 63.4 (2014), pp. 830–849 (cit. on pp. 119, 122, 131).

[49] Ewen Denney and Ganesh Pai. “Tool Support for Assurance Case Development”. In: *Automated Software Engineering* 25.3 (2018), pp. 435–499 (cit. on pp. 118, 119).

[50] Ewen W Denney and Ganesh J Pai. *Safety case patterns: theory and applications*. Tech. rep. 2015 (cit. on p. 128).

[51] Homayoon Dezfuli et al. “NASA System Safety Handbook. Volume 2: System Safety Concepts, Guidelines, and Implementation Examples”. In: (2015) (cit. on p. 119).

[52] Arnaud Dieumegard et al. “Compilation of synchronous observers as code contracts”. In: *SAC’15*. 2015, pp. 1933–1939 (cit. on pp. 4, 8, 29, 60, 62, 86).

[53] Chris Elliott. “An Example Set of Cyber-Physical V&V Challenges for S5, Lockheed Martin Skunk Works”. In: *Safe & Secure Systems and Software Symposium (S5), 12-14 July 2016, Dayton, Ohio*. Ed. by Air Force Research Laboratory. 2016 (cit. on pp. 73, 84, 100, 101).

[54] Chris Elliott. “On Example Models and Challenges Ahead for the Evaluation of Complex Cyber-Physical Systems with State of the Art Formal Methods V&V, Lockheed Martin Skunk Works”. In: *Safe & Secure Systems and Software Symposium (S5), 9-11 July 2015, Dayton, Ohio*. Ed. by Air Force Research Laboratory. 2015 (cit. on pp. 84, 100).

[55] Marie Farrell, Matt Luckcuck, and Michael Fisher. “Robotics and Integrated Formal Methods: Necessity meets Opportunity”. In: *International Conference on Integrated Formal Methods*. Springer. 2018, pp. 161–171 (cit. on pp. 118, 126, 130).

[56] Aaron W. Fifarek et al. “SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements”. In: *NASA Formal Methods Symposium*. 2017, pp. 420–426 (cit. on p. 117).

[57] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3—where programs meet provers”. In: *European symposium on programming*. Springer. 2013, pp. 125–128 (cit. on p. 118).

[58] Goran Frehse. “Reachability of hybrid systems in space-time”. In: *2015 International Conference on Embedded Software, EMSOFT 2015, Amsterdam, Netherlands, October 4-9, 2015*. 2015, pp. 41–50 (cit. on p. 133).

[59] Andrew Gacek et al. “The JKind Model Checker”. In: *CoRR* abs/1712.01222 (2017). arXiv: `1712.01222` (cit. on pp. 4, 8, 29, 90).

[60] Andrew Gacek et al. “Towards realizability checking of contracts using theories”. In: *NASA Formal Methods Symposium*. Springer. 2015, pp. 173–187 (cit. on p. 111).

[61] Pierre-Loïc Garoche, Arie Gurfinkel, and Temesghen Kahsai. “Synthesizing Modular Invariants for Synchronous Code”. In: *HCVS'14*. 2014 (cit. on pp. 90, 92, 94).

[62] Pierre-Loïc Garoche, Temesghen Kahsai, and Xavier Thirioux. “Hierarchical State Machines as Modular Horn Clauses”. In: *HCVS'16*. 2016, pp. 15–28 (cit. on pp. 29, 59, 62, 86, 90).

[63] Pierre-Loïc Garoche, Temesghen Kahsai, and Xavier Thirioux. *LustreC* (cit. on pp. 84, 90, 91).

[64] Pierre-Loïc Garoche et al. “Testing-Based Compiler Validation for Synchronous Languages”. In: *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*. 2014, pp. 246–251 (cit. on p. 91).

[65] Pierre-Loïc Garoche et al. “Testing-based compiler validation for synchronous languages”. In: *NASA Formal Methods Symposium*. Springer. 2014, pp. 246–251 (cit. on p. 63).

[66] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. “The Zonotope Abstract Domain Taylor1+”. In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. 2009, pp. 627–633 (cit. on p. 134).

[67] Shalini Ghosh et al. “ARSENAL: Automatic Requirements Specification Extraction from Natural Language”. In: *NASA Formal Methods Symposium*. 2016, pp. 41–46 (cit. on p. 117).

[68] Dimitra Giannakopoulou et al. “Formal Requirements Elicitation with FRET”. In: *Requirements Engineering: Foundation for Software Quality (Demo-Track)*. 2020 (cit. on p. 118).

[69] Dimitra Giannakopoulou et al. “Generation of Formal Requirements from Structured Natural Language”. In: *REFSQ2020*. 2020 (cit. on pp. 61, 84).

[70] Dimitra Giannakopoulou et al. “Generation of Formal Requirements from Structured Natural Language”. In: *Requirements Engineering: Foundation for Software Quality*. Ed. by Nazim Madhavji et al. Cham: Springer International Publishing, 2020, pp. 19–35 (cit. on pp. 73, 78, 83, 110).

[71] CriSys Group. *Generic patient controlled analgesia infusion pump project*. `https://crysis.cs.umn.edu/gpca.shtml` (cit. on p. 94).

[72] *GSN Community Standard Version 2*. Tech. rep. Assurance Case Working Group of The Safety-Critical Systems Club, Jan. 2018 (cit. on p. 119).

[73] N. Halbwachs et al. “The synchronous dataflow programming language LUSTRE”. In: *Proceedings of the IEEE*. 1991, pp. 1305–1320 (cit. on pp. 5, 31).

[74] Nicholas Halbwachs et al. “The synchronous data flow programming language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320 (cit. on pp. 26, 77).

[75] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. “Synchronous Observers and the Verification of Reactive Systems”. In: *AMAST’93*. 1993, pp. 83–96 (cit. on pp. 32, 60).

[76] Nicolas Halbwachs and Pascal Raymond. “Validation of Synchronous Reactive Systems: From Formal Verification to Automatic Testing”. In: *ASIAN’99*. 1999, pp. 1–12 (cit. on p. 60).

[77] Grégoire Hamon. “A denotational semantics for stateflow”. In: *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*. 2005, pp. 164–172 (cit. on pp. 5, 24, 30, 31, 52, 53, 56, 57).

[78] Grégoire Hamon and John M. Rushby. “An Operational Semantics for Stateflow”. In: *FASE 2004*. 2004, pp. 229–243 (cit. on pp. 5, 24, 30, 31).

[79] Grégoire Hamon and John M. Rushby. “An operational semantics for Stateflow”. In: *STTT* 9.5-6 (2007), pp. 447–456 (cit. on pp. 5, 24, 30, 59).

[80] Heber Herencia-Zapana et al. “PVS Linear Algebra Libraries for Verification of Control Software Algorithms in C/ACSL”. In: *NASA Formal Methods - Forth International Symposium, NFM 2012, Norfolk, VA USA, April 3-5, 2012. Proceedings*. Ed. by Alwyn Goodloe and Suzette Person. Vol. 7226. Lecture Notes in Computer Science. Springer, 2012, pp. 147–161 (cit. on p. 133).

[81] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (1969), pp. 576–580 (cit. on pp. 31, 60).

[82] Bertrand Jeannet and Fabien Gaucher. “Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study”. In: *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. TOULOUSE, France, Jan. 2016 (cit. on pp. 73, 117).

[83] Temesghen Kahsai and Cesare Tinelli. “PKind: A parallel k-induction based model checker”. In: *PDMC’11*. 2011, pp. 55–62 (cit. on pp. 62, 86, 95, 96).

[84] Andreas Katis et al. “Synthesis from Assume-Guarantee Contracts using Skolemized Proofs of Realizability”. In: *CoRR* abs/1610.05867 (2016) (cit. on pp. 5, 31).

[85] John C Kelly. “Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems Volume II: A Practitioner’s Companion”. In: (1997) (cit. on pp. 119, 131).

[86] Uri Klein and Amir Pnueli. “Revisiting Synthesis of GR(1) Specifications”. In: *HVC’10*. 2010, pp. 161–181 (cit. on pp. 5, 31).

[87] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. “SMT-based model checking for recursive programs”. In: *Formal Methods in System Design* 48.3 (2016), pp. 175–205 (cit. on p. 90).

[88] Anvesh Komuravelli et al. “Automatic Abstraction in SMT-Based Unbounded Software Model Checking”. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 2013, pp. 846–862 (cit. on p. 90).

[89] Orna Kupferman and Moshe Y Vardi. “Vacuity detection in temporal model checking”. In: *International Journal on Software Tools for Technology Transfer* 4.2 (2003), pp. 224–233 (cit. on p. 116).

[90] Julia L. Lawall and Olivier Danvy. “Separating Stages in the Continuation-Passing Style Transformation”. In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*. 1993, pp. 124–136 (cit. on p. 52).

[91] Meng Li and Ratnesh Kumar. “Recursive Modeling of Stateflow as Input/Output-Extended Automaton”. In: *IEEE Trans. Automation Science and Engineering* 11.4 (2014), pp. 1229–1239 (cit. on p. 30).

[92] Jing Liu et al. *From Design Contracts to Component Requirements Verification*. 2016 (cit. on pp. 5, 31, 70, 84).

[93] Levi Lúcio et al. “EARS-CTRL: Generating Controllers for Dummies”. In: *Proceedings of MODELS 2017 Satellite Event*. Vol. 2019. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 566–570 (cit. on p. 117).

[94] Matt Luckcuck et al. “Formal Specification and Verification of Autonomous Robotic Systems: A Survey”. In: *ACM Computing Surveys (CSUR)* 52.5 (2019), pp. 1–41 (cit. on pp. 118, 126).

[95] MathWorks. *Simulink Coder*. `https://www.mathworks.com/help/dsp/ug/generate-code-from-simulink.html` (cit. on p. 93).

[96] MathWorks. *Stateflow*. `http://www.mathworks.com/products/stateflow/` (cit. on pp. 3, 24).

[97] *MATLAB Simulink (R2018b)*. The Mathworks, Inc. Natick, Massachusetts, 2018 (cit. on pp. 2, 18).

[98] A. Mavridou et al. “The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained”. In: *International Requirements Engineering Conference (RE)*. IEEE, 2020, pp. 300–310 (cit. on pp. 14, 72, 100).

[99] Anastasia Mavridou et al. “Bridging the Gap Between Requirements and Simulink Model Analysis”. In: *Joint 26th International Conference on Requirements Engineering: Foundation for Software Quality Workshops, Doctoral Symposium, Live Studies Track, and Poster Track*. Pise, Italy, Mar. 2020 (cit. on pp. 14, 72, 100, 109, 117).

[100] Anastasia Mavridou et al. *Evaluation of the FRET and CoCoSim tools on the Ten Lockheed Martin Cyber-Physical Challenge Problems*. Tech. rep. 84 pages. NASA, Aug. 2019 (cit. on pp. 15, 72, 84, 100, 101).

[101] David McComas. “NASA/GSFC’s Flight Software Core Flight System”. In: *Flight Software Workshop*. Vol. 11. 2012 (cit. on p. 118).

[102] B. Meenakshi, Abhishek Bhatnagar, and Sudeepa Roy. “Tool for Translating Simulink Models into Input Language of a Model Checker”. In: *Formal Methods and Software Engineering*. Ed. by Zhiming Liu and Jifeng He. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 606–620 (cit. on pp. 4, 5, 30).

[103] Stefano Minopoli and Goran Frehse. “SL2SX Translator: From Simulink to SpaceEx Models”. In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016*. 2016, pp. 93–98 (cit. on p. 133).

[104] Shiva Nejati et al. “Evaluating Model Testing and Model Checking for Finding Requirements Violations in Simulink Models”. In: *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19*. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 1015–1025 (cit. on pp. 100, 117).

[105] Claire Pagetti et al. “Multi-task Implementation of Multi-periodic Synchronous Programs”. In: *Discrete Event Dynamic Systems* 21 (Sept. 2011), pp. 307–338 (cit. on pp. 46, 49).

[106] Claire Pagetti et al. “The ROSACE case study: From Simulink specification to multi/many-core execution”. In: *RTAS'14*. 2014 (cit. on p. 96).

[107] P. J. Pingree et al. “Validation of mission critical software design and implementation using model checking [spacecraft]”. In: *Digital Avionics Systems Conference, 2002. Proceedings. The 21st*. Vol. 1. Oct. 2002, 6A4-1-6A4–12 vol.1 (cit. on p. 30).

[108] Gordon D. Plotkin. “Call-by-Name, Call-by-Value and the lambda-Calculus”. In: *Theor. Comput. Sci.* 1.2 (1975), pp. 125–159 (cit. on p. 52).

[109] Wolfgang Puffitsch, Eric Noulard, and Claire Pagetti. “Off-line Mapping of Multi-rate Dependent Task Sets to Many-core Platforms”. In: *Real-Time Systems* 51.5 (Sept. 2015), pp. 526–565 (cit. on p. 46).

[110] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. 2009, p. 5 (cit. on p. 118).

[111] Vıctor Rivera and Néstor Catano. “Translating Event-B to JML-specified Java programs”. In: *ACM Symposium on Applied Computing*. 2014, pp. 1264–1271 (cit. on p. 118).

[112] Pierre Roux and Pierre-Loïc Garoche. “Practical Policy Iterations A practical use of policy iterations for static analysis - The quadratic case.” In: *Formal Methods in System Design* 46.2 (2015), pp. 163–196 (cit. on p. 133).

[113] Pierre Roux, Romain Jobredeaux, and Pierre-Loïc Garoche. “Closed Loop Analysis of Control Command Software”. In: *18th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'15, Seattle, Washington, USA, April 14-16, 2015,* ed. by Antoine Girard and Sriram Sankaranarayanan. 2015, pp. 108–117 (cit. on p. 133).

[114] Pierre Roux et al. “A Generic Ellipsoid Abstract Domain for Linear Time Invariant Systems”. In: *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2012, Beijing, China, April 17-19, 2012,* ed. by Thao Dang and Ian Mitchell. ACM, 2012, pp. 105–114 (cit. on p. 133).

[115] Special C. RTCA. *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*. 2011 (cit. on p. 72).

[116] Special C. RTCA. *DO-333 Formal Methods Supplement to DO-178C and DO-278A*. Tech. rep. Radio Technical Commission for Aeronautics, Dec. 2011 (cit. on p. 72).

[117] John Rushby. “The Versatile Synchronous Observer”. In: *Specification, Algebra, and Software, A Festschrift Symposium in Honor of Kokichi Futatsugi*. LNCS. 2014 (cit. on pp. 32, 60).

[118] M. Sampson and V. Derevenko. “Interface definition document (IDD) for international space station (ISS) visiting vehicles (VVs)”. In: *NASA Technical Report* (2000) (cit. on p. 94).

[119] Norman Scaife et al. “Defining and translating a "safe" subset of Simulink/Stateflow into Lustre”. In: *EMSOFT 2004*. 2004, pp. 259–268 (cit. on pp. 30, 31, 93).

[120] Steve Schneider, Helen Treharne, and Heike Wehrheim. “A CSP approach to control in Event-B”. In: *International Conference on Integrated Formal Methods*. Springer. 2010, pp. 260–274 (cit. on p. 118).

[121] Yassamine Seladji and Olivier Bouissou. “Numerical Abstract Domain Using Support Functions”. In: *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*. 2013, pp. 155–169 (cit. on p. 133).

[122] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. “Checking Safety Properties Using Induction and a SAT-Solver”. In: *FMCAD’00*. 2000, pp. 127–144 (cit. on pp. 90, 96).

[123] B. I. Silva and B. H. Krogh. “Formal verification of hybrid systems using CheckMate: a case study”. In: *Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No.00CH36334)*. Vol. 3. June 2000, 1679–1683 vol.3 (cit. on pp. 4, 30).

[124] *Simulation and Model-Based Design*. 2020 (cit. on p. 100).

[125] Neeraj Kumar Singh et al. “Stateflow to Tabular Expressions”. In: *Proceedings of the Sixth International Symposium on Information and Communication Technology, Hue City, Vietnam, December 3-4, 2015*. Ed. by Huynh Quyet Thang et al. ACM, 2015, pp. 312–319 (cit. on p. 30).

[126] Irfan Sljivo et al. “Tool-Supported Safety-Relevant Component Reuse: From Specification to Argumentation”. In: *Reliable Software Technologies – Ada-Europe 2018*. Springer, 2018, pp. 19–33 (cit. on p. 128).

[127] Emmanouela Stachtiari et al. “Early validation of system requirements and design through correctness-by-construction”. In: *Journal of Systems and Software* 145 (2018), pp. 52–78 (cit. on p. 117).

[128] Diomidis H Stamatis. *Failure mode and effect analysis: FMEA from theory to execution*. Quality Press, 2003 (cit. on p. 122).

[129] Esterel Technologies. *SCADE* (cit. on p. 31).

[130] Ashish Tiwari. *Formal semantics and analysis methods for Simulink Stateflow models*. Technical report. `http://www.csl.sri.com/users/tiwari/html/stateflow.html`. SRI International, 2002 (cit. on p. 31).

[131] Stavros Tripakis et al. “Translating discrete-time simulink to lustre”. In: *ACM Trans. Embedded Comput. Syst.* 4.4 (2005), pp. 779–818 (cit. on pp. 4, 30, 42, 48, 78, 85).

[132] Timothy Wang et al. “Formal Analysis of Robustness at Model and Code Level”. In: *19th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC’16, Vienna, Austria, April 12-14, 2016*. 2016, pp. 125–134 (cit. on p. 133).

[133] Timothy Wang et al. “From Design to Implementation: An Automated, Credible Autocoding Chain for Control Systems”. English. In: *Advances in Control System Technology for Aerospace Applications*. Ed. by Eric Feron. Vol. 460. Lecture Notes in Control and Information Sciences. Springer Berlin Heidelberg, 2016, pp. 137–180 (cit. on p. 133).

[134] Martin Westhead, Simin Nadjm-tehrani, and Forrest Hill Edinburgh U. K. “Verification of Embedded Systems Using Synchronous Observers”. In: *In Proceedings of the 4th International Conference on Formal Techniques in Real-time and Fault-tolerant Systems, LNCS 1135*. Springer Verlag, 1996, pp. 405–419 (cit. on p. 32).

[135] M. W. Whalen et al. “Your What Is My How: Iteration and Hierarchy in System Design”. In: *IEEE Software* 30.2 (Mar. 2013), pp. 54–60 (cit. on pp. 5, 31).

[136] Michael Whalen. *A Parametric Structural Operational Semantics for Stateflow, UML Statecharts, and Rhapsody*. Technical report. `https://www.umsec.umn.edu/sites/www.umsec.umn.edu/files/Parametric%20SOS%202_1.pdf`. UMSEC, 2010 (cit. on pp. 31, 56).

[137] Yixiao Yang et al. “Verifying Simulink/Stateflow model: timed automata approach”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. 2016, pp. 852–857 (cit. on p. 30).

[138] Xi Zheng et al. “Perceptions on the state of the art in verification and validation in cyber-physical systems”. In: *IEEE Systems Journal* 11.4 (2015), pp. 2614–2627 (cit. on p. 100).

[139] Changyan Zhou and Ratnesh Kumar. “Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata”. In: *Discrete Event Dynamic Systems* 22.2 (June 2012), pp. 223–247 (cit. on pp. 4, 19–23, 30).

[140] Liang Zou et al. “Formal Verification of Simulink/Stateflow Diagrams”. In: *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015*. Ed. by Bernd Finkbeiner, Geguang Pu, and Lijun Zhang. Vol. 9364. Lecture Notes in Computer Science. Springer, 2015, pp. 464–481 (cit. on p. 30).