Automated verification and synthesis of distributed systems : in particular applied to SDN-IoT platform

Abdul Majith Noordheen

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *« l'informatique »*

Par

« Abdul Majith NOORDHEEN »

« Automated Verification and Synthesis of Distributed Systems »

« Applied to *SDN* and *SDN-based IoT* platform »

Thèse présentée et soutenue à Rennes, le 7 Juin 2022
Unité de recherche : « SUMO Team, Inria, Bretagne »

Composition du Jury :

Présidente :	Sophie PINCHINAT	Professeure, Université de Rennes
Rapporteurs :	Thi-Mai-Trang NGUYEN	Maitresse de conférence, HDR, LIP6, Sorbonne Université
	Gwen SALAUN	Professeur, CONVECS, LIG, Université de Grenoble
Examinateurs :	Béatrice BÉRARD	Professeure, LIP6, Sorbonne Université
	Stefan Haar	DR Inria, INRIA, LSV, ENS Paris-Saclay,
Dir. de thèse :	Hervé MARCHAND	Chargé de Recherche, HDR, INRIA, Rennes
Co-dir. de thèse :	Dinh Thai BUI	Nokia Bell Labs France
Co-dir. de thèse :	Ocan SANKUR	Chargé de Recherche, CNRS, Université de Rennes

RÉSUMÉ

1 - Introduction

Notre principale motivation pour ce travail est l'automatisation de la mise à jour cohérente et l'intégration incrémentale de nouvelles fonctionnalités pour des applications distribuées. Un système distribué est une collection d'entités communicantes qui sont en général complexes. Chaque entité est unique en termes de comportement, de type de matériel, de procédure intégrée et de contraintes locales. La collection de ces diverses entités doit satisfaire des exigences globales tout en respectant leurs contraintes locales. Concevoir, maintenir et mettre à jour des fonctionnalités supplémentaires dans l'application du système distribué sans erreur est un défi en lui-même. Dans le cadre de ce document, nous utilisons des schémas de vérification formelle pour vérifier que le système distribué satisfait aux exigences globales. Afin d'effectuer la vérification formelle d'un système distribué par rapport aux exigences globales, nous modélisons l'ensemble d'entités communicantes comme un système de transitions distribué. Nous utilisons la composition d'automates et de systèmes de transition symbolique communicants pour modéliser le système distribué donné. Les exigences globales du système distribué sont modélisés par l'intermédiaire de logique temporelle linéaire (*LTL*). Dans le but d'intégrer de nouvelles fonctionnalités dans l'application distribuée de manière incrémentale, nous utilisons des techniques de synthèse modulaire et compositionnelle. Nous modélisons les fonctionnalités correspondantes dans un modèle abstrait et ses exigences comme une spécification de sécurité. Basé sur le modèle distribué, en utilisant la technique de synthèse réactive, nous générons des contrôleurs de supervision pour ce nouveau modèle. Le contrôleur de supervision produit contrôle ce nouveau modèle, de sorte que celui-ci soit cohérent et satisfasse les spécifications globales. En général, la génération du contrôleur de synthèse pour les modèles distribués donnés est coûteuse en termes d'exigences de mémoire et de temps de calcul. Afin de générer le contrôleur en temps réel et d'ouvrir la possibilité d'intégrer de nouvelles fonctionnalités de manière incrémentale, nous utilisons des techniques de synthèse compositionnelle et modulaire.

Nous sommes particulièrement intéressés par l'application des techniques de vérification formelle et de synthèse vis-a vis du réseau *Software Design Design* (*SDN*) qui coordonne l'application de l'internet des objets (*IoT*). L'application *SDN-IoT* est un système distribué qui est bien adapté pour se fonder sur la synthèse et la vérification formelle. Les applications *IoT* ont un important impact dans divers domaines comme l'industrie, les appareils ménagers, la télésurveillance, etc. Les systèmes distribués permettent un mécanisme de contrôle économique et flexible permettant l'application de mise en réseau permettant de coordonner diverses applica-

tions *IoT* dans une plate-forme commune.

Pour montrer l'utilité de la synthèse de contrôle réactif et générer automatiquement les contrôleurs de supervision pour les différentes entités dans l'application *SDN-IoT*, nous prenons le comportement abstrait du véhicule pour tout ce qui est application et spécification *V2X*, et le modélisons dans l'outil de synthèse réactif *BZR*. Nous avons généré un contrôleur de supervision pour le modèle *V2X* ainsi que diverses spécifications de sécurité. En utilisant le même outil, nous avons généré le code *C* du contrôleur à partir du modèle et des spécifications correspondants. Cette façon de modéliser et de générer le code implémentable ouvre la voie à l'automatisation de l'intégration de nouvelles fonctionnalités à partir de comportements de fonctionnalités abstraites d'une manière cohérente.

2 - Vérification et contrôle de systèmes distribués

Les applications des systèmes distribués sont omniprésentes dans notre monde moderne. Prenons l'exemple des rovers martiens : depuis la terre, on peut commander un dispositif d'actionnement sur une autre planète et recueillir diverses données scientifiques. Ces boucles de commande et d'actions sont rendues possibles par la communication de divers dispositifs informatiques, capteurs et actionneurs. Un système distribué est un mot-clé vague et ambigu utilisé dans toute la littérature informatique. Lorsque nous mentionnons un système distribué (*DS*), nous voulons surtout dire ce qui suit :

Systèmes Distribués

Un système distribué est une coordination de plusieurs éléments informatiques autonomes qui apparaissent à ses utilisateurs comme un seul système cohérent. Chaque élément informatique autonome est capable de se comporter de manière indépendante et peut être un dispositif matériel ou un processus logiciel. Du point de vue de l'application, cette coordination de multiples éléments autonomes peut être considérée comme un système unique.

2.1 - Modèle de systèmes distribués

Dans un *DS* donné, deux entités quelconques communiquent via des files d'attente synchrones ou asynchrones. Ces files d'attente synchrones sont des canaux de réseaux locaux à haut débit qui transfèrent efficacement le message instantanément à l'autre, ce qui nécessite souvent un canal de connexion dédié entre la paire d'entités informatiques dans le *DS*. En revanche, les files d'attente asynchrones ne sont pas à haut débit et ces types de files d'attente sont le plus souvent du type *FIFO*, de sorte qu'une seule entité informatique centralisée peut répondre aux demandes

de nombreux utilisateurs en disposant d'un seul canal de file d'attente, ce qui constitue une solution rentable dans les services de réseaux typiques [ST16]. En théorie, les communications entre n'importe quelle paire d'entités dans le *DS* seront soit synchrones soit asynchrones.

Dans ce document, nous considérons l'aspect synchrones et asynchrone. Nous utilisons la composition d'automates et de modèles de systèmes de transitions symboliques pour modéliser les divers composants. Pour les files d'attente asynchrones *FIFO*, nous avons défini le modèle de systèmes de transitions symboliques communicants avec des canaux *FIFO*. Nous considérons les systèmes composés de plusieurs systèmes qui communiquent de manière asynchrone au moyen de canaux FIFO supposés non limités et fiables (c'est-à-dire que nous ne sommes pas intéressés par le temps exact que prend un événement pour être transmis mais plutôt par le fait qu'il se produise) [Kal+14][PP91][MW08][LJJ06]. Le fait de choisir des canaux FIFO est motivé par le fait que nous nous intéressons aux systèmes distribués, dans lesquels les actions sont transmises d'un composant à un autre. Nous supposons que chaque composant est modélisé

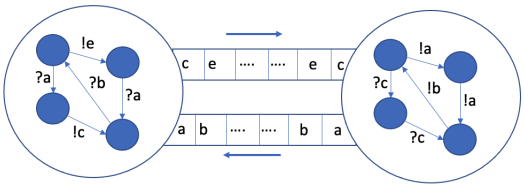


FIGURE 1 : Exemple de systèmes distribués communicant via des canaux FIFO

par un système de transitions symbolique. Ainsi, chaque composant du système est un système de transition avec des variables, dont le domaine peut être infini, et est composé de transitions symboliques. Chaque transition possède une garde sur les variables du système et une fonction de mise à jour qui indique l'évolution des variables lorsque la transition est déclenchée. De plus, les transitions sont étiquetées avec des symboles pris dans un alphabet fini. Ce modèle permet la représentation compacte de systèmes infinis lorsque la modélisation du système distribué et les variables d'exigences prennent leurs valeurs dans un domaine infini.

2.2 - Vérification de systèmes distribués

Dans le cadre général des systèmes distribués, pour analyser la correction du système vis-à-vis des exigences du système, on peut utiliser un schéma de vérification formelle, c'est-à-dire le principe du model checking pour les applications critiques de sécurité. Le principe de vérification de modèle peut être vu comme suit :

Model-checking

Pour un ensemble donné de spécifications : si le système satisfait les spécifications, alors il produit le résultat disant que le système est correct par rapport aux spécifications données. Si ce n'est pas le cas, la vérification notifie l'utilisateur avec un *contre exemple* : une trace de la séquence d'actions effectuée dans le système qui viole la spécification.

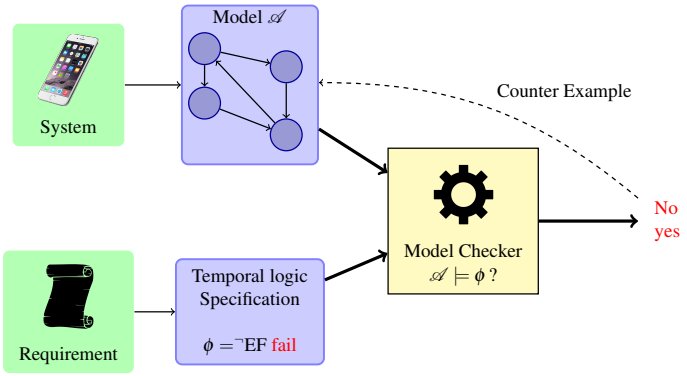


FIGURE 2 : Vérification formelle du système

L'expression des spécifications est importante dans le processus de model checking, pour exprimer les exigences du système. Il existe de nombreuses façons d'exprimer celles-ci ; l'une d'entre elles est la logique temporelle linéaire (LTL) [Pnu77], un langage qui convient bien aux systèmes basés sur l'état (comme les automates).

Comme nous considérons les systèmes distribués, nous devons trouver un moyen de faire face à l'explosion de l'espace d'état due à la composition parallèle des différents composants des systèmes distribués. Le raisonnement compositionnel apparaît naturellement lors de la conception d'un système distribué, puisque le concepteur conçoit souvent des spécifications pour chaque composant du système sur lequel des exigences globales doivent être vérifiées. Le raisonnement compositionnel nous permet de vérifier les propriétés de chaque sous-système séparément, et de combiner ces règles pour déduire les propriétés du système global.

Étant donné un automate $\mathscr{A}$ et des formules de sécurité *LTL* ϕ, ψ, nous désignons par $\langle\phi\rangle\mathscr{A}\langle\psi\rangle$ un triple tel que ϕ représente l'*hypothèse* que l'on peut faire sur l'environnement de $\mathscr{A}$, tandis que ψ représente la *garantie* que $\mathscr{A}$ fournit sous l'hypothèse que l'environnement satisfait ϕ.

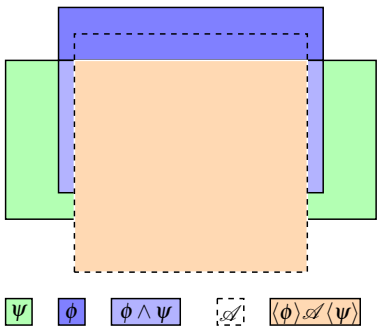


FIGURE 3 : $\langle\phi_1\rangle\mathscr{A}\langle\phi_2\rangle$ illustration

En résumé et sans donner de conditions sur les différents composants et spécifications, ce que nous voulons prouver est le suivant :

$$\frac{\langle \phi_{as} \rangle \mathscr{A}_1 \langle \phi_I \rangle \qquad \langle \phi_I \rangle \mathscr{A}_2 \langle \phi_{guar} \rangle}{\langle \phi_{as} \rangle \mathscr{A}_1 \| \mathscr{A}_2 \langle \phi_{guar} \rangle} \tag{1}$$

l'équation (1) signifie que, si, sous l'hypothèse que $\mathscr{A}_1 \models \phi_{as}$, $\mathscr{A}_1 \models \phi_I$ et de même pour $\mathscr{A}_2$ par rapport à ϕ_I, où ϕ_I est une spécification intermédiaire et ϕ_{guar}, alors il entraîne que, si, sous l'hypothèse que le système composé $\mathscr{A}_1 \| \mathscr{A}_2 \models \phi_{as}$, alors ce système vérifie ϕ_{guar}.

Dans le document, nous avons considéré ce cadre de deux points de vue

- techniques de raisonnement compositionnel basées sur le langage
- techniques de raisonnement compositionnel basées sur des spécifications *LTL*

2.3 - Contrôle de systèmes distribués

Dans le processus d'automatisation visant à faire fonctionner correctement le modèle de système en cas de violation de la spécification du système, on peut utiliser la technique de synthèse de contrôle réactif, qui repose sur le principe du contrôle des actions contrôlables du système de manière à ce que le comportement du système modélisé satisfasse la spécification du système en désactivant les actions spécifiques à certains instants spécifiques. Un schéma de synthèse abstrait est le suivant :

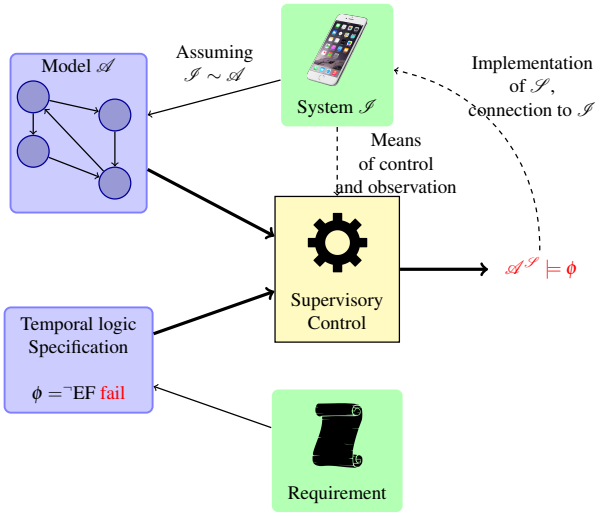


FIGURE 4 : Principe de la synthèse de contrôleur

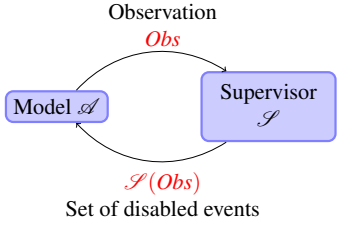


FIGURE 5 : Contrôle par boucle fermée

Synthèse de Contrôleur

Cette théorie permet l'utilisation de méthodes constructives assurant, a priori, et au moyen du contrôle, les propriétés requises sur le comportement d'un système. Étant donné un modèle $\mathscr{A}$ d'un système (qui est censé représenter correctement le comportement d'une implémentation) et une *spécification* ϕ (c'est-à-dire des exigences), un contrôleur $\mathscr{S}$ doit être dérivé par divers moyens de sorte que le comportement résultant du système en boucle fermée réponde aux exigences.

Dans ce document, nous avons d'abord étendu la théorie du contrôle de supervision pour des spécifications *LTL*, puis nous avons considéré le contrôle modulaire ainsi que le raisonnement compositionnel pour la théorie du contrôle décrite comme suit :

Étant donné deux automates $\mathscr{A}_1$ et $\mathscr{A}_2$ et deux spécifications de sécurité *LTL* ϕ et ψ, alors

- **Contrôle Modulaire**
$$\frac{(\mathscr{A}_1^{\phi})^{\mathscr{S}_\phi} \models \phi \qquad (\mathscr{A}_2^{\psi})^{\mathscr{S}_\psi} \models \psi}{((\mathscr{A}_1^{\phi})^{\mathscr{S}_\phi} \| (\mathscr{A}_2^{\psi})^{\mathscr{S}_\psi}) \models \phi \wedge \psi}$$

- **Raisonnement compositionnel**
$$\frac{\langle\phi_1\rangle((\mathscr{A}_1)^{\mathscr{S}_{\phi_1\Rightarrow\phi_2}})\langle\phi_2\rangle \qquad \langle\phi_2\rangle((\mathscr{A}_2)^{\mathscr{S}_{\phi_2\Rightarrow\phi_3}})\langle\phi_3\rangle}{\langle\phi_1\rangle((\mathscr{A}_1)^{\mathscr{S}_{\phi_1\Rightarrow\phi_2}})\|((\mathscr{A}_2)^{\mathscr{S}_{\phi_2\Rightarrow\phi_3}})\langle\phi_3\rangle}$$

Les deux techniques de contrôle ci-dessus ouvrent la voie à l'intégration de nouvelles fonctionnalités distribuées de manière incrémentale et cohérente.

3 - Software Defined Networks.

Dans ce manuscrit, nous nous intéressons particulièrement aux *software defined networks* contrôlant la coordination des applications de l'internet des objets. Nous avons appliqué le schéma de vérification et de synthèse formelles à cette plateforme *SDN-IoT*. Les méthodes de la vérification formelle et de la synthèse des systèmes distribués que nous fournissons ici ne soient pas limités à la plate-forme *SDN-IoT*, ils peuvent être appliqués à des systèmes distribués génériques.

SDN fournit une manière flexible de mettre en réseau les applications. Grâce à l'indépendance du contrôleur par rapport au réseau de traitement des données, l'entité de contrôle peut mettre en œuvre des algorithmes complexes propriétaires. Il fonctionne sur le principe de l'abstraction des applications, des services réseau. Elle contrôle les événements du réseau et planifie

les ressources du réseau à partir d'une vue abstraite et d'informations sur les réseaux de données à grande échelle et les utilisateurs du réseau. Une telle vue abstraite du principe de contrôle du réseau permet aux contrôleurs *SDN* de coordonner les divers utilisateurs et applications dans une plate-forme commune.

Un tel mécanisme de contrôle abstrait de *SDN* permet d'orchestrer diverses applications *IoT*. Les dispositifs *IoT* sont des systèmes intégrés qui sont intrinsèquement diversifiés et utilisés de manière intensive dans les applications industrielles, les appareils domestiques intelligents et les applications à distance à des fins diverses.

La figure 6 illustre l'architecture typique d'un *SDN* orchestrant les applications *IoT*. Le

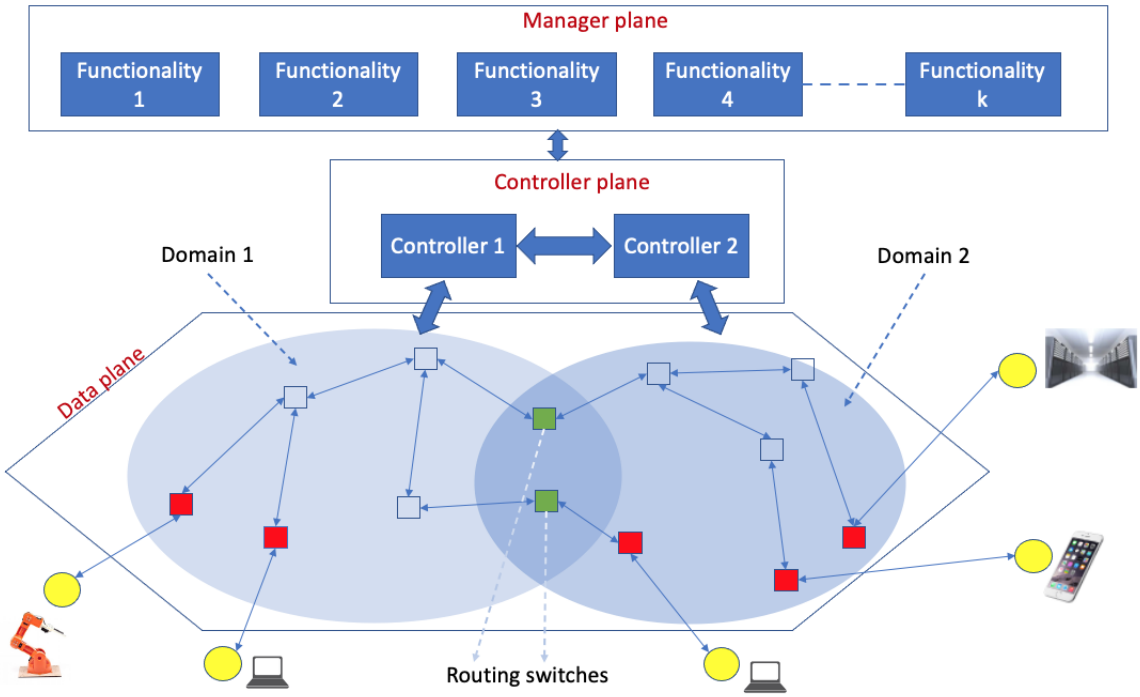


FIGURE 6 : Exemple d'une architecture *SDN*

data plane représente la couche contenant les *Open Flow Virtual* commutateurs, les clients agissant comme des sources et des terminaux. Les clients peuvent être un serveur, un ordinateur portable, un ordinateur personnel ou tout autre dispositif *IoT* (capteurs, actionneurs, etc.). Le gestionnaire d'applications réseau héberge un certain nombre de politiques réseau et *IoT* et communique avec le plan de contrôle *SDN* par le biais d'une interface de programmation d'applications ouverte, l'interface concernant l'état du réseau et ses exigences. Les contrôleurs, situés dans le data plane, dictent des règles d'acheminement aux éléments d'acheminement des données (commutateurs *Open Flow Virtual*) par le biais d'une *API*. Les fonctionnalités typiques de la couche du plan de gestion (ou couche du gestionnaire d'application) consistent à définir divers accès de contrôle du réseau, un planificateur de ressources, l'établissement de chemins de données sûrs et sécurisés entre diverses politiques de dispositifs authentifiés. Elle maintient également des informations de niveau abstrait sur les dispositifs *IoT* et leurs diverses exigences

d'application. En utilisant ces informations abstraites, il affine les politiques de réseau de niveau fin. En outre, en utilisant les politiques de réseau de niveau fin, il donne des instructions au plan de contrôle du SDN pour mettre en œuvre les politiques dans le plan de données et les utilisateurs du réseau.

Dans un tel contexte, nous utilisons des méthodes de vérification formelle pour assurer la sécurité d'un environnement *SDN*. En particulier, nous prenons le protocole spécifique *SDN* conçu par Nokia-Bell labs. L'une des principales difficultés était de comprendre ce protocole particulier et de le modéliser de manière formelle, afin de pouvoir vérifier l'exactitude du système par rapport aux spécifications de sécurité du système. Pour vérifier l'exactitude, nous avons choisi de modéliser le protocole dans l'outil de model checking *SPIN*. Lorsque les appareils connectés aux réseaux ont une position fixe, nous avons pu prouver que certaines propriétés fondamentales sont remplies par le protocole Nokia. Pour un cas dynamique (c'est-à-dire lorsque les appareils peuvent se déplacer d'une partie à l'autre ou quitter et revenir au réseau), nous avons constaté que le protocole viole l'objectif requis (exactement une propriété de confidentialité des données). Nous avons alors proposé avec l'aide de Nokia une solution pour éviter un tel scénario. Nous avons ajouté une telle solution dans notre modèle et testé le modèle final en vérifiant qu'il répondait bien à la spécification requise. Pendant la phase de vérification, nous avons également rencontré une explosion de l'espace d'état dans notre processus de vérification du modèle *SDN*. Pour éviter l'énorme espace d'état dans la vérification, nous avons utilisé le raisonnement par composition. Nous avons écrit un article sur la modélisation d'un tel protocole *SDN* et sa vérification automatisée, intitulé "Compositional model checking of an SDN platform", publié lors de la conférence Design of Reliable Communication Network,2021 Milan, Italie.

A partir de ce travail, nous sommes passés à la synthèse de commande réactive pour la génération automatisée de contrôleurs. L'étude des règles de raisonnement compositionnel dans notre dernier travail nous a aidé à proposer une "technique de synthèse compositionnelle" pour réduire le problème d'explosion de l'espace d'état dans la synthèse de contrôle discret. Nous avons décrit un cadre détaillé et une preuve de correction concernant la synthèse compositionnelle des systèmes distribués. Cette technique de synthèse compositionnelle ainsi que les techniques modulaires peuvent être utilisées pour trouver des superviseurs pour chaque entité locale d'un système distribué de manière efficace et en temps réel. Afin de montrer l'utilité de l'application de la synthèse, nous avons modélisé le comportement abstrait de *V2X*. En utilisant la synthèse de contrôle avec une approche compositionnelle et modulaire, nous avons généré le code implémentable de bas niveau de ces contrôleurs en utilisant un outil de contrôleur réactif synchronisé *BZR* comprenant l'outil *Heptagon* pour exprimer le modèle du système et *ReaX* pour automatiser la technique de synthèse afin de générer le contrôleur réactif.

TABLE OF CONTENTS

CHAPITRE 1

INTRODUCTION

This document is a result of collaborative work between Nokia Bell Labs, Paris-Saclay IoT-Control network team and Inria, Rennes Sumo team within the *ADR Sapiens* project. The main theme of this project is the automation of an Software-Defined Network (*SDN*) based architecture which serves as support for an Internet of Things (*IoT*) platform. This automation could be applied to the creation of the user interfaces to specific user's application in the platform or to the integration of new functionalities into the platform in real-time. In a more generic view, we aim to create high level models for various and useful functionalities, providing the tools and methods to convert the model into low-level machine code and ensuring the consistent integration of functionalities to the existing platform. This document is about the exploration of theoretical and implementable possibilities of the above themes. In particular, we are using automated model checking to verify that the system is working correctly with respect to the various functionalities requirements (process of verification) and providing the working methodology for introducing high level functionalities model and integrating the same to the existing real time platform by means of synthesis controller.

Readers Digestion : In this introduction chapter, we will introduce various terminologies related to *SDN*, model-checking and synthesis without much explanation, definitions or references. We choose to do this way so that it does not complicate the readers and gets involved into the specific details. It will also help to understand the overall contents with little details but the main theme of this thesis. We do this with the consideration of many papers published in the literature of distributed systems, *SDN*, *IoT*, model checking and control synthesis. These fields are wide and also have many subtle variations in the definitions, design, procedures and so on. We strongly believe that if one thinks of the interesting subject in a more abstract way will help to bring the core of the problem. The advantage of thinking about the core problem in an abstract way is it allows us to choose the well aligned modeling framework and propose the solution in a simplistic yet the best implementable procedures or designs. This introduction chapter will slowly introduce various concepts of model checking, synthesis, distributed system concepts in this chapter and the following chapters. Please continue to read this chapter with common sense rather than expecting formal definitions for each notion or notation. We will progressively introduce each notion formally and give comprehensive comments with examples in the following chapters.

Brief introduction about the working teams : *Sumo* team is a research team that mainly works

on theoretical studies in the fields of Model-Checking, Synthesis of Discrete Event System, Game Theory, Fault Tolerant and Diagnosis, Markov process/chain etc. Nokia, Paris-Saclay IoT Control network team works on 5G network, *IoT* system, *SDN* etc and concentrate on design, testing and implementation.

1.1 Road Map To Thesis

In this ADR project, we were interested in the automated procedure of *Verification* and *Synthesis* of *Distributed Systems*, and we focused on the Software defined networks (*SDN*) application to the Internet of things (*IoT*) platform. *Software Defined Networks* is an example of such a Distributed system. The modeling and technique we used in this document is not particular for *SDN* or *IoT*, it can be used for any generic distributed systems. In the path of our work, we also tackled the *state space explosion*, which appears to be a severe problem in the verification and synthesis process. We provide our contribution from the theoretical side to avoid this *state space explosion* problem. In order to overcome this *state space explosion*, we revisited the composition technique (assume-guarantee technique) and generalized little but enough to use for postponing the state space explosion in the process of *SDN* verification and control synthesis. Based on our knowledge, *State space explosion problem* can't be avoided completely, can only be postponed.

In chapter 2, we will go through the overall working nature of *SDN* system, *IoT* platform and we will also mention short notes about Nokia,Nozay *SDN-IoT* platform design.

In chapter 3, we will give a generic modeling scheme for the distributed system to do verification and discrete control synthesis. Most of them are from literature but some of them are further improved by the author and advisors. Main improvements are in the direction of synthesis of safety *LTL* specifications (temporal properties about the system) for the automaton model of monolithic and distributed systems.

In chapter 4, we will state the important problem of verification and synthesis related to 'State Space Explosion' and we go through some of the techniques from the literature and authors contribution in this aspect to reduce the state space to limit the state space explosion effects on the experimentation of model checking and discrete control synthesis procedure. In the same chapter, we developed compositional control synthesis of *LTL* safety specifications.

In chapter 5, we model the specific Nokia-SDN platform as an automaton and express the system requirements as *LTL* specifications and do the verification experiments using the model checking tool *SPIN*.

In chapter 6, in the direction of producing the discrete controller for the expressed model and specification of the *SDN-IoT* platform as a demonstration, we state some of the specific functionalities of *SDN-IoT* platform and model them in high level language called *Heptagon* (a reactive modeling of distributed system) and produce the control synthesis using *BZR-ReaX* and translate them into classical high level languages like *C* and *Java* and provide the possibility of

integrating the produced control model with real time systems.

1.2 Distributed Systems

A Distributed system [ST16] is a vague and ambiguous keyword used throughout the computer science literature. When we mention distributed system (*DS*), we mostly mean the following :

Distributed systems

A distributed system is a coordination of multiple autonomous computing elements that appear to its users as a single coherent system. Each autonomous computing element is able to behave independently and can be either hardware devices or software processes. From the application side, these coordination of multiple autonomous elements can be viewed as a single system.

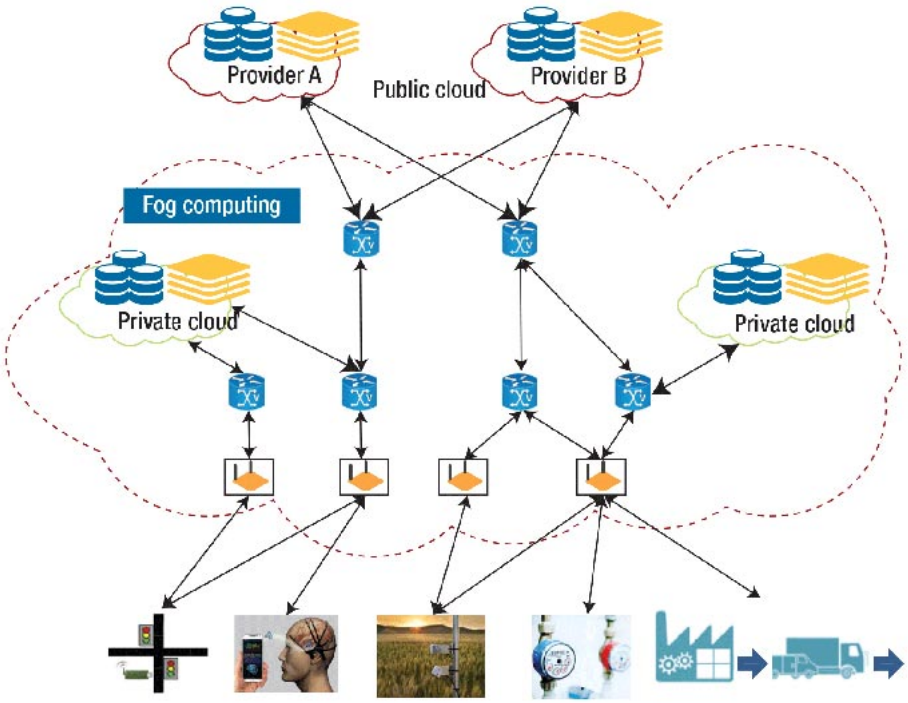


FIGURE 1.1 : A Typical Distributed System (*DS*) : Image downloaded from https ://www.icar.cnr.it/en/sistemi-distribuiti-e-internet-delle-cose/

There are two main characteristics that can be referred to from the above definition.

1 It is a collection of computing elements able to behave independently of each other.

2 It is believed as a single system by the users (being people or applications).

Autonomous computing entities of the distributed system might be either a super computer, a personal computer or smaller computing devices or even a software process. This of course includes server farms, supercomputers, sensor networks, organizational Intranet, and the Internet, but also our laptop in which many components have some independence. In principle, they are acting independently from each other and their aims were to achieve common or separate (individual) goals. *DS* (and also distributed computing) captures a wide variety of situations which includes computers or processes and also serves as a model for numerous phenomena in the natural sciences. Clearly, this is a vast and heterogeneous field. Meanwhile, they are programmed to communicate with each other by exchanging messages to realize specific applications. How to establish this communication channel i.e collaborations between computing processors is at the heart of developing the distributed systems.

In a given *DS* any two entities communicate via synchronous or asynchronous queues. These synchronous queues are high speed Local-area networks channels which effectively transfer the message instantly to one another which often needs the dedicated connection channel between the pair of computing entities in *DS*. Where-us the asynchronous queues are not high speed and these queue types are mostly First In First Out (*FIFO*), so that a single centralized computing entity can serve many user's requests by having a single queue channel which is a cost effective solution in typical network services [ST16]. In theory, communications between any pair of entities in *DS* will fall into either synchronous or asynchronous. In an asynchronous setting, *DS* reads messages by some polling mechanism or FIFO manner ; it is specific about the architecture and protocol. In synchronous setting, reading and sending a message action based on either shared variables (common memory block access) between the processors or by means of a dedicated synchronous channel (*RDV* channel). This communication can be assumed reliable (meaning the message sent by some processor never fails to reach the receiver). In case of asynchronous *FIFO* communication, one can assume the queue length is bounded (in the real system this is the case, but in the theoretical model one typically assumes unbounded queue length). However, When dealing with *DS*, one can not assume the existence of a global clock in all the entities in the considered distributed system. This leads to the fundamental problem of synchronization and coordination of common actions and progress in the computation process within a distributed system. The information one computing entity expects from another computing entity may be delayed in an unbounded way, so there is no way to guarantee that such a computing entity is simply slow, completely broken, or maliciously refusing to cooperate at this point. Even when all messages from other computing entities eventually arrive, message arrival order may be arbitrary. Note that asynchronous is a real headache : it can arise from computational differences between the computing entities, properties of the underlying communication channel between them or other reasons, and hence must be dealt with care. This issue is so central that even this sub-field of asynchronous distributed systems is large [Lyn96 ; AW04 ; Wig17].

When a distributed system is designed with a set of coordinated computing elements that have to meet certain requirements, the goals of a *DS* we mention here after as either *network wide-invariant* or *specifications* in the rest of the document. In fact, it is the designer's responsibility to design the communications interface between each computing element as well as the behavior of each computing element of the *DS* to meet the required *specifications*. It is important to ensure the correct behavior of the designed *DS* to meet required *specifications*. Since in our Nokia-Inria (*ADR*) project, we are interested in automating techniques, we are using Verification and Synthesis mechanisms to check and correct the designed *DS* for the given *specifications*.

There are some well known reasons in the literature as mentioned in the survey [ST16], why the design of *DS* fails to meet the required specifications, even-though it is designed by experienced designers. To name some of them : assuming that the network communication between the different entities is reliable and secure with respect to a given architecture, i.e the connection of network among the entities is stable with zero latency together with an unbounded bandwidth, then there is only one controller for each autonomous computing entity, and its computing speed are same of the given *DS* components.

1.3 Challenges

In the automated verification and control synthesis of *DS* systems, there are a couple of challenges which depend on the kind of distributed system one is considering, since distributed systems itself is a vast and highly diverse field. One of the main problems of verification and control synthesis of a given distributed system is *State Space Explosion*, a problem that arises when one tries to trace all possible execution paths of given *DS*. In our case, a *DS* contains too many components, and/or the communications between the components of *DS* are asynchronous i.e communicating via *FIFO* queues which makes the formal analysis very difficult to complete the process. There are some techniques to avoid this brute force analysis to name few : *partial order* technique [BK08], *assume guarantee* technique [HQR98 ; Sta85], *abstraction* [Cla+18] of the system in case of verification process. For the synthesis purpose, one can use *modular* nature of property and the system, a well known technique to reduce *state space explosion* in the literature [WR88]. We will provide more details of these techniques in the following chapters. Our main contribution to alleviate the state space explosion in this document is the *compositional reasoning* for verification and control synthesis for *DS*. This compositional reasoning technique is inspired from the rely-guarantee technique introduced by [Sta85]. Apart from this theoretical result, the main works towards the automation apart from the mathematical framework (or definition) involves in this thesis as follows :

1 Properly understanding the design *SDN-IoT* of Nokia, Nozay team, so that it can be formally modeled to proceed with model checking and control synthesis schemes.

2 Identifying the set of requirements and expressing them as a specification so that we can test the model with respect to the specification.

3 Providing the use case in the direction of proposing the high level model description and producing the control synthesis for the proposed model design automatically and converting them into classical high level language so that one can integrate the produced high level codes with real working *SDN* system.

Digestion of Proposed Automated Procedure : First and foremost import things to stress that what we are doing is automated verification and synthesis, and every automatic scheme where there is a hidden automated procedure. The usefulness of an automatic scheme purely depends on robustness and adaptability of the written automated procedure. If one wants to maintain a distributed system that works without errors, then it will hugely depends on one's knowledge and envision capability about the kind of environment the proposed system or design will going to work and the expected applications from the design and unsafe states can possibly occurs in the proposed design so that one can express and quantify the required properties and check the proposed design with the expressed properties in the model checking process i.e Verification of proposed system model with respect to various expected properties. In case of synthesis, one should check the limitation of control synthesis techniques and the kind of properties control synthesis solution can support to the *DS* system and making the *DS* system to work without reaching an error state (unsafe situation) and also in non-blocking way (i.e have to work and provide useful application rather than just avoiding the error states).

CHAPITRE 2

SOFTWARE-DEFINED NETWORKS AND IOT PLATFORMS

Data networks are made up of many routers, data forwarding switches, firewalls and so on. Such communication networks are complex and dynamic in nature i.e the multiple events (like mobility of users, requirement changes, traffic in the routing path) that can occur simultaneously throughout the network. The main objective of the network operators is to control the communication network and respond to a wide range of network events such as intrusions, loss of traffic in the data path and so on. The difficulties in operating the networks lies in its scale (number of users, number of network nodes). In traditional *IP* networks (both wired and wireless), the network controller and the network fabric are tightly coupled within a network node. Such a network node architecture is quite effective in terms of network performance but the outcome is a complex and *relatively static* overall architecture [Kre+15].

Such a tightly bundled control and data plane collocated within the network node often makes it difficult to operate (e.g. upgrade) the network and generate high operational cost as one has to perform the same operation on all nodes associated with the control plane. Adding a new application to such a static architecture is not to be a cost effective one, since it often requires the reconfiguration of the control plane on all the network nodes spread across the network topology. The operational cost of maintaining a network infrastructure is significant to the expenditure capabilities of network service providers, which does not provide more space for innovation.

Thus, it is preferable to have stable network fabrics and data plane, but a dynamic programmable control plane with an abstraction level to serve various applications. In such a case, an architecture that makes most sense could consist in network controllers that program the network fabrics based on network manager instructions e.g. automated system to implement the network dynamic policies. Within this architecture, the network manager plays the role of the abstraction level which interfaces with various applications and which translates high-level policies received from the latter into low-level network policies. This is part of the *Software-Defined Network* (abbreviated as *SDN*) framework.

Within such a framework, one can centralize the control software (e.g. within a data center) for ease of upgrade/update and make the network node a white box. As a result, a simple white box network node can be bought from any vendor (i.e. off the shelf) with simple data

plane functionalities that leads to a reduction in network node complexity and thus network operational cost. The programmability and the flexibility of the network relies on the control plane software which communicates with network nodes in a much standardized way via open application program interfaces (*API*).

Additionally, the advent of *IoT* technology and its applications impose important new constraints on the communication network. Indeed, the latter network also has to handle the scale related to a huge number of *IoT* devices. Roughly we have more than 15 billion *IoT* devices (like various sensors signaling, GPS system of cars, smart phones and their application, RFID, Google glass, smart home, and so on) connected to the *internet* and this number is expected to grow exponentially with more and more interactions between devices due to smart applications brought by the *IoT* technologies and platforms.

For both fixed and wireless communication networks, which have to effectively support the 'Internet of Things' (*IoT* will be introduced later in this chapter), they are expected to support a wide range of applications coming from autonomous communication among sensors and actuators as well as the interactions between users (humans) and machines, like smart cities, intelligent mobility, industry automation just to name a few. It is clear that modern networks have to face human-type and machine-type interactions. Heterogeneous characteristics of the applications (sensors, actuators, control entities) together with a large range of qualities of service are part of the set of requirements. There should be enough flexibility in the network to deal with such heterogeneous applications. In contrast with hardware-based deployments of previous generations, the design of new communication networks took the advantage of *softwarization* and *virtualization*.

Softwarization is a paradigm which runs, defines and dynamically updates the functionalities via software rather than via pre-programmed hardware. It guarantees a high degree of flexibility and reconfigurability by solely updating software code rather than adding or removing the existing hardware from the service. This leads to a cost effective solution.

Virtualization is a concept of creating abstract entities for the hardware platforms, operating systems, storage devices and data network resources that exploit the low cost virtual machines instead of dedicated hardware for various intended network applications. *Virtualization* offers different applications and services that share a pool of configurable resources in a cloud environment. A cloud environment provides the processing, storage, and networks resources to the users. The network users can deploy their applications in this cloud environment and use the underlying cloud infrastructure and services. It is noted that virtualization is an enabler for softwarization.

From the network point of view, *Software-Defined Network* is considered as a main realization of *softwarization* concept [LKR14]. Another key concept in recent network architecture is the *Network Function or NF* [LKR14],[Kre+15]. The latter is defined as "a functional block within a network infrastructure with well-defined interfaces and well-defined behavior". The

NF can be physical (i.e. implemented by hardware) or virtual (i.e. software-based). The latter could be placed and migrated across physical network infrastructure according to the needs of network operator and traffic management, providing room for making the independence of network functions from hardware i.e. a Virtual Network Function (VNF) can be decoupled from hardware and placed in different hardware locations to maintain the services without interruption. This is called the Network Function Virtualization (*NFV*) paradigm. Together with the *NFV* framework, the *SDN* framework allows to partition the operator network infrastructure into isolated logical networks of varying sizes and structures with each of them dedicated to a different type of services. This allows for multiple usages and tenants sharing the same operator network infrastructure lead to network cost optimization.

2.1 Introduction to SDN

Software-Defined Network [Kre+15] is an emerging network architecture that decouples network control function from network forwarding functions, enabling the network control to become directly programmable. In such a case, the underlying network infrastructure has to be abstracted for applications and network services.

SDN and *NFV* are complementary in the softwarization of the network. While the former decouples the network control functions from the network forwarding functions, the latter aims essentially at decoupling any network function from the hardware on which it runs. Both facilitate the network design and help the infrastructure to be virtualized and abstracted with different software building blocks. *SDN* uses *NFV* infrastructure (e.g. virtual machines) to run the network control functions so that the latter can easily be deployed on different network infrastructures. The aforementioned list of virtualization functions are intensively deployed on an *NFV* infrastructure as *SDN* network controller functions to set the data forwarding rules on the network nodes (both physical and virtual switches or routers).

SDN architecture consists generally of three building blocks : network managers (or management plane), network controllers (or control plane) and data forwarding elements (or data plane) as depicted in Figure 2.1.

Data Plane : Network Elements [Kre+15] A data plane or data forwarding plane contains a set of network elements that receives data packets on its (communication) ports and performs few simple network operations on them. For example, the network device may forward a received packet, drop it, alter the packet header (source, destination information about junk of data piece), and so on (refer the figures 2.1, 2.2). The network elements are, for example, routers which operate generally at the *Internet Protocol* i.e *IP* layer. Additional examples include network elements that may operate at a layer above *IP* such as firewalls, load balancers, and video transcoders or below *IP* such as Layer 2 switches and optical or microwave network elements.

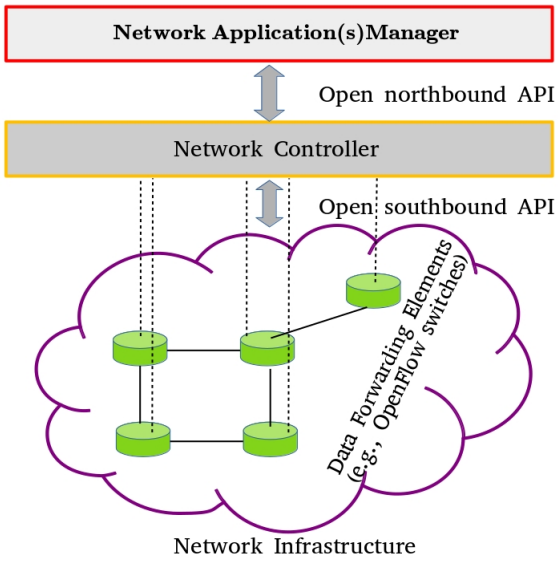


FIGURE 2.1 : SDN infrastructure with layered connections

Network elements can be implemented in hardware or software and can be either physical or virtual.

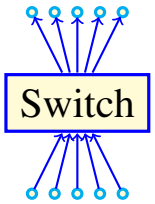


FIGURE 2.2 : Simple Network Switch

.	Packet_header,In port	Action
$rule_1$	10*** , pt_1	pt'_3 forward
$rule_2$	111** , pt_3	drop
$rule_3$	101** , pt_2	modify.packet.header
.	.	.
.	.	.
$rule_k$	01*** , pt_4	ask controller

TABLE 2.1 : Forward table of switch *S*

Control Plane [Kre+15] The control plane is a logical decision implementer of *SDN* network manager instruction to the data plane. The control plane, usually a set of control entities, is responsible for the configuration of forwarding elements in the data plane by communicating via the (Control-plane) South bound Interface (*SI*) protocol[Kre+15]. The control plane is responsible for instructing data plane elements about how to handle network (data) packets. A control plane entity, called as *SDN controller*, translates a new request from the application into a sequence of instructions that will implement new forwarding rules on network devices via Southbound interfaces [Kre+15] (an *API*'s refer the figure 2.3). Communication between control-plane entities (i.e between *SDN controllers*), referred as "east-west" interface, is usually implemented through gateway protocols such as Border Gateway Protocol (*BGP*) or protocols such as Path Computation Element (*PCE*) [Kre+15] and so on. These corresponding protocol messages are usually exchanged in-band and subsequently redirected by the forwarding plane

to the control plane for further processing. Control plane functionalities usually include : topology discovery (i.e data plane and various network users/clients), data packet route selection, path fail-over mechanisms in case of dynamic change of physical network or change in traffic. Control plane southbound interface *SI* is usually defined with following characteristics : time-critical interface that requires low latency with higher bandwidth in order to perform many operations in a short-time period, oriented towards wire efficiency (efficient machine readable and function activation code) and device representation instead of human readability. Examples include fast- and high-frequency of flow table updates, and robustness for packet handling and events. Control plane *SI* can be implemented using a protocol, an *API*, or even inter-process communication. If the control plane and the network device are not collocated, then this interface is certainly a protocol. Examples of control plane *SI* are the *ForCES* and the *OpenFlow* protocols. Control-plane service examples include a virtual private *LAN* (local area network) service, service tunnels, topology services, etc. The control plane also has a North Bound Interface *NI* via which it communicates the system state, the quality of network services and high level information about the data plane to the management plane.

The *SDN* controller algorithms are designed to catch up and implement various instructions from the *SDN* manager. Some of the main activities of an *SDN* controller are listed below :

1 Shortest Path Forwarding rule : to set the communication path between various users in the data plane by constantly monitoring the traffic scenario and the data plane network switches availability,

2 Security Mechanisms : set the routing data packet rule such that it avoids from reaching the data to malicious users,

3 Monitoring the data plane topology and users positions in the data plane,

4 Process the data plane events, intent requests and inform the same to manager,

5 and so on.

Management Plane [Kre+15] The management plane is a logical decision maker and decides the functionalities of *SDN* usage which aims to introduce constraints to the network functions and ensures that the network as a whole runs optimally by communicating with the network controllers using a Management-Plane Northbound Interface. Management-plane functionalities are typically initiated, based on an overall network view, and traditionally have been human-centric. However, artificial intelligence algorithms are lately replacing several human interventions. Management plane functionalities include fault monitoring, configurations of the data plane users and services availability in addition. Its functionalities may also include entities such as orchestrators, Virtual Network Function Managers and so on.

The typical management plane roles are listed below, but it may vary depending on the applications

1 Classify user-defined policies,

2 translate the above policies to network policies,

3 provide to the users various network virtualization functions which could change dynamically according to the requirements or based on requests from these users,

4 and many more depending on the applications.

Communications between and within SDN layers [Kre+15] In the set up of well separated layers as in *SDN* architecture, there should be a clear communication protocol defined between each pair of layers and one defined between entities within each layer. For each of these protocol frameworks, data and information models should be specified. In the literature there are various protocols that are defined to build the communication links between and within *SDN* layers - i.e. northbound, southbound, east-and-westbound interface protocols are available. For Northbound *API*'s (i.e. communication between the control plane and the management plane) we have *ad-hoc API*'s, *RESTful API*'s, etc. For Southbound *API*'s (i.e. communication between the control plane and the data plane) we have *OpenFlow*, *ForCES* and *OVSDB*, just to name a few. The East-and-Westbound (essentially between controllers when we have distributed controllers compared to centralized controllers) communication *API*'s totally depends on the kind of controllers, the details of the data abstraction used in those controllers for communication. Just to name a few examples of *SDN* controllers : *Onix*, *Beacon*, *Floodlight*, *OpenDaylight*, *ONOS* and etc. The fig. 2.3 explains the overall working nature of various *SDN* entities and interfaces.

That's the end of *SDN* introductory notes, now we move on to the introduction about *IoT* devices and its applications.

2.2 *IoT* Devices and Related Applications

Thing in the *Internet of things* or *IoT* devices that can communicate via a communication network. *IoT* is the set of physical objects like sensors, actuators, vending machine, automatic teller machine (*ATM*), etc (mentioned as things) can be primarily characterized as embedded system (smart electronic devices) that are used for the purpose of connecting and exchanging data with other (*IoT*) devices and systems via Internet (or local network services) or mobile networks. *IoT* has evolved a lot thanks to the help of the evolution of *Internet* (including wireless technologies) like *LAN*, *sub-net*, *Ad-hoc* network and mobile networks.

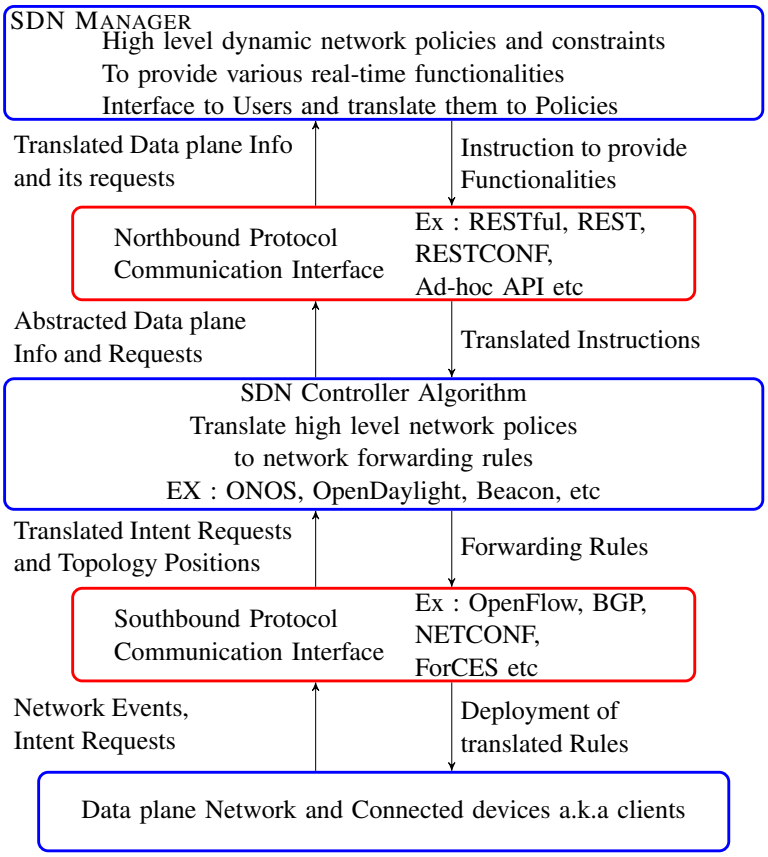


FIGURE 2.3 : SDN Entities and its Working Nature

Traditional *IoT* devices are embedded systems, wireless sensors, control systems, automation systems (home, building, car, ship, airplane, industrial automation, and so on), and others, all contribute to enable the 'Internet of things'. *IoT* technologies are sets of products pertaining to the concept of *smart devices*, including devices and appliances (such as lighting fixtures, thermostats, home security systems and cameras, and other home appliances) of a common platform, and can be controlled via devices associated with that platform, such as smartphones and smart speakers. The concept of first smart devices emerged from Carnegie Mellon university where computer science network geeks played around the coca-cola vending machine to monitor from a remote server the current availability of coke canes and its temperature by accessing sensors installed on the machine. These traditional embedded systems devices used to be high energy consuming entities due to direct point-to-point connections (i.e. long range wireless communication). With the ubiquity of the internet nowadays, one can go for low-energy consuming and short-range wireless communication devices. This is one of the benefits of modern network to *IoT* applications.

As by the European Telecommunications Standards Institute (*ETSI*) the *IoT* applications are projected for smart devices, smart cities, smart grids, the connected car, eHealth, home automation and energy management, public safety and remote industrial process control. Apart from the wide applications of *IoT*, it is important to emphasize the fact that *IoT* devices are

highly heterogeneous and not a unique type of electronic devices. They vary depending on the type of hardware they are using (Micro-controller, micro-processes), computing power, energy consumption, action-behavior response with respect to environment (in case of actuators), the data handling devices and so on. In such heterogeneous types of *IoT* devices, it is important to coordinate them to make an application secure and safe. In the sense that actuating a wrong actuator or providing open access to sensitive data handling devices can cause severe damage either physically or economically, also brings privacy related issues. These are the challenges faced (or has to) by the *IoT* applications developer for coordinating such heterogeneous devices using global controlled software in a common platform.

2.3 Orchestration of *IoT* Platform using *SDN* Concept

In the modern 'communication network' era, the communication network is highly connected. It provides *global network control* framework for discovering, deploying various *IoT* devices, applications and its interactions with humans. In such a scenario it needs smart control, actuation and automation to manage the global network control mechanism. The main challenges of global network control mechanisms is the realization of *IoT* applications which involve the interoperability among various *IoT* entities. The fact of applying *SDN* (layered network) concept to coordination (orchestration) of *IoT* devices is not new, for example the paper [Sar+14] gave brief summary about realization of *IoT* application using *SDN* framework. The reason for using such layered network architecture is mainly for quick response to the dynamic network instances using data abstraction from bottom to top layer of *SDN* network. *SDN* inter-operable architecture for *IoT*, is assumed to overcome most of the obstacles in the process of large scale expansion of *IoT*. It specifically addresses heterogeneity of *IoT* devices, and enables seamless addition of new devices across applications. Large number (approx. a billion) of *IoT* devices are going to be connected with *Internet* within a few years, which will generate enormous amounts of data and exchanges of those data among *IoT* devices and various applications. Using a global centralized (or co-ordinated) *SDN* controller paves the way to minimize the delay and load balancing in *IoT* applications network.

In a complex highly connected network operation, knowing the fine details of the network and connected devices is very difficult especially as a network user. Maybe an expert level or an experienced network user can manage the network integration but as a layman it's not as trivial. As a result, most end-users do not share their connected devices and do not benefit from *IoT* services which could result from rich associations. To benefit from some popular services, most end-users purchase connected devices for their application without being aware of all the services they could benefit from, provided their *IoT* services are associated with devices belonging to other smart environments. This situation is explained by considering human skills required to face complexity in terms of [Sar+14] :

- Awareness of *IoT* services and ability to select suitable connected objects.
- Ability to deploy and configure all functions to make expected services work. This particularly involves the configuration of the network layer to support *IoT* services involving different physical functions spanning multiple smart environments.

The scope of *SDN* to coordinate *IoT* devices can be illustrated with the following three key concepts,

1 Discovery and positioning the *IoT* resources :
SDN working nature naturally provides the functionalities of discovering various *IoT* devices and its current position in the given application network. It keeps track of the real time position of various *IoT* devices and its operation conditions and modes.

2 Allocation of *IoT* resources in an *IoT* services :
Based on the tracking of *IoT* positions, operational modes and requirements, it provides the external resources to those devices like allocating the virtual hardware to support the computation required by the *IoT* devices.

3 Dynamic grouping of *IoT* devices :
This is one major advantage of providing dynamic grouping of various *IoT* devices in the required applications, which make use of the resources efficiently to serve various functionalities in real time.

EXAMPLE 2.1 *Typical SDN architecture to orchestrate the* IoT *applications is depicted in figure 2.4. The data plane represents the layer holding the* Open Flow Virtual *switches, the clients acting as traffic sources and sinks. The clients can be a server, laptop, personal computer or any IoT devices (sensors, actuators etc.). The network applications manager hosts the number of* IoT *policies and communicates with the* SDN *control plane through open northbound Application Programming Interface. Interface regarding the status of the network and its particular requirements. The controllers, situated in the control plane, dictate forwarding rules to the data forwarding devices through open southbound* API.

The typical functionalities in the manager plane layer (or application layer) will be network control access, resource scheduler, establishing safe and secure data path between various authenticated devices. Maintaining abstract level information about IoT *devices involved in various applications. Further this abstract information can be about the performance of those devices or status about its working nature, etc.* ◇

The next section explains a particular *SDN-IoT* platform developed by Nokia, France 5G network team.

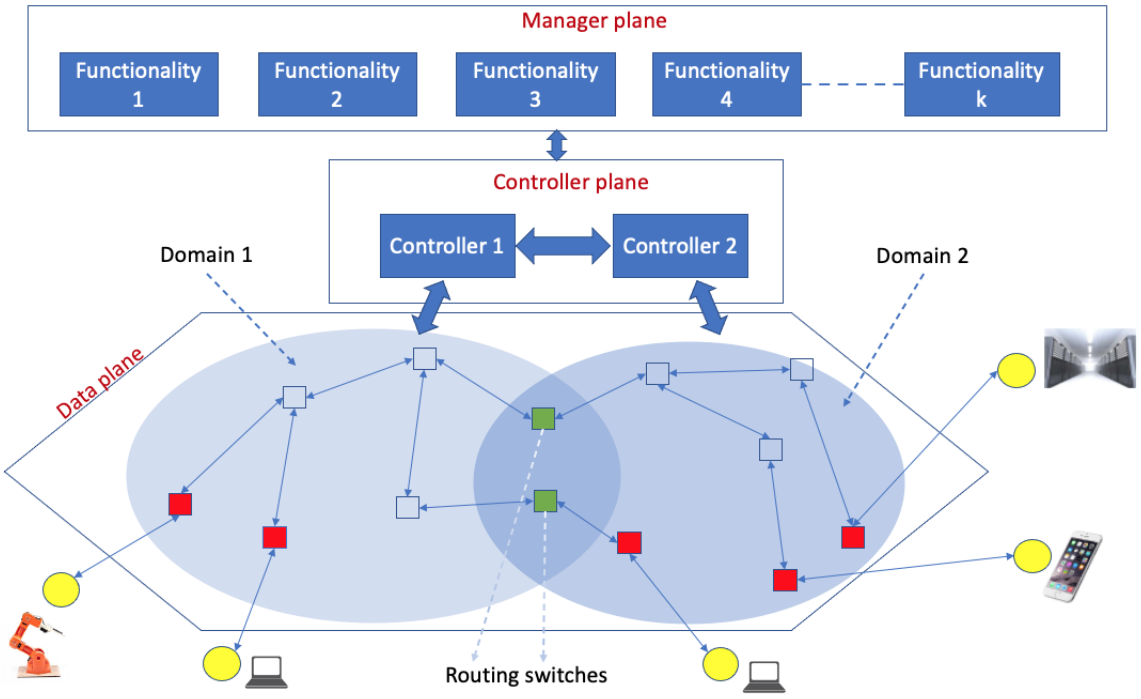


FIGURE 2.4 : Example of *SDN* architecture orchestrating the *IoT* applications

2.4 Nokia *SDN-IoT* Platform

Here, we will describe the general principle behind the Nokia *SDN* architecture. We will give a very narrow and general intention of *SDN- IoT* platform constructed by the Nokia Bell Labs team. For our modelization, verification and synthesis scheme, we only focus on a specific part of Nokia *SDN-IoT* platform ; more details will be given in the respective forthcoming chapters.

Before going into the further details of Nokia *SDN* design, we will recall here three characteristics of distributed systems that are universally desirable that we call *CAP* theorem.

1 ***C****onsistency* (information should be consistent throughout the network), in the sense that system should responds *identically* no matter which node (i.e in our case decentralized *SDN* controllers and manager) receives the request from the data plane.

2 ***A****vailability* (response to the requests) means that system (in our case *SDN* controller and manager) should responds to a query (in our case intend requests although the response may not be consistent or correct)

3 ***P****artition Tolerance* in case of communication failure of any one of the node (either switch, controller or manager) system should continue to function. Although this specific partition tolerance is not considered or analyzed in detail in the forthcoming chapters. Nonetheless, it is an important criteria to consider in the distributed environments.

In the distributed community [GL02], it was identified only any two of above three characteristics can be full-filled at any given time but not all three. It was first conjectured by Eric Brewer

later proved by Gilbert and Lynch. It is also important to note that the CAP theorem not only applies to asynchronous but also synchronous distributed systems.

Network switches, *SDN* controllers and managers all are computational entities. In case of a switch - slave computation entity which receives and forward the data packets, and receives and sends the intent request to update the forward rule table. Controller receives and processes the data plane requests and follows the instruction from the manager plane to satisfy various specifications. Whereas the manager observes the abstract information about the data plane and its requests also for the needs of application, it dynamically changes the network policies. *CAP* theorem naturally applies to the working nature of *SDN*.

In the context of the CAP theorem, if one considers partition tolerance of paramount importance, traditional control-plane operations are usually local and fast (available, and it goes to inconsistent state for a shorted period of time once in a while), while management-plane operations are usually centralized (consistent) and may be slow (can't available all the time so naturally slow in response to the requests). In this theoretical limitations, when we build such complex distributed system, it is important to analyze the *safety-critical* specifications (this is one of the reason *ADR sapiens* (who is funding my Phd thesis) project choose to do model checking of Nokia *SDN-IoT* platform) (properties or requirements) with respect to *CAP*'s *availability*, *consistency* and *partition tolerance*.

2.4.1 General Problem and Assumptions

Smart devices are simply connected logical objects interconnected with other logical objects via the service links provided by the network operators through network routers or switches. The connected entities are objects that can produce, receive, forward or process data. These connected objects can be a physical object, an application, a piece of software, etc. Such a collection of smart devices communicating with each via an established network are referred as *IoT* application. In the presence of a large number of heterogeneous smart devices to make an *IoT* application works for various requirements either Industrial automation or smart home appliance or some other applications which involves controlling the blend of virtual and physical systems together. Manually, managing such a complex system because of the sheer number and heterogeneous device types involved in the applications.In such a complex enterprise application, *SDN-IoT* platform facilitates heterogeneous behavior of *IoT* devices with a unified way of data communication protocol. They can advantageously be used in smart and complex applications scenarios to group similar devices while enforcing safe and secure *IoT* applications. Apart from the heterogeneous nature of *IoT* devices, the proposed *SDN* solution should be able to handle the sheer number of *IoT* devices involved in the application. That is *SDN* solution should be well scaled to handle latency problem, quick action responses to the *IoT* applications. In order to scale down the problem of controlling the bigger network and variety and too many

numbers of *IoT* devices involved in the application, two major concepts were introduced to control. These concepts are *Virtual Places* and *Virtual Spaces*.

Virtual Places, concept of partitioning and classifying the cyber-physical system voluntarily to respond to the controllers instructions in fast and also network manager can monitor the global system in an efficient manner. Challenges in realizing such a concept lie in the definition and implementation of the companion cloud infrastructure and network gateways, realizing a distributed test bed allowing to easily provision, deploy and dynamically extend to multiple geographical locations, and interconnect these *Virtual Places*. *Virtual Spaces* concepts are to create secured, dynamic collections of resources regrouped by usage, and interconnecting these resources over one or more Virtual Places (i.e., crossing multiple administrative domains). Realizing those Virtual Spaces requires on the one hand a mix of mechanisms to automatically adapt their content according to their definition and the current context.

It is important to note that *Virtual Places* is defined so that the *SDN* network is to control and monitor the data plane in a scalable manner. Furthermore, concept of partitioning the entire data plane into number of *Virtual Places* is not just a blind partition based on the number of network routers in each *Virtual Places* but it is depends on pre-evolution of the application requirements on each *virtual places* and targeting different kinds of user's devices. In the following texts, we interchangeably use *domain* for the *Virtual Place* because the *domain* name we used in our *SDN* model checking work earlier compared to the original name *Virtual Place* as in the paper [Bou+18].

Apart from the above assumptions, in terms of building the protocol, it has the following technical assumption.

> One of the Nokia *IoT* Platform assumptions is that devices connected to the platform communicate via at least the IEEE 802.3 MAC layer. This allows for deploying MAC learning algorithms and forwarding rules based on IEEE 802.3 MAC header - for devices not supporting IEEE 802.3 MAC protocol, they can connect to the platform via a IEEE 802.3-enabled gateway. Platform design also assumes that applications are running over *IP* which is the case of most applications nowadays.

2.4.2 Cluster of *IoT* Devices or Virtual Space

In order to enable communications among the group of devices which itself is dynamic based on the real-time intended requests from various smart devices, it will be better to keep track of clusters of devices, to maintain and update them in real time. Within the dynamic nature of intended requests, keeping high-level information about the clusters of devices by the management plane will guide to generate a high level instruction set. This set of instructions will be used by the *SDN* control plane to enforce in the data plane communication connections among the devices from the same cluster. Such a cluster of devices is called as *Virtual Spaces* or shortly

vSpaces.

EXAMPLE 2.2 *For an example, the set of* IoT *Devices* $:= \{D_a, D_b, D_c, D_d, D_e\}$ *are connected to the data plane layer Switches, let say* $\{D_a, D_b\}$, $\{D_c, D_d\}$ *and* $\{D_e\}$ *forms the clusters of devices or vSpaces i.e* vSpace_1, vSpace_2 *and* vSpace_3 *respectively. There should be communication links enabled between devices of each* vSpace*'s. The management plane only knows the unique identity number for each device and the one for each vSpace. It provides high-level instructions to the controller to enforce communication links between devices of each vSpace. The control plane sets up such communication links by enforcing forwarding rules on the data plane switches.*

As mentioned earlier, cluster of *IoT* devices i.e set of *vSpace*s are defined so that not all the *IoT* devices can communicate but only the common devices belongs to some *vSpace* can communicate via the data plane. These sets of *vSpaces* are dynamically defined by the manager based on the applications and devices intent request.

2.4.3 Decentralized Nokia-*SDN* Network

Apart from the separation of control, management and data plane layers in *SDN*, Nokia *SDN* architecture further partitioned the data plane into n number of domains as mentioned in the earlier texts as *Virtual Places*. Each domain contains a set of network switches connected together in a tree-like topology (i.e. loop-free topology) with the root of the tree being played by a network switch called the root switch. Such loop-free topology aims at simplifying the controller algorithms which are focusing on the core functionalities, namely device discovery and network isolation between *vSpaces*. The global data plane is a connected topology with a unique root switch for each domain and with all the root switches from various domains connected together in a full mesh fashion. The communication link between any two switches within the same domain is referred as *Infra* link and the related communication ports on the switches are called *Infra* ports. Communication links between any pair of root switches are referred to as *Inter* links and related communication ports on the root switches are called as *Inter* ports. To scale the controllability of the data plane, each *vPlace* (or domain) is controlled by a unique *SDN* controller. One such instance is illustrated in the following example.

EXAMPLE 2.3 *For an example, take 3 domains, and* $\text{root}_1 = sw_1, \text{root}_2 = sw_2$ *and* $\text{root}_3 = sw_3$ *are the root switches of the domains* $\text{domain}_1 = \{sw_1, sw_4, sw_5\}$, $\text{domain}_2 = \{sw_2, sw_6, sw_8\}$ *and* $\text{domain}_3 = \{sw_3, sw_7, sw_9\}$ *respectively forms a fully connected components in the data plane topology. Schematic picture is shown in 2.5.*

As mentioned earlier, we split the entire data plane into n number of domains (i.e. *Virtual Places*) to control the SDN network in a scalable manner.

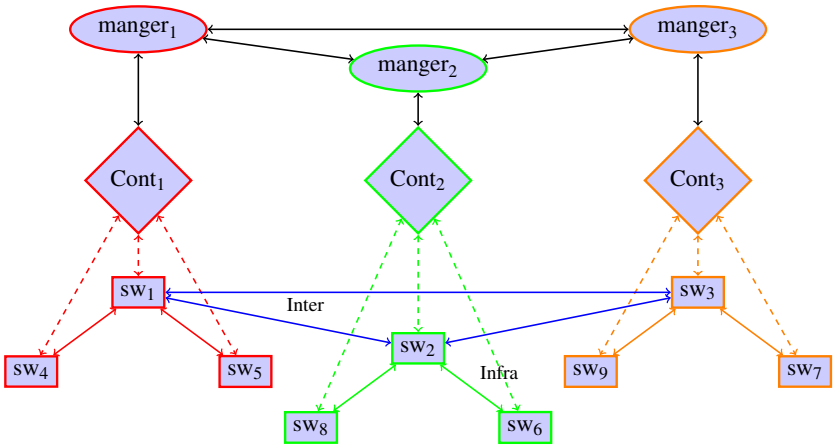


FIGURE 2.5 : Nokia-SDN Architecture

2.4.4 Communication Procedures

Here, we present various types of communication between *SDN* layers for the device discovery and setting up the various network services

MAC learning via MAC Protocol : whenever an *IoT* device connects to a vPlace *SDN* network, the vPlace *SDN* controller should be able to monitor in real time the *IoT* device position (i.e. the switch communication port it is connected) in the data plane in order to provide the device with global communication services. In the rest of the document whenever a user connects to the network we mean a user's device connects to the network.

Users connect to Data plane switches In all domains, we have a set of open access ports (*IoT* ports) : a service link for external users (or say client). If an external user wants to use the *SDN*'s services, it has to connect to one of the free available open access ports in the data plane topology. Users have to choose freely available ports based on its needs.

EXAMPLE 2.4 *One such example of external user joining to the SDN network by connecting to the open access port* op_2 *of the switch* Sw *which has three open access port* $\mathrm{op}_1, \mathrm{op}_2, \mathrm{op}_3$ *(IoT ports) is depicted in figure 2.6.*

Communication Links : The Control Plane and its interaction with Management Plane : The control plane is responsible for building the correct data packet transfer path in the data plane such that it satisfies the management plane properties (set of *vSpaces*). Each controller maintains a list of devices connected together from its perspective and reports this list to the management plane. Further, it stores the information about vSpace properties and the list of devices that satisfy the set vSpace properties and their current domain locations. Based on this information, each controller updates/adds the rules in its respective domain to eventually satisfy

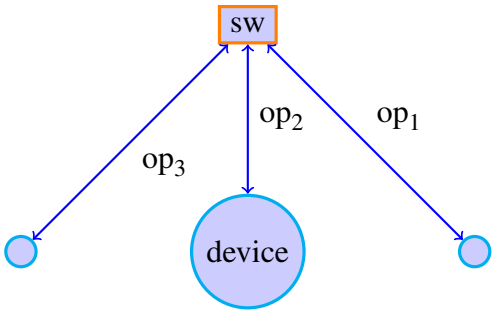


FIGURE 2.6 : An example of external user's device connecting to SDN network via Open access port

the required properties imposed by the management plane (connectivity data path among the same vSpace users and so on).

Interaction between switches and Controllers : The interaction between switches and controllers is essential for discovering SDN network devices (referred as MAC learning) and updating the flow table of switches based on the locations of SDN network user's devices in the data plane.

Device Discovery and Related Position Monitoring : Let's assume a user's device connects to a switch (sw $\in$ domain$_i$) and sends its first packet into the SDN network. In order to notify the controller about the arrival of a user in the data plane, the controller adds a default rule (MAC learning) to all switches. This rule consists in a condition for switches to forward messages with unknown MAC source addresses to the controller. *MAC* media access control address is a unique identifier assigned to a network interface controller for use as a network address in communications within a network segment. This use is common in most IEEE 802 networking technologies, including Ethernet, Wi-Fi, and Bluetooth (source Wikipedia).

In the *SDN* network, communication is the primary reason for building such a network, and this communication is usually made by transferring the data packets between various entities contained in the *SDN* network. It is important to have the *data packet* with appropriate *header field* information. The *header field* information varies among different *data packets* depending on the communication protocol. For a new device arriving or changing its position in the data plane, it should communicate to the SDN manager, for that it sends the *data packet* which contains the *header field* information about its *MAC ID* and the port in which it is connected in the data plane. In case of communicating with another device in the *SDN* network, it sends the *data packet* with *header field* information contains about its *MAC ID* (i.e. source MAC ID) and the other device *MAC ID* (i.e. destination MAC ID). In the case of a group of users (multicast communication) instead of a single destination user, *data packet* contains a reserved multicast MAC address. The *SDN* controller is responsible for building the multicast path to the correct devices within the same vSpace.

DEFINITION 2.1 *MAC Learning Rule :*
source == MAC ID *and* incoming port : == *pt* $\xrightarrow{actions}$ *send MAC ID and pt number to* Cont_i *and send MAC ID to all outgoing Infra ports of the switch sw* $\in \text{domain}_i$.
Where cont_i *is a controller from control plane topology and MAC is the unique Identity for the user i.e each user has a unique MAC Identity number.* ◇

Building the Data Path Between connected Devices : MAC learning is an algorithm, ensuring that a new user's device or an existing user's device location (domain number and switch id and port it is connected to) should be noted correctly by the controller. The controller also reports the existing users and its connected domain position to its manager. With this MAC learning algorithm, *SDN* manager gathers the information about various devices and its current position in the data plane and using this information it instructs the *SDN* controller about establishing the data path among various devices in the data plane. To build data paths between various devices, *SDN* controllers set up (deploy) the *Unicast* and *Multicast* rules in the network switches presented in the data plane.

When a new user's device joins the network by connecting to $\text{sw} \in \text{domain}_i$ via open access port z, sw will forwards this information to all its outgoing FIFO queues within the domain domain_i and controller cont_i of the domain i this is a default rule (MAC learning rule) described above.

The controller of that domain answers by updating the flow table by adding *multicast* and *unicast* rules, which are described in the following texts.

- *Unicast* rule : This rule is to forward the given data packet to a single host.
  Further, we refer to a data packet as a unicast data packet when it is sent by an user and has to be received by an user.
- *Multicast* rule : This rule is to forward the given data packet to multiple hosts.
  Further, we refer to a data packet as a multicast data packet when it is sent by a user and has to be received by multiple users.

Unicast Rule Recall that each controller knows the list of devices connected to its domain. The controller of each domain has a local perspective of device connections and has the information about the list of vSpace_i for all i and devices which belong to it. With all the information in hand, the controller adds/updates a *unicast* type rule appropriately in the flow table of the respective switch. For each i, and pair of user's devices in the network, the controller adds the following rule if the given pair of users at least one of them is connected to that switch.

DEFINITION 2.2 *Unicast Rule :*
source == MAC ID($userY$), incoming port : == pt' *and* destination == MAC ID($userX$) $\xrightarrow{actions}$

send the data packet to the port pt.
Where userX (connected to the switch sw via the port pt) and userY in the vSpace$_i$ *for some* $i \in \{1,2,..,l\}$ *and the data packet contains the header information of source id, destination id of userY and userX respectively.* ◇

Multicast Rule is a broadcast message type which allows to transfer a data packet from a user to all the users which belongs to the same vSpace$_i$ for some $i \in \{1,2,...,l\}$.

DEFINITION 2.3 *Multicast Rule :*
source == MAC ID(*userY*) *and userY in the* vSpace$_i$ $\xrightarrow{actions}$ *send the data packet to the ports* pt_1 , $pt_2 ... pt_k$.
Where switch has users which are in the vSpace$_i$ *connected through these ports* pt_1 , $pt_2 ... pt_k$.
The data packet contains the header field information about the source id userY and vSpace$_i$ *group name.* ◇

Switch Flow table

The overall simplified decision tree to execute various rules from the flow table of a given switch looks as in figure 2.7.

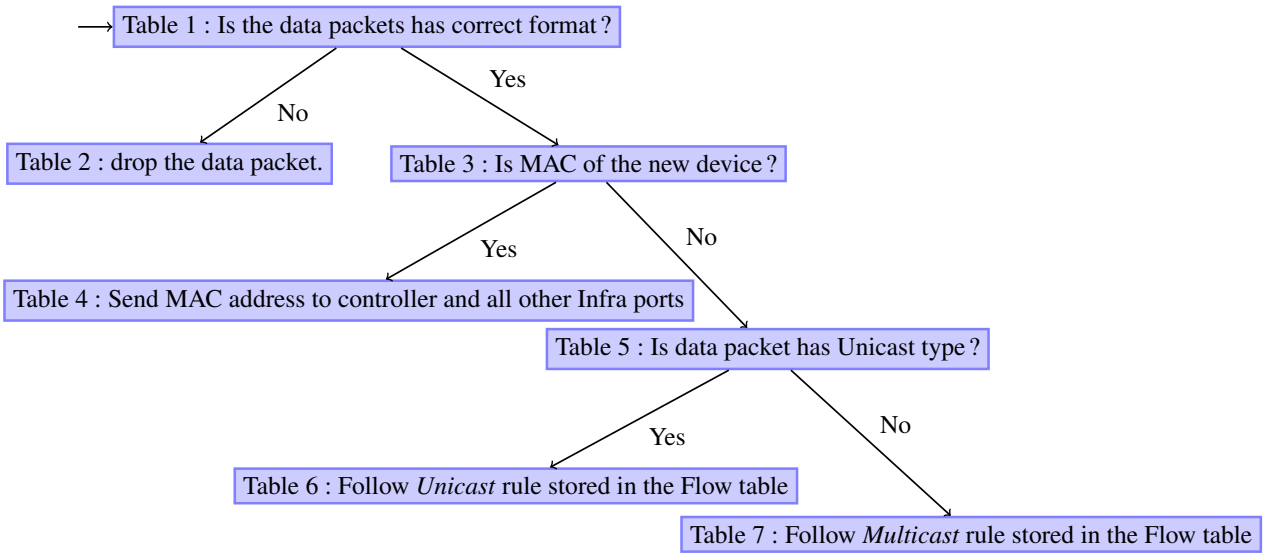


FIGURE 2.7 : Decision Structures of Network switch Flow Table

A switch is a slave machine in the *SDN* architecture in the sense that it refreshes its flow table by adding, deleting the existing flow table rule based on the instruction from the respective *SDN* controllers. When the data packet arrives at one of its incoming ports it takes the data packet header and compares it against the condition structure of the flow table rules as referred in 2.7 and executes the actions related to the rule which condition structure best matches the packet header. action command can either be dropping, modifying the packet header, and/or forwarding the data packet to the particular outgoing ports.

Initially, the Flow table of each switch has empty *Unicast*, *Multicast* rules, and has default *Mac learning* rules. As SDN controller instructions, each switch gets updated on the *Unicast*, *Multicast* and *Mac* rules in their *flow table*.

2.5 Chapter Conclusion

The general goal of *Nokia-SDN-IoT* platform is to control the complex and large scale data plane network. Also to control the data plane in a more scalable way while realizing the application with the help of dynamically re-grouping the set of *IoT* devices (*vSpaces*) based on the real time requests and global application requirements. In such a complex large scale data plane network, in order to process the intent requests fast and scale down the computation process, Nokia SDN platform uses the decentralized controllers and splits the data plane into *n* number of domains (*Virtual Places*). Normally each controller knows the local information about the particular data plane domain rather than the entire data plane configuration (topology and positions of different *IoT* devices). In order to have the global data plane configurations, the manager plane forms the cluster and gathers the global data plane configuration by enquiring each controller in the decentralized controllers set and inform each controller to implement the global specifications. To obtain the local configuration of the *IoT* devices in the data plane (position), it uses the *MAC* learning protocol continuously and regularly updates the information about each data plane domain and informs the managers for global monitoring of the data plane configurations.

Now, we proceed to mathematical modeling of distributed systems and model the specific details of the Nokia *SDN* platform to perform model checking (verification) and synthesize various *SDN* functionalities.

CHAPITRE 3

MODELING THE DISTRIBUTED SYSTEM AND REQUIREMENTS

In this chapter, we will see how to model a *Distributed System* (or *DS*) and express its requirements as temporal *Specifications*. We will demonstrate with small examples on how we are intended to do formal verification and generate supervisors by using control synthesis technique to apply it for a real distributed system *DS* in the following chapters.

3.1 Modeling Notations

In this section, we provide notations and known results that will be used throughout this document. We introduce the notion of languages and a model of an automaton as well as some basic transformations associated with this model. They are borrowed or adapted mostly from [CL08], [BCC98], [Kal+14], [Kal+12].

3.1.1 Languages

We assume a given finite alphabet set $\Sigma = \{\sigma_1, .., \sigma_l\}$ for some $l \in \mathbb{N}$. The set of finite words over Σ is denoted by Σ^* (finite sequence of the alphabets from Σ forms the word), with ε being an empty word. For each $w, w' \in \Sigma^*$ of the form $w = \sigma_1 \dots \sigma_n$ and $w' = \sigma'_1 \dots \sigma'_m$ $(n, m \in \mathbb{N})$, the *concatenation of w and w'* is again a word defined by $w.w' = \sigma_1 \dots \sigma_n.\sigma'_1 \dots \sigma'_m$. The *length* of $w \in \Sigma^*$ is denoted $|w|$ (the length of the empty word is zero). We let Σ^n with $n \in \mathbb{N}$ denote the words of length n over Σ. A set $\mathscr{L}$ of finite words over Σ, $\mathscr{L} \subseteq \Sigma^*$, is called a *language* over Σ.

Given a finite alphabet set $\Sigma_1 \subseteq \Sigma$, we define the *projection* operator on finite words, $P_{\Sigma_1} : \Sigma^* \to \Sigma_1^*$, that removes in a word of Σ^* all the events that do not belong to Σ_1. Formally, P_{Σ_1} is recursively defined as follows : $P_{\Sigma_1}(\varepsilon) = \varepsilon$ and for $\sigma \in \Sigma, w \in \Sigma^*$, $P_{\Sigma_1}(w.\sigma) = P_{\Sigma_1}(w).\sigma$ if $\sigma \in \Sigma_1$ and $P_{\Sigma_1}(w)$ otherwise. Let $\mathscr{L} \subseteq \Sigma^*$ be a language. The definition of projection for words extends to languages : $P_{\Sigma_1}(\mathscr{L}) = \{P_{\Sigma_1}(s) \mid s \in \mathscr{L}\}$.

Conversely, let $\mathscr{L} \subseteq \Sigma_1^*$. The *inverse projection* of $\mathscr{L}$ with respect to Σ, with $\Sigma_1 \subseteq \Sigma$, is $P_{\Sigma_1}^{-\Sigma}(\mathscr{L}) = \{w \in \Sigma^* \mid P_{\Sigma_1}(w) \in \mathscr{L}\}$. For example given string $s = \sigma_1\sigma_2...\sigma_n \in \Sigma_1 \in \Sigma_1^*$, the set of all possible inverse projection for this string (i.e $P_{\Sigma_1}^{-\Sigma}(s)$) is $(\Sigma \setminus \Sigma_1)^*\sigma_1(\Sigma \setminus \Sigma_1)^*\sigma_2(\Sigma \setminus$

$\Sigma_1)^*...\sigma_n(\Sigma \setminus \Sigma_1)^*$. Here we make the inverse projection of the word $s \in \Sigma_1^*$ with respect to the alphabet set Σ. When clear from the context, we will simply denote $P_{\Sigma_1}^{-\Sigma}$ by $P_{\Sigma_1}^{-1}$.

3.1.2 Automaton

Eventhough languages are very useful to describe the behavior of a given system, it is more convenient to use a finite and compact representation to model it. To this effect, the model of automata is commonly used to represent the behavior of systems at a very abstract level. It is composed of a (possibly infinite) number of states (or configurations) and transitions between those states, labeled by actions representing the atomic evolution of the system as well as a set of atomic propositions attached to each state representing the fact that some particular properties/requirements hold in that state.

DEFINITION 3.1 *c* ◇

Action set : For a given automaton, Σ refers to possible actions that can be triggered by the automaton to move from one state to another state. These actions represent the execution behaviour of modelled system while the states represent the current evaluation of modelled system variables.

Atomic Predicates Roughly speaking, the set $L(q) \subseteq$ AP corresponds to the atomic propositions that holds in $q \in Q$. *AP* is nothing but set of atomic proposition to represents the various properties that hold in a state of the given automaton. For example, $L(q) = \{a,b\}$ for $q \in Q$, $\{a,b\} \subseteq$ AP of an automaton indicates that the state q which satisfying the atomic predicates $\{a,b\}$ and not satisfying atomic predicates AP $\setminus \{a,b\}$. To be more precise, an atomic proposition $a = (x \leq i)$, with x a variable and i a numerical constant express an arithmetic relation condition on the value assigned to the variable x. Atomic propositions are used to create various properties. We can introduce an atomic proposition b to express an existence of path between two vertices u, v in a given graph $G = \langle V, E \rangle$, so that b is *true* or *false* based on the existence of path between vertices u and v. The importance and needs of these atomic propositions is purely based on the kind of properties we want to express about the state of the given system model which we are interested in.

EXAMPLE 3.1 *For example, figure 3.1 describes an automaton $\mathscr{A}$, $Q = \{q_0, q_1, r_1, B_1, B_2\}$, $\Sigma = \{a,b\}$, $AP = \{P_1, P_2\}$ with $L(q_0) = \emptyset$, $L(q_1) = \{P_1\}$, $L(r_1) = \{P_2\}$, $L(B_1) = \{P_1, P_2\}$, $L(B_2) = \{P_2\}$, whereas Figure 3.2 is an automaton that models the device that can break down during its normal evolution and further be repaired, where the atomic propositions c, d respectively represent the fact that this device is connected and disconnected.* ◇

◇

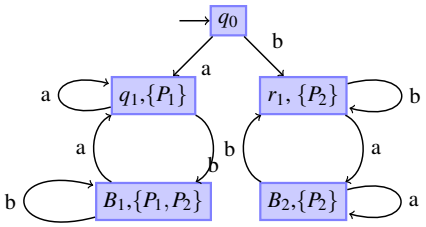


FIGURE 3.1 : Automaton Example

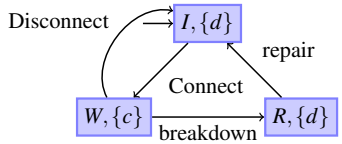


FIGURE 3.2 : Example of the behavior a simple device.

For simplicity, we write $\Delta(q,\sigma,q')$ for $(q,\sigma,q') \in \Delta$. We extend Δ to arbitrary sequences by setting : $\Delta(q,\varepsilon,q)$, and $\Delta(q,w\sigma,q')$ whenever $\Delta(q,w,q'')$ and $\Delta(q'',\sigma,q')$ for some $q'' \in Q$. Similarly, $\Delta(q,\sigma) = \{q' \in Q \mid (q,\sigma,q') \in \Delta\}$, which is naturally extended to a sequence w and denoted by $\Delta(q,w)$ which corresponds to the set of states that can be reached by triggering w in $\mathscr{A}$ starting from the q. A transition $\delta = (q,\sigma,q') \in \Delta$ will also be written as $q \xrightarrow{\sigma}_{\Delta} q'$ (or $q \xrightarrow{\sigma} q'$ if Δ is clear from the context). $\mathscr{A}$ is *deterministic* whenever $\forall q \xrightarrow{\sigma} q'$ and $q \xrightarrow{\sigma} q''$ then $q' = q''$. For an automaton $\mathscr{A} = (Q, q_0, \Sigma, \Delta, \mathrm{AP}, L)$, a *finite run* is $\mathrm{run} = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \ldots q_{n-1} \xrightarrow{\sigma_n} q_n$ for some $n \in \mathbb{N}$ s.t $\forall i \in [1,..,n]$, $q_{i-1} \xrightarrow{\sigma_i} q_i \in \Delta$. We denote the set of all possible finite runs as $\mathrm{Run}(\mathscr{A})$.

A language of an automaton $\mathscr{A}$ is denoted as $\mathscr{L}(\mathscr{A}) := \{w \mid w \in \Sigma^* \wedge \emptyset \neq \Delta(q_0,w) \subseteq Q\}$, where $\Delta(q_0,w)$ is the extended transition of Δ transition in automaton $\mathscr{A}$ that starts from the initial state q_0 through the sequence w.

EXAMPLE 3.2 *For example, in the automaton in example 3.1, one such word in the language of* $\mathscr{L}(\mathscr{A})$ *is* $w := aab$ *(for the run* $\mathrm{run} = q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_1 \xrightarrow{b} B_1$*).* ◇

In the sequel, we will often need to distinguish a subset $F \subseteq Q$ to denote final states. Such states can be used to encode configurations of the system satisfying some properties (e.g. an atomic proposition becomes *true*). The *language of (finite) words* $\mathscr{L}_F(\mathscr{A})$ is the set of words accepted by $\mathscr{A}$, i.e., $\mathscr{L}_F(\mathscr{A}) = \{w \in \Sigma^* \mid \Delta(q_0,w) \subseteq F\}$.

For a given finite $\mathrm{run} = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \ldots . q_{n-1} \xrightarrow{\sigma_n} q_n \in \mathrm{Run}(\mathscr{A})$, with $n \in \mathbb{N}$, we define the sequence of a run as $\mathrm{seq}(\mathrm{run}) = \sigma_1 \ldots \sigma_n$. Note that for any given finite run from an automaton $\mathscr{A}$ starts from the initial state q_0 i.e $\mathrm{run} \in \mathrm{Run}(\mathscr{A})$, the corresponding *seq*(run) belongs to the language of $\mathscr{A}$, i.e $\forall \mathrm{run} \in \mathrm{Run}(\mathscr{A})$, $\mathrm{seq}(\mathrm{run}) \in \mathscr{L}(\mathscr{A})$. $\mathrm{Run}(\mathscr{A},q)$ represents all the runs of $\mathscr{A}$ that start from the state q, where $q \in Q$.

Instead of considering only the run $\mathrm{run} = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \ldots . q_{n-1} \xrightarrow{\sigma_n} q_n$, we might be interested in the sequence the set of atomic propositions that are *true* along this run,

i.e. $L(q_0).L(q_1).L(q_2)\dots L(q_n)$.

DEFINITION 3.2 *For a given run* $\text{run} = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \dots q_{n-1} \xrightarrow{\sigma_n} q_n \in \text{Run}_f(\mathscr{A})$*, the* trace *of* run *is defined by*

$$\text{trace}(\text{run}) = L(q_0).L(q_1).L(q_2)\dots L(q_n) \in AP^*$$

and the set of traces of the automaton is defined by $Tr(\mathscr{A}) := \{\text{trace}(\text{run}) \mid \text{run} \in \text{Run}(\mathscr{A})\} \subseteq AP^*$ *the set of traces of* $\mathscr{A}$. ◇

EXAMPLE 3.3 *For example, given the automaton in example 3.1, one such possible finite run is*

$$run := q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_1 \xrightarrow{b} B_1 \xrightarrow{a} q_1,$$

and the corresponding trace of this run is

$$\text{trace}(\text{run}) = L(q_0).L(q_1).L(q_1).L(B_1).L(q_1) = \emptyset\{P_1\}.\{P_1\}.\{P_1,P_2\}.\{P_1\}$$

◇

For $w \in \text{Tr}(\mathscr{A})$, we define Run^w as

$$\text{Run}^w = \{\text{run} \mid \text{trace}(\text{run}) = w \wedge \text{run} \in \text{Run}(\mathscr{A})\}$$

as the set of runs such that each trace of each run is equal to w. Note that for an automaton $\mathscr{A}$ and $w \in \text{Tr}(\mathscr{A})$, there exists $\text{run} \in \text{Run}(\mathscr{A})$ such that $\text{trace}(\text{run}) = w$ and $\text{seq}(\text{run}) \in \mathscr{L}(\mathscr{A})$.

Let us now introduce some more notations that will be useful in the sequel. We are interested in computing set of states that are not reachable from the set of initial states of an automaton $\mathscr{A}$, or dually that a set of states is always reachable from its sets of initial states by triggering events of $\Sigma' \subseteq \Sigma$.

Given a set of states $E \subseteq Q$ of an automaton $\mathscr{A}$, the functions $\mathsf{Post}^{\mathscr{A}}_{\Sigma'}$, $\mathsf{Pre}^{\mathscr{A}}_{\Sigma'}$ from $2^Q \to 2^Q$ are defined as follows :

$$\mathsf{Pre}^{\mathscr{A}}_{\Sigma'}(E) = \{q \in Q \mid \exists \sigma \in \Sigma', \Delta(q,\sigma) \cap E \neq \emptyset\} \quad (3.1)$$

$$\mathsf{Post}^{\mathscr{A}}_{\Sigma'}(E) = \{q' \in Q \mid \exists \sigma \in \Sigma', q' \in E,\ \Delta(q,\sigma,q')\} \quad (3.2)$$

The states of $\mathsf{Pre}^{\mathscr{A}}_{\Sigma'}(E)$ are such that at least one immediate successor belongs to E by Σ', whereas $\mathsf{Post}^{\mathscr{A}}_{\Sigma'}(E)$ corresponds to the immediate successors of E in $\mathscr{A}$.

Given an automaton $\mathscr{A}$, $\mathsf{CoReach}^{\mathscr{A}}_{\Sigma'}(E)$ is the set of states from which E is reachable by triggering a sequence of Σ'. It is given by the following least fix-point

$$\begin{cases} E_0 &= E \\ E_{i+1} &= E_i \cup \mathsf{Pre}^{\mathscr{A}}_{\Sigma'}(E_i) \end{cases} \tag{3.3}$$

Note that by Tarski's theorem [Tar55], since the $E \cup \mathsf{Pre}^{\mathscr{A}}_{\Sigma'}(E)$ is monotonic, the limit of the fix-point actually exists (but may be uncomputable as the state space is possibly infinite).

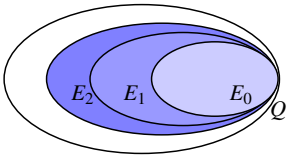


FIGURE 3.3 : Monotonic of Pre set

EXAMPLE 3.4 *For example, given automaton in example 3.1 take $E_0 = \{B_2\}$ and $\Sigma' = \{a\}$. Then $E_1 = \{B_2, r_1\}$ and $E_i = E_1$ for all $i \geq 2$, so that we have a fix point for E_0 ; here $E = E_1$.* $\diamond$

Similarly, $\mathsf{Reach}^{\mathscr{A}}_{\Sigma'}(E)$ is the set of states that can be reached from E by $(\Sigma')^*$

$$\begin{cases} E_0 &= E \\ E_{i+1} &= E_i \cup \mathsf{Post}^{\mathscr{A}}_{\Sigma'}(E_i) \end{cases} \tag{3.4}$$

If $E = \{q_0\}$ then we obtain the set of reachable states in $\mathscr{A}$ by only triggering events in Σ'.

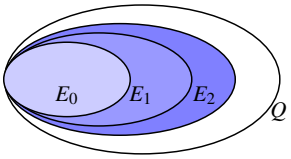


FIGURE 3.4 : Monotonic Post set

EXAMPLE 3.5 *For example, given automaton in example 3.1 take $E_0 = \{q_0\}$ and $\Sigma' = \{a, b\}$, then post of E is $\{q_1, r_1\}$ so that we have $E_1 = \{q_0, q_1, r_1\}$ and we have $E_2 = \{q_0, q_1, r_1, B_1, B_2\}$, and so on such that we have $E_i = E_2$ for $i \geq 2$, we finally get $E = Q$ for the automaton given in the example 3.1.* $\diamond$

Composition of automata In most situations, a system is initially given by a set of components modeled by automata that interact with each other by sharing common events. Its global behavior is obtained by composing these automata together using the parallel composition operator that represents the concurrent behavior of the automata with synchronization on the common events (for example sending a message from one component to another component, and receiving the message by one component).

DEFINITION 3.3 *Given two automata* $\mathscr{A}_i = (Q_i, q_{0,i}, \Sigma_i, \Delta_i, AP_i, L_i)$, $i \in \{1,2\}$ *as in definition 3.1, the* composition *of these two automata is defined as,*
$\mathscr{A} := \mathscr{A}_1 \parallel \mathscr{A}_2 = (Q, q_0, \Sigma, \Delta, AP, L)$, *where* $Q := Q_1 \times Q_2$, $q_0 := q_{0,1} \times q_{0,2}$, $\Sigma := \Sigma_1 \cup \Sigma_2$, $\Delta \subseteq Q \times \Sigma \times Q$, *s.t for all* $\sigma \in (\Sigma_1 \setminus \Sigma_2)$ *and symmetrically for all* $\sigma \in (\Sigma_2 \setminus \Sigma_1)$ *:*

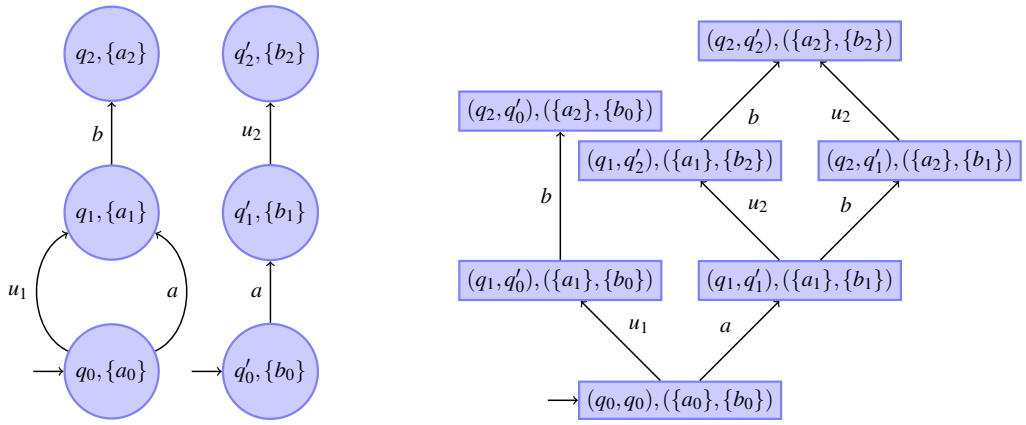
$$\frac{q_1 \xrightarrow{\sigma} q_1' \in \Delta_1, q_2 \in Q_2}{(q_1,q_2) \xrightarrow{\sigma} (q_1',q_2) \in \Delta} \qquad \frac{q_2 \xrightarrow{\sigma} q_2' \in \Delta_2, q_1 \in Q_1}{(q_1,q_2) \xrightarrow{\sigma} (q_1,q_2') \in \Delta}$$

and for all $\sigma \in \Sigma_1 \cap \Sigma_2$

$$\frac{q_1 \xrightarrow{\sigma} q_1' \in \Delta_1, q_2 \xrightarrow{\sigma} q_2' \in \Delta_2}{(q_1,q_2) \xrightarrow{\sigma} (q_1',q_2')}$$

$AP := AP_1 \times AP_2$, $L := Q \to 2^{AP_1} \times 2^{AP_2}$ *where* $L(q_1,q_2) := (L_1(q_1), L_2(q_2))$, *for all* $(q_1,q_2) \in Q$. ◇

EXAMPLE 3.6 *Let* $\mathscr{A}_1$ *and* $\mathscr{A}_2$ *be two automata over respectively* $\Sigma_1 = \{a, u_1, b\}$ *and* $\Sigma_2 = \{a, u_2\}$ *represented in Fig. 1(a). The automaton* $\mathscr{A}_1 \| \mathscr{A}_2$ *obtained by composing these two automata is depicted in Fig. 1(b),*



(a) $\mathscr{A}_1$ and $\mathscr{A}_2$ (b) $\mathscr{A}_1 \| \mathscr{A}_2$

FIGURE 3.5 : Parallel composition of two automata

Using the projection $P_{\Sigma_i} : \Sigma^* \to \Sigma_i^*$, $i = 1,2$, we can characterize the language resulting from the parallel composition as follows :

$$\mathscr{L}(\mathscr{A}_1 \| \mathscr{A}_2) = P_{\Sigma_1}^{-1}(\mathscr{L}(\mathscr{A}_1)) \cap P_{\Sigma_2}^{-1}(\mathscr{L}(\mathscr{A}_2)) \tag{3.5}$$

Equivalently, we can state that

$$\mathscr{L}(\mathscr{A}_1 \| \mathscr{A}_2) = \{s \in \Sigma^* \mid P_{\Sigma_1}(s) \in \mathscr{L}(\mathscr{A}_1) \wedge P_{\Sigma_2}(s) \in \mathscr{L}(\mathscr{A}_2)\} \tag{3.6}$$

Moreover, if the two automata are equipped with a set of final states (F_1 and F_2), then we have :

$$\mathscr{L}_{F_1 \times F_2}(\mathscr{A}_1 \| \mathscr{A}_2) = P_{\Sigma_1}^{-1}(\mathscr{L}_{F_1}(\mathscr{A}_1)) \cap P_{\Sigma_2}^{-1}(\mathscr{L}_{F_2}(\mathscr{A}_2)) \tag{3.7}$$

DEFINITION 3.4 *Given the composition of two automata $\mathscr{A}_1 \parallel \mathscr{A}_2$, and a* $\text{run} = (q_{0,1}, q_{0,2}) \xrightarrow{\sigma_1} (q_{1,1}, q_{1,2}), ..., \xrightarrow{\sigma_n} (q_{n,1}, q_{n,2}) \in \text{Run}_f(\mathscr{A}_1 \parallel \mathscr{A}_2)$. *The projection of such a run over Σ_i (alphabet of automata $\mathscr{A}_i$) for some $i \in \{1,2\}$ is denoted as* $\text{run} \downarrow \mathscr{A}_i := q_{0,i} \xrightarrow{\sigma'_1} q_{1,i}, ..., \xrightarrow{\sigma'_n} q_{n,i}$ *such that : for all $j \in [1,..,n]$*
If $\sigma_j \in \Sigma_i$ and $q_{j-1,i} \xrightarrow{\sigma_j} q_{j,i} \in \Delta_i$ then $\sigma'_j = \sigma_j$
If $\sigma_j \notin \Sigma_i$ then $q_{j-1,i} \xrightarrow{\varepsilon} q_{j,i} \in \Delta_i$ and $\sigma'_j = \varepsilon$ where ε is an empty transition and $q_{j-1,i} = q_{j,i}$. $\diamond$

3.2 Symbolic Transition System

Automata are the basic way to represent the behaviour of systems and to analyse them. However they suffer from different drawbacks :

- Nowadays, when modelling realistic systems, it is often convenient to manipulate state variables instead of simply atomic states, allowing a compact way to specify systems handling data. For example, one can consider a system where the number of system states is infinite but can be generated by finite actions [Kal+14][Kal+12].
  Within this framework, a state of the underlying state machine can be seen as a particular instantiating of a vector of variables. If the domain of the variables are infinite, the semantics of such a system is therefore given by a potentially infinite automaton where the states are valuations of the variables. Other interests are that modeling such systems in a symbolic way avoids a state space explosion due to the fact that numerical aspects have to be taken into account and that it eases the modeling of these systems.

- As already mentioned, the system we are considering is most of the time defined by several components. Definition 3.3 gives a way to compose such components in a single way with the limitation that the communications are synchronous (i.e. take zero time), which obviously is not the case in many applications (when there is asynchronous communication between the components in the *DS*). In other words, when considering asynchronous distributed systems, the communication delays between the components of the system must also be taken into account, whereas, at the same time since we can reason about them without considering the actual size of the queues
  To tackle such considerations, one can consider a timed automaton [Alu99] allowing to consider the delays between the transmission and the reception of an action. In this document, we do prefer to consider systems that are composed of several systems that communicate asynchronously by means of FIFO channel that are assumed to be unbounded

and reliable (i.e., we are not interested by the exact time an event takes to be transmitted but rather than it takes some delays) [Kal+14][PP91][MW08][LJJ06]. The fact that we choose FIFO channels is motivated by the fact that we are interested in distributed systems, in which actions are transmitted from one component to another one.

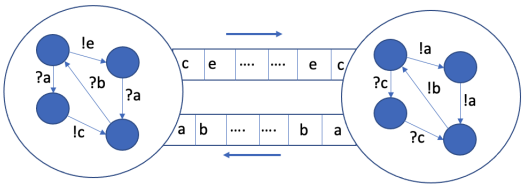


FIGURE 3.6 : Example of Distributed System Communicating via FIFO Queues

From now on, we shall consider Symbolic Transition Systems that communicate with each other via FIFO channels. Each component of the system is a transition system with variables, whose domain can be infinite, and is composed of symbolic transitions. Each transition has a guard on the variables of the system and an update function which indicates the evolution of the variables when the transition is fired. Furthermore, transitions are labeled with symbols taken from a finite alphabet. This model allows the representation of infinite systems whenever the variables take their values in an infinite domain. It has a finite structure and offers a compact way to specify systems handling data.

Symbolic System The system models are created by writing symbolic transitions of the system. We have already seen an automaton model to capture the behavior of a given system, but it's an extremely tedious job to capture the behaviour of the system by automaton model. We do have automated tools to create such an automaton model from the given symbolic transition model, writing the symbolic transition model of the system is relatively easy and it's more practical than writing extensive and detailed models in automaton form. So here we introduce what is a symbolic transition model, these definitions are borrowed from the paper [Kal+14],[Kal+12].

Let us define a system model $\mathscr{T}$ with a set of variables $V = \langle v_1, v_2, ..., v_n \rangle$ and whose assignment range over $\mathscr{D}_V = \prod_{i\in[1,..,n]} \mathscr{D}_{v_i}$. The valuation of n-tuple variable belongs in $\mathscr{D}_V := \prod_{i\in[1,..,n]} \mathscr{D}_{v_i}$ (this set represents all possible values of V). A valuation of variable V is denoted $\nu \in \mathscr{D}_V$, and corresponds to a predicate value of the variable V. The set of initial conditions is denoted by $\Theta \subseteq \mathscr{D}_V$. This model finite set of actions Σ and transition set Δ and finite number of FIFO queues $Q = Q_{\text{in}} \cup Q_{\text{out}}$ where Q_{in} is a finite number of incoming queues and Q_{out} is a finite number of outgoing queues for some $l \in \mathbb{N}$, each queue can hold an infinite number of messages from the message set M (note that M can carry the values on some variables as far as number of variables are finite). The action involves in sending a message $a \in M$ to the output queue $q \in Q_{\text{out}}$ indicated by $(q, !, a)$, reading a message ($b \in M$) action from an input queue $q \in Q_{\text{in}}$ indicated by $(q, ?, b)$. It is stressed to mention that reading a message and receiving a message

differ by taking the header message and appending the message to the tail of the current queue content respectively.

We now shall introduce more formally a *symbolic communicating transition model* for the system in the following definition. This definition is adapted from [Kal+14] and [Kal+12].

DEFINITION 3.5 *A* symbolic communicating transition system *(SCTS) is a tuple of* $\mathscr{T} := \langle V, \Theta, M, Q, L, l_0, U, AP, \Sigma, \Delta \rangle$, *where*

- $V := \langle v_1, v_2, .., v_n \rangle$ *is a n-tuple variable, valuation of this n-tuple variable belongs in* $\mathscr{D}_V := \prod_{i \in [1,..,n]} \mathscr{D}_{v_i}$ *(this set represents all possible assignment to the variable V),*
- $\Theta \subseteq \mathscr{D}_V$ *a set of initial condition on the assignments to the variable V,*
- *M a finite set of messages,*
- $Q := Q_{in} \cup Q_{out}$, *s.t* $|Q_{in}| = |Q_{out}| = l$ *and each queue in* $q \in Q$ *is a FIFO queue can hold infinite number of messages from the set M,*
- *L is a nonempty, finite set of locations and* $l_0 \in L$ *is the initial location,*
- *AP set of atomic predicates to represent the set of proposition will be useful to represents the current nature of the system state,* $U : L \times \mathscr{D} \to \mathscr{D} \times 2^{AP}$ *is the labeling function which set the atomic propositions of the system based on the current location and the valuation of the system variable V,*
- $\Sigma := (Q \times \{?, !\} \times M) \cup \Sigma_{\text{int}}$ *is a finite set of alphabets, where* $(q_{in,i}, ?, m)$ *represents reading the message m from the queue header* $q_{in,i}$, $(q_{out,j}, !, m)$ *represents sending the message m (or adding the message m at the tail) to the queue* $q_{out,j}$,
- Δ *is a finite set of symbolic transitions, each transition* $\delta \in \Delta$ *is a tuple* $\delta = \langle l, \sigma, G, A, l' \rangle$ *where a location* $l \in L$, *called the origin location of* δ *transition, an action* $\sigma \in \Sigma$ *called the action of* δ *transition,* $G \subseteq \mathscr{D}_V$ *a predicate set called as guard of* δ *transition,* $A : \mathscr{D}_V \to \mathscr{D}_V$ *is an update function, which update the assignments to the variable V, a location* $l' \in L$, *called the target location of* δ *transition.*

$\diamond$

As you may note, by definition, the atomic predicates set AP can also be the condition on the variables V with specific values from the domain set $\mathscr{D}_V$. One such example given in forthcoming texts (Example 3.7).

Furthermore, we can compose any two given communicating transition system $\mathscr{T}_1$, $\mathscr{T}_2$ as follows :

DEFINITION 3.6 *Given two symbolic communicating transition systems* $\mathscr{T}_1 = \langle V_1, \Theta_1, M_1, Q_1, L_1, l_{0,1}, U_1, AP_1, \Sigma_1, \Delta_1 \rangle$, $\mathscr{T}_2 = \langle V_2, \Theta_2, M_2, Q_2, L_2, l_{0,2}, U_2, AP_2, \Sigma_2, \Delta_2 \rangle$, $\mathscr{T} := \mathscr{T}_1 \parallel \mathscr{T}_2$ *defined as,*

- $V = (V_1, V_2)$,
- $L := L_1 \times L_2$,
- $\Theta := \Theta_1 \times \Theta_2$,
- $M, := M_1 \cup M_2$,
- $l_0 := (l_{0,1}, l_{0,2})$,
- $U = (U_1, U_2)$, *such that* $U := (L_1 \times \mathscr{D}_1 \times L_2 \times \mathscr{D}_2) \to \mathscr{D}_1 \times 2^{AP_1} \times \mathscr{D}_2 \times 2^{AP_2}$,
- $Q := Q_{in} \cup Q_{out}$ $(= Q_{in,1} \cup Q_{in,2} \cup Q_{out,1} \cup Q_{out,2})$,
- *and furthermore* $q_{in_{1,2}} = q_{out_{2,1}}$ *and* $q_{in_{2,1}} = q_{out_{1,2}}$, $\Sigma := \Sigma_1 \cup \Sigma_2$ *and* $\Sigma_1 \cap \Sigma_2 = \emptyset$, *finite set of transition are as follows,*

$$\Delta := \begin{array}{l} \{\langle (l_1, l_2), \sigma_1, G, A, (l_1^{'}, l_2) \rangle \mid \sigma_1 \in \Sigma_1 \wedge \langle l_1, \sigma_1, G, A, l_1^{'} \rangle \in \Delta_1 \wedge l_2 \in L_2\} \\ \cup \{\langle (l_1, l_2), \sigma_2, G, A, (l_1, l_2^{'}) \rangle \mid \sigma_2 \in \Sigma_2 \wedge \langle l_2, \sigma_2, G, A, l_2^{'} \rangle \in \Delta_2 \wedge l_1 \in L_1\} \end{array}$$

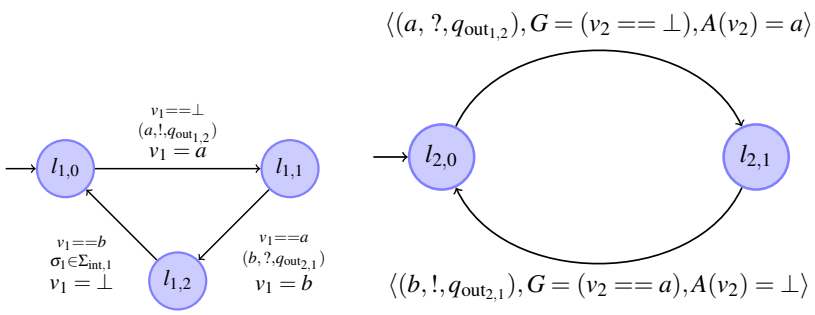
◇

In the sequel, we will define a set of distributed systems $\{\mathscr{T}_1, \mathscr{T}_2, .., \mathscr{T}_m\}$, communication between those subsystems $\mathscr{T}_i$, for all $i \in [1, .., m]$

DEFINITION 3.7 (Distributed System) *: For a fixed m number of SCTS systems* $\mathscr{T}_i$, $i \in [1, .., m]$, *the resulting distributed system is given by* $\mathscr{T} := \|_{i \in [1,..,m]} \mathscr{T}_i$, *i.e the product of each* $\mathscr{T}_i$ *acting in parallel and exchanging information through FIFO channels.* ◇

This operation is associative and commutative up to state renaming.

EXAMPLE 3.7 *For example, let see two such communicating transition system* $\mathscr{T}_1, \mathscr{T}_2$ *and its composition* $\mathscr{T} = \mathscr{T}_1 \parallel \mathscr{T}_2$ *in fig. 3.7 and fig. 3.7 respectively.* ◇



(a) $\mathscr{T}_1$ (b) $\mathscr{T}_2$

FIGURE 3.7 : Two Symbolic Transition Systems $\mathscr{T}_1$, $\mathscr{T}_2$

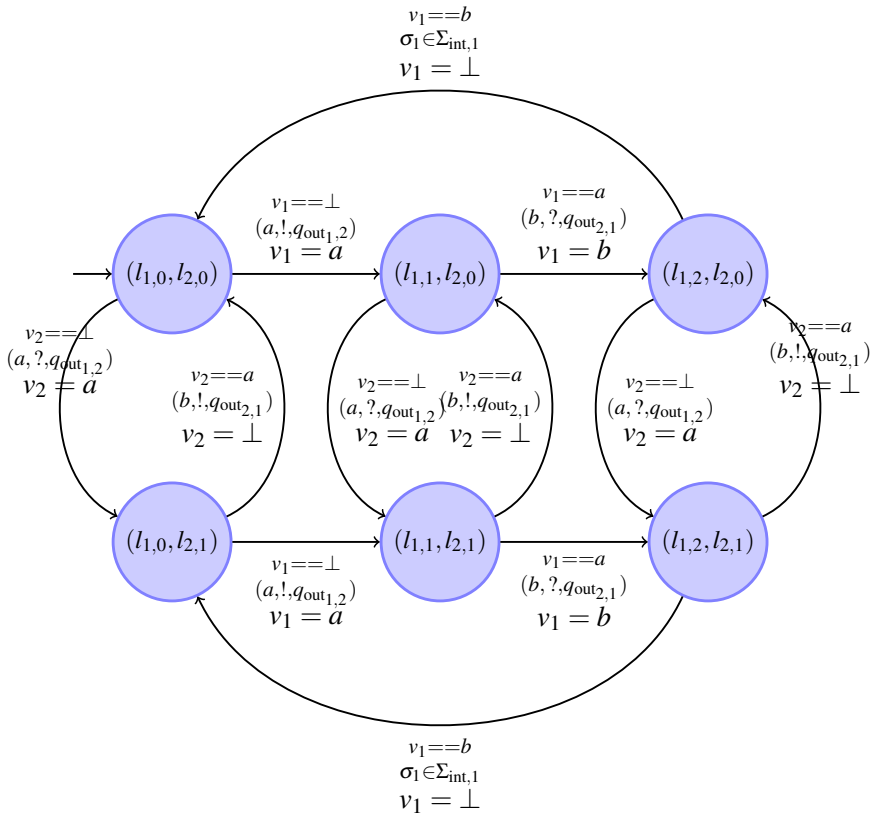


FIGURE 3.8 : Composition of communicating transition systems $\mathscr{T} = \mathscr{T}_1 \parallel \mathscr{T}_2$.

For the above *SCTS* system definition 3.5, we are introducing the semantics as follows,

DEFINITION 3.8 *The* semantics *of an* SCTS $\mathscr{T} := \langle V, \Theta, M, Q, L, l_0, , \Sigma, \Delta, U, AP\rangle$ *is an automaton* $[\mathscr{T}] := \langle X, X_0, \Sigma, \rightarrow, U, AP\rangle$*, where*

- *the set of states are* $X := L \bigtimes \mathscr{D}_V \bigtimes_{q\in Q}\{w_q \mid w_q \in M^*\}$ *where* $w_q \in M^*$ *for all queue* $q \in Q$*,*
- *the set of initial state are* $X_0 := \{l_0\} \bigtimes \Theta \bigtimes_{q\in Q} w_q^{\varepsilon}$*, where* $w_q^{\varepsilon} = \varepsilon$ *an empty string,*
- *the set of action labels are* Σ*, and*
- *the transition relation* $\rightarrow = \bigcup_{\delta\in\Delta} \subseteq X \times \Delta \times X$*, such that*

$$\frac{(l,\nu,w_q) \in X, \delta = \langle l,\sigma,G,A,l'\rangle \in \Delta \wedge \sigma = (q,!,a) \wedge \nu \in G}{\big((l,\nu,w_q),\sigma,(l',A(\nu),w_q.a)\big) \in \rightarrow}$$

In the above transition only q-th queue content will change, remaining queues content will be the same.

$$\frac{(l,\nu,w_q) \in X, \delta = \langle l,\sigma,G,A,l'\rangle \in \Delta \wedge \sigma = (q,?,a) \wedge w_q = a.w'_q \wedge \nu \in G}{\big((l,\nu,w_q),\sigma,(l',A(\nu),w'_q)\big) \in \rightarrow}$$

In the above transition only q-th queue content will change, remaining queues content will be the same.

$$\frac{(l,\nu,w_q) \in X, \delta = \langle l,\sigma,G,A,l'\rangle \in \Delta \wedge \sigma \in \Sigma_{\text{int}} \wedge \nu \in G}{\big((l,\nu,w_q),\sigma,(l',A(\nu),w_q)\big) \in \rightarrow}$$

In the above transition all the queue contents will remain the same.

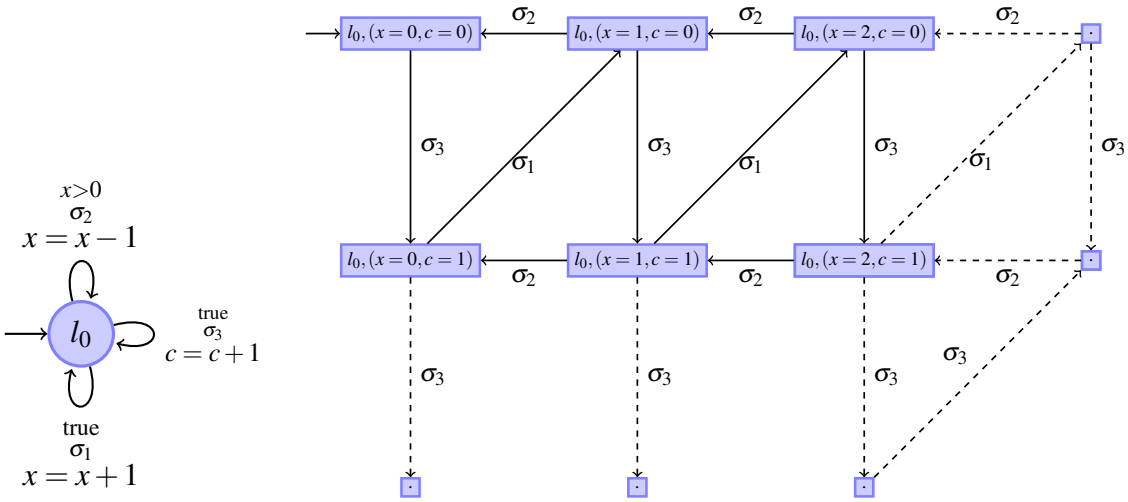
- *Finally, the labeling function* $U : X \rightarrow \mathscr{D} \times 2^{AP}$*, based on the current valuation of the system variable and location, keeps the atomic predicates which are true in the current state of the system.*

$\diamond$

EXAMPLE 3.8 *Let's take an example,* SCTS *:* $\mathscr{T} := \langle V = \{x,c\}, \Theta = \{(x=0, c=0)\}, M = \{a,b,c\}, Q = Q_{in}(=\{q_{2,1},q_{3,1}\}) \cup Q_{out}(=\{q_{1,2}\}), L = \{l_0\}, l_0, \Sigma = \{\sigma_1 = (b,?,q_{2,1}), \sigma_2 = (a,!,q_{1,2}), \sigma_3 = (c,?,q_{3,1})\}, \Delta = \{\langle l_0,\sigma_1,true,x=x+1,l_0\rangle, \langle l_0,\sigma_2,(x>0),x=x-1,l_0\rangle, \langle l_0,\sigma_3,true,c=c+1,l_0\rangle\}\rangle$*, for the transitions* $\langle l_0,\sigma_1,true,x=x+1,l_0\rangle$ *the guard is always true and update function is* $A(x) = x+1$*,* $\langle l_0,\sigma_2,(x>0),x=x-1,l_0\rangle$ *guard is* $(x>0)$ *and update function is* $A(x) = x-1$*,* $\langle l_0,\sigma_3,true,c=c+1,l_0\rangle\}$ *guard is always true and update function is* $A(c) = c-c$*, and we will build corresponding* $[\mathscr{T}]$ *showed in the figure 3.9. The* $q_{1,2}$ *is outgoing queue and* $q_{2,1}, q_{3,1}$ *are incoming queues for the defined* SCTS $\mathscr{T}$*.*

The action $\sigma_1 = (b, ?, q_{2,1})$ *indicates the reading of message b from the queue* $q_{2,1}$, $\sigma_2 = (a, !, q_{1,2})$ *indicates sending the message a to the queue* $q_{1,2}$ *and* $\sigma_3 = (c, ?, q_{2,1})$ *indicates the reading of message c from the queue* $q_{3,1}$.

Although the transition $\langle l_0, \sigma_1, true, (x = x+1), l_0 \rangle$ *has guard true, this transition can only happen if the message channel* $q_{2,1}$ *has message b in its head.*
Similarly the transition $\langle l_0, \sigma_3, true, (c = c+1), l_0 \rangle$ *has guard true, this transition can only happen if the message channel* $q_{3,1}$ *has message c in its head. The symbolic communicating transition system (*SCTS*) and corresponding semantics are depicted in fig. 3.9.*



(a) $\mathscr{T}$ (b) LTS of $\mathscr{T}$

FIGURE 3.9 : Automaton $[\mathscr{T}]$ of $\mathscr{T}$

◇

By understanding the transition in the semantics of *SCTS* $[\mathscr{T}]$ (or for simplicity we represent them as $[\mathscr{T}]$), we can extend the concept of reachable and co-reachable as mentioned in equations 3.1, 3.2 for automata to the semantics of symbolic transition systems as follows. For a given set $Y \subseteq X$, and the transition set $\Delta' \subseteq \Delta$, the reachable and co-reachable sets are defined as

$$\mathsf{Reach}^{\mathscr{T}}_{\Delta'}(Y) := \bigcup_{n \geq 0} (\mathsf{Post}^{\mathscr{T}}_{\Delta'}(Y))^n \tag{3.8}$$

$$\text{Co-reach}^{\mathscr{T}}_{\Delta'}(Y) := \bigcup_{n \geq 0} (\mathsf{Pre}^{\mathscr{T}}_{\Delta'}(Y))^n \tag{3.9}$$

Note the difference in equations 3.8,3.9 compare to 3.1, 3.2 : in the earlier case we have Δ' in computing the Post and Pre set, and in the latter case we use Σ' to compute Post and Pre set, where $\mathsf{Pre}^{\mathscr{T}}_{\Delta'}(Y) := \{x' \in X \,|\, \exists x \in Y, \exists \delta \in \Delta' : x' \xrightarrow{\delta} x \in \rightarrow\}$ is the set of states from which doing an execution from the transition Δ' set to reach a state in $Y \subseteq X$,
$\mathsf{Post}^{\mathscr{T}}_{\Delta'}(Y) := \{x' \in X \,|\, \exists x \in Y, \exists \delta \in \Delta' : x \xrightarrow{\delta} x' \in \rightarrow\}$ is the set of possible states in X can be

reachable by doing an execution from the transition Δ' set from one of the state of the set $Y \subseteq X$. $(\mathsf{Pre}^{\mathscr{T}}_{\Delta'}(Y))^n$ and $(\mathsf{Post}^{\mathscr{T}}_{\Delta'}(Y))^n$ are computed by computing recursively n times from the set Y. The eq. (3.9) is **undecidable** only if the set $[\mathscr{T}]$ is infinite.

3.3 Formal Verification of Distributed system

We have seen so far how we intend to model the distributed system *DS* using automata and symbolic transition systems. Now we will move on to express the system requirements as a set of specifications. The overall formal verification of system scheme is illustrated in figure 3.3.

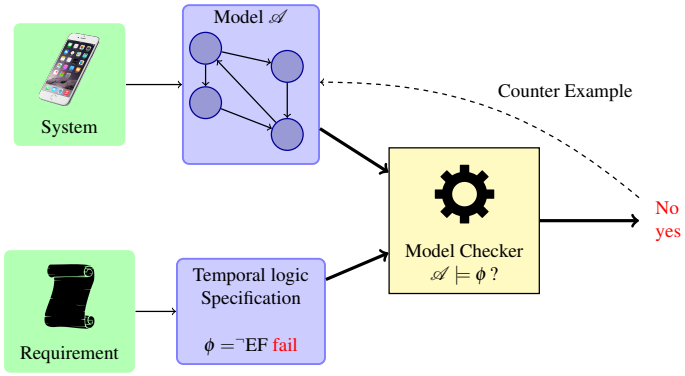


FIGURE 3.10 : Formal Verification of System

Verification or Model checking [BK08] is the process of verifying and simulating a given system .

Model-checking :

For the given required specifications, in the verification mode : if the system satisfies the specifications, then it produces the result saying that the system is correct to the given specifications. If not then verification will notify the user with a *counter example* : a trace of sequence of actions performed in the system that violates the specification.

3.3.1 Expressing the System Requirements as Specifications

Expressing specification is very important in the model checking process, to express the correctness of the system. If we are not expressing the specification (or requirements) of the concerned system, we are not going to find the correctness or failure of the expressed (modelled) system. There are many ways to express the expected specified specification ; one such scheme is Linear Temporal Logic (*LTL*) [Pnu77] a language which is well suited for state-based systems (like automata).

Linear Temporal Logic Specification

LTL is built over propositional logic (based on Boolean operators $\neg,\vee,\wedge$) by adding the notion of discrete (*i.e* natural number) timeline and temporal operators such as the following $\{\Box, \Diamond, X, \mathscr{U}\}$ just to name a few : $\Box\psi$ means ψ is True in **all** future moments, $\Diamond\psi$ means ψ is True in **some** future moments. $X\psi$ means that the formula ψ is true in all the possible next states. $\phi\mathscr{U}\psi$ means ϕ must remain true until ψ becomes true.

LTL formulas are built over a finite set of propositional variables referred to as *Atomic Propositions* (AP) combined with aforementioned Boolean and temporal operators. Temporal operators can be nested. For instance $\Box\Diamond\phi$ refers to executions in which at all moments, there is a moment in future where ϕ holds ; thus, ϕ holds at infinitely many points in time. Formula $\Diamond\Box\phi$ refers to executions in which there is a moment in time after which ϕ holds at all positions ; thus ϕ becomes an invariant after a while.

Within the Model Checking (MC) framework, to verify that system $\mathscr{A}$ satisfies the specification ϕ (i.e $\mathscr{A} \vDash \phi$) there are available software tools such as Spin [Hol97], NuSMV [Cim+02]. $\mathscr{A} \vDash \phi$ means that all the traces from the system $\mathscr{A}$ satisfy the formula ϕ. If this is not the case, then we write $\mathscr{A} \nvDash \phi$, i.e whenever there is a trace violating ϕ in $\mathscr{A}$, we have $\mathscr{A} \nvDash \phi$.

EXAMPLE 3.9 *Let's say that a safety property* $\phi := \Box(a \rightarrow b)$*, and say a finite trace* $\text{trace}_1 = \{\emptyset\},\{a,b\},..\{b\}$*, then this trace satisfies the formula* ϕ*. Now, let's take the trace* $\text{trace}_2 = \{\},\{\},...,\{a,\neg b\}$*, then this trace does not satisfy the formula* ϕ*. Iff such a trace can be triggering in the automaton* $\mathscr{A}$*, then* $\mathscr{A}$ *does not satisfies the safety property* ϕ*.* ◇

Let us give some more details about the *LTL* framework.

The Linear Temporal Logic *LTL* over infinite traces was originally proposed in computer science as a specification language for concurrent programs which is the fragment of first order logic (*FO*). Let AP be a set of atomic predicates or propositions. Expressions in *LTL* can either be *state formulas* or *path formulas*. The set of path formulas can be defined inductively as follows :

- Every *LTL* state formula is also a *LTL* path formula
- For every *LTL* path formula ψ, we have that $\neg\psi, \Box\psi, \Diamond\psi$ and $X\psi$ (read $\neg, \Box, \Diamond, X$ as negation, always, eventually and next respectively) are also path formulas
- For all *LTL* path formulas ϕ, ψ, we have that $\psi\mathscr{U}\phi, \psi \wedge \phi$ and $\phi \vee \psi$ are also *LTL* path formulas.

The set of state formulas is defined as follows :

- For every $p \in$ AP, p is a *LTL* state formula

- For all *LTL* state formulas ϕ and ψ, we have that $\phi \wedge \psi, \phi \vee \psi$ and $\neg\psi$ are also *LTL* state formulas

Given an automaton $\mathscr{A}$ and set of state of automaton Q with atomic predicates AP,

- For a path *LTL* formula ϕ $\langle \mathscr{A}, q \rangle \vDash \phi$ if and only if for all the run $\in \text{Run}(\mathscr{A}, q)$, trace(run) $\vDash \phi$
- $\langle \mathscr{A}, q \rangle \vDash p$ for some $p \in \text{AP}$ if $p \in L(q)$
- $\langle \mathscr{A}, q \rangle \vDash \neg\phi$ if and only if not $\langle \mathscr{A}, q \rangle \vDash \phi$ where ϕ a *LTL* formula over atomic predicates AP
- $\langle \mathscr{A}, q \rangle \vDash \phi \vee \psi$ if and only if $\langle \mathscr{A}, q \rangle \vDash \phi$ or $\langle \mathscr{A}, q \rangle \vDash \psi$
- $\langle \mathscr{A}, q \rangle \vDash \phi \wedge \psi$ if and only if $\langle \mathscr{A}, q \rangle \vDash \phi$ and $\langle \mathscr{A}, q \rangle \vDash \psi$
- Any *LTL* formula ϕ, $\langle \mathscr{A}, q_0 \rangle \vDash \phi$ where q_0 initial state of automaton $\mathscr{A}$ then we denote them as $\mathscr{A} \vDash \phi$.

For given safety regular property expressed in *LTL*, we can construct a non-deterministic finite automaton *NFA* details can be referred to in the model checking book [BK08] chapter 4.

DEFINITION 3.9 *Non Deterministic Finite automaton(NFA)* $\mathscr{M} := \langle S, s_0, \Sigma, \Delta, \mathscr{B} \rangle$*, where S is a finite set of states,* s_0 *an initial state,* Σ *a finite set of action alphabets, finite set of transitions denoted as* $\Delta \subseteq S \times \Sigma \times S$*, and* $\mathscr{B} \subseteq S$ *is the set of bad state for the* NFA. ◇

For the given *NFA* $\mathscr{M}$, Run($\mathscr{M}$), seq(run), and $\mathscr{L}(\mathscr{M})$ can be defined analogously as defined for the automaton $\mathscr{A}$. For any regular safety property expressed in *LTL* ϕ formed over atomic predicate set AP_ϕ, there exist an *NFA* model $\mathscr{M}_\phi := \langle S, s_0, \Sigma_\phi, \Delta_\phi, \mathscr{B}_\phi \rangle$ with $\Sigma_\phi = 2^{\text{AP}_\phi}$ the set of actions, S is the set of states, s_0 is an initial state and $\mathscr{B}_\phi \subseteq S$ is the set of bad states and $s_0 \notin \mathscr{B}_\phi$. The transition set $\Delta_\phi \subseteq S \times \Sigma_\phi \times S$. Constructing such models can be referred to the book [BK08]. It is important to note here that there won't be a unique model $\mathscr{M}_\phi$ for a given specification ϕ, but in terms of the trace behaviour (i.e $\text{Tr}(\mathscr{M}_\phi)$) all the *NFA* model will be same for given *LTL* specifications formed over the atomic predicates set AP.

We denote the set of runs for *NFA* $\mathscr{M}_\phi$ for the given specification ϕ as $\text{Run}(\mathscr{M}_\phi)$ and the set of bad runs as $\text{Run}_{\mathscr{B}_\phi}(\mathscr{M}_\phi) \subseteq \text{Run}(\mathscr{M}_\phi)$ so that $\forall \text{run} \in \text{Run}_{\mathscr{B}_\phi}(\mathscr{M}_\phi)$ such that run $= s_0 \xrightarrow{\text{AP}_{\phi,1}} s_1 ... \xrightarrow{\text{AP}_{\phi,n}} s_n$, where each $\text{AP}_{\phi,i} \subseteq \text{AP}_\phi$ and there exist $s_j \in \mathscr{B}_\phi$ for some $j \in \{1, ..., n\}$ in this run. The sequence of such run $\text{seq(run)} = \text{AP}_{\phi,1}\text{AP}_{\phi,2}....\text{AP}_{\phi,n}$ does not satisfy the specification ϕ (i.e $\text{AP}_{\phi,1}\text{AP}_{\phi,2}....\text{AP}_{\phi,n} \nvDash \phi$). This is by the construction of $\mathscr{M}_\phi$. Note here the specification ϕ characterizes the sequence of the model $\mathscr{M}_\phi$.

EXAMPLE 3.10 *Let's define one such* NFA *model for the given* LTL *specification. Let's take the simplest specification :* $\phi = \Box(\mathbf{a})$ *(always* **a** *should be satisfied in the given trace). The respective model (NFA) is shown in fig. 3.11 (double round on the state* s_1 *indicates that it is a bad state).*

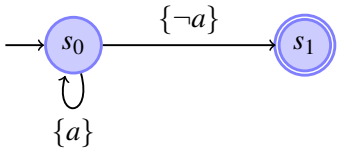


FIGURE 3.11 : An illustrative example of the model for the given Specification

3.3.2 Verification of Monolithic System

In order to check that the automaton $\mathscr{A}$ satisfies a certain regular safety property specification ϕ, that is, whether all the traces in the automaton $\mathscr{A}$ satisfy the specification ϕ, we have to introduce the *synchronized product* [BK08] notion between automata and the specification. We also construct a synchronized product between an automaton $\mathscr{A}$ and an *NFA* model $\mathscr{M}_\phi$ for the corresponding regular safety *LTL* specification ϕ as follows.

DEFINITION 3.10 *Synchronized Product :* $\mathscr{A}^\phi := \mathscr{A} \otimes \mathscr{M}_\phi = \langle Q^\phi, q_0^\phi, \Sigma, \Delta^\phi, L^\phi, AP^\phi (= S) \rangle$ *with bad states* $\mathscr{B}^\phi = Q \times \mathscr{B}_\phi \subseteq Q^\phi$*, where* $Q^\phi = Q \times S$ *(set of states),* $q_0^\phi = (q_0, s_0')$ *(initial state) where* $s_0 \xrightarrow{L(q_0)\downarrow AP_\phi} s_0' \in \Delta_\phi$*,* Σ *is same as from the automaton* $\mathscr{A}$*, the rest will be as follows,*

- *the set of transition* Δ^ϕ

$$\frac{q_1 \xrightarrow{\sigma} q_2 \in \Delta \;\; q_1, q_2 \in Q, s_1 \xrightarrow{L(q_2)\downarrow AP_\phi} s_2 \in \Delta_\phi \;\; s_1, s_2 \in S}{(q_1, s_1) \xrightarrow{\sigma} (q_2, s_2) \in \Delta^\phi}$$

- $L^\phi(q,s) := s$ *for all* $(q,s) \in Q^\phi$.

$\diamond$

In order to do the model checking of monolithic system i.e formal verification of system modelled in an automaton and checking its correctness with respect to the safety specification ϕ, we have the following theorem borrowed from the book [BK08] (theorem **4.19**).

THEOREM 3.1 (Verification of Regular Safety Properties) *Consider the given system model as automaton* $\mathscr{A} := (Q, q_0, \Sigma, \Delta, L, AP)$ *and a regular safety specification* ϕ *over the alphabet AP. The following statements are equivalent*

- $\mathscr{A} \vDash \phi$ *i.e all the traces in* $\mathrm{Tr}(\mathscr{A})$ *satisfy the specification* ϕ *(*$\mathrm{Tr}(\mathscr{A}) \vDash \phi$*).*

- $\mathrm{Tr}(\mathscr{A}) \cap \mathscr{L}(\mathscr{M}_\phi) = \emptyset$*, where* $\mathscr{M}_\phi$ *is a non-deterministic automaton model for the given safety property* ϕ*.*

$\diamond$

From the above theorem, we infer the following lemma using run notation.

LEMMA 3.1 *For given automaton* $\mathscr{A}$ *and specification* ϕ *formed over atomic predicates* $AP_\phi \subseteq AP$*, corresponding NFA model* $\mathscr{M}_\phi$ *and the synchronized product* $\mathscr{A}^\phi = \mathscr{A} \otimes \mathscr{M}_\phi$*, if* $\mathrm{Run}_{\mathscr{B}^\phi}(\mathscr{A}^\phi) = \emptyset$ *then* $\mathrm{Tr}(\mathrm{Run}(\mathscr{A})) \downarrow AP_\phi \vDash \phi$*.*

Proof : Let's say that we have $\mathrm{run} \in \mathrm{Run}_{\mathscr{B}^\phi}(\mathscr{A}^\phi)$, $\mathrm{run} = (q_0, s_0') \xrightarrow{\sigma_1} (q_1, s_1) \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_n} (q_n, s_n)$. Then for some $j \in \{1, .., n\}$, $(q_j, s_j) \in \mathscr{B}^\phi$ (by definition also $s_j \in \mathscr{B}_\phi$ for the specification ϕ *NFA* model $\mathscr{M}_\phi$). For any $\mathrm{run} \in \mathrm{Run}(\mathscr{A} \otimes \mathscr{M}_\phi)$ projected into the run of automaton and specification model as $\mathrm{run} \downarrow \mathscr{A} = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_n} q_n$ and $\mathrm{run} \downarrow \mathscr{M}_\phi = s_0' \xrightarrow{L(q_1)\downarrow \mathrm{AP}_\phi} s_1 \xrightarrow{L(q_2)\downarrow \mathrm{AP}_\phi} \ldots \xrightarrow{L(q_n)\downarrow \mathrm{AP}_\phi} s_n$ respectively and considering the *NFA* run from its initial state we have $\mathrm{run}(\mathscr{M}_\phi, s_0) = s_0 \xrightarrow{L(q_0)\downarrow \mathrm{AP}_\phi} s_0' \xrightarrow{L(q_1)\downarrow \mathrm{AP}_\phi} s_1 \xrightarrow{L(q_2)\downarrow \mathrm{AP}_\phi} \ldots \xrightarrow{L(q_n)\downarrow \mathrm{AP}_\phi} s_n$.

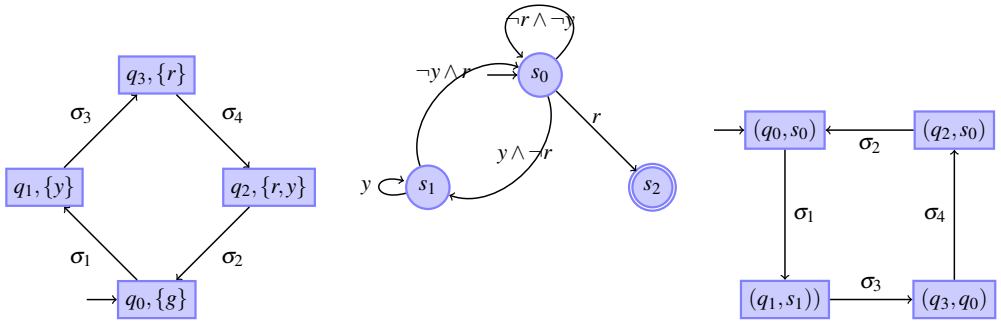
If there is a run from $\mathrm{Run}_{\mathscr{B}^\phi}(\mathscr{A} \otimes \mathscr{M}_\phi)$ then $\mathrm{seq}(\mathrm{run}(\mathscr{M}_\phi, s_0)) \nvDash \phi$ which means $\mathrm{trace}(\mathrm{run} \downarrow \mathscr{A}) \downarrow \mathrm{AP}_\phi \nvDash \phi$. $\diamond$

Here after we denote that given $\mathscr{A} \vDash \phi$ if $\mathrm{Run}_{\mathscr{B}^\phi}(\mathscr{A} \otimes \mathscr{M}_\phi) = \emptyset$ (which also means that for all $\mathrm{trace} \in \mathrm{Tr}(\mathrm{Run}(\mathscr{A})) \downarrow \mathrm{AP}_\phi$ $\mathrm{trace} \vDash \phi$).

Let us see one such example, the following German traffic light protocol example taken from the book [BK08].

EXAMPLE 3.11 (German Traffic Light Model and Specification) *Let's take a specification* $\phi := \Box(\neg(\neg r \wedge \neg y \Rightarrow Xr))$ *(read r, y, g as red, yellow and green respectively). This specification indicates that there should not be two consecutive red signal states and red signal state should be preceded by yellow signal state. One such* NFA *for the given specification* ϕ *is constructed and denoted by* $\mathscr{M}_\phi$ *in fig. 3.12. The typical German traffic light routing is modeled as an automaton and denoted as* $\mathscr{A}$*, the respective synchronized product is constructed as* $\mathscr{A} \otimes \mathscr{M}$*. Note that in the final synchronized product, there is no double rectangle states (i.e there is no state of the form* (q, s_2) *in* $\mathscr{A} \otimes \mathscr{M}_\phi$ *that can be reached for any state* $q \in Q$ *from the automaton* $\mathscr{A}$*) which indicates that the German traffic light system model does satisfy the specification* ϕ*.*

Now we will move on to extend the verification of distributed system in the following section.



(a) $\mathscr{A}$ and $\mathscr{M}_\phi$ (b) $\mathscr{A} \otimes \mathscr{M}_\phi$

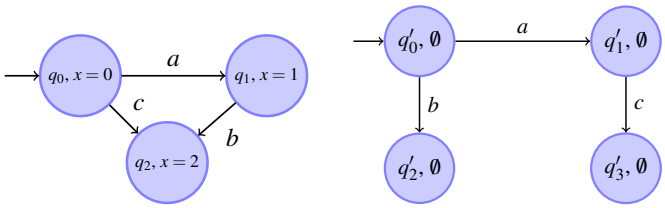
FIGURE 3.12 : Synchronized product construction of the automaton and the given NFA Model

3.3.3 Formal Verification of Distributed System

For a given distributed system *DS*, we model it either by the composition of automata as mentioned in the section 3.1 or generating such composition of automaton by translating the written distributed model description in symbolic transition system as mentioned in section 3.2. For both cases, we have to carefully introduce a set of predicates which is suitable for the kind of requirements we expect from the designed distributed system (please refer to both sections 3.1 and 3.2). Finally express the distributed system requirements as mentioned in section 3.3. In general, when we model the system as either automaton or symbolic transition system, we have to carefully construct system model and specification otherwise we end up either checking the irrelevant specification or checking the relevant specification to the not correctly modelled system or maybe both.

Let say that we modelled the given distributed system as the composition of automaton $\mathscr{A}_{\text{DS}} := \mathscr{A}_1 \parallel \mathscr{A}_2 ... \parallel \mathscr{A}_n$ and expressed the set of requirements as *LTL* specification ϕ, in order to do verification process, what we have to do is checking all the traces of the composition of automaton satisfies the specification ϕ. In order to do that we have to check all the traces in $\text{Tr}(\text{Run}(\mathscr{A}_{\text{DS}} \otimes \mathscr{M}_\phi))$ and its satisfaction to the expressed specification ϕ (i.e $\forall \text{tr} \in \text{Tr}(\text{Run}(\mathscr{A}_{\text{DS}} \otimes \mathscr{M}_\phi))$, check whether $\text{tr} \vDash \phi$ or $\text{tr} \vDash \neg\phi$). If all the traces do satisfy the specification (i.e $\forall \text{tr} \in \text{Tr}(\text{Run}(\mathscr{A}_{\text{DS}} \otimes \mathscr{M}_\phi))\, \text{tr} \vDash \phi$) then the distributed system *DS* satisfy the expressed requirements. If there is a trace which violate the specification (i.e $\exists \text{tr} \in \text{Tr}(\text{Run}(\mathscr{A}_{\text{DS}} \otimes \mathscr{M}_\phi))\, \text{tr} \vDash \neg\phi$), we found the counter example (trace and corresponding run) which violate the expressed specification, hence the *DS* failed to satisfy the requirement ϕ. More importantly if the counter example produced, it gives the reason why the given *DS* failed to satisfy the requirements, which can be used to refine the system or correct the system (i.e redesigning it) manually.

EXAMPLE 3.12 *Let's give an imaginative distributed system denoted by* $\mathscr{A} = \mathscr{A}_1 \parallel \mathscr{A}_2$ *as in the figure 3.13. The specification we are interested is the system never reaches the state which assigns the variable value to 2 i.e* $\phi := \square\neg(x = 2)$ *Here, the given composition of automaton*



(a) $\mathscr{A}_1$ (b) $\mathscr{A}_2$

FIGURE 3.13 : An imaginative distributed system for Illustration purpose

3.13 the only possible trace of this $\mathscr{A} = \mathscr{A}_1 \parallel \mathscr{A}_2$ *composition of automaton is for the run* $\text{run} = (q_0, q'_0) \xrightarrow{a} (q_1, q'_1)$ *which is* $\text{tr} = ((x=0), \emptyset)\,((x=1), \emptyset)$*, and it does satisfy the specification* ϕ*.* $\diamond$

In case, we model the given distributed system *DS* in terms of *SCTS* $\mathscr{T}_{\text{DS}} := \|_{i \in [1,..,m]}\ \mathscr{T}_i$, then for each $\mathscr{T}_i$ generate the $[\mathscr{T}_i]$ and by the relation of semantics representation of symbolic communicating transition system with automata ($\mathscr{A} = [\mathscr{T}_{\text{DS}}]$) check for all traces in the $\text{Tr}(\text{Run}(\mathscr{A}))$ satisfy the specification (i.e $\forall \text{tr} \in \text{Tr}(\text{Run}([\mathscr{T}_{\text{DS}}]))$, $\text{tr} \vDash \phi$). Note by this method, we can only handle the finite state space size of the generated $[\mathscr{T}_{\text{DS}}]$ from the given symbolic communicating transition system $\mathscr{T}$.

In order to formal verification of distributed system, we can use the monolithic technique by modifying the labelling function for the distributed case as $L(q_1, q_2, ..q_n) = \cup_{i \in \{1,..,n\}} L(q_i)$ instead of earlier notation $L(q_1, q_2, ..q_n) = (L_1(q_1), ..., L(q_n))$. Using this definition, we express the distributed specification in *LTL* and verify the *DS* as mentioned in *Verification of Monolithic System*. By doing a monolithic verification to the distributed system (or any concurrent system) is highly inefficient reason is state space explosions (explained in detail in the next chapter). To avoid such state space explosions and verify the *DS* in efficient way there are couple of sophisticated techniques exists in the model checking literature, some of the techniques are mentioned in next chapter.

3.4 Control Synthesis of Distributed system

Discrete Control theory, introduced in the 80's by Ramadge and Wonham [WR88] is now a well-established theory (see [CL08] for a detailed review). Compared to model-checking, the discrete supervisory control problem is to correct the model of the system w.r.t. some requirements. This theory allows the use of constructive methods ensuring, a priori and by means of control, required properties on a system's behavior. Roughly speaking, given a model $\mathscr{A}$ of the system (that is supposed to correctly represent the behaviour of an implementation) and some requirements ϕ (modelled as formal properties expressed in *LTL* or *CTL* w.r.t. the specification), a supervisor $\mathscr{S}$ must be derived by various means such that the resulting behavior of the closed-loop system meets requirements.

Given a model of the system $\mathscr{A}$ and the supervisor $\mathscr{S}$ for the system such that $\mathscr{A}^{\mathscr{S}}$ satisfies the required property ϕ, the closed loop system $\mathscr{A}^{\mathscr{S}}$ here means that $\mathscr{S}$ captures the entire set of possible behaviors of the system $\mathscr{A}$ and it disables some controllable actions and observing all uncontrollable actions such that the closed loop system can be satisfies the property ϕ.

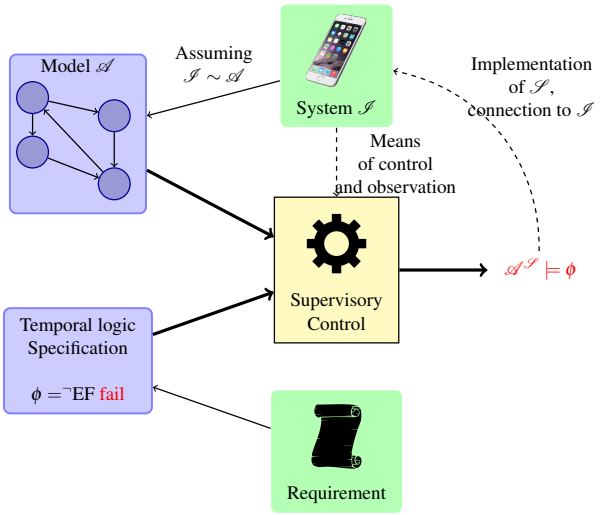


FIGURE 3.14 : Discrete Supervisory Control principle

It is assumed that some actions of the system can be controlled (i.e. can be disabled) based on the observation one can have on the system, while some others, called uncontrollable, cannot be prevented from occurring but observable. In this document, we assume that the model of the system is completely observable and that only a part of the actions are controllable meaning these can be disabled by a supervisor.

The observations made by the supervisor can be a subset of the actions performed by the system or an estimation of the global system states in which it can be. This is the latter choice we shall consider in the sequel. A supervisor $\mathscr{S}$ takes the decision of disabling some controllable events based on this observation as illustrated in figure 3.15.

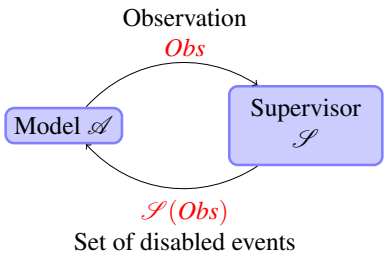


FIGURE 3.15 : Closed-loop Feedback Control

Supervisory control

This theory allows the use of constructive methods ensuring, a priori, and by means of control, required properties on a system's behavior. Given a model $\mathscr{A}$ of a system (that is supposed to correctly represent the behavior of an implementation) and a *specification* ϕ (modeled as formal property expressed in *LTL* or *CTL* (computation Tree Logic) w.r.t. the specification), a controller $\mathscr{S}$ must be derived by various means such that the resulting behavior of the closed-loop system meets requirements.

3.4.1 Control Synthesis of Finite State System

For the elementary discrete event system and synthesizing supervisors are introduced by first defining the system in terms of finite discrete event system (or finite state machine FSM) rather than automaton, automaton model are involved with clear specification of atomic predicates, but FSM are denoted without atomic predicates rather it is system states are denoted with either safe or unsafe for the system to reach. The systems actions set are partitioned into controllable and uncontrollable events, further going into the details, we have to partition the system actions into controllable and uncontrollable and set of states into observable and non-observable sets. In this entire document we partition the system actions into controllable and uncontrollable sets, and assume all the system states and actions are observable.

Synthesis of Finite State Machine We first describe the synthesis of simplest finite state machine here and gradually we will develop the theory based on the existing literature for the composition of many finite state synchronous system later in this sections.

DEFINITION 3.11 *We will define a finite state discrete event system as automaton $\mathscr{A}$ (finite state machine-*FSM*) with minor modification. $\mathscr{A} = \langle Q, q_0, \Sigma, \Delta, L, AP\rangle$ with $AP = \{\text{Bad}\}$, where Q a finite set of states, q_0 an initial state, Σ a finite set of action alphabets such that $\Sigma = \Sigma_c \cup \Sigma_{uc}$, Σ_{uc} denotes the set of uncontrollable actions, Σ_c denotes the set of controllable actions ($\Sigma_c \cap \Sigma_{uc} = \emptyset$), $\Delta \subseteq Q \times \Sigma \times Q$ a finite set of transition relation, and $\mathscr{B} \subseteq Q$ is $\{q \mid L(q) = \{\text{Bad}\}\}$.* ◇

For simplicity in the sequel we shall denote $\mathscr{A}$ by $\langle Q, q_0, \Sigma, \Delta, \mathscr{B}\rangle$ when it is clear from the context. In the definition of *FSM*, we seek for a supervisor such that behaviour of $\mathscr{A}$ is limited by disabling some of the controllable actions such that $\mathscr{A}$ won't reach (or enter) a bad state (i.e a state $q \in Q$ where $L(q) = \{\text{Bad}\}$).

DEFINITION 3.12 *A state-based supervisor $\mathscr{S}$ for a FSM $\mathscr{A}$ is a function $\mathscr{S} : Q \to 2^{\Sigma_c}$ s.t $\mathscr{S}$ blocks the controllable actions $\mathscr{S}(q) \subseteq \Sigma_c$ (i.e disable) at system state $q \in Q$.* ◇

DEFINITION 3.13 *For the given FSM $\mathscr{A}$ and the state based supervisor $\mathscr{S}$, the controlled FSM $\mathscr{A}^{\mathscr{S}}$ is an automaton (FSM). $\mathscr{A}^{\mathscr{S}} = \langle Q, q_0, \Sigma, \Delta', \mathscr{B} \rangle$ where $\Delta' \subseteq \Delta$, if $\sigma \notin \mathscr{S}(q)$ and $(q, \sigma, q') \in \Delta$ then $(q, \sigma, q') \in \Delta'$.* ◇

EXAMPLE 3.13 *Let's take a simple FSM $\mathscr{A} = \langle Q = \{q_0, q_1, q_2\}, q_0, \Sigma = \{a, b, c\}, \Delta = \{(q_0 \xrightarrow{a} q_1), (q_0 \xrightarrow{c} q_2), (q_1 \xrightarrow{b} q_2)\}, \mathscr{B} = \{q_2\} \rangle$ denoted in figure 3.13. Here all the actions from the set Σ are controllable. For such a FSM, the state based supervisor for $\mathscr{A}$ is a function $\mathscr{S}$ such that*

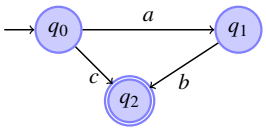


FIGURE 3.16 : A simple FSM

$\mathscr{S}(q_0) = \{c\}, \mathscr{S}(q_1) = \{b\}, \mathscr{S}(q_2) = \emptyset$. So here, for the given FSM, we do have the state based supervisor $\mathscr{S}$ to control it to avoid reaching a bad state. ◇

The computation of such supervisors $\mathscr{S}$ are well-known in the community ; we briefly recall it, and refer to [CL08] for details. In a nutshell, one have to compute the set of states $I_{\text{Bad}} = \text{CoReach}^{\mathscr{A}}_{\Sigma_{uc}}(\text{Bad})$ that can reach in an *uncontrollable way* the set Bad (i.e global bad states $\mathscr{B}$ of the given system $\mathscr{A}$) ; that is, this is the set of states from which, some uncontrollable actions lead the system to Bad independently from the choice of the controllable actions. The supervisor $\mathscr{S}$ is then computed by disabling in any subset $E \in Q$, the events that lead to I_{Bad}. We will show you some illustrative examples in the following texts.

EXAMPLE 3.14 *The following example explains how I_{Bad} is computed from the set of* Bad *states and when the supervisor $\mathscr{S}$ avoids to reach any state of I_{Bad}. The state 1 has to be*

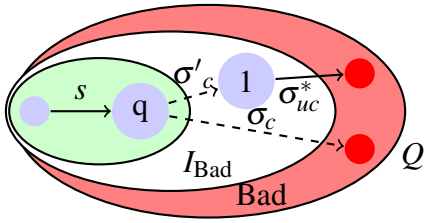


FIGURE 3.17 : Intuition behind I_{Bad} and $\mathscr{S}$ computations

disabled as a bad state can be reached by triggering $\sigma^*_{uc} \in \Sigma^*_{uc}$, so it belongs to I_{Bad}. The dashed transitions represent the events that are disabled by the supervisor $\mathscr{S}$, i.e. $\mathscr{S}(\{q\}) = \{\sigma_c, \sigma'_c\}$ as they may lead to I_{Bad} by triggering one of these two events. ◇

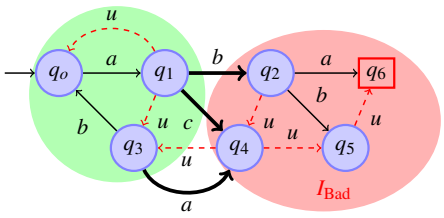


FIGURE 3.18 : example of Control

EXAMPLE 3.15 *To illustrate this, let us consider the following FSM* $\mathscr{A}$*, with* $\Sigma_c = \{a,b,c\}$*,* $\Sigma_{uc} = \{u\}$ *and* $\text{Bad} = \{q_6\}$*.*

It is easy to see that $I_{\text{Bad}} = \{q_2, q_4, q_5, q_6\}$ *as the path*

$$q_2 \xrightarrow{u} q_4 \xrightarrow{u} q_5 \xrightarrow{u} q_6$$

is an uncontrollable path (i.e. q_6 *can be reached from* q_2 *by triggering only uncontrollable events. By supervision* $\mathscr{A}^{\mathscr{S}}$ *is then reduced to the automaton inside the green circle, with* $\mathscr{S}$ *defined as follows :* $\mathscr{S}(q_0) = \emptyset$*,* $\mathscr{S}(q_1) = \{b,c\}$ *and* $\mathscr{S}(q_3) = \{a\}$*.* ◇

We say a valid supervisor exists if and only if $I_{\text{bad}} \neq Q$.

THEOREM 3.2 *Given an Automaton* $\mathscr{A}A = (Q, q_0, \Sigma = (\sigma_c \cup \Sigma_{uc}), \Delta, \mathscr{B})$*, then, the supervisor* $\mathscr{S}$ *such that*

$$\mathscr{S}(q) = \{\sigma \in \Sigma_c \mid (q,\sigma,q') \in \Delta \wedge q' \in I_{\text{bad}}\}$$

is such that $\mathscr{A}^{\mathscr{S}}$ *avoids* $\mathscr{B}$*.* ◇

An important concept in the synthesis of supervisor theory is maximal permissiveness, the fact that the supervisor $\mathscr{S}$ must disable actions in the system model $\mathscr{A}$ only when it is strictly necessary to enforce the closed system $\mathscr{A}^{\mathscr{S}}$ to avoid bad states, meaning that it forces the supervisors to disable the controllable actions as little as possible. An important feature of synthesis supervisor theory of finite system is that there is a unique maximally-permissive supervisor, but when the system model is infinite, the existence of a unique maximal permissive supervisor can not be guaranteed but we can compare any two supervisors $\mathscr{S}_1, \mathscr{S}_2$ to the system model $\mathscr{A}$ with respect to the permissive nature.

DEFINITION 3.14 (State-based) *Given a FSM and two supervisors* $\mathscr{S}_1, \mathscr{S}_2$*.* $\mathscr{S}_1$ *is more permissive than* $\mathscr{S}_2$ *when* $\mathscr{S}_2(q) \subseteq \mathscr{S}_1(q)$ *for all state* $q \in Q$ *of the FSM* $\mathscr{A}$*.* ◇

We call strictly $\mathscr{S}_1$ more permissive than $\mathscr{S}_2$ when $\mathscr{S}_2(q) \subseteq \mathscr{S}_1(q)$ for all state $q \in Q$ and there exist a state $q \in Q$ s.t $\mathscr{S}_2(q) \subsetneq \mathscr{S}_1(q)$.

Permissiveness definition can be easily extended to languages :

DEFINITION 3.15 *Given a FSM $\mathscr{A}$, $\mathscr{S}$ is maximally permissive whenever there does not exist a valid supervisor $\mathscr{S}'$ such that $\mathscr{L}(\mathscr{A}^{\mathscr{S}'}) \subseteq \mathscr{L}(\mathscr{A}^{\mathscr{S}})$.* ◇

3.4.2 Control Synthesis of LTL Safety Specifications

To synthesis the safety regular *LTL* specifications formed over the atomic predicates set AP on the generic automaton $\mathscr{A}$, we have to know the set of states and also the set of traces which violates the specification ϕ, in order to construct supervisor which avoids not just bad states but also all the traces which violates the specification ϕ, we have to build the model for the specification. We will use NFA model for the given specification ϕ and construct the control synthesis for the automaton $\mathscr{A}$ as follows.

Synthesis of safety *LTL* properties. When dealing with *safety regular property* ϕ, as for model checking, we first build the *NFA* model $\mathscr{M}_\phi$ with a subset of system states $\mathscr{B}_\phi \subseteq S$ (state of the model $\mathscr{M}_\phi$) representing the states that do not fulfill the property (i.e. all the runs that do not cross a state in $\mathscr{B}_\phi$ satisfies ϕ). We then perform the synchronized product between $\mathscr{A}$ and $\mathscr{M}_\phi : \mathscr{A}^\phi = \mathscr{A} \otimes \mathscr{M}_\phi$ and we denote by $\mathscr{B}^\phi$ the set of states of the form $Q \times \mathscr{B}_\phi$. A supervisor $\mathscr{S}_\phi$ avoiding $\mathscr{B}^\phi$ can then be computed on $\mathscr{A}^\phi$ and the resulting automaton $(\mathscr{A}^\phi)^{\mathscr{S}_\phi}$ defined as in Definition 3.13 fulfills the property ϕ.

For the given automaton $\mathscr{A}$, and for the specification ϕ, we have NFA model $\mathscr{M}_\phi$, and then we construct the automaton $\mathscr{A}^\phi := \mathscr{A} \otimes \mathscr{M}_\phi$, the main objective of the supervisor $\mathscr{S}_\phi$ is to avoid the co-reach of bad states $\mathscr{B}^\phi = Q \times \mathscr{B}_\phi$.

ALGORITHM 3.1 (Synthesis *LTL* Specification)

- *Given an automaton $\mathscr{A}$ and* LTL *regular safety specification ϕ, construct* NFA *model $\mathscr{M}_\phi = \langle S_\phi, s_0, \Sigma_\phi, \Delta_\phi, \mathscr{B}_\phi \rangle$ (can be constructed automatically, even SPIN software has interface to do this task).*
- *Construct the synchronized automaton $\mathscr{A}^\phi := \mathscr{A} \otimes \mathscr{M} = \langle Q^\phi, q_0^\phi, \Sigma, \Delta^\phi, L^\phi, AP^\phi (= S_\phi) \rangle$ with bad states $\mathscr{B}^\phi = Q \times \mathscr{B}_\phi$.*
- *Compute the co-reach of the bad states $Q \times \mathscr{B}_\phi$ (we denote it as $I_{\text{Bad}} = \mathsf{CoReach}_{\Sigma_{uc}}^{\mathscr{A}^\phi}(\mathscr{B}^\phi)$) by triggering the uncontrollable actions Σ_{uc}.*
- *Compute the supervisor $\mathscr{S}_\phi$ avoiding I_{Bad} states in the automaton $\mathscr{A}^\phi$.*

◇

THEOREM 3.3 *The algorithm 3.1 for constructing the control synthesis $\mathscr{S}$ (if exists) for the specification ϕ for the automaton $\mathscr{A}$, then $(\mathscr{A}^\phi)^{\mathscr{S}_\phi} \models \phi$.* ◇

The proof is trivial. Arguments as follows, supervisor $\mathscr{S}_\phi$ avoids the automaton $\mathscr{A}^\phi$ to reach the bad state $\mathscr{B}^\phi$, if the automaton $(\mathscr{A}^\phi)^{\mathscr{S}_\phi}$ wont enter the bad states $\mathscr{B}^\phi$ then $\text{Run}_{\mathscr{B}^\phi}((\mathscr{A}^\phi)^{\mathscr{S}_\phi}) = \emptyset$ then by lemma 3.1 we have $\text{Tr}(\text{Run}_{\mathscr{B}^\phi}((\mathscr{A}^\phi)^{\mathscr{S}_\phi})) \vDash \phi$ hence the theorem.

We denote $(\mathscr{A}, \mathscr{S}_\phi)$ to refer the automaton $(\mathscr{A}^\phi)^{\mathscr{S}_\phi}$, and this supervisor $\mathscr{S}_\phi : Q \times S_\phi \to 2^{\Sigma_c}$, i.e by knowing the current states of the automaton $\mathscr{A}$ and the model $\mathscr{M}_\phi$, supervisor $\mathscr{S}_\phi$ blocks the set of controllable actions $\mathscr{S}_\phi(q,s) \subseteq \Sigma_c$ for all $(q,s) \in Q \times S_\phi$.

Note that the methodology is very similar to the one used in language-based approach of control synthesis [CL08]. The main difference is that we make use of *LTL* properties versus language-based properties. We thus benefit from the know-how of the model checking community, in particular to model the properties in a high-level setting.

Relation Between Synthesis and Verification Process Verification process is mostly on checking correctness of given expressed system model with respect to the required specifications where-us the synthesis is about computing the controller so that the expressed system model satisfy the given specification.

Synthesis to Verification

In case of synthesis the given finite state machine (FSM) modeled as an automaton $\mathscr{A} = \langle Q, q_0, \Sigma = (\Sigma_c \cup \Sigma_{uc}), \Delta, \mathscr{B} \rangle$ can be equally expressed as an automaton $\mathscr{A} = \langle Q, q_0, \Sigma, \Delta, L, \text{AP} \rangle$ with $\text{AP} = \{\text{Bad}\}$ and we modify the labelling function L as $L(q) = \{\}$ if $q \notin \mathscr{B}$ and $L(q) = \{\text{Bad}\}$ if $q \in \mathscr{B}$ as by definition 3.11.

The redefined automaton $\mathscr{A}$ should not enter into bad state is the requirement in case of synthesis. Such requirement can be expressed in *LTL* safety specification as $\Box(\neg\text{Bad})$. So finding a supervisor $\mathscr{S}$ for FSM automaton $\mathscr{A}$, such that $\mathscr{A}^{\mathscr{S}}$ never enters the bad state $\mathscr{B}$. If there is a valid supervisor, then the resultant FSM $\mathscr{A}^{\mathscr{S}}$ (it is a FSM by definition 3.13) satisfies $\mathscr{A}^{\mathscr{S}} \vDash \Box(\neg\text{Bad})$ by theorem 3.2.

Verification to Synthesis

In converse, suppose we have an automaton $\mathscr{A} = \langle Q, q_0, \Sigma, \Delta, \text{AP}, L \rangle$ and its action set Σ can be partitioned into controllable and uncontrollable action sets. We want the automaton $\mathscr{A}$ to satisfy the specification ϕ (*LTL* specification formed over the atomic predicates AP). For any safety *LTL* property, we can construct synchronized product of the automaton $\mathscr{A}$ and the model $\mathscr{M}_\phi$ for the given specification ϕ as by definition 3.10. In the automaton $\mathscr{A}^\phi$, the state space can be partitioned as $Q^\phi = (Q \times (S \setminus \mathscr{B}_\phi)) \sqcup (Q \times \mathscr{B}_\phi)$ indicating that in the automaton (i.e in the synchronized product) $\mathscr{A}^\phi$ the states $(q,s) \in (Q \times (S \setminus \mathscr{B}_\phi))$ not violating the specification ϕ and the states $(q,s) \in (Q \times \mathscr{B}_\phi)$ violating the specification.

Now, use the $\mathscr{A}^{\phi}$ to construct *FSM* (say $\mathscr{F}$ for not to confuse the notation) for the given synchronized product $\mathscr{A}^{\phi} = \mathscr{A} \otimes \mathscr{M}_{\phi}$ by excluding the labelling function L and the atomic predicates AP, and defining the bad set of states $\mathscr{B} = (Q \times \mathscr{B}_{\phi})$.

A supervisor $\mathscr{S}$ is build to avoid $I_{\text{Bad}} = \mathsf{CoReach}^{\mathscr{F}}_{\Sigma_{uc}}(\mathscr{B})$ for the *FSM* $\mathscr{F}$, if there exists $\mathscr{S}_{\phi}$ then we have $\mathscr{F}^{\mathscr{S}_{\phi}} \vDash \phi$ (i.e constructed FSM never enters the defined bad state hence all the trace from this finite state machine $\mathscr{F}^{\mathscr{S}_{\phi}}$ satisfies the specification ϕ). The same constructed supervisor $\mathscr{S}_{\phi}$ used in the automaton $\mathscr{A}^{\phi} \equiv \mathscr{A} \otimes \mathscr{M}_{\phi}$ by disabling the controllable action as defined by the supervisor $\mathscr{S}_{\phi}$, then we have $(\mathscr{A}^{\phi})^{\mathscr{S}_{\phi}} \vDash \phi$. That is the automaton $\mathscr{A} \otimes \mathscr{M}_{\phi}$ controlled by the supervisor $\mathscr{S}_{\phi}$ never enters the state $Q \times \mathscr{B}_{\phi}$ then we have all the traces $\text{Tr}(\text{Run}(\,(\mathscr{A} \otimes \mathscr{M}_{\phi})^{\mathscr{S}_{\phi}} \downarrow \mathscr{A})) \vDash \phi$ (i.e all the traces of run from $\text{Run}(\,(\mathscr{A} \otimes \mathscr{M}_{\phi})^{\mathscr{S}_{\phi}}\,)$ projected to the automaton $\mathscr{A}$ satisfy the specification ϕ)

3.4.3 Synthesis of Finite State Distributed System

We will see the simplest synthesis of distributed systems with synchronous communications by introducing the finite discrete event system way of modelling the distributed system *DS* in this section and more involved synthesis scheme model and property will be developed in later sections and also in chapter 4.

In case of defining the composition of automaton for the control synthesis of distributed system which is communicating in synchronous way (there is no queue concept) we have to define the controllable and uncontrollable events and also the actions shared between various sub-systems. We will revisit the composition of automaton for the control synthesis case as follows,

DEFINITION 3.16 *Given set of finite state machines $\mathscr{A}_1, ..., \mathscr{A}_n$, where each $i \in \{1, .., n\}$ $\mathscr{A}_i = \langle Q_i, q_{0,i}, \Sigma_i, \Delta_i, \mathscr{B}_i \rangle$*
the composition of these FSM's is $\mathscr{A} = (\,||_{i \in \{1,..,n\}} \mathscr{A}_i) = \langle Q, q_0, \Sigma, \Delta, \mathscr{B} \rangle$,
where Q, q_0, Σ, Δ as defined in definition 3.3

- *Each subsystem $\mathscr{A}_i$ partition the set of actions Σ_i into controllable and uncontrollable actions i.e $\Sigma_i = \Sigma_{c,i} \sqcup \Sigma_{uc,i}$ ($\Sigma_{c,i} \cap \Sigma_{uc,i} = \emptyset$)*
- *$\mathscr{B} := \prod_{i \in [1,..,n]} \mathscr{B}_i$ where $\mathscr{B}_i \subseteq Q_i$.*

$\diamond$

Note : In the above definition, we can consider the following assumptions depends on the system we model and synthesis.

- There can be common actions shared with more than two different sub *FSM*'s and if any one of the *FSM*'s trigger that common action in the composition of *FSM*'s all the sub

FSM's which has this action should trigger the same action (well that is because of the composition of automata definition not because of the limitation of distributed system).

- Suppose for the given action $\sigma \in \Sigma_{i_1}, \Sigma_{i_2}, ...\Sigma_{i_l}$ for $i_1, i_2, ..., i_l \in \{1, .., n\}$ then it is controllable by only one automaton i.e for the common action (synchronous action) σ shared among the processors $\{i_1, i_2, ..., i_l\}$ then it is at-most controllable by only one processor. This is the case if the distributed system is reactive in nature, i.e the system sends command actions - more like sending a command message in a synchronous (rendezvous) queue.

- To define global controllable actions, if there is a subsystem which can control an action $\sigma \in \Sigma$ then that action can be considered as a global controllable action. Because if the local subsystem controls that action (i.e disable) then this particular action σ can't be triggered by any of the subsystems because of the composition of the automaton. Moreover if we consider that any shared actions between a set of subsystems from $\{\mathscr{A}_1, ...\mathscr{A}_n\}$ it can be at-most controllable by one subsystem. That is to say if $\sigma \in \Sigma_{c,i}$ for some $i \in \{1, .., n\}$ then for all $j \neq i$ $\sigma \notin \Sigma_{c,j}$. The global controllable actions can be defined as $\Sigma_c = \cup_{i \in \{1,..,n\}} \Sigma_{c,i}$ and global uncontrollable actions is $\Sigma_{uc} = \Sigma \setminus \Sigma_c$.

- Although, there can be more than two automata can synchronize a shared action, but in real system I suppose that common action is triggered by (controlled) one processor and all other sub-processors has that action will do handshake and execute that common actions but it requires atomicity of executing that handshake by all the sub-processors, maybe we can redefine the above definition by saying at-most only two process can share any actions in the alphabet Σ. This is the case for the reactive system.

When it comes to the set of *FSM* as in the definition 3.16, finding the local supervisor $\mathscr{S}_i$ controlling *FSM* $\mathscr{A}_i$ knows about *FSM* $\mathscr{A}_i$ state, but in the presence of many *FSM*'s and the common action triggered among many *FSM*'s, the local supervisor $\mathscr{S}_i$ has to estimate the current global state i.e $(q_1, q_2...q_n) \in Q$ (not just $q_i \in Q_i$). In order to do that, each local supervisor $\mathscr{S}_i$ should have a global state estimator to have the knowledge on the current position of the global system, for that we are going to introduce the state estimator inspired from the work [Kal+14].

DEFINITION 3.17 *A* State estimate *procedure E_j for a specific local FSM $\mathscr{A}_j$ of given system $\mathscr{A} = ||_{i \in \{1,..,n\}} \mathscr{A}_i$ is a function $E_j : 2^Q \to 2^Q$ such that for any $Q' \subseteq Q$ set of states, $E_j(Q') \subseteq Q$ which contains the possible current global state of the system $\mathscr{A}$.* ◇

Here after for the sake of simplicity, we use E_j for the current global states for the automaton $\mathscr{A}_j$ at any given instant instead of explicitly mention it as $E_j(Q')$ where Q' is the previous global state estimation for the subsystem $\mathscr{A}_j$.

DEFINITION 3.18 *A local supervisor $\mathscr{S}_i$ is a function $\mathscr{S}_i := E_i \times I_{\text{bad}} \to 2^{\Sigma_{i,c}}$, such that E_i is the state estimate for a system $\mathscr{A}_i$ (according to $\mathscr{S}_i$) which estimate the global state $Q' \subseteq Q$, and $I_{\text{bad}} \subseteq Q$ is co-reach of bad states $\mathscr{B}$ i.e $\mathscr{B}_1 \times ...\mathscr{B}_i \times ...\mathscr{B}_n$ reached by triggering the uncontrollable global action set Σ_{uc} and computed by $I_{\text{bad}} = \mathsf{CoReach}^{\mathscr{A}}_{\Sigma_{uc}}(\mathscr{B}_1 \times ...\mathscr{B}_i \times ...\mathscr{B}_n)$. The function $\mathscr{S}_i(E_i, I_{\text{bad}})$ are the set of controllable actions are blocked for $\mathscr{A}_i$ for the given global state estimation E_i.* ◇

Note, the above local supervisor aim is to compute the possible global current state by using the state estimator E_i and co-reach of global bad states $\mathscr{B}$ i.e $\mathsf{CoReach}^{\mathscr{A}}_{\Sigma_{uc}}(\mathscr{B}_1 \times ...\mathscr{B}_i \times ...\mathscr{B}_n)$. We have the assumption of if $\sigma \in \Sigma_{c,i}$ for some $i \in \{1,..,n\}$ then for all $j \neq i$ $\sigma \notin \Sigma_{c,j}$. The set of global controllable action set $\Sigma_c = \cup_{i\in\{1,..,n\}}\Sigma_{c,i}$ and global uncontrollable action set $\Sigma_{uc} = \Sigma \setminus \Sigma_c$.

In our setting, we explicitly computing a local supervisor for each *FSM* involved in the composition of automata $\mathscr{A} = ||_{i\in\{1,..,n\}}\mathscr{A}_i$, the earlier work in the literature has the same notation with different control architecture. For example [GM05] computes one global supervisor for the composition of automata, their technique is almost the same as the technique mentioned here. The main difference is in the control architecture of given composition of system and the assumption about the shared common actions between subsystems they have the assumption that for any two subsystem $i, j \in \{1,..,n\}$ $\Sigma_{c,i} \cap \Sigma_{uc,j} = \emptyset$ that is if a subsystem $\mathscr{A}_i$ control an action σ and it is also an action in subsystem $\mathscr{A}_j$ that should be controllable. They find one global supervisor by first computing product of co-reach of bad states $I_{\text{bad}} = I_{\text{bad}_1} \times ...I_{\text{bad}_n}$ where $I_{\text{bad}_i} = \mathsf{CoReach}^{\mathscr{A}_i}_{\Sigma_{uc,i}}(\mathscr{B}_i)$ and the detail architecture depicted in the figure 3.19. Since the global supervisor is aware of the current local state of all subsystem in the given FSM $\mathscr{A}$, it don't need the local state estimator, and directly control each FSM involves in the automata $(\mathscr{A})^{\mathscr{S}}$.

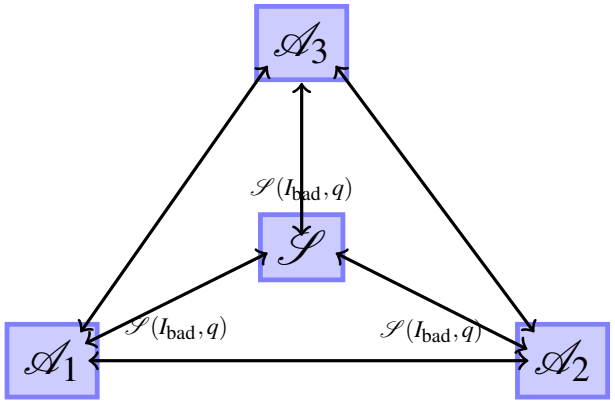


FIGURE 3.19 : A Centralized Control architecture for a Synchronous distributed system

DEFINITION 3.19 *Given composition of FSM $\mathscr{A} = ||_{i\in\{1,..,n\}}\mathscr{A}_i$, the set of distributed supervisors $\mathscr{S} := \{\mathscr{S}_i \mid i \in \{1,..,n\}\}$, and for each $\mathscr{A}_i$ finite state machine, the local supervisor $\mathscr{S}_i$ is*

a function $\mathscr{S}_i : E_i \times I_{\text{bad}_i} \to 2^{\Sigma_{c,i}}$. *The resulting supervisor* $\mathscr{S} = (\mathscr{S}_1, ..\mathscr{S}_n)$ *with global system* $\mathscr{A}$ *is* $\mathscr{A}^{\mathscr{S}} := ||_{i\in\{1,..,n\}} \mathscr{A}_i^{\mathscr{S}_i}$ *again a FSM.* ◇

PROBLEM 3.1 Distributed State avoidance control problem : *Given a composition of finite state machine* $\mathscr{A} = ||_{i\in\{1,..,n\}} \mathscr{A}_i = \langle Q, q_0, \Sigma, \Delta, \mathscr{B}\rangle$ *and the set of forbidden states* $\mathscr{B}$, *the distributed state avoidance control problem consists in synthesizing a distributed supervisors* $\mathscr{S} := \{\mathscr{S}_i \mid i \in \{1,..,n\}\}$ *such that each controlled execution of the system* $\mathscr{A}_i$ *under the control of* $\mathscr{S}_i$ *avoids the forbidden states* $\mathscr{B}$. ◇

Let us give one such state estimate procedure for a local FSM $\mathscr{A}_i$ of given global system $\mathscr{A} = ||_{i\in\{1,2...n\}} \mathscr{A}_i$, we recall that the reachable and co-reachable for a given set $Q' \subseteq Q$ and for the action alphabets $\Sigma' \subseteq \Sigma$ can be defined in the same way as in equations 3.1 and 3.2 respectively. We state the pseudo algorithm below for the state estimator.

The basic idea here it will be finding the local supervisor $\mathscr{S}_i$ for each subsystem $\mathscr{A}_i$ so that the global bad state $\mathscr{B}$ is avoided in the global system $\mathscr{A} = ||_{i\in\{1,..,n\}} \mathscr{A}_i$ the simplistic view is presented in the figure 3.20.

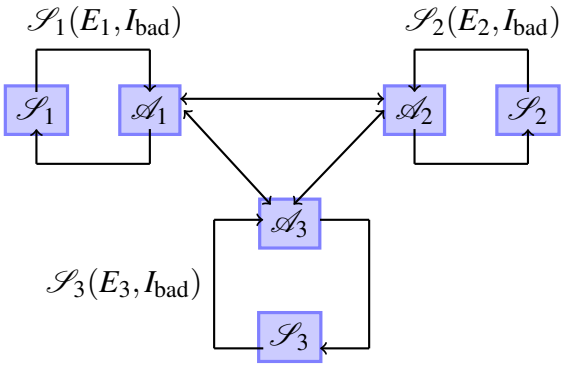


FIGURE 3.20 : A Decentralized Architecture for a Synchronous Distributed System

Here after we refer there exists a valid supervisors $\mathscr{S} = \{\mathscr{S}_1, ..., \mathscr{S}_n\}$ for *FSM* (i.e the composition of automata) $\mathscr{A} = ||_{i\in\{1,..,n\}} \mathscr{A}_i$ if $\{(q_{0,1}, ..q_{0,n})\} \in I_{\text{bad}}$, where $(q_{0,1}, ..q_{0,n})$ is an initial state of $\mathscr{A}$.

ALGORITHM 3.2 (State Estimator) *We compute the state estimate procedure* E_j *for the local FSM* $\mathscr{A}_j$ *of given system* $\mathscr{A}$ *recursively as follows,*

- *At the starting instant,* $E_j(\emptyset) = \{(q_{0,1}, .., q_{0,n})\}$ *which contains only the initial state of the system* $\mathscr{A}$,
- *for an internal actions alphabets in* $\mathscr{A}_j$ *is fired i.e* $\sigma \in \Sigma_j \setminus \Sigma$ *then for given* $Q' \subseteq Q$, $E_j(Q') = \{ (q_1, q_2, .., q'_j, .., q_n) \mid (q_1, q_2, .., q_j, .., q_n) \in Q' \wedge (q_j, \sigma, q'_j) \in \Delta_j \}$

- *for a synchronized transition* $\sigma \in \Sigma_j$ *shared in the set* $\Sigma_{i_1} \cap \Sigma_{i_2} .. \cap \Sigma_{i_k}$ *for* $\{i_1, i_2, .., i_k\} \subseteq \{1, .., n\}$ *and for given* $Q' \subseteq Q$
  take $A = \{(q_1, q_2, .., q'_{i_1}, .., q'_{i_2}, ..q'_{i_k}, .., q_n) \mid (q_1, q_2, .., q_{i_1}, .., q_{i_2}, ..q_{i_k}, .., q_n) \in Q' \wedge (q_l, \sigma, q'_l) \in \Delta_l \forall l \in \{i_1, i_2, .., i_k\}\}$,
  $E_j(Q') = \mathsf{Reach}^{\mathscr{A}}_{\Sigma \setminus \Sigma_j}(A)$.

$\diamond$

For the sake of notation simplicity, we refer E_j for the current global state estimation of the given local system $\mathscr{A}_j$ without explicitly mentioning that $E_j(Q')$ where Q' is the previous instant of the global state estimated by the state estimator E_j for the same local system.

Note that for any given state estimator E_i, all the possible states in this estimation, the i-th co-ordinate will be the same and equal to current state of the automaton $\mathscr{A}_i$.

Now we will give one simple algorithm for computing the local supervisor $\mathscr{S}_i$ for the given automaton $\mathscr{A}_i$ of the composed system $\mathscr{A} = ||_{i \in [1,..,n]} \mathscr{A}_i$.

ALGORITHM 3.3 (Local Supervisor) *Given composition of automaton (*FSM*)* $\mathscr{A} = ||_{i \in [1,...,n]} \mathscr{A}_i$, *finding the local supervisor* $\mathscr{S}_j$ *for the automaton* $\mathscr{A}_j$ *with state estimator* E_j *as mentioned in algorithm 3.2 with co-reach of global bad state* $I_{\text{bad}} = \mathsf{CoReach}^{\mathscr{A}}_{\Sigma_{uc}}(\mathscr{B}_1 \times ... \mathscr{B}_j \times ... \mathscr{B}_n)$, *the local supervisor is computed as follows,*

- *At any given instant, when the state estimator* $E_j \subseteq Q$ *(subset of global state) is non-empty when it is intersected with* I_{bad} *i.e* $E_j \cap I_{\text{bad}} \neq \emptyset$ *and the local state of* $\mathscr{A}_j$ *is* q_j *then* $\mathscr{S}_j(E_j, I_{\text{bad}}) = \Sigma_{c,j} \cap \{\sigma \mid (q_j, \sigma, q'_j) \in \Delta_j \wedge \sigma \in \Sigma_{c,j}\}$ *(i.e all the controllable action* $\Sigma_{c,j}$ *possible from the state* q_j*),*

- *If* $E_j \cap I_{\text{bad}} = \emptyset$ *then*

$$\mathscr{S}_j(E_j, I_{\text{bad}}) = \{\sigma \mid \sigma \in \Sigma_{c,j} \wedge \mathsf{Post}^{\mathscr{A}}_{\sigma}(E_j) \cap I_{\text{bad}} \neq \emptyset\}$$

  (i.e all the controllable actions leads to the sate I_{bad} *from the state in state estimation* E_j*)*

$\diamond$

THEOREM 3.4 (Control Synthesis for finite distributed system) *Given the composition of FSM* $\mathscr{A} = ||_{i \in \{1,...,n\}} \mathscr{A}_i$, *using the algorithm 3.3 there exists a valid supervisors* $\mathscr{S} = \{\mathscr{S}_1, .., \mathscr{S}_n\}$. *Then the system* $||_{i \in \{1,..,n\}} \mathscr{A}_i^{\mathscr{S}_i}$ *never enters the bad state* $\mathscr{B}$ *and solves the problem 3.1.* $\diamond$

The argument as follows, Suppose let say that the computed supervisors with their respective state estimator reaches the bad state $\mathscr{B}$.

Take a minimum length run
$\text{run} = (q_{0,1}, ..q_{0,n}) \xrightarrow{\sigma_1} .... \xrightarrow{\sigma_l} (q_{l,0}, ..q_{l,n}) \in \text{Run}(\mathscr{A}^{\mathscr{S}}) = \text{Run}(||_{i \in \{1,..,n\}} \mathscr{A}_i^{\mathscr{S}_i})$ such that $(q_{l,0}, ..q_{l,n})$ a first state to enter into I_{bad}.

claim σ_l should be an uncontrollable action i.e $\sigma \in \Sigma_{uc}$. Suppose let say that $\sigma_l \in \Sigma_{c,i}$ for some $i \in \{1,..,n\}$ i.e $\sigma \in \Sigma_c$ then we will have $(q_{l-1,0},..q_{l-1,n}) \in I_{\text{bad}}$ which contradicts the assumption that run a minimum length run which enters the set I_{bad}

By the above **claim** we have $\sigma \in \Sigma_c$ more precisely it is controllable by one such subsystem, say $\mathscr{A}_j$. If that is the case then the supervisor $\mathscr{S}_j$ knowing the state estimation E_j such that $((q_{l-1,0},..q_{l-1,n})) \in E_j$ then the supervisor $\mathscr{S}_i$ would have blocked the controllable action σ_l and if it is controllable action by the subsystem $\mathscr{A}_j$ it would not be controllable by any other subsystem $\mathscr{A}_i$ for any $i \neq j \in \{1,...,n\}$ and because of the composition definition, if that particular action σ disabled in the subsystem $\mathscr{A}_j$ then it can't be triggered by any other subsystem. $\diamond$

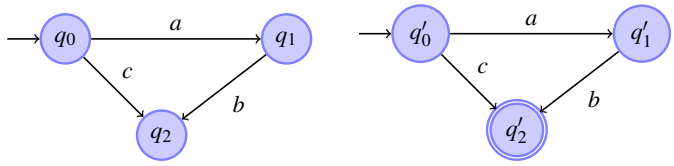
Note : From the argument of above theorem, it clearly indicates that the produced supervisor might be blocking but will not let the system $\mathscr{A}$ controlled by supervisor $\mathscr{S}$ will reach global bad state $\mathscr{B}$.

The proof of this theorem is trivial, since each local supervisor $\mathscr{S}_i$ with their global state estimator E_i avoids to enter the bad state I_{bad} at any given instant, then the global bad state $\mathscr{B}$ never be reached in the composition of global system $\mathscr{A}$ controlled by set of supervisors $\{\mathscr{S}_1, \mathscr{S}_2, ..., \mathscr{S}_n\}$.

The control of distributed finite state machine as mentioned in our texts is not new, for instant the paper [GM04] discuss in depth of this problem and solves without using state estimator for each local system $\mathscr{A}_j$ for the given global system $\mathscr{A} = ||_{i\in\{1,..,n\}}\mathscr{A}_i$. Although their settings is more general in one sense and less general in another sense. The settings in [GM04] is that common actions between any two subsystem is empty but in our case the distributed system communicating in synchronous way and we allow the non-empty actions between any collection subsystems.

It may be useful to study the permissive nature between our control synthesis of distributed system with respect to the control synthesis of the work [GM04]. We might consider this possible direction in our future work.

EXAMPLE 3.16 *Let's take an hypothetical distributed system modelled as FSM denoted in the figure 3.21. In this model* $\mathscr{A}_1 = \langle Q_1 = \{q_0,q_1,q_2\}, q_0, \Sigma_1 = \{a,b,c\}, \Delta_1 = \{(q_0 \xrightarrow{a} q_1),(q_1 \xrightarrow{b} q_2),(q_2 \xrightarrow{c} q_0)\}, \mathscr{B}_1 = \emptyset\rangle$ *and* $\mathscr{T}_2 = \langle Q_2 = \{q'_0,q'_1,q'_2\}, q'_0, \Sigma_2 = \{a,b,c\}, \Delta_2 = \{(q'_0 \xrightarrow{a} q'_1),(q_1 \xrightarrow{b} q'_2),(q'_2 \xrightarrow{c} q'_0)\}, \mathscr{B}_2 = \{q'_2\}\rangle$. *Take here* $\Sigma_{1_c} = \{b,c\}$ *(controllable actions for FSM* $\mathscr{A}_1$*) and* $\Sigma_{2_c} = \{c\}$ *(controllable actions for FSM* $\mathscr{A}_2$*). In this example, the global bad states are* $Q_1 \times \{q'_2\}$. *Note : In this distributed FSM, it is not trivial to introduce state based supervisor as mentioned in the* ***Example 3.14****. Because, here for each local system, it has to maintain the global state estimator, so the supervisor is not just a function from* Q_i *(domain) to power set of controllable actions* Σ_{i_c} *for* $i \in \{1,2\}$. *In this simplest example if we find the supervisors based on the algorithm as mentioned in algorithm 3.2 and the local controller as mentioned in 3.18,* $\mathscr{S}_1(q_1,Q'_2) = \{b\}$ *for all* $Q'_2 \subseteq Q_2$ *and for all other* $q \in Q_1 \setminus \{q_1\}$ $\mathscr{S}_1(q,Q'_2) = \emptyset$. *Similarly, for*



(a) $\mathscr{A}_1$ (b) $\mathscr{A}_2$

FIGURE 3.21 : An imaginative distributed system as FSM's

FSM $\mathscr{A}_2$, $\mathscr{S}_2(Q'_1, q') = \emptyset$ *for all* $Q'_1 \subseteq Q_1$, *all* $q' \in Q_2$. *Further, the current global state has to be computed by the local state estimator* E_i *for* $i \in \{1,2\}$. ◇

As mentioned earlier, the problem and algorithm we presented in this section to generate synthesis may not be maximal permissive ; it can even be a deadlock controller. This problem arises due to the fact that each local state estimator computes the set of possible global current state by using Reach computation, and also the fact that our local controllers $\{\mathscr{S}_1, ...\mathscr{S}_n\}$ do not communicate with each other. In order to get a better controller in-terms of permissive nature, we can use piggyback technique as introduced in the paper [Kal+14]. The main idea of piggyback is whenever there is an exchange of messages between two subsystems, along with the actions, it appends its current state information. In this case, the actions set have to be modified slightly. It should be defined as $\Sigma_j = \Sigma_j^? \cup \Sigma_j^! \cup \Sigma_{\text{int},j}$ where $\Sigma_{\text{int},j}$ is purely an internal action set, $\Sigma_j^?$ is the set of message receiving actions, when such action is triggered, it receives a message from say $\mathscr{A}_i$, along with the message it also receives the current state of the system $\mathscr{A}_i$. The message sending actions set $\Sigma_j^!$, when it is triggered, it sends a message to say $\mathscr{A}_k$ it appends the current state with this message so that system $\mathscr{A}_k$ knows the current state of $\mathscr{A}_j$. By regularly sending the current state of each local automaton to one another and updating the state estimator for each E_j using this information can given the better (narrow) possible global states which may leads to better controller compared to the method we mentioned in our document.

3.4.4 Extending the Synthesis concept to the Infinite System

Finite state machine definition is good for the control synthesis of finite state system, in order to extend the control synthesis concept to infinite case, we have to describe the interested distributed system *DS* in terms of symbolic transition (we know why it is good idea to express infinite system in terms of symbolic transition system which was mentioned in section 3.2) with clear partition of the action set Σ into controllable and uncontrollable action set in the definition 3.8.

Once we partitioned the action set Σ into controllable action set Σ_c and uncontrollable action set Σ_{uc}, using this partition we can naturally partition the transition Δ into controlled transition (will be denoted by Δ_c) and uncontrolled transition (denoted by Δ_{uc}). Execution of controlled transition by the controllable action Σ_c and execution of uncontrolled transition by the uncontrollable action Σ_{uc}.

It is important to emphasize the fact given two symbolic transition system $\mathscr{T}_1, \mathscr{T}_2$ and their action set Σ_1, Σ_2 respectively, the intersections of this action set is empty $\Sigma_1 \cap \Sigma_2 = \emptyset$. In other word, given composition of symbolic transition system $\mathscr{T} = \|_{i\in\{1,..,n\}}\ \mathscr{T}_i$ for any two $i \neq j \in \{1,..,n\}\ \Sigma_i \cap \Sigma_j = \emptyset$. In addition to that all the reading action will be under uncontrollable action set in the sense that whenever there is a message in the incoming FIFO queues ($q \in Q$) reading the messages from that incoming queues should be uncontrollable. So the controllable actions are almost all the sending message actions via outgoing queues $q \in Q$ and all other actions excluding the sending, receiving and reading messages.

We have to describe here the bad states of the given *SCTS* $\mathscr{T}$. For a given *SCTS* $[\mathscr{T}] = \langle V, \Theta, M, Q, L, l_0, \Sigma(= \Sigma_c \cup \Sigma_{uc}), \Delta, U, \mathrm{AP}\rangle$, the bad states $\mathscr{B}$ is the subset of domain of the set $\mathscr{D}_V$ ($\mathscr{B} \subseteq \mathscr{D}_V$). The generated labelled transition system (semantic representation of the given $\mathscr{T}$) $[\mathscr{T}]$ for given *SCTS* $\mathscr{T}$ the bad states in terms of the system state represented as $X_b \subseteq L \bigtimes \mathscr{B} \bigtimes_{q\in Q} \{w_q \mid w_q \in M^*\}$. It is important to notice here that bad system state is based only on the system state variable values i.e $\mathscr{B} \subseteq \mathscr{D}_V$ and it does not depends on the system location L and the queue content w_q for any queue from queue set Q.

For a given set of symbolic transition system $\mathscr{T}_i = \langle V_i, \Theta_i, M_i, Q_i, L_i, l_{0,i}, U_1, \mathrm{AP}_1, \Sigma_i(= \Sigma_{i,c} \cup \Sigma_{i,uc}), \Delta_i\rangle$ where $i \in \{1,..,n\}$ and $\Sigma_{i,c} \cap \Sigma_{i,uc} = \emptyset$. The composition of the symbolic communicating transition system as mentioned in definition 3.7 is $\mathscr{T} = \|_{i\in\{1,..,n\}}\ \mathscr{T}_i$. The corresponding semantics of *SCTS* is $[\mathscr{T}] = \|_{i\in\{1,..,n\}}\ [\mathscr{T}_i]$. The set of bad states are $\mathscr{B}_i$ for each $\mathscr{T}_i$ and corresponding bad system states are $X_{b,i} = L_i \bigtimes \mathscr{B}_i \bigtimes_{q\in Q_i} \{w_q \mid w_q \in M^*\}$. For further details one can refer [Kal+12] and [Kal+14].

PROBLEM 3.2 Distributed State avoidance control problem for SCTS : *Given a composition of symbolic transition system $\mathscr{T} = ||_{i\in\{1,..,n\}} \mathscr{T}_i$, corresponding semantics of the system $[\mathscr{T}] = \|_{i\in\{1,..,n\}}$ $[\mathscr{T}_i]$ and the set of forbidden states (global system states) $\mathscr{B} = \cup_{i\in\{1,..,l\}} \prod_{j\in\{1,..,n\}} \mathscr{B}^i_j$ disjunction of global bad states where $\mathscr{B}^i_j \in X_j$ for all $j \in \{1,..,n\}$ and $i \in \{1,..,l\}$ for some $l \in \mathbb{N}$, the distributed state avoidance control problem consists in synthesizing a distributed supervisors $\mathscr{S} := \{\mathscr{S}_i \mid i \in \{1,..,n\}\}$ such that each controlled execution of the system $[\mathscr{T}_i]$ under the control of $\mathscr{S}_i$ avoids the global system bad states $\mathscr{B}$.* ◇

If the state space of the semantics of given composition of *SCTS* ($[\mathscr{T}]$) is finite, we can use the state estimator 3.17 and procedure 3.2 to find the local supervisor for each subsystem $\mathscr{T}_i$.

If the state space of the semantics is infinite, then to find the local supervisor for each symbolic communicating transition system $\mathscr{T}_i$, we need state estimator with little modification to the definition 3.17 and the state estimator procedure 3.2 with little modification, although I mentioned them as little modification in order to do this little modification we have to do over-approximation and abstraction on the state space.

Over-Approximation by Abstraction For the synthesis of composition of SCTS (i.e distributed system modelled in SCTS as $\mathscr{T} = \|_{i\in\{1,..,n\}}\ \mathscr{T}$ and respective semantic model $[\mathscr{T}] = \|_{i\in\{1,..,n\}}$

$[\mathscr{T}_i]$), the procedure mentioned in algorithm 3.2 a global state estimator for each local supervisor $\mathscr{S}_i$ won't terminate for infinite system (the state space of the semantics of given *SCTS* is infinite). This general problem is undecidable [Kal+14]. The main reason for the undecidability nature for this problem is because the procedure mentioned in algorithm 3.2 for the infinite state space X of given semantics $[\mathscr{T}]$ to compute Reach, CoReach operators will not terminate. The abstract interpretation-based techniques [CC77] allows us to compute infinite number of steps to compute the over-approximations of Reach, CoReach operators, and thus of the set $I(\text{Bad})$ and the global state estimator E_i for each local supervisor $\mathscr{S}_i$, $i \in \{1,...,n\}$.

For the *SCTS*, one such over-approximation techniques is the abstract interpretation of the incoming and outgoing queues content by a regular language, and each such regular language expression can be expressed as a finite automaton, which makes a way to approximate the state estimator procedure to compute the I_{bad} state, and gives a sound local supervisor $\mathscr{S}_i$ for each subsystem model $\mathscr{T}_i$.

3.5 Chapter Conclusion

In this chapter, we introduced and reviewed various model definitions to represent the distributed system *DS* in an abstract way. In the model checking process i.e verification, we need automaton model for the *DS*, but for the bigger system, it is tedious task to create such automaton model, so we stated the *symbolic transition system* (*SCTS*) definition to model *DS* in more compact way, from this *SCTS* we can generate automaton model by using existing software tools like *SPIN*, *NuSMV*, etc. In fact, in our verification process, we wrote *SCTS* model for our Nokia *SDN* architecture and protocol in *Promela* language and correspondingly generated the automaton model using the tool *SPIN* (more details given in chapter 5). Using this generated automaton model we (well actually the model checking tool) perform the verification, more details about the model checking *SDN* can be found in chapter 5. Although our mathematical model in this section can also handle infinite behavior, we are concerned only for the finite sized automaton in the experimental section presented in chapter 5.

In case of synthesis, the distributed system model *DS* is defined with controllable and uncontrollable action sets, the *FSM* modeling or *SCTS* modeling one can do the synthesis supervisor for the expressed *DS* by solely controlling the controllable action executions i.e disable some controllable actions to avoid reaching system bad states. In case the modeled *FSM* has an infinite number of states, we have to use over-approximation via abstraction. The experimental case study for the control synthesis of *SDN* application provided in chapter 6.

CHAPITRE 4

STATE SPACE REDUCTION TECHNIQUES

In this chapter, we state one of the biggest problems in model checking as well as the synthesis of discrete control systems which is *state space explosion* [Cla08], we will state the problem and some of the literature techniques to reduce the drastic state space explosion. More specifically the partial order reduction and assume guarantee techniques. We present our contribution in the assume guarantee techniques in the direction of compositional reasoning for distributed system verification and synthesis concepts.

4.1 State Space Explosion Problem in Formal Verification and Synthesis

Formal Schemes are costly in the sense that while verifying system correctness by verification process and while generating the supervisors using discrete control synthesis techniques, we have to explore all the possible states for the modelled system (either the automaton or the semantics of Symbolic Communicating Transition System for the given distributed system) and the trace through that each state is reached, a same state can be reached by multiple traces. For example, in a given system having two variables say x, y, at initial state both are assigned to zero and each step (or timestamp) it increment either x or y value by one, in order to reach a system state $x = 1, y = 1$. The system can reach this state by first increment x, second step to increment y, and vice versa. But these two ways of reaching the same state have to be explored by going along all possible traces. Generally, not just the enormous system states that creates the computational (in terms of memory and time consumption in these procedures) inefficiency for generating the control synthesis and performing the model checking process, it's also the possible number of traces i.e the sequence of steps it does internally to reach the given states of the system. Of course, depending on the properties and system model, we could avoid some of the traces to limit the brute force analysis in formal methods. In literature, this problem is referred to as '*state-space explosion*'.

In order to tackle such state-space explosion, model checking and discrete event system communities build various techniques such as partial order reduction, abstraction, Bisimulation, Compositional reasoning, modularity nature of properties and the system and so on. In our work, we applied compositional reasoning [Maj+21] to overcome such '*state-space explosion*' during

the formal verification *SDN* protocol on the device discovery using *MAC Learning* scheme. We also extended this compositional reasoning concept to synthesize the supervisor and applied it to the *SDN* platform to make sure it satisfies the given required properties.

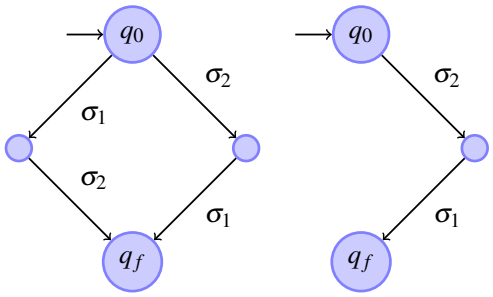
4.2 Partial Order Reduction for Model Checking Process

Partial Order Reduction To tackle enormous computation time and memory to automatically verify the proposed model of the given distributed system, partial order reduction uses the concept of commutative between concurrently executable transitions in the proposed distributed model. We will give brief notes on the partial order reduction for linear temporal logic specification. This concept can be used for *CTL* logic as well [Cla+18] (Chapter 6). Partial order reduction puts constraints on the order of transition events while verifying the model with respect to the specification.

The model checking the distributed system correctness w.r.t the set of requirements can easily become intractable because of huge State space explosion. Compare to the traditional mathematical way of proving the correctness of proposed algorithmic solution (like proving the correctness of Practical Byzantine Fault Tolerance using consensus and Proactive Recovery method [CL02], of course this method needs human ingenuity !, but we are here only care about how to prove the system correctness by automated way. But in this automated way i,e model checking procedure suffers from 'state space explosion problem'. In the model checking (i.e verification) process, the number states of the model grows exponentially (drastically) with the number of components (entities) in the modelled distributed system. The key concept in this partial order reduction is the kind of property we are expressing as a specification that does not depend on some transitions execution order (of course we have to be careful in choosing the list of transitions whose order of sequence do not matter for the given specification, in particular this partial order reduction used to reduce the state space while proving the *SDN* protocol correctness [MDW14a]).

Given automaton $\mathscr{A} = (Q, q_0, \Sigma, \Delta, \mathrm{AP}, L)$, an action $\sigma \in \Sigma$ is enabled from the state $q \in Q$ if there exists $q' \in Q$ and $\delta \in \Delta$ s.t $\delta = q \xrightarrow{\sigma} q'$ such set of actions we will denote here as enabled(q) (i.e set of actions that are enabled from q). Now we will define the *independence* relation set : $I \subseteq \Sigma \times \Sigma$. For a state $q \in Q$, $\delta_1 (= q \xrightarrow{\sigma_1} q_1), \delta_2 (= q \xrightarrow{\sigma_2} q_2) \in \Delta$ and $(\sigma_1, \sigma_2) \in I$ such that $\sigma_1, \sigma_2 \in$ enabled(q) with the following condition : $\sigma_2 \in$ enabled(q_1), $\sigma_1 \in$ enabled(q_2) i.e independent actions cannot disable each other and $q \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_3$ and $q \xrightarrow{\sigma_1} q_2 \xrightarrow{\sigma_1} q_3$ i.e executing two enabled independent actions in any order results the same global state. We will present this idea in the following figure 4.2 borrowed from the handbook of model checking [Cla+18].

For example, two local transitions in different automata of a given composition of automata that do not use any global variables and doing two different updates in the systems are independent of each other. In an asynchronous distributed system, sending and receiving by two



(a) Full state space

(b) Reduced state space using commutative and Independent conditions

FIGURE 4.1 : Concept of Partial Order Reduction

different components of the given distributed system are independent.

We will denote a transition $\delta = q \xrightarrow{\sigma} q'$ is invisible if $L(q) = L(q')$. In the presence of an invisible transition in the run we can define shutter equivalent as follows, for given two runs $\text{run}_1 \neq \text{run}_2 \in \text{Run}(\mathscr{A})$ such that $\text{trace}(run_1) = \text{trace}(run_2)$ then these two runs are shutter equivalent. For example given run such that $\text{trace}(\text{run}) = (a)(a)(a \wedge b)$ will be shutter equivalent to the $\text{run}' \in \text{Run}(\mathscr{A})$ whose $\text{trace}(\text{run}') = (a)(a \wedge b)$. The reason is two runs which are stuttering equivalent sequences can not be distinguished by the *LTL* specification [PW97]. We now give the basic partial order reduction technique for *LTL* specification borrowed from the handbook of model checking [Cla+18] for more detailed and sophisticated methods. Please refer to the same book or even google search will give you dozens of papers on this technique.

Reduction for LTL In order to do the partial order reduction technique, we have to define the *ample* set for each $q \in Q$, denoted as $\text{ample}(s) \subseteq \Delta$ this set is calculated on the fly while doing partial order reduction, please refer the book [Cla+18] for more details about this ample set. The basic partial order reduction algorithms are usually described as a variant of classic depth first search *DFS*. The procedure *hash* is a standard hashing of its parameter in a hash table to keep track of the set of states already visited in the model checking procedure. One can check whether that value was hashed i.e was visited already. The simple *DFS* algorithm for model checking process procedure (yes model checking is an algorithm to do certain things) is depicted in fig. 4.2.

The partial order reduction have to rely on calculating subset of enabled transitions i.e $\text{ample}(q) \subseteq \text{enabled}(q)$ for each state $q \in Q$ of the automaton model. The revised model checking algorithm fig. 4.2 with partial order technique is shown in fig. 4.3. Model checking by exploring the $\text{ample}(q)$ rather than $\text{enabled}(q)$ for the state $q \in Q$ reduces the state space by partial order reduction concept, condition to evaluate the ample set and the proof of this concept can be found in the model checking handbook [Cla+18].

proc DFS(q) ;
    local variable q' ;
    hash(q) ;
    for each $\delta \in$ enabled(q) do
        Let q' be such that $q \xrightarrow{\sigma} q' = \delta$
        if $\neg$hashed(q') then DFS(q') ;
end DFS(q) ;

FIGURE 4.2 : DFS to explore the state space of the automaton model

proc DFS(q) ;
    local variable q' ;
    hash(q) ;
    Calculate ample(q)
    for each $\delta \in$ ample(q) do
        Let q' be such that $q \xrightarrow{\sigma} q' = \delta$
        if $\neg$hashed(q') then DFS(q') ;
end DFS(q) ;

FIGURE 4.3 : DFS to explore the reduced state space using partial order reduction technique

4.3 Avoiding State Space Explosion Problem by Compositional Reasoning

Before presenting the compositional reasoning technique, we will give short notes on *Rely-Guarantee technique* introduced by Stark W.Eugene [Sta85] and *Interface theory* written by E.M Clarke [CLM89].

A Proof Technique for Rely-Guarantee Properties [Sta85]. This methodology allows to reduce the huge state space arising in the verification process when several components are used to model the system. It is a state based model and each processor contains a set of variables and the actions are the assignment on the variables and the communication is based on the shared variable assignment made between the processors. The idea is to encode the protocol invariant as a *LTL* formula and encode the fairness assumption in the *LTL* formula if needed and then check the global specification. Further, the technique called as *Rely/Guarantee style method* of proving the *global liveness* (informally, eventually some good things happens globally, like producing the useful result to the user) property by checking the *local liveness* property and the logical approach of how to *infer the global liveness property from the local liveness property*.

Formally, assume given a finite number of processors $P_1, P_2, \ldots$ that form a synchronous and non-blocking processors $P = \|_{0<i<\inf} P_i$. The goal is to check the global liveness property $P \vDash R \rightarrow G$ call R, G as rely/guarantee specification. Asking the question by proving $P \vDash R_i \rightarrow G_i$

for all the processors does it imply $P \vDash R \rightarrow G$?. Here R_i, G_i is local rely/guarantee specification for the processor P_i. The answer is yes by checking additional sets of constraints. Here we refer to a processor model for the system which has a set of variables, possible assignments to these variables, and a set of actions triggered based on the guard of the variable values.

DEFINITION 4.1 *[Sta85] For a given set* $I = \{1,2,...,n\}$ *, set of finite number of processors* $\{P_i | 0 < i < \inf\}$ *and the program* $P = \|_{0<i<\inf} P_i$ *and specifications R, G,* $\{R_i, G_i | i \in I\}$*. Let define a* cut set *for the program* P *:* $\{RG_{i,j} | i, j \in I \cup \{ext\}\}$ *a set of specifications if*

$$P \vDash R \rightarrow \bigwedge_{j \in I} RG_{ext,j} \tag{4.1}$$

$$P \vDash \bigwedge_{i \in I} RG_{i,ext} \rightarrow G \tag{4.2}$$

$$P \vDash \bigwedge_{i \in I \cup \{ext\}} RG_{i,j} \rightarrow R_j, \ \forall j \in I \tag{4.3}$$

and

$$P \vDash G_i \rightarrow \bigwedge_{j \in I \cup \{ext\}} RG_{i,j}, \ \forall i \in I \tag{4.4}$$

◇

In the above definition $i, j \in I$, specification $\mathrm{RG}_{i,j}$ should be thought of what processor P_i guarantees to the processor P_j or what processor P_j relies on the processor P_i. The specification $\mathrm{RG}_{\mathrm{ext},j}$ express what environment guarantees to the processor P_j and $\mathrm{RG}_{i,\mathrm{ext}}$ express what processor P_i guarantees to the environment.

DEFINITION 4.2 *[Sta85] For the given finite set I from definition 4.1, define a* cycle *as* $\{(i_1,i_2),(i_2,i_3),...,(i_m-1,i_m)\}$ *for some* $1 \leq m \leq n$ *where all* $i_l \in I, l \in [1,..,m]$ *such that* $i_m = i_1$.
We say that the cut set collection from definition 4.1 is acyclic *if :*

$$P \vDash \bigvee_{k \in [1,..,m-1]} RG_{i_k,i_{k+1}}$$

for all cycles of I. ◇

Importantly the cut set should be hand made by the programmer who wants to do compositional model check on the program P to verify the required property. In the above definition the

paper [Sta85] introduced for the reason that at-least in the given cycle of indices, any one of the local rely/guarantee is true means, by the cyclic nature of inference rule one by one rest of the rely/guarantee will become true and keep repeating the specification.

THEOREM 4.1 *[Sta85]* Rely/Guarantee Proof Rule- *Suppose P is a program, I is a finite index set, and the collection* $RG = \{RG_{i,j} \mid i, j \in I \cup \{ext\}\}$ *is an* acyclic *cut set for program P and specifications* R, G, $\{R_i, G_i \mid i \in I\}$. *Then to prove the statement* $P \vDash R \rightarrow G$, *it suffices to show* $P \vDash R_i \rightarrow G_i$, *for all* $i \in I$. ◇

Interface Theory Here, we will give a short presentation of interface theory used by [CLM89] to derive the compositional reasoning for *Computation Tree logic* (*CTL*). Let P be a set of finite state processes, and assume that we know what it means for two processes P_1 and P_2 to be equivalent ($P_1 \equiv P_2$ means have same behavior w.r.t the action set i.e $\mathscr{L}(P_1) = \mathscr{L}(P_2)$). Each process will have associated with it a certain set of atomic propositions that are used in distinguishing states and transitions. Σ_P will denote the set associated with the process P. The set of propositions associated with the parallel composition of two process will be the union of the sets associated with the individual processes : $\Sigma = \Sigma_1 \cup \Sigma_2, P \downarrow \Sigma_1$ will be the restriction of P to Σ_1. This process is obtained by hiding all of the symbols in Σ_P that are not in Σ_1.

The interface rule deals with the parallel composition of two processes P_1 and P_2. Let A_1, A_2 be the interface processes after hiding the transition words $\Sigma_2 \setminus \Sigma_1, \Sigma_1 \setminus \Sigma_2$ for P_1 and P_2 respectively. Let ϕ, ψ are a formula formed over the transition alphabets of the processor P in a CTL logic ($\mathscr{L}$).

Interface rule from the paper[CLM89] is as follows,

$$
\begin{aligned}
&P_1 \downarrow \Sigma_1 \cap \Sigma_2 \equiv A_1 \wedge \\
&\phi \in \mathscr{L}(\Sigma_2) \wedge \\
&A_1 \parallel P_2 \vDash \phi \implies \\
&P_1 \parallel P_2 \vDash \phi
\end{aligned}
$$

Similarly,

$$
\begin{aligned}
&P_2 \downarrow \Sigma_1 \cap \Sigma_2 \equiv A_2 \wedge \\
&\psi \in \mathscr{L}(\Sigma_1) \wedge \\
&P_1 \parallel A_2 \vDash \psi \implies \\
&P_1 \parallel P_2 \vDash \psi
\end{aligned}
$$

where by $P_1 \downarrow \Sigma_1 \cap \Sigma_2 \equiv A_1$, it refers to $P_{\Sigma_1 \cap \Sigma_2} \mathscr{L}(P_1)$ and $A_1 || P_2 \vDash \phi$ refers to $\mathscr{L}(A_1 || P_2) \models \phi$ (i.e all the sequences from this language set satisfy the specification ϕ).

The soundness of the above interface rule can be derived from the following properties as mentioned in the paper [CLM89] :

- Suppose $\Sigma_1 = \Sigma_2$, then $P_1 \equiv P_2$ implies $\forall \phi \in \mathscr{L}(\Sigma_1)[P_1 \vDash \phi \leftrightarrow P_2 \vDash \phi]$.

That is when the actions set between the processors P_1, P_2 are same (Σ) and there language behaviour is also same then for any specification formed over the action set Σ such that all the sequences from the language of P_1 satisfy this specification then all the sequence from the language of P_2 do satisfy this specification and converse also true.

- If $P_1 \equiv P_2$ and Q is another process, then $P_1 \parallel Q \equiv P_2 \parallel Q$.

  That is to say that when the behaviour of P_1, P_2 are same in terms of language, then composing the processor Q with either P_1 or P_2 gives the same language behaviour i.e $\mathscr{L}(P_1||Q) = \mathscr{L}(P_2||Q)$.

- $(P_1 \parallel P_2) \downarrow \Sigma_1 \equiv P_1 \parallel (P_2 \downarrow \Sigma_1)$ and similarly
  $(P_1 \parallel P_2) \downarrow \Sigma_2 \equiv (P_1 \downarrow \Sigma_2) \parallel P_2$.

  That is to say that for any other processor Q such that $\mathscr{L}(Q) = P_{\Sigma_1}(\mathscr{L}(P_2))$ so that $P_{\Sigma_1}(\mathscr{L}(P_1||P_2)) = \mathscr{L}(P_1||Q)$.

- If $\phi \in \mathscr{L}(\Sigma^{'})$ and $\Sigma^{'} \subseteq \Sigma$, then $P \vDash \phi$ iff $P \downarrow \Sigma^{'} \vDash \phi$.

  That is to say that for the any specification formed over the action set Σ' then $\mathscr{L}(P) \models \phi$ if and only if $P_{\Sigma'}(\mathscr{L}(P)) \models \phi$.

The above four properties (as mentioned in the paper) are used to derive the inference rule in the paper [CLM89]. A similar technique can be used to show the soundness of simple rules like : the above 'interface rule' can be derived to infer the rule $P_1 \parallel P_2 \vDash \phi \wedge \psi$ as well.

The above two results i.e Rely-guarantee and Interface rule inspired us to frame the following compositional rule for our purposes.

4.3.1 Introduction to Compositional Reasoning

Compositional reasoning appears naturally while designing a distributed system, since the designer often designs specifications for each component of the system on whose global requirements have to be verified. Compositional reasoning allows us to verify properties of each sub-systems separately, and combine these rules to infer properties for the global system.

Given an automaton $\mathscr{A}$ and *LTL* safety formulas ϕ, ψ, let us denote by $\langle\phi\rangle\mathscr{A}\langle\psi\rangle$ a triple such that ϕ represents the *assumption* that can be made on the environment of $\mathscr{A}$, while ψ represents the *guarantee* that $\mathscr{A}$ provides under the assumption that environment satisfies ϕ.
In a nutshell and without giving any conditions on the various components and specifications, what we want to prove is the following :

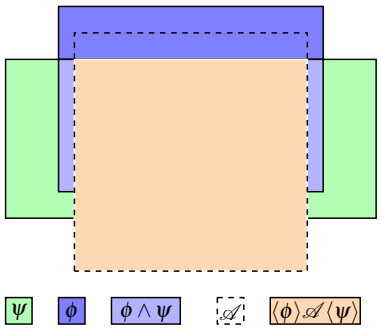


FIGURE 4.4 : $\langle\phi_1\rangle\mathscr{A}\langle\phi_2\rangle$ illustration

$$\frac{\langle\phi_{as}\rangle\mathscr{A}_1\langle\phi_I\rangle \qquad \langle\phi_I\rangle\mathscr{A}_2\langle\phi_{guar}\rangle}{\langle\phi_{as}\rangle\mathscr{A}_1\|\mathscr{A}_2\langle\phi_{guar}\rangle} \tag{4.5}$$

(4.5) means that, if under the assumption that $\mathscr{A}_1 \models \phi_{as}$, $\mathscr{A}_1 \models \phi_I$ and similarly for $\mathscr{A}_2$ w.r.t. ϕ_I, where ϕ_I is an intermediate specification and ϕ_{guar}, then it entails that, if, under the assumption that the composed system $\mathscr{A}_1\|\mathscr{A}_2 \models \phi_{as}$, then this system verifies ϕ_{guar}.

In the sequel, we shall consider this framework from different points of view

- Model-checking of *DS* system using
  - language-based compositional reasoning techniques
  - *LTL* compositional reasoning techniques

  in the next two subsections.
- Synthesis of supervisors ensuring
  - Language-based specifications
  - *LTL* specifications

  in the section 3.4 using compositional reasoning techniques

4.3.2 Compositional Reasoning from a language-based point view

In this section, we will see the behaviour of composition of automata $\mathscr{A}_1\|\mathscr{A}_2$ in-terms of languages and how its coordinates with respect to the common actions set (i.e $\Sigma_1 \cap \Sigma_2$). In the spirit of rely/guarantee and inference rule, we shall try to locally prove some global properties by proving local properties on each component.

LEMMA 4.1 *For $\Sigma' \subseteq \Sigma$ and a prefix-closed language $\mathscr{L} \subseteq (\Sigma')^*$, then $\forall s \in \Sigma^*$, if $P_{\Sigma'}(s) \in \mathscr{L}$, then we have $s \in P_{\Sigma'}^{-\Sigma}(\mathscr{L}) \subseteq \Sigma^*$* ◇

LEMMA 4.2 (Language Projection) *Consider two automata $\mathscr{A}_1$, $\mathscr{A}_2$ and $\mathscr{A} = \mathscr{A}_1 \parallel \mathscr{A}_2$ (definition 3.3). Then, $\forall i \in \{1,2\}$, $P_{\Sigma_i}(\mathscr{L}(\mathscr{A})) \subseteq \mathscr{L}(\mathscr{A}_i)$.*

Proof : According to definition 3.3, $\mathscr{L}(\mathscr{A}) = P_{\Sigma_1}^{-1}(\mathscr{L}(\mathscr{A}_1)) \cap P_{\Sigma_2}^{-1}(\mathscr{L}(\mathscr{A}_2))$ which can be rephrased as $\mathscr{L}(\mathscr{A}_1 \| \mathscr{A}_2) = \{s \in \Sigma^* \mid P_{\Sigma_1}(s) \in \mathscr{L}(\mathscr{A}_1) \wedge P_{\Sigma_2}(s) \in \mathscr{L}(\mathscr{A}_2)\}$. This entails that, given $s \in \mathscr{L}(\mathscr{A})$, $P_{\Sigma_i}(s) \in \mathscr{L}(\mathscr{A}_i)$, $i = 1,2$. Hence the result ◇

LEMMA 4.3 *Given $w \in (\Sigma_1 \cup \Sigma_2)^*$, and $\mathscr{L} \subseteq \Sigma_1^*$. If $w \in P_{\Sigma_1}^{-\Sigma_2}(\mathscr{L})$ then $P_{\Sigma_1}(w) \in \mathscr{L}$.* ◇

THEOREM 4.2 *Given composition of automata $\mathscr{A} = \mathscr{A}_1 \parallel \mathscr{A}_2$, where $\mathscr{L}(A_i) \subseteq \Sigma_i^*$ and specifications language $\mathscr{L}_{\text{as}} \subseteq \Sigma_1^*$, $\mathscr{L}_I \subseteq (\Sigma_1 \cap \Sigma_2)^*$ and $\mathscr{L}_{\text{guar}} \subseteq \Sigma_2^*$ with respect the* LTL *specifications Φ_{as}, Φ_I and Φ_{guar} respectively, then the following modus ponen (or inference rule) will hold*

$$\frac{(1)\ P_{\Sigma_1\cap\Sigma_2}(\mathscr{L}(\mathscr{A}_1) \cap \mathscr{L}_{\text{as}}) \subseteq \mathscr{L}_I \qquad (2)\ P_{\Sigma_1\cap\Sigma_2}^{-\Sigma_2}(\mathscr{L}_I) \cap \mathscr{L}(\mathscr{A}_2) \subseteq \mathscr{L}_{\text{guar}}}{P_{\Sigma_1}^{-\Sigma_2}(\mathscr{L}_{\text{as}}) \cap \mathscr{L}(\mathscr{A}_1 \| \mathscr{A}_2) \subseteq P_{\Sigma_2}^{-\Sigma_1}(\mathscr{L}_{\text{guar}})} \tag{4.6}$$

◇

Equation 4.6 rephrases the equation 4.5 in a language based perspective.

Before seeing the argument for the above Theorem, let us explain the meaning of equation 4.6

- (1) means that whenever a sequence in $\mathscr{L}(\mathscr{A}_1)$ is within the assumption specification language $\mathscr{L}_{\text{as}}$, then this sequence projected on $\Sigma_1 \cap \Sigma_2$ also belongs to the intermediate specification language $\mathscr{L}_I$.
- (2) means that whenever a sequence in $\mathscr{L}(\mathscr{A}_2)$ is within the inverse projection of the intermediate specification $\mathscr{L}_I$ wr.t. Σ_2 then it also belongs to the guarantee specification language $\mathscr{L}_{\text{guar}}$.

Let us now prove Theorem 4.2 :

Proof : Lets $s \in P_{\Sigma_1}^{-\Sigma_2}(\mathscr{L}_{\text{as}}) \cap \mathscr{L}(\mathscr{A}_1 \| \mathscr{A}_2)$, then by lemma 4.2, we have $P_{\Sigma_i}(s) \in \mathscr{L}(\mathscr{A}_i)$ for $i \in \{1,2\}$. As $s \in P_{\Sigma_1}^{-\Sigma_2}(\mathscr{L}_{\text{as}}) \cap \mathscr{L}(\mathscr{A}_1 \| \mathscr{A}_2)$ that means we have $s \in P_{\Sigma_1}^{-\Sigma_2}(\mathscr{L}_{\text{as}})$. If $s \in P_{\Sigma_1}^{-\Sigma_2}(\mathscr{L}_{\text{as}})$ by lemma 4.3 we have $P_{\Sigma_1}(s) \in \mathscr{L}_{\text{as}}$. As $P_{\Sigma_1}(s) \in \mathscr{L}_{\text{as}}$ and $P_{\Sigma_1}(s) \in \mathscr{L}(\mathscr{A}_1)$ then by (1), $P_{\Sigma_1\cap\Sigma_2}(P_{\Sigma_1}(s)) = P_{\Sigma_1\cap\Sigma_2}(s) \in \mathscr{L}_I$. Furthermore, let us call $s' = P_{\Sigma_2}(s)$. As $P_{\Sigma_1\cap\Sigma_2}(s) = P_{\Sigma_1\cap\Sigma_2}(s')$, $P_{\Sigma_1\cap\Sigma_2}(s') \in \mathscr{L}_I$. If $P_{\Sigma_1\cap\Sigma_2}(s') \in \mathscr{L}_I$, then we have $s' \in P_{\Sigma_1\cap\Sigma_2}^{-\Sigma_2}(\mathscr{L}_I)$ by lemma 4.1. Hence $P_{\Sigma_2}(s) \in P_{\Sigma_1\cap\Sigma_2}^{-\Sigma_2}(\mathscr{L}_I)$. Since we already established that $P_{\Sigma_2}(s) \in \mathscr{L}(\mathscr{A}_2)$ by using second premise (2), we can state that $P_{\Sigma_2}(s) \in \mathscr{L}_{\text{guar}}$. Finally, as $P_{\Sigma_2}(s) \in \mathscr{L}_{\text{guar}}$ then it is trivial to notice $P_{\Sigma_2}^{-\Sigma_1}(P_{\Sigma_2}(s)) \subseteq P_{\Sigma_2}^{-\Sigma_1}(\mathscr{L}_{\text{guar}})$. Finally as $s \in P_{\Sigma_2}^{-\Sigma_1}(P_{\Sigma_2}(s))$, we can conclude that $s \in P_{\Sigma_2}^{-\Sigma_1}(\mathscr{L}_{\text{guar}})$. Hence the Theorem. ◇

4.3.3 Compositional Reasoning for *LTL* Specification

We will now focus on compositional reasoning of *LTL* specifications, but first we need to introduce some new concepts. In the rest of the chapter it is always assumed that for all the actions $\sigma \in \Sigma$, we consider an atomic proposition $\mathsf{Sync}(\sigma) \in \mathrm{AP}$. In the given automaton $\mathscr{A}$, whenever there is a transition $q \xrightarrow{\sigma} q'$ then we have $\mathsf{Sync}(\sigma) \in L(q')$.

ASSUMPTION 4.1 *When we consider the composition of two automata $\mathscr{A}_1$, $\mathscr{A}_2$, we always assume that $\{\mathsf{Sync}(\sigma) \mid \sigma \in \Sigma_1 \cap \Sigma_2\} = AP_1 \cap AP_2 = A$, and $|L_1(q_{i,1}) \cap A| \leq 1$, $|L_2(q_{j,2}) \cap A| \leq 1$ for all states $q_{i,2} \in Q_1$, $q_{j,2} \in Q_2$.*

REMARK 4.1 *Such an assumption is not new. For example, paper [BCC98] mentions this restriction while using compositional reasoning to reduce the state-space complexity for asynchronous communicating system where the actions triggered in synchronized and specifications are expressed in* CTL *logic. That is to say in the given composition automata $\mathscr{A}_1 \parallel \mathscr{A}_2$, $\mathscr{A}_i$ for $i \in \{1,2\}$, if $(q \xrightarrow{\sigma_1} q') \in \Delta_i$ and $(q'' \xrightarrow{\sigma_2} q') \in \Delta_i$ either $\sigma_1 = \sigma_2$ or for the sake of modeling we introduce the new state q_2 to the automaton state Q_i such that $q \xrightarrow{\sigma_1} q' \in \Delta_i$ and removing the transition $q'' \xrightarrow{\sigma_2} q'$ and then adding the transitions $q'' \xrightarrow{\sigma_2} q_2, q_2 \xrightarrow{\varepsilon} q' \in \Delta_i$ so that we have at most one $\mathsf{Sync}(\sigma)$ belongs to any labelling of automaton state (i.e $|L_i(q_i) \cap A| \leq 1$ for all $q_i \in Q_i$).* ◇

OBSERVATION 4.1 *In the composition of automata $\mathscr{A}_1 \parallel \mathscr{A}_2$, $\mathrm{run} \in \mathrm{Run}(\mathscr{A}_1 \parallel \mathscr{A}_2)$, $\mathrm{run} = (q_{1,0}, q_{2,0}) \xrightarrow{\sigma_1} (q_{1,1}, q_{2,1}) \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_n} (q_{1,n}, q_{2,n})$ with the assumption 4.1, it is always true that $L_1(q_{1,j}) \cap A = L_2(q_{2,j}) \cap A$ for all $j \in \{1, \ldots, n\}$.* ◇

In the question of $\mathscr{A}_1 \parallel \mathscr{A}_2 \vDash \phi_{\text{as}} \Rightarrow \phi_{\text{guar}}$ where ϕ_{as} is the assumption specification to the automaton $\mathscr{A}_1$, and ϕ_{guar} is the property expected (expected specification) from the automaton $\mathscr{A}_2$.

In the automaton $\mathscr{A}_1$ working under the assumption ϕ_{as} such that it guarantee the language specification ϕ_I, under such guarantee, when such automaton is in the composition with an automaton $\mathscr{A}_2$, and this automaton under ϕ_I, we have to check that the automaton $\mathscr{A}_2$ satisfy the specification $\phi_I \Rightarrow \phi_{\text{guar}}$ i.e $\mathscr{A}_2 \vDash \phi_I \Rightarrow \phi_{\text{guar}}$ in order to guarantee that $\mathscr{A}_1 \parallel \mathscr{A}_2 \vDash \phi_{\text{as}} \Rightarrow \phi_{\text{guar}}$. So that we can guaranty equation 4.5.

Before that we will define some notations to build the compositional reasoning in assume guarantee way for state based specification. Before stating the compositional reasoning we slowly grasp the meaning and behaviour of the automaton and its trace behaviour for the specification formed over the actions (Σ) and atomic predicates (AP) set.

DEFINITION 4.3 *For given Atomic predicates sets AP, AP' and in the given sequence $w \in (2^{AP})^*$, $w \downarrow AP'$ is the restriction of sequence w to AP'. I.e for the given trace $w = AP_1.AP_2 \ldots AP_n$ restriction with respect to the atomic predicates set AP' as $w \downarrow AP' = (AP_1 \cap AP').(AP_2 \cap AP') \ldots (AP_n \cap AP')$.* ◇

LEMMA 4.4 (Stutter-Invariant for the Atomic Predicate Specification) *Consider atomic predicates set AP and a* LTL$\setminus X$ *formula* ϕ *formed over atomic predicates set* $AP_\phi \subseteq AP$*. For all traces* $w \in (2^{AP})^*$*,* $w \models \phi$ *if, and only if* $w \downarrow AP_\phi \models \phi$*.* $\diamond$

Proof This is immediate from the semantics of *LTL* since atomic propositions in $\mathrm{AP} \setminus \mathrm{AP}_\phi$ do not influence the satisfaction. $\diamond$.

LEMMA 4.5 (Run Projection :) *Consider any two given automata* $\mathscr{A}_1$*,* $\mathscr{A}_2$ *and* $\mathscr{A} = \mathscr{A}_1 \| \mathscr{A}_2$ *as defined in definition 3.3. Then we have for all* $i \in \{1,2\}$*,* $\mathrm{Run}(\mathscr{A}) \downarrow \mathscr{A}_i \subseteq \mathrm{Run}(\mathscr{A}_i)$*.* $\diamond$

Proof : Let's take a run from $\mathrm{Run}(\mathscr{A}_1 \| \mathscr{A}_2)$, $\mathrm{run} = (q_{0,1}, q_{0,2}) \xrightarrow{\sigma_1} .. \xrightarrow{\sigma_n} (q_{n,1}, q_{n,2})$, take the projection of this run to the automaton $\mathscr{A}_1$ run $\downarrow \mathscr{A}_1$ as by the definition 3.4, then this is trivially true that run $\downarrow \mathscr{A}_1 \in \mathrm{Run}(\mathscr{A}_1)$, so this is similarly true for the automaton $\mathscr{A}_2$ hence the lemma. $\diamond$

COROLLARY 4.1 *From the definition of the synchronized product between the automaton* $\mathscr{A}$ *and model of any* LTL *specification* ϕ *as by the definition 3.10, the* $\mathrm{Run}(\mathscr{A}) = \mathrm{Run}(\mathscr{A} \otimes \mathscr{M}_\phi) \downarrow \mathscr{A}$*.* $\diamond$

The notation $\mathrm{Run}(\mathscr{A} \otimes \mathscr{M}_\phi) \downarrow \mathscr{A}$ simply refers to discard the specification *NFA* model states (because transition actions are the super-set of the atomic predicates AP of automaton $\mathscr{A}$ and the $\mathscr{A} \otimes \mathscr{M}_\phi$ does not block any of the transition in $\mathscr{A}$).

LEMMA 4.6 (Stutter-Invariant for Language Specification) *Consider an automaton* $\mathscr{A} = (Q, q_0, \Sigma, \Delta, L, AP)$ *and a* LTL$\setminus X$ *(i.e* LTL *specification without next state operator) specification formed over the action set* $\Sigma' \subseteq \Sigma$*. Then, for all sequence* $\sigma \in \Sigma^*$*, if* $\sigma \downarrow \Sigma' \vDash \phi$ *(here the specification* ϕ *classify the language a subset of* $(\Sigma')^*$*), we have* $\sigma \vDash \phi$ *(here the specification* ϕ *classify the language a subset of* $(\Sigma)^*$*).* $\diamond$

Proof : Since the specification ϕ is without next state operator, the semantics of *LTL* without next state operator applies to the Stutter-Invariant [PW97] for the language specification. $\diamond$

THEOREM 4.3 (Compositional Reasoning Theorem for State Based Specification) *Given composition of automata* $\mathscr{A}_1 \parallel \mathscr{A}_2$*, with global specification* $\phi_{\mathrm{as}} \Rightarrow \phi_{\mathrm{guar}}$ *where the safety critical specifications* LTL$\setminus X$ *(i.e* LTL *specification without next state operator)* $\phi_{\mathrm{as}}, \phi_{\mathrm{guar}}$ *formed over the atomic predicate set* AP_1, AP_2 *respectively. If there exists a* LTL$\setminus X$ *language specification* ϕ_I *formed over the atomic predicates set* $\{\mathsf{Sync}(\sigma) \mid \sigma \in \Sigma_1 \cap \Sigma_2\} (= AP_1 \cap AP_2)$ *then we have the following inference rule holds.*

$$\frac{\langle \phi_{\mathrm{as}} \rangle \mathscr{A}_1 \langle \phi_I \rangle \qquad \langle \phi_I \rangle \mathscr{A}_2 \langle \phi_{\mathrm{guar}} \rangle}{\langle \phi_{\mathrm{as}} \rangle \mathscr{A}_1 \| \mathscr{A}_2 \langle \phi_{\mathrm{guar}} \rangle}$$

The meaning of above equation is

$$\frac{\text{Run}_{\mathscr{B}^{\phi_{\text{as}}\Rightarrow\phi_I}}(\mathscr{A}_1 \otimes \mathscr{M}_{\phi_{\text{as}}\Rightarrow\phi_I}) = \emptyset \qquad \text{Run}_{\mathscr{B}^{\phi_I\Rightarrow\phi_{\text{guar}}}}(\mathscr{A}_2 \otimes \mathscr{M}_{\phi_I\Rightarrow\phi_{\text{guar}}}) = \emptyset}{\text{Run}_{\mathscr{B}^{\phi_{\text{as}}\Rightarrow\phi_{\text{guar}}}}((\mathscr{A}_1 \parallel \mathscr{A}_2) \otimes \mathscr{M}_{\phi_{\text{as}}\Rightarrow\phi_{\text{guar}}}) = \emptyset} \tag{4.7}$$

◇

Proof : Let's take run from $\text{Run}((\mathscr{A}_1 \| \mathscr{A}_2) \otimes \mathscr{M}_{\phi_{\text{as}}\Rightarrow\phi_{\text{guar}}})$ and say that this run is in $\text{Run}_{\mathscr{B}^{\phi_{\text{as}}\Rightarrow\phi_{\text{guar}}}}((\mathscr{A}_1 \| \mathscr{A}_2) \otimes \mathscr{M}_{\phi_1\Rightarrow\phi_{\text{guar}}})$, then by the corollary 4.1, this run excluding the model states ($S_{\phi_{\text{as}}\Rightarrow\phi_{\text{guar}}}$) will be in $\text{Run}(\mathscr{A}_1 \| \mathscr{A}_2)$.

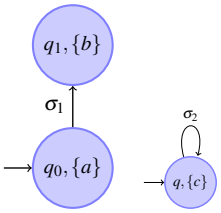
By lemma 4.5, the projected run $\text{run} \downarrow \mathscr{A}_1 \in \text{Run}(\mathscr{A}_1), \text{run} \downarrow \mathscr{A}_2 \in \text{Run}(\mathscr{A}_2)$, if the run $\text{run} \downarrow \mathscr{A}_1$ does not satisfy ϕ_{as} then this run trivially satisfy $\phi_{\text{as}} \Rightarrow \phi_{\text{guar}}$, so assume that $\text{trace}(\text{run} \downarrow \mathscr{A}_1) \vDash \phi_{\text{as}}$.

By the premise $\text{Run}_{\mathscr{B}^{\phi_{\text{as}}\Rightarrow\phi_I}}(\mathscr{A}_1 \otimes \mathscr{M}_{\phi_{\text{as}}\Rightarrow\phi_I}) = \emptyset$, $\text{trace}(\text{run} \downarrow \mathscr{A}_1) \vDash \phi_{\text{as}} \Rightarrow \phi_I$, hence $\text{trace}(\text{run} \downarrow \mathscr{A}_1) \vDash \phi_I$. By the observation 4.1, $\text{trace}(\text{run} \downarrow \mathscr{A}_2) \vDash \phi_I$ because the specification ϕ_I formed over the atomic predicates set $\{\text{Sync}(\sigma) \mid \sigma \in \Sigma_1 \cap \Sigma_2\}$.

By the premise $\text{Run}_{\mathscr{B}^{\phi_I\Rightarrow\phi_{\text{guar}}}}(\mathscr{A}_2 \otimes \mathscr{M}_{\phi_I\Rightarrow\phi_{\text{guar}}}) = \emptyset$, projected run to the automaton $\mathscr{A}_2$ satisfy the specification ϕ_{guar} ($\text{trace}(\text{run} \downarrow \mathscr{A}_2) \vDash \phi_I \Rightarrow \phi_{\text{guar}}$) hence the theorem. ◇

REMARK 4.2 *In the theorem 4.3,*

1 : Next state operator is removed for a simple reason, let's take an simple example. In the



(a) $\mathscr{A}_1$ (b) $\mathscr{A}_2$

FIGURE 4.5 : example

automaton, the specification $\phi := a \rightarrow X\{b\}$ *is hold (i.e automaton* $\text{trace}(\text{Run}(\mathscr{A}_1)) \vDash \phi$*) but in the composition of automaton* $\mathscr{A}_1 \parallel \mathscr{A}_2$ *the specification not holds* $\text{trace}(\text{Run}(\mathscr{A}_1 \parallel \mathscr{A}_2) \downarrow \Sigma_1) \nvDash \phi$*. On the other hand* $\psi := a \rightarrow \Diamond\{b\}$ *holds for both (*$\text{trace}(\text{Run}(\mathscr{A}_1)) \vDash \psi$ *and* $\text{trace}(\text{Run}(\mathscr{A}_1 \parallel \mathscr{A}_2) \downarrow \Sigma_1) \vDash \psi$*) with fairness condition i.e both automaton can able to trigger their internal actions from the current state (if exists) with out struck in that position for ever. Because of our chosen formalism to verify the trace of the composition of automata, the* next *state operator X is not behaving well as opposed to the* eventual *operator* $\Diamond$*. This is the reason why we remove the* next state *operator in the last theorem.*

2 : In order to use the compositional reasoning technique we also have to keep in mind the following statement.

Remember that two automata in the composition product, they communicate (share or trigger) each other to assign a value to the variable by doing the common actions, not based on the common atomic predicates. It is completely valid to have a state $q = (q_1, q_2) \in Q$ *s.t* $L_1(q_1) = \neg p$ *and* $L_2(q_2) = p$*. We choose this definition to use the compositional reasoning (defined and elaborated in the next chapter), in the application of compositional model checking. We avoid such (i.e* $L((q_1, q_2)) = (p, \neg p)$*), by introducing* $AP_1 \cap AP_2 = \emptyset$ *but to use the compositional reasoning in an assume guarantee way, we are introducing the concept of* $\mathsf{Sync}(\sigma)$ *for all the common actions (i.e* $\sigma \in \Sigma^{'} = \Sigma_1 \cap \Sigma_2$*) between two automaton in the composition automaton, as introduced in [BCC98]. So that there won't be a state in the composition product we don't have* $L((q_1, q_2)) = (\mathsf{Sync}(\sigma), \neg\mathsf{Sync}(\sigma))$ *i.e all the reachable state in the composition of automaton* $\mathscr{A} = \mathscr{A}_1 \parallel \mathscr{A}_2$*,* $(q_1, q_2) \in Q$*, if* $\mathsf{Sync}(\sigma) \in L_1(q_1)$*, then we will have* $\mathsf{Sync}(\sigma) \in L_2(q_2)$ *and vice versa.*

3 : Even though, we stated in the theorem 4.3 the intermediate specification can be formed over the atomic predicates of type $\mathsf{Sync}(\Sigma)$*, we can express the intermediate specification formed over the normal atomic predicates along with the sync of common actions set provided that those normal atomic predicates should be from global predicates set in the sense that those global predicates are common to all the automaton involves in the composition of automata and at any instant (state) of the composition of automata, all the sub-processor or automaton agrees with the global atomic predicates that is if* $a \in$ GlobalPredicates *then if* $a \in L_1(q_1)$ *then it should be the case* $a \in L_2(q_2)$ *for the state* $(q_1, q_2) \in Q_1 \times Q_2$ *of the given composition of automata* $\mathscr{A}_1 || \mathscr{A}_2$ *in the sense that if a is a global atomic predicate and given the composition of automata* $\mathscr{A}_1 || \mathscr{A}_2$*, there wont be a state* $(q_1, q_2) \in Q_1 \times Q_2$ *such that* $a \in L_1(q_1)$ *and* $a \notin L_2(q_2)$ *or vice versa.*

In our experimental section in the next chapter we use this particular remark to make the intermediate specification.

4.4 Extending Compositional Reasoning to Control Synthesis of *LTL* Specifications

In this section, we will show how to apply the compositional reasoning technique in synthesis of supervisors, especially the synthesising the regular safety property specification expressed *LTL* without next state operator. Synthesizing the *LTL* safety specification of system as mentioned in the last chapter is dealt only with a single automaton (finite state machine). The computation of the supervisors $\mathscr{S}$ are well-known in the community [CL08] for either single automaton or composition of automata when the bad state are expressed as $\mathscr{B} = \cup_{i \in \{1,..,n\}} Q_1 \times ... \mathscr{B}_i ... \times Q$.

In a nutshell, one have to compute the set of states $I_{\text{Bad}} = \mathsf{CoReach}^{\mathscr{A}}_{\Sigma_{uc}}(\mathscr{B})$ that can reach in an *uncontrollable way* the set $\mathscr{B}$; that is, this is the set of states from which some uncontrollable actions lead the system to $\mathscr{B}$ independently from the choice of the controllable actions. The supervisor $\mathscr{S}$ is then computed by disabling in any subset $E \subseteq Q$, the events that lead to I_{Bad}.

Recall the section 3.4, when it comes to find the supervisor for the *LTL* specification, reaching a particular state can't be tagged as bad state to reach rather it depends on the sequence of states are traversed before reaching that particular state, in other words recall that temporal logic like *LTL* can express not just state based property but also path specification for the system execution.

Compositional reasoning technique is a proof rule for checking the model correctness introduced by the model checking community to tackle the state-space explosion problem in the verification of models. We believe that this same technique can be used to ensure properties on a modular system with the spirit of finding the supervisors to assume some properties on each subsystem and guarantee some properties for the given global model.

We first tackle the problem of finding a supervisor as follows.

PROBLEM 4.1 *Given an automaton $\mathscr{A}$ with atomic propositions $\{\mathsf{Sync}(\sigma) \mid \sigma \in \Sigma\} \subseteq AP$, and two* LTL$\setminus X$ *safety formula ϕ_1 and ϕ_2 over $AP_1, AP_2 \subseteq AP$ compute a supervisor $\mathscr{S}$ such that $\langle\phi_1\rangle(\mathscr{A}^{\phi})^{\mathscr{S}_{\phi}}\langle\phi_2\rangle$, where $\phi := \phi_1 \Rightarrow \phi_2$. In other words compute the supervisor $\mathscr{S}^{\phi}$ such that* $\text{Run}_{\mathscr{B}^{\phi}}\big((\mathscr{A} \otimes \mathscr{M}_{\phi})^{\mathscr{S}_{\phi}}\big) = \emptyset$ ◇

THEOREM 4.4 *With the notations of Problem 4.1, and the supervisor $\mathscr{S}$ computed as in Algorithm 3.1, then $\langle\phi_1\rangle(\mathscr{A}^{\phi_1\Rightarrow\phi_2})^{\mathscr{S}_{\phi_1\Rightarrow\phi_2}}\langle\phi_2\rangle$ i.e* $\text{Tr}\big(\text{Run}(\mathscr{A}^{\phi_1\Rightarrow\phi_2})^{\mathscr{S}_{\phi_1\Rightarrow\phi_2}}\big) \vDash \phi_1 \Rightarrow \phi_2$. ◇

Proof The proof is immediate from lemma 3.1, the results of section 3.4.2 as well as the explanations given in theorem 3.3. ◇

We shall now turn our attention to the synthesis of supervisors for compositional automata based on the compositional reasoning techniques described in section 4.3.1.

PROBLEM 4.2 *Given two automata $\mathscr{A}_1$ and $\mathscr{A}_2$ with the atomic predicates in AP_1 and AP_2, and two* $\mathsf{LTL}_f \setminus \mathsf{X}$ *formulas ϕ_1 and ϕ_3 over the atomic predicates set AP_1 and AP_3, compute a supervisor $\mathscr{S}_{\phi_1\Rightarrow\phi_3}$ such that $\langle\phi_1\rangle\big((\mathscr{A}_1 \| \mathscr{A}_2) \otimes \mathscr{M}_{\phi_1\Rightarrow\phi_3}\big)^{\mathscr{S}_{\phi_1\Rightarrow\phi_3}}\langle\phi_3\rangle$.* ◇

A basic solution to this problem would be to compute the automaton $\mathscr{A}^{\phi} := (\mathscr{A}_1 \| \mathscr{A}_2) \otimes \mathscr{M}_{\phi}$ where $\phi := \phi_1 \Rightarrow \phi_3$ and compute a supervisor $\mathscr{S}_{\phi}$ such that $\text{Run}_{\mathscr{B}^{\phi}}\big(\big((\mathscr{A}_1 \| \mathscr{A}_2) \otimes \mathscr{M}_{\phi}\big)^{\mathscr{S}_{\phi}}\big) = \emptyset$, which might lead to state-space explosion since this means computing the whole system. Instead, we would like to use the result of Theorem 4.3 in order to split the computation of the supervisor into several ones in order to avoid building the whole system. In the rest of the section, we present an algorithm providing a compositonal computation of supervisors avoiding the whole computation of the system.

ALGORITHM 4.1 *Given automata $\mathscr{A}_1$ and $\mathscr{A}_2$ with the atomic predicates in AP_1 and AP_2, and two $\mathsf{LTL}_f \setminus \mathsf{X}$ specification ϕ_1 and ϕ_3 over the subset of atomic predicate sets AP_1 and AP_2 :*

- *Step1 : Guess an intermediate specification ϕ_2 over the atomic predicates set $\{\mathsf{Sync}(\sigma) \mid \sigma \in \Sigma_1 \cap \Sigma_2\} = AP_1 \cap AP_2$ (an assumptions of Theorem 4.3)*
  Compute the supervisor $\mathscr{S}_{\phi_1 \Rightarrow \phi_2}$ by solely knowing the automata and the specification $\phi_1 \Rightarrow \phi_2$ model $\mathscr{M}_{\phi_1 \Rightarrow \phi_2}$ using the algorithm 3.1 s.t $\mathrm{Run}_{\mathscr{B}^{\phi_1 \Rightarrow \phi_2}}((\mathscr{A}_1 \otimes \mathscr{M}_{\phi_1 \Rightarrow \phi_2})^{\mathscr{S}_{\phi_1 \Rightarrow \phi_2}}) = \emptyset$

- *Step2 : Based on ϕ_2, compute a supervisor $\mathscr{S}_{\phi_2 \Rightarrow \phi_3}$ by solely knowing the automata and the specification $\phi_2 \Rightarrow \phi_3$ model $\mathscr{M}_{\phi_2 \Rightarrow \phi_3}$ using the algorithm 3.1, s.t $\mathrm{Run}_{\mathscr{B}^{\phi_2 \Rightarrow \phi_3}}((\mathscr{A}_1 \otimes \mathscr{M}_{\phi_1 \Rightarrow \phi_2})^{\mathscr{S}_{\phi_2 \Rightarrow \phi_3}}) = \emptyset$*

◇

THEOREM 4.5 (Compositional Synthesis) *Under the assumption of Theorem 4.3 with the notations of Algorithm 4.1, we have that*

$$\frac{\langle \phi_1 \rangle ((\mathscr{A}_1)^{\mathscr{S}_{\phi_1 \Rightarrow \phi_2}}) \langle \phi_2 \rangle \qquad \langle \phi_2 \rangle ((\mathscr{A}_2)^{\mathscr{S}_{\phi_2 \Rightarrow \phi_3}}) \langle \phi_3 \rangle}{\langle \phi_1 \rangle ((\mathscr{A}_1)^{\mathscr{S}_{\phi_1 \Rightarrow \phi_2}}) \| ((\mathscr{A}_2)^{\mathscr{S}_{\phi_2 \Rightarrow \phi_3}}) \langle \phi_3 \rangle}$$

In other words

$$\frac{\mathrm{Run}_{\mathscr{B}^{\phi_1 \Rightarrow \phi_2}}((\mathscr{A}_1 \otimes \mathscr{M}_{\phi_1 \Rightarrow \phi_2})^{\mathscr{S}_{\phi_1 \Rightarrow \phi_2}}) = \emptyset \qquad \mathrm{Run}_{\mathscr{B}^{\phi_2 \Rightarrow \phi_3}}((\mathscr{A}_2 \otimes \mathscr{M}_{\phi_2 \Rightarrow \phi_3})^{\mathscr{S}_{\phi_2 \Rightarrow \phi_3}}) = \emptyset}{\mathrm{Run}_{\mathscr{B}^{\phi_1 \Rightarrow \phi_3}}((\mathscr{A}_1^{\phi_1 \Rightarrow \phi_2})^{\mathscr{S}_{\phi_1 \Rightarrow \phi_2}} \| (\mathscr{A}_2^{\phi_2 \Rightarrow \phi_3})^{\mathscr{S}_{\phi_2 \Rightarrow \phi_3}}) = \emptyset}$$

In other words when we have $\mathrm{Run}_{\mathscr{B}_{\phi_1 \Rightarrow \phi_2}}((\mathscr{A}_1 \otimes \mathscr{M}_{\phi_1 \Rightarrow \phi_2})^{\mathscr{S}_{\phi_1 \Rightarrow \phi_2}}) = \emptyset$,
$\mathrm{Run}_{\mathscr{B}_{\phi_2 \Rightarrow \phi_3}}((\mathscr{A}_2 \otimes \mathscr{M}_{\phi_2 \Rightarrow \phi_3})^{\mathscr{S}_{\phi_2 \Rightarrow \phi_3}}) = \emptyset$ then the trace in $\mathrm{Tr}(\mathrm{Run}((\mathscr{A}_1^{\phi_1 \Rightarrow \phi_2})^{\mathscr{S}_{\phi_1 \Rightarrow \phi_2}} \| (\mathscr{A}_2^{\phi_2 \Rightarrow \phi_3})^{\mathscr{S}_{\phi_2 \Rightarrow \phi_3}}))$ is projected to the atomic predicates AP_1 satisfies the specification ϕ_1 then the same trace projected to the atomic predicates AP_2 satisfies the specification ϕ_3. In other words the automaton $\mathscr{A}_1$ controlled by the supervisor with the model $\mathscr{M}_{\phi_1 \Rightarrow \phi_2}$ guarantee the specification ϕ_2, and the automaton $\mathscr{A}_2$ controlled by the supervisor with the model $\mathscr{M}_{\phi_2 \Rightarrow \phi_3}$ guarantee the specification ϕ_3. Hence the composition of the automata $\mathscr{A}_1, \mathscr{A}_2$ controlled by the supervisors $\mathscr{S}_{\phi_1 \Rightarrow \phi_2}, \mathscr{S}_{\phi_2 \Rightarrow \phi_3}$ with the *NFA* model $\mathscr{M}_{\phi_1 \Rightarrow \phi_2}, \mathscr{M}_{\phi_2 \Rightarrow \phi_3}$ respectively satisfy the specification $\phi_1 \Rightarrow \phi_3$.
Proof The above theorem follows from the previous Theorems 4.3, 3.3 and 4.4 ◇.

Note that this result is sound but is not complete since it depends on the choice of intermediate specification ϕ_2.

Modularity Aspects As usual, when dealing with several safety *LTL* specifications ϕ and ψ without next state operator, we can decompose the problem into computation of two supervisors to satisfy specifications ϕ and ψ.

PROPOSITION 4.1 (Modular Synthesis of *LTL* Specifications) *Given an automaton $\mathscr{A}$, and two regular safety specifications ϕ and ψ such that $AP_\phi, AP_\psi \subseteq AP$, such that there exists supervisors $\mathscr{S}_\phi$ and $\mathscr{S}_\psi$ such that $(\mathscr{A}^\phi)^{\mathscr{S}_\phi} \models \phi$ and $(\mathscr{A}^\psi)^{\mathscr{S}_\psi} \models \psi$ then the supervisor $\mathscr{S} = \mathscr{S}_\phi \cup \mathscr{S}_\psi$ is such that $(\mathscr{A}^{\phi\wedge\psi})^{\mathscr{S}} \models \phi \wedge \psi$.* ◇

The proof of this proposition is well-known for prefix-closed language-based properties [WR88] and can be easily extended to our framework.

Proof : This proposition is true can be revealed with simple arguments, but importantly it reveals how the supervisors works for the given *LTL* regular safety property specification. First the supervisor $\mathscr{S}_\phi$ for the given automaton $\mathscr{A}$ and the NFA model $\mathscr{M}_\phi$ for the given safety property ϕ can be viewed as $\mathscr{S}_\phi : Q \times S_\phi \to 2^{\Sigma_c}$ for the automaton $\mathscr{A}^\phi$ that is to say the supervisor observes the system state of the automaton $\mathscr{A}$ and updates the specification model state (S_ϕ) accordingly, and by knowing the current state of automaton $\mathscr{A}$ and model $\mathscr{M}_\phi$ it blocks the subset of controllable actions Σ_c so that the co-reach of bad states $Q \times \mathscr{B}_\phi$ never reached uncontrollably.

Similarly, the supervisor $\mathscr{S}_\psi$ is a function $\mathscr{S}_\psi : Q \times S_\psi \to 2^{\Sigma_c}$ where S_ψ is the system state of NFA model $\mathscr{M}_\psi$ of the specification ψ.

The resultant supervisor $\mathscr{S} : \mathscr{S}_\phi \cup \mathscr{S}_\psi$ is a function $\mathscr{S} : Q \times S_\phi \times S_\psi \to 2^{\Sigma_c}$, such that $\mathscr{S}(q, s_1, s_2) = \mathscr{S}_\phi(q, s_1) \cup \mathscr{S}_\psi(q, s_2)$ for all $(q, s_1, s_2) \in Q \times S_\phi \times S_\psi$ ◇

The above proposition 4.1 can be extended to conjunction of many safety properties as depicted in the figure 4.6.

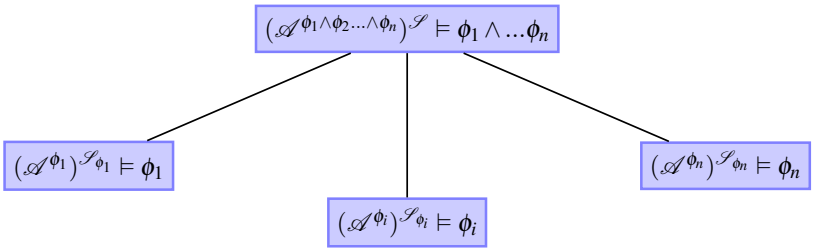


FIGURE 4.6 : Modular Synthesis of Specifications

When dealing with modular systems specifications, we have this result, borrowed from [WH91] :

PROPOSITION 4.2 (*LTL* Synthesis of Modular System) *Given two automata $\mathscr{A}_1$ and $\mathscr{A}_2$ and two safety* LTL *specifications without next state operator ϕ and ψ over AP_1 and AP_2, with $AP_1 \cap AP_2 = \emptyset$, then*

$$\frac{(\mathscr{A}_1^\phi)^{\mathscr{S}_\phi} \models \phi \qquad (\mathscr{A}_2^\psi)^{\mathscr{S}_\psi} \models \psi}{((\mathscr{A}_1^\phi)^{\mathscr{S}_\phi} \| (\mathscr{A}_2^\psi)^{\mathscr{S}_\psi}) \models \phi \wedge \psi}$$

◇

This proposition holds whenever the global property is separable into two properties that need to be ensured on each component. However, if the global property is not separable, then one

has to find some other techniques to reduce the complexity of the supervisor synthesis of this property like of compositional reasoning or any other method. ◇

The above proposition 4.2 can be extended to conjunction of many safety properties as depicted in the figure 4.7.

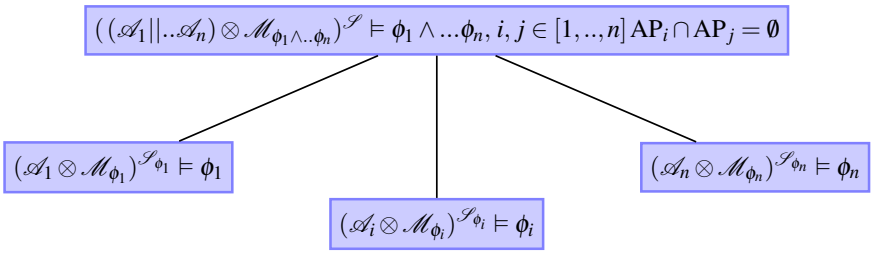


FIGURE 4.7 : Specification Synthesis of Modular Systems

State Space Reduction Techniques for Control Synthesis of Distributed Finite State Machines

In this section we will presents the literature results about solving the state space explosion problem in computing the set of supervisors for the given distributed system *DS* using the modular nature of the system and properties in terms of the language.

Given a distributed system *DS* modeled as a set of finite state machines (*FSM*) $\mathscr{A}=\|_{i\in\{1...n\}}\mathscr{A}_i$ as in definition 3.16. Computing the global supervisor $\mathscr{S}=\{\mathscr{S}_1,\mathscr{S}_2...\mathscr{S}_n\}$ remember from last chapter section 3.4, for finding local supervisor $\mathscr{S}_i$ for each *FSM* $\mathscr{A}_i$, we have to use global state estimator E_i, this procedure is somewhat expensive in the sense that when n is too large then the computational steps and memory needed for each state estimator E_i can grow dramatically [WR88]. In order to avoid such drastic state space explosions, we can use the concurrent nature of given distributed system *DS* and modular property of set of requirements.

In the rest of this section we will show the literature in this aspect in some details but not comprehensive.

Modular synthesis by Property [WR88] For the given system model as FSM $\mathscr{A}$. Consider the safety property as a set of words formed over the action alphabet set Σ. Producing synthesis inductively uses the fact that often the desired behavior is specified with a collection of requirements i.e collection of $\{\phi_1,\phi_2...\phi_n\}$ prefix closed properties $\{\mathscr{L}_1,\mathscr{L}_2,...,\mathscr{L}_n\}$. Each ϕ_i express the condition on the language in the sense that $\mathscr{L}_i\subseteq\Sigma$ so that $\forall w\in\mathscr{L}_i w\vDash\phi_i$. In producing control synthesis of such constraint instead of finding the supervisor such that $\mathscr{L}(\mathscr{A}^{\mathscr{S}})\vDash\bigwedge_{i\in\{1,2,..,n\}}\phi_i$. Finding the supervisors $\mathscr{L}(\mathscr{A}^{\mathscr{S}_i})\vDash\phi_i$ (which blocks the controllable actions as defined in definition 3.14) for each $i\in\{1,2,..,n\}$, so that resulting supervisor will be $\mathscr{S}=\{\mathscr{S}_1,\mathscr{S}_2,...,\mathscr{S}_2\}$ acting on the FSM $\mathscr{A}^{\cap_{i\in\{1,2,...,n\}}\mathscr{S}_i}$ where $\mathscr{S}(q)=\cup_{i\in\{1,2,...,n\}}\mathscr{S}_i(q)$ such

that $\mathscr{A}^{\mathscr{S}} \vDash \bigwedge_{i\in\{1,2,..,n\}} \phi_i$. Such techniques were already used successfully in the synthesis of supervisory controllers. For instance please refer [Goo+21].

PROPOSITION 4.3 *Given a* FSM $\mathscr{A}$*, and two safety critical prefix language* $\mathscr{L}_1, \mathscr{L}_2$*. If there exists supervisors* $\mathscr{S}_1$ *and* $\mathscr{S}_2$ *such that* $\mathscr{A}^{\mathscr{S}_1} \subseteq \mathscr{L}_1$ *and* $\mathscr{A}^{\mathscr{S}_2} \subseteq \mathscr{L}_2$ *then there exists* $\mathscr{S}$ *such that* $\mathscr{L}(\mathscr{A}^{\mathscr{S}}) \subseteq \mathscr{L}_1 \cap \mathscr{L}_2$.

Proof argument The supervisor $\mathscr{S} := \cup_{i\in\{1,2\}} \mathscr{S}_i$ will block the controllable actions already blocked by supervisor $\mathscr{S}_1$ and $\mathscr{S}_2$ for the given state $q \in Q$ of the *FSM* $\mathscr{A}$. Hence the proposition. ◇

Modular Synthesis by System When dealing with modular systems, we have this result, borrowed from [WH91]. Given composition of FSM $\mathscr{A} = \|_{i\in\{1,..,n\}} \mathscr{A}_i$, and the set of safety prefix closed languages $\mathscr{L}_i \subseteq \Sigma_i^*$, in order to find the supervisor such that $\mathscr{A}^{\mathscr{S}} = (\|_{i\in\{1,..,n\}} \mathscr{A}_i)^{\mathscr{S}} \vDash \bigwedge_{j\in\{1,..,n\}} \phi_j$. Note that each specification language $\mathscr{L}_i$ is the property of local actions Σ_i.

PROPOSITION 4.4 *Given two FSM* $\mathscr{A}_1$ *and* $\mathscr{A}_2$ *and two safety prefix closed languages* $\mathscr{L}_1$*,* $\mathscr{L}_2$*, with* $\mathscr{L}_1 \subseteq \Sigma_1^*$*,* $\mathscr{L}_2 \subseteq \Sigma_2^*$ *and* $\Sigma_1 \cap \Sigma_2 = \emptyset$*, then*

$$\frac{\mathscr{L}(\mathscr{A}_1^{\mathscr{S}_1}) \subseteq \mathscr{L}_1 \qquad \mathscr{L}(\mathscr{A}_2^{\mathscr{S}_2}) \subseteq \mathscr{L}_2}{\mathscr{L}(\mathscr{A}_1^{\mathscr{S}_1} \| \mathscr{A}_2^{\mathscr{S}_2}) \subseteq P_{\Sigma_1}^{-\Sigma_2}(\mathscr{L}_1) \cap P_{\Sigma_2}^{-\Sigma_1}(\mathscr{L}_2)}$$

◇

Proof argument is trivial, as we see the property $\mathscr{L}_1$ is local to the FSM $\mathscr{A}_1$ similarly property $\mathscr{L}_2$ is local to the FSM $\mathscr{A}_2$ if we have supervisors $\mathscr{S}_1, \mathscr{S}_2$ such that $\mathscr{L}(\mathscr{A}_1^{\mathscr{S}_1}) \subseteq \mathscr{L}_1$ and $\mathscr{L}(\mathscr{A}_2^{\mathscr{S}_2}) \subseteq \mathscr{L}_2$. The intersection these two languages will be $P_{\Sigma_1}^{-\Sigma_2}(\mathscr{L}_1) \cap P_{\Sigma_2}^{-\Sigma_1}(\mathscr{L}_2) \subseteq \Sigma^*$ where the action set $\Sigma = \Sigma_1 \cup \Sigma_2$.

Take a generic $w \in \mathscr{L}(\mathscr{A}_1^{\mathscr{S}_1} \| \mathscr{A}_2^{\mathscr{S}_2})$ by lemma 4.2 $P_{\Sigma_i}(w) \in \mathscr{L}(\mathscr{A}_i^{\mathscr{S}_i})$ for $i \in \{1,2\}$.
If $P_{\Sigma_1}(w) \in \mathscr{L}_1$ then $w \in P_{\Sigma_1}^{-\Sigma_2}(\mathscr{L}_1)$ by lemma 4.1. Similarly if $P_{\Sigma_2}(w) \in \mathscr{L}_2$ then $w \in P_{\Sigma_2}^{-\Sigma_1}(\mathscr{L}_2)$
Hence the proposition.

4.5 Chapter Conclusion

Compositional Reasoning for Distributed System : Nature of the distributed system makes the collections of computing entities work independently for their own specified task and make them communicate with each other to meet the specific global application or goal. In this task, expecting every component to move at the same speed or move together coherently is too ambitious because of intrinsic computing power, diversity of hardware, and the difference in kind

of procedures involved for each assigned function. Naturally, *DS* can't be taken as one single computing entity and at the same time we want *DS* to fulfill the global application needs. In this setting, it is very natural to expect that each entity does and satisfies the specific assigned task and satisfies specific local goals. When each such entity satisfies the specific local goals then as a *DS* designer we can achieve the required application (global goal) requirements by setting the communication between those individual computing entities. Compositional reasoning exactly makes the distributed system achieve the common global goal by making each entity satisfy specific local specifications.

For the given distributed system modeled as composition of finite state machines i.e automaton, in order to verify the global specification ϕ on the given distributed system, instead of computing the global system as $\mathscr{A}_{\text{DS}} = \mathscr{A}_1 || \mathscr{A}_2$, we keep only the local system as $\mathscr{A}_1, \mathscr{A}_2$ and inferring the global specification ϕ by introducing the local specification ϕ_I.

One can note that (4.5) is of limited to 2 components but can be used for several subsystems with a methodology as depicted in the figure 4.8.

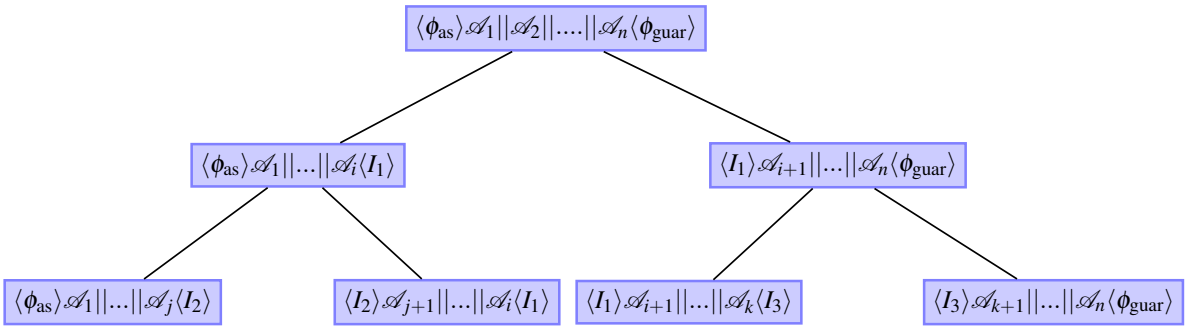


FIGURE 4.8 : Recursive usage of Compositional Reasoning

The intermediate specifications I_1, I_2, I_3 mentioned in the figure 4.8 should be formed over $\{\mathsf{Sync}(\sigma) \mid \sigma \in (\cup_{p\in[1,..,i]}\Sigma_p) \cap (\cup_{q\in[i+1,..,n]}\Sigma_q)\}$, $\{\mathsf{Sync}(\sigma) \mid \sigma \in (\cup_{p\in[1,..,j]}\Sigma_p) \cap (\cup_{q\in[j+1,..,i]}\Sigma_q)\}$ and $\{\mathsf{Sync}(\sigma) \mid \sigma \in (\cup_{p\in[i+1,..,k]}\Sigma_k) \cap (\cup_{q\in[k+1,..,n]}\Sigma_q)\}$ respectively.

Advantage to Verification and Synthesis of *SDN* There is a natural reason in applying Compositional reasoning rules to Software defined networking in the case of verification and synthesis process, as we know that verification and synthesis struggles a lot to complete the process because of state space explosion. Even though there are a couple of techniques to avoid the state space explosion, it's really difficult to apply them while you are encoding your system of interest either as a symbolic transition system definition 3.5 or definition 3.1. In order to use the techniques like abstraction or partial order reduction techniques or modular aspects, we have to write the code (modeling of our system) more explicitly on the kind of abstraction or partial order techniques we are intended to use for our system verification and synthesis process. Sometimes we have to prove (manually) that the applied abstraction or partial order techniques are

correct to the system we are interested in. Even when we want to use compositional reasoning, we have to describe the intermediate specification. That is to say to verify $\langle \phi_{\text{as}} \rangle \mathscr{A}_1 \parallel \mathscr{A}_2 \langle \phi_{\text{guar}} \rangle$, we have to find the intermediate specification ϕ_I so that we can verify the whole system correctness by checking the sub properties (local specification) $\langle \phi_{\text{as}} \rangle \mathscr{A}_1 \langle \phi_I \rangle$ and $\langle \phi_I \rangle \mathscr{A}_2 \langle \phi_{\text{guar}} \rangle$. Although, as we are human, when we build the distributed system *DS*, we do know what the system properties we are trying to resolve by building the *DS*. This knowledge for sure will give the hint at intermediate specification (formula) when we want to use the compositional reasoning techniques in the verification and synthesis process. We have to recall that, *SDN* is the composition of three layers, Manager, controller and the data plane layers. Have to be aware of why we build such layers and what we expect from each layer, which will give us hints at producing the intermediate specification in the process of verification of *SDN* with the help of compositional reasoning technique.

Guessing the specification ϕ_I as mentioned in the algorithm 4.1, it is not exactly guessing, it's all about formulating appropriate intermediate specifications. Usually it is attached to the design principle not really in the logical construction. By using the compositional reasoning it is safe enough to bring the intermediate specifications between *SDN* manager and controller as well as between *SDN* and data plane devices (i.e clients) since there is no direct communication between *SDN* manager and data plane devices. In case of synthesis, we can also hugely benefit from the modularity aspects such use can be trivially visible in our synthesis experiment case as mentioned in this document chapter 6 and also used in chapter 5.

CHAPITRE 5

FORMAL VERIFICATION SCHEME FOR NOKIA SDN-IOT PLATFORM

Before diving into our formal verification scheme for the *SDN*-based *IoT* platform designed by Nokia Bell Labs, we will review some of the existing verification analysis of *SDN*-based systems.

The aim of the following subsection is not to be exhaustive on the topic which is both very active and very large, but to describe the most important works as per my knowledge and close enough to the present manuscript in terms of SDN verification.

5.1 Existing Modelisation and Verification of SDN systems

The closest to our work is that of [MDW14b] which presents a tool called *Kuai*. The latter applies optimizations based on *partial-order reduction* to simplify (more exactly to reduce the state space explosion in verification process) models of SDN protocols, and presents several model checking benchmarks to assess the performance of its optimizations. The tool translates formal models given in the *Murphi* format and uses *PReach* [Bin+10] for distributed model checking. In [El-+16], the authors study the concurrency in the semantics of SDNs and present the tool *SDNRacer* which is a dynamic analyzer. Algorithms based on an extension of *Petri nets* for verifying network configurations and concurrent updates were given in [Fin+19]. Several other works consider the formal analysis of SDN-based systems. For instance, [KVM12] provides a static analysis of header spaces which detect inconsistencies in routing tables. However, the work does not consider more dynamic behaviors and complex issues due to interleavings as it is done in [MDW14b] and in our work.

[MDW14b] references to KUAI and solves the protocol correctness using the partial order reduction techniques. It is superior to the previous one because it models *SDN* as an asynchronous system. FIFO queues and barrier messages in switch incoming message channels (from the controller) are particularly used.

A similar approach was adopted in [Mai+11]. A modeling language called *FlowLog* tailored for

SDN-based systems was given in [Nel+13]. In this work, the authors present model checking experiments with the Spin model checker [Hol04] but no particular optimizations are presented. In [Can+12], model checking and symbolic execution are combined in order to check the most popular *SDN* protocol *OpenFlow* (*OF*) for a given number of packets. The paper [SNM13] presents data and network abstractions applied on *SDN* models combined with a manual refinement process based on *non-interference lemmas*.

In [Bal+14], the authors develop a framework where the system is modeled using first-order logic, and user-provided inductive invariants are checked to prove correctness. This allows for checking the system correctness for all network topologies, and for an unbounded number of exchanged packets. The approach developed in [Khu+13] consists in checking the effect of rule updates in real-time, without affecting the system performance. This is complementary to *off-line* verification approaches which analyze the system globally before execution. Abstractions and other transformations that preserve the properties of networks for rendering the verification are presented in [Plo+16].

5.1.1 VERIFLOW

Verifying Network-Wide In-variants in Real Time The paper [Khu+13] discusses the possibility of checking network-wide in-variants in real time, as the network evolves. An example of invariants property consists in checking the absence of routing loops in the data plane. This would enable it to check updates before data plane rules update its flow table, thus raising alarms, or even prevent bugs as they occur by blocking problematic changes (updates). For such real time verification the automated techniques should operate on timescales of seconds otherwise the automated verification introduces latency between the network updates computed by the *SDN* controller and while its deployment the same in the data plane. That is to say that delaying updates for processing (automated checks) can harm consistency of network state,and reduce reaction time of protocols with real-time requirements such as routing and fast failover.

In this verification process, an *SDN* system comprises at a high level :[1] a standardized and open interface to read and write the data plane elements such as switches and routers [2] a controller, logically centralized element that can run the custom code and it is responsible for transmitting commands to the data plane elements. *Veriflow* consists of a shim layer between the controller and the network data plane as depicted in the figure 5.1. It observes all required forwarding rule modifications performed by the *SDN* controller, and has full knowledge of data plane element forwarding tables.

That is controller changes dynamically the network routing rules based on the request made by the host, and get verified by the *Veriflow* (real-time verification process) that changes made by the SDN controller do satisfy the network-wide invariants before applying the changes to the

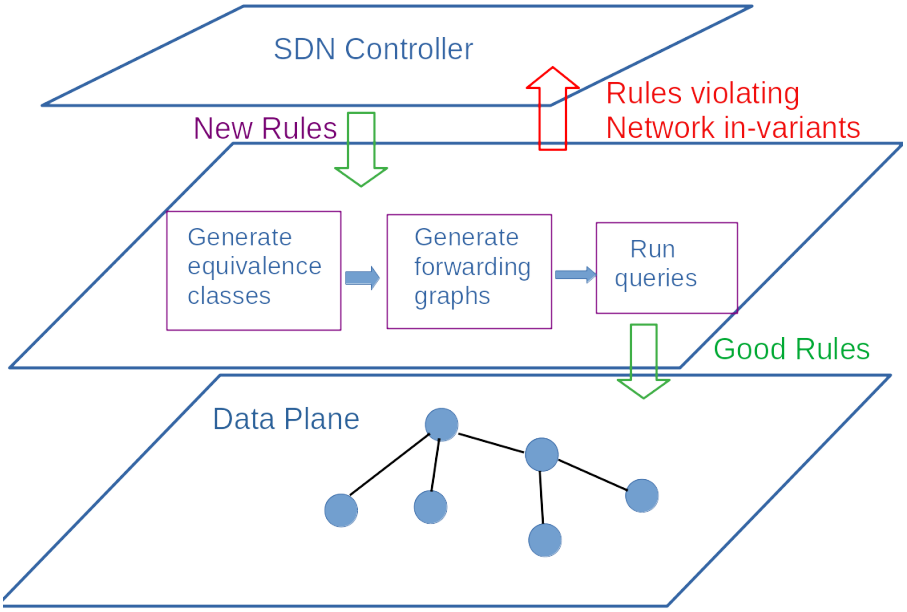


FIGURE 5.1 : VeriFlow : Checks the SDN controller updates rules on the data plane.

data plane elements.

Working mechanism of VeriFlow involves three steps :

1 **Generating the Equivalence classes of data packets** : First, the network is sliced into a set of equivalence classes of packets (say *EC*). Packets belonging to an *EC* experience the same forwarding actions throughout the network. Intuitively, each change in the network will typically only affect a very small number of equivalence classes. Therefore, finding the set of equivalence classes whose operations could be altered by a new rule, and verifying network invariants is only reduced to those classes.
EC is a set P of data packets s.t for any $p_1, p_2 \in P$ and any data plane network elements S (like switches), the forwarding action is identical for p_1, p_2 at S. Separating the entire packet space into individual equivalence classes allows VeriFlow to pinpoint the affected set of packets in case a bug is discovered.
Veriflow uses the prefix-tree or tire data structure (tree type data structure for strings) to keep and update the set of equivalence classes in the data plane. More detail about the tire data structure and maintaining the equivalence classes can be found in the book [Var05] chapter 11 and 12.

2 **Generating or Updating the Forwarding graphs** : VeriFlow builds individual forwarding graphs for every *EC* using the current data network state.
Veriflow has a forwarding graph for each equivalence class computed in the previous step. This forwarding graph is a representation of how data packets within an *EC* will be forwarded. The set of nodes and directed edges in this forwarding graph represents the set of data plane elements and forwarding actions to corresponding equivalence. There

is a directed edge from node u to node v if according to the forwarding table at network elements A represented by the node u, the next hop (i.e network switch or router) for the equivalence class is network elements B represented by the node v. For generating the forwarding graph for each equivalence class, the Veriflow traverses the tree structure (abstracted data tree for each equivalence class) to find the data plane network elements and rules that match packets from that equivalence class, and builds the graph using this information.

3 **Run Queries** : VeriFlow traverses these forwarding graphs to determine the status of the invariants for various equivalence classes. Here, veriflow won't check all the forwarding graphs, but the forwarding graphs changed or modified for the new set of rules provided by the SDN controller.

5.1.2 KUAI

Labelled Transition Model for checking the *SDN* Property Correctness in Offline Unlike *Veriflow*, *Kuai* is not a real-time verification but an offline verification process. The main hurdle in the verification [MDW14b] of SDN controller algorithms within a given network topology (to prove correctness of working SDN controller and controllable network topology of switches or finding the bugs in the SDN controller) is scalability since the state space grows very quickly. This well-known problem in model checking community[Cla08] which is often mentioned as the *state-space explosion* issue. However, one does not need to verify all possible sequences of orders of update or movement of data packets within the network if the events are independent. Partial order reduction techniques as already seen in the previous chapter could be applied to reduce the verification state space.

An *SDN* switch contains many input and output ports. It receives and transmits data packets with a header which has information about source and destination addresses. Let say a switch S_1 receives a packet pkt in the port pt (say) number i, then switch S_1 should have some forward rules containing the info about where to send this data packet pkt. These rules are installed with the instruction from the SDN controller. So, each switch contains some set of rules, where each rule contains the packet header type, incoming port number and the outgoing port number with some priority. A switch on a single packet header arrived in port i may have many matching rules. Switch chooses the highest priority rule (by *best-match* mechanism) to match the data packet and action rule from its flow table.

Whenever the switch lacks the rule for the specific data packet header and such a data packet is waiting in that switch, it asks the controller for a rule to process this data packet. Each switch S contains the set of incoming ports, outgoing ports, and set of data packets waiting in the queue pq (packet queue) to be processed. It also contains the set of forwarded messages in the output port of the switch denoted as fq (forward queue). Apart from the data packets, switch contains

the list of control commands forwarded by the SDN controller waiting in the controller queue cq, set of installed rules called flow tables ft. It also has a boolean variable *wait* (a switch internal action to wait to process the data packets before updating the controller's new forward rules.

Each switch picks one of the data packet in the packet queue pq and perform the bestmatch$(\text{sw}, \text{rules}, \text{pkt})$, it outputs the highest priority rule existed in the flow table of that switch for the given packet header. If there is no match (i.e no rule existed) it sends the no-match(sw, pkt) to the SDN controller and enters the *wait* mode i.e set wait $= 1$. If there is a match found by *bestmatch* mechanism (i.e. result will be the matching rule with possible highest priority rule) then fwd (i.e. forward the packet based on this *bestmatch*).

So the controller stores each such request from the switch or hosts in its queue, picks them in some fashion either some pre-fixed order or polling mechanism or even some random order based on the controller updating algorithm. The controller replies with commands like *add* (to add the rule in the existed flow table of that switch), *delete* (to delete the existed rule if it is there in the flow table of that switch) for the request made by various switch in the data plane networks.

Based on the command sequences in the controller queue cq, the switch performs *add* or *delete* the rules in its forwarding table. There is a special command from the SDN controller to the switch denoted as *Barries*. If this *Barries* command is sent to a switch, then that switch has to perform all the command sequence queues from the cq until the first *barries* message. During this operation the switch does not forward any data packet. *Kuai* also uses the following optimization techniques apart from the partial order reduction to reduce the state space and complete the verification process.

Barrier Optimization

When a switch has *barrier* control message in its control queue cq, then it's not really matter in which order the switch updates the control commands upto the first barrier control command, so no need to check which order a switch executing the commands instructed by the controller of this switch. This reduces the state space by not considering the switch updates the command sequence up to the first *barrier* command.

Client Optimization

When two different switches receive the packet in one of its ports pt. It is not important to know which switch received the packet and stored it in its incoming data packet queue pq first and second. These are independent actions in the network topology. This will make us not consider which order the switches receive the data packet in the verification process.

All Packets in One Shot Abstraction

When a switch has a number of rules which match a given set of data packets in its packet queue pq, then it considers all the match actions for this data packet set in one shot instead of considering the match actions for each individual packet separately. This contributes to reducing the state space in the verification process.

Controller Optimization

The idea here is while considering the *intent* requests from data plane switches, no need to consider which order the controller performs the calculation for the incoming *intent* request in the incoming controller queues. Controller takes one request at a time, analyze it. Further controllers evaluate the required flow tables changes one at a time by knowing the existing flow tables rules of the data plane network.
Apart from the previous listed optimization and abstraction, *Kuai* also uses $(0, \infty)$-abstractions on packet queue contents which consist in storing whether a given packet is absent (0) or present an unknown number of times (∞).

In a nutshell, *Kuai* consists in using the *partial order reduction* technique as well as various queue abstraction and optimization techniques to verify the security properties of SDN systems.

In our work, we model the *SDN* system almost the same way as this *Kuai* technique but we use the compositional reasoning technique (introduced later in this chapter) to reduce the state space explosion problem in the verification process. The use of compositional reasoning is more natural for *SDN* system (explained later in this chapter), which is new compared to [MDW14b]. Moreover, we consider models with a management plane which renders systems to be analyzed more comprehensive and more complex.

5.1.3 VERICON

Vericon [Bal+14] is also an offline verification process. It uses a different framework as compared to *Kuai*. *Vericon* does not take the account of latency between the *SDN* controller and the data plane. That is *vericon* can be used to check the *SDN* system correctness with respect to specifications which are not affected by the latency between *SDN* controller and the data plane. (because in *Vericon* there is no concept of queues between any pairs of *SDN* entities). *VeriCon* claims to be the first solution capable of verifying that an *SDN* program is correct on all admissible topologies and for all possible (infinite) sequences of network events. **VeriCon, unlike finite state model checking, verifies that the invariants (i.e specification) hold under any admissible network topology of any size. By default, the admissible topologies are all the possible network graphs.**

Vericon either confirms the correctness of the controller program on all admissible network topologies or outputs a concrete counter-example. Vericon uses *first-order logic* (*FOL*) to specify admissible network topologies and desired network-wide in-variants. It implements classical *Floyd Hoare-Dijkstra deductive* verification using *Z3* software tool (an application in *Python* programming language). The problem with *Vericon* approach is that both switch and controller events are executed atomically (i.e synchronous updation). This assumption won't hold if there is a limited bandwidth between the data plane and controller plane (or having non-negligible latency between data plane and controller plane) in general *SDN* system. *Vericon* approach is an efficient method to check the correctness of *SDN* system correctness when there is no problem of latency between controller and data plane. With *Vericon* techniques, one has to be careful that *First order logic* can express network topology invariants but it can't express the connectivity of the arbitrary size of network topology. In the sense that for each network topology one has to introduce the specific atomic predicates to express the connectivity of the given network.
For example, the invariant $T_3 := \text{rcv}^{\text{this}}(S,P,I) \implies \text{path}(S,I,\text{P.src})$ meaning packets arrive from reachable hosts, which asserts that packets cannot be received from disconnected hosts. In this expression $\text{path}(S,I,\text{P.src})$ its an atomic predicate to express the existence of a path between the switch S and the port I. The authors of *VeriCon* are very well aware about the limitation of the expressiblity of *FOL*, made a statement in the experimentation part of the paper [Bal+14] (not in the background theoretical part) that for a given topology invariant expressing the graph connectivity, they made a library of possible networks (i.e. wrote explicitly, not generated by the software tool) for the given topology in-variants.
For the expressed topology in-variants, *Vericon* verifying *non-blockingness* of the data packets (connectivity between the two secure hosts in the data plane) and *safety* property (only trusted host can send the data and reach other hosts).

5.2 Model checking Tool

In order to do model checking of the system to verify the system correctness with respect to the specification, we need to build the abstract model of the respective system, there are quite a lot of model checking tools that are available among them *SPIN* [Hol97], *NuSMV* [Cim+02] are the most used tools in academic literature. The choice of choosing SPIN model checking tool will be made clear after knowing the model describer language *Pro mela* for the SPIN model checker in the following subsection.

5.2.1 Promela Language

Pro Mela is an acronym for Process Meta-Language. The specification language is intended to make it easier to find good abstractions of systems designs. Promela is not meant to be an

implementation language but a systems description language. The features of promela are intended to facilitate the construction of high-level models of distributed systems which supports non-deterministic control structures, rich set of primitives for inter-process communication. A Promela model is constructed from three basic types of objects *Processes*, *Data objects*, and *Massage channels*.

Processes

Processes are instantiations of *proctypes*, and are used to define behavior of each subsystem of a model.

Let see couple of examples which are taken from book "The SPIN MODEL CHECKER"[Hol97].

EXAMPLE 5.1 *Creating two similar processes*

```
active [2] proctype you_run()
{
  printf("my pid is : ", _pid)
}
```

This creates two processes of type you_run and each process prints its activation pid id a unique pid number for each process.

EXAMPLE 5.2 *Creating two different processes which communicate via global variable with provided functionality*

```
 bool toogle = true     /* global variables*/
short cnt;     /* visible to A and B */

active proctype A()     provided (toggle == true)
{
L : cnt++;     /* means : cnt = cnt +1*/
    printf("A : cnt = %d",cnt);
    toggle = false;     /* yield control to B */
    goto L
}
active proctype B()     provided (toggle == false)
{
L : cnt--;     /* means : cnt = cnt -1*/
    printf("B : cnt = %d",cnt);
    toggle = true;     /* yield control to B */
    goto L
}
```

A process cannot take any step unless its provided clause evaluates to true. An absent provided clause defaults to the expression true, imposing no additional constraints on process execution.

Data objects There are only two levels of scope in Promela models : global and local process. Scope of local variables cannot be referred by another process. Basic data types are *bit* $(0,1)$, *bool* (true, false), *byte* (0...255), *chan* (1..255), *mtype* (1..255, same as *enum* type in C programming), *pid* (0...255), *short*, *int*, *unsigned* are same as in C programming. From these basic data types one can also define derived data structures same as *struct* in C programming.

Message Channels Message channels are used to model the exchange of data between processes. They are declared either locally or globally. For example, in the declaration message channel chan quene_name $=[16]$ of $\{$ short, byte, bool $\}$ the type name *chan* introduces a channel declaration. In this case, the channel is named 'queue_name' and is declared to be capable of storing up to sixteen messages. And each message has three fields : the first is declared to be of type *short*, the second is of type *byte*, and the last is of type *bool*. The statement 'queue_name !expr1,expr2,expr3' sends a message with the values of the three expressions listed to the channel that we just created. Similarly for reading the message from the queue 'queue_name ?var1,var2,var3' So we have seen the FIFO bounded queues definition between any two processes. For the *Rendezvous* communication between any two processes, we have to define the communication link as (for example of communicating byte type values) 'chan quene_name $=[0]$ of $\{$ byte $\}$'

Control Flow : Compound Statements There are various types of compound statements in Promela language : some of them are Atomic sequences, Selections, and Repetitions sequences. *Atomic Sequences :* The simplest compound statement is the atomic sequence. Within a scope of atomic sequence all the execution steps will be executed continuously and considered as one instant i.e in the interleaving of process executions, no other process can execute statements when the statements are executed from defined atomic sequences.

Selection : Using the relative values of two variables say a,b, we can define a choice between the execution of two different options with a selection structure, as follows :

if
$::(a!=b) \rightarrow$ options$_1$
$::(a==b) \rightarrow$ options$_2$
fi

Only one sequence from the list will be executed based on the truth of the guard statement. If more than one guard is true in the list, then one of them will be randomly chosen and the corresponding statement will be executed.

Repetitions sequences : In this sequence a particular blocks of statement will be executed repetitively, the respective code will be

do
{
: : guard$_1$ −> statement$_1$
: : guard$_2$ −> statement$_2$
.....
: : guard$_n$ −> statement$_n$
}
do

Only one option can be selected for execution at a time. After the options complete, the execution of the repetition structure is repeated. In order to escape from this repetition of the above code, we have to use the Escape sequence command *break* and write with a guard condition in the above code. For further details of promela language one can refer [Hol04] and "The SPIN MODEL CHECKER " book written by the tool developer Gerard J. Holzmann. In this book you also find the notes on how to install the SPIN software (which is open source !).

5.2.2 SPIN

The tool that we will use to check our model is SPIN, and the specification language that it accepts is called promela. The name S PIN can be used in two basic modes : as a simulator and as a verifier. In simulation mode, SPIN can be used to get a quick impression of the types of behaviour that are captured by a system model, as it is being built.

Here we will an example to see how one can use SPIN for simulation and verification of the model written in Promela language. For more detail one can refer [Hol04]. The following example are taken from the same book "The SPIN MODEL CHECKER" [Hol97] but recreated by the author while learning the tool.

EXAMPLE 5.3 ***Mutual Exclusion : Dekker's Algorithm*** *Designing correct coordination schemes for asynchronously executing processes is the classic mutual exclusion problem. The problem here is to find a way to grant mutually exclusive access to a shared resource while assuming only the indivisibility of read and write operations. That is, to solve this problem we cannot assume the availability of atomic sequences to make a series of tests and set operations indivisible.*

You can notice in the verification result (right in the figure 5.3) that the written Mutual exclusion algorithm (model left in figure 5.3) does satisfies the expressed specification (a assert*) statement in the model code.*

```
bit turn;
bool flag[2];
byte cnt;
active [2] proctype mutex()
{
  pid i, j;
  i = _pid;
  j = 1 - _pid;
again :
  flag[i] = true;
  do
  : :  flag[i] ->
      if
      : : turn == j -> flag[i] = false;
      : : turn != j -> flag[i] = true;
      : :  else -> skip
      if
  : :  else -> break
  od;
  cnt++;
  assert(cnt == 1);
  cnt--;
  turn = j;
  flag[i] = false;
  goto again
}
```

```
Spin -a mutex.pml
gcc -o pan pan.c
./pan
(Spin Version 6.5.0 – July 2019)
        Partial Order Reduction
Full statespace search for :
        never claim -(none specified)
        assertion violations +
        acceptance cycles -(not selected)
        invalid end states +
State-vector 28 byte, depth searched 51, errors : 0
        some statistics
Stats on memory usage (in Megabytes) :
        some statistics
unreached in proctype mutex
        mutex.pml :33, state 24, "-end"
        (1 of 24 states)
pan : elapsed time 0 seconds
```

FIGURE 5.2 : Mutual Exclusion Algorithm model and Verification result

5.3 Nokia-SDN platform

In this section, we build a formal and abstract model of an SDN platform [Bou+15] which is taken here as a case study object. We think that this SDN platform has a representative architecture to make our study as generic as possible.

5.3.1 Architecture Building Blocks

As most SDN platforms, the Nokia SDN platform [Bou+15] consists of three layers, namely a management plane, a control plane and a data plane. The management planes are formed of entities called 'managers'. Each manager has the role of interfacing with the user and of converting the user's intents into high level network policies. The control plane consists of entities called 'controllers' which have the role of converting managers' high level network policies into fine grain network rules and of enforcing those rules onto the data plane. The data plane consists of network elements embodied in the selected platform as Open vSwitches (OVS) [Fou16a]. OVSes are open-source OpenFlow switches [Fou14] controlled by SDN controllers.

The platform is made of several administrative domains with each domain built up with one manager, one controller and several switches (i.e. OVSes). A given switch can only be part of one unique domain so that switches of all domains form a partition of the overall data plane. In order to simplify the controller algorithm in the forwarding of multicast/broadcast messages, the data plane topology of each domain is loop-free (the LLDP and the spanning-tree mechanisms should be added to the controller logics if non loop-free topologies are considered). In this document, such a topology is a tree topology departing from a root switch. The latter is the domain border switch. Root switches of all domains are interconnected together in a full meshed topology so that there is only one network hop between any pairs of root switches. Figure 5.3 provides us with a schematic representation of the SDN platform.

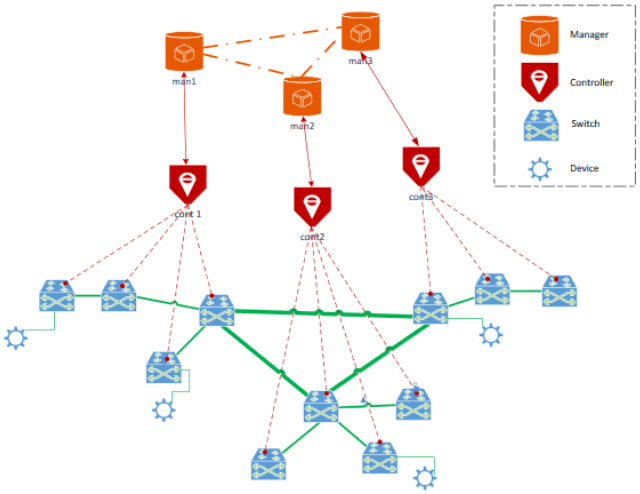


FIGURE 5.3 : SDN Platform Description

The data plane is also completed with devices. Each of them can be connected to a switch port called as an *Access-port* via a network link called as an *Access-link* (thin green links). Similarly, switches within a domain are connected together via network links called as *Intra-links* (medium-size green links) thanks to their *Intra-ports*. Root switches from different domains are interconnected together via *Inter-links* (thick green links) thanks to their *Inter-ports*.

5.3.2 The User's Intent

The user's intent consists in a set of predicates on device characteristics or device capabilities [PBB17]. This set of predicates or intent allows the user to select devices and gather them into a private group called in the following sections as *Virtual Space* or *VS*. Devices of the same *VS* have to be connected together by controllers on the different domains where these devices are detected (i.e. via the MAC learning process). A *VS* is enforced by the control plane as a network slice in the data plane. While enforcing OpenFlow rules, controllers should make sure that a pair of devices which do not belong to any common *VS* are not connected together. This is defined as the network slice isolation property or isolation property for short (i.e. data traffic isolation between network slices) in this document. For the sake of simplicity, the user's intent will be presented in the rest of the document as a nominative list of devices - e.g. $\{d1, d2, d3\}$. This is done without losing the genericity of our discussions since the present verification work is not focusing on the consistency of the overall users' intents but rather on the safety of the underlying SDN mechanisms. Moreover, we do not present details on how the controller which identifies a given device by its MAC address, and the associated manager which identifies the same device by its name, agree with each other on the device identity, as per the limited size of the document. Please refer to [Bou+15] for further information.

5.3.3 Device Discovery via MAC Learning

The controller implements the well-known MAC learning mechanism as per [Fou16b]. Thanks to this mechanism, the controller can discover newly connected device (e.g. new MAC address) and inform the associated manager. In return, it receives from the manager high level network policies (derived from users' intents) regarding the new device.

Thus, a MAC learning OF Table (e.g. Table 10) is created by default on each switch by the domain controller. The default rules with low priority (e.g. priority 0) in this OF Table consists in forwarding any packet from an 'unknown' source MAC address to the controller for processing but also in multicasting the same packet to all neighbor switches of the domain except to the switch where the packet comes from. Such a multicasting process is called *flooding* and the implied packet is called the *MAC learning* packet in this document. In a first step such *flooding* is limited to the domain. It allows for all switches in the domain to receive the *MAC learning* packet once. This latter is then forwarded by each switch as per the aforementioned default rules

to the controller in the form of an OF PacketIn message. Thanks to such OF PacketIn messages, the controller 'learns' the 'unknown' source MAC address 'A' together with the incoming OF port 'P' of the given switch. Port 'P' is marked by the controller as the relative position of the new device (identified by MAC address 'A') with regards to the switch. It can then insert high priority (e.g. priority 1) OF rules into the OF Table (of the given switch) in order to forward now on any packet destined to 'A' to port 'P'. This is done for both unicast and multicast layer-2 packets. The controller also notifies the associated manager that it has detected and located a new device. In return, it receives from the manager high level network policies (derived from users' intents) implying the newly discovered device.

Once the new device has been discovered by the domain controller, the latter 'leaks' multicast/broadcast packets from the former to all neighbor domains where devices of the same VS are located (and only to those domains). Such packets are treated as MAC learning packets in such neighboring domains, meaning that they are flooded within the aforementioned domains except to their *Access-ports*. The leaking process is triggered thanks to knowledge coming from managers which synchronize between them, VS predicate and device location. In order to avoid forwarding loops during the leaking process, controllers apply a simple algorithm known as the *split horizon* algorithm. Within such an algorithm, a root switch never forwards to other root switches a packet it has received from another root switch. The *split horizon* algorithm does properly ensure reachability between domains given that we have a full mesh topology between root switches.

The work in this document is modeling the above device discovery procedure through MAC learning at all planes.

5.3.4 Packet Forwarding

A unicast packet with source MAC address 'S' and destination MAC address 'D' is forwarded if MAC addresses 'S' and 'D' are considered as part of the same *VS* (assuming the right mapping between MAC addresses and device names as discussed previously). Otherwise, the packet is dropped.

Concerning multicast (including broadcast) packets, the controller derives a multicast tree for each source MAC address 'S' [BDB16] based on the different *VSes* this MAC address 'S' belongs to. In order to avoid network loops when forwarding multicast packets over Inter-links, each domain controller implements the well-known *split horizon* algorithm [Rab+21] taking into account the full mesh nature of the *Inter-links* topology. The *split horizon* algorithm simply consists in not forwarding to other *Inter-links* a packet coming from an *Inter-link*.

Apart from multicast packet forwarding, all the data planes including unicast forwarding and flooding mechanism which is part of the MAC learning process are modelled during this work. As it concerns the control plane and the management plane, their essential behaviors are

completely captured by the models described in this document.

5.4 Modelisation of SDN

Here a transition $\delta = (q, \sigma, q') \in \Delta$ will also be written as $q \xrightarrow{\sigma} q'$ if δ is clear from the context. Moreover, for better readability, we will use $\sigma!$ if $\sigma \in \Sigma$ is a message sending actions, and $\sigma?$ if $\sigma \in \Sigma$ a message reading actions. This is only to help the reader ; formally the label does not contain the symbols ? and !.

Important Concepts An instance of the platform is defined by providing the topology for different planes. Recall that there is exactly one manager associated with each controller. Each controller only communicates with its own manager and the switches in its domain. Managers are interconnected together via a complete graph of communication.

Formally, consider a set of controllers $\text{Cont} = \{c_1, \ldots, c_k\}$, a set of managers $\text{Man} = \{\text{man}_1, \ldots, \text{man}_k\}$ with the same cardinal, and a set of switches $\text{Sw} = \text{Sw}_1 \cup \ldots \cup \text{Sw}_k$ given as a partition. Within the selected platform, the switches of Sw_i belong to the *domain i* which is controlled by the controller c_i. The latter is itself managed by man_i. We consider a set Dev of devices which can connect to switches of different domains. Since the management plane topology forms a complete graph, and that there is no link between controllers, these topologies are considered fixed in our modelling.

In each domain i, there is one designated switch called the *root switch*, denoted by $\text{root}_i \in \text{Sw}_i$. The data plane topology is a graph $G = (\text{Sw}, E)$ such that the subgraph restricted to each Sw_i is an undirected tree rooted at root_i, and the subgraph induced by the set of roots $\{\text{root}_1, \ldots, \text{root}_k\}$ is a complete graph.

In our studied platform, we distinguish so-called *Access-ports* through which devices connect to switches. We also have *Intra-ports* which are used to interconnect switches that belong to the same domain, and *Inter-ports* which are used to interconnect *root switches* of different domains. We also define a finite set Pts which refers to the set of *Access-ports* available at all switches.

Platform instance : an instance of the studied platform is defined as a tuple $(\text{Man}, \text{Cont}, \text{Sw}, G)$ where :

- $\text{Man} = \{\text{man}_1, \ldots, \text{man}_k\}$,
- $\text{Cont} = \{c_1, \ldots, c_k\}$,
- $\text{Sw} = \bigcup_{i=1}^{k} \text{Sw}_i$,
- $G = (\text{Sw}, E)$ is the topology graph as explained above.

We also use the function $\text{cont} : \text{Sw} \to \text{Cont}$ which identifies the controller associated to each switch, thus determining its domain : for all $i \in \{1, \ldots, k\}$ and $v \in \text{Sw}_i$, we have $\text{cont}(v) = c_i$.

Device Positions. We fix a finite set Dev of device identifiers *e.g.* based on their MAC addresses. Let $\text{Pos} = \text{Dev} \to (\text{Sw} \times \text{Pts}) \cup \{\perp\}$ denote the set of *device position functions*. Such a function assigns to each device the pair (sw, pt) of switch and port number to which it is connected, or $\perp$ if the device is not connected at any switch.

Virtual Spaces (VSes) : we consider the covering of Dev by the set of *Virtual Spaces* $\mathsf{VS} = \{V_1, V_2, .., V_f\}$, with $V_i \subseteq \text{Dev}$ for all $1 \leq i \leq f$, and $\cup_{i=1}^{l} V_i = \text{Dev}$. Let us define the function $\mathsf{VS}(x) = \{V \in \mathsf{VS} \mid x \in V\}$ which assigns to each device x the set of VSes to which x belongs.

In practice, the user can change VS dynamically by creating or deleting VSes for instance. Similarly, the user can dynamically change V(x) by modifying for instance predicates assigned to some VSes. The latter then include or exclude the device x. For simplicity, we assume in this work that VS and all V(x) for all $x \in$ Dev are static.

We also assume in this work that VS and all V(x) are known by all managers. As these former are static, they do not appear amongst states of modeled managers as per section 5.5.4.

Exchanged Data Packets : two types of packets are exchanged between switches and devices. *MAC learning packets* are sent by devices and forwarded by switches, and they contain the device identifier; while *ping packets* contain the source and target device identifiers.

These are defined, respectively, as $\text{MacPkts} = \{\mathsf{mac}\} \times \text{Dev}$ and $\text{PingPkts} = \{\mathsf{ping}\} \times \text{Dev} \times \text{Dev}$. These packets will be written as $\mathsf{mac}(x)$ and $\mathsf{ping}(x, y)$ respectively, for $x, y \in \text{Dev}$.

Exchanged Control Packets : let us define $\text{ManPkts} = \text{Dev} \times \{1, .., k\} \times 2^{\mathsf{VS}}$ a set of packets from the management plane to the controller plane and $\text{ContPkts} = \{(\text{man}_i, c_i) | i \in \{1, .., k\}\} \times \text{Dev}$ is the set of packets from the controller plane to the management plane.

It is noted that MacPkts forwarded by switches to the associated controllers are also part of exchanged control packets.

OpenFlow Rules : controller sends two types of rule updates to switches.

Rules impacting the forwarding of ping packets are defined as $\text{PingRules} = \text{Dev} \times \text{Dev} \times (\text{Pts} \cup \text{Sw})$ where the triple contains the source and target devices identifiers and the port or the switch to forward to.

Rules impacting the forwarding of MAC learning packets are $\text{MacRules} = \text{Dev} \times \text{Sw}$ where the pair contains the identifier of the device that has generated the mac learning packet, and the switch to forward to.

We let $\text{Rules} = \text{PingRules} \cup \text{MacRules}$.

Notice that we use switch identifiers as destinations instead of *Intra-port* identifier or *Inter-port* identifier to which both ping and MAC packets are forwarded to. This is because switch identifiers are used as port numbers for the communication between switches (see more detailed in section 5.5.2).

5.5 Generated Automata Models

We now describe the different automata that respectively model devices, switches, controllers, and managers. We provide descriptive information on the transitions in each model and interactions between *SDN*. layers

We denote the set of words in the alphabet A by $\mathrm{Seq}(A) = A^*$, while the empty word is ε. This notation will be used to encode packet queues in the automaton below. Note that we describe unbounded packet queues in our formal model, although the real platform uses bounded ones (see below).

5.5.1 Automaton for Devices

We define an automaton $\mathscr{A}^{\mathrm{Dev}}$ that describes the behaviors of the whole set of devices Dev. The automaton stores the positions (i.e. the *Access-port* which a device is connected to) of all devices, transmits MAC learning and ping packets, receives pings packets and allows devices to change positions. Moreover, the automaton also stores the set of ping packets that have been sent, and the set of ping packets that have been received by each device since the start. This information is used to check whether transmitted and received packets match the verification specifications. We restrict to the case where each device pings at most once any other device.

The device state space contains :

- a position function **pos** of type $\mathrm{Dev} \to (\mathrm{Pts} \times \mathrm{Sw}) \cup \{\bot\}$ which assigns each device a switch and an *Access-port* if it is connected, and $\bot$ otherwise ;
- a set **spings** $\subseteq$ PingPkts which stores send ping packets ;
- a set $\mathbf{rpings}_x \subseteq$ PingPkts each device x which stores received ping packets.

Initially, these spings and rpings_x for all $x \in$ Dev sets are empty, and pos maps all devices to $\bot$. Intuitively, a device x can send ping or MAC learning packets m to the switch v through port p (this has the form $(x, v, p, (\mathsf{mac}, x))$ or $(x, v, p, (\mathsf{ping}, x, y))$), and they can receive ping packets m from switch v via port p (this has the form $(v, p, (\mathsf{ping}, z, x))$). The latter does not contain the device identifier. In fact, this packet is forwarded by OVS v to port p, and any device that is connected to this port at that moment receives the packet.

Moreover, a device sends a new MAC learning packet whenever it changes its position.

The automaton $\mathscr{A}^{\mathrm{Dev}}$ has the following transitions.

► Changing position and sending a MAC learning packet : $q \xrightarrow{(x,v,p,\mathsf{mac}(x))!} q\,[\mathrm{pos} \leftarrow \mathrm{pos}[x \mapsto (v,p)]]$, for all $p \in \mathrm{Pts}, v \in \mathrm{Sw}, x \in \mathrm{Dev}$.

► Sending a ping packet :
$q \xrightarrow{(x,v,p,\mathsf{ping}(x,y))!} q\,[\mathrm{spings} \leftarrow \mathrm{spings} \cup \{\mathsf{ping}(x,y)\}]$,
for all $x \in \mathrm{Dev}$ and $\mathsf{ping}(x,y) \notin \mathrm{spings}$,

► Receiving a ping packet :
$q \xrightarrow{(v,p,\mathsf{ping}(x,y))?} q\left[\mathrm{rpings}_z \leftarrow \mathrm{rpings}_z \cup \{\mathsf{ping}(x,y)\}\right]$
for all $v \in \mathrm{Sw}, p \in \mathrm{Pts}$ and $\mathrm{pos}(z) = (v,p)$.

5.5.2 Automaton for Switches

Here we define an automaton $\mathscr{A}^{\mathrm{Sw}}$ that captures the behaviors of all the switches. Remember that we use switch identifiers as destinations instead of *Intra-port* identifier or *Inter-port* identifier to which both ping and MAC packets are forwarded to. This is because switch identifiers are used as port numbers for the communication between switches.

For simplicity of modeling and to reduce the state space, we abstract away from port numbers for communication between switches, and use switch identifiers instead. For instance, while flow tables are used to map packets to ports to forward to in the real system, our flow tables map packets to switch identifiers except when the outgoing port is an *Access-port*. This is without loss of generality since a switch identifier determines the port to which it is connected and vice versa (once the topology between switches is defined). Moreover, as mentioned in the beginning of this section, we omit outgoing packet queues, and assume that sending a packet consists in writing directly in the incoming packet queue of the recipient switch.

The automaton contains for each switch $v \in \mathrm{Sw}$ the following components :

- ping packet forwarding rules $\mathbf{pfwd}_v : \mathrm{Dev} \times \mathrm{Dev} \to \mathrm{Sw} \cup \mathrm{Pts} \cup \{\perp\}$ that is a partial function which, for a given ping packet $\mathsf{ping}(x,y)$ from device x to device y, assigns a switch or an *Access-port* to forward to ;
- MAC learning packet forwarding rules $\mathbf{mfwd}_v : \mathrm{Dev} \to 2^{\mathrm{Sw}}$ which gives the set of switches to which to forward the MAC learning packet received from given device ;
- a data packet queue $\mathbf{dque}_v$ of type $\mathrm{Seq}(\mathrm{MacPkts} \cup \mathrm{PingPkts})$ to store packets received from other switches ;
- and a control packet queue $\mathbf{cque}_v$ of type $\mathrm{Seq}(\mathrm{Rules})$ that stores rule packets coming from the controller.

The switches receive MAC learning or ping packets from devices (packets of the form (x,v,p,m)) which are put to dque_v. They forward MAC learning packets they have received to the controller (packets of the form
$((v,\mathrm{cont}(v)),(p,\mathsf{mac}(x))))$, and receive new rule updates from the controller (packets of the form $(\mathrm{cont}(v),v,r)$), which are put into cque_v.

Initially, pfwd_v is empty, and mfwd_v maps all devices to the set of neighboring switches of the same domain, that is, $\mathrm{mfwd}_v(x) = \{v' \in \mathrm{Sw}_i \mid \mathrm{cont}(v) = \mathrm{cont}(v') \wedge (v,v') \in E\}$ for all $x \in \mathrm{Dev}$. Moreover, all queues dque_v and cque_v are empty at the initial state.

When processing a MAC rule, the process pops a packet (x, p) from the queue cque_v, updates and triggers its mfwd_v function to forward MAC learning packets for device x to all neighboring v' switches. Similarly, when processing a ping rule (x, y, p), the flow table pfwd_v function is updated and triggered so that ping packets from x to y are forwarded to p.

The process forwards MAC learning packets $(p, \mathsf{mac}(x))$ to all neighbor switches v' (except the switch p where the packet comes from) in $\text{mfwd}_v(x)$, but it also forwards it to the controller, via the synchronizing transition (with label $((v, \text{cont}(v)), (p, \mathsf{mac}(x)))$.

When a *root switch* receives a MAC learning packet from another *root switch* (i.e. of another domain), the former only forwards the packet to its domain as per the previously described *split horizon* principle in order to avoid loops.

Forwarding ping packets are internal transitions. A packet $\mathsf{ping}(x, y)$ is forwarded by $\text{pfwd}_v(x, y)$ if this value is defined otherwise the packet is dropped. Transitions for $\mathscr{A}^{\text{Sw}}$ are defined as follows.

► Receiving a packet from the controller :
$q \xrightarrow{((\text{cont}(v),v),r)?} q\,[\text{cque}_v \leftarrow \text{cque}_v \cdot r]$, for all $r \in \text{Rules}$

► Receiving a MAC learning packet from a device :
$q \xrightarrow{(x,v,p,\mathsf{mac}(x))?} q\,[\text{dque}_v \leftarrow \text{dque}_v \cdot (p, \mathsf{mac}(x))])$,
for all $x \in \text{Dev}, p \in \text{Pts}$.

► Receiving a ping packet from a device :
$q \xrightarrow{(x,v,p,m)?} q\,[\text{dque}_v \leftarrow \text{dque}_v \cdot m])$,
for all $x \in \text{Dev}, p \in \text{Pts}, m \in \text{PingPkts}$,

► Processing a MAC rule : $q\,[\text{cque}_v = (x, v') \cdot \text{cque}_v] \xrightarrow{\tau}$
$q\,[\text{mfwd}_v \leftarrow \text{mfwd}_v[x \mapsto \text{mfwd}_v(x) \cup \{v'\}]]$,

► Processing a ping rule : $q\,[\text{cque}_v = (x, y, p) \cdot \text{cque}'_v] \xrightarrow{\tau}$
$q\,[\text{pfwd}_v \leftarrow \text{pfwd}_v[(x, y) \mapsto p], \text{cque}_v \leftarrow \text{cque}'_v]$,
for all $x, y \in \text{Dev}, p \in \text{Pts} \cup \text{Sw}$.

► Forwarding a MAC learning packet :
$q\,[\text{dque}_v = (p, \mathsf{mac}(x)) \cdot \text{dque}'_v] \xrightarrow{((v,\text{cont}(v)),(p,\mathsf{mac}(x)))!} q\big[\text{dque}_v \leftarrow \text{dque}'_v, \forall v' \in S',$
$\text{dque}_{v'} \leftarrow \text{dque}_{v'} \cdot (v, \mathsf{mac}(x))\big]$,
if either $p \in \text{Pts}$ or $p \in \text{Sw}$ and p is not a root switch with $S' = \text{mfwd}_v(x) \setminus \{v, p\}$ in this case; or $p = \text{root}_j$ for some j and $S' = \text{mfwd}_v(x) \cap \text{Sw}_i \setminus \{v\}$ where $v \in \text{Sw}_i$.
where $p \in \text{Pts} \cup \text{Sw}$; if p is not a root switch $root_j$, then $S' = \text{mfwd}_v(x) \setminus \{p\}$, else $S' = \text{mfwd}_v(x) \cap \text{Sw}_i$ with $i \neq j$. where $p \in \text{Pts} \cup \text{Sw}$, and $S' = \text{mfwd}_v(x) \cap \text{Sw}_i \setminus \{v\}$ such that $v \in \text{Sw}_i$ if $v \in \{\text{root}_1, \ldots, \text{root}_n\}$, and $S' = \text{mfwd}_v(x) \setminus \{v\}$ otherwise.

► Forwarding a ping (to OVSs) :
$q\,[\text{dque}_v = \mathsf{ping}(x, y) \cdot \text{dque}'_v] \xrightarrow{\tau}$
$q\,[\text{dque}_{v'} \leftarrow \text{dque}_{v'} \cdot \mathsf{ping}(x, y), \text{dque}_v \leftarrow \text{dque}'_v]$,

for all $v' = \mathrm{pfwd}_v(x,y) \in \mathrm{Sw}$,

► Forwarding a ping (to devices) :
$q\,[\mathrm{dque}_v = \mathsf{ping}(x,y) \cdot \mathrm{dque}'_v] \xrightarrow{(y,v,p,\mathsf{ping}(x,y)!)}$
$q\,[\mathrm{dque}_v \leftarrow \mathrm{dque}'_v]$, for all $p = \mathrm{pfwd}_v(x,y) \in \mathrm{Pts}$,

Note that in our model, packet communication between switches is modeled by directly writing in the packet queues of receiving switches.

The state space of the controllers also have Boolean variables toggle to indicate whether a ping update message has been sent to switches already.

5.5.3 Automaton for Controllers

We define an automaton $\mathscr{A}^{\mathrm{Cont}}$ that describes the behaviors of the entire control plane.

The state space contains for each controller c_i the following components :

- a *device relative position* function $\mathbf{rpos}_i$ of type $\mathrm{Dev} \times \mathrm{Sw}_j \rightharpoonup \mathrm{Pts} \cup \mathrm{Sw} \cup \{\perp\}$ which gives the position of a given device at the given switch as known to the controller c_i, that is, the port from which the MAC learning packet from the device is received and to which ping packets for the device are to be forwarded at that switch.
- a *device information function* $\mathbf{dinfo}_i$ of type $\mathrm{Dev} \rightharpoonup \{1,\ldots,k\} \times 2^{\mathsf{VS}}$ which stores the domain and Virtual Spaces to which each device belongs ;
- a queue $\mathbf{cdque}_i$ of type $\mathrm{Seq}(\mathrm{Sw}_i \times \mathrm{Pts} \times \mathrm{MacPkts})$ that stores packets that arrive from the data plane,
- and a queue $\mathbf{mque}_i$ of type $\mathrm{Seq}(\mathrm{ManPkts})$ that stores packets that arrive from the management plane.

Initially, partial functions rpos_i and dinfo_i, and both queues are empty.

Packets that come from the management plane are put in mque_i, while MAC learning packets from the data plane are put in cdque_i. When processing a MAC learning packet $(v,p,\mathsf{mac}(x))$, the controller updates its rpos_i function to store the relative position p of the device x at a given switch v. When processing a packet (x,dom,V) from the management plane, the controller updates its dinfo_i function to store the fact that device x belongs to domain dom and VS V. Furthermore, the controller sends ping rule updates of the form (x,y,p) to the switches whenever devices x and y belong to a common VS, and $\mathrm{rpos}_i(y,v) = p$. This ensures that ping packets with destination y be forwarded to p at switch v. Finally, the controller also sends MAC learning rule updates to *root switches*. It sends the rule (x,root_j) to switch root_i whenever device x is known

to belong to domain i and there exists some device y in domain j which shares a common VS with x. This ensures that MAC learning packets will be forwarded from domain i to domain j.

The transitions of $\mathscr{A}^{\text{Cont}}$ are defined as follows.

► Receiving a packet from the management plane :
$q \xrightarrow{(\text{man}_i, c_i, (x, dom, V))?} q\,[\text{mque}_i \leftarrow \text{mque}_i \cdot (x, dom, V)]$
for all $x \in \text{Dev}, 1 \leq dom \leq k, V \subseteq \mathsf{VS}$.

► Receiving a MAC packet from the data plane :
$q \xrightarrow{((v,c_i),(p,\mathsf{mac}(x)))?} q\,[\text{cdque}_i \leftarrow \text{cdque}_i \cdot (v, p, \mathsf{mac}(x))]$,
for all $x \in \text{Dev}, v \in \text{Sw}_i$, $p \in \text{Pts} \cup \text{Sw}$.

► Processing a Mac learning packet (from IOT port) :
$q\,[\text{cdque}_i = (v, p, \mathsf{mac}(x)) \cdot \text{cdque}'_i] \xrightarrow{(c_i, \text{man}_i, x)!}$
$q\,[\text{rpos}_i \leftarrow \text{rpos}_i[(x, v) \mapsto p],\ \text{cdque}_i \leftarrow \text{cdque}'_i]$,
toggle = false, for all $v \in \text{Sw}_i, p \in \text{Pts}$.

► Processing a Mac learning packet (from Non-IOT port) :
$q\,[\text{cdque}_i = (v, p, \mathsf{mac}(x)) \cdot \text{cdque}'_i] \xrightarrow{\tau}$
$q\,[\text{rpos}_i \leftarrow \text{rpos}_i[(x, v) \mapsto p],\ \text{cdque}_i \leftarrow \text{cdque}'_i]$,
toggle = false, for all $v \in \text{Sw}_i, p \in \text{Sw}$.

► Processing a packet from man_i :
$q\,[\text{mque}_i = (x, \mathsf{dom}, V) \cdot \text{mque}'_i] \xrightarrow{\tau}$
$q\,[\text{dinfo}_i \leftarrow \text{dinfo}_i[x \mapsto (\mathsf{dom}, V)], \text{mque}_i \leftarrow \text{mque}'_i]$,
toggle = false, for all $x \in \text{Dev}$, $1 \leq \mathsf{dom} \leq k$ and $V \subseteq \mathsf{VS}$.

► Sending ping rule updates to switches :
$q\,[\text{toggle} == \text{false}] \xrightarrow{(c_i, v, (x,y,p))!} q\,[\text{toggle} = \text{true}]$
for all $x, y \in \text{Dev}, v \in \text{Sw}_i$, $\text{rpos}_i(y, v) = p \neq \perp$, and writing $\text{dinfo}_i(x) = (j_x, V)$, $\text{dinfo}_i(y) = (j_y, V')$ and $V \cap V' \neq \emptyset$.

► Sending MAC rule update to root_i :
$q \xrightarrow{(c_i, \text{root}_i, (x, \text{root}_j))!} q$ for all $x \in \text{Dev}$ such that $\text{rpos}_i(x, \text{root}_i) \neq \perp \wedge \text{dinfo}_i(x) = (i, V)$, and there exists $y \in \text{Dev}$ s.t. $\text{dinfo}_i(y) = (j, V')$ with $j \neq i$ and $V \cap V' \neq \emptyset$.

5.5.4 Automaton for Managers

We define automaton $\mathscr{A}^{\text{man}}$ that describes the behaviors of the entire management plane. The state space contains for each management node man_i the following components :

- *a mapping function* $\textbf{dinfo}_i : \text{Dev} \to \{1, \ldots, k\} \cup \{\perp\}$ determining the domain in which the device is connected.
- a packet queue $\textbf{cque}_i$ that stores packets that come from c_i ;

- and a queue $\mathbf{mque}_i$ for packets from other management nodes.

Initially, both queues are empty and dinfo_i maps all devices to $\perp$.

The management node man_i learns about the domains to which devices belong through packets received from the controller c_i, and forwards this information to all other management nodes. More precisely, packets sent by the controllers are put in the queues cque_i. When processing a controller packet for device $x \in \text{Dev}$, the manager man_i sends the packet (x,i) to all other managers informing them that device x belongs to domain i. This helps all management nodes to eventually have full information on the domain where each device is located. Moreover, the management node man_i sends to the controller c_i the VSes to which each known device belongs by sending packets of the form (x, dom, V) where x is a device, dom its domain, and V the set of VSes to which the device x belongs. The transitions of $\mathscr{A}^{\text{man}}$ are defined as follows.

► Receiving controller packet :
$q \xrightarrow{(c_i,\text{man}_i,x)?} q\,[\text{cque}_i \leftarrow \text{cque}_i \cdot x]$. where $x \in \text{Dev}$

► Processing a controller packet :
$q\,[\text{cque}_i = x \cdot \text{cque}_i'] \xrightarrow{\tau}$
$q\big[\forall j \neq i, \text{mque}_j \leftarrow \text{mque}_j \cdot (i,x),$
$\qquad \text{dinfo}_i \leftarrow \text{dinfo}_i[x \mapsto i], \text{cque}_i \leftarrow \text{cque}_i'\big]$.
where $x \in \text{Dev}$ and $1 \leq j \leq k$.

► Processing a management packet :
$q\,[\text{mque}_i = (j,x) \cdot \text{mque}_i'] \xrightarrow{\tau}$
$q\,[\text{dinfo}_i \leftarrow \text{dinfo}_i[x \mapsto j], \text{mque}_i \leftarrow \text{mque}_i']$.

► Sending device information to controller :
$q \xrightarrow{(\text{man}_i,c_i,(x,j,V))!} q$ for all $x \in \text{Dev}, 1 \leq j \leq k$ such that $\text{dinfo}(x) = j$ and $V = \mathsf{VS}(x)$.

5.5.5 SDN specification

We verify two important properties of our platform. The first is a *safety* property, called the *isolation* property. It states that only devices that belong to a common VS can exchange ping packets and a device not belonging to a particular VS cannot eavesdrop ping packets exchanged among that VS group. This can be expressed in *LTL* as follows.

$$\mathsf{Isolation} = \Box\Big(\begin{array}{l} \bigwedge_{\substack{x,y\in\text{Dev},\\ \mathsf{VS}(x)\cap\mathsf{VS}(y)=\emptyset}} \mathsf{ping}(x,y) \notin \text{rpings}_y \\ \bigwedge_{\substack{x,y,z\in\text{Dev},\\ z\neq y}} \mathsf{ping}(x,y) \notin \text{rpings}_z \end{array} \Big).$$

This *LTL* formula thus describes the set of executions which never enter a state where a packet $\mathsf{ping}(x,y)$ with $\mathsf{VS}(x) \cap \mathsf{VS}(y) = \emptyset$ is received, and no device can eavesdrop on the packet which has to be received by different devices.

The second property is called the *connectivity* property. It states that whenever a ping packet is sent to a device that belongs to the same VS group, the packet is eventually received ; however this property only holds after some point in time, that is, when the MAC learning algorithm has finished. So the *SDN* platform eventually allows a common VS group to exchange the data packets among themselves. This can be expressed in *LTL* as follows :

$$\mathsf{Connect} = \bigwedge_{\substack{x,y\in \mathrm{Dev} \\ \mathsf{VS}(x)\cap\mathsf{VS}(y)\neq\emptyset}} \begin{array}{l} \Diamond\big(\mathsf{ping}(x,y) \in \mathrm{spings} \wedge \\ \quad \Diamond \mathsf{ping}(x,y) \in \mathrm{rpings}_y\big). \end{array}$$

In our work we consider two scenarios. In the *no-mobility* scenario, devices do not change their respective position once they are connected to an *Access-port*. In the *mobility* scenario, devices can change their position at any time.
The first scenario called as Nomobility can be expressed in *LTL* as follows :

$$\mathsf{Nomobility} = \bigwedge_{\substack{x\in \mathrm{Dev}, v\in \mathrm{Sw}, \\ p\in \mathrm{Sw}\cup \mathrm{Pts}}} \begin{array}{l} \Box\big((v,p) = \mathrm{pos}(x) \rightarrow \\ \quad \Box(v,p) = \mathrm{pos}(x)\big) \end{array}$$

And to use Theorem 4.3 we need to formally state the specifications for each component in $\mathrm{LTL}\setminus X$. Finding these properties requires domain knowledge. In fact, one needs to understand with which intentions the system was designed in order to write these formulas.
The management plane and control plane should work together to satisfy the following requirement :

$$\mathsf{I}_1 = \bigwedge_{\substack{1\leq i\neq j\leq k, x,x'\in \mathrm{Dev} \\ \mathsf{VS}(x)\cap\mathsf{VS}(x')\neq\emptyset \\ v\in \mathrm{Sw}_i, v'\in \mathrm{Sw}_j \\ p,p'\in \mathrm{Pts}}} \begin{array}{l} \Box\big(\mathrm{pos}(x) = (v,p) \wedge \mathrm{pos}(x') = (v',p') \\ \quad \wedge(\mathrm{rpos}_i(x,\mathrm{root}_i) \neq \bot) \rightarrow \\ \quad \Diamond \mathsf{Sync}(c_i, \mathrm{root}_i, (x, \mathrm{root}_j))\big). \end{array}$$

The above formula says that whenever devices x and x' are respectively connected at switch-port pairs (v,p) and (v',p') in respectively domains i and j, if the root switch of domain i has received a MAC learning packet originated from device x, then eventually the root switch will receive a MAC update rule from the controller to forward MAC packets of device x towards root_j. This I_1 captures the fact that the control plane sends appropriate MAC rule updates to the root switch of its domain, thus to the data plane.

$$\mathsf{I}_2 = \bigwedge_{\substack{1\leq i\leq k, x,x'\in \mathrm{Dev} \\ \mathsf{VS}(x)\cap\mathsf{VS}(x')\neq\emptyset \\ v,v'\in \mathrm{Sw}, p,p'\in \mathrm{Pts} \\ u\in \mathrm{Sw}_i, q,q'\in \mathrm{Pts}\cup \mathrm{Sw}}} \begin{array}{l} \Box\big(\mathrm{pos}(x) = (v,p) \wedge \mathrm{pos}(x') = (v',p') \\ \quad \wedge(\mathrm{rpos}_i(u,x) = q) \\ \quad \wedge(\mathrm{rpos}_i(u,x') = q') \rightarrow \\ \quad \Diamond \mathsf{Sync}(c_i, u, (x,x',q'))\big). \end{array}$$

The above formula states that, given devices x, x' connected at (v, p) and (v', p') respectively, if another switch u has received and forwarded MAC learning packets to the controller witnessed by $(\text{rpos}_i(u,x) = q) \wedge (\text{rpos}_i(u,x') = q')$, then a ping update rule will eventually be sent to switch u to forward packets $\mathsf{ping}(x,x')$ to q'. Thus, I_2 expresses that the control plane sends appropriate ping rule updates to all the switches and thus to the data plane of its domain.
The above requirements should be satisfied jointly by the management and control planes, that is, we would like to establish that $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \parallel \mathscr{A}^{\text{Cont}} \langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle$ should be true. However, this check can still be costly for large topologies. Therefore, we apply again Theorem 4.3 to check this compositionally.
We need to introduce another intermediate formula I_3 between the management plane and controller plane. This captures that whenever a management node man_i receives information about a device's position from its controller c_i, eventually, all management nodes man_j forward this information to their respective controller c_j. Thus, the information about a device of domain i is eventually shared with the controller of other domains j.

$$\mathsf{I}_3 = \bigwedge_{\substack{1 \leq i \leq k, x \in \text{Dev} \\ j \in [1,..,k]}} \begin{array}{l} \Box\big((\text{dinfo}_i(x) \neq \bot) \rightarrow \\ \qquad \Diamond \mathsf{Sync}(\text{man}_j, c_j, (x, \text{dinfo}_i(x), \mathsf{VS}(x)))\big) \end{array}$$

So we can check the satisfaction of $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \parallel \mathscr{A}^{\text{Cont}} \langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle$ by verifying $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \langle \mathsf{I}_3 \rangle$, and $\langle \mathsf{I}_3 \rangle \mathscr{A}^{\text{Cont}} \langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle$.

5.5.6 Experimental Results

No-Mobility Scenario Validation

Let us call the Isolation, Connect, and Nomobility as ϕ_I, ϕ_C and ψ respectively. Recall the intermediate formula I_1, I_2, and I_3 We are going to verify, the specified *SDN* platform satisfies the Isolation and Connect properties under the assumption of Nomobility property.
We are going to compare the usage of Compositional Reasoning (CR) in verifying the SDN platform over the monolithic approach. Monolithic approach consists in verifying the entire system i.e checking whether $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \parallel \mathscr{A}^{\text{Cont}} \parallel \mathscr{A}^{\text{Sw}} \parallel \mathscr{A}^{\text{Dev}} \langle \psi \rightarrow \phi_I \wedge \phi_C \rangle$ is true or false. Compositional rule method 1 (CR method 1) consists in splitting the control plane from the data plane and thus in verifying separately the two automata : $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \parallel \mathscr{A}^{\text{Cont}} \langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle$, and $\langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle \mathscr{A}^{\text{Sw}} \parallel \mathscr{A}^{\text{Dev}} \langle \psi \rightarrow \phi_I \wedge \phi_C \rangle$. Compositional rule method 2 (CR method 2) consists in further splitting the management plane from the control plane and checking three triples : $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \langle \mathsf{I}_3 \rangle$, $\langle \mathsf{I}_3 \rangle \mathscr{A}^{\text{Cont}} \langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle$, and $\langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle \mathscr{A}^{\text{Sw}} \parallel \mathscr{A}^{\text{Dev}} \langle \psi \rightarrow \phi_I \wedge \phi_C \rangle$.
The results for various sizes of the *SDN* platform topology and the three methods are shown in Table 5.1. For a given size of *SDN* platform, we checked all possible *SDN* constrained topologies as mentioned earlier. and for all possible random choices carried by the devices to

connect to their *Access-port* positions. Within the Nomobility condition, once the device has selected its position and sent its first MAC learning packet, it waits for a timeout. After the first timeout, it sends a second MAC learning packet and waits for the second timeout. After the second timeout, it sends ping packets to all other devices. In this scenario we check that devices sharing a common VS can receive ping packets from each other (i.e. Connect property), and that devices not belonging to this particular VS cannot receive the ping packets from the group and vice versa (i.e. Isolation property). In order to test these Isolation and Connect properties we create the following set of VSes, $\mathsf{VS} = \{\{d_1, d_2\}, \{d_3\}\}$, where $\{d_1, d_2\}$ is one VS containing devices d_1 and d_2 and $\{d_3\}$ the other VS containing a single device d_3. We verify that d_3 cannot receive ping packets from devices in $\{d_1, d_2\}$ and vice-versa in order to assert that the SDN platform satisfies the Isolation property. We also verify that eventually d_1 and d_2 can receive ping packets from one another to assert that the SDN platform satisfies the Connect property. For the particular case of no-mobility, the SDN platform satisfies both Isolation and Connect properties. To shortly capture the SDN platform architecture we adopt for the rest of the document the following notations. We denote by $n, (X, Y, Z)$ the set of topologies with n domains ($1 \leq n \leq 3$ in our case), where (X,Y,Z) refers to the number of OVSes in each domain. For the single domain with 3, 4, and 5 switches there are respectively two, four and five possible data plane topologies since we restrict to trees. For the case of two domains ($n = 2$), the possibilities are $(2,1,-)$, $(3,1,-)$, $(2,2,-)$, and $(3,2,-)$; in this case, there are, respectively, one, two, one, and two possible data plane topologies. With three domains ($n = 3$), the possibilities are $(1,1,2)$ and $(1,1,3)$, in which case there are one and two possible data plane topologies respectively. In our experiments, we enumerate all topologies that match a given template (X, Y, Z) to check exhaustively all possible scenarios. Therefore, the cases with larger numbers of topologies take more time to check.

We ran our experiments on an Intel i7 processor with 13GB of available RAM. Table 5.1 compares the monolithic and two compositional approaches, called Method 1 and Method 2, with respect to CPU time and memory usage on SDN topologies of varying sizes.
While the monolithic approach fails to scale above trivial topologies, compositional methods helped us to complete the verification process up to 3 domains in our experiments. However, we still fail to complete the verification process for one domain with 6 switches with both CR methods. The bottleneck during this experiment was checking $\langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle \mathscr{A}^{\mathrm{Sw}} \parallel \mathscr{A}^{\mathrm{Dev}} \langle \phi_I \wedge \phi_C \rangle$.
CR method 1 and method 2 provide the same performance up to 2 domains. When the number of domain number 3, CR method 2 gives advantage over the RAM usage and CPU time. Within CR method 1, for 3 domains, much of its CPU and RAM resources are consumed in checking $\langle \mathrm{true} \rangle \mathscr{A}^{\mathrm{man}} \parallel \mathscr{A}^{\mathrm{Cont}} \langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle$. Thanks to the additional decomposition between the management plane and the control plane, CR method 2 provides us with better performance. Here the majority of CPU and RAM resources are consumed in checking $\langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle \mathscr{A}^{\mathrm{Sw}} \parallel \mathscr{A}^{\mathrm{Dev}} \langle \psi \rightarrow \phi_I \wedge \phi_C \rangle$.
The sizes of state space produced in the monolithic method reaches the order of 10^6 for two

switches in a single domain, while the compositional approaches produce around 10^4 states for the same architecture. However, the compositional approaches still can not manage the model with 6 switches in a single domain and went out of memory.

n domains,	Monolithic Method		CR Method1		CR Method2	
(X,Y,Z) OVSes	CPU time	RAM used	CPU time	RAM used	CPU time	RAM used
$1,(1,-,-)$	0.38s	133*MB*	0.85s	134*MB*	1.11s	135*MB*
$1,(2,-,-)$	166s	723*MB*	1.05s	135*MB*	1.32s	135*MB*
$1,(3,-,-)$	-	-	6.8s	168*MB*	7s	168*MB*
$1,(4,-,-)$	-	-	2m 34s	532*MB*	2m 32.4s	531*MB*
$1,(5,-,-)$	-	-	39m 33s	5002*MB*	39m 40s	5002*MB*
$2,(1,1,-)$	-	-	4.07s	145*MB*	1.66s	135*MB*
$2,(2,1,-)$	-	-	9.5s	164*MB*	4.05s	164*MB*
$2,(3,1,-)$	-	-	1m 29s	505*MB*	1m 10.69s	505*MB*
$2,(2,2,-)$	-	-	47.73s	502*MB*	35.77s	502*MB*
$2,(3,2,-)$	-	-	15m 21s	4692*MB*	14m 31.3s	4692*MB*
$3,(1,1,1)$	-	-	18m 25s	4292*MB*	4.22s	162*MB*
$3,(1,1,2)$	-	-	36m 11s	6982*MB*	34.62s	472*MB*
$3,(1,1,3)$	-	-	1h 14m 40s	12254*MB*	13m 50s	4482*MB*

TABLE 5.1 : Monolithic vs Compositional Approaches (with no-mobility)

Issue with Device Mobility

We now present an interesting part of our formal verification experiments. In the last section, we found that the studied *SDN* platform does satisfy the required specifications, namely the *isolation* property and the *connectivity* property. What if we allow devices to move freely but still use the data plane service to exchange packets ? Given this freedom, devices can select their new positions wherever and whenever they want.

This freedom makes the *SDN* system vulnerable w.r.t. the data isolation property. We identified an error, that is, an execution which violates the isolation property. We can produce the error simply with our monolithic model checking for one domain and two switches, that is, $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \parallel \mathscr{A}^{\text{Cont}} \parallel \mathscr{A}^{\text{Sw}} \parallel \mathscr{A}^{\text{Dev}} \langle \phi_I \rangle$ is violated, where ϕ_I the Isolation property.

We find this error by considering $\mathsf{VS} = \{\{d_1, d_2\}, \{d_3\}\}$ and the following mobility scenario : once first MAC learning packets are exchanged and ping rules are updated, devices d_2 and d_3 swap their respective *Access-port*. After the mobility events, Spin raises an Isolation violation reporting that device d_3 receives a ping packet sent by d_1 to d_2.

Network Slice Isolation Violation

SPIN reveals a counterexample trace but does not give the probability of its occurrence. In this section, we discuss an illustrative scenario to derive an estimation of the probability.

Let's consider δ as the minimum time needed by the controller to receive a given data plane packet and to respond with a new set of rules to the originating switch (due to the controller queues cdque and mque). Further, let's call ε as the time needed by the switch to inform the controller about the possible reconfiguration of the data plane network topology (due to switch queues cque and dque). Finally, let's consider λ as the average number of data packets exchanged per time unit by the switch and devices in the data plane.
With the SDN platform made of a single domain $k = 1$, we have the controller c in the control plane, and three devices $\{d_1, d_2, d_3\}$ and one switch sw which has three ports $\{1,2,3\}$ in the data plane. Initially devices d_1, d_2, d_3 are connected respectively to the ports $1,2,3$ of switch sw. The management plane applies the following VS of devices $\{\{d_1, d_2\}, \{d_3\}\}$ where $\{d_1, d_2\}$ forms one VS and $\{d_3\}$ is lonely device in another VS. The controller has to ensure that the set of rules it forwards to the data plane (i.e to the switch sw) prevent devices d_3 from receiving packets exchanged between d_1 and d_2. After receiving MAC learning packets from the switch sw, the controller c forwards ping rules to the latter, enforcing that packets from the port 1 should be delivered to port 2 and those from port 2 should be delivered to port 1 and any data packet from the port 3 should be dropped.
When the data plane topology undergoes a transition, such as d_2 and d_3 are exchanging their respective position, it takes the controller c a time interval of $(\delta + \varepsilon)$ before being aware of the new situation and actually enforcing new rules. During this lap of time, there are possible scenarios of device d_3 being able to receive packets exchanged between devices d_1 and d_2. For instance, device d_3 occupies now the port 2 of the switch sw and devices d_1, d_2 occupy respectively ports $1,3$. Due to the communication delay between the switch and the controller, the chance (probability) of device d_3 receiving ping packets delivered by d_1 to d_2 is in the order of $\mathcal{O}(\lambda \times (\delta + \varepsilon))$. We are able to reproduce such an Isolation violation with the network simulator called GNS3.
Thus, the probability of the Isolation violation occurrence is roughly proportional to the packet data rate (i.e. λ) and the overall latency (i.e. $(\delta + \varepsilon)$) between the domain controller and the switch, called as the *control link latency*. The first proportionality means that higher bandwidth data paths require a faster control plane or smaller *control link latency*. However, the latter has its own limitations and is very dependent on the SDN architecture. On the studied SDN platform, the controller is centralized so that there is a one-to-one mapping between the controller and the manager in a given domain. This allows for simplifying the synchronization of user's intents and device identity between the two components. Unfortunately, this centralization could induce important *control link latency*. First, it's about communication delay. As uplink communications from different switches within the domain are aggregated when they arrive at the controller, the bandwidth required becomes higher and higher as the number of controlled switches and the number of connected devices increase. Congestion at the controller incoming link could occur if the link was not correctly dimensioned - e.g. a massive arrival of

new devices into the data network leads to a massive arrival of MAC-Learning-related PacketIn messages at the controller. Second, it's about processing delays. If the controller was limited in terms of processing power, then its incoming queues could fill up leading to excessive delay and even to loss of control messages (i.e. infinite delay). Furthermore, this also illustrates the fact that the order in which the reading of messages is important for safety properties for SDN platforms.

Local Controller Solution

In order to work around previous *control link latency* issues, one possible solution consists in implementing a hierarchical controller architecture with a new controller embedded together with each switch that we call as *local controller* in addition to the existing centralized controller that we call as the *central controller*. Figure 5.4 illustrates such a controller hierarchical architecture. The *local controller* role simply consists in blocking any outgoing traffic towards a port (e.g. port 2) when the existing device (e.g. d_2) is disconnecting from this port (e.g. a port down status message sent to controllers). It maintains such a blocking rule until a new device (e.g. d_3) is connected to the port and MAC learning messages from this device are processed by the *central controller* and new forwarding rules related to the new device are received by the switch. In order to ensure new rules are actually received by the switch, the *local controller* is implemented as an OpenFlow proxy between the switch and the *central controller*.

As the *local controller* is colocated with the switch, we can assume a zero *control link latency* between the two.

Moreover, the number of devices the *local controller* should deal with is smaller than the one the *central controller* is dealing with by the order of magnitude of the number of switches. The variability of control message rate is to be smaller from the *local controller* perspective, which leads to easier engineering in terms of processing resources. Such a local controller can be modeled by an automata as $\mathscr{A}^{\mathsf{loc.Cont}}$.

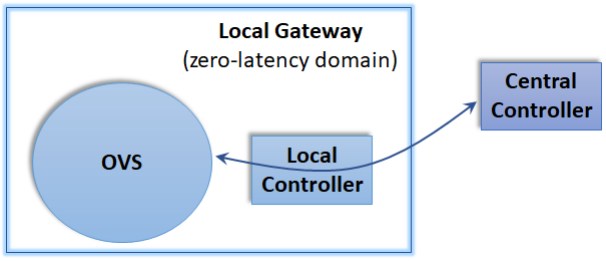


FIGURE 5.4 : Local Controller Architecture

Local Controller Solution Validation

In this section, we verify that the proposed local controller solution satisfies the Isolation property, i.e $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \parallel \mathscr{A}^{\text{Cont}} \parallel \mathscr{A}^{\text{Sw}} \parallel \mathscr{A}^{\text{loc.Cont}} \parallel \mathscr{A}^{\text{Dev}} \langle \phi_I \rangle$, where ϕ_I is the Isolation property

defined in Section 5.5.5 and $\mathscr{A}^{\mathsf{loc.Cont}}$ proposed local controller method specified previously. By using the Theorem 4.3, we check $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \parallel \mathscr{A}^{\text{Cont}} \langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle$ and $\langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle \mathscr{A}^{\text{Sw}} \parallel \mathscr{A}^{\mathsf{loc.Cont}} \parallel \mathscr{A}^{\text{Dev}} \langle \phi_I \rangle$ to prove $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \parallel \mathscr{A}^{\text{Cont}} \parallel \mathscr{A}^{\text{Sw}} \parallel \mathscr{A}^{\mathsf{loc.Cont}} \parallel \mathscr{A}^{\text{Dev}} \langle \phi_I \rangle$, where I_1 and I_2 are the intermediate formula defined in Section 5.5.5. The proposed local controller does satisfy the required isolation property. We carry out the experiment for different topologies of the SDN platform. For each topology, we monitor the computational time and the memory usage. Table 5.2 provides a synthetic view of these experimental data. We made full *SDN* SPIN model at a public-access repository[1].

n domains, (X,Y,Z) Switches	Compositional reasoning rule $\langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle \mathscr{A}^{\text{Sw}} \parallel \mathscr{A}^{\mathsf{loc.Cont}} \parallel \mathscr{A}^{\text{Dev}} \langle \phi_I \rangle$ $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \parallel \mathscr{A}^{\text{Cont}} \langle \mathsf{I}_1 \rangle$ and $\langle \text{true} \rangle \mathscr{A}^{\text{man}} \parallel \mathscr{A}^{\text{Cont}} \langle \mathsf{I}_2 \rangle$	
	Total Computation time	Maximum RAM memory used
$1, (1,-,-)$	4.8sec	188 *MB*
$1, (2,-,-)$	8.4sec	224 *MB*
$1, (3,-,-)$	1min 2sec	448 *MB*
$1, (4,-,-)$	6min 22sec	982 *MB*
$1, (5,-,-)$	1hr 5min 8sec	7245 *MB*
$2, (1,1,-)$	11sec	214 *MB*
$2, (2,1,-)$	33sec	395 *MB*
$2, (3,1,-)$	3min 10.3sec	865 *MB*
$2, (2,2,-)$	96.5sec	848 *MB*
$2, (3,2,-)$	25min 13sec	6327 *MB*
$3, (1,1,1)$	19min 9.8sec	4291 *MB*
$3, (1,1,2)$	36min 44.39sec	6981 *MB*
$3, (1,1,3)$	1hr 24min 3.45sec	12254 *MB*

TABLE 5.2 : Local Controller Proposal Validation using CR method (mobility scenario)

5.6 Chapter Conclusion

In our modeling case, we explicitly take asynchronous aspect while modeling the *SDN* system and verified the basic requirements (specifications) of network *Isolation* and *Connectivity* property compare to most of the other *SDN* verification scheme in the literature as mentioned in the brief literature survey. Although this basic property with the abstracted model, it is not difficult to come up with proof arguments why those properties are satisfied in the non-mobility case and why it is not satisfying the safety property i.e *Isolation property* in the mobility case. Our aim is to do automated verification, but this verification in a monolithic way took too much time and memory.It is well known that the Model checking process will take huge time and memory [Cla08] to finish the verification process. But when we started this verification process,

1. `https://gitlab.inria.fr/anoordhe/drcn2021milan`

this huge time and space consumption is not what the *ADR sapiens project team* expected for this small sized network. For this huge state space, there are couple of reasons, first of all, the modeled system is asynchronous, this spin model checker itself a classic verification tools, we don't know how well it is optimized in terms of encoding the state and transition, may be recent model checking tools have better encoding schemes, this we really don't know, I'm just speculating it. Because of this reason we spent a considerable amount of time in coming up with a state space reduction technique and found the compositional reasoning a natural fit for well separated and layered *SDN* architecture. With the help of the compositional reasoning we managed to verify the network specifications *Isolation* and *Connectivity* for the increased *SDN* network size. Maybe a person well trained Model checking (experts) knows the details of the model checking process, knows sophisticated algorithms and details in depth can give even better experimental results.

CHAPITRE 6

DISCRETE CONTROL SYNTHESIS FOR AN *SDN-IoT* PLATFORM

Discrete Control synthesis has not been yet utilized in *SDN* platform because of the complexity of control synthesis of various specifications which are at least the complexity of model checking (refer the relation between synthesis and model checking in chapter 3). Hence, discrete control synthesis for practical purposes is only used for high level model and specification rather than detailed model or description of distributed systems. However, several theoretical works related to synthesis have been done to control/supervise a *SDN* platform. Synthesis supervision for a *SDN* platform tackles the synthesis problem of forwarding data packets, updating network instances and maintaining the network update procedure based on the condition imposed by the *SDN* network manager which enforces requirements from end-users. The research works of papers [FSV19 ; McC+15a ; CWL18a], etc discuss synthesis of various SDN network specifications. Consistent network updates [FSV19 ; McC+15a], order of network events update with predefined specifications. Automated Input-output *SDN* rules refinement [CWL18a] synthesis scheme to make *SDN* to generate the safe output for the given input at various times based on the specification setup. The work [Cvi+13] is towards healthy bandwidth to minimize the latency of heterogeneous *SDN* scenarios. More details of the relevant synthesis of *SDN* works from the literature are discussed in the following section.

- [JRG21] tackle the synthesis of routing data packets, updating the network instances and maintain the network update procedure based on the condition imposed by *SDN* network manager and requirements from network end-users,
- Consistent network updates [FSV19] and order of network events update [McC+15a] with predefined specifications,
- Automated Input-output *SDN* rules refinement [CWL18a],
- Model synthesis to make the *SDN* work towards healthy bandwidth to minimize the latency [Cvi+13].
- In [Wan+13], reactive synthesis is used to automatically synthesize network update rules in response to network requests. Other works consider the synthesis of update rules

that makes sure that the intermediate configurations during the update are all admissible [Rei+12 ; McC+15b], or compute correct network-wide configurations [El-+17].

6.1 Existing Synthesis of Network Services

6.1.1 Synthesis of Consistence updates

Consistent network updates [FSV19 ; McC+15a] discuss the synthesis of consistency updates of the *SDN* controller commands on the data plane switches such that the specification on the data plane remains valid in the update phase. The specification involves *connectivity* (or existence of path without disconnection in the update phase) between any two valid secure hosts.

EXAMPLE 6.1 (Problem Statement) *When a manager decides to change the path between two switches (or hosts) due to the traffic or some other reason, during the update phase of changing the routing between two switches, how to make the connectivity (there should be a data path) between these end switches (or hosts) in the figure 6.1 between hosts H_1, H_3.*

For instance, in the figure 6.1 we have three highlighted paths

Red path

$$T_1 \longrightarrow A_1 \longrightarrow C_1 \longrightarrow A_3 \longrightarrow T_3$$

Green path

$$T_1 \longrightarrow A_1 \longrightarrow C_2 \longrightarrow A_3 \longrightarrow T_3$$

Blue path

$$T_1 \longrightarrow A_2 \longrightarrow C_1 \longrightarrow A_4 \longrightarrow T_3$$

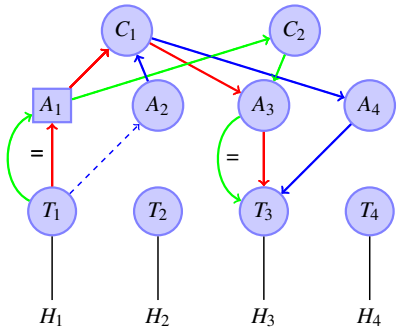


FIGURE 6.1 : Consistent Update Synthesis

In order to change the path from red to green, the controller should update the forwarding table of the switches $\{A_1, C_2\}$ in the given network (figure 6.1). For changing the path from red to blue, the controller should update the forwarding table of switches $\{T_1, A_2, C_1, A_4\}$. For changing the red to green path if we update the switches in the following order C_2, A_1, neither

connectivity is lost (i.e in between the updating we have connection form H_1 to H_2) or consistency is lost (i.e the connection between H_1 and H_3 is either red path or green path).
Suppose if we want to change the data connectivity path for H_1, H_3 from red to blue path, we have to update the switches $\{T_1, A_2, C_1, A_4\}$, now the question arises in which order we should update the switches ? At first A_2 and A_4 forwarding tables can be updated any order it won't introduce any new path or break the existed red path and followed by T_1 and C_1 in what order ?
***Case 1** : Suppose T_1 updated and then C_1 is updated, there is new path which is neither a red or blue in between these updates which is*

$$T_1 \longrightarrow A_2 \longrightarrow C_1 \longrightarrow A_3 \longrightarrow T_3$$

of course connectivity holds between H_1 and H_3 in this transition but it is not consistent in the update phase (here consistent refers to the strict of following single coloured edges rather then mixture of coloured edges).
***Case 2** : Suppose C_1 is updated first before T_1, here also there is new path which is neither a complete red or blue path*

$$T_1 \longrightarrow A_1 \longrightarrow C_1 \longrightarrow A_4 \longrightarrow T_3$$

here too connectivity holds but consistency is not maintained.

From this example, one can understand that guaranteeing such strict consistency by finding the right order of updating the rules to the switch forwarding tables is not always possible. If the property is little weaker than this strict consistency property one may find ordering updates solution !

Let say changing the path from red to blue, in addition to connectivity between the hosts H_1 and H_3, we also want the connectivity path between these hosts should go through the nodes A_2 or A_3 (let say these switches are special nodes, can think in terms of security-way-pointing switches). So any insecure data transmission between the hosts can be detected and deny the passage in these special nodes. This is a weak consistency not a strong consistency property. For such consistency we have the ordering updates (from red to blue path), $A_2 A_3 T_1 C_1$ and $A_3 A_2 T_1 C_1$ sequence is fine. That is initially a red path

$$T_1 \longrightarrow A_1 \longrightarrow C_1 \longrightarrow A_3 \longrightarrow T_3$$

by doing the update of A_2 and A_3 not creating any new path between H_1 to H_3, after updating T_1 the result is neither a red path nor a blue path but a path which is following the weak consistency

(which is path should be connected via either A_2 or A_3).

$$T_1 \longrightarrow A_2 \longrightarrow C_1 \longrightarrow A_3 \longrightarrow T_3$$

after updating C_1 we have the blue path

$$T_1 \longrightarrow A_2 \longrightarrow C_1 \longrightarrow A_4 \longrightarrow T_3$$

Note : Suppose a packet is forwarded by T_1 before it updated its forwarding rule and reached C_1 after it is updated its forwarding rule, then the (weak) condition is violated in order to avoid such things give a wait time between the updating which make sure that all the data packets forwarded by the T_1 will be reached to corresponding hosts before C_1 gets updated. ◇

The (weak) consistency update problem discussed in the above example is a requirement expressed as *LTL* specification and expressing the network topology and semantics of network events as *kripke* [Cav+15] structure. Using the variant of *Depth First Search algorithm* the solution for consistent order of network updates is computed. In my understanding, the proposed algorithm complexity lies in factorial of number of updates i.e $\mathcal{O}(n!)$ where n is the path length between the hosts in the network.

6.1.2 Guided Network Synthesis

In the network guided synthesis, the topic address about checking the program correctness with respect to the specification while transforming the *core legacy programs* into the *Domain Specific Language* (basically abstracting and exchanging the data from high level language to another model) using the synthesis technique with constraints on output for the given input (i.e same as ***contract*** defined in the *Heptagon* program).

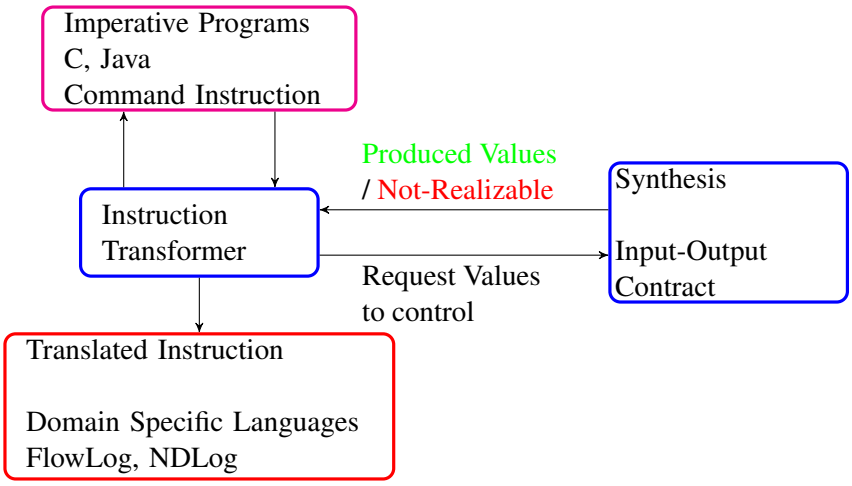


FIGURE 6.2 : A network Guided Synthesis Scheme.

The emergence of *SDN* paved the path for sharing the common resources in multi-tenancy framework, and global control mechanism and unified communication scheme for the large-scale network with wide and different kinds of devices and applications. To fully leverage these *SDN* platforms, many high-level domain-specific languages (*DSL*) have been proposed. This high-level language provides a data abstraction model to communication interfaces between various entities (also between *SDN* layers). These include logic-based languages such as *declarative networking*, *FlowLog*, functional languages such as *Frenetic* and *Pyretic*, and state-machine based languages *Kinetic*. These *DSL*'s enabled many new capabilities not found in traditional networks ranging from automatic verification, composition, debugging, to consistent updates. Despite these advantages, *DSL*'s had not gained wide-spread adoption in practice. There are many potential reasons (e.g.,performance concerns, limited expressiveness,etc.), but one major hurdle is the learning curve associated with the new programming paradigm (syntax and semantics). Particularly, declarative *DSL*s, like *FlowLog* and *ND-Log*, would require significant changes in programmer's reasoning process, due to the semantic differences between imperative and declarative programming [CWL18b].

In the guided network synthesis studies, it's about converting the legacy program into domain descriptive language rather than the traditional synthesis of a discrete event system. The working synthesis methodology for converting high level system instruction from the legacy programs into the domain specific language as depicted in the figure 6.2 is the theme of network guided synthesis tools like *Facon* [CWL18b], *NetEgg* [Yua+15].

Further Discussion : The importance of the instruction transformer in the network guided synthesis scheme has two-folds,

1 This imperative level instruction coming from the interface of manager or controller is purely logical based decision instructed either by the network manager (human) and/or code written by the system designer. This code should be human understandable.

  When it comes to deploy such instruction to the data plane, the data plane has to handle millions of data packets per minute (of-course it depends on the size of data plane). So it is better the network switches or routers code instruction should be efficient machine executable code rather than human readable.

  This job is solely done by the southbound protocols [Kre+15] (or more general term data abstraction). It takes the (*SDN*) manager or controller instruction and converts it into the data plane machine efficient code in the chosen domain specific language *DSL*.

2 In the process of translating the instruction from *SDN* controller to network switches or

routers in the data plane, there is an opportunity for synthesis and model check the consistency in the updates (deployment) of new instructed rules (i.e translated instruction). So, it will be useful in storing the partial information about the data plane in the contract box (Input-output contract) for checking the consistency of network updates and/or synthesis the updates. In fact this is what exactly the *Veriflow* [Khu+13] does, but it does elaborate model checking on complex specifications (connectivity for the different hosts, non-blockingness, data path route cycle), which of course needs much more information about the data plane and time to process before deploying the new rules into the data plane (please refer experiment case of the paper [Khu+13] : for more than 170 hosts, the *Veriflow* takes more than 20% of the southbound processing time, which will add further latency in deploying the translated rules). In our understanding, instead of checking the complex specifications, just checking the safety properties (like necessary and minimal sanity check) won't introduce too much lag between the order (forward instruction rule) from the controller and deploying the translated rule into respective network elements.

The reason why we mentioned the connectivity, non-blockingness and data path route cycle are complex specification is because of the time complexity (more precisely the computation time) which is around the order of data plane network size $\mathbf{O}(n), n = |V(\text{Dataplane})|$. Let's take the connectivity property between say two user's devices d_1, d_2 which locates in the data plane switches Sw_1, Sw_2. When the controller plane got the instruction about setting the data connectivity path between these two users, first it observes the data plane position (in the data plane network) and to generate the forwarding rule, the controller has to go through the data plane network (topology) and find the physical network path and then generate the forwarding rules for the switches involves in the physical network path between users d_1, d_2. In this case clearly the time complexity for the controller to generate the data forwarding path is around $\mathbf{O}(n)$ (well it's only upper bound). Further this model checking (or Input-output contract check) in the middle layer (i.e Southbound Interface) to check the connectivity specification, it is very clear that it's worst complexity will be $\mathbf{O}(n)$ (because it has to receive and read $\mathbf{O}(n)$ forward table update rules and further do additional computation on checking the correctness). For a small size data plane network, this complexity is fine but imagine when the size of data plane network is large, then however smart techniques we have, for sure the computation involves in the input-output contract in the middle layer (i.e Southbound Interface) will increase the latency between generated forward table rules in the controller plane and deployment of those rules (well translated rules) in the data plane switches. This is exactly the reason behind the increase in computation time in the *Veriflow* when the number of hosts and or the data plane switches increases. Here, we mentioned the upper bound complexity only i.e $\mathbf{O}(n)$ where n is the number of nodes in the given data plane, but in some situations, when the

data plane network is highly connected in the sense that between any router or switches the distance will be in the order of $\log(n)$, so the actual computation procedure will be much less, but note one thing for each and every computation, we are not just using the computing resources but also consuming energy, it is always advised to avoid unnecessary consumption of energy so as computation especially in the huge network scale like mobile, fix or wireless networks and there are dedicated communities works on this problem about reducing the energy consumption among various service networks [Bol+14], and *IoT* applications [Bad+18].

Let's revisit the idea about the local controller introduced in our *SDN* Nokia platform to avoid the security property (or more specifically the 'data privacy' property) for the mobility case as described in chapter 5.

Problem of Latency between Control and Data plane : It is important to analyze the latency problem arising with *SDN* network framework, this latency problem can introduce safety property violation as mentioned in last chapter section 5.5.6.

In the data plane network, physical connection of users, *IoT* devices are dynamic in choosing the open access port in the data plane and devices will change their position from time to time. In such a scenario, the control plane has to observe the network user's device position in the data plane and update and deploy networking rules in the data plane routers or switches instantly otherwise there can be data privacy violation this is the case in mobility of devices in section 5.5.6 *Isolation* property violation in section 5.5.5. This violation of *Isolation* property made us to introduce an extra local controller to each switch such that before sending a data packets to the outgoing open access *IoT* port of the switch, it checks the destination address of the data packets refer to the current device Id (or more exactly MAC Id) connected to this open access port. This solution itself is a synthesis solution to data privacy safety specification i.e *Isolation* property.

It is important to know that any instruction coming from the *SDN* controller will be deployed by south bound interface protocol (see fig. 6.3), it essentially transforms controller instruction rule into *DSL* rule and deploy it into the respective switches. When we are going for the *SDN* layers concept to configure and reconfigure the networking switches to meet various applications and deploy in real time, for such a cost effective solutions, it may not possible to avoid latency completely between the switch intent request and the *SDN* controller response. In the sense that, when a switch sends an intent request about the local changes, the *SDN* controller has to manage a large number of switches intent's request and other applications related procedure, it might not respond to the requested intent immediately. Such cases the latency can't be simply avoided and may be designed in such a way to reduce the possible latency.

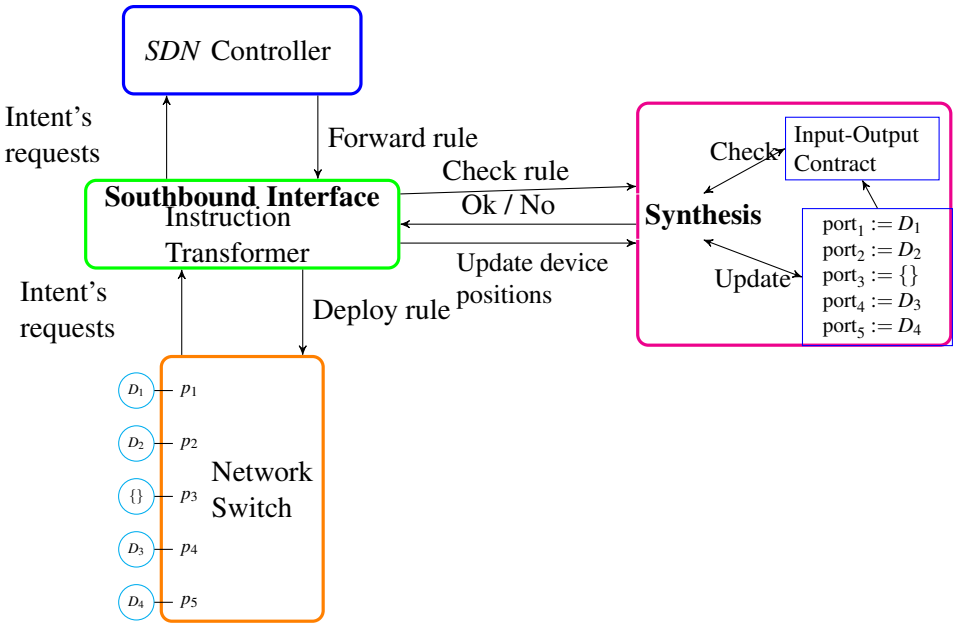


FIGURE 6.3 : Synthesis Forwarding Rule

Suppose that a data plane network switch sw has a l number of open access ports refer to them as $\{p_1, p_2, ...p_5\}$. Let say that d_1, d_2, d_3, d_4 are the current devices connected to the open access ports p_1, p_2, p_4, p_5 respectively, whenever a switch got the new device (connection) in its open access port, it sends an intent request immediately, if the south bound protocol has state full machine, that for each such intent request from the switch, it stores the current devices in the respective open ports. Whenever the *SDN* sends deployment instructions i.e data packet forwarding rule which contains the source, destination devices id and the incoming, outgoing ports numbers. The south bound open interface protocol checks the current device id and its open access position with the destination and outgoing port number from the forwarding rule, if it matches, it will translate the forwarding rule and deploy it into the respective flow table switch otherwise drop the rule. Such a solution is depicted in fig. 6.3. Although this solution does not solve the problem completely, the reason is that an already deployed rule inside the switch has to be blocked when there is a change of devices in its given port. That is to say that a device D_1 is initially connected to a open access port p_i of the switch sw and there is a (data packet) forwarding rule in the forward table of the switch, and there is change of device say D_1 is disconnected and D_2 joined the open access port p_i, so that switch sent an intent's request for this new connection, but in the meantime of updating the new forwarding rule for this particular open access port, the switch sw should block any data packet forward to this particular open access port op_i. This mechanism I refer to as the adaptive switch mechanism. Notice one thing, such an adaptive mechanism is not needed when there is a dedicated local controller for each switch as in section 5.5.6. So this adaptive switch mechanism is not really complex just editing the control mechanism (decision structure fig. 2.7 little bit we can ensure the data privacy property) as explained below :

Destination Check Rule :

- In addition to the flow table rule, each switch maintains a list of devices currently connected to its open access ports the list looks as follows,

  connection list : = $\{\text{port}_1 : D_1, \text{port}_2 : D_2, ...\}$

- Whenever there is rule matching and switch has to deliver a *data packet* pkt to the open access port say port_i before delivering the packet pkt to this port, it will check whether the current device connected to the port port_i are the same of the destination address of the packet pkt.

- The decision structure has to be modified as explained in fig. 6.4.

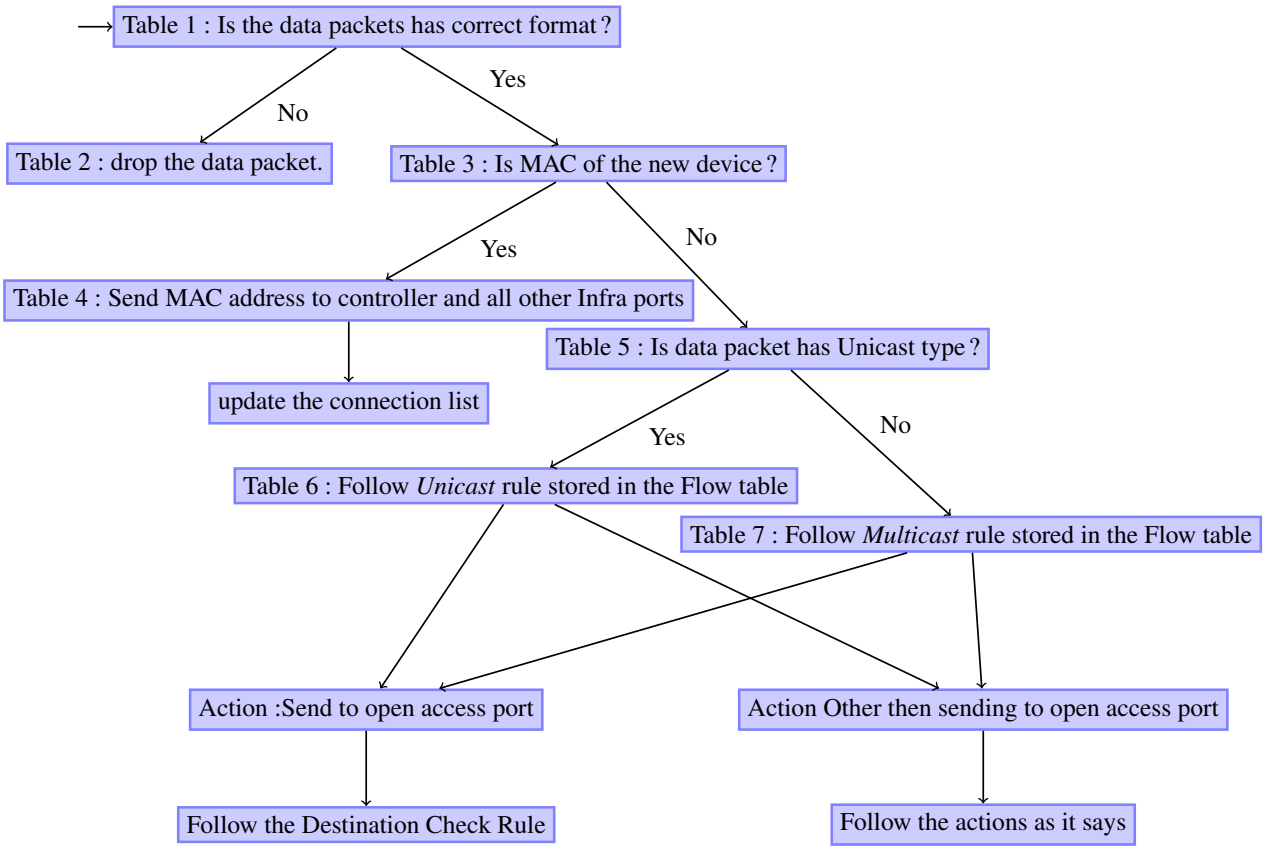


FIGURE 6.4 : Adaptive Decision Structures of Network switch Flow Table

This solution will help us to save from not sending the action to different actuator in case the devices are *IoT* type, it coordinate different sensors and actuators in safe way, so that the data packets instructions exchanged between the *IoT* devices are safe and correct.

6.1.3 *SMT* based Synthesis of *SDN*

The most recent work in synthesis of *SDN* model is supervisory control and data acquisition [JRG21] (*SCADA*) network in a smart grid. This work is really recent, author of this thesis got to know only after we did the experiments for the case study of *SDN-IoT* platform. TIt [JRG21] gathers the real-time data (intent requests), *SDN* topology and produce the network response (i.e controller) based on the specification which are expressed in satisfiability modulo theories (SMT) [BT18]. It encodes the specifications of maintaining the *connectivity* and providing alternative path in case existing connection fails by constantly monitoring the availability of switches and connections links, it also encodes *bandwidth* constraint, *priority* in handling the users requests as *SMT* specifications and using the variant of *DPLL* algorithm [OC99] it produces the output for the every input request based on the condition expressed in *SMT* specifications.

Satisfiability Modulo Theories (SMT)

Boolean logic with propositional predicates combined with non-boolean variables and relational predicates but still less expressive then first-order or higher order logics [Cla+18]. This trade of compromise makes SMT problems decidable with great advantage of practical efficiency when it is restricted to the certain symbols of models like theory of equality, integer, real, arrays, and lists [BFT16].

In fact the practical efficiency of *SMT* solvers are very well exploited in this work [JRG21] as well as model checking of *SDN* by *Vericon* [Bal+14] as mentioned in the chapter 5. Although usage of *SMT* solvers show the practical efficiency, when using it for either synthesis or verification one has to use it with care i.e have to consider the decomposition of the expressed specifications and the entire system. There are some works on these aspects i.e using *SMT* solvers with decomposition [Poz+15] and divide and conquer [McC18] method in synthesis of network events.

Now we move on to describe the particular reactive synthesis tool we are using for the experimentation of various *SDN-IoT* functionalities.

6.2 Synthesis Tool

In our experiments on the synthesis of *SDN* distributed system, we are using reactive synthesis tool called as *BZR* have Heptagon [DRM13] for model creation, and specification and ReaX [BM15 ; Ber+17] for producing the control synthesis supervisor for the given model and specification pair.

Heptagon/BZR Reactive Synthesis Tool Heptagon is a reactive synchronous data flow language [DRM13] which allows us to describe the composition of automata models of the given distributed system, with a syntax allowing the expression of control structures.

For a given subsystem of the distributed system *DS* can be expressed as an automaton by writing the heptagon code under the name *node* with inputs and outputs. Syntax for creating such a given entity of *DS* will be as follows,

node entity_name($x_1 : t_1; ..x_n : t_n$) = ($y_1 : t'_1; ..y_p : t'_p$)
    var $z_1 : t_1$"; $..z_q : t_q$";
let
    Declaration statements
tel

FIGURE 6.5 : Creating the model description of entity of the given distributed system

In this node (figure 6.2), it creates the model for the given entity of the distributed system with set of inputs $\{x_1, x_2, ..., x_n\}$ and set of outputs $\{y_1, y_2, ..., y_p\}$ each variable has certain types can be either Boolean, integer, enumerated type or the array of these types. Apart from those input and output variables each entity can also have its own internal variables set, which have to be declared using the statement **var** as shown in the figure6.2. The behavior of the entity should be declared inside the node function **let** and **tel**.
The enumerated types can be created for our purpose to describe the behavior of the system. For example, if we want to create a certain enumerated type to mention the system status being critical, normal or safe can be created as follows.
type modes = Critical | Normal | Safe
One can create as many as enumerated types to write behavior of the interested distributed system *DS*. The node of each entity produces the output based on the input values and current internal variable values and also the previous values of the same internal variable. For example when we defined a variable say x as **last** x : int in the declarative statement when we call x, it uses the current value of the variable x, when we call it by last x it uses the previous value stored in the variable x.

The sequence of declaration statements inside the **let** and **tel** of a node describes how inputs are transformed into outputs, with a set of control structure statements. These declaration statements define the values of outputs and internal variables using the current values of the inputs, and the current state of the node. New values for input values are given at each execution step, where declarative statements proceed to evaluate all the output and internal variables together based on the current input values and last internal variable values.

The set of declarative statements one can use for writing the behavior of the distributed system *DS*.

- an equation $x = e$, defining variable x by the expression e at each activation instance.

- creating a new node to write the code in functional way $(y_1,...,y_p) = f(e_1,e_2,...,e_n)$ defining the variables $(y_1,...,y_p)$ by the application of the node f (i.e the node created to do a specific function) with values $(e_1,e_2,....,e_n)$ at each activation instants.

- switch, present, reset control structures

- an automaton.

We will give some details about the automaton declaration statement, one is interested to know the full syntax usage for those control structures refer to the site "heptagon.gforge.inria.fr/pub" to find the heptagon user manual. In this document you can learn the detailed syntax and procedure to install the Heptagon/BZR tool (which is an open source like SPIN verification tool).

Compared to all other declarative statements which are executed based on the current input, internal variable values and/or previous values (if that variable is defined with last) set the new values to the internal variable and output values. The automaton which sets the current state of the values in the following ways.

automaton
  state Up
    do $x =$ **last** $x + 1$
    until (**last** $x \geq 10$ **then** Down)
  state Down
    do $x =$ **last** $x - 1$
    until (**last** $x \leq 0$ **then** Up)
end

FIGURE 6.6 : Automaton example

In this example (as mentioned in the figure 6.6), based on the current **state** (only one state is possible at any activation under this node), it will execute the declarative statement written under this state and then it will go the the statement **until** *condition* **then** *State_name*, it will check the *condition*, if it is true then in the next activation instant of this node, when the automaton executing its statement, it will execute the declaration statements written under the **state** *State_name*.

Contract for Controller Synthesis Contracts are an extension of the Heptagon language, so as to allow discrete controller synthesis on Heptagon programs. The extended language is named *BZR* (pronounced the same way as bizarre !).

The Heptagon associate to each node a contract, which is a program associated with two outputs : an output e_A (assumption condition on the the input variables value) representing the environment model and an invariance objective e_G (requirement condition on the output

variables value) with a set of set of controllable variables $\{c_1, c_2, ..., c_n\}$ used for ensuring this objective.

Declaration of contracts for a given node f looks as follows,

node $f(x_1 : t_1; ..x_n : t_n) = (o_1 : t'_1; ..o_p : t'_p)$
contract
var internal variables for condition declaration
let
equation declaration for internal variables based on input variables value
tel
assume e_A
enforce e_G
with $(c_1 : t_1"; ..c_m : t_m")$ (* controllable variable *)
var system variables
let
set of output variables values declared based on inputs, system and controllable variables value i.e distributed entity behaviour here
tel

FIGURE 6.7 : Contract syntax - for System Specification

An abstract view of this synthesis of input-output contract of heptagon model can be depicted as in figure 6.8, These controllable variables are not assigned by the programmer but produced by the *BZR* reactive control synthesis tool. This declarative contract statement contains predicates that the functioning of the system model must always satisfy. These properties are declared as control objectives in the *enforce* statement. When the system model that describes the dynamics of the system does not meet the properties, *BZR* reactive tool *ReaX* [BM15 ; Ber+17] generates a controller that enforces the latter when controllable variables are defined in the model, declared as local variables in the *with* statement. The generated controller determines the value to assign to the controllable variables in order to restrain the modeled behaviors to satisfy the properties.

In order to produce the control synthesis of model of the given distributed system in Heptagon language, we have to translate the heptagon model into the *ReaX* program, and the *Reax* tool

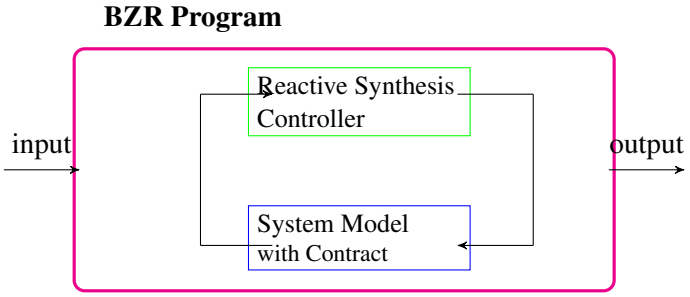


FIGURE 6.8 : Abstract Input-Output Contract Synthesis

compute the controller synthesis for the written distributed model and the expressed contract (specifications), produced *Reax* controller has to translate back into Heptagon program, converting system model from heptagon to *ReaX* and vice versa is automatable i.e Heptagon and *Reax* compilers do these translations. Once we have the controller of a written distributed model in heptagon we can directly use the controller and encapsulated in the Heptagon model. One great advantage of using the heptagon model is we can convert the written model into conventional high level programming languages *C* and *Java*.

ReaX for producing the controllable variable values *ReaX* is also a reactive synchronous data flow language as Heptagon. As in Heptagon, one can write the distributed system in terms of the composition of automata and *STS* (symbolic transition system). In *ReaX* also, one can write both automaton and *STS* with arithmetic functions (refer chapter 3 section 3.2 definition 3.5). In addition to that, *ReaX* tool can generate the synthesize controller for the expressed specification and model (Input-output contract). For more details about writing distributed models in *ReaX* language we refer the reader to read the manual. "http ://reatk.gforge.inria.fr/" a link where one can find the user manual, installation step and examples. Meanwhile, the important things is that this *STS* with optional contract synthesized controller generated by *ReaX* tool (either a predicate over the input, output and system state of the Heptagon program or another *STS*) is automatically derived by the Heptagon compiler.

More formally, the derived *STS* from the Heptagon of 6.8 is given by $S_f = (X, Y = Y_c \cup Y_{uc}, T, \Theta_0)$ and $K_\Phi = \Box(a_g \Rightarrow e_g)$ (meaning : always guarantee e_g if the assumption a_g holds) a state-predicate over X (assuming from simplicity than the latter one is derived from the state variables of the Heptagon system model) depicted by figure 6.9. X corresponds to the state variables, Y_c (Y_{uc}) are the controllable (resp, uncontrollable) variables whereas T is the transition relation given by $T(X, Y_c, Y_{uc}, X')$.

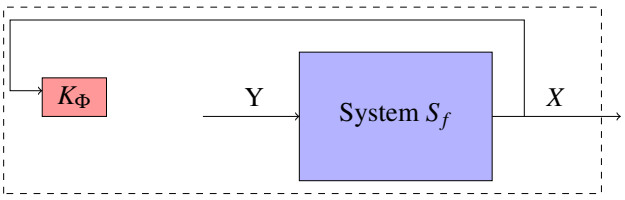


FIGURE 6.9 : From an Heptagon node to an uncontrolled ReaX STS

From S_f and K_Φ, *ReaX* is automatically deriving the controller C ensuring K_Φ on S_f as depicted in 6.10 following the scheme of section 3.4.3 for finite or infinite system S_f by means of abstract interpretation in the latter case.

Example For the sake of completeness, we give one example about how to generate the reactive controller for the given Heptagon model using *ReaX* tool.

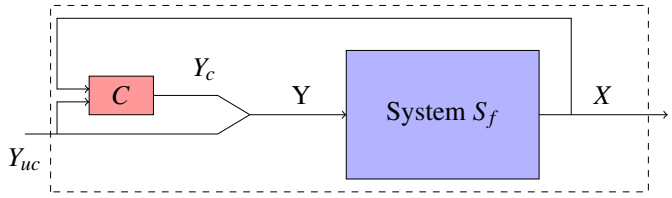


FIGURE 6.10 : The corresponding controllable ReaX STS

EXAMPLE 6.2 *We demonstrate the control synthesis for the simplest mutex (mutual exclusive) to manage the two machines A, B seeks to use a common single resource which can be used by only one at a single time instant. The written code in the fig. 6.11 left side*

Using the BZR *tool to compile and generate the controller for the above mutex program using the command "bzreax -s mutex.ept mutex" will produce the executable program name sim when we run this program it will show the simulation result for the code in left side of fig. 6.11 generated discrete control synthesis synthesis result as in right side of fig. 6.11 :*

In the figure 6.11 right side, the input command Want No *refers to the first machine (a) seeking the common resource and second machine (b) not seeking that resource and* No Want *refers to the vice versa. Similarly the input command* Want Want *refers to both machines seeking the common resource. Where the output command* Ok Nexttime *refers to granting the permission to machine a and* Nexttime Ok *refers to granting the permission to machine b.*

Even though the above program avoids output Ok *to both machine (a and b) requests, when both machines keep sending* Want *messages, the discrete control output is* Ok *to machine a only. In order to avoid such a scenario, we have to refine the mutex model to include the fairness condition for both machines.* ◇

In the above example, computation involved before producing the executable code (and simulation) of the model with contract is depicted in the figure6.12.

In the rest of the chapter, we will describe the control synthesis experiments carried out with the *BZR* tool.

The use case for the experiments consists of managing the *SDN*-based network resources and connectivity knowing the Edge Computing server location to meet the stringent latency requirements of *Vehicle to Everything* (abbreviated as *V2X*) communications. Thus, we will synthesize a supervisor for the manager allowing the latter to meet the different V2X latency requirements. The manager here is often called as *orchestrator* as it manages both the SDN-based network resources and the Edge Computing compute and storage resources.

```
type req = Want | No
type acpt = Ok | Nexttime
node mutex(ma_req, mb_req : req)
      return = (ma_acpt, mb_acpt : acpt)
contract
var a1, a2, g : bool;
let
   a1 = (ma_acpt = Ok);
   a2 = (mb_acpt = Ok);
   g = a1 & a2;
tel
assume true
enforce (not g)
with (c :bool)
let
   automaton
              state Init
              do
                 if ((ma_req = Want) & (mb_req = Want)) then
                   if c then ma_acpt = Ok;
                             mb_acpt = Nexttime;
                   else ma_acpt = Nexttime;
                        mb_acpt = Ok;
                   end;
                 else
                   if (ma_req = Want) then ma_acpt = Ok;
                   else ma_acpt = Nexttime;
                   end;
                   if (mb_req = Want) then mb_acpt = Ok;
                   else mb_acpt = Nexttime;
                   end;
                 end;
              until true then Init
   end
tel
```

```
./sim
Input : Want No
output : Ok Nexttime
Input : No Want
output : Nexttime Ok
Input : Want Want
output : Ok Nexttime
```

FIGURE 6.11 : Mutex Code with Input-Output Contract (Left) and Simulation result (right)

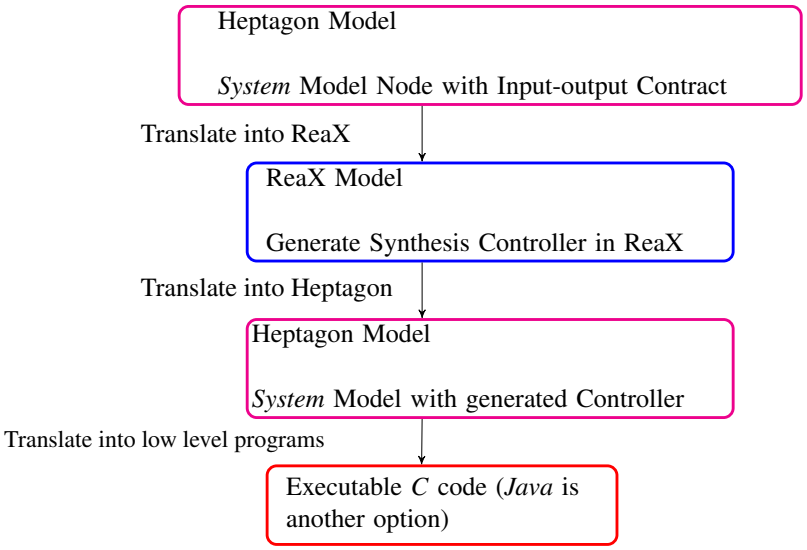


FIGURE 6.12 : Procedure in simulating the Synthesis of Input-output Contract Heptagon Model.

6.3 Control Synthesis of a typical *SDN* Application - A Modular Approach

We are interested in the application of synthesis techniques to the *SDN* system, in particular the *SDN* manager to satisfy and impose various specifications in the network node (data plane) with a set of network users. We illustrate an application scenario of Edge Computing for *V2X* [HHK20] application and modelization of the application in particular by expressing the requirements as a set of synthesis specifications and generating the specifications synthesis controller using the reactive synthesis tool *BZR* (Heptagon for modelization and ReaX for generating the controller).

6.3.1 An Application Scenario : Edge Computing V2X communications

Edge Computing for *V2X* Application

Edge computing is a Fog/Multi-access Edge Computing type which moves cloud resources in proximity of users which provides strong support for latency-sensitive applications such as *V2X* communications and services. *V2X* framework provides a number of applications such as maintaining healthy road congestion, safe interactions between vehicles and non-vehicle road users, advanced driving assistance like autonomous vehicle operation, vehicles sharing local sensor data data like trajectories and maneuvers coordination, and so on.

Let us now describe a high level application scenario :

- Edge computing servers are distributed at the network edge in order to offload processing requirements from user's devices with limited processing resources.
- The application client is running on the user's device while the application server is running on an edge server.
- There are two categories of devices : high-priority ones (with critical applications) and low-priority ones.
- The application client and the application server should belong to the same *vSpace* (recall the earlier notion of virtual space or *VS* - group of devices), so that they can have connectivity with each other via the network.
- Device-to-device communications are also possible. For this purpose, the to-be-interconnected devices should be part of the same *vSpace*.
- For critical applications (on high-priority devices), the latency between the client and the server should be maintained under a maximum defined by the user's service level agreement (abbreviated as *SLA*).
- The manager is aware of each user's *SLA*, of the *SDN*-based network resource consumption and computes resource consumption accordingly to the service requests
- The manager orchestrates the sharing of the *SDN*-based network and computes resources between user's devices to meet high-priority user's *SLAs* and to maintain continuity for the low-priority users services.
- User's devices are supposed to be in permanent mobility.

For instance, we have a *V2X* application relying on edge computing offload (i.e. vehicles are users' devices in this application). High-priority devices in real situations might be vehicles like ambulances or patrol vehicles. The low-priority devices will be all the other vehicles. Ambulances and patrol vehicles could be part of the same smart city *vSpace* group, since there could be voice/media communications between them. Also, ambulances pertain to the hospital *vSpace* and patrol vehicles pertain to the security *vSpace*, and ambulances and patrol vehicles pertain to safety *vSpace*. Other vehicles could be part of a common open public *vSpace* so that they can share road information together and other information (e.g. social media).

In the scenario picture 6.13, the Orchestrator/Manager access the status information about the distributed set of edge computing *EC* servers via the compute controller and gathers the various devices requests about creating the *VM* requests updates via *SDN* controller. From the various inputs command and status information Orchestrator/Manager set up (send back) the commands to full-fill the *V2X* applications via the Synthesis of supervisors concepts.

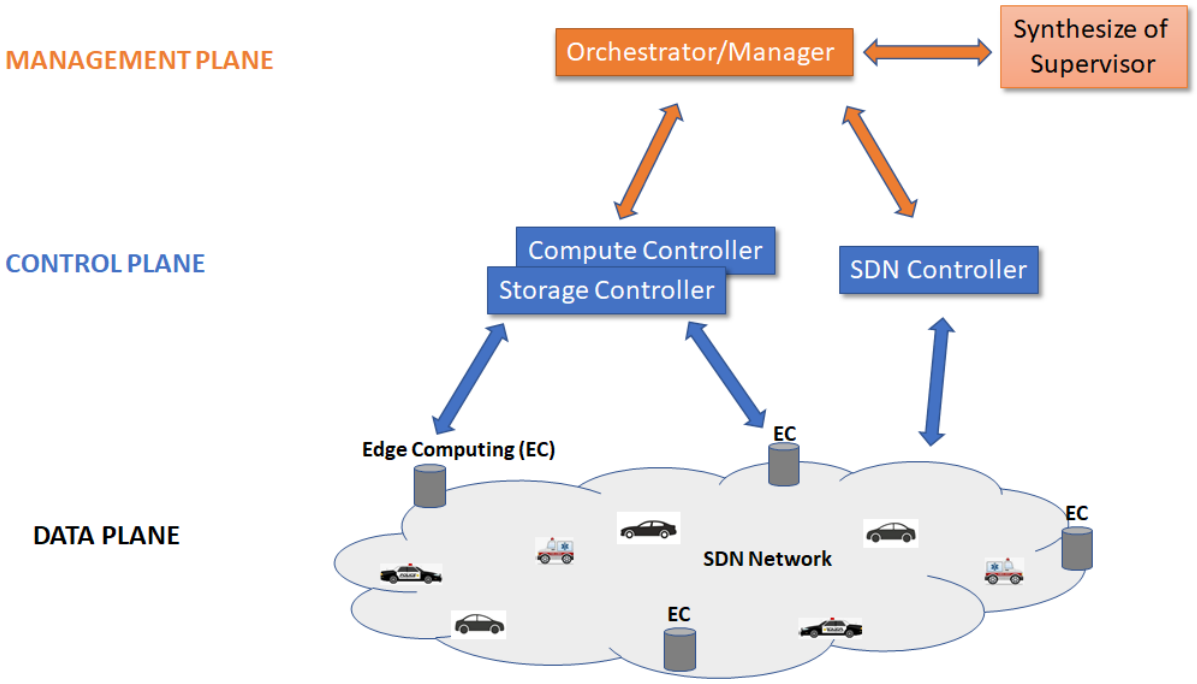


FIGURE 6.13 : Proposed V2X Synthesis Scheme

Application Architecture Let's the set of devices which use this application denoted as $\text{Dev} = U_n \sqcup U_s$ where the set $U_n = \{D_{n_1}, D_{n_2}, .., D_{n_m}\}$ are *normal* devices (i.e low-priority devices), whose requirements is connect to the application and create a *processor* (or we can refer it as *VM* (for Virtual Machine)) in one of the edge server available in the network, and the set $U_s = \{D_{s_1}, D_{s_2}, .., D_{s_l}\}$ are the *Special* users (i.e high-priority devices), whose requirements is connect to the application and create a *processor* (i.e *VM*) for its own application in the local server (nearest server to keep the latency as minimum as possible). In the application, a network has a set of distributed servers; each server has a constraint in providing the number of *processor*'s (i.e *VM*'s) at any given time. Apart from this basic functionality, the system architecture has a natural dynamic of its position in the application in the sense that the user joins the network application via open access port and can change its location by disconnecting from the current open access port and join the application from other different open access port positions.

In the given *SDN* layered architecture 6.3.1 with concept of *virtual Places* (*vPlaces*), that is the network application is partitioned into n number of domains. Let's say the network architecture resource management placed an edge server in each data plane domain which is available for the users which is connected to the data plane to create an *VM* (a processing unit). Each edge server has a limit (obviously each server can have finite memory size and can do finite computations) on the number of *VM* services it can provide to the end users.

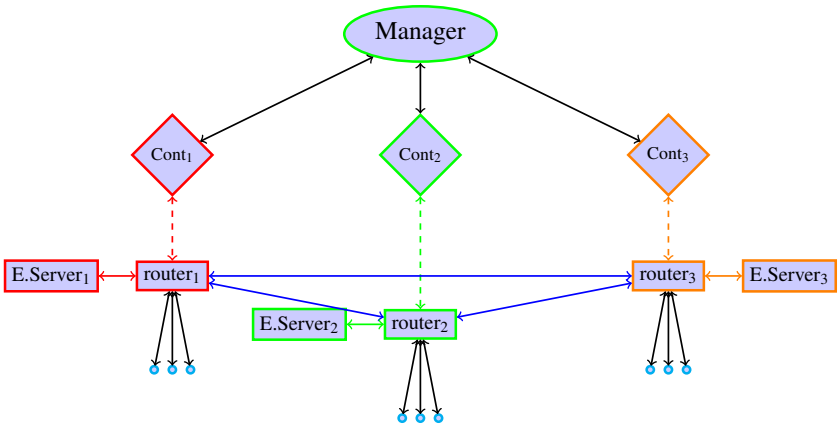


FIGURE 6.14 : System Architecture

6.3.2 Abstract Specifications and Model

System Specifications

Here we will list the set of specifications as objective for the synthesis controller. *VM Limit Objective* : Each edge server can provide only a limited number of *VM*'s and should not overload the server condition by allocating too many *VM* to the network users. In the set of edge computing servers, each server can accommodate only a limited number of *VM* to the set of users (i.e set of devices either *special* or *normal* type) that we say the limited number as a normal condition.

$$\Phi_{\text{VM}} = \Box \bigwedge_{\text{vP}\in\text{Dom}} (\text{vP}, \text{Normal}) \tag{6.1}$$

Whenever the *special* characteristic devices requested to connect to a local edge server and to create a *VM* for it, it should be allocated.

Service-Objective Whenever the *normal* characteristic devices requested to connect to a server (may be local or remote) to create *VM* for it, manager should at least allocate some remote server for this application, when it comes for the special device, the manager should create the *VM* in the requested server and if needed the normal devices *VM* should be moved to different server in order to meet the continuity of the service to the normal devices.

$$\Phi_{\text{Priority}} = \bigwedge_{d\in\text{Dev},\text{vP}\in\text{Dom}} \begin{array}{l} \Box\big((d, \text{reqvm}, \text{vP})\,? \wedge (\text{vP}, \text{Normal}) \\ \qquad \Longrightarrow (d, \text{acpt}, \text{vP})\,!\big) \end{array} \tag{6.2}$$

Objective- Connecting vSpace *Devices* : In the given set of devices and clusters of devices as *vSpace*, the manager should sets up the connection request of the device to connect to its com-

mon *vSpace* devices to satisfy the communication between various common *vSpace* devices.

$$\Phi_{\text{vS}} = \bigwedge_{d_1,d_2 \in \text{Dev}} \begin{array}{l} \square \big((\text{VS}(d_1) == \text{VS}(d_1)) \wedge (\text{connect}, d_1, d_2)\,? \\ \quad \Longrightarrow (\text{acpt}, d_1, d_2)\,! \big) \end{array} \tag{6.3}$$

System Model

In the dynamic nature of network architecture (users changing the open access port positions), *Manager* has to full-fill the above objectives or specifications.

Manager is a policy maker, by knowing the abstract view on the current system position (state value of the current network architecture) it produces the set of instruction to the logical implementer i.e Cont as mentioned in the figure 6.3.1, based on the instruction provided by the manager, each logical implementer Cont re-route the paths between various users to connect to servers by setting appropriate routing rules in the network routers.

Routers gather the status of *open access* position status and pass the users messages to the manager through the *logical implementer* i.e *SDN* controller. *Logical implementer* gather those information and abstract the message content by only considering the recent changes and inform the abstract current network scenario to the *manager*.

In the following texts, we provide a minimal but the enough description about the model description to meet the above set of specifications $\Phi_{\text{VM}}, \Phi_{\text{Priority}}$ and Φ_{vS}.

The key concepts involved in this model are listed below,

- **Device type concept** the set of application users referred as device Dev which is further partitioned into normal $U_n := \{D_{n_1}, D_{n_2}, .., D_{n_m}\}$ and special $U_s := \{D_{s_1}, D_{s_2}, .., D_{s_l}\}$ devices

- **Devices *vSpace* concept** and further we cluster the set of devices as a common *vSpace* group $\mathsf{VS} = \{V_1, V_2, .., V_f\}$, with $V_i \subseteq \text{Dev}$ for all $1 \leq i \leq f$, and $\cup_{i=1}^{l} V_i = \text{Dev}$.

- **Domain *vPlace* concept** as in the figure 6.3.1 we have number of domains and each domain has dedicated edge computing server as mentioned earlier.

- **Main function** The main function to implement this application for the manager primary function is reading three different types of inputs from the users namely requesting an *VM* creation $(\text{Req}, \text{dom}, \text{device id})$ to the requested edge computing server, deleting $(\text{Del}, \text{dom}, \text{device id})$ an existing *VM*, and request for building device to device communication $(\text{Conct}, \text{device id}, \text{dest device id})$. Upon receiving one such incoming message, the main function from the location ReadMes call the following functions,

  - For the incoming message of type requesting an *VM* creation $(\text{Req}, \text{dom}, \text{device id})$ it calls the function **SyntDecisionVMalloc** which has the *VM* specification equation 6.1, it responds by either accepting the request (*Acpt*) or rejecting the request

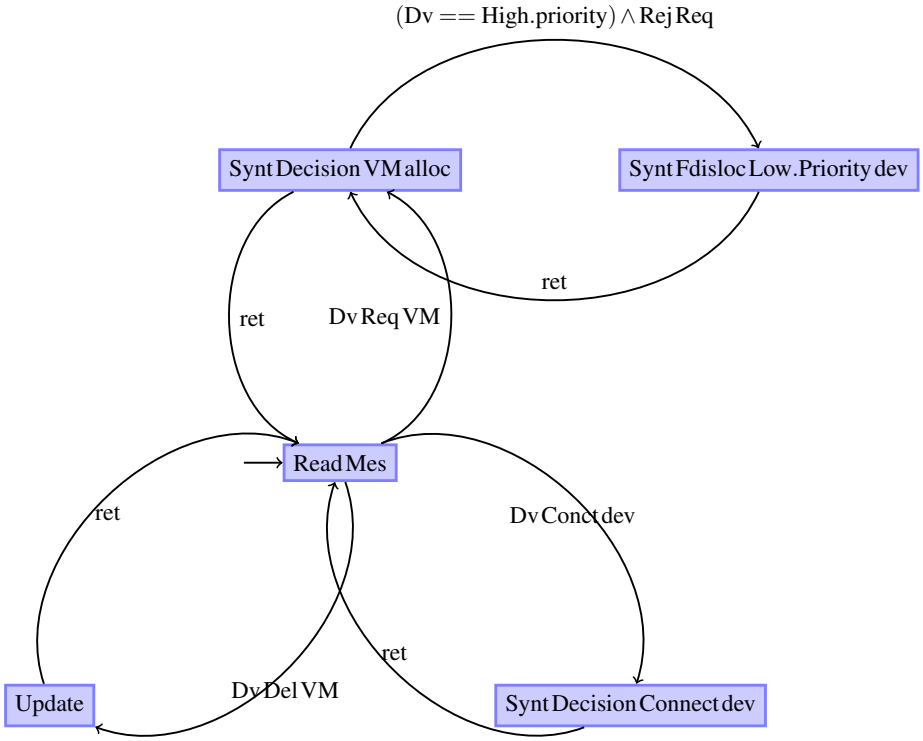


FIGURE 6.15 : A high level model description.

(*Rej*). In case there is already an enough number of users using that particular requested domain server and the requested device is a *special* device type, it will call the synthesis function **Synt Fdisloc Low.Priority dev** which has the specification equation 6.2 which find the low priority device i.e *normal* device type and move its *VM* from that particular server to the remote (another) edge computing server. By this mechanism it accepts the special device type *VM* requests.

– For the incoming message of type deleting (Del, dom, device id), it call the **Update** function to delete the *VM* and update the system (manager) information.

– For the incoming message of type request for building device to device communication (Conct, device id, dest device id), it calls the synthesis function **Synt Decision Connect dev** which has the specification equation 6.3and decide to command the application Cont's to build the communication link between the requested device to the destination device dest device id or reject the requests.

– A short pictorial description of main function calling all above mentioned functions are depicted in the figure 6.15. Among the list of node composed in the figure 6.15 **ReadMes**, **Update**,**Synt Decision VM alloc**, **Synt Fdisloc Low.Priority dev** and **Synt Decision Connect dev**, the last three nodes are the node with contracts (specifications) allocating emphVM for the requested device, low priority device force to move to different domains to free the space for the high priority devices (priority

specification), and establishing data connecting path among various devices (*vSpace* specification) respectively.

For the sake of completeness, we show one synthesis node function which we used in the model 6.15. The figure 6.16 is for the contract synthesis node expressed for the **Synt Decision Connect dev** which is the simplest synthesis node among the others. In this modeling, we took the device and server as an abstract entity and their respective intent requests and response from the manager node via data plane is the respective input and output of the manager node. One such simple device behaviour is modeled in the next section.

node connectingvSapce_device (devX,devY :vSpace) returns (outm :messageout)
contract
var g_1, g_2, ab :bool;
let
ab $= (devX = devY)$
$g_1 = (outm =$ Acpt$)$
$g_2 = (outm =$ Rej$)$
tel
assume true
enforce (not ab or g_1) & (ab or g_2)
with (c_m : messageout) (* controllable variable *)
let
outm $= c\_m$
tel

FIGURE 6.16 : Contract Node for connecting the same *vSpace* devices

Experimental Results

We experimented the above proposed experimentation on our manager modeled in Heptagon and expressed the above three specifications as Input-output contract, and by using the modularity aspect as mentioned in chapter 3 and 4, we generated the controller supervisors for the above listed specifications. The produced controller for these three specifications are integrated with the main function node and generated the final *C* code for the above manager model. We are showing the detailed compilation stats in the table 6.1 for the written manager model in heptagon, controller for the above listed specifications and final generation of high level *C* programming code.

Now, we will move on proof of concepts of compositional control synthesis applied to the layered *SDN* platform.

Parameter, Settings	Code Transformation		Reax Generation of synthesis controller			C code Generation
	Hept to Reax	Reax to Hept	VM Const	VS Con	Priority	
8 Devices, 3 domains	0.164sec	0.036sec	0.063sec	0.037sec	0.156sec	0.960sec
13 Devices, 3 domains	0.346sec	0.031sec	0.041sec	0.036sec	0.179sec	2.434sec
18 Devices, 3 domains	0.825sec	0.046sec	0.042sec	0.035sec	0.296sec	5.837sec
18 Devices, 4 domains	1.071sec	0.057sec	0.046sec	0.033sec	2.148sec	6.656sec
18 Devices, 5 domains	1.537sec	0.082sec	0.076sec	0.038sec	3.430sec	7.670sec
18 Devices, 6 domains	2.370sec	0.078sec	0.060sec	0.037sec	5.393sec	8.989sec

TABLE 6.1 : Experimental Results on the Modular Synthesis of three specifications

6.4 Compositional Control Synthesis Framework for the Layered *SDN* Architecture

In this section, instead of taking a specific scenario and apply the compositional reasoning technique, we will illustrate about the applicability of compositional control synthesis technique developed in the chapter 4 to the *SDN* platform a layered structure and synthesis generic high level *SDN* manager policies and network users (devices) requirements. With regards to an *SDN* platform, we introduce the following specification or the global property for the illustration of compositional synthesis technique, the local property or the intermediate specification. In this modeling, we capture the behaviour of manager and devices alone leaving the *SDN* controller and data plane network elements as a slave machine in the sense that these latter entities work based on the instruction provided by the manager and exchange the message coming from various network user's device. The decision making and dynamcity are because of the manager and devices alone.

In the layered *SDN* as mentioned in the figure 6.17. we are considering the following three subsections as the list of specifications on the global *SDN* system, the manager and the set of devices.

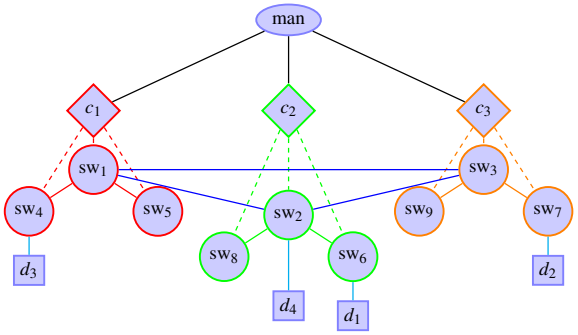


FIGURE 6.17 : SDN Architecture

6.4.1 Global Properties to be fulfilled by the *SDN* platform

At the higher level, several global properties/policies needs to be ensured on the global system. Note that they may be difficult to implement as they may depends of the kinds of devices connected, the possible routing connections between them, their local policies as well as some privacy properties. Amongst other, one need to ensure that : $\Phi := \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Phi_4$, where

- ***Path* Building or *vSpace* connection specification :** Φ_1 means that there exists a path for data forwarding between any pair of devices who are sharing a common *Vspace* $V_i \in \text{VS}$ (once the network stabilized) :

$$\Phi_1 = \begin{array}{l} \big(\big(\forall d_1, d_2 \in \text{Dev}, \exists V_i \in \text{VS}, d_1, d_2 \in V_i \wedge \\ \quad \exists i, j \in [1,..,k], d_1 \in \mathsf{dom}_i, d_2 \in \mathsf{dom}_j \wedge \\ \quad \text{net\_stable}\big) \implies (d_1, d_2) \in \text{Rout\_Path}\big) \end{array}$$

  The predicate net_stable refers to the situation where all the devices are connected to an access position and that the SDN manager is not receiving any new request from other devices, whereas Rout_Path means that a connection is possible between the two devices.

- **Bandwidth specification :** Φ_2 requests that the network bandwidth, $\mathscr{C}_{\max}^{\text{Dom}_i}$ of each domain dom_i is never overloaded.

$$\Phi_2 = \big(\forall\ \text{Dom}_i \in \text{Dom},\ \text{Cost}^{\text{Dom}_i} \leq \mathscr{C}_{\max}^{\text{Dom}_i}\big)$$

- **Priority specification :** Φ_3 requests that when a high priority device d requests to be connected to domain dom_i, then the manager should accept its request if by disconnecting several devices with a lower priority, the new global bandwidth is affordable by the domain dom_i.

$$\Phi_3 = \begin{array}{l} \big(\big(\exists d \in \text{Dev}\ R_{\text{req}}(d,i) \wedge (C_d \leq \mathscr{C}_{\max}^{\text{Dom}_i}) \wedge \\ \quad \forall d^{'} \in \mathsf{dom}_i (\mathscr{F}_{\text{prio}}(d) > \mathscr{F}_{\text{prio}}(d^{'})) \wedge \\ \quad \nexists j \in [1,..,k] d \in \mathsf{dom}_j\big) \implies d \in \mathsf{dom}_i\big) \end{array}$$

- **Exchange data with privacy specification :** Φ_4 requests that devices that do share a common Vspace $V_i \in \mathsf{VS}$ are allowed to exchange encoded messages only (referred as $\text{rcv}_{d,d'}(E.\text{data})$ and $\text{rcv}_{d',d}(E.\text{data})$).

$$\Phi_4 = \begin{array}{l} \big(\big(\forall d, d^{'} \in \text{Dev}, \exists V_i \in \text{VS},\ \{d, d^{'}\} \subseteq V_i\big) \\ \implies \text{rcv}_{d',d}(E.\text{data}) \wedge \text{rcv}_{d,d'}(E.\text{data})\big) \end{array}$$

Φ_1, Φ_2, Φ_3 are inherently related to the specification of manager i.e they can be realized by the manager itself, but Φ_4 is a global specification that can't be ensured by the manager or

the devices by themselves. Meanwhile, a part of the requested properties can be ensured or already ensured by the manager by building the appropriate path between devices which share a common *vSpace* and should not build the path between the devices which does not share any common *vSpace*. The final global specification Φ_4 can only be realized by the devices. Overall, the following problem has to be solved : Find a supervisor such that

$$\langle \text{true} \rangle (\mathscr{A}_{\text{man}} \parallel_{i\in[1,..,k]} \mathscr{A}_{c_i} \parallel_{\text{sw}_j\in\text{Sw}} \mathscr{A}_{\text{sw}_j} \parallel_{d\in\text{Dev}} \mathscr{A}_d)^{\mathscr{S}} \langle \Phi \rangle$$

Following the scheme of Problem 4.2, one can decompose the problem as follows : Compute supervisors such that

$$\langle \text{true} \rangle \mathscr{A}_{\text{man}}^{\mathscr{S}_{\text{man}}} \langle \Psi \wedge \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \rangle \tag{6.4}$$

$$\langle \Psi \wedge \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \rangle \parallel_{d\in\text{Dev}} \mathscr{A}_d^{\mathscr{S}_d} \langle \Phi_4 \rangle \tag{6.5}$$

where Ψ is an intermediate formula or local satisfaction property that have to be guessed (See section 6.4.2) and we obtain the result by compositional synthesis of theorem 4.5. Note that, we do not have supervisors for controllers and switches since these are just slave machines, meaning all *SDN* controllers actions are based on the instructions of the manager and predefined algorithms, and switches in the data plane just follow the data forward rules implemented by the sdn controllers. Indeed, all messages between the manager and the devices are not filtered by the switches and controllers, so we do not take them into account in eq. (6.5). Next we show how to solve eq. (6.4) and (2) in a modular way :

6.4.2 Properties that have to be fulfilled by the Manager

In order to satisfy the specifications to the manager, first of all we need a model for the manager, we provide the minimal modelization of the manager so that we can build a synthesis controller for the manager model with respect to the manager specifications.

Minimal manager behaviour expressed as transition system

where,

- $\sigma_{\text{req}} := \big(\text{req}, d, i, C_d\big) ? \in \Sigma_{\text{man,uc}}$ refers to the incoming message about the request of device d wants to connect to the domain (*vPlace*) Dom_i with service bandwidth of C_d, such incoming message will be always uncontrollable since it is not decided by the manager.

- $\sigma_{\text{disconnect}} := \big(\text{disconnect}, d, i, C_d\big) ? \in \Sigma_{\text{man,uc}}$ refers to the incoming message about the disconnection of device d from the domain Dom_i with its allocated bandwidth C_d such an message makes the manager to allow the disconnection of device d and free the usage of previously allocated bandwidth C_d.

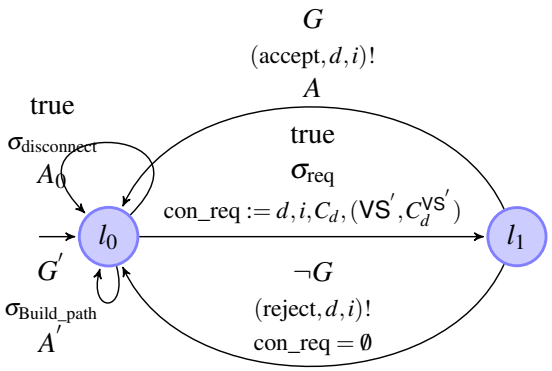


FIGURE 6.18 : Behavior of manager modeled as transition system

- $(\text{accept}, d, i)\,!, (\text{reject}, d, i)\,! \in \Sigma_{\text{man},c}$ refers to the manager's outgoing message of accepting the device d connection request to the domain Dom_i, and this message is a controllable and decided by the respective bandwidth specification Φ_2 controller.

- $\sigma_{\text{Build\_path}} := (\text{send}, c, \text{Build\_path}(d, d^{'}))! \in \Sigma_{\text{man},c}$ refers to the manager's outgoing message of commanding the *SDN* controllers to build the path between the devices d, d' and this message is a controllable and decided by the respective *vSpace* specification ϕ_1 controller.

- $G := (\text{Cost}^{\text{Dom}_i} + C_d \leq \mathscr{C}_{\max}^{\text{Dom}_i})$ a guard about the possibility of current bandwidth that can be provided by the domain Dom_i.

- $A_0 := \big((\mathsf{dom}_i \leftarrow \mathsf{dom}_i \setminus \{d\}) \wedge (\text{Cost}^{\text{Dom}_i}\text{-=}C_d)\big)$ refers to an updating actions which remove the device d from the connected devices list to the domain Dom_i and freeing up the previously allocated bandwidth C_d to this device d from the bandwidth resource of the domain Dom_i.

- $A := (\mathsf{dom}_i \leftarrow \mathsf{dom}_i \cup \{d\}) \wedge (\text{Cost}^{\text{Dom}_i}\text{+=}C_d) \wedge (\text{con\_req} = \emptyset)$ refers to an updating actions which append the device d to the connected devices list to the domain Dom_i and allocating the bandwidth C_d to the device d from the bandwidth resource of the domain Dom_i.

- $G^{'} := \forall d, d^{'} \in \text{Dev}, \exists i, j \in [1,..,k] d \in \text{Dom}_i d^{'} \in \text{Dom}_j$ a guard about checking the connectivity of the devices d, d' to some domains $\text{Dom}_i, \text{Dom}_j \in \{\,\text{Dom}_l \mid l \in [1,...,k]\,\}$.

- $A^{'} := \text{Rout\_Path} \leftarrow \text{Rout\_Path} \cup \{(d, d^{'})\}$ its an updating action about the list of path that should be created between various pairs of devices.

Further, in the given transition system figure 6.4.2 at the location l_0, by synthesis scheme it should satisfy $\Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Psi$. So it should disable some sending actions $\text{send}, c, \text{Build\_path}(d, d^{'})$ such that the controller won't build the data forwarding path between the devices d and $d^{'}$.

Note : We added only limited number of transitions in the above transition system figure 6.4.2, it has more involved transition to instruct the *SDN* controllers, for simplicity we are not mentioned all those transitions here.

For computing supervisors ensuring eq. (6.4), we can use the result of proposition 4.1 to synthesize 4 supervisors, one for each property Ψ, Φ_1, Φ_2 and Φ_3. Apart from Φ_1, Φ_2, Φ_3, which are known, the manager has to satisfy Ψ, which has to be guessed. In fact, it can be decomposed as $\Psi := \phi_{11} \wedge \phi_{12}$ these all ϕ_{11}, ϕ_{12} are considered as modular nature of the intermediate specification Ψ. This intermediate specification is guessed purely based on the knowledge we have acquired from our earlier study [Maj+21] and with a *SDN* system expertise within the *ADR* sapiens project.

$$\begin{aligned}\phi_{11} = & \big(\big(\exists d \in \text{Dev} \wedge \exists i \in \{1,..,k\},\ d \in \mathsf{dom}_i\big) \\ & \quad \implies \forall j(\neq i) \in \{1,..,k\} d \notin \mathsf{dom}_j\big) \\ \phi_{12} = & \big(\big(d_1, d_2 \in \text{Dev} \wedge\ \nexists V_i \in \text{VS}, \{d_1, d_2\} \subseteq V_i\big) \\ & \quad \implies (d_2, d_1) \notin \text{Rout\_Path}\big)\end{aligned}$$

The formula ϕ_{11} expresses that manager should not allow device d from Dev to connect to more than one domain at any given time. The formula ϕ_{12} expresses the following property : if both devices d_1, d_2 do not belong to any of the clusters of device (vspace), then there is no network path in the data plane created between d_1, d_2. In the given synthesis of the specifications $\Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Psi$, it is important to note that when the manage model satisfies Φ_1 then it can be inferred that Ψ i.e $\Phi_1 \implies \Psi$ for the given manager model, so here we have to aim for the synthesis of the specifications $\Phi_1 \wedge \Phi_2 \wedge \phi_3$. Here again, one can use the results of proposition 4.1 to compute the corresponding supervisors.

Experiments For the expressed manager model in Heptagon with three specifications $\Phi_1 \wedge \Phi_2 \wedge \Phi_3$, we got the following experimental results as shown in the table 6.2,

Parameter, Settings	Code Transformation		Reax Generation of synthesis controller			C code Generation
	Hept to Reax	Reax to Hept	Φ_1	Φ_2	Φ_3	
8 Devices, 3 domains	0.100sec	0.036sec	0.037sec	0.055sec	0.234sec	1.024sec
13 Devices, 3 domains	0.229sec	0.037sec	0.034sec	0.056sec	0.187sec	2.467sec
18 Devices, 3 domains	0.727sec	0.066sec	0.061sec	0.047sec	0.314sec	6.011sec
18 Devices, 4 domains	0.714sec	0.059sec	0.037sec	0.050sec	2.549sec	6.706sec
18 Devices, 5 domains	0.748sec	0.080sec	0.067sec	0.053sec	2.659sec	8.111sec
18 Devices, 6 domains	0.809sec	0.093sec	0.067sec	0.091sec	2.737sec	9.570sec

TABLE 6.2 : Experimental Results on the Manager Synthesis

6.4.3 Properties that has to be fulfilled by Devices

In order to satisfy the specification for each local device and the global specification Φ_4 inferred from the intermediate assumption Ψ and the *vSpace* specification Φ_1, we provide a minimal model for the device as transition system as follows,

Minimal device behaviour expressed as transition system

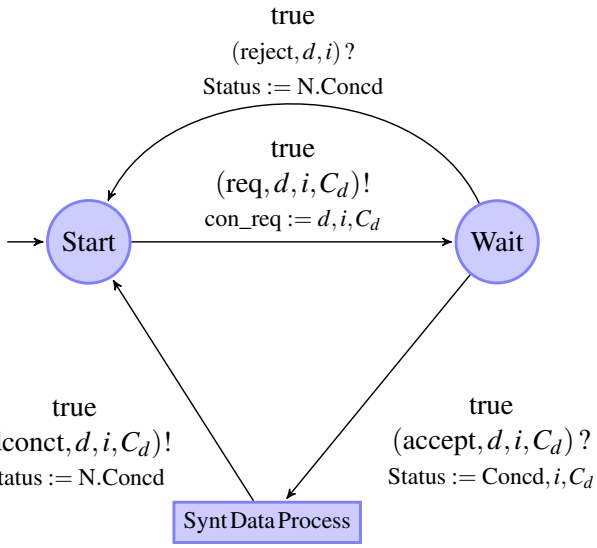


FIGURE 6.19 : Behavior of Device modeled as transition system

The various transition notation in the figure 6.4.3 refers,

- From the initial position Start, device d tries to connect to the data plane network of domain i with bandwidth C_d by sending the request message (req, d, i, C_d) ! to the manager via the open access port available in the data plane and enters in the state Wait.
- From the state Wait, it either gets the reject message $((\text{reject}, d, i)\,?)$ and goes back to the initial state Start or accept message $((\text{accept}, d, i, C_d)\,?)$ and enters the synthesis data process node function Synt Data Process, from this node, it exchanges the data between various devices via the data plane and authorize the various devices to send the *sensitive* data.
- From the state Synt Data Process, the device can send the disconnect messages $((\text{dconct}, d, i, C_d))$ and enters into the initial Start state.

From the previous intermediate property we have to find a supervisor such that $\langle \Psi \wedge \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \rangle \parallel_{d \in \text{Dev}} \mathscr{A}_d^{\mathscr{S}_d} \langle \Phi_4 \rangle$.

Further each device has to fulfill his own property which is not related to the global property. Such a local property can be added as an additional property for the device. Indeed, when a device is a server (e.g. in a data center) $d \in$ Dev, it has partition of data sets say *sensitive* data (referred as $\text{Data}^d_{\text{sens}}$ in the specification) and *primitive* data (referred as $\text{Data}^d_{\text{prim}}$ in the

specification). Such a device can share the primitive data to other devices belonging to a common vspace. For sensitive data, it however requires some authentication mechanism with further conditions on the devices who request sensitive data. This can be formally expressed as

$$\phi_{21}^{d} = \left(\text{req}_{\text{data}}^{d'} \in \text{Data}_{\text{prim}}^{d}\right) \implies \text{send}_{d,d'}(\text{data})$$

$$\phi_{22}^{d} = \begin{array}{l}\left(\text{req}_{\text{data}}^{d'} \in \text{Data}_{\text{sens}}^{d} \wedge (\text{Sign}_{d'}, d) \notin \text{Authorize}_{d}\right) \\ \implies \neg\text{send}_{d,d'}(\text{data})\end{array}$$

ϕ_{21}^{d} indicates that a device d will send primitive data to device $d^{'}$, if the device d' requests a data from the primitive data set. ϕ_{22}^{d} means that, if a device $d^{'}$ requests a sensitive data, then device d will check the signature of device $d^{'}$ and verify whether it is authorized or not. $\mathscr{A}_{d}^{\mathscr{S}_{d}^{'}} \vDash \phi_{21}^{d} \wedge \phi_{22}^{d}$ can be computed according to proposition 4.1, where as proposition 4.2 ensures us the final result on the set of devices.

6.5 Chapter Conclusion

In the last section, we provided minimal description about the *SDN* layered architecture and possibility of using the compositional synthesis technique to model and synthesize constraints of bandwidth, priority, and *vSpace* (on providing the data path between various devices) specifications. These specifications are there for illustrative purposes, we can introduce various specific applications in an incremental way by expressing associated specifications and models. In this way one can automate the generation of synthesis controller and implementable code in an efficient way by using compositional and modular synthesis techniques.

This control synthesis technique implementation using BZR (Heptagon with ReaX controller) will be suited more for modeling and synthesizing the *IoT* devices. In the last two sections, we showed experiments in SDN-IoT typical application. In those experiments we increased the number of domains, devices and showed the experimental stats (performance in terms of computation time). The statistics are fairly low (both run time and memory usage), we could increase the number of devices and domains. It is also fairly simple to do by writing an automation script (either by writing *Makefile* or *Shell* script) to generate the generic high level model ; it does not affect computation statistics much. But the real issue will come in the length (number of lines) of generated *C* code hence the high compilation time statistics (imagine if the number of devices are 100). Generally speaking, using various data structure concepts, one can develop a fairly simple and efficient program in *C* to realize manager automation in this aspect. But unfortunately as of now, the *BZR* generated *C* code does not utilize various data structure concepts of *C* programming efficiently. At the same time, when the model does not have to worry about the parameters like number of devices or domains, but only has to capture

the behaviour of certain *IoT* devices, servers, etc, then *BZR* scheme is a real advantage. By capturing the complex behaviour pattern by writing the respective behaviour as an automaton and expressing the requirements as specification (input output contract) and using reaX, one can produce control synthesis fairly efficiently and can integrate to *IoT* device in real time.

In this synthesis chapter, we used both modular and compositional synthesis techniques for the possibility of producing an efficient synthesis controller for various specifications in real-time for the designed abstract model and integrating the same to the *SDN* real-time platform. The advantage of doing a synthesis experiment with reactive synthesis tool *BZR* is its ability to model the system behavior in an abstract high level automaton, and produce the control synthesis in real time. Using the model and produced controllers for the various specifications, one can translate them into high level *C* and *Java* programming languages, this produced code can be directly merged with the working *SDN* system. Unlike the model checking, here produced synthesize controllers using *ReaX* tool and converting into high level language are more efficient because of the high level model specification rather than detailed low level model specification and/or the tool's efficient coding scheme. In any case both verification in the last chapter and synthesis in this chapter are in the realm of *ADR* sapiens project theme.

CONCLUSION

> It is through science that we prove, but through intuition that we discover.
>
> Henri Poincaré

In the last century, humankind benefited enormously from automation. Either a medical, outer space program, large scale manufacturing, transport, agriculture and so on, there is nothing that will work as efficiently and precisely as of now without the help of automation.

In the *SDN* architecture, the management plane has to integrate various network operations via network controllers (i.e *SDN* Controllers) and to deploy the rules accordingly into network elements (i.e data plane switches) to finally provide the service to network users (i.e represented by their devices). The network management has various optimization constraints such as resource allocation in case of sharing common resources, keeping the data plane under safe bandwidth when quality of service is to be guaranteed, and various heterogeneous applications oriented constraints. On top of such requirements, the management plane has to satisfy the policies that should be handled precisely and efficiently. Due to increased network complexity, the trend is to implement automation processes at the management plane. Clearly, this is envisioned by various network operators and standard bodies such as the Internet Engineering Task Force (*IETF*), the European Telecommunications Standards Institute (*ETSI*), and so on.

Within network management automation efforts, the *IoT* applications appear to take a good part of the benefits. As we saw in chapter 2, the growth of *IoT* applications are enormous. As of now, various standards bodies (like *IETF*, *ETSI*, etc) have foreseen numerous key features and applications for the *IoT* frameworks. Such a list can not be really exhaustive since the application of such *IoT* devices will be limited only by our vision. There will always be room for further innovation and improvement. One can come up with new developments and schemes, for example developing *IoT* application based weather forecast (alert), or an automated feedback sensor-actuator for the agricultural industry, etc. Clearly there is still a lot of development in the *IoT* devices design, technology and innovations in terms of envisioning the application from the *IoT* devices.

To solve certain complex automation problems, in recent years, the interest in machine learning and artificial intelligence took a huge spike among general society after the success of *AlphaGo* which defeated the South Korean Go professional. Machine learning/Artificial Intelligence is an active and well-funded research field in academics (existed way back to the date of development of programming languages like *LISP*, *C* program).

Furthermore, the encoding scheme of deep learning neural network models started to perform better than expert level encoding scheme, for instance the encoding of image (digital photograph) by deep neural network compared to the JPEG encoding scheme [Gue+18]. The revolution of deep learning network is very much related to the technology advancement in the computing power. The interest of using deep learning also intruded into the networking community, especially in the optimization problem of resource allocations [Zie+19 ; Yrj20] across the various layers of networking. In fact, advanced 5G and 6G will intensively use AI/ML to automatise the network management with close-loops [KKC20] [1]. The ultimate automation target is for the autonomous operation driven by high-level policies and rules, enabling self-configuration, self-monitoring, self-healing and self-optimization – without further human intervention in the network operation and management.

> If there exists an *automated procedure* to perform a certain task.
> Who will prove the working correctness of this automated procedure ?

> It is easy to make mistake, but very hard
> to know something is absolutely true.
>
> Richard Feynman

Within certain use cases (e.g. mission-critical), it is mandatory to exhaustively test the autonomous management of networking applications like *SDN-IoT* platform especially when it is driven by ML/AI algorithms, because the latter, as of now, are highly driven by statistical inference and approximate model generation from the provided data set. They must therefore be validated before being deployed into the real situations so that they are thrust worthy with regards to millions of network communication service consumers.

Does the model checking (verification) or in general Formal methods provide the mechanism to validate the ML/AI algorithms ? with respect to the safety measures as a specification. Are the Synthesis controllers able to control the ML/AI algorithms to work in safe boundaries ? These are natural questions that can be posted from the Formal methods communities which intend to work towards the real life applications. There are numerous works which are looking for combining the power of both statistical and formal methods such as the field of neural symbolic AI.

While thinking of using formal methods and synthesis of supervisor to certify the ML/AI model correctness and limits the behaviour to work within safe boundaries, one might develop a set of constraints as a specification (input-output contract) and automate the validation of ML/AI model by creating possible inputs satisfying the specification and verify that corresponding ML/AI model output satisfying the input-output contract. Similarly, to avoid unsafe

1. https ://www.etsi.org/newsroom/blogs/entry/a-recent-study-at-etsi-zsm-addresses-potential-security-threats-to-zero-touch-network-and-service-automation

output behaviours (violating the specification), one might use back propagation (or synthesis the data) extensively and generate the possible unsafe inputs of the ML/AI algorithms. In this case, one might automatise in generating unsafe output behaviours from expressed input-output contract specification and in using back propagation techniques of ML/AI model to enumerate the possible inputs behaviours to understand the MI/AI driven automated schemes.

Formal procedures provide a great deal of what is the core working procedure of the designed system. At the end of this model creation and expressed specifications, there are comprehensive details about the system and the working nature of the proposed system.

Within formal methods ecosystem, the model checking process will verify the system model with respect to the model specification, and makes us be careful and aware about the kind of safety properties we should expect from the system. If we are not having the great understanding about the system model and the kind of hardware under which the written system models, then the automation process we propose are not to be trustworthy. Here, we should emphasize on the importance of formal method and the extremely carefulness in modeling the system and in analysing the system correctness with respect to the safety specifications (safety requirements).

Of course, formal methods alone are not enough to evaluate how good the designed system is and as well as its real-time performance. In order to evaluate the real-time performance we can use other theoretical computer science tools like computation complexity (time and memory), communication complexity, information theory for the quality of encoding procedures involved in the system design or protocol.

BIBLIOGRAPHIE

[Alu99] Rajeev ALUR, “Timed Automata”, *in* : *Computer Aided Verification*, sous la dir. de Nicolas HALBWACHS et Doron PELED, Berlin, Heidelberg : Springer Berlin Heidelberg, 1999, p. 8-22, ISBN : 978-3-540-48683-1.

[AW04] Hagit ATTIYA et Jennifer WELCH, *Distributed Computing : Fundamentals, Simulations and Advanced Topics*, Hoboken, NJ, USA : John Wiley & Sons, Inc., 2004, ISBN : 0471453242.

[Bad+18] Michael BADDELEY, Reza NEJABATI, George OIKONOMOU, Mahesh SOORIYABANDARA et Dimitra SIMEONIDOU, “Evolving SDN for Low-Power IoT Networks”, *in* : *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, p. 71-79, DOI : 10.1109/NETSOFT.2018.8460125.

[Bal+14] Thomas BALL, Nikolaj BJØRNER, Aaron GEMBER, Shachar ITZHAKY, Aleksandr KARBYSHEV, Mooly SAGIV, Michael SCHAPIRA et Asaf VALADARSKY, “VeriCon : towards verifying controller programs in software-defined networks”, *in* : *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, p. 282-293, DOI : 10.1145/2594291.2594317, URL : https://doi.org/10.1145/2594291.2594317.

[BCC98] Sergey BEREZIN, Sérgio CAMPOS et Edmund M. CLARKE, “Compositional Reasoning in Model Checking”, *in* : *Compositionality : The Significant Difference*, sous la dir. de Willem-Paul de ROEVER, Hans LANGMAACK et Amir PNUELI, Berlin, Heidelberg : Springer Berlin Heidelberg, 1998, p. 81-102, ISBN : 978-3-540-49213-9.

[BDB16] D. T. BUI, R. DOUVILLE et M. BOUSSARD, “Supporting multicast and broadcast traffic for groups of connected devices”, *in* : *2016 IEEE NetSoft Conference and Workshops*, 2016, p. 48-52.

[Ber+17] N. BERTHIER, F. ALVARES°, H. MARCHAND, G. DELAVAL et É. RUTTEN, “Logico-numerical control for software components reconfiguration”, *in* : *2017 IEEE Conference on Control Technology and Applications (CCTA)*, 2017, p. 1599-1606.

[BFT16] Clark BARRETT, Pascal FONTAINE et Cesare TINELLI, *The Satisfiability Modulo Theories Library (SMT-LIB)*, www.SMT-LIB.org, 2016.

[Bin+10] B. BINGHAM, J. BINGHAM, F. M. de PAULA, J. ERICKSON, G. SINGH et M. REITBLATT, "Industrial Strength Distributed Explicit State Model Checking", *in* : *2010 Ninth Int. Workshop on Parallel and Distributed Methods in Verification, and Second Int. Workshop on High Performance Computational Systems Biology*, USA, 2010, p. 28-36.

[BK08] Christel BAIER et Joost-Pieter KATOEN, *Principles of Model Checking (Representation and Mind Series)*, The MIT Press, 2008, ISBN : 026202649X.

[BM15] N. BERTHIER et H. MARCHAND, "Deadlock-free Discrete Controller Synthesis for Infinite State Systems", *in* : *54th IEEE Conference on Decision and Control*, Osaka, Japan, déc. 2015, p. 1000-1007.

[Bol+14] Raffaele BOLLA, Chiara LOMBARDO, Roberto BRUSCHI et Sergio MANGIALARDI, "DROPv2 : energy efficiency through network function virtualization", *in* : *IEEE Network* 28.*2* (2014), p. 26-32, DOI : 10.1109/MNET.2014.6786610.

[Bou+15] M. BOUSSARD, D. T. BUI, L. CIAVAGLIA, R. DOUVILLE, M. L. PALLEC, N. L. SAUZE, L. NOIRIE, S. PAPILLON, P. PELOSO et F. SANTORO, "Software-Defined LANs for Interconnected Smart Environment", *in* : *2015 27th International Teletraffic Congress*, 2015, p. 219-227.

[Bou+18] Mathieu BOUSSARD, Dinh Thai BUI, Richard DOUVILLE, Pascal JUSTEN, Nicolas Le SAUZE, Pierre PELOSO, Frederik VANDEPUTTE et Vincent VERDOT, "Future Spaces : Reinventing the Home Network for Better Security and Automation in the IoT Era", *in* : *Sensors* 18.*9* (2018), p. 2986, DOI : 10.3390/s18092986, URL : https://doi.org/10.3390/s18092986.

[BT18] Clark BARRETT et Cesare TINELLI, "Satisfiability Modulo Theories", *in* : *Handbook of Model Checking*, sous la dir. d'Edmund M. CLARKE, Thomas A. HENZINGER, Helmut VEITH et Roderick BLOEM, Cham : Springer International Publishing, 2018, p. 305-343, ISBN : 978-3-319-10575-8, DOI : 10.1007/978-3-319-10575-8_11, URL : https://doi.org/10.1007/978-3-319-10575-8_11.

[Can+12] Marco CANINI, Daniele VENZANO, Peter PEREŠINI, Dejan KOSTIĆ et Jennifer REXFORD, "A NICE Way to Test Openflow Applications", *in* : *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, San Jose, CA : USENIX Association, 2012, p. 10.

[Cav+15] Ana CAVALCANTI, Wen-ling HUANG, Jan PELESKA et Jim WOODCOCK, "CSP and Kripke Structures", *in* : *Theoretical Aspects of Computing - ICTAC 2015*, sous la dir. de Martin LEUCKER, Camilo RUEDA et Frank D. VALENCIA, Cham : Springer International Publishing, 2015, p. 505-523, ISBN : 978-3-319-25150-9.

[CC77] Patrick COUSOT et Radhia COUSOT, “Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”, *in* : *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL ’77, Los Angeles, California : Association for Computing Machinery, 1977, p. 238-252, ISBN : 9781450373500, DOI : 10.1145/512950.512973, URL : https://doi.org/10.1145/512950.512973.

[Cim+02] Alessandro CIMATTI, Edmund M. CLARKE, Enrico GIUNCHIGLIA, Fausto GIUNCHIGLIA, Marco PISTORE, Marco ROVERI, Roberto SEBASTIANI et Armando TACCHELLA, “NuSMV 2 : An OpenSource Tool for Symbolic Model Checking”, *in* : *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV ’02, Berlin, Heidelberg : Springer-Verlag, 2002, p. 359-364, ISBN : 3540439978.

[CL02] Miguel CASTRO et Barbara LISKOV, “Practical Byzantine Fault Tolerance and Proactive Recovery”, *in* : 20.*4* (2002), ISSN : 0734-2071, DOI : 10.1145/571637.571640, URL : https://doi.org/10.1145/571637.571640.

[CL08] C. CASSANDRAS et S. LAFORTUNE, *Introduction to Discrete Event Systems*, Kluwer Academic Publishers, 2008.

[Cla+18] Edmund M CLARKE, Thomas A HENZINGER, Helmut VEITH et Roderick BLOEM, *Handbook of model checking*, t. 10, Springer, 2018.

[Cla08] Edmund M. CLARKE, “The Birth of Model Checking”, *in* : *25 Years of Model Checking : History, Achievements, Perspectives*, sous la dir. d’Orna GRUMBERG et Helmut VEITH, Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 1-26, ISBN : 978-3-540-69850-0, DOI : 10.1007/978-3-540-69850-0_1, URL : https://doi.org/10.1007/978-3-540-69850-0_1.

[CLM89] E. M. CLARKE, D. E. LONG et K. L. MCMILLAN, “Compositional model checking”, *in* : *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, IEEE, 1989, p. 353-362.

[Cvi+13] N. CVIJETIC, M. ANGELOU, A. PATEL, PHILIP NAN JI et TING WANG, “Defining optical software-defined networks (SDN) : From a compilation of demos to network model synthesis”, *in* : *2013 Optical Fiber Communication Conference and Exposition and the National Fiber Optic Engineers Conference (OFC/NFOEC)*, 2013, p. 1-3, DOI : 10.1364/OFC.2013.OTh1H.1.

[CWL18a] Haoxian CHEN, Anduo WANG et Boon Thau LOO, "Towards Example-Guided Network Synthesis", *in* : *Proceedings of the 2nd Asia-Pacific Workshop on Networking*, APNet '18, Beijing, China : Association for Computing Machinery, 2018, p. 65-71, ISBN : 9781450363952, DOI : 10.1145/3232565.3234462.

[CWL18b] Haoxian CHEN, Anduo WANG et Boon Thau LOO, "Towards Example-Guided Network Synthesis", *in* : *Proceedings of the 2nd Asia-Pacific Workshop on Networking*, APNet '18, Beijing, China : Association for Computing Machinery, 2018, p. 65-71, ISBN : 9781450363952, DOI : 10.1145/3232565.3234462, URL : https://doi.org/10.1145/3232565.3234462.

[DRM13] Gwenaël DELAVAL, Éric RUTTEN et Hervé MARCHAND, "Integrating discrete controller synthesis into a reactive programming language compiler", *in* : *Discrete Event Dynamic Systems* 23.*4* (2013), p. 385-418.

[El-+16] Ahmed EL-HASSANY, Jeremie MISEREZ, Pavol BIELIK, Laurent VANBEVER et Martin VECHEV, "SDNRacer : Concurrency Analysis for Software-Defined Networks", *in* : *SIGPLAN Not.* 51.*6* (juin 2016), p. 402-415, ISSN : 0362-1340, DOI : 10.1145/2980983.2908124, URL : https://doi.org/10.1145/2980983.2908124.

[El-+17] Ahmed EL-HASSANY, Petar TSANKOV, Laurent VANBEVER et Martin VECHEV, "Network-Wide Configuration Synthesis", *in* : *Computer Aided Verification*, sous la dir. de Rupak MAJUMDAR et Viktor KUNČAK, Cham : Springer International Publishing, 2017, p. 261-281, ISBN : 978-3-319-63390-9.

[Fin+19] Bernd FINKBEINER, Manuel GIESEKING, Jesko HECKING-HARBUSCH et Ernst-Rüdiger OLDEROG, "Model Checking Data Flows in Concurrent Network Updates", *in* : *Automated Technology for Verification and Analysis*, sous la dir. d'Yu-Fang CHEN, Chih-Hong CHENG et Javier ESPARZA, Cham : Springer International Publishing, 2019, p. 515-533, ISBN : 978-3-030-31784-3.

[Fou14] Open Networking FOUNDATION, *OpenFlow Switch Specification - Version 1.5.0*, 2014, URL : https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.0.pdf.

[Fou16a] Linux FOUNDATION, *Open vSwitch*, 2016, URL : https://www.openvswitch.org/.

[Fou16b] Linux FOUNDATION, *Open vSwitch Advanced Features*, 2016, URL : http://docs.openvswitch.org/en/latest/tutorials/ovs-advanced/.

[FSV19] K. FOERSTER, S. SCHMID et S. VISSICCHIO, "Survey of Consistent Software-Defined Network Updates", *in* : *IEEE Communications Surveys Tutorials* 21 (2019), p. 1435-1461, DOI : 10.1109/COMST.2018.2876749.

[GL02] Seth GILBERT et Nancy LYNCH, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”, *in* : *SIGACT News* 33.*2* (juin 2002), p. 51-59, ISSN : 0163-5700, DOI : 10.1145/564585.564601, URL : https://doi.org/10.1145/564585.564601.

[GM04] Benoit GAUDIN et Hervé MARCHAND, “Supervisory control of product and hierarchical discrete event systems”, *in* : *European Journal of Control* 10.*2* (2004), p. 131-145.

[GM05] Benoit GAUDIN et Hervé MARCHAND, “EFFICIENT COMPUTATION OF SUPERVISORS FOR LOOSELY SYNCHRONOUS DISCRETE EVENT SYSTEMS : A STATE-BASED APPROACH”, *in* : *IFAC Proceedings Volumes* 38.*1* (2005), 16th IFAC World Congress, p. 145-150, ISSN : 1474-6670, DOI : https://doi.org/10.3182/20050703-6-CZ-1902.00309, URL : https://www.sciencedirect.com/science/article/pii/S1474667016363212.

[Goo+21] Martijn GOORDEN, Joanna VAN DE MORTEL-FRONCZAK, Michel RENIERS, Martin FABIAN, Wan FOKKINK et Jacobus ROODA, “Model properties for efficient synthesis of nonblocking modular supervisors”, *in* : *Control Engineering Practice* 112 (2021), p. 104830, ISSN : 0967-0661, DOI : https://doi.org/10.1016/j.conengprac.2021.104830, URL : https://www.sciencedirect.com/science/article/pii/S0967066121001076.

[Gue+18] Lionel GUEGUEN, Alex SERGEEV, Ben KADLEC, Rosanne LIU et Jason YOSINSKI, “Faster Neural Networks Straight from JPEG”, *in* : *Advances in Neural Information Processing Systems*, sous la dir. de S. BENGIO, H. WALLACH, H. LAROCHELLE, K. GRAUMAN, N. CESA-BIANCHI et R. GARNETT, t. 31, Curran Associates, Inc., 2018, URL : https://proceedings.neurips.cc/paper/2018/file/7af6266cc52234b5aa339b16695f7fc4-Paper.pdf.

[HHK20] Shimaa A Abdel HAKEEM, Anar A HADY et HyungWon KIM, “5G-V2X : Standardization, architecture, use cases, network-slicing, and edge-computing”, *in* : *Wireless Networks* 26.*8* (2020), p. 6015-6041.

[Hol04] Gerard J HOLZMANN, *The SPIN model checker : Primer and reference manual*, t. 1003, Addison-Wesley Reading, 2004.

[Hol97] G. J. HOLZMANN, “The model checker SPIN”, *in* : *IEEE Transactions on Software Engineering* 23 (1997), p. 279-295, DOI : 10.1109/32.588521.

[HQR98] Thomas A. HENZINGER, Shaz QADEER et Sriram K. RAJAMANI, “You assume, we guarantee : Methodology and case studies”, *in* : *Computer Aided Verification*, sous la dir. d’Alan J. HU et Moshe Y. VARDI, Berlin, Heidelberg : Springer Berlin Heidelberg, 1998, p. 440-451, ISBN : 978-3-540-69339-0.

[JRG21] A. H. M. JAKARIA, Mohammad Ashiqur RAHMAN et Aniruddha GOKHALE, “Resiliency-Aware Deployment of SDN in Smart Grid SCADA : A Formal Synthesis Model”, *in* : *IEEE Transactions on Network and Service Management* 18.*2* (2021), p. 1430-1444, DOI : 10.1109/TNSM.2021.3050148.

[Kal+12] Gabriel KALYON, Tristan GALL, Hervé MARCHAND et Thierry MASSART, “Symbolic Supervisory Control of Infinite Transition Systems Under Partial Observation Using Abstract Interpretation”, *in* : *Discrete Event Dynamic Systems* 22.*2* (2012), p. 121-161, ISSN : 0924-6703, DOI : 10.1007/s10626-011-0101-3.

[Kal+14] G. KALYON, T. LE GALL, H. MARCHAND et T. MASSART, “Symbolic Supervisory Control of Distributed Systems with Communications”, *in* : *IEEE Transaction on Automatic Control* 59.*2* (fév. 2014), p. 396-408.

[Khu+13] Ahmed KHURSHID, Xuan ZOU, Wenxuan ZHOU, Matthew CAESAR et Philip Brighten GODFREY, “VeriFlow : Verifying Network-Wide Invariants in Real Time”, *in* : *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, USENIX Association, 2013, p. 15-27, URL : https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid.

[KKC20] Enis KARAARSLAN, Eren KARABACAK et Cihat CETINKAYA, “Design and Implementation of SDN-Based Secure Architecture for IoT-Lab”, *in* : *Artificial Intelligence and Applied Mathematics in Engineering Problems*, sous la dir. de D. Jude HEMANTH et Utku KOSE, Cham : Springer International Publishing, 2020, p. 877-885, ISBN : 978-3-030-36178-5.

[Kre+15] D. KREUTZ, F. M. V. RAMOS, P. E. VERÍSSIMO, C. E. ROTHENBERG, S. AZODOLMOLKY et S. UHLIG, “Software-Defined Networking : A Comprehensive Survey”, *in* : *Proceedings of the IEEE* 103.*1* (2015), p. 14-76, ISSN : 1558-2256, DOI : 10.1109/JPROC.2014.2371999.

[KVM12] Peyman KAZEMIAN, George VARGHESE et Nick MCKEOWN, “Header Space Analysis : Static Checking for Networks”, *in* : *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA : USENIX Association, avr. 2012, p. 113-126, URL : https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

[LJJ06] Tristan LE GALL, Bertrand JEANNET et Thierry JÉRON, “Verification of Communication Protocols Using Abstract Interpretation of FIFO Queues”, *in* : *Algebraic Methodology and Software Technology*, sous la dir. de Michael JOHNSON et

Varmo VENE, Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, p. 204-219, ISBN : 978-3-540-35636-3.

[LKR14] Adrian LARA, Anisha KOLASANI et Byrav RAMAMURTHY, "Network Innovation using OpenFlow : A Survey", *in* : *IEEE Communications Surveys Tutorials* 16.*1* (2014), p. 493-512, DOI : 10.1109/SURV.2013.081313.00105.

[Lyn96] Nancy A LYNCH, *Distributed algorithms*, Elsevier, 1996.

[Mai+11] Haohui MAI, Ahmed KHURSHID, Rachit AGARWAL, Matthew CAESAR, P. Brighten GODFREY et Samuel Talmadge KING, "Debugging the Data Plane with Anteater", *in* : *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, Toronto, Ontario, Canada : Association for Computing Machinery, 2011, p. 290-301, ISBN : 9781450307970, DOI : 10.1145/2018436.2018470, URL : https://doi.org/10.1145/2018436.2018470.

[Maj+21] A. MAJITH, O. SANKUR, H. MARCHAND et D. T. BUI, "Compositional model checking of SDN platform", *in* : *17th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2021.

[McC+15a] J. MCCLURG, H. HOJJAT, P. ČERNÝ et N. FOSTER, "Efficient Synthesis of Network Updates", *in* : *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland, OR, USA : Association for Computing Machinery, 2015, p. 196-207, ISBN : 9781450334686.

[McC+15b] Jedidiah MCCLURG, Hossein HOJJAT, Pavol ČERNÝ et Nate FOSTER, "Efficient Synthesis of Network Updates", *in* : *SIGPLAN Not.* 50.*6* (juin 2015), p. 196-207, ISSN : 0362-1340, DOI : 10.1145/2813885.2737980, URL : https://doi.org/10.1145/2813885.2737980.

[McC18] Jedidiah MCCLURG, "Program synthesis for software-defined networking", thèse de doct., PhD. thesis, University of Colorado Boulder, USA, 2018.

[MDW14a] R. MAJUMDAR, S. DEEP TETALI et Z. WANG, "Kuai : A model checker for software-defined networks", *in* : *2014 Formal Methods in Computer-Aided Design (FMCAD)*, oct. 2014, p. 163-170, DOI : 10.1109/FMCAD.2014.6987609.

[MDW14b] R. MAJUMDAR, S. DEEP TETALI et Z. WANG, "Kuai : A model checker for software-defined networks", *in* : *2014 Formal Methods in Computer-Aided Design (FMCAD)*, oct. 2014, p. 163-170, DOI : 10.1109/FMCAD.2014.6987609.

[MW08] Anca MUSCHOLL et Igor WALUKIEWICZ, “A lower bound on web services composition”, *in* : *Logical Methods in Computer Science* Volume 4, Issue 2 (mai 2008), DOI : 10.2168/LMCS-4(2:5)2008, URL : https://lmcs.episciences.org/824.

[Nel+13] Tim NELSON, Arjun GUHA, Daniel J. DOUGHERTY, Kathi FISLER et Shriram KRISHNAMURTHI, “A Balance of Power : Expressive, Analyzable Controller Programming”, *in* : *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN ’13, Hong Kong, China : Association for Computing Machinery, 2013, p. 79-84, ISBN : 9781450321785, DOI : 10.1145/2491185.2491201, URL : https://doi.org/10.1145/2491185.2491201.

[OC99] Ming OUYANG et Vasek CHVATAL, “Implementations of the Dpll Algorithm”, AAI9947888, thèse de doct., USA, 1999, ISBN : 0599501200.

[PBB17] P. PELOSO, D. T. BUI et M. BOUSSARD, “Enforcing users’ constraints in dynamic, software-defined networks of devices”, *in* : *2017 19th Asia-Pacific Network Operations and Management Symposium*, 2017, p. 106-111.

[Plo+16] Gordon D. PLOTKIN, Nikolaj BJØRNER, Nuno P. LOPES, Andrey RYBALCHENKO et George VARGHESE, “Scaling Network Verification Using Symmetry and Surgery”, *in* : *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’16, St. Petersburg, FL, USA : Association for Computing Machinery, 2016, p. 69-83, ISBN : 9781450335492, DOI : 10.1145/2837614.2837657, URL : https://doi.org/10.1145/2837614.2837657.

[Pnu77] A. PNUELI, “The Temporal Logic of Programs”, *in* : *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS ’77, 1977, p. 46-57.

[Poz+15] Francisco POZO, Wilfried STEINER, Guillermo RODRIGUEZ-NAVAS et Hans HANSSON, “A decomposition approach for SMT-based schedule synthesis for time-triggered networks”, *in* : *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, 2015, p. 1-8, DOI : 10.1109/ETFA.2015.7301436.

[PP91] Wuxu PENG et S. PUROSHOTHAMAN, “Data Flow Analysis of Communicating Finite State Machines”, *in* : *ACM Trans. Program. Lang. Syst.* 13.*3* (juill. 1991), p. 399-442, ISSN : 0164-0925, DOI : 10.1145/117009.117015, URL : https://doi.org/10.1145/117009.117015.

[PW97] Doron PELED et Thomas WILKE, “Stutter-Invariant Temporal Properties Are Expressible without the next-Time Operator”, *in* : *Inf. Process. Lett.* 63.*5* (sept. 1997), p. 243-246, ISSN : 0020-0190, DOI : 10.1016/S0020-0190(97)00133-6, URL : https://doi.org/10.1016/S0020-0190(97)00133-6.

[Rab+21] Jorge RABADAN, Kiran NAGARAJ, Wen LIN et Ali SAJASSI, *EVPN Multi-Homing Extensions for Split Horizon Filtering*, Internet-Draft draft-ietf-bess-evpn-mh-split-horizon-02, Work in Progress, Internet Engineering Task Force, oct. 2021, 15 p., URL : https://datatracker.ietf.org/doc/html/draft-ietf-bess-evpn-mh-split-horizon-02.

[Rei+12] Mark REITBLATT, Nate FOSTER, Jennifer REXFORD, Cole SCHLESINGER et David WALKER, “Abstractions for network update”, *in* : *ACM SIGCOMM 2012 Conference, SIGCOMM ’12, Helsinki, Finland - August 13 - 17, 2012*, 2012, p. 323-334, DOI : 10.1145/2342356.2342427, URL : https://doi.org/10.1145/2342356.2342427.

[Sar+14] Chayan SARKAR, S. N. Akshay Uttama NAMBI, R. Venkatesha PRASAD et Abdur RAHIM, “A scalable distributed architecture towards unifying IoT applications”, *in* : *2014 IEEE World Forum on Internet of Things (WF-IoT)*, 2014, p. 508-513, DOI : 10.1109/WF-IoT.2014.6803220.

[SNM13] D. SETHI, S. NARAYANA et S. MALIK, “Abstractions for model checking SDN controllers”, *in* : *Formal Methods in Computer-Aided Design*, 2013, p. 145-148.

[ST16] Maarten van STEEN et Andrew S. TANENBAUM, “A brief introduction to distributed systems”, *in* : *Computing* 98.*10* (2016), p. 967-1009, DOI : 10.1007/s00607-016-0508-7, URL : https://doi.org/10.1007/s00607-016-0508-7.

[Sta85] Eugene W. STARK, “A Proof Technique for Rely/Guarantee Properties”, *in* : *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings*, 1985, p. 369-391, DOI : 10.1007/3-540-16042-6_21, URL : https://doi.org/10.1007/3-540-16042-6%5C_21.

[Tar55] A. TARSKI, “A Lattice-theoretical Fixpoint Theorem and its applications”, *in* : *Pacific Journal of Mathematics* 5 (1955), p. 285-309.

[Var05] “Front matter”, *in* : *Network Algorithmics*, sous la dir. de George VARGHESE, The Morgan Kaufmann Series in Networking, San Francisco : Morgan Kaufmann, 2005, p. i-iii, DOI : https://doi.org/10.1016/B978-0-12-

088477-3.50026-6, URL : https://www.sciencedirect.com/science/article/pii/B9780120884773500266.

[Wan+13] Anduo WANG, Salar MOARREF, Boon Thau LOO, Ufuk TOPCU et Andre SCEDROV, "Automated synthesis of reactive controllers for software-defined networks", *in* : *2013 21st IEEE International Conference on Network Protocols, ICNP 2013, Göttingen, Germany, October 7-10, 2013*, 2013, p. 1-6, DOI : 10.1109/ICNP.2013.6733666, URL : https://doi.org/10.1109/ICNP.2013.6733666.

[WH91] Y. WILLNER et M. HEYMANN, "Supervisory control of concurrent discrete-event systems", *in* : *International Journal of Control* 54.5 (1991), p. 1143-1169.

[Wig17] Avi WIGDERSON, "Mathematics and Computation", *in* : *Princeton University Press* (2017).

[WR88] W. M. WONHAM et P. J. RAMADGE, "Modular Supervisory Control of Discrete Event Systems", *in* : *Mathematics of Control Signals and Systems* 1 (1988), p. 13-30.

[Yrj20] Seppo YRJOLA, "Technology antecedents of the platform-based ecosystemic business models beyond 5G", *in* : *2020 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, IEEE, 2020, p. 1-8.

[Yua+15] Yifei YUAN, Dong LIN, Rajeev ALUR et Boon Thau LOO, "Scenario-Based Programming for SDN Policies", *in* : *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, Heidelberg, Germany : Association for Computing Machinery, 2015, ISBN : 9781450334129, DOI : 10.1145/2716281.2836119, URL : https://doi.org/10.1145/2716281.2836119.

[Zie+19] Volker ZIEGLER, Thorsten WILD, Mikko UUSITALO, Hannu FLINCK, Vilho RÄISÄNEN et Kimmo HÄTÖNEN, "Stratification of 5G evolution and Beyond 5G", *in* : *2019 IEEE 2nd 5G World Forum (5GWF)*, IEEE, 2019, p. 329-334.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Vérification et synthèse automatisées de systèmes distribués

Mot clés : *SDN*, *IoT* platform, Control Synthesis, Verification, Automation

Résumé : Dans le cadre de ce document, nous nous sommes intéressés à l'automatisation des plates-formes *IoT*. Plus précisément, nous utilisons des techniques d'analyse et de synthèse formelles pour garantir la sûreté de fonctionnement du comportement de ces plateformes. Le réseau défini par logiciel (*SDN*) consiste en une mise en réseau flexible, et à faible coût, des différents composants et fournit des applications de systèmes distribués dynamiques dont les plateformes *IoT* font partie. Il existe un fort besoin d'intégration cohérente et correcte de l'application *IoT* dans l'environnement *SDN*, ce qui induit une utilisation de méthodes de vérification formelles pour analyser la sécurité d'un environnement *SDN*-*IoT*. Nous fournissons également un cadre de synthèse détaillé pour modéliser le comportement abstrait de haut niveau des composants *IoT* et générer automatiquement le code d'implémentation de bas niveau à intégrer dans ceux-ci en se basant sur une approche compositionnelle.

Title: Automated Verification and Synthesis of Distributed Systems

Keywords: *SDN*, *IoT* platform, Control Synthesis, Verification, Automation

Abstract: Towards the automation of Software-Defined Network (*SDN*) based Internet of Things (*IoT*) platforms, we are using formal analysis and synthesis techniques to ensure their safe behaviours. *SDN* a flexible and low cost networking principle which provides dynamic distributed system applications, *IoT* is one such kind. There is a strong need for consistent and correct integration of *IoT* applications in *SDN* environment. Using formal compositional verification methods for analyzing the safety of an *SDN*-*IoT* environment, we provide a detailed synthesis framework to model the abstract behaviour of *IoT* devices, *SDN* manager and automatically generate low level implementation code to be integrated with *IoT* applications.