An incremental approach for hardware discrete controller synthesis

Mingming Ren

▶ **To cite this version:**

Mingming Ren. An incremental approach for hardware discrete controller synthesis. Other. INSA de Lyon, 2011. English. NNT : 2011ISAL0071 . tel-00679296

HAL Id: tel-00679296

https://theses.hal.science/tel-00679296

Submitted on 15 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSA DE LYON

An incremental approach for hardware Discrete Controller Synthesis

by

Mingming Ren

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Laboratoire Ampère

June 2011

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Abstract

The Discrete Controller Synthesis (DCS) technique is used for automatic generation of correct-by-construction hardware controllers. For a given plant (a state-based model), and an associated control specification (a behavioral requirement), DCS generates a controller which, composed with the plant, guarantees the satisfaction of the specification. The DCS technique used relies on binary decision diagrams (BDDs). The controllers generated must be compliant with standard RTL hardware synthesis tools.

Two main issues have been investigated: the *combinational explosion*, and the *actual generation of the hardware controller*. To address combinational explosion, common approaches follow the "divide and conquer" philosophy, producing modular control and/or decentralized control. Most of these approaches do not consider explicit communication between different components of a plant. Synchronization is mostly achieved by sharing of input events, and outputs are abstracted away. We propose an incremental DCS technique which also applies to communicating systems. An initial modular abstraction is followed by a sequence of progressive refinements and computations of approximate control solutions. The last step of this sequence computes an exact controller. This technique is shown to have an improved time/memory efficiency with respect to the traditional global DCS approach.

The hardware controller generation addresses the control non-determinism problem in a specific way. A partially closed-loop control architecture is proposed, in order to preserve the applicability of hierarchical design. A systematic technique is proposed and illustrated, for transforming the automatically generated control equation into a vector of control functions.

An application of the DCS technique to the correction of certain design errors in a real design is illustrated. To prove the efficiency of the incremental synthesis and controller implementation, a number of examples have been studied.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Résumé

La synthèse de contrôleurs discrets (SCD) est appliquée pour générer automatiquement des contrôleurs matériels corrects par construction. Pour un système donné (un modèle à états), et une spécification de contrôle associée (une exigence comportementale), cette technique génère un contrôleur qui, composé avec le système initial, garantit la satisfaction de la spécification. La technique de SCD utilisée dans ce travail s'appuie sur les diagrammes de décision binaire (BDDs). Les contrôleurs générés doivent être compatibles avec les outils standards de synthèse matérielle de niveau RTL.

Deux problèmes principaux ont été examinés: l'*explosion combinatoire* et la *génération effective du contrôleur matériel*. La maîtrise de l'explosion combinatoire s'appuie sur des approches de type «diviser pour régner », exploitant la modularité du système ou du contrôleur. La plupart des approches existantes ne traitent pas la communication explicite entre différents composants du système. Le mécanisme de synchronisation le plus couramment envisagé est le partage des événements d'entrée, faisant abstraction des sorties. Nous proposons une technique de SCD incrémentale qui permet de traiter également les systèmes communicants. Une étape initiale d'abstraction modulaire est suivie par une séquence progressive de raffinements et de calculs de solutions approximatives de contrôle. La dernière étape de cette séquence engendre un contrôleur exact. Nous montrons que cette technique offre une efficacité améliorée en temps/mémoire par rapport à l'approche traditionnelle globale de la SCD.

La génération du contrôleur matériel s'appuie sur un traitement spécifique du non-déterminisme de contrôle. Une architecture de contrôle à boucle partiellement fermée est proposée, afin de permettre une conception hiérarchique. Une technique automatique transformant une équation de contrôle en vecteur de fonctions de contrôle est proposée et illustrée.

La SCD est ensuite appliquée et illustrée sur la correction de certaines erreurs de conception. L'efficacité des techniques proposées est illustrée sur un ensemble d'exemples de conception matérielle.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Contents

Abstract iii

Résumé iv

List of Figures ix

List of Tables xiii

1 Introduction 1
- 1.1 Motivations and context . . . 1
- 1.2 Contributions . . . 3

2 Discrete Controller Synthesis 5
- 2.1 Introduction . . . 5
- 2.2 Modeling discrete-event behaviors : the finite state machine model . . . 7
  - 2.2.1 Event-driven modeling . . . 7
    - 2.2.1.1 Formal language . . . 7
    - 2.2.1.2 Finite state automata . . . 8
      - Remark. . . . 10
    - 2.2.1.3 Execution paradigm of event-driven models . . . 11
  - 2.2.2 Sample-driven modeling . . . 13
    - 2.2.2.1 Formal model . . . 13
    - 2.2.2.2 Execution paradigm of sample-driven modeling . . . 14
    - 2.2.2.3 From event-driven to sample-driven . . . 14
    - 2.2.2.4 Example . . . 15
  - 2.2.3 Modeling outputs . . . 16
  - 2.2.4 Boolean representation . . . 19
  - 2.2.5 Symbolic representation . . . 20
      - Remark. . . . 22
  - 2.2.6 Efficient representation using BDDs . . . 22
- 2.3 Control specifications . . . 24
  - 2.3.1 Logic specifications . . . 25
    - 2.3.1.1 Specification in Computation Tree Logic (CTL) . . . 25
    - 2.3.1.2 Temporal logic evaluation . . . 27
      - Predicate transformers . . . 27
      - Fix point computation of CTL formulae . . . 28
  - 2.3.2 Operational specifications . . . 29

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Monitor-based control specifications 29
2.4 Supervisor synthesis 31
2.4.1 About controllability 31
2.4.2 Symbolic supervisor synthesis 32
2.5 Conclusion 34

3 Incremental Discrete Controller synthesis **35**
3.1 Introduction 35
3.1.1 Memory evolution during DCS. 35
3.1.2 Related works 37
3.2 Definitions 40
3.2.1 Controllable modular finite state machines (CFSM) 40
3.2.2 Synchronous composition 41
3.3 Incremental DCS technique 43
3.3.1 Abstraction 45
3.3.2 Abstract set refinement 45
3.3.3 The Incremental DCS algorithm 46
3.4 First example : an arbiter model 48
3.4.1 Applying global DCS to the arbiter model 49
3.4.2 Applying IDCS to the arbiter model 51
3.4.2.1 Abstraction 51
3.4.2.2 Approximate $\mathcal{IUC}$ computation 51
3.4.2.3 Refinement and final synthesis 51
3.5 Second example: a 3-way arbiter with tokens 52
3.5.1 Defining the DCS problem 54
3.5.2 Applying IDCS to the 3-way arbiter 55
3.5.2.1 Abstraction 55
3.5.2.2 Approximate $\mathcal{IUC}$ computation. 57
3.5.2.3 Refinement 57
3.5.2.4 Global synthesis and final results 57
3.6 Implementation and performance issues 59
3.7 Experimental results and performance figures 60
3.7.1 A more complex bus arbiter with token 61
Functional description. 61
Control specification. 63
Incremental DCS. 63
3.7.2 Fault-tolerant task scheduling problem 64
Functional description. 64
Incremental DCS. 66
3.7.3 The “Cat and mouse” problem 66
Functional description 67
Incremental DCS. 67
3.7.4 The “Dining philosophers” problem 68
Functional description. 68
Control specification. 68
Incremental DCS. 70
3.7.5 PI-Bus arbiter 71

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Functional description. . . . 71
Incremental DCS. . . . 73
3.7.6 Results 75
3.8 Conclusion 75

4 Implementation of supervisors **77**
4.1 Introduction 77
4.1.1 Contributions 78
4.1.2 State of the art 78
4.2 The supervisor implementation problem 79
The control non-determinism problem 80
The structural compatibility problem 80
4.3 Symbolic Supervisor Implementation 81
Supervisor implementation algorithm 82
4.4 Application of the supervisor implementation algorithm 85
4.4.1 Functional description 85
4.4.2 Supervisor implementation 86
4.4.3 Interpretation of the implemented supervisor 87
4.5 Applying DCS to hardware design 88
4.5.1 Component-based design 88
Example of the 3-way arbiter with token 89
4.5.2 Automatic error correction 90
4.5.2.1 A serial-parallel converter 90
4.5.2.2 Correction analysis 93
4.5.2.3 Discrete Controller Synthesis 94
4.6 Implementation 95
4.7 Conclusion 96

5 Conclusion **99**

A Résumé **101**
A.1 Introduction à la SCD Incrémentale 101
A.1.1 Evolution de l'usage de mémoire pendant la SCD. 101
A.2 Définitions 103
A.2.1 Machines d'état fini contrôlable (CFSM) 103
A.2.2 Composition synchrone 104
A.3 Technique de SCD incrémentale 106
A.3.1 Abstraction 107
A.3.2 Raffinement d'ensemble d'états abstraits 109
A.3.3 L'algorithme de la SCD incrémentale 110
A.4 Premier exemple: un modèle d'arbitre 111
A.4.1 Appliquer la SCD globale au modèle de l'arbitre 113
A.4.2 Appliquer la SICD au modèle de l'arbitre 115
A.4.2.1 Abstraction 115
A.4.2.2 Le calcul du $\mathcal{IUC}$ approximatif 116
A.4.2.3 Raffinement et la synthèse finale 116
A.5 Deuxième exemple: un arbitre avec jeton 117
A.5.1 Définition du problème de la SCD 118

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

A.5.2 Appliquer la SICD à l'arbitre 118
A.5.2.1 Abstraction 119
A.5.2.2 Calcul du $\mathcal{IUC}$ approximatif. 120
A.5.2.3 Raffinement 120
A.5.2.4 Synthèse finale et résultats finaux 121
A.6 Mise en oeuvre et la performance 121
A.7 Introduction à l'implémentation du superviseur 124
A.8 Définitions 126
A.8.1 Machines d'états finis contrôlables 126
A.8.2 Synthèse de contrôleurs discrets 127
A.9 Implémentation Symbolique du Superviseur 129
A.9.1 Conditions pour la compatibilité structurelle 129
A.9.2 Algorithme d'implémentation symbolique 130
A.10 Application de la SCD à la conception de matériel 133
A.10.1 Conception basée sur composants 133
Exemple de l'arbitre avec jeton 134
A.10.2 Correction automatique des erreurs 135
A.10.3 Le convertisseur série-parallèle 136
A.10.4 Analyse de correction 137
A.10.5 Correction de l'erreur par SCD 138
A.11 Mise en œuvre 139
A.12 Conclusion 141

Bibliography **143**

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

List of Figures

2.1 A simple 3-state machine . . . 10
2.2 Two 3-state machines . . . 12
2.3 Synchronous composition of two simple machines . . . 12
2.4 From Event-driven to Sample-driven model . . . 15
2.5 Examples of Moore and Mealy machine . . . 17
2.6 Structural representation of the synchronous product . . . 18
2.7 A 3-state machine . . . 19
2.8 A 3-state machine . . . 22
2.9 BDD of $f = x_1 \cdot x_2 + x_3$ . . . 23
2.10 ROBDD of different variable order . . . 24
2.11 Illustration by computation trees . . . 26
2.12 Example of CTL specifications . . . 27
2.13 Monitor of state splitting specification . . . 30
2.14 Monitor for event alternation specification . . . 30
2.15 Recognizer of illegal substring specification . . . 30
2.16 Monitor of illegal substring specification . . . 31
2.17 Schematic control architecture achieved by DCS . . . 32
2.18 First iteration . . . 33
2.19 Second iteration . . . 34

3.1 Peak memory usage during DCS . . . 36
3.2 Modular structure of a controllable system . . . 41
3.3 Incremental DCS technique versus direct DCS . . . 44
3.4 The arbiter model . . . 48
3.5 Global DCS for $M_1||M_2$ and *spec* . . . 50
3.6 Abstract model $M_1||M_2^{abs}$ and the approximate $\mathcal{IUC}^{abs}$ computation . . . 52
3.7 Final global DCS step ensuring *spec* . . . 53
3.8 Model of a single cell i . . . 53
3.9 Internal connection of the arbiter with token . . . 54
3.10 Abstraction sequence for the 3-way arbiter . . . 56
3.11 Abstract model $M_1||M_2||M_3^{abs}$ and the approximate $\mathcal{IUC}$ computation . . . 56
3.12 Refinement of $M_1||M_2||M_3^{abs}$ and the $\mathcal{IUC}$ computation . . . 58
3.13 Single cell of bus arbiter . . . 61
3.14 Token structure of bus arbiter chain . . . 62
3.15 Incremental synthesis procedure . . . 63
3.16 Task model with $i = \{1, 2\}$ . . . 65
3.17 Processor model with $j = \{1, 2, 3\}$ . . . 66
3.18 System architecture . . . 66

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

3.19 Cat and mouse example . . . 67
3.20 Cat and mouse model . . . 67
3.21 Dining philosophers problem . . . 69
3.22 FSM model of Dining philosophers problem . . . 69
3.23 Incremental DCS procedure . . . 70
3.24 System architecture of PI Bus . . . 72
3.25 Internal interaction . . . 72
3.26 Automaton of bus arbitrator . . . 74
3.27 Arbiter structure . . . 74

4.1 Traditional control loop . . . 79
4.2 Control loop with a symbolic supervisor $\mathcal{SUP}$ . . . 81
4.3 Removal of control non-determinism . . . 81
4.4 Target control architecture . . . 82
4.5 A manufacture line . . . 85
4.6 FSM models of the system . . . 85
4.7 Component-based design flow . . . 89
4.8 Implementation of the controlled 3-way arbiter . . . 90
4.9 Error correction design flow . . . 91
4.10 Serial-parallel converter . . . 91
4.11 Architecture of the serial-parallel converter . . . 91
4.12 FSM model of block A . . . 92
4.13 FSM model of block B . . . 92
4.14 FSM model of block C . . . 92
4.15 Error correction by DCS . . . 94
4.16 Manually solving non-determinism during interactive simulation: controllable rqi is clickable. . . . 96
4.17 Design flow with supervisor implementation . . . 96

A.1 Pic de l'usage de mémoire pendant la SCD . . . 102
A.2 Structure modulaire d'un système contrôlable . . . 104
A.3 La SCD incrémentale versus la SCD direct . . . 108
A.4 Le modèle de l'arbitre . . . 112
A.5 SCD globale pour $M_1||M_2$ et *spec* . . . 114
A.6 Modèle abstrait $M_1||M_2^{abs}$ et le calcul du $\mathcal{IUC}^{abs}$ approximatif . . . 115
A.7 La SCD finale assurant *spec* . . . 116
A.8 Modèle d'une seule cellule i . . . 117
A.9 Connexion interne de l'arbitre avec jeton . . . 118
A.10 Ordre d'abstraction de l'arbitre . . . 119
A.11 Modèle abstrait $M_1||M_2||M_3^{abs}$ et le calcul de $\mathcal{IUC}$ approximatif . . . 120
A.12 Raffinement de $M_1||M_2||M_3^{abs}$ et le calcul du $\mathcal{IUC}$ . . . 122
A.13 Boucle de contrôle classique . . . 128
A.14 Boucle de contrôle avec un superviseur $\mathcal{SUP}$ symbolique . . . 128
A.15 levée du non-déterminisme de contrôle . . . 129
A.16 architecture de contrôle souhaitée . . . 129
A.17 Flot de conception basée sur composants . . . 134
A.18 Implémentation de l'arbitre contrôlé . . . 135

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

A.19 Flot de conception de la correction des erreurs 136
A.20 architecture du convertisseur série-parallèle 136
A.21 correction d’erreurs de conception par SCD 137
A.22 Flot de conception avec implémentation de contrôleurs 140

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

List of Tables

3.1 Experiment results 75

4.1 Truth table of the Boole decomposition of $\mathcal{SUP}$ with respect to c_i 83
4.2 Values of c_i when $\mathcal{SUP}$ is satisfiable 83

A.1 a) décomposition de Boole appliquée à $\mathcal{SUP}$ par rapport à la variable c_i; b) Valeurs de c_i lorsque $\mathcal{SUP}$ est satisfaisable 131

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Chapter 1

Introduction

1.1 Motivations and context

Safe design of hardware embedded systems is a fundamental concern of all major semiconductor companies. Most embedded systems are safety critical, which means that they should be error-proof, and sometimes even fault-tolerant. Moreover, such systems are concurrent "by nature": different processes run in parallel and possibly cooperate; they are modeled as synchronous reactive systems, by using communicating finite states machines. This aspect adds a new dimension of complexity to the design problem. Ensuring correction calls for a specific design process, which encompasses a dedicated design method, as well as specific tools for design and verification. At any rate, if safety is not critical, the design correction still remains an extremely important aspect, much more important than it is in software design. Indeed, software can be corrected indefinitely by issuing correcting "patches". This is not the case of hardware: the design process ends with the production of a physical piece of hardware, which is a very expensive step. VLSI[1] hardware such as ASIC[2]'s cannot be modified anymore, unless it is refactored once again. Unfortunately, design errors may still be discovered after the system design, verification and manufacture steps. The notorious Pentium FDIV bug has been reported after the product release and costs Intel $475 million dollars [1]. When such a bug is discovered, manual correction of the bug is a very delicate step which requires deep insight of the system behavior. What makes such an intervention worse is that, correcting an error manually can cause even more bugs. Sometimes, rewriting code is physically impossible. Thus, prior to the physical implementation, it remains critical to ensure that the design is error-free.

[1] Very-large-scale integration (VLSI) is the process of creating integrated circuits by combining thousands of transistors into a single chip

[2] Application-specific integrated circuit (ASIC) is integrated circuit customized for a particular use, rather than intended for general-purpose use.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

For that purpose, formal methods are fundamental. Besides the design method which can be formal, such as VDM[3], UML[4] or B[5], followed by actual coding in VHDL or Verilog, design verification is an essential part of the design process. It can even take a major part of the total development time. Traditional simulation remains ubiquitous; designers heavily rely on this technique in order to acquire insight and confidence in their design. Formal tools such as *property checking* of *theorem proving* are complementary to simulation. Property checking (also known as model checking) allows the discovery of *corner case situations*: these are unwanted configurations which are very difficult to uncover by simulation, due to the complexity of the design at hand.

Property checking relies on a formal representation of the design which is the finite state machine model, and the formal expression of an assertion, using temporal logic [2], which it attempts to prove against the design model. Even though this technique seems very powerful, it reaches its theoretical limits in practice, due to the growing complexity of the designs. A more profitable approach combines the formal expression of assertions and random simulation. This has the same complexity as ordinary simulation, and thus remains much more affordable.

However, let us highlight the following two situations, which we consider as frequent during a hardware design process:

- a medium-sized function needs to be prototyped within a limited amount of time, with a guarantee of correctness with respect to its specifications;
- a critical design error has been uncovered and formally characterized. A reliable fix must be proposed.

In both situations, the designer needs to write code manually, and ensure afterwards that his code is correct, and that there is no regression induced by the bug fix. This is usually done by simulation, applied in conjunction with formal verification, and requires an amount of time which is incompressible or even growing.

In order to support the designer's work in such situations, we advocate the use of the Discrete Controller Synthesis (DCS) [3] technique as a complementary formal design tool for automating the production of correct-by-construction code. DCS is a promising approach and works for systems which can be represented as finite state machines. The DCS technique relies on the supervisory control theory. It has already been applied in the automatic control design for manufacturing systems [4].

DCS starts with both the description of a system and of the desired behavior, also known as the control specification, for that system. The system model must be compliant with the finite state machine model.

[3]Vienna Development Method (VDM) is a formal method for the development of computer-based systems.

[4]Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of object-oriented software engineering.

[5]B is a tool-supported formal method used in the development of computer software. It was originally developed by Jean-Raymond Abrial. B supports development of programming language code from specifications.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

The control specification is expressed either as a temporal logic formula, or operationally, as a finite state machine modeling the desired dynamic behavior by a succession of states/transitions. DCS performs a static analysis of the system with respect to the control specification, and automatically constructs a *supervisor*, if it exists. The supervisor is composed with the initial system and the result is guaranteed to satisfy the control specification.

It is important to note that the DCS technique is based on the same state set representation and traversal technology as property checking: the symbolic techniques and the binary decision diagrams (BDDs) [5]. State variables are used to encode sets of states, and the BDD technology achieves a compact representation of these sets. Unfortunately, even though BDDs have brought an important performance enhancement for symbolic techniques, their theoretical spatial complexity is exponential in the number of state variables used. This puts a firm bound on the size of the systems that can be handled, while this size is continuously increasing.

We investigate the symbolic BDD-based implementation of DCS. As stated above, unfortunately, DCS shares the same theoretical complexity as property checking. However, property checking is supported by very efficient compositional methods which enhance its performance. Concerning DCS, such compositional techniques and methods are fewer and less adequate for the hardware design context.

1.2 Contributions

Our main objective is to assess the utility of DCS in hardware design, and to propose a design flow which integrates the use of this technique besides formal verification and simulation, in order to enforce design correction. This work only considers reactive synchronous systems, modeled by implicitly clocked, communicating finite state machines, and compliant with hardware synthesis tools.

In this document, we propose an incremental DCS (IDCS) technique which tackles the complexity of symbolic DCS, offering better performance results. Though IDCS exploits the modular structure of a system, it cannot be qualified as compositional. Indeed, IDCS does not produce local solutions which only need to be "composed together" in order to obtain a final global solution. In that, IDCS gives worse performance results than other compositional techniques. However, the compositional DCS techniques available in the literature do not currently handle explicit communication between concurrent modules of a given system. Still, communication is supported under the form of synchronization on the occurrence of a shared input event. However, hardware designs call for the ability to model point to point communication, where a module can send information to its neighbors, and can also receive information. The incremental technique we propose fully supports such communication mechanisms.

The second contribution of this work is a supervisor implementation algorithm. The main advantage of symbolic BDD-based techniques is that they systematically manipulate sets of states, represented

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

symbolically by their characteristic equation. This advantage becomes a drawback when the solution of DCS, the supervisor, is also represented as a characteristic equation. Indeed, in hardware design, the components are the structural design elements. Each component must have an input/output interface. This is not the case for the equational representation of a supervisor component, which only encodes acceptable values of a set of variables, without producing any output, and without any distinction between input, state and output variables. To solve this problem, we propose an automatic technique for solving symbolically the control equation produced by DCS. This produces a set of Boolean control functions.

This dissertation is structured as following:

- Chapter 2, introduces the formal Finite State Machine model and notations used throughout the remaining of this document. The Discrete Controller Synthesis algorithm is also recalled and illustrated, as well as the different ways of expressing control specifications: temporal logic and monitors;
- Chapter 3 details the incremental supervisor synthesis procedure. The FSM model introduced in chapter 2 is extended in order to handle inter-component communication. Based on model, the incremental Discrete Controller Synthesis is developed and illustrated by two examples. A number of experimental results on different examples are also given in order to evaluate the performance of IDCS.
- Chapter 4 is dedicated to supervisor implementation. A symbolic control equation is transformed into a series of control functions. This implementation technique not only removes the structural incompatibility of the symbolic supervisor, but it also move the non-determinism of the supervisor to its environment.

Finally, a conclusion and global perspectives are commented in the conclusion of this document.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Chapter 2

Discrete Controller Synthesis

This chapter recalls the fundamentals of the Discrete Supervisor Synthesis technique. The most frequent approaches for formally modeling the system behavior and for specifying functional requirements are discussed. One formal model is chosen and presented in detail, together with the supervisor synthesis technique used in the remaining part of this document.

2.1 Introduction

In control theory, "feedback" is a fundamental concept. A *controller* continuously adjusts the behavior of a *system* by tuning a set of dedicated variables, known as *controllable variables*. Control is achieved according to a desired *setpoint* and the observation of the state of the system. In this control paradigm, information is looped back to the system. Thus, its past evolutions are used at each adjustment of the controllable variables.

The discrete supervisory control theory is a branch of the control theory which applies to discrete *finite state/transition* systems by providing a similar "closed-loop" feedback control scheme. In the supervisory control paradigm, a supervisor continuously gets state information from the system and the environment, and feeds control information back to the system, in order to enforce some functional properties: either make invariant a given behavior which the system should always feature, or enforce a target configuration, which the system should *eventually* reach. The technique which constructs such a supervisor is known as Discrete Controller Synthesis, which we abbreviate DCS. Strictly speaking, a controller is generally not identical to a supervisor, but obtained from a supervisor. The DCS technique encompasses both the production of a supervisor and its transformation into a controller, as explained in chapter 4.

Several DCS techniques exist, and they differ according to their underlying formal model and their implementation technique. The *formal language* theory is the underlying framework formalizing *event-driven* systems. In this framework, sequences of events are fundamental in modeling the dynamic behavior of

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

the system, its functional requirements, and in achieving the actual synthesis. On the other hand, *sample-driven* systems are inspired from electronic design techniques. Their dynamic behavior is expressed as sequences of system's states, and functional requirements as subsets of states.

The dynamics of event-driven and sample-driven models differ conceptually. Event-driven models react according to their internal state, upon an event, whenever it occurs. This is also known as an "event-driven" reaction. Sample-driven models admit only one kind of events: those generated by a clock. At each clock event, both the inputs and the current state are *probed*, or *sampled* before reacting. Systems reacting to the events generated by one external clock are also known as time-driven systems.

The implementation of most DCS techniques is either symbolic or enumerative. Symbolic DCS relies on an implicit, symbolic representation of a set of states, as well as a symbolic state-space traversal technique, whose incontestable advantage is the efficient manipulation of large sets of states. Initially, the implementation of the Ramadge-Wonham DCS algorithm [6] was enumerative, and has been recently redesigned, to take advantage from the symbolic representation and traversal techniques [7].

Given the variety of models and techniques available for performing DCS, a preliminary choice is needed, according to the particular context of this work: hardware modeling. The modeling paradigm used in hardware systems design is the synchronous finite state machine with Boolean representation, which is a sample-driven model. Thus, the sample-driven paradigm is the most adequate in the context of our work.

This chapter introduces notations used in the remaining part of this thesis, and formalizes the DCS technique on top of which our contribution is built. Three fundamental aspects are detailed: state-based modeling, functional requirement specification and supervisor synthesis. These are organized as follows:

- Section 2.2 introduces the state-based modeling approaches for discrete-event systems, by first presenting the Event-driven modeling in subsection 2.2.1 and the Sample-driven modeling in subsection 2.2.2; the differences between these two modeling paradigms are explained. A systematic procedure for transforming an event-driven model into a sample-driven, which can be behaviorally equivalent is also presented;
- the Boolean representation for finite state machines is presented in subsection 2.2.4 followed by the symbolic representation in subsection 2.2.5; Binary Decision Diagrams (BDD) are recalled in subsection 2.2.6 for efficient symbolic representation;
- Section 2.3 presents the control specification techniques used throughout this document;
- Section 2.4 recalls and illustrates a symbolic DCS approach used in the remaining part of this document.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

2.2 Modeling discrete-event behaviors : the finite state machine model

2.2.1 Event-driven modeling

This modeling paradigm defines dynamic behaviors as a collection of sequences of events. At a given moment, only one event may occur. The finite set of all possible events is called an alphabet. Sequences of events are known as words, and the set of all possible words on an alphabet is a language. These notions are recalled in the following.

2.2.1.1 Formal language

More details on the theory of formal language can be found in [8].

Definition 2.1 (Formal language)**.** Each DES has an associated underlying event set Σ called an *alphabet*. A sequence of events from the alphabet is called a "word" or "string". An empty event string is denoted by ε. We denote the cardinality of an event set Σ as $|\Sigma|$. For a particular event set Σ, we denote its set of all possible finite strings of events by Σ^*. A *formal language* defined over an event set Σ is a subset of Σ^*.

Example 2.1. *Some examples of string and language are shown below:*

- $\Sigma = \{a, b, c\}$ *is an alphabet;*
- a, ac, baa *are strings over alphabet* Σ*;*
- $L = \{\varepsilon, acc, bacb, cabbb\}$ *is a language over* Σ*.*

A set of operations are defined over languages. Let L_1,L_2 be two languages over a same alphabet Σ, we have operations as follows:

- union: $L_1 \cap L_2 = \{s \in \Sigma^* | s \in L_1 \text{ or } s \in L_2\}$;
- intersection: $L_1 \cup L_2 = \{s \in \Sigma^* | s \in L_1 \text{ and } s \in L_2\}$;
- difference: $L_1 \setminus L_2 = \{s \in \Sigma^* | s \in L_1 \text{ and } s \notin L_2\}$;
- complement: $\Sigma^* \setminus L_1 = \{s \in \Sigma^* | s \notin L_1\}$;
- concatenation:
$$L_1 L_2 = \{s \in \Sigma^* | s = s_1 s_2 \text{ and } s_1 \in L_1 \text{ and } s_2 \in L_2\};$$
- Prefix-closure: $\overline{L_1} = \{s \in \Sigma^* | \exists s' \in \Sigma^*, ss' \in L_1\}$;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- Kleene-closure:

$$L_1^* = \{\varepsilon\} \cup L_1 \cup L_1L_1 \cup L_1L_1L_1 \cup \cdots .$$

Definition 2.2 (Projection)**.** For a language L defined on event set Σ, its *projection* to a smaller event set $\Sigma_1 \subseteq \Sigma$ is a language $Proj_{\Sigma_1} L$ constructed by applying the $Proj$ operations on each of its strings:

$$Proj(\varepsilon) = \varepsilon;$$

$$Proj(\sigma) = \begin{cases} \sigma & \text{if } \sigma \in \Sigma_1 \\ \varepsilon & \text{if } \sigma \in \Sigma \setminus \Sigma_1 \end{cases};$$

$$Proj(s\sigma) = Proj(s)Proj(\sigma).$$

The inverse projection $Proj^{-1} : \Sigma_1 \rightarrow 2^{\Sigma^*}$ is a map from a smaller event set to larger event set and is defined as follows:

$$Proj^{-1}(t) = \{s \in \Sigma_1^* \quad | \quad Proj(s) = t\}.$$

We restrict ourselves to dynamic behaviors, for which any sequence of events can be represented as a *regular expression*. Recall that for any finite alphabet Σ, regular expressions describe sets of strings over Σ by using the following operations: symbol concatenation, alternative expression, grouping and cardinality. Syntactically, the symbol concatenation operation is implicit; the alternative expression operation is represented by the symbol "|", and the grouping is expressed by using brackets. Cardinality operations express optionality "?" (zero or one occurrences), arbitrary occurrence "$*$" (zero or more), and minimal occurrence "$+$" (one or more). Formal languages which can be entirely described by regular expressions, are known as *regular languages*.

2.2.1.2 Finite state automata

Regular languages are seldom used to exhaustively describe the behavior of a reactive system. Sometimes, regular expressions can be an intuitive way for partially specifying repetitive behavior requirements, but they are usually not used to exhaustively design a system. Instead, the most usual modeling artifact is the finite state automaton. Each regular language defined on a finite set of events can be *generated* by a deterministic finite state automaton, or Finite State Machine (FSM). This notion is recalled below.

Definition 2.3 (Deterministic Event-driven automaton)**.** A deterministic Event-driven automaton is defined as a 5-tuple:

$$M^E = (Q, \Sigma, \delta, q_0, Q_m)$$

where:

- Q is a finite set of states;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- Σ is a finite set of input events;
- $\delta : Q \times \Sigma \to Q$ is the transition function;
- q_0 is the initial state;
- $Q_m \subseteq Q$ is a set of marked or accepting states.

Note that the transition function δ is usually a partial function. Thus each state can be associated with a set of events which are admissible at that particular state, i.e. for which δ is defined.

Definition 2.4 (Active event function)**.** Define active event function $\Gamma : Q \to 2^{\Sigma}$ as following:

$$\Gamma(q) = \{\sigma \quad | \quad \delta(q, \sigma) \text{ is defined } \}.$$

In the definition of deterministic automata presented above, if the transition function δ is under the form $\delta : Q \times \{\Sigma \cup \epsilon\} \to 2^{Q}$, such an automaton is called a non-deterministic Event-driven automaton.

Definition 2.5. Let $q, q' \in Q$ be two states of a deterministic FSM M. We say q' is a *successor* of q, noted as $q \to q'$ if and only if $\exists \sigma \in \Sigma : q' = \delta(q, \sigma)$; q is said to be a *predecessor* of q'. Two functions can be defined:

- *Pred:* $Q \to 2^{Q}$, which is defined as $Pred(q) = \{q' | q' \text{ is a predecessor of } q\}$;
- *Succ:* $Q \to 2^{Q}$, which is defined as $Succ(q) = \{q' | q' \text{ is a successor of } q\}$.

Definition 2.6 (Execution path)**.** An execution path of a state machine M is a finite or infinite sequence $q_0 q_1 q_2 \cdots$ such that $q_{i+1} = \delta(q_i, \sigma_i), i = 0, 1, \cdots$.

Definition 2.7 (Reachable states)**.** A state q of machine M is reachable if there is an execution $q_0 \cdots q$, where q_0 is the initial state of M. All the states reachable from the initial state constitute the set of reachable states.

Two languages generated by a finite state automaton M are usually considered:

- the language generated by $M = (Q, \Sigma, \delta, q_0, Q_m)$ is defined as:

$$\mathcal{L}(M) = \{s = \sigma_1 \sigma_2 \cdots \in \Sigma^* | q_i = \delta(q_{i-1}, \sigma_i) \in Q, i = 1, \cdots\};$$

- the marked language generated by $M = (Q, \Sigma, \delta, q_0, Q_m)$ is defined as:

$$\mathcal{L}_m(M) = \{s = \sigma_1 \sigma_2 \cdots \sigma_n \in \mathcal{L}(M) | q_i = \delta(q_{i-1}, \sigma_i) \in Q, (i = 1, \cdots, n) \text{ and } q_n \in Q_m\}.$$

The following example illustrates the notion of event-based deterministic automaton.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

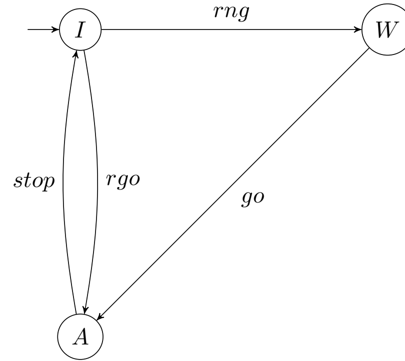


FIGURE 2.1: A simple 3-state machine

Example 2.2. *A simple event-based finite state machine with 3 states.*

Figure 2.1 shows a simple 3-state machine $M = (Q, \Sigma, \delta, q_0, Q_m)$ *where the state set* $Q = \{\boldsymbol{I}, \mathbf{W}, \boldsymbol{A}\}$ *represents an idle state* ***I****, a waiting state* **W***, and an active state* ***A****. The event set* $\Sigma = \{\boldsymbol{rng}, \boldsymbol{rg}, \boldsymbol{stop}, \boldsymbol{go}\}$ *represents a request not yet acknowledged* rng*, an acknowledged request* rgo*, a* stop *signal, and an acknowledge signal* go*. The state* ***I*** *is the only marked state.*

The transition function δ *can be written as:*

$$\delta(I, rng) = W$$
$$\delta(I, rg) = A$$
$$\delta(W, go) = A$$
$$\delta(A, stop) = I.$$

$Pred(A) = \{I, W\}$*,* $Succ(I) = \{W, A\}$*;* $I \to W \to A \to I$ *is one possible execution path. All three states are reachable from the initial state. This machine generates the language* $L = \{\epsilon,\ rng,\ rng.go,\ (rng.go.stop)^*,\ rgo,\ (rgo.stop)^*,\ \cdots\}$*, where* $s*$ *means substring* s *occurs at least once.*

This machine also generates the marked language $L_m = \{\epsilon, (rgo.stop)^*, (rng.go.stop)^*\}$*, which is a subset of the generated language* L*.*

Remark. Accepting (or marked) states Q_m are used in order to distinguish between desired and undesired behaviors. Desired behaviors are execution paths which end with an accepting state $q_m \in Q_m$. Thus, accepting states q_m are used to specify the point where a finite behavior should terminate. According to this notion of termination, an FSM can stop operating. This is more or less acceptable, depending on the application domain. Regular languages have been replaced by ω-regular languages, whose strings are necessarily infinite. They are generated by Büchi [9] automata. In a Büchi automaton, accepting states must be visited *infinitely often*. The notion of marking a state as accepting is a mention about *what* the design at hand should do, rather than *how* it is supposed to do it. Therefore, we relate this mechanism to design specification. However, this specification mechanism is incomplete; it needs to be

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

completed at least by invariant specifications. Therefore, in the sequel, we note $M^E = (Q, \Sigma, \delta, q_0)$ and do not represent marking states as part of an FSM definition and leave that to the specification process.

2.2.1.3 Execution paradigm of event-driven models

The dynamic behavior modeled by an event-driven finite state automaton can be described algorithmically, by a permanently-operating process modeled by an infinite loop. For a given Event-driven FSM, $M^E = (Q, \Sigma, \delta, q_0)$, this can be summarized in Algorithm 1.

Algorithm 1 Event-driven paradigm

1: current state declaration : $q \in Q$
2: next state declaration : $q' \in Q$
3: {initialization: $q = q_0$}
4: **loop**
5:    wait for event $\sigma \in \Sigma$
6:    next state computation: $q' = \delta(q, \sigma)$
7:    update state $q = q'$
8: **end loop**

In event-driven models, events can only occur one at a time. Simultaneous occurrence of two or more events is not allowed. This paradigm is used to model concurrent software systems where events cannot arrive simultaneously.

Concurrency is a fundamental property of most control systems: a collection of building blocks are composed together in order to achieve more complex control behaviors. The composition operation expresses a requirement of simultaneous run of the blocks composed together. This is formally defined by Milner's synchronous product recalled below.

Definition 2.8 (Synchronous product)**.** The synchronous product of two FSM $M_1 = (Q_1, \Sigma_1, \delta_1, q_{01})$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_{02})$ is an automaton denoted as $M_1||M_2$ and defined by:

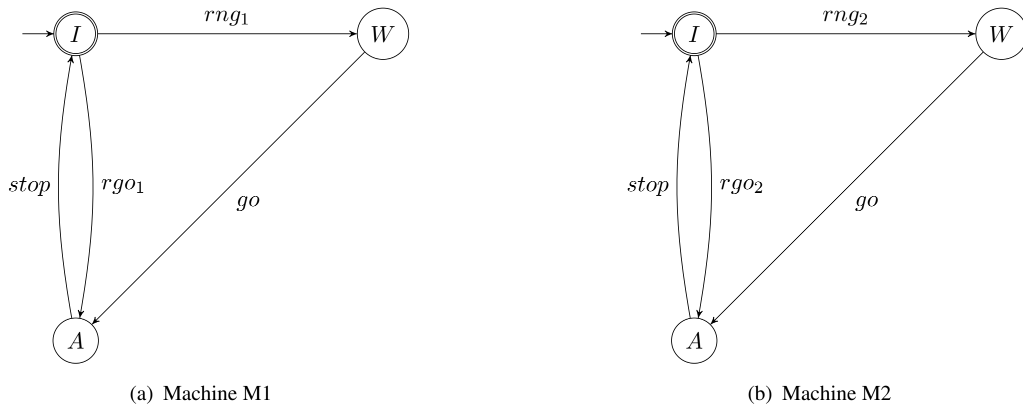
$$M_1||M_2 = \big(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{M_1||M_2}, (q_{01}, q_{02})\big)$$

where the transition function $\delta_{M_1||M_2} : Q_1 \times Q_2 \times (\Sigma_1 \cup \Sigma_2) \rightarrow Q_1 \times Q_2$ is defined by:

$$\delta\big((q_1, q_2), \sigma\big) = \begin{cases} \big(\delta_1(q_1, \sigma), q_2\big) & \text{if } \sigma \in \Gamma_1(q_1) \backslash \Gamma_2(q_2) \\ \big(q_1, \delta_2(q_2, \sigma)\big) & \text{if } \sigma \in \Gamma_2(q_2) \backslash \Gamma_1(q_1) \\ \big(\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)\big) & \text{if } \sigma \in \Gamma_1(q_1) \cap \Gamma_2(q_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example 2.3. *Synchronous product of two simple machines.*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés



(a) Machine M1 (b) Machine M2

FIGURE 2.2: Two 3-state machines

Figure 2.2 shows two simple 3-state machines, with shared event set $\Sigma_1 \cap \Sigma_2 = \{go, stop\}$. *The synchronous composition of such two machines is illustrated in Figure 2.3.*

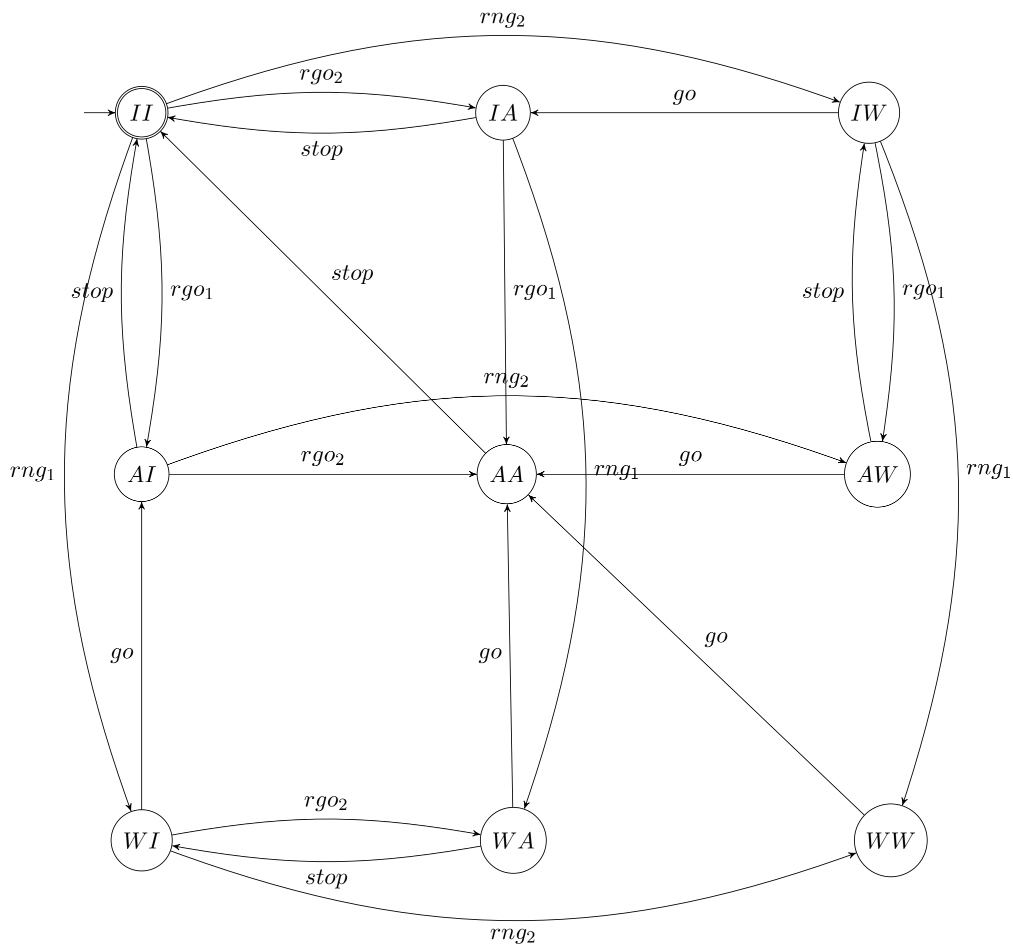


FIGURE 2.3: Synchronous composition of two simple machines

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

2.2.2 Sample-driven modeling

Unlike in the event-driven domain, sample-driven models are only sensitive to one event: an abstract clock. This clock mechanism is assumed to be unique for the whole system. It is used for sampling the environment: at each clock event, the values of some environment variables are read, or sampled. A reaction is computed according to the values read and the internal state of the model. The clock mechanism is said to be abstract because it does not describe explicit timing. It only issues successive ticks, at a frequency which is assumed to be sufficient for an accurate observation of the environment. All transitions of the model are triggered by the clock event, which is why the clock mechanism is left implicit.

Strictly speaking, event-driven models have an *asynchronous* behavior, which is more accurate than sample-driven ones'. Indeed, a sampled environment is only observed at clock ticks. However, for reasons related to physical implementation, event-driven models are sometimes translated into sample-driven ones. This translation can be done systematically, provided that a clock mechanism can be chosen and assumed as sufficient for observing, or sampling, any event occurrence.

In practice, the choice between event- and sample-driven models is application dependent. Event-driven models are considered more abstract, and are often found as semantic models for software designs, while sample-driven models are closer to a hardware implementation.

This section recalls the notations for sampled-driven models and shows one translation method from an event-driven to a sample-driven model.

2.2.2.1 Formal model

Definition 2.9. A sample-driven FSM M^S is defined as a 4-tuple, (Q, X, q_0, δ), where:

- Q: a finite set of states;
- X: a finite set of input variables over the Boolean domain;
- $q_0 \in Q$: the initial state;
- $\delta : Q \times \mathbb{B}^{|X|} \rightarrow Q$: the transition function;

Thus, as shown above, sampled-driven FSMs are no longer sensitive to environment events, but sample environment values at each clock event, which is left implicit.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Algorithm 2 Sample-driven paradigm

1: {initialization: $i = 0$, $q^i = q_0$}
2: **for** each clock tick i **do**
3: sample all input variables x^i
4: calculate next state: $q^{i+1} = \delta(q^i, x^i)$
5: **end for**

2.2.2.2 Execution paradigm of sample-driven modeling

For a given sample-driven FSM $M^S = (Q, X, q_0, \delta)$, its execution is represented by Algorithm 2.

In the sample-driven paradigm, each input variable is sampled on every tick of a clock, even when their values stay unchanged. The sample interval is assumed to be fine enough so that every change on input variables can be caught. The transition function δ is completely defined, because environment variables hold values at any moment. This paradigm has a hardware foundation where each component is synchronized by a clock and input signals are sampled on each tick of clock.

2.2.2.3 From event-driven to sample-driven

Our work applies to hardware systems, which have sample-driven underlying FSM models. However, our result can also be applied to event-driven models. These can be systematically translated into sample-driven representations. In the following we present one method for transforming an event-driven model into a sample-driven model. According to the specific modeling needs, the set of input events is mapped to a set of Boolean vectors, one for each input event. Thus, each event is mapped to a Boolean encoded value according to a function $enc : \Sigma \rightarrow \mathbb{B}^{\lceil \log_2 |\Sigma| \rceil}$. It maps each event in Σ into a unique Boolean vector encoding.

The inverse function enc^{-1} maps a Boolean vector back to its corresponding symbol, if it exists. If a vector has no associated symbol, it corresponds to a self-loop transition, modeling the absence of an event, symbolized by a fictive event abs. Thus $enc^{-1} : \mathbb{B}^{\lceil \log_2 |\Sigma| \rceil} \rightarrow \Sigma \cup \{abs\}$ is defined as:

$$enc^{-1}(\mathbf{x}) = \begin{cases} \sigma & \text{if } enc(\sigma) = \mathbf{x} \\ abs & \text{if } \nexists \sigma \in \Sigma, \text{ such that } enc(\sigma) = \mathbf{x} \end{cases}$$

The sampling of any event-driven FSM $M^E = (Q^E, \Sigma, q_0^E, \delta^E)$ through an encoding function enc produces a sample-driven counterpart $M^S = (Q^S, X^S, q_0^S, \delta^S)$, such that:

- $Q^S = Q^E$;
- $q_0^S = q_0^E$;

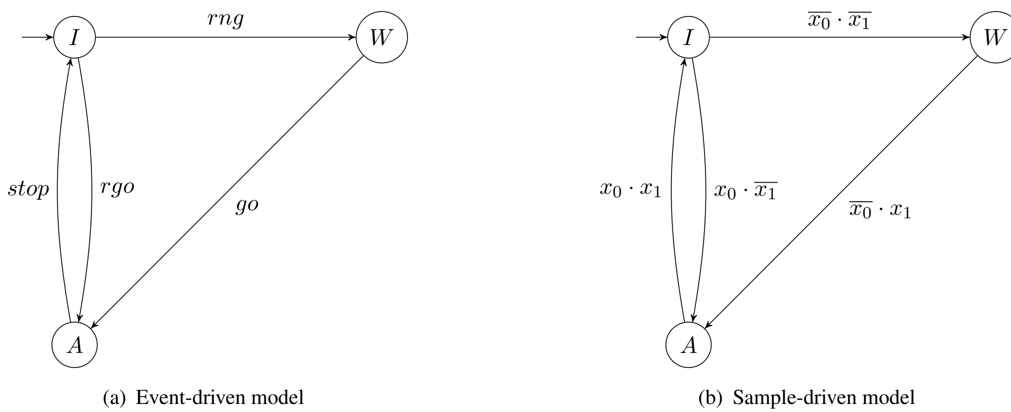
Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- $X^S = x_0^S x_1^S \cdots x_{n-1}^S$, where $n = \lceil \log_2 |\Sigma| \rceil$;
- $\delta^S : Q^S \times \mathbb{B}^n \to Q^S$ is the transition function defined as:

$$\delta^S(q^S, x^S) = \begin{cases} \delta^E\big(q^S, enc^{-1}(x^S)\big) & \text{if } enc^{-1}(x^S) \in \Sigma \\ q^S & \text{if } enc^{-1}(x^S) = abs \end{cases}$$

2.2.2.4 Example

Example 2.4. *Consider an event-driven FSM $M^E = (Q, \Sigma, q_0, \delta)$ shown in Figure 2.4(a).*



(a) Event-driven model (b) Sample-driven model

FIGURE 2.4: From Event-driven to Sample-driven model

To encode the events, we need at least 2 variables for input events. Let $X = \{x_0, x_1\}$. Consider a possible encoding function $enc(\sigma)$:

$$enc(rng) = \bar{x}_0\bar{x}_1;$$
$$enc(rgo) = x_0\bar{x}_1;$$
$$enc(go) = \bar{x}_0 x_1;$$
$$enc(stop) = x_0 x_1;$$

all the other possible combination of x_0 and x_1 fall into abs. In this case, $\emptyset$. The sample-driven model is shown in Figure 2.4(b).

From now on, unless explicitly specified, we use sample-driven FSM models with Boolean inputs.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

2.2.3 Modeling outputs

Concurrency in control systems calls for continuous interaction. The environment of a building block can be physical, such as a collection of sensors, or logical: a (collection of) building block(s). Thus, concurrent blocks must be able to communicate with each other through *outputs*.

Two modeling choices are possible and discussed here. On the one hand, outputs can be considered as being *implicit*: the input alphabet of a event- or sample-driven FSM can be defined as a *set of pairs* of input/output events. This method is advocated in [8] as being sufficient for modeling concurrent control systems. The modeling approach we chose represents outputs explicitly, and separately from the inputs. This choice is closer to the modeling style of hardware design engineers.

Definition 2.10. A FSM with outputs M is defined as a tuple:

$$M = (Q, X, q_0, \delta, PROP, \lambda),$$

where Q, X, q_0 and δ are defined as before, and :

- $PROP = \{p_1, \cdots, p_k\}$ is a set of k atomic Boolean propositions;
- $\lambda : Q \to \mathbb{B}^k$ is a labeling function such that $\lambda^j(q) = 1, j = 1, \cdots, k$ iff p_j is true in state q.

The labeling function λ models the outputs of M.

In practice, outputs can be assigned either to states, or to transitions. These two variants are referred to as *Moore* and *Mealy* FSMs :

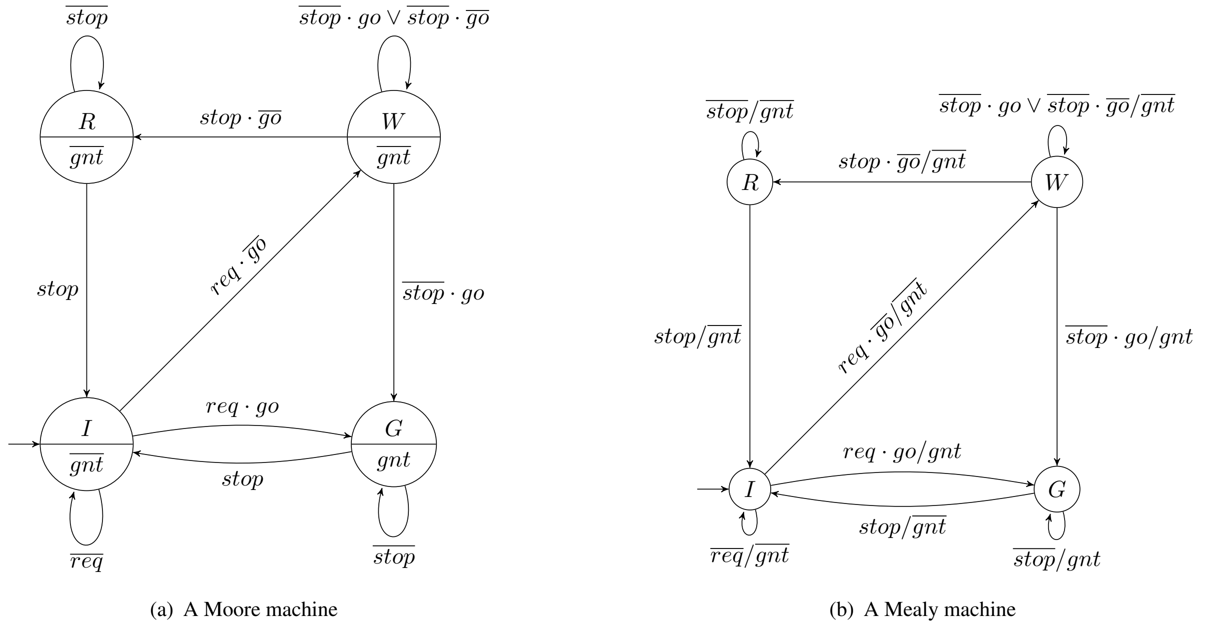
- for a Moore machine, the output function $\lambda : Q \to \mathbb{B}^k$ maps outputs to states;
- for a Mealy machine, the output function $\lambda : Q \times \mathbb{B}^{|X|} \to \mathbb{B}^k$ maps outputs to transitions.

Example 2.5. *Examples of Moore machine and Mealy machine are presented in Figure 2.5. Both models have identical event set and state set, as well as a unique output variable gnt. They only differ from the moment when the output variables are evaluated. For the Moore machine, as long as it stays in the same state, the output variables don't change their values; while for the Mealy machine, output variables are evaluated whenever a transition (even a self-directed transition) is triggered.*

Let **PROP** be the set of all Boolean propositions expressed over the elements of $PROP$. We define, by abuse of the notation, a reverse association λ^{-1} : **PROP** $\to 2^Q$, of a Boolean proposition with the subset of states in Q where it holds. The function λ^{-1} has the following properties:

- $\lambda^{-1}(p_1 \cdot p_2) = \lambda^{-1}(p_1) \cap \lambda^{-1}(p_2)$;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés



(a) A Moore machine (b) A Mealy machine

FIGURE 2.5: Examples of Moore and Mealy machine

- $\lambda^{-1}(p_1 + p_2) = \lambda^{-1}(p_1) \cup \lambda^{-1}(p_2)$;
- $\lambda^{-1}(\overline{p}) = Q \setminus \lambda^{-1}(p)$;

where $p_1, p_2,$ and p are Boolean propositions expressed over the elements of $PROP$.

Example 2.6. *The Moore machine in Figure 2.5 can be associated with a Boolean proposition set $PROP = \{gnt, \overline{gnt}\}$. The labeling function λ has the following values:*

$$\lambda(I) = \{\overline{gnt}\};$$
$$\lambda(W) = \{\overline{gnt}\};$$
$$\lambda(R) = \{\overline{gnt}\};$$
$$\lambda(G) = \{gnt\}.$$

Furthermore, $\lambda^{-1}(\overline{gnt}) = \{I, W, R\}$, $\lambda^{-1}(gnt) = \{G\}$.

The synchronous product of two communicating FSMs $M_1||M_2$ takes into account their possible interactions. A user-defined mapping can associate pairs of inputs of M_1 to outputs of M_2 and vice-versa. We assume that $PROP_1 \cap PROP_2 = \emptyset$: an output cannot be driven by more than one FSM inside a synchronous product, otherwise contradictions may occur.

We compose FSMs according to the synchronous paradigm defined in [10]. Let $M_i, i = 1, 2$ defined as $M_i = (Q_i, X_i, \delta_i, q_{0i}, PROP_i, \lambda_i)$ be two FSMs. First, the inputs of M_i are partitioned into two disjoint sets : $X_i = L_i \cup I_i$.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Let $Prop_1^{out} \subseteq PROP_1$ be the subset of Boolean propositions connected to M_2 via L_2 and $Prop_2^{out} \subset PROP_2$, the subset of atomic properties connected to M_1 via L_1.

The synchronous composition $M_1||M_2$ is defined as:

$$M_1||M_2 = (Q_{12}, X_{12}, \delta_{12}, (q_{01}, q_{02}), PROP_{12}, \lambda_{12}),$$

where:

- $Q_{12} = Q_1 \times Q_2$;
- $X_{12} = I_1 \cup I_2$;
- $\delta_{12} : Q_1 \times Q_2 \times \mathbb{B}^{|X_1|+|X_2|} \to Q_1 \times Q_2$ is defined as:

$$\begin{aligned}\delta_{12}(q_1, q_2, \mathbf{i_1}, \mathbf{i_2}) =&(\delta_1(q_1, \mathbf{i_1}, \lambda_2^1(q_2), \cdots, \lambda_2^{|L_1|}(q_2)),\\ &\delta_2(q_2, \mathbf{i_2}, \lambda_1^1(q_1), \cdots, \lambda_1^{|L_2|}(q_1)));\end{aligned}$$

- $PROP_{12} = PROP_1 \cup PROP_2$;
- $\lambda_{12} : Q_{12} \to \mathbb{B}^{|PROP_1|+|PROP_2|}$ is defined as:

$$\lambda_{12}(q_1, q_2) = (\lambda_1(q_1), \lambda_2(q_2)).$$

The reverse function $\lambda_{12}^{-1} : PROP_1 \cup PROP_2 \to 2^{Q_1 \times Q_2}$ is defined as :

$$\lambda_{12}^{-1}(p) = \lambda_1^{-1}(p) \times Q_2 \cup \lambda_2^{-1}(p) \times Q_1.$$

The synchronous composition with communication is illustrated in Figure 2.6.

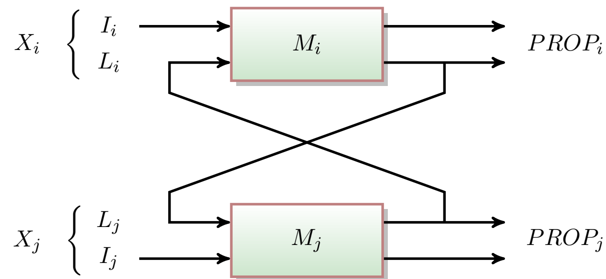


FIGURE 2.6: Structural representation of the synchronous product

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

2.2.4 Boolean representation

For the specific needs of hardware modeling, the set of states of an FSM is also represented in a Boolean manner, just as the inputs. Thus, the set of states is represented by a set of Boolean state variables and their value gives the current state of the FSM.

Definition 2.11 (Boolean model of a finite state machine)**.** Let M be a finite state machine with outputs $M = (Q, X, q_0, \delta, PROP, \lambda)$. Let $S_B = \{s_1, s_2, \cdots, s_n\}$, $X_B = \{x_1, x_2, \cdots, x_m\}$, be the sets of Boolean state and input variables, where $n = \lceil log_2|Q|\rceil$, $m = \lceil log_2|X|\rceil$, are the sizes of sets Q, X. The Boolean representation of M is $M_B = (S_B, X_B, \delta_B, s_{B0}, PROP, \lambda_B)$, constructed from the Boolean encoding functions $F_X : X \to \mathbb{B}^m$, and $F_S : Q \to \mathbb{B}^n$, such that:

- $s_{B0} = F_S(q_0)$;
- $\delta_B : \mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}^n$ satisfies:

$$\forall q \in Q, \forall x \in X, \quad \delta_B\big(F_S(q), F_X(x)\big) = F_S\big(\delta(q, x)\big);$$

- $\lambda_B : \mathbb{B}^n \to \mathbb{B}^k$ satisfies:

$$\forall q \in Q, \quad \lambda_B\big(F_S(q)\big) = \lambda(q).$$

Example 2.7. *Boolean representation of a 3-state machine. Take Example 2.2. To obtain a Boolean*

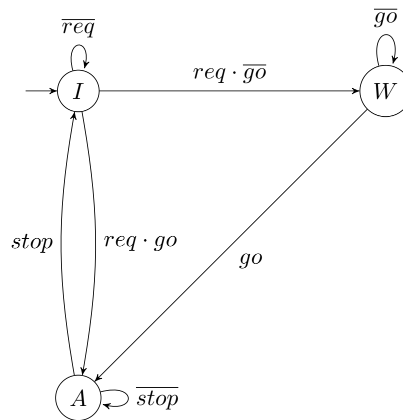


FIGURE 2.7: A 3-state machine

representation, we first determine vector sizes: $m = \lceil log_2|I|\rceil = 3$*,* $n = \lceil log_2|S|\rceil = 2$*. Thus we need two state variables* $S_v = \{s_1, s_2\}$*. For the input variables, we use the corresponding event name* $I = \{req, go, stop\}$*. We choose an encoding distribution, and let*

$$F_S = \Big\{\big(I, \langle 00\rangle\big), \big(W, \langle 01\rangle\big), \big(A, \langle 10\rangle\big)\Big\},$$
$$F_I = \Big\{\big(req, \langle 1**\rangle\big), \big(go, \langle *1*\rangle\big), \big(stop, \langle **1\rangle\big)\Big\},$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

where $*$ *means the value can take either* 0 *or* 1. *The transition function is*

$$\delta_B = \Big\{\big(\langle 00\rangle, \langle 0**\rangle, \langle 00\rangle\big), \big(\langle 00\rangle, \langle 10*\rangle, \langle 01\rangle\big), \big(\langle 00\rangle, \langle 11*\rangle, \langle 10\rangle\big), \big(\langle 01\rangle, \langle *0*\rangle, \langle 01\rangle\big),$$
$$\big(\langle 01\rangle, \langle *1*\rangle, \langle 10\rangle\big), \big(\langle 10\rangle, \langle **0\rangle, \langle 10\rangle\big), \big(\langle 10\rangle, \langle **1\rangle, \langle 00\rangle\big)\Big\}. \quad (2.1)$$

The FSM models exposed so far are formal models representing dynamic behaviors. They are fundamental in the mathematical definition of formal techniques, such as formal verification and discrete controller synthesis. Unfortunately, their size strongly limits the application of such techniques. One significant breakthrough is the definition of symbolic FSM, which allow the efficient manipulation of sets of states, instead of individual states.

2.2.5 Symbolic representation

In the Boolean representation of FSMs, states are still manipulated in an individual manner, as valuations of variables in S_B. However, Sets of states and/or transitions can be represented symbolically by constructing and manipulating their characteristic functions.

Definition 2.12 (Characteristic function)**.** Let $q \in Q$ be a state of a machine M, $F_S : Q \to \mathbb{B}^n$ is a state encoding function. The characteristic function of state q is a function $\mathcal{C}_s : \mathbb{B}^n \to \mathbb{B}$, defined by:

$$\mathcal{C}_s\big(\langle s_1, s_2, \cdots, s_n\rangle\big) = \prod_{j=1}^{n} \big(s_j \Leftrightarrow F_S^j(q)\big).$$

Let $E \subseteq S$ be a set of states, its characteristic function is the disjunction of all its elements' characteristic function:

$$\mathcal{C}_E\big(\langle s_1, s_2, \cdots, s_n\rangle\big) = \bigvee_{s \in E} \prod_{j=1}^{n} \big(s_j \Leftrightarrow F_S^j(s)\big).$$

The commonly used set operations have corresponding Boolean operators :

- union: $\mathcal{C}_{(A \bigcup B)} = \mathcal{C}_A + \mathcal{C}_B$;
- intersection: $\mathcal{C}_{(A \bigcap B)} = \mathcal{C}_A \cdot \mathcal{C}_B$;
- complement: $\mathcal{C}_{\overline{E}} = \overline{\mathcal{C}_E} : \mathbb{B}^n \to \mathbb{B}$ and $\mathcal{C}_{\overline{E}}(e) = 1 \Leftrightarrow e \notin E$.

Definition 2.13 (Symbolic transition relation)**.** Let s' be the vector of "next state" variables associated to s. We have: $\mathrm{s}' = \delta_B(\mathbf{s}, \mathbf{i})$. The symbolic transition relation $\mathcal{T}$ of M is defined as the set of all legal transitions $\mathrm{s} \overset{\mathbf{i}}{\mapsto} \mathrm{s}'$ such that $\mathrm{s}' = \delta_B(\mathbf{s}, \mathbf{i})$:

$$\mathcal{T}(\mathbf{s}, \mathbf{i}, \mathbf{s}') = \prod_{j=1}^{n} s'_j \Leftrightarrow \delta_B^j(\mathbf{s}, \mathbf{i}).$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Definition 2.14 (Symbolic model of FSM)**.** The symbolic representation of a finite state machine $M = (Q, I, \delta, q_0, PROP, \lambda)$ is a 6-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{C}_{s0}, \mathcal{T}, \mathcal{PROP}, \lambda_B)$, where

- $\mathcal{I}$: the set of input variables;
- $\mathcal{S}$: the set of state variables;
- $\mathcal{PROP}$: the set of atomic propositions;
- $\mathcal{C}_{s0}$: the characteristic function of the initial state;
- $\mathcal{T} : \mathbb{B}^n \times \mathbb{B}^m \times \mathbb{B}^n \to \mathbb{B}$ is the transition relation;
- $\lambda_B : \mathbb{B}^n \to \mathbb{B}^k$ is the labeling function.

The set of states where an atomic proposition $p_j \in \mathcal{PROP}$ is asserted *true* is defined as :

$$\mathcal{C}_{p_j}(\mathbf{s}) = \bigvee_{\mathbf{s} \in \mathbb{B}^n} \mathcal{C}_{\mathbf{s}}(\mathbf{s}).\lambda^j(\mathbf{s}).$$

Thus, the expression of the reverse labeling function $\lambda^{-1}(\mathbf{p})$ is obtained by substituting $\mathcal{C}_{p_j}$ for each atomic proposition $p_j \in \mathcal{PROP}$.

Definition 2.15 (Synchronous product [10])**.** Let $\mathcal{M}_1$, $\mathcal{M}_2$ be two symbolic models of finite state machine with $\mathcal{M}_1 = (\mathcal{S}_1, \mathcal{I}_1, \mathcal{T}_1, \mathcal{C}_{s01}, \mathcal{PROP}_1, \lambda_{B1})$ and $\mathcal{M}_2 = (\mathcal{S}_2, \mathcal{I}_2, \mathcal{T}_2, \mathcal{C}_{s02}, \mathcal{PROP}_2, \lambda_{B2})$, if $\mathcal{PROP}_1 \cap \mathcal{PROP}_2 = \emptyset$, then we have the synchronous product or parallel composition of $\mathcal{M}_1$ and $\mathcal{M}_2$, noted as $\mathcal{M}_1||\mathcal{M}_2$, which is a symbolic model of a finite state machine:

$$\mathcal{M}_{1||2} = (\mathcal{S}_{1||2}, \mathcal{I}_{1||2}, \mathcal{T}_{1||2}, \mathcal{C}_{s12}, \mathcal{PROP}_{1||2}, \delta_{1||2}),$$

where $\mathcal{S}_{1||2} = \mathcal{S}_1 \cup \mathcal{S}_2$, $\mathcal{I}_{1||2} = \mathcal{I}_1 \cup \mathcal{I}_2$, $\mathcal{PROP}_{1||2} = \mathcal{PROP}_1 \cup \mathcal{PROP}_2$, $\mathcal{C}_{s12} = \mathcal{C}_{s01} \cdot \mathcal{C}_{s02}$, and $\mathcal{T}_{1||2} = \mathcal{T}_1 \cdot \mathcal{T}_2$.

Example 2.8. *Symbolic representation of the 3-state machine model.*

Consider Example 2.2. Symbolic representation of each state is determined by their characteristic function:

$$\mathcal{C}_I(s_1, s_2) = \bar{s}_1 \cdot \bar{s}_2, \quad \mathcal{C}_W(s_1, s_2) = \bar{s}_1 \cdot s_2, \quad \mathcal{C}_A(s_1, s_2) = s_1 \cdot \bar{s}_2.$$

The whole set of states can be characterized by the disjunction of all its member state characteristic function:

$$\mathcal{C}_S(s_1, s_2) = \mathcal{C}_I(s_1, s_2) \vee \mathcal{C}_W(s_1, s_2) \vee \mathcal{C}_A(s_1, s_2)$$

which can then be simplified as

$$\mathcal{C}_S(s_1, s_2) = \bar{s}_1 \vee \bar{s}_2.$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

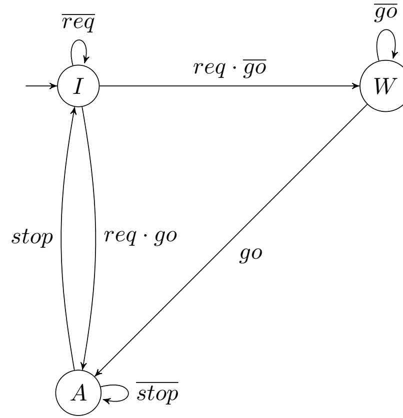


FIGURE 2.8: A 3-state machine

The transition function δ_B is defined as the disjunction of two functions δ_{B1} and δ_{B2}, one for each state variable respectively. According to equation 2.1 and the symbol assignment, three transitions lead to states where s_1 is true (state A):

$$\delta_{B1}(s_1, s_2, req, go, stop) = \bar{s}_1 \cdot \bar{s}_2 \cdot req \cdot go + \bar{s}_1 \cdot s_2 \cdot go + s_1 \cdot \bar{s}_2 \cdot \overline{stop} \tag{2.2}$$

and two transitions lead to states where s_2 is true (state I and W):

$$\delta_{B2}(s_1, s_2, req, go, stop) = \bar{s}_1 \cdot \bar{s}_2 \cdot req \cdot \overline{go} + \bar{s}_1 \cdot s_2 \cdot \overline{go}. \tag{2.3}$$

The propositional representation of transition relation is:

$$\mathcal{T} = \big(s'_1 \Leftrightarrow \delta_{B1}(s_1, s_2, req, go, stop)\big) \cdot \big(s'_2 \Leftrightarrow \delta_{B2}(s_1, s_2, req, go, stop)\big).$$

Remark. Symbolic representation is more compact that enumerative representation. Theoretically, it needs $log_2|S|$ variables to represent the state space. Moreover, the symbolic operations manipulate on a set of states instead of individual state. These advantages of scalability make it suitable to handle a larger set of problems.

2.2.6 Efficient representation using BDDs

To put the symbolic methods into practice, an efficient representation and manipulation of Boolean formulae is needed. Among the representations available, *Binary Decision Diagram* (BDD) [5] is most widely used.

Definition 2.16 (Binary Decision Diagram (BDD))**.** A Binary Decision Diagram is a rooted, directed acyclic graph whose vertex set V contains two types of vertexes:

- *terminal* vertices, which have a value of 0 or 1;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- *non-terminal* vertices, which have two children $low(v), high(v) \in V$, and an argument index $index(v) \in 1, \cdots, n$.

The root vertex v of a BDD G denotes a Boolean function f_v, whose value is defined as follows:

Definition 2.17 (Evaluation procedure)**.** f_v is evaluated from root through the graph:

- If v is a terminal vertex, $f_v = value(v)$;
- If v is a non-terminal vertex with $index(v) = i$,

$$f_v(\mathbf{x}) = \overline{x_i} \cdot f_{low(v)}(\mathbf{x}) + x_i \cdot f_{high(v)}(\mathbf{x}).$$

Example 2.9. *Figure 2.9 shows the original Binary Decision Diagram of Boolean function* $f(\mathbf{x}) = x_1 \cdot x_2 + x_3$.

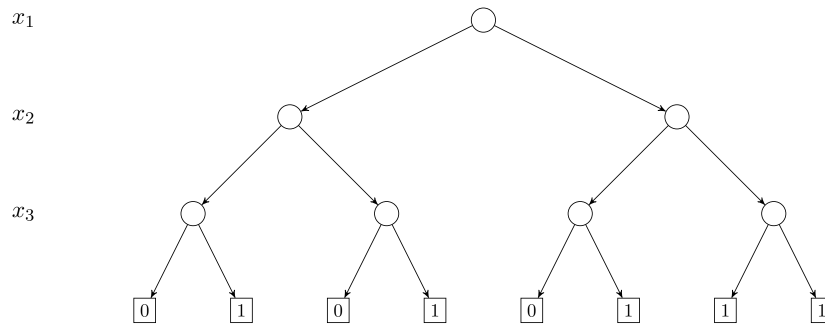


FIGURE 2.9: BDD of $f = x_1 \cdot x_2 + x_3$

Definition 2.18 (Ordered Binary Decision Diagram (OBDD))**.** A BDD is *Ordered* if the variables always respect a given order $x_1 < x_2 < \cdots x_n$, along any path from root to leaf through the graph.

The BDD size of a Boolean function depends largely on variable ordering. Selecting appropriate order is not a trivial task. Heuristic variable ordering and reordering are usually used to reduce the BDD size.

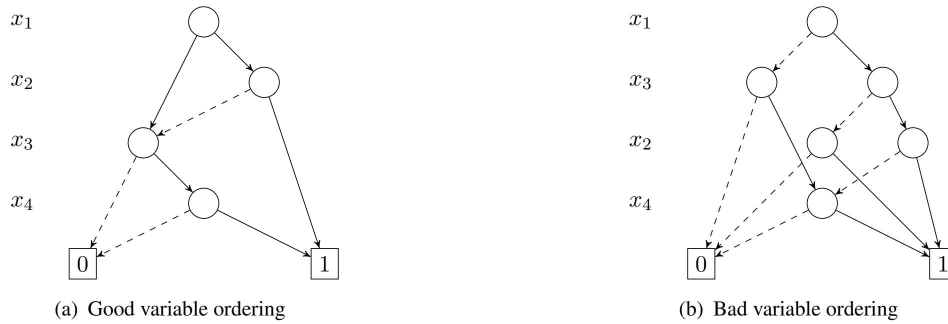
Definition 2.19 (Reduced Ordered BDD (ROBDD))**.** An OBDD is *Reduced* if it satisfies:

- *non-redundant tests* $\forall v \in V, low(v) \neq high(v)$;
- *uniqueness* $\forall v \in V, \nexists u \in V$, such that:

$$index(u) = index(v), \quad low(u) = low(v), \quad high(u) = high(v).$$

Example 2.10. *Figure 2.10 shows the corresponding ROBDDs of Boolean function f with different variable ordering. The variable ordering has a great impact on size of a BDD. For this particular function $f(\mathbf{x}) = x_1 \cdot x_2 + x_3 \cdot x_4$, evaluation procedure cannot decide the value of the first sub-formula until it reaches both variables x_1 and x_2. In the second graph, x_2 appears after x_3, thus cause bigger size.*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés



(a) Good variable ordering (b) Bad variable ordering

FIGURE 2.10: ROBDD of different variable order

Any BDD can be transformed into a corresponding ROBDD by application of two rules to itself. The *merging rule* identify and merge nodes that violate the *uniqueness* property. The *deletion rule* remove nodes that violate the *non-redundant tests*.

There's plenty of implementations of BDD in the form of C or Java libraries. Among them, CUDD are most widely used. Others are BuDDy [11], JavaBDD [12], CAL [13], etc. These libraries provide data structures of BDD, as well as a bunch of manipulations.

Ternary Decision Diagram (TDD) [14] is another structure which uses three-valued logic (true, false, unknown). It is adopted as internal data structure in the symbolic synthesis software SIGALI, which is the only symbolic tool suitable and available.

2.3 Control specifications

A specification defines the expected behavior of a system. Most frequently, behavioral requirements combine several aspects:

- *coherence*, also known as safety: some "bad" behavior should never occur ;
- *function delivery*, also known as liveness: some expected behavior must eventually occur.

Most behavioral requirements can be expressed by combining these two notions and their negations.

A control specification can be intrinsically satisfied by a system. However, when this is not the case, Discrete Controller Synthesis can enforce its satisfaction by attempting to make *invariant* the largest set of states of the system, satisfying that specification. This shall be explained in detail in section 2.4. Thus, control specifications addressed in this work are those that can be mapped systematically to sets of states where they hold.

Two main categories of control specifications are shown to be useful:

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- *logic specifications* describe the behavior of a system using propositional and/or temporal logic;
- *operational specifications* describe the behavior of a system as one or more desired sequences of states.

2.3.1 Logic specifications

Logic specifications express behavioral requirements by using propositional logic and some of its extensions. Among these extensions, we focus on Computation Tree Logic (CTL) [15]. It is a powerful means for expressing temporal behaviors. It also has the advantage that CTL formulae can evaluate to sets of states where they hold. These are recalled in the following.

2.3.1.1 Specification in Computation Tree Logic (CTL)

CTL expresses branching-time behaviors associated with Finite State Machine via execution trees. By *branching*, it is meant that at each moment (in each state), there are different paths towards future, thus yielding a tree-shaped execution. Thus, any FSM can be unwinded to an infinite *computation tree* which represents a collection of execution paths. Each computation tree starts from the initial state.

CTL temporal formulae are constructed by combining Boolean propositional logic with temporal operators: path and state quantifiers.

- Quantifiers over paths:
  - Universal path quantifier $A\varphi$: φ holds on all paths starting from the current state;
  - Existential path quantifier $E\varphi$: φ holds on at least one path starting from the current state.
- Path-specific quantifiers:
  - neXt-time operator $X\varphi$: φ holds at the next state;
  - Future operator $F\varphi$: φ holds at some state in the future;
  - Global operator $G\varphi$: φ holds on every state in the future;
  - Until operator $\varphi U\psi$: φ holds until ψ holds.

Definition 2.20 (Syntax of CTL formulae)**.** is defined as follows:

- Each Boolean atomic proposition is a CTL formula;
- if φ and ψ are CTL formulae, then $\overline{\varphi}$, $\varphi \cdot \psi$, $AX\varphi$ $EX\varphi$, $A(\varphi U\psi)$, $E(\varphi U\psi)$ are CTL formulae.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Other CTL operators can be represented by the operators above:

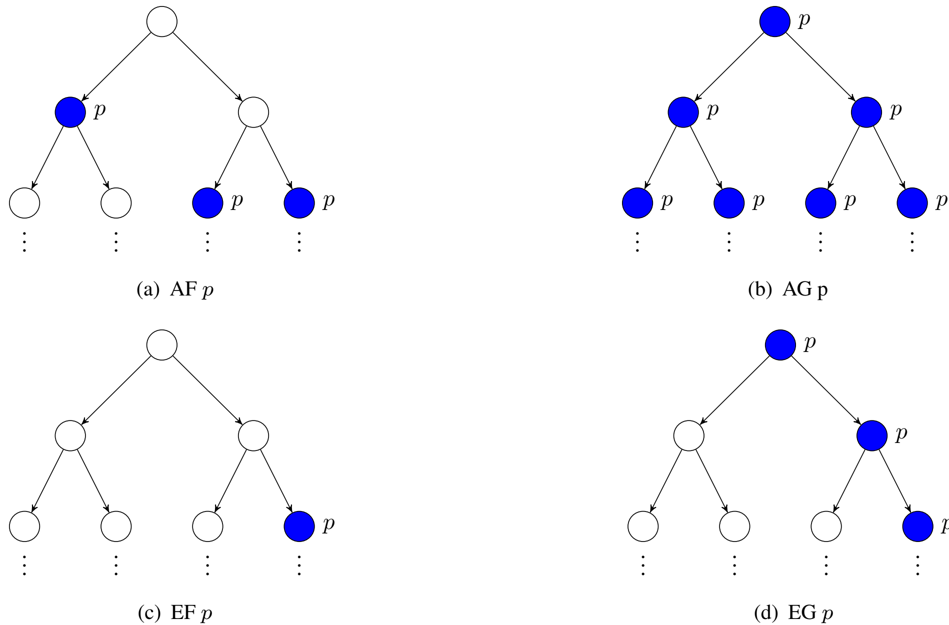
$$\begin{aligned}
\varphi + \psi &= \overline{\overline{\varphi} \cdot \overline{\psi}};\\
AF\varphi &= A(\text{true}\, U\varphi);\\
EF\varphi &= E(\text{true}\, U\varphi);\\
AG\varphi &= \overline{E(\text{true}\, U\overline{\varphi})};\\
EG\varphi &= \overline{A(\text{true}\, U\overline{\varphi})}.
\end{aligned}$$

CTL formulae are associated to Finite State Machines as assertions, which are either true or false in each state. We note $P, s \models \varphi$ that the formula φ is true in state s of P. The signification of some simple CTL formulae is recalled below:

$$\begin{aligned}
s_0 \models AX\varphi &\iff \forall(s_0, s_1, \cdots), s_1 \models \varphi;\\
s_0 \models EF\varphi &\iff \exists(s_0, s_1, \cdots), \exists i, s_i \models \varphi;\\
s_0 \models A(\varphi U \psi) &\iff \forall(s_0, s_1, \cdots), \exists i, s_i \models \psi, \forall j < i, s_j \models \varphi.
\end{aligned}$$

As CTL formulae can evaluate to truth values in each state of an FSM, sets of states can be designated by temporal formulae.

Example 2.11. *Figure 2.11 illustrates some CTL formulae by their corresponding computation tree.*



(a) AF p (b) AG p (c) EF p (d) EG p

FIGURE 2.11: Illustration by computation trees

Example 2.12. *CTL specifications*

- *AG $\overline{\varphi}$, something bad φ is never reached in any cases (safety);*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- $AG\ AF\ \varphi$, *φ is satisfied infinitely often (fairness);*
- $AG\ EF\ \varphi$, *from every reachable state, there must exist an execution path that leads to a state where φ is satisfied, thus φ is always possible (restart);*
- $AG\ (req \rightarrow AF\ ack)$, *whenever a request req is issued, an acknowledgment ack is always received. (reactivity, liveness) ($a \rightarrow b$ means implication, which is equivalent to $\overline{a} + b$);*
- $AG\ (req \rightarrow A\ (req\ U\ ack))$, *when a request is raised, an acknowledgment will eventually received, and the request must keep on until then.*

Example 2.13. *Consider a symbolic FSM as shown in Figure 2.12. Each state is assigned a set of atomic propositions. In this example, we consider state variables s_0, s_1, s_2 as such atomic propositions. Assume that state G and state RET are two states that are expected to be avoided. The question "is it true that for every execution path, state G and RET are always avoided?" can be expressed in CTL formulae $AG(\overline{\overline{s_0} \cdot s_1 \cdot \overline{s_2} + s_0 \cdot \overline{s_1} \cdot \overline{s_2}})$. This formula is not true, since it is always possible to get to state G or state RET. Then we want to know "is there an execution path that state G and RET are always avoided?", or in CTL formula $EG(\overline{\overline{s_0} \cdot s_1 \cdot \overline{s_2} + s_0 \cdot \overline{s_1} \cdot \overline{s_2}})$, which is true. The question "is it always possible to return to the initial state?" can be expressed in CTL formula $AG(EF(\overline{s_0} \cdot \overline{s_1} \cdot \overline{s_2}))$.*

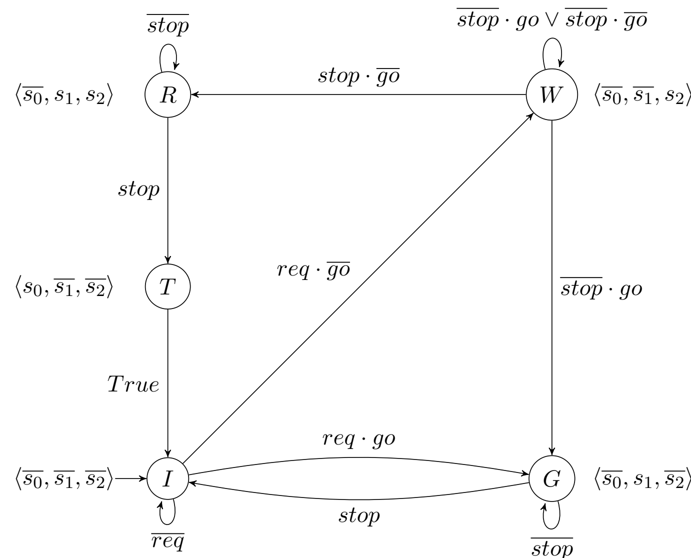


FIGURE 2.12: Example of CTL specifications

In the following, we recall the mapping between CTL formulae and sets of states where they hold.

2.3.1.2 Temporal logic evaluation

Predicate transformers Let $M = (S, I, \delta, s_0, PROP, \lambda)$ be a Boolean FSM. Let $Pred(S)$ be the lattice of Boolean predicates expressed over the variables in S. Each predicate of $Pred(S)$ identifies

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

a set of states of M, reachable or not. The ordering relation in $Pred$ is set inclusion. Thus, the least element in the lattice is the empty set, denoted by $\bot$, and the greatest element in the lattice is the set of all states, denoted by $\top$.

Definition 2.21 (Predicate transformer)**.** A predicate transformer is a functional $f : Pred(S) \to Pred(S)$; it is said to be *monotone* if

$$\forall P, Q \subseteq Pred(S), \quad P \subseteq Q \Rightarrow f(P) \subseteq f(Q).$$

According to the Tarski-Knaster theorem, a monotone predicate transformer f always has:

- a *least fix point*:
$$\mu Y.f(Y) = \bigcup_{i \geq 0} \underbrace{f \circ f \cdots \circ f(\bot)}_{i};$$
- a *greatest fix point*:
$$\nu Y.f(Y) = \bigcap_{i \geq 0} \underbrace{f \circ f \cdots \circ f(\top)}_{i}.$$

The calculation of the least fix point can be viewed as a progressive computation which applies the predicate transformer iteratively on a series of sets starting from the global set. Each application of the transformer excludes a subset from the result until a fix point is reached. The computation of the greatest fix point is an inverse procedure.

Fix point computation of CTL formulae CTL formulae can be evaluated by fix point calculation. Consider the CTL formula $EF\ p$, which means either p is true at the current state, or there must exist a next state where either p is true at that state or it has a next state which has the same property, etc. Thus the set $EF\ p$ can be characterized by the least fix point $\mu Y.(\lambda^{-1}(p) + EX\ Y)$. Why the least fix point but not the greatest fix point? Because any execution path containing a state that satisfies p can make $EF\ p$ true. On the other hand $EG\ p$ requires a greatest fix point, since it requires q to be true globally along at least one execution path.

Other useful characterizations are summarized below:

- $EG\ p = \nu Y.(p \cdot EX\ Y)$;
- $E(q\ U\ p) = \mu Y.\big(p + (q \cdot EX\ Y)\big)$;
- $AX\ p = \overline{EX\ \overline{p}}$;
- $AG\ p = \overline{EF\ \overline{p}} = \nu Y.(p \cdot AX\ Y)$;
- $A(q\ U\ p) = \mu Y.\big(p + (q \cdot AX\ Y)\big)$.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Our interest in CTL comes essentially from the ability of transforming any formula, expressed on the variables of an FSM, into a set of states where it holds. For a random CTL formula, this relies on regular symbolic model checking algorithms [16]. The set generated from CTL is then made invariant by applying DCS.

2.3.2 Operational specifications

Operational specifications are behavioral descriptions of requirements, expressed as one or more communicating FSMs. These FSMs have a particular structure: their inputs are dedicated to observations of system's variables; they take transitions according to the values observed. The violation of the desired behavior is modeled by a transition towards an error sink state. An output is asserted to false, as long as the behavior observed is correct. Once in the error state, the output is set to true. This particular kind of FSM is known as a monitor or an observer [17]. They are of particular interest for our work, because they are well suited for use with symbolic DCS.

Monitor-based control specifications A monitor for a specification can be realized as an automaton which identifies all behaviors that are not permitted by the specification. Consider a FSM $M = (Q, I, \delta, q_0, PROP, \lambda)$. A monitor automaton of a property p is an automaton M_m whose inputs chosen among $I \cup \lambda$. It "catches" all possible counter-examples where property p is violated during the execution of M.

A monitor is generally created manually, and its construction is not trivial. This manual construction amounts to an actual error-prone design process. In practice, only simple monitors are acceptable, as correct expressions of an informal requirement. The frontier between "simple" and "complex" monitors is totally subjective, and fixed by designer's insight and confidence.

Monitors can be generated from temporal logic formulae. This technique is described in [18].

In the sequel, we highlight a collection of behaviors that are easier to express operationally (by using monitors), than by logical specifications.

Example 2.14. *Forbidden state. Consider an automaton G and a specification which identifies a set of states as forbidden. A monitor of such specification requires only two states I and ERR. The monitor stays at state I as long as the system stays in its admissible states. Any transition from admissible state to forbidden state of the system model triggers a transition from I to ERR. In this kind of specifications, I can be considered as a "macro" state which is composed of all admissible states of system process; while the ERR state is a "macro" state of all forbidden system states. The transition from I to ERR is a conjunction of all system transitions from admissible state to forbidden state.*

Example 2.15. *State splitting. Consider the example shown in Figure 2.13. The specification prevents the occurrence of string $a_1a_2b_2b_1$ and $a_2a_1b_1b_2$. To remember how state 4 in the system model is achieved,*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

we have to split state 4 into two states: state 4 and 9 in the specification. Executions of $a_1a_2b_2b_1$ *and* $a_2a_1b_1b_2$ *are then excluded from the specification. To specify the same property, a monitor automaton can simply includes all possible "dangerous" transitions to the forbidden state.*

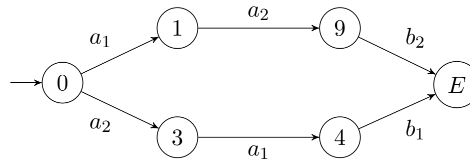
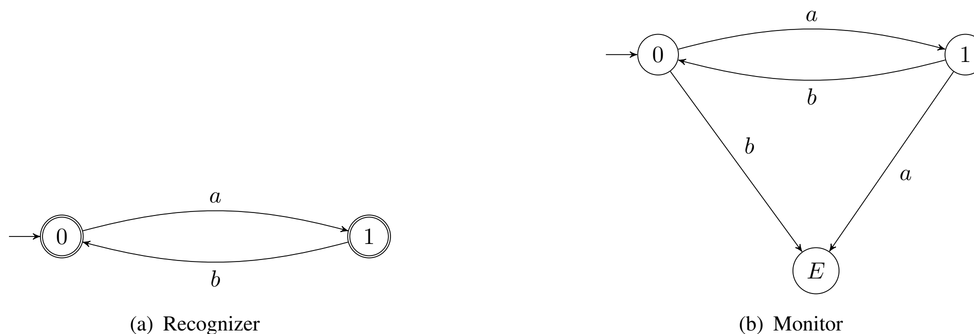


FIGURE 2.13: Monitor of state splitting specification

Example 2.16. *Event alternation. Consider a specification which requires that event* a *and* b *occur alternately, starting by* a*. This specification can be recognized by an automaton shown in Figure 2.14(b). The monitor is shown in Figure 2.14(b).*



(a) Recognizer (b) Monitor

FIGURE 2.14: Monitor for event alternation specification

Example 2.17. *Illegal substring. Figure 2.15 shows a specification expressed by an automaton. This automaton recognize a language which includes all strings in* $\{a, b, c, d\}^*$ *except those contain the substring* $abcd$*. The same specification can also be represented by a monitor shown in Figure 2.16 which traps itself in error state on the occurrence of* $abcd$*.*

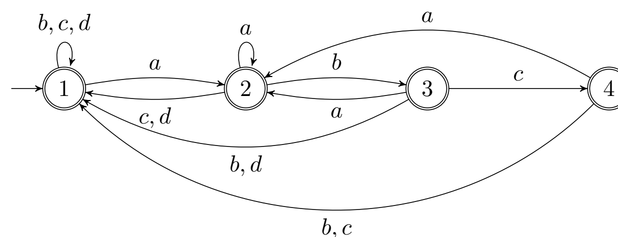


FIGURE 2.15: Recognizer of illegal substring specification

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

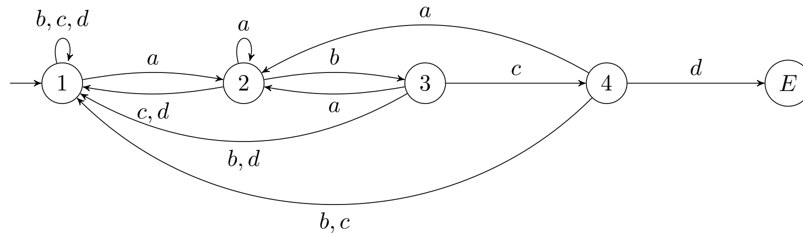


FIGURE 2.16: Monitor of illegal substring specification

2.4 Supervisor synthesis

2.4.1 About controllability

In the hardware design area, the notion of control features some ambiguities. All hardware designs achieve some dedicated computation function, which may involve more or less complex arithmetic operations. Two sorts of building blocks are identified: *controllers* and *operating parts* or *data-paths*. A datapath achieves arithmetic operations; it is a building block featuring inputs, outputs and a sequential behavior, and whose semantics is given by the Boolean FSM model. These operations are managed by the controller, which schedules them according to a global function to be implemented. High-level synthesis [19] is a technique which can automatically generate the controller-datapath (system) pair achieving a required function, which is expressed by an iterative algorithm. This dichotomy is analogue to the closed-loop architecture used in supervisory control: the datapath corresponds to the “plant”, which is the operating part of a system, whose operations are scheduled by the controller, according to the control specification.

In hardware design, the interface between the controller and the operative part is partitioned according to the controllability point of view. Environment inputs are considered *uncontrollable*. The inputs of the operative part are driven by the controller, and thus, are said to be *controllable*. It is important to note that these inputs signify *actions*, that are ordered or prevented by the controller: additions, multiplications, start/stop an engine, move a robot arm, etc. Thus, the distinction between uncontrollable and controllable inputs is conceptually obvious.

Our application context of DCS is different. Sometimes, control functions can have complex specifications, involving concurrent interacting behaviors. They are manually developed by hardware designers. Their production is thus error-prone. We consider such building blocks as systems, which may need additional control in order to enforce their functional correctness. In this context, the notion of controllability is less obvious: *all available inputs are environment inputs*. Thus, controllable inputs need to be chosen among the environment inputs of the design under construction. This yields a target control architecture as represented in Figure 2.17. The input set of the system is partitioned into controllable and uncontrollable inputs: $\mathcal{I} = \mathcal{U} \cup \mathcal{C}$. The details of the DCS algorithm are recalled below.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

2.4.2 Symbolic supervisor synthesis

The symbolic DCS technique we consider has been developed in [3]. It relies on BDD-based symbolic traversal techniques [16]. Its purpose is to enforce the control specification by making invariant, if possible, the set of states of the system where this specification holds. The resulting set of states is known as an invariant under control, which we abbreviate $\mathcal{IUC}$. The resulting supervisor is the set of all system's transitions leading to the set $\mathcal{IUC}$. At each moment, given the system's current state and the values of the uncontrollable variables, the supervisor computes the values of the controllable variables so that only transitions leading to $\mathcal{IUC}$ can be followed. This control architecture is schematically represented in Figure 2.17. This control architecture is studied in detail and refined in Chapter 4.

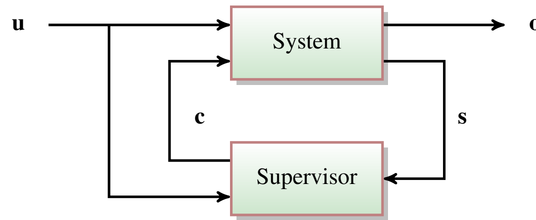


FIGURE 2.17: Schematic control architecture achieved by DCS

The fundamental step in computing $\mathcal{IUC}$ is the construction of the controllable predecessors of a given set of states.

Let $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{C}_{s0}, \mathcal{T}, \mathcal{PROP}, \lambda_B)$ be a symbolic FSM. The controllable predecessors of the set of states $E \subset \mathbb{B}^{|S|}$ of M, having the characteristic function $\mathcal{C}_E$, is defined as follows.

Definition 2.22 (Controllable predecessors of a set of states)**.** A state s is a controllable predecessor of E iff for any uncontrollable value $\mathbf{u}$, there exists a controllable value $\mathbf{c}$, such that the transition defined by $\mathbf{s}$, $\mathbf{u}$, $\mathbf{c}$ leads to a state in E.

$$\mathcal{CPRED}(\mathcal{C}_E, \mathcal{T})(\mathbf{s}) = \forall \mathbf{u}, \exists \mathbf{c}, \exists \mathbf{s}', : \mathcal{T}(\mathbf{s}, \mathbf{u}, \mathbf{c}, \mathbf{s}') \cdot \mathcal{C}_E(\mathbf{s}').$$

Let $spec$ be a control specification, and M be a system modeled by an FSM. The supervisor synthesis follows two steps: computation of the invariant under control $\mathcal{IUC}$ and construction of the supervisor. The $\mathcal{IUC}$ set is obtained by a fix point calculation through recursive applications of the $\mathcal{CPRED}$ operator. The initial point is $\lambda^{-1}(spec)$, the set of states of M satisfying P. The algorithm operates as follows.

If the initial state s_0 of M belongs to $\mathcal{IUC}$, then a supervisor $\mathcal{SUP}$ can be constructed:

$$\mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) = \exists \mathbf{s}' : \mathcal{T}(\mathbf{s}, \mathbf{u}, \mathbf{c}, \mathbf{s}') \cdot \mathcal{IUC}(\mathbf{s}').$$

If $s_0 \notin \mathcal{IUC}$, then it is impossible to enforce the system to stay in $\mathcal{IUC}$ from the initial state, thus no control solution exists.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Algorithm 3 Symbolic computation of $\mathcal{IUC}$

1: {inputs:

- M a symbolic FSM,
- $\mathcal{I}^0 = \lambda^{-1}(spec)$, the set of states satisfying $spec$, to be made invariant

}

2: {starts with transition relation and specification}
3: {intermediate results: $\mathcal{I}^1, \mathcal{I}^2, \cdots$ }
4: $P_0(\mathbf{s}') \leftarrow [P(\mathbf{s})]/(\mathbf{s} \leftarrow \mathbf{s}')$
5: **repeat**
6: $\quad R(\mathbf{s}) \leftarrow \mathcal{CPRED}(P_i(\mathbf{s}'), \mathcal{T})$
7: $\quad \mathcal{I}^{i+1}(\mathbf{s}') \leftarrow [R(\mathbf{s})]/(\mathbf{s} \leftarrow \mathbf{s}') \cdot \mathcal{I}^i(\mathbf{s}')$
8: **until** $\mathcal{I}^{i+1} = \mathcal{I}^i$
9: $\mathcal{IUC} \leftarrow [\mathcal{I}^i(\mathbf{s}')]/(\mathbf{s}' \leftarrow \mathbf{s})$

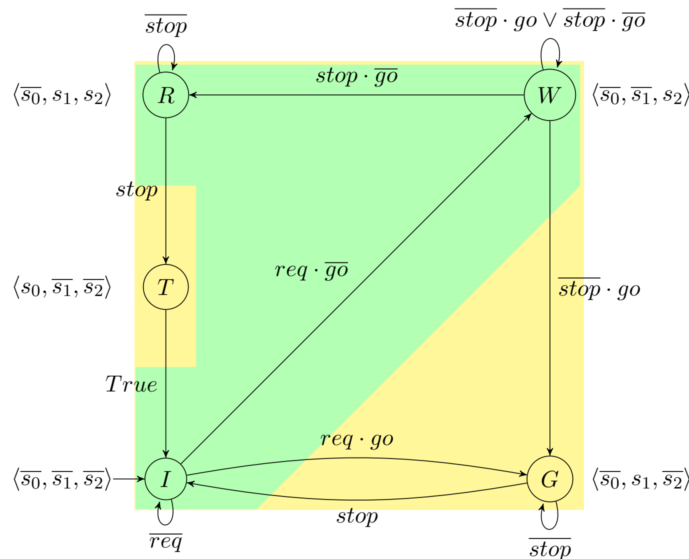


FIGURE 2.18: First iteration

Example 2.18. *In Example 2.13, the CTL formula* $AG(\overline{\overline{s_0} s_1 \overline{s_2} \vee s_0 \overline{s_1} \overline{s_2}})$ *is not satisfied naturally by the system. In fact,* $AG(\overline{\overline{s_0} s_1 \overline{s_2} \vee s_0 \overline{s_1} \overline{s_2}})$ *can be characterized by a fix point calculation. Starting from the initial state set* $S_0 = \{I, W, R\}$ *where the property* $\overline{\overline{s_0} s_1 \overline{s_2} \vee s_0 \overline{s_1} \overline{s_2}}$ *is naturally satisfied, as shown in Figure 2.18. Suppose that events go is controllable, other events are uncontrollable. There exists one transition from state* R *to the outside of* S_0 *which cannot be disabled. Thus state* R *should be excluded from* S_0 *and* $S_1 = \{I, W\}$*. From these two states, there's no uncontrollable transition which leads to the complementary of* S_1*, thus* $\mathcal{IUC} = S_1$*, as shown in Figure 2.19. Since the initial state* $I \in \mathcal{IUC}$*, the specification is controllable. A controller can be extracted by simply disable controllable events from each state in* $\mathcal{IUC}$ *to enforce the system to stay in* $\mathcal{IUC}$*. For this example, the controller only enable event* go *at state* W*.*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

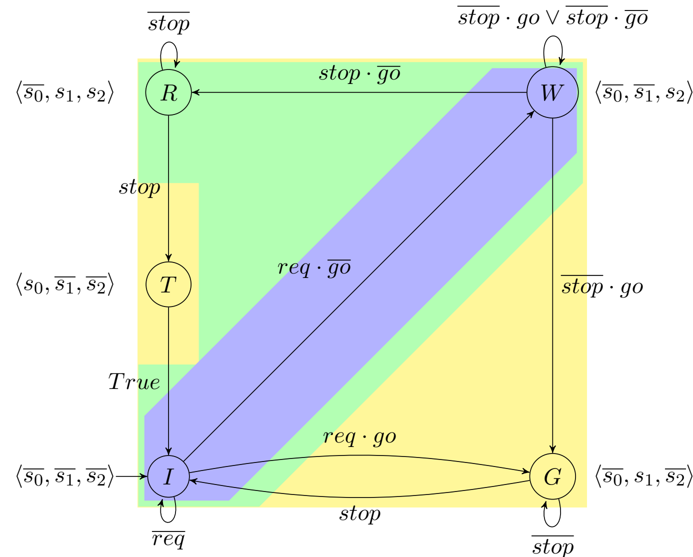


FIGURE 2.19: Second iteration

2.5 Conclusion

This chapter has introduced the notations and notions which are used in the following parts. The introduction of supervisor synthesis is recalled by covering the following aspects: modeling, control specification and supervisor synthesis algorithm.

It should be noted that the DCS algorithm considers all system variables as being observable. Even the results presented in the next chapter rely on the assumption of total observability. Encapsulation is only considered as a design method for achieving modularity.

The specific implementation of the DCS technique we have used, relies on Ternary Decision Diagrams (TDDs), rather than BDDs. However, in this work, TDDs are used exclusively in representing Binary systems, excluding systematically the “third value”. It is important to emphasize that the results presented in the next chapter only apply to the invariant enforcing DCS algorithm presented above. So, they only cover control specifications which can be made invariant.

The Ramadge-Wonham supervisory control theory defines the DCS technique for both invariance and reachability control specifications. Invariance requirements are expressed by operational specifications, and reachability requirements are implied by the presence of a marked states. In this work, we do not address reachability.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Chapter 3

Incremental Discrete Controller synthesis

3.1 Introduction

The efforts made to apply DCS to industrial-size designs have come across the same difficulty: the growing size of the system leads to time and/or memory blow-up during the computation of the supervisor. DCS is implemented on top of symbolic algorithms which have exponential spatial complexity in the number of Boolean variables they manipulate. This complexity is inherent to Binary Decision Diagrams manipulations, which are fundamental in the implementation of symbolic traversal algorithms. Symbolic BDD-based traversal is a generic technique used in solving formally many problems related to Boolean state/transition systems. In particular, symbolic CTL model-checking consists of a collection of symbolic traversal algorithm which produce the set of states where a formula is true, and checks whether the result contains the initial state. DCS problems also benefit from the symbolic traversal technique: as shown in Chapter 2, a supervisor is built (if it exists) as a set of all possible control solutions satisfying the control specification.

3.1.1 Memory evolution during DCS.

It can be observed that during synthesis, a memory usage peak often appears during the calculation of intermediate results. This can be shown by an example. The model is described in detail in section 3.7.2. In the illustration, the synthesis procedure proceeds as follows:

1. identify the set of states which satisfy the control property;
2. compute the invariant under control set. This process is iterative. It computes the controllable predecessors of the set obtained as the previous iteration and continues until a fix point is reached;
3. extract a controller from the invariant under control.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

During each calculation of a controllable predecessor state set, the number of BDDs are recorded. The memory usage evolution in terms of BDD number during the synthesis is shown in Figure 3.1. We start with a small model which has only 4000 BDD nodes. The first predecessor calculation results in a peak which reaches 35000 BDD nodes. The following 4 calculations drop to lower node numbers. The last calculation is the generation of the controller from the $\mathcal{IUC}$, which uses 15000 nodes. From this example, we note that a small scale model can lead to a high memory usage and the peak of memory usage usually appears in the middle of $\mathcal{IUC}$ calculation. This peak of memory usage is due to the exponential worse-case complexity of BDD-based DCS. In general, this peak is critical to the feasibility of DCS for real-life designs. All computers have bounded memory resources, so it is important to tackle the exponential blowup, so that this hard memory boundary is never crossed, or at least delayed.

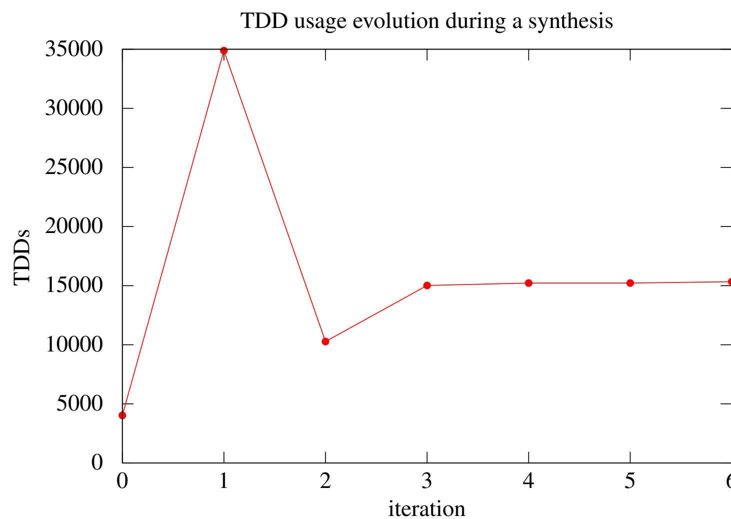


FIGURE 3.1: Peak memory usage during DCS

In order to avoid combinational explosion, an alternative solution has been recently proposed. It advocates the use of SAT-based algorithms. Thus, to establish the truth of a formula, it is enough to search for one satisfying solution, instead of computing the set of all possible solutions. This solution is very efficient in formal verification, but not possible in DCS; by definition, a supervisor should be maximally permissive: it must include all possible control solutions. Thus, symbolic algorithms remain fundamental for DCS.

The method extensively used to overcome combinational explosion during symbolic traversal is decomposition, following the “divide and conquer” approach. In particular, modular decomposition according to the modular structure of a system, is a natural and intuitive approach exploited in several research contributions on DCS. This technique follows two steps: (1) compute an “approximate” control solution for each individual module. This is efficient, as DCS acts locally on each module; (2) recombine approximate local solutions into one exact global solution. As argued in more detail in the “related works” section, the modular DCS techniques we are aware of, apply to event-based models which do not feature explicit communication between modules: the only synchronization mechanism between modules is via controllable/uncontrollable input sharing.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Our contribution is an *incremental*, rather than modular, supervisor synthesis procedure. This technique also takes advantage from the modular structure of the system, and takes into account explicit communication: each module has an input/output interface, and interconnections are possible according to a communication model presented below.

For a system featuring two or more modules, the incremental DCS is an iterative process. It starts with an initial abstraction: all but one modules are abstracted away. They are reincorporated progressively during the following iterations:

- approximate DCS. This constructs an invariant under control set which is an over-approximation (a superset) of the exact invariant under control;
- incremental refinement: add one module to the abstract system, among those previously abstracted.

When the progressive refinement adds the last module, a last DCS step is applied to the whole system and produces an exact solution. This procedure is illustrated in Figure 3.3. In the following, the incremental DCS technique (IDCS) is presented in detail. Two examples are used in order to illustrate this technique. A small example is presented in section 3.4. It features two modules and is used for illustrating all IDCS steps on a simple system. For readability reasons, this example does not implement internal communication between modules. Then, a slightly larger example, presented in section 3.5, illustrates the application of IDCS on a more realistic design involving three communicating modules.

The outline of this chapter is the following:

- the definitions and notations required follow in section 3.2;
- section 3.3 presents the IDCS technique in detail, together with the preliminary definitions of notions used to build the final synthesis algorithm;
- section 3.4 presents a simple running example used to illustrate this technique. Both global and incremental synthesis are applied to this example to illustrate synthesis procedure;
- in section 3.5, the incremental technique is applied to a more complex system with internal communications;
- implementation and performance issues are discussed in section 3.6;
- section 3.7 illustrates the efficiency and advantages of IDCS by a series of experimental results.

3.1.2 Related works

Modular supervisory control is first studied in [20]. In this approach, control specification is split into a series of modular specifications. A set of modular "subcontrollers" are synthesized and applied to the

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

system. The conditions under which the modular synthesis is applicable are also identified. Since, a number of methods have been proposed to reduce computation and/or memory efforts.

Generally speaking, the synthesized supervisor contains not only the constraints required by the specification but also redundant information such as transition constraints which are already enforced by the system. The reduction of such redundancy from supervisors can simplify implementation. This reduction is first studied in [21]. In [22], authors proposed a polynomial-time algorithm which significantly reduces supervisor size while preserves control properties. This kind of supervisor reduction requires that a supervisor is synthesized beforehand, thus it may be more useful for the supervisor implementation but not for synthesis. The above supervisor reduction technique can be applied to modular supervisor synthesis to improve efficiency, but it also requires to build modular supervisors first.

In [23] authors apply abstraction techniques for modular and decentralized control. This technique relies on the assumption that the specification is given on a subset of the system alphabet, and the system behavior is reduced to this alphabet. Supervisors are computed for each reduced subsystem system alphabet by employing both modular and decentralized approach. However, the supervisors for the original system are not guaranteed to be maximally permissive. Further more, the requirements on the specifications are not always satisfied.

In [24, 25], language projection-based abstraction is used to coordinate modular subsystems but not for synthesizing supervisor itself. Coordination is realized in a hierarchical structure. Low level modular supervisors are computed conventionally without concerns of non-conflict property; High level coordination supervisor is synthesized based on the abstraction of subsystems. Compared to our method, this approach is still based on modular control paradigm. Abstraction is applied only on the supervised subsystem modules for a high level coordination.

An abstraction based on automata rather than on language projections was proposed in [26] in order to preserve non-blocking properties. This abstraction technique is based on the event set. The idea is to map an automaton to a non-deterministic automaton which is defined on a smaller alphabet. However, in this abstraction technique, the selection of events to be abstracted is not obvious. Furthermore, the reduction of state numbers with this event-based abstraction is not guaranteed to be sufficient.

In [27, 28], the authors present a framework for compositional synthesis, using abstractions based on a process equivalence called supervision equivalence. The idea is to simplify modules while preserving all information necessary for the synthesis. Local uncontrollable events are substituted by one identity-less uncontrollable event. The resulted supervisor in a compact representation by using non-deterministic intermediate automata, while it is still guaranteed to be least restrictive and non-blocking. This work is similar to ours in that both approaches try to reduce intermediate model by using abstraction techniques. But this work deals with local uncontrollable events while our approach exploit state-based abstraction which is potential to achieve more abstraction and is compatible to incremental synthesis.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

In [29], authors proposed an approach which first efficiently determine if the control problem has a solution without building the whole system model. It then generates an *over-approximation* of the least restrictive control solution. An equivalence of non-deterministic abstract processes, called synthesis equivalence, is proposed. This condition is coarser than both bisimulation equivalence and supervision equivalence.

Authors of [30] follows the modular control scheme and proposed an incremental procedure for generating a set of modular supervisors that are non-conflicting by construction. They show that the conjunction of the modular supervisors satisfy given specifications without blocking. The abstraction is applied to the modular sub-controlled components by projecting away strictly private events and specifications. Supervisory synthesis is achieved in an incremental bottom-up manner until all specifications are satisfied. However, the resulting behavior is not guaranteed to be optimal. The efficiency of this technique relies strongly on the local property of specifications. It has little improvement if the specifications address every module in the system.

In [31], modular supervisors are built; potential conflicts between modular supervisors are solved by a set of coordinating filters. Conflict-equivalent abstractions are employed to detect conflict. Non-determinism introduced by the abstraction is resolved by employing a static state-feedback approach. In [30] [31], abstractions are used for reducing complexity of the verification of the non-conflict property of the modular supervisors, but not for the computation of the supervisor. Their abstraction is based on language projection.

A modular technique based on concurrent automata decomposition is presented in [32, 33]. The global supervisor is obtained by treating each automaton of a modular composition separately and composing local approximate supervisors together. The automata are supposed to share input events, but they do not communicate, i.e. no outputs of one automaton are connected to the inputs of another automaton.

The techniques enumerated above mostly exploit qualitative properties of the system, the specification or the supervisor: modular composition, locality of input events, locality of the specifications, behavioral equivalence, modularity of the supervisor, etc.

In [34], authors develop a controller synthesis procedure based on game theory. Contracts are used as assumptions during the synthesis process. This method is further extended to a modular (compositional) approach in [35], where contracts on the sub-modules are considered as assumptions to handle the complexity of the global model. These techniques operate efficiently on systems composed of communicating modules and they implement the "assume-guarantee" compositional approach. The efficiency of this technique is undeniable in terms of complexity reduction. However, assumptions are to be developed manually, and it is hard to determine how many assumptions are needed in order to give a realistic abstract model of a module. Besides, we believe that these techniques produce control solutions which are not guaranteed to be maximally permissive. Indeed, for each module, the computation of the control solution quantifies universally the local inputs driven by the outputs of neighbor modules. Thus, the

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

solution is built for *any* neighboring environment, and is thus potentially more restrictive than the one computed by “brute force” DCS.

The IDCS approach we propose is incremental, but not compositional. Compositional techniques produce local solutions which are assembled in order to obtain the final, global control result. The IDCS technique exploits the modularity of a design but builds the final control solution incrementally. Moreover, the final step in IDCS is a systematic application of DCS, which benefits from the previous incremental steps in order to converge faster. The performance issues related to this technique are discussed in detail in section 3.7.

On the other hand, IDCS supports FSM hierarchy with *communicating* modules. The techniques recalled above handle communication through synchronizing input events: two or more modules can synchronize by awaiting the arrival of the same input event. Instead, IDCS supports “producer-consumer” communication: outputs of one module can be “fed” to inputs of its neighbors. This communication mechanism is fundamental in hardware design. Finally, IDCS produces an exact DCS result, which is maximally permissive.

3.2 Definitions

In the following, we define the notion of communicating controllable FSM, based on the Boolean sample-driven FSM model with outputs, presented in Chapter 2.

3.2.1 Controllable modular finite state machines (CFSM)

The FSM model introduced in Chapter 2 is not sufficient to reflect the communication between modules. An attribute called “interface inputs” is added to the FSM model.

A controllable modular Boolean FSM (CFSM) M is defined as a tuple:

$$M = (Q, I, L, \delta, q_0, PROP, \lambda),$$

such that:

- Q : a finite set of n states $\{q_1, q_2, \ldots, q_n\}$;
- I : a set of Boolean input variables, such that $I = U \cup C$ and $U \cap C = \emptyset$:
  - $U = \{u_1, \cdots, u_m\}, m \geq 0$: the set of uncontrollable input variables;
  - $C = \{c_1, \cdots, c_p\}, p \geq 0$: the set of controllable input variables;
- $L = \{l_1, \cdots, l_r\}, r \geq 0$: a set of interface inputs;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- $\delta : Q \times \mathbb{B}^{m+p+r} \to Q$ is the transition function of M;
- $q_0 \in Q$ is the initial state of M;
- $PROP = \{p_1, \cdots, p_v\}, v \geq 0$: a set of v Boolean atomic propositions;
- $\lambda : Q \to \mathbb{B}^v$ is a labeling function.

Remark The functions λ and λ^{-1} are identical to those defined in Chapter 2.

In the following, we note $\mathbf{u}, \mathbf{c}, \mathbf{l}$, the vectors containing the variables of U, C, L respectively.

3.2.2 Synchronous composition

The synchronous product between two communicating controllable FSMs is defined with respect to the DCS problem to be solved. A system M features an input/output interface. Inputs are either controllable or uncontrollable. However, M can have an *internal* hierarchic structure: it can be composed of two or more communicating modules. Each of them may exhibit controllable and/or uncontrollable inputs, but also *interface* inputs dedicated to internal communication with other modules of M. This particular structure is illustrated in Figure 3.2.

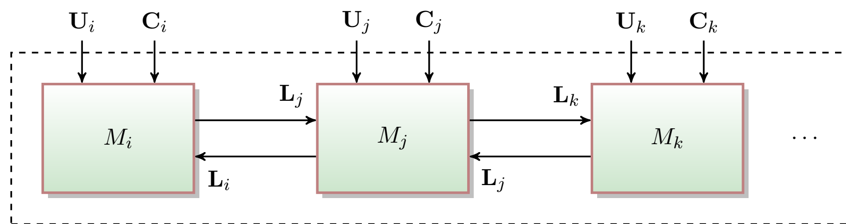


FIGURE 3.2: Modular structure of a controllable system

Let us note $M = ||_{i=1}^{K} M_i$ the synchronous composition between K communicating modules $M_i = (Q_i, I_i, L_i, \delta_i, q_{0i}, PROP_i, \lambda_i), i = 1, \cdots, K$. In the following, we assume that an output cannot be driven by more than one CFSM inside a synchronous product:

$$\forall i, j = 1, \cdots, K, i \neq j : PROP_i \cap PROP_j = \emptyset.$$

We also assume that the sets of controllable and uncontrollable variables are coherent with respect to the synchronous composition: if an input is shared between M_i and M_j, it cannot be controllable in M_i and uncontrollable in M_j.

According to the actual hierarchical structure of M, the set of interface inputs of module M_i can be partitioned into $K-1$ disjoint subsets $\{L_i^j | j = 1, \ldots, K, j \neq i\}$ with

$$L_i^k \cap L_i^j = \emptyset, (\forall k, j = 1, \cdots, K, j \neq k)$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

and

$$\bigcup_{j=1,\cdots,K,j\neq i} L_i^j = L_i$$

and

$$0 \leq |L_i^j| \leq |L_i|.$$

The set of interface inputs L_i^j belongs to M_i and is connected to outputs of M_j.

Let $PROP_i^{out_j} \subseteq PROP_i$ be the subset of output propositions of M_i which are connected to module M_j through L_j^i. Similarly, the output functions λ_i of M_i are partitioned according to their connectivity. Let $(\lambda_i^{j,k}), 0 \leq k \leq |L_j^i|$ be the outputs of M_i connected to M_j.

The synchronous product between CFSMs $M = ||_{i=1}^K M_i$ is defined as follows:

- $Q = Q_1 \times \cdots \times Q_K$;
- $I = I_1 \cup \cdots \cup I_K$, such that $I_i = U_i \cup C_i$;
- $L = \emptyset$;
- $\delta : Q \times \mathbb{B}^{m_1+p_1+r_1} \times \cdots \times \mathbb{B}^{m_K+p_K+r_K} \to Q$ is defined as:

$$\begin{aligned}\delta = \Big(&\delta_1\big(q_1, \mathbf{i_1}, && \lambda_2^{1,1}(q_2), \cdots, \lambda_2^{1,|L_1^2|}(q_2),\\ & && \lambda_3^{1,1}(q_3), \cdots, \lambda_3^{1,|L_1^3|}(q_3),\\ & && \cdots\big),\\ &\cdots,\\ &\delta_K\big(q_K, \mathbf{i_K}, && \lambda_1^{K,1}(q_1), \cdots, \lambda_1^{K,|L_K^1|}(q_1),\\ & && \lambda_2^{K,1}(q_2), \cdots, \lambda_2^{K,|L_K^2|}(q_2),\\ & && \cdots\big)\\ \Big);\end{aligned}$$

- $q_0 = (q_{01}, \cdots, q_{0K})$;
- $PROP = PROP_1 \cup \cdots \cup PROP_K$;
- $\lambda : Q \to \mathbb{B}^{\sum_{i=1}^K |PROP_i|}$ is defined as:

$$\lambda\big((q_1, \cdots, q_K)\big) = \big(\lambda_1(q_1), \cdots, \lambda_K(q_K)\big).$$

The definition and properties of the reverse function λ^{-1} remain unchanged.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

In the following, we first define the incremental DCS procedure, and then illustrate each of its steps with on a simple two-way arbiter example in section 3.4. The results of IDCS are compared to those obtained by direct "brute force" DCS application.

3.3 Incremental DCS technique

The incremental (IDCS) procedure operates on the same inputs as the direct DCS technique, but requires a supplementary user-specified indication: an ordering between the modules which constitute the global system. IDCS works iteratively: for a system containing $K \geq 2$ modules, it requires K steps, one for each module, with $K!$ possible application orders. We argue that an ordering between modules can be user specified, and determined according to the structure and the connectivity of the global system.

Just like direct DCS, IDCS enforces the assertion $AG(spec)$, where $spec$ is a Boolean proposition expressed over the set $PROP_1 \cup \cdots \cup PROP_K$ of M by using the classical Boolean connectors " $\cdot$ " (conjunction), " $+$ " (disjunction) and " $\overline{p}$ " (negation). The incremental DCS acts as follows.

First, it builds an abstract model $M_1||M_2^{abs}||\cdots||M_K^{abs}$. The abstraction replaces $M_2, \cdots, M_K$ by a most permissive abstract FSM model, which models all the possible behaviors for those outputs initially driven by $M_2, \cdots, M_K$, and having an influence either on the behavior of M_1 or on the satisfaction of $spec$. Such an abstract FSM model for a given variable x is a two-state non-deterministic automaton, having all transitions enabled and asserting $x = 0$ or $x = 1$, according to its current state. Modules $M_2^{abs}, \cdots, M_K^{abs}$ are said to be an *loose*, i.e. nonrestrictive environment for M_1, in the sense that it allows more behaviors of their outputs than $M_2, \cdots, M_K$ actually do.

The number of abstract FSMs obtained depends on the number of output variables shared with M_1 and $spec$. For m shared variables, the loose environment $M_i^{abs}, i = 2, \cdots, K$ models all possible values of these variables, and thus has 2^m states. Thus, the gain achieved through abstraction strongly depends on the connectivity between M_i with M_1 and $spec$, and the concrete model of M_i itself. The abstraction operation is formalized in the next section.

Second, an intermediate, approximate control solution $\mathcal{IUC}_1^{abs}$ is synthesized for $spec$ on the abstract system model $M_1||M_2^{abs}\cdots M_K^{abs}$. This is achieved by applying DCS and has a lower computation cost, because it operates on a smaller model, depending on the size of the abstraction previously achieved.

Finally, the abstraction is partially refined: M_2^{abs} is replaced by M_2. The intermediate result $\mathcal{IUC}_1^{abs}$ is used as the starting point for synthesizing a new control solution for $spec$ and $M_1||M_2||M_3^{abs}\cdots||M_K^{abs}$. Modules $M_3 \cdots M_K$ are added progressively, and successive intermediate results are computed.

The last step operates on the whole system model. It is expected to benefit from the computations achieved at the previous steps on the abstract model. The IDCS technique is illustrated in Figure 3.3 for $K = 2$, by comparison to the direct application of DCS.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

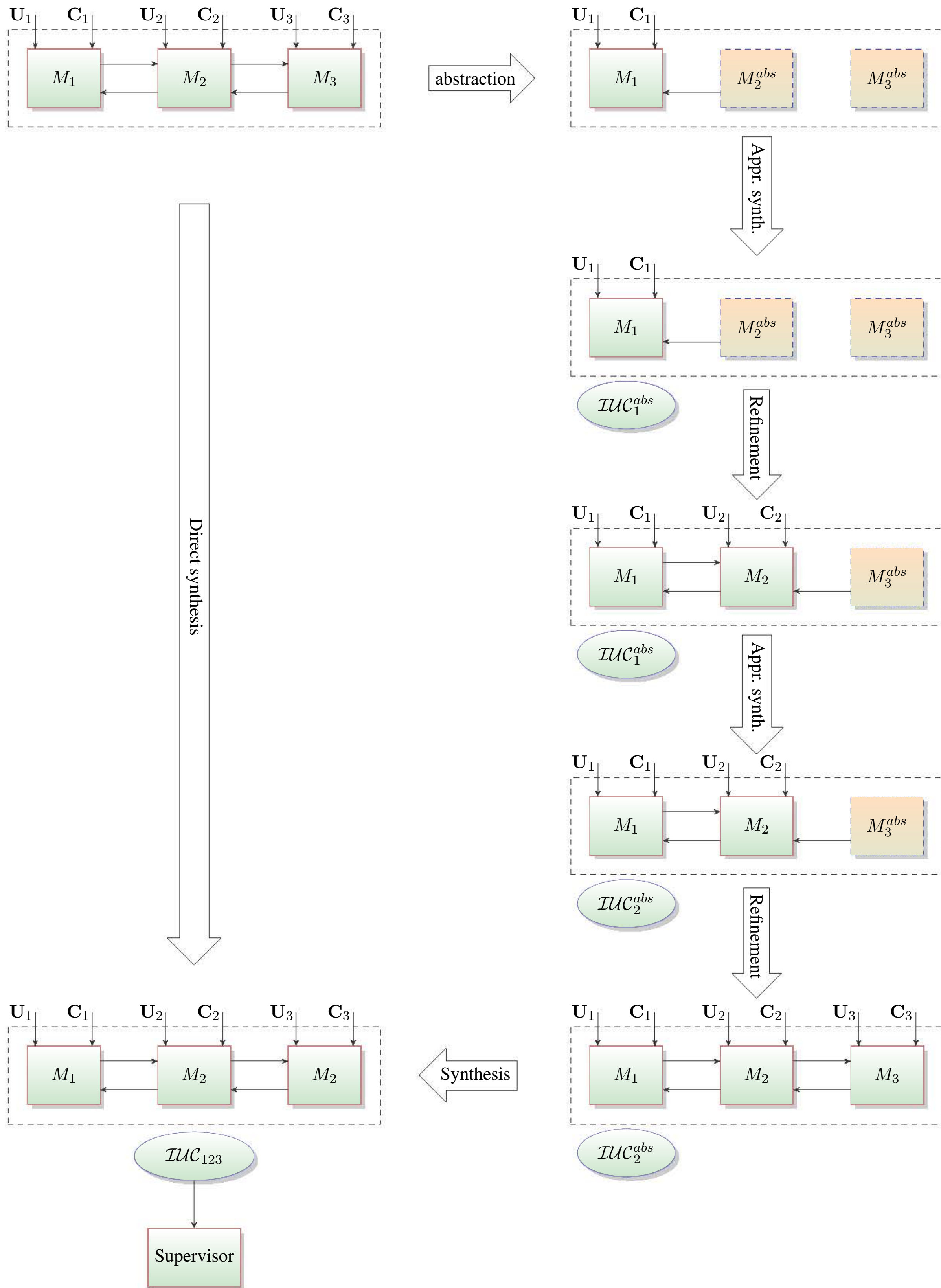


FIGURE 3.3: Incremental DCS technique versus direct DCS

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

3.3.1 Abstraction

Given a finite state machine M, and a corresponding Boolean proposition $p \in PROP$. The abstraction of M with respect to p is a non-deterministic FSM whose states satisfy either p or $\overline{p}$:

$$abs(M,p) = (Q^{abs}, I^{abs}, L^{abs}, \delta^{abs}, Q^{abs}, PROP^{abs}, \lambda^{abs})$$

where

- $Q^{abs} = \{q_p, \tilde{q_p}\}$, $I^{abs} = \emptyset$ and $L^{abs} = \emptyset$;
- $\delta^{abs} : Q^{abs} \to 2^{Q^{abs}}$ is defined as: $\delta^{abs}(q_p) = \delta^{abs}(\tilde{q_p}) = Q^{abs}$;
- the initial state can be any among Q^{abs};
- $PROP^{abs} = \{p\}$;
- $\lambda^{abs} : Q^{abs} \to \mathbb{B}$ is defined as:
  $\lambda^{abs}(q_p) = 1$ and $\lambda^{abs}(\tilde{q_p}) = 0$.

The abstraction of M with respect to a subset $PROP^m \subset PROP$, with $1 \leq m \leq |PROP|$ is defined as the synchronous product of the individual abstractions defined on $p_i \in PROP^m$:

$$Abs(M, PROP^m) = ||_{i=1,\cdots,m}\ abs(M, p_i).$$

This operation abstracts away all the behaviors of M, and produces an FSM model which reproduces all the 2^m possible assertions for the propositions of $PROP^m$ as well as all the possible sequences of assertions.

As shown below, IDCS applies the abstraction operation iteratively. For a system $M = ||_{i=1}^{K} M_i$, each iteration j of the incremental DCS technique abstracts modules $M_j, \cdots, M_K$ according to their connectivity to modules $M_1, \cdots, M_{j-1}$ and to the atomic propositions $SPEC_j \in PROP_j$ used to express $spec$. This operation is defined as:

$$\mathcal{ABS}(M, j, spec) = ||_{k=1}^{j-1} M_k || \big(||_{k=j}^{K} Abs(M_k, \bigcup_{l=1}^{k-1} PROP_k^{out_l} \cup SPEC_k) \big).$$

3.3.2 Abstract set refinement

The abstract set refinement operation projects a set of abstract states back on the original states previously abstracted away by $\mathcal{ABS}$. The abstract state refinement with respect to module M_j is a function which

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

projects a subset of states Q^{abs_j} of $\mathcal{ABS}(M, j, spec)$ to the set $Q^{abs_{j+1}}$ of $\mathcal{ABS}(M, j+1, spec)$ with respect to the satisfaction of a set of predicates PROP.

It is defined as:

$$ref(q^{abs_j}, PROP, Q^{abs_{j+1}}) = \{q^{abs_{j+1}} \in Q^{abs_{j+1}} | \forall p \in \text{PROP}, \\ q^{abs_j} \in \lambda^{-1abs_j}(p) \rightarrow q^{abs_{j+1}} \in \lambda^{-1abs_{j+1}}(p)\},$$

where $\lambda^{-1abs_j}(p)$ and $\lambda^{-1abs_{j+1}}(p)$ are the reverse labeling functions of $\mathcal{ABS}(M, j, spec)$ and $\mathcal{ABS}(M, j+1, spec)$ respectively.

The refinement of an abstract set of states Q^{abs} is defined as:

$$Ref(Q^{abs_j}, PROP, Q^{abs_{j+1}}) = \bigcup\{ref_j(q^{abs_j}, PROP, Q^{abs_{j+1}}) | q^{abs} \in Q^{abs}\}.$$

3.3.3 The Incremental DCS algorithm

The IDCS algorithm starts with the description of M and the specification $spec$ to be enforced. It also requires an ordering between the modules of $M = M_1 || M_2 || \cdots || M_K$, which is to be applied during the successive refinement steps. At each iteration j, an abstraction $\mathcal{IUC}(M, j, spec)$ is computed with respect to:

- the outputs of $M_{j+1} \cdots M_K$ which are connected to local inputs of $M_1 \cdots M_j$;
- the set of atomic propositions of $M_{j+1} \cdots M_K$ used to express $spec$.

Algorithm 4 presents the IDCS algorithm. Each iteration j computes an approximate control solution on an abstract system $\mathcal{ABS}(M, j, spec)$. Each iteration produces an intermediate result used at the next iteration. At the end, a final DCS step is performed on the whole system.

The performance and implementation details of the IDCS technique are commented in detail in section 3.6. In the following, we establish that the IDCS algorithm also produces a maximally permissive supervisor.

Theorem 3.1. *The DCS and IDCS algorithms produce the same result.*

Proof. The above statement is true iff the incremental and direct synthesis algorithms produce the same invariant under control sets.

Let us recall that for any DCS problem, we have $\mathcal{IUC} \subseteq \lambda^{-1}(spec)$.

Let $\mathcal{IUC}$ be the result produced by direct DCS and $\mathcal{IUC}^{inc}$ be the result produced by the IDCS algorithm. It can be observed that $\mathcal{IUC}_{inc} \subseteq \mathcal{IUC}$. Indeed, the last step of IDCS operates on the whole system M;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Algorithm 4 IDCS algorithm

1: {inputs:

- $M = M_1||M_2|| \cdots ||M_K, (K \geq 2)$, the system to be controlled;
- $spec$ the specification to enforce;

output: $\mathcal{IUC}^{inc}$, the invariant under control set for M and $spec$ }
2: $M^{abs} \leftarrow \mathcal{ABS}(M, 2, spec)$
3: $\mathcal{I} \leftarrow \lambda^{-1abs}(spec)$
4: $\mathcal{I} \leftarrow \mathcal{IUC}(M^{abs}, \mathcal{I})$
5: $j \leftarrow 3$
6: **while** $\mathcal{I} \neq \emptyset$ **and** $j \leq K$ **do**
7: $M^{abs} \leftarrow \mathcal{ABS}(M, j, spec)$
8: $\mathcal{I} \leftarrow Ref(\mathcal{I}, \{spec\}, Q^{abs})$
9: $\mathcal{I} \leftarrow \mathcal{IUC}(M^{abs}, \mathcal{I})$
10: $j \leftarrow j + 1$
11: **end while**
12: **if** $\mathcal{I} \neq \emptyset$ **then**
13: {The last invariant under control is projected back on the states Q of M:}
14: $\mathcal{I} \leftarrow Ref(\mathcal{I}, \{spec\}, Q)$
15: $\mathcal{IUC}^{inc} \leftarrow \mathcal{IUC}(M, \mathcal{I})$
16: **end if**

it attempts to make invariant the set $\mathcal{I}$, refined at iteration $j = K$ with respect to $spec$. By construction, all these states are included in Q^M and satisfy $spec$. Thus, $\mathcal{I} \subseteq \lambda^{-1}(spec)$. The last DCS application computes $\mathcal{IUC}^{inc} \subseteq \mathcal{I}$ by making invariant the set $\mathcal{I}$.

So, on the one hand, direct DCS makes invariant the set $\lambda^{-1}(spec)$ on M and produces the set $\mathcal{IUC} \subseteq \lambda^{-1}(spec)$. On the other hand, the last step of IDCS starts with a subset of $\lambda^{-1}(spec)$: it makes invariant the set $\mathcal{I} \subseteq \lambda^{-1}(spec)$ on M and produces $\mathcal{IUC}^{inc}$.

Now, let us consider the state $q \in \mathcal{IUC}^{inc}$; q is contained in $\lambda^{-1}(spec)$. Assume that q is not an element of $\mathcal{IUC}$. This means that direct DCS has pruned the state q on M. The last step of IDCS performs an ordinary direct DCS operation on M, making invariant the set of states $\mathcal{I}$, containing q. Thus, state q should also be pruned by IDCS and shouldn't be included in $\mathcal{IUC}^{inc}$. So it is true that $\forall q \in Q^M : q \in \mathcal{IUC}^{inc} \implies q \in \mathcal{IUC}$. We can conclude that $\mathcal{IUC}^{inc} \subseteq \mathcal{IUC}$.

Let us now assume that the above inclusion is strict. This means that there exists at least one state $q \in Q^M$ such that $q \in \mathcal{IUC}$ and $q \notin \mathcal{IUC}^{inc}$. State q has been pruned by the incremental DCS algorithm: there exists a transition leaving q and leading out of $\mathcal{IUC}^{inc}$ for a given uncontrollable value and for any value taken by the controllable variables. However, if such a transition exists, state q would also be pruned by DCS from $\mathcal{IUC}$. Hence, we conclude that $\mathcal{IUC}^{inc} = \mathcal{IUC}$.

□

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

3.4 First example : an arbiter model

The presented example is inspired from [36]. It models an arbiter which manages the exclusive access to a shared resource for two independent clients. This model only focuses on the access management feature. The actual shared resource as well as its interactions with each client are left unmodeled.

The access management is modeled by two concurrent state machines, $M1$ and $M2$, as shown in Fig. 3.4. Each machine M_i, $(i = \{1, 2\})$, receives a request req_i from a client and issues a corresponding access grant gnt_i to the client. When it receives an access request req_i, M_i either enters a *waiting* state W_i or into a *granting* state G_i. Each client has the possibility to retract its request, by issuing $stop_i$ before they are granted. The access grant decision is taken upon the value of inputs go_i ; each machine M_i can stay at *waiting* state as long as go_i is *false* and as long as the requesting client does not retract its request, by issuing $stop_i$. At state G_i, M_i issues gnt_i to its requester. Access remains granted for client i until it rises the signal $stop_i$ and returns to the *idle* state I_i.

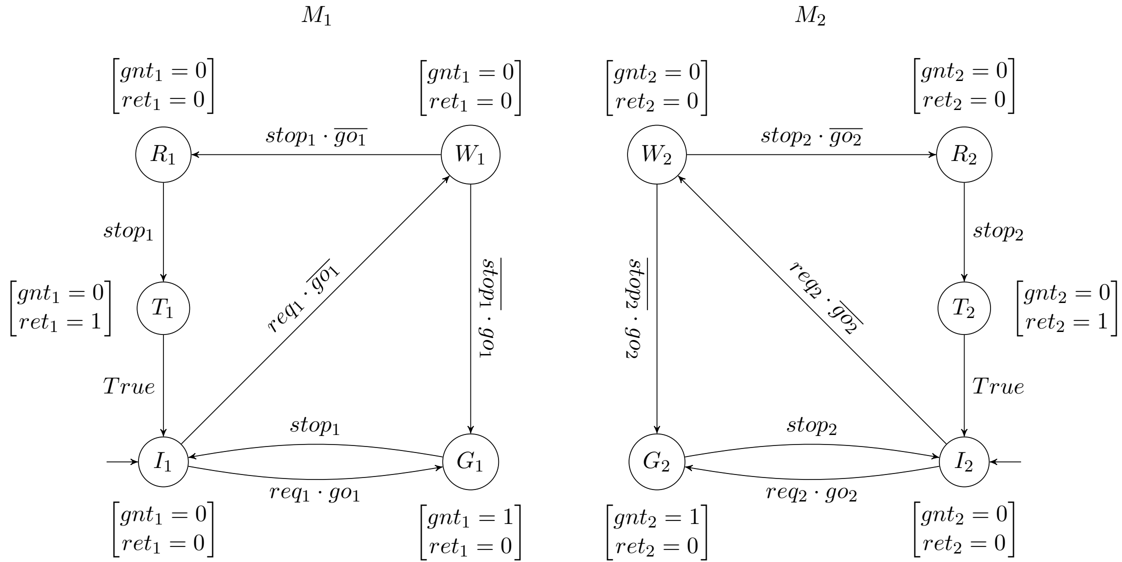


FIGURE 3.4: The arbiter model

The expected access grant policy is a *mutual exclusion* : machines $M_i, (i = \{1, 2\})$ should never emit their gnt_i signal at the same moment. This requirement can be expressed by the proposition:

$$spec_1 : \overline{gnt_1 \cdot gnt_2}.$$

Moreover, it is assumed that the retract mechanism has not been implemented yet, and thus it is required that the retract states T_i be forbidden:

$$spec_2 : \overline{ret_1 + ret_2}.$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

The conjunction of three requirements are expressed by the proposition:

$$spec : \overline{gnt_1 \cdot gnt_2} \cdot \overline{ret_1} \cdot \overline{ret_2}.$$

The proposition *spec* must be made invariant over the whole arbiter model. This is formally specified in CTL [37] as

$$enforce : \ AG(spec).$$

According to our example, input variables $\{req_1, req_2, stop_1, stop_2\}$ are uncontrollable variables, meant to be assigned by the clients. Input variables $\{go_1, go_2\}$ are controllable. This simple example does not feature interface variables. The only global synchronization between M_1 and M_2 is achieved through the satisfaction of *spec*, as shown below.

The sets of Boolean propositions associated to $M_i, i = 1, 2$ are $PROP_i = \{gnt_i, ret_i\}$. The labeling functions λ_i associate these atomic propositions to the states of M_i:

$$\lambda_i^{gnt_i, ret_i}(I_i) = (0, 0),$$
$$\lambda_i^{gnt_i, ret_i}(G_i) = (1, 0),$$
$$\ldots$$

Conversely, the set of states of M_i satisfying proposition $gnt_i \cdot \overline{ret_i}$ is given by:

$$\begin{aligned} \lambda_i^{-1}(gnt_i \cdot \overline{ret_i}) &= \lambda_i^{-1}(gnt_i) \cap \lambda_i^{-1}(\overline{ret_i}) \\ &= \{G_i\} \cap Q_i \setminus \{T_i\} = \{G_i\}. \end{aligned}$$

A supervisor needs to be built by DCS in order to implement the access grant policy *spec* by adequately assigning values to the controllable variables.

3.4.1 Applying global DCS to the arbiter model

With direct global synthesis, we first build the synchronous composition M_{12} of M_1 and M_2. To illustrate the reverse labeling function λ_{12}^{-1}, let us apply it to the Boolean proposition *spec* we wish to enforce. We obtain the following results:

$$\lambda_{12}^{-1}(\overline{gnt_1 \cdot gnt_2} \cdot \overline{ret_1} \cdot \overline{ret_2}) =$$
$$\lambda_{12}^{-1}(\overline{gnt_1 \cdot gnt_2}) \cap \lambda_{12}^{-1}(\overline{ret_1}) \cap \lambda_{12}^{-1}(\overline{ret_2}).$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

By successively applying the properties cited in section 2.2.3 and the definition of λ_{12} we obtain:

$$\begin{aligned} \lambda_{12}^{-1}(\overline{gnt_1 \cdot gnt_2}) &= \\ Q_1 \times Q_2 \setminus \lambda_{12}^{-1}(gnt_1 \cdot gnt_2) &= \\ Q_1 \times Q_2 \setminus (\{G_1\} \times Q_2 \cap Q_1 \times \{G_2\}) &= \\ Q_1 \times Q_2 \setminus \{G_1\} \times \{G_2\}. \end{aligned}$$

A similar application of the same calculus leads to the global result :

$$\begin{aligned} \lambda_{12}^{-1}(\overline{gnt_1 \cdot gnt_2} \cdot \overline{ret_1} \cdot \overline{ret_2}) &= \\ Q_1 \times Q_2 \setminus (\{G_1\} \times \{G_2\} \cup \{T_1\} \times Q_2 \cup Q_1 \times \{T_2\}). \end{aligned}$$

The DCS algorithm starts with the set of states satisfying *spec* : $\mathcal{I}^0 = \lambda_{12}^{-1}(spec)$. The invariant under control set $\mathcal{IUC}$ with respect to *spec* contains all the states of $M_1||M_2$ such that for any uncontrollable tuple $(req_1, req_2, stop_1, stop_2) \in \mathbb{B}^4$ there always exist $(go_1, go_2) \in \mathbb{B}^2$ such that the transition obtained leads to $\mathcal{IUC}$. Thus, we obtain an invariant under control set which prunes states R_1, T_1, R_2, T_2 as well as the global state $\{G_1\} \times \{G_2\}$ as shown in Figure 3.5.

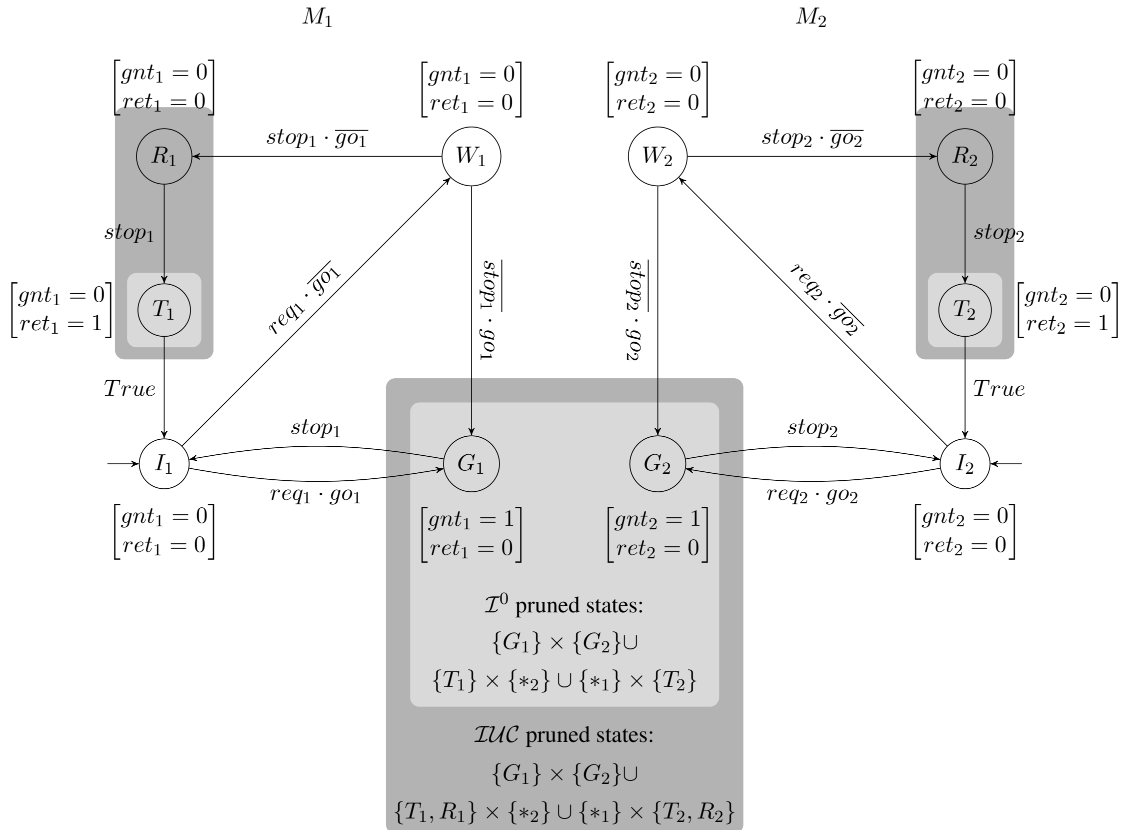


FIGURE 3.5: Global DCS for $M_1||M_2$ and *spec*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

The corresponding supervisor is represented below :

$$\begin{aligned}
(I_1, I_2) &\Rightarrow (\overline{req_1 \cdot go_1} + \overline{req_2 + go_2})+ \\
(I_1, W_2) &\Rightarrow (\overline{req_1 \cdot go_1} \cdot \overline{stop_2} + stop_2 \cdot go_2 + \overline{stop_2} \cdot \overline{go_2})+ \\
(I_1, G_2) &\Rightarrow (\overline{req_1 \cdot go_1} + stop_2)+ \\
(W_1, I_2) &\Rightarrow (\overline{go_1} + \overline{req_2 \cdot go_2})+ \\
(W_1, W_2) &\Rightarrow (\overline{go_1} + \overline{go_2})+ \\
(W_1, G_2) &\Rightarrow (\overline{go_1} + stop_2)+ \\
(G_1, I_2) &\Rightarrow (stop_1 + \overline{req_2 \cdot go_2})+ \\
(G_1, W_2) &\Rightarrow (stop_1 + \overline{go_2}).
\end{aligned}$$

It represents all $M_1||M_2$ transitions leading to $\mathcal{IUC}$. For instance, in state (I_1, I_2), if req_1 and req_2 are both active, the supervisor allows either $go_1 = 1$ or $go_2 = 1$ but not both.

3.4.2 Applying IDCS to the arbiter model

In the following, $\{*_i\}$ abbreviates the set of all states of a given model M_i.

3.4.2.1 Abstraction

Let us abstract M_2 and replace it with M_2^{abs}. The abstraction rule applied to M_2 concerns its output variables shared with M_1 and $spec$. We have $PROP_2^{out} = \emptyset$ and $SPEC_2 = \{gnt_2, ret_2\}$. Figure 3.6 represents the abstract model obtained. For readability reasons, abstract states are directly labeled with their corresponding Boolean proposition. Note that model M_2^{abs} is entirely non-deterministic: at any moment, all transitions are enabled.

3.4.2.2 Approximate $\mathcal{IUC}$ computation

The approximate invariant under control for property *spec* is built upon the abstract model $M_1||M_2^{abs}$. As shown in Figure 3.6, $\mathcal{IUC}^{abs}$ prunes the following states : $\{G_1\} \times \{gnt_2\}$, R_1, T_1 and the abstract state ret_2.

3.4.2.3 Refinement and final synthesis

We replace M_2^{abs} by M_2 and refine the states pruned from $\mathcal{IUC}^{abs}$ on the states of $M_1||M_2$ with respect to $spec$. This is done by applying $Ref(\mathcal{IUC}^{abs}, spec, Q^{M_1||M_2})$. Thus, the state in M_2 satisfying $gnt_2 =$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

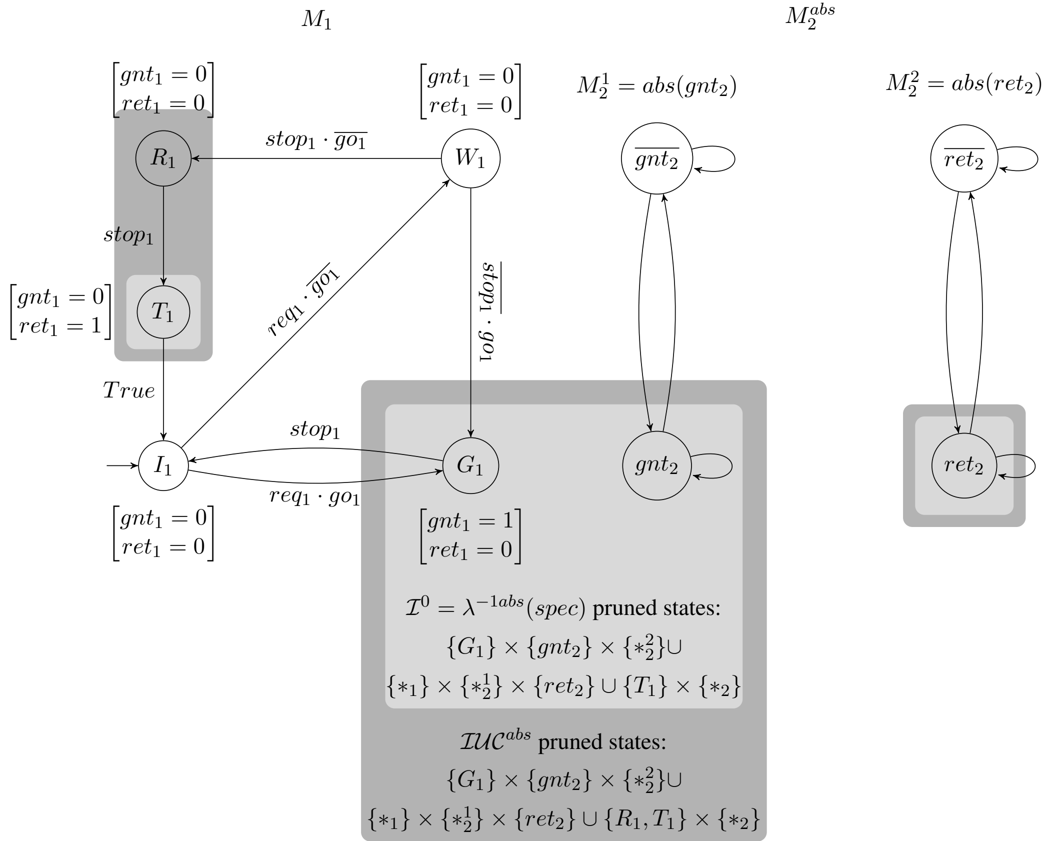


FIGURE 3.6: Abstract model $M_1||M_2^{abs}$ and the approximate $\mathcal{IUC}^{abs}$ computation

$1 \wedge ret_2 = 0$ is G_2, and the state satisfying $ret_2 = 1$ is $\{T_2\}$. The refined set obtained is shown in Figure 3.7.

The computation of the final supervisor tries to make invariant the set $Ref(\mathcal{IUC}^{abs}, spec, Q^{M_1||M_2})$. This last step prunes state R_2 from the final solution, as shown in Figure 3.7. This final result is exactly the same as the one obtained by direct DCS.

This example illustrates the basic principles of the incremental synthesis technique. However, the two components shown above have no explicit communications between them. Their synchronization is forced by the satisfaction of the global specification $spec$.

3.5 Second example: a 3-way arbiter with tokens

This is a slightly more complex system featuring communication between its components. It implements an arbiter which manages the exclusive access to a shared resource for three independent clients. The model presented below only focuses on the access management feature. The actual shared resource as well as its interactions with each client are left unmodeled.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

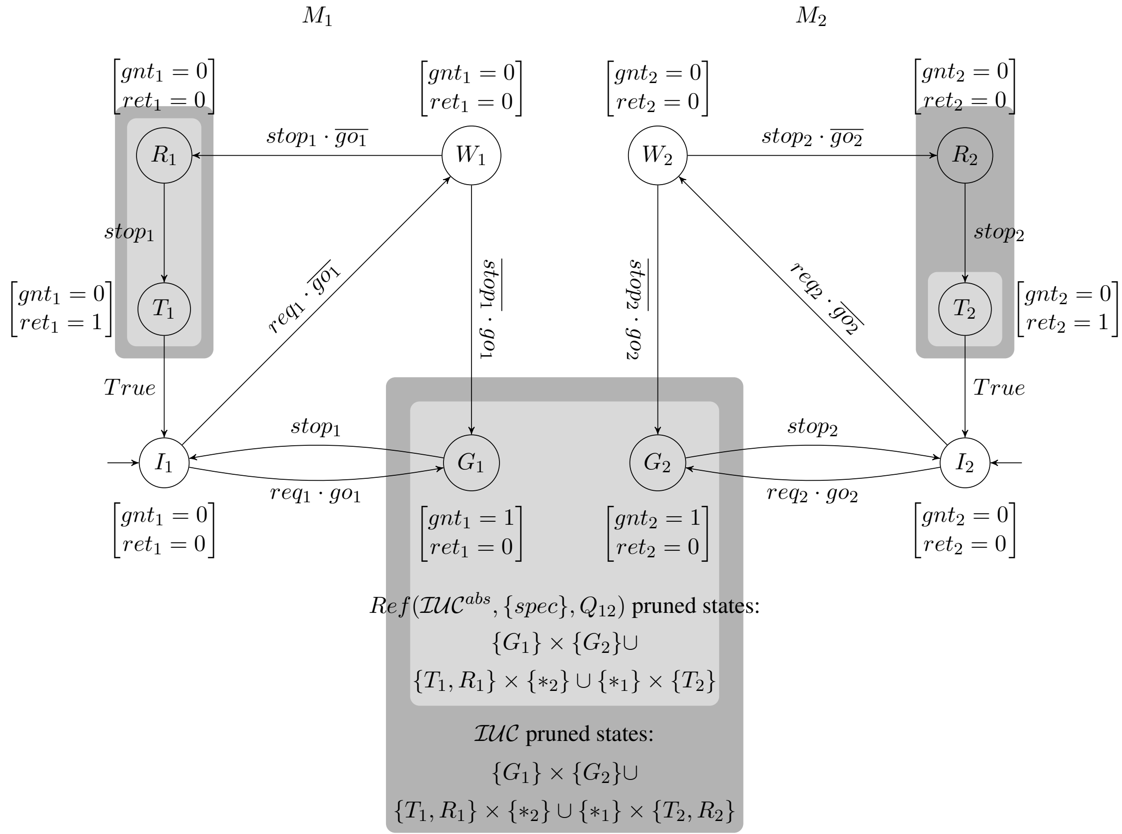


FIGURE 3.7: Final global DCS step ensuring *spec*

The access management is modeled by three concurrent cells, $M_i, (i = \{1, 2, 3\})$. They share identical function and structure. Each cell i receives a request req_i from its client and issues a corresponding access grant ack_i to that client. The access control is enforced by a token mechanism. A cell may acknowledge its client only if it holds the token.

Cells M_i are modeled by two automata: M_i^1 receives the token. M_i^2 implements the access grant to the shared resource according to the availability of the token. It also forwards the token, once it has been effectively used. The automata of a single cell are shown in Figure 3.8. M_i^1 and M_i^2 communicate via the signal go_i, which is set to 1 whenever a token has been received and 0 otherwise. The state N_i means "no token received". State T_i means "a token has been received". State I_i is an idle state, waiting for a client request. State G_i is an active state, where a client has just been granted access and the token is forwarded.

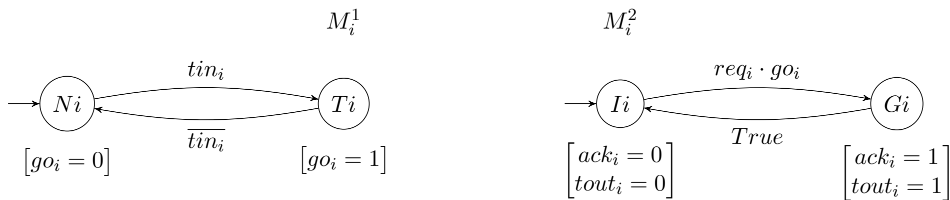


FIGURE 3.8: Model of a single cell i

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

The 3-way arbiter is constructed by instantiating the three cells $M_i, i = 1, 2, 3$ and by connecting their interface and output variables, as shown in Figure 3.9. Tokens are fed to M by the environment through input tin_1.

According to the synchronous composition definition for communicating CFSMs, modules M_i have both environment inputs and local inputs. Environment inputs are dedicated to communication with components situated outside M. Local inputs are used for modeling communication between the modules M_i of M. Here we have $L_1 = \emptyset$, $L_2 = \{tin_2\}$, $L_3 = \{tin_3\}$.

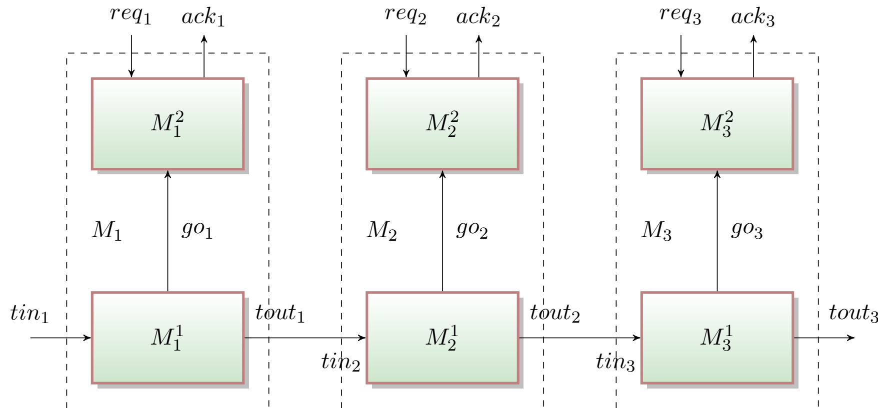


FIGURE 3.9: Internal connection of the arbiter with token

The expected access grant policy is *mutual exclusion* : machines M_i should never emit their ack_i signal at the same moment. This requirement can be expressed by the following predicate:

$$spec : \overline{ack_1 \cdot ack_2 + ack_2 \cdot ack_3 + ack_1 \cdot ack_3}$$

3.5.1 Defining the DCS problem

Modules M_i are building blocks for designing a correct 3-way arbiter. They are assembled according to Figure 3.9. On the resulting model, the token mechanism is only partially defined. A token can be inserted from the environment, via input tin_1 which is set to 1. Once inserted, the token is passed from a cell to the following as soon as it is used. However, in order to satisfy $spec$, there should never be more than one token present inside the arbiter. Hence, the value of tin_1 cannot be chosen at random. It should not be set to 1 until no token is present inside M.

We attempt to make invariant proposition $spec = 1$ by applying IDCS. As tin_1 seems crucial for ensuring token unicity inside M we chose it as *controllable*. The incoming requests $req_i, i = 1, 2, 3$ can arrive at any moment, and so they are considered *uncontrollable*.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

So the resulting supervisor should control M via its token tin_1 input, according to the uncontrollable inputs $\{req_1, req_2, req_3\}$ and the resulting controlled arbiter should satisfy the CTL [37] property:

$$enforce: \ AG(spec).$$

3.5.2 Applying IDCS to the 3-way arbiter

The first step consists of computing $\lambda_{123}^{-1}(spec)$. We obtain the following results:

$$\begin{aligned}\lambda_{123}^{-1}(\overline{ack_1 \cdot ack_2 + ack_2 \cdot ack_3 + ack_1 \cdot ack_3}) &= \\ \lambda_{123}^{-1}(\overline{ack_1 \cdot ack_2} \cdot \overline{ack_2 \cdot ack_3} \cdot \overline{ack_1 \cdot ack_3}) &= \\ \lambda_{123}^{-1}(\overline{ack_1 \cdot ack_2}) \cap \lambda_{123}^{-1}(\overline{ack_2 \cdot ack_3}) \cap \lambda_{123}^{-1}(\overline{ack_1 \cdot ack_3}).\end{aligned}$$

By successively applying the properties cited in section 2.2.3 and the definition of λ_{123}^{-1} we obtain:

$$\begin{aligned}\lambda_{123}^{-1}(\overline{ack_1 \cdot ack_2}) &= \\ Q_1 \times Q_2 \times Q_3 \setminus \lambda_{123}^{-1}(ack_1 \cdot ack_2) &= \\ Q_1 \times Q_2 \times Q_3 \setminus (\{*_1^1\} \times \{G_1\} \times Q_2 \times Q_3 \cap Q_1 \times \{*_2^1\} \times \{G_2\} \times Q_3) &= \\ Q_1 \times Q_2 \times Q_3 \setminus (Q_1^1 \times \{G_1\} \times Q_2 \times Q_3 \cap Q_1 \times Q_2^1 \times \{G_2\} \times Q_3).\end{aligned}$$

A similar application of the same calculus leads to the global result :

$$\begin{aligned}\lambda_{123}^{-1}(\overline{ack_1 \cdot ack_2 + ack_2 \cdot ack_3 + ack_1 \cdot ack_3}) &= \\ Q_1 \times Q_2 \times Q_3 \setminus (Q_1^1 \times \{G_1\} \times Q_2 \times Q_3 \cap Q_1 \times Q_2^1 \times \{G_2\} \times Q_3 & \\ \cup Q_1 \times Q_2^1 \times \{G_2\} \times Q_3 \cap Q_1 \times Q_2 \times Q_3^1 \times \{G_3\} & \\ \cup Q_1^1 \times \{G_1\} \times Q_2 \times Q_3 \cap Q_1 \times Q_2 \times Q_3^1 \times \{G_3\}).\end{aligned}$$

3.5.2.1 Abstraction

Several abstraction-refinement possibilities exist according to the user-defined ordering. One possibility is to start with the abstract model $M_1||M_2^{abs}||M_3^{abs}$ and incrementally replace M_2^{abs} with M_2 and M_3^{abs} with M_3. Here, for more simplicity, we start with the abstract model $(M_1||M_2)||M_3^{abs}$, given by $\mathcal{ABS}(M, 3, spec)$, as shown in Figure 3.10.

The abstraction operation applied to M_3 concerns its output variables shared with $M_1||M_2$ and $spec$. We have $PROP_3^{out} = \{ack_3\}$ and $SPEC_3 = \{ack_3\}$. Note that ack_3 and $tout_3$ always have identical values. Figure 3.11 represents the abstract model obtained. For readability reasons, abstract states are directly labeled with their corresponding Boolean proposition.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

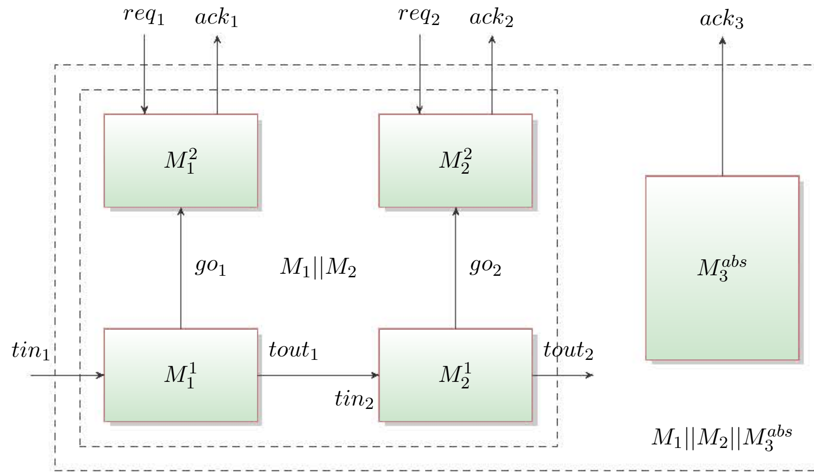


FIGURE 3.10: Abstraction sequence for the 3-way arbiter

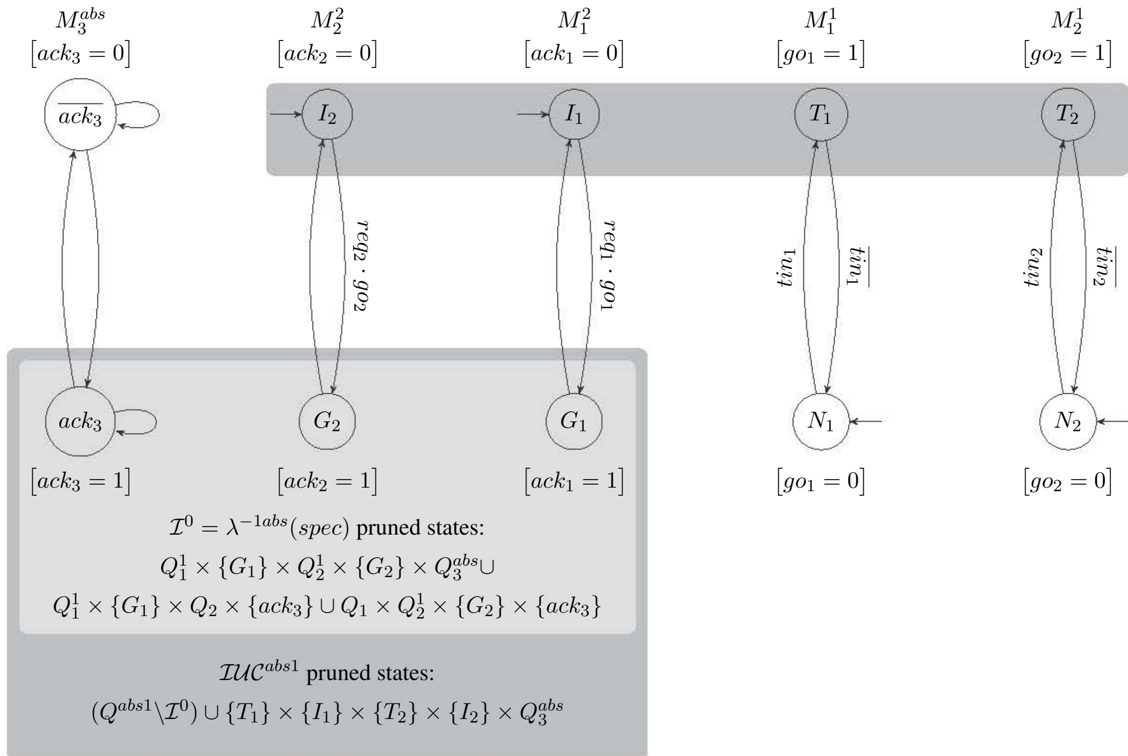


FIGURE 3.11: Abstract model $M_1||M_2||M_3^{abs}$ and the approximate $\mathcal{IUC}$ computation

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

3.5.2.2 Approximate $\mathcal{IUC}$ computation.

The approximate invariant under control for property *spec* is built upon the abstract model $M^{abs1} = M_1||M_2||M_3^{abs}$. As shown in Figure 3.11. The computation first identifies the initial set of states to be pruned $\lambda^{-1abs1}(spec)$. The first iteration leads to the $\mathcal{IUC}^{abs1}$ which prunes the following states from the set $Q_1 \times Q_2 \times Q_3^{abs}$:

$$\mathcal{IUC}^{abs1} = Q_1 \times Q_2 \times Q_3^{abs} \setminus Q_1^1 \times \{G_1\} \times Q_2^1 \times \{G_2\} \times Q_3^{abs} \cup Q_1^1 \times \{G_1\} \times Q_2 \times \{ack_3\} \cup Q_1 \times Q_2^1 \times \{G_2\} \times \{ack_3\} \cup \{T_1\} \times \{I_1\} \times \{T_2\} \times \{I_2\} \times Q_3^{abs}.$$

3.5.2.3 Refinement

The abstract model M^{abs1} is refined by replacing M_3^{abs} with M_3, and by projecting $\mathcal{IUC}^{abs1}$ to the states $Q = Q_1 \times Q_2 \times Q_3$ as shown in Figure 3.12.

The approximate invariant under control $\mathcal{IUC}^{abs1}$ calculated from the previous step is then projected back on this model:

$$\mathcal{I}^0 = Ref(\mathcal{IUC}^{abs1}, \{spec\}, Q)$$

For example, the abstract state set $Q_1^1 \times \{G_1\} \times Q_2 \times \{ack_3\}$ is refined to $Q_1^1 \times \{G_1\} \times Q_2 \times Q_3^1 \times \{G_3\}$ as shown in Figure 3.12.

3.5.2.4 Global synthesis and final results

The final IDCS result is obtained by applying a last DCS step on the global model, using as a starting point the result obtained during the iterative process. Here, we calculate the final invariant under control $\mathcal{IUC}$, starting from $\mathcal{I}^0$, as shown in Figure 3.12.

After this step we obtain:

$$\mathcal{IUC} = Q_1 \times Q_2 \times Q_3 \setminus \{T_1\} \times \{I_1\} \times Q_2 \times \{T_3\} \times \{I_3\} \cup Q_1 \times \{T_2\} \times \{I_2\} \times \{T_3\} \times \{I_3\} \cup \{\{T_1\} \times \{G_1\} \times \{T_2\} \times \{I_2\} \times \{N_3\} \times \{I_3\}, \{T_1\} \times \{I_1\} \times \{T_2\} \times \{G_2\} \times \{N_3\} \times \{I_3\}\}.$$

The construction of the final supervisor from $\mathcal{IUC}$ is straightforward. By simulation of the global controlled system, it can be seen that the resulting supervisor controls the input tin_1 as follows. When cell

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

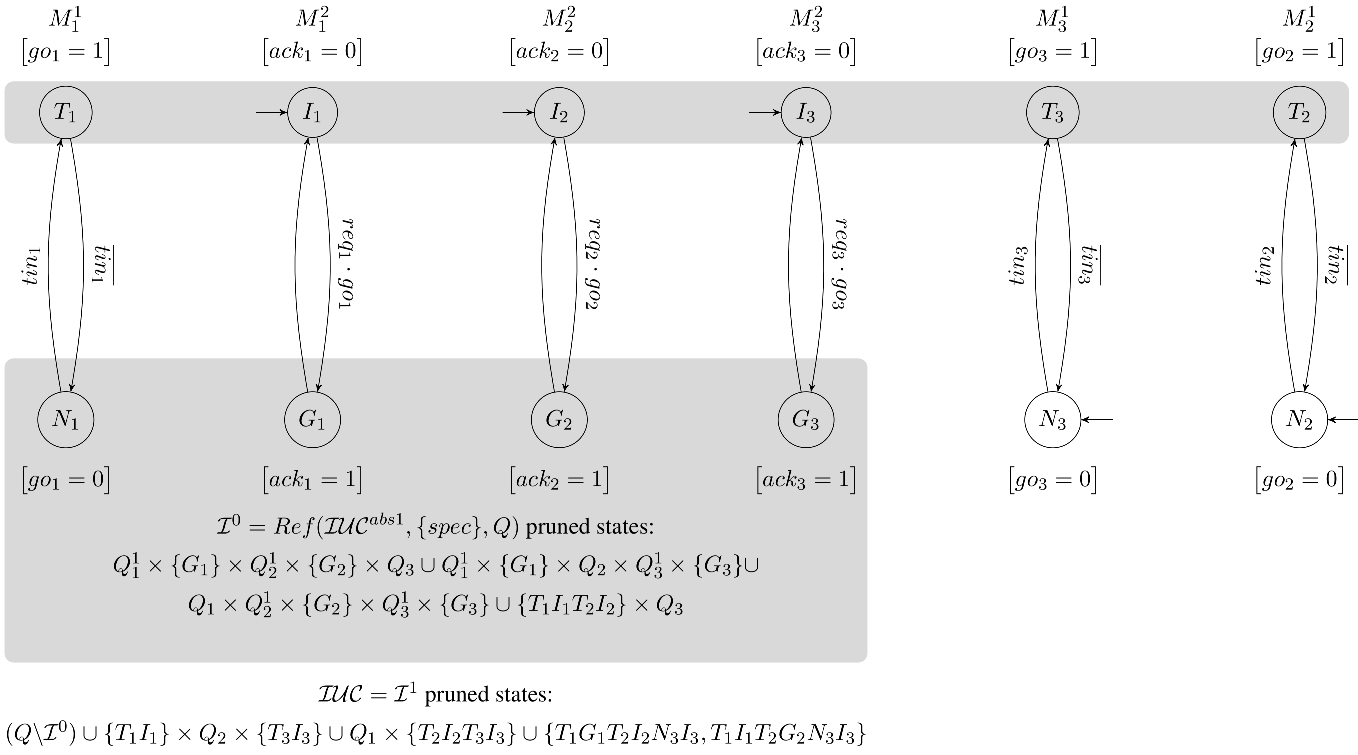


FIGURE 3.12: Refinement of $M_1||M_2||M_3^{abs}$ and the $\mathcal{IUC}$ computation

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

3 has acknowledged an incoming request ($ack_3 = 1$), the token held is forwarded through output $tout_3$, and thus, is lost. No more tokens are present inside M. At this moment, the supervisor allows the insertion of a new token by the environment at any moment. This is done by assigning $tin_1 = 1$. Once a token has been inserted, it remains inside M, going from M_1 to M_2 and then to M_3, until the next acknowledgment on ack_3 is done. During this time, the supervisor disables the insertion of a new token. Indeed, if a new token is inserted, then more than one cell can hold a token at a given moment, and thus, two or more clients can be acknowledged at the same time.

However, M cannot be directly composed with its supervisor. This happens because supervisors generated by symbolic DCS are characteristic functions (i.e. equations), which cannot by manipulated as components, having inputs and outputs. Hence, each supervisor needs to be *implemented* before it can be "plugged" to the system to be controlled. This issue is developed in the next chapter. The final hardware architecture of the controlled 3-way arbiter is shown and commented in Figure 4.8.

3.6 Implementation and performance issues

The performance of the IDCS algorithm strongly depends on the actual implementation of the underlying DCS technique. The technical possibilities available are *enumerative DCS* and *symbolic BDD-based DCS*.

Enumerative DCS techniques represent explicitly the set of states of the system. The complexity of the DCS is $O(n|\Sigma|)$ where n is the total number of states of the system composed to its specification, and $|\Sigma|$ represents the size of the input alphabet.

Symbolic BDD-based DCS manipulates sets of states, rather than individual states, and uses binary decision diagrams (BDDs) [5] to represent them. The performance of this technique is promising, but remains bound by the spatial complexity for constructing a BDD, which is exponential in the number of Boolean variables representing the system. Hence, memory is a critical computing resource for symbolic DCS.

Regardless of the underlying DCS technique used, the computation of an abstract $\mathcal{IUC}$ is definitely faster, as it operates on a reduced model. The speedup mainly depends on the structural decomposition achieved and thus on the final size of the abstract model obtained. However, the final application of the global DCS starting with an intermediate solution produced incrementally shall still feature the same theoretic complexity as classical DCS, plus the overhead generated by the computation of the abstract solution.

However, the ability of symbolic DCS to efficiently manipulate state sets instead of individual states is an important advantage. Hence, we choose to build the IDCS algorithm on top of the symbolic DCS technique developed in [38], which is the only symbolic DCS platform suitable and available. Despite the

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

theoretical bounds recalled above, we expect an *average* performance improvement of symbolic IDCS over symbolic DCS for the following reasons.

First, the computation of the abstract $\mathcal{IUC}$ operates on the reduced model $\mathcal{ABS}(M, j, spec)$. As abstraction removes most states from M, the impact on the size of the BDDs built for the symbolic traversal of $\mathcal{ABS}(M, j, spec)$ state space is indubitable. Second, the computation of $\mathcal{I}$ produces an *intermediate, approximate* solution of the DCS problem. It relies on a more compact BDD representation, as it is built over an abstract system containing less variables. Moreover, the set $\mathcal{I}$ is a subset of $\lambda^{-1}(spec)$. Thus, a number of states of $\mathcal{ABS}(M, j, spec)$ are pruned at a lower computation cost and need not be reconsidered anymore during subsequent DCS applications. Thus, we expect the final DCS step to converge faster (with less fix-point iterations) towards the final solution. Besides, if a DCS problem does not have a solution, this can be detected on the abstract system at a much lower cost.

Still, these performance considerations cannot be theoretically generalized. We can expect bad results for some problems for which there is no actual efficient BDD representation, reaching the exponential performance bound. Typically, systems featuring arithmetic operations such as bit-level multiplications are known to have intractable BDD representations. Besides, the execution of symbolic DCS requires additional fine tuning related to BDD manipulations. First, global and incremental DCS can benefit a lot from a good initial choice of the system variable ordering. Indeed, some Boolean expressions can have either a very compact BDD representation, or a huge one, depending on the variable order chosen to build their BDD. Generally, finding such a good order is done heuristically, most often by exploiting the knowledge of the structural properties of the system. The optimization problem of finding the best variable ordering is known to be NP-complete. Second, the performance of BDD manipulation can be improved by applying *dynamic variable reordering*. This is a fundamental tool implemented by most BDD packages. At run-time, it achieves local permutations between connected variables, which also gives *average* memory improvements, though it is computationally intensive. Again, the usage of this technique can give significant improvements, but can also lead to bad performance results. Its application requires mature expertise from the designer.

In conclusion, IDCS can be used as a tool to leverage the complexity of DCS, in combination with static and dynamic variable ordering. Moreover this technique can be easily automated.

The performance figures of symbolic IDCS over symbolic DCS are presented in section 3.7.

3.7 Experimental results and performance figures

To evaluate the performance improvement, both direct and incremental DCS have been achieved using the SIGALI [3] symbolic supervisor synthesis tool. A number of typical DES control problems have been studied. For each of the control problems, global and incremental synthesis have been applied separately. In order to conserve access to detailed memory profiling figures, the iterations of IDCS have been applied

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

manually. In most of our examples, the application of one incremental iteration followed by the last DCS step seems enough for obtaining interesting gains. For most examples, the abstraction/refinement order has been chosen at random. This choice is justified for those models which contain identical components. For the others, we have rather abstracted away in priority those modules which did not share variables with the control specification. However, this “rule” only makes the intermediate results more readable. Finding “good” abstraction/refinement orders is left as a future research direction.

For each example, memory usage evolution and CPU occupation have been recorded. In order to validate the correction of the IDCS implementation, a systematic formal proof has been applied to guarantee that the two supervisors obtained by DCS and IDCS are identical. All performance figures are presented in section 3.7.6.

3.7.1 A more complex bus arbiter with token

Functional description. This example is first presented in [16] as a showcase for symbolic model-checking. This bus arbiter is intended to manager the access of a number of clients to the bus. Figure 3.13 shows the internal structure of one single cell which serves only one single client’s requests. Each cell has a request input from its client and an acknowledgment output to its client. The entire working bus arbiter is a network connecting a number of single cells, as shown in Figure 3.14.

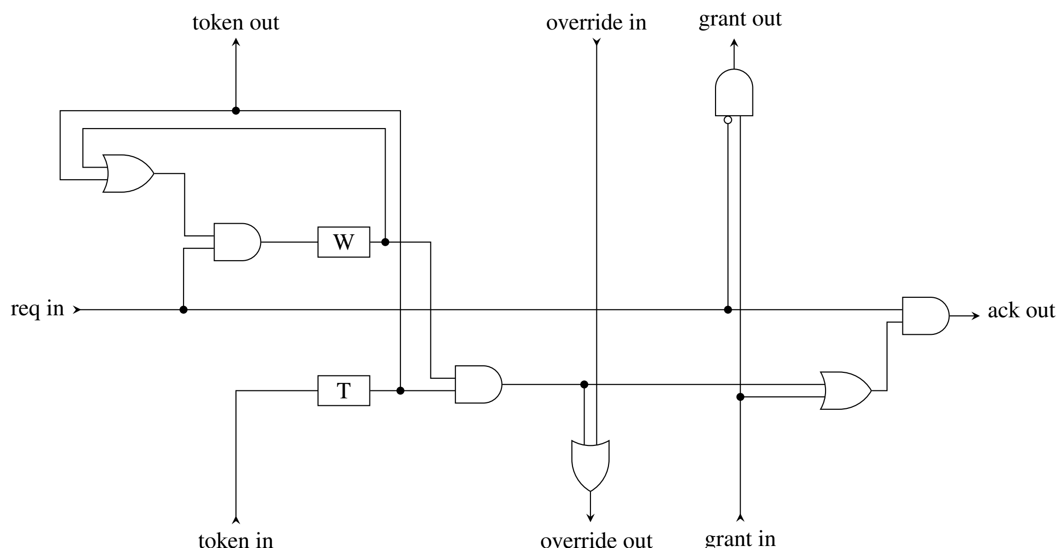


FIGURE 3.13: Single cell of bus arbiter

The entire bus arbiter works in two modes:

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

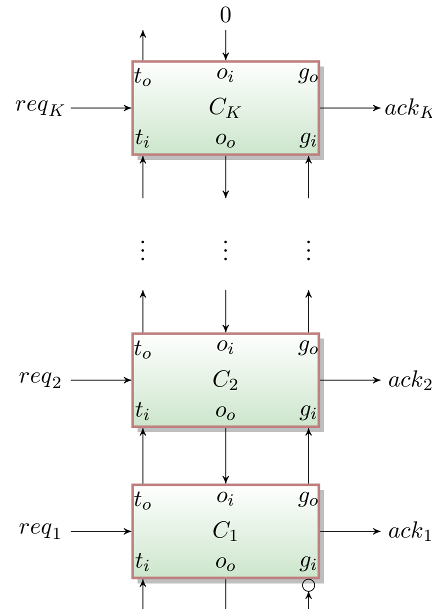


FIGURE 3.14: Token structure of bus arbiter chain

- *priority mode*: the arbiter asserts the acknowledge signal of the requesting client with the lowest index;
- *round-robin mode*: every requesting client is eventually acknowledged by circulating a token in a ring of arbiter cells. The token is passed once every clock cycle. Only the requesting client with the token will be granted access to the bus.

The k-th cell gives the grant to the next $k + 1$-th cell when it has no request from its client and it has the grant from its predecessor $k - 1$-th cell. The "grant output" signal of the k-th cell means that no clients of index less than or equal to k are requesting. Thus the priority mode could be achieved.

On the other hand, each cell has two registers: T and W stand for "token" and "waiting" respectively. When the token is passed in, the T register is set to "1", otherwise, it is reset to "0". In the entire bus arbiter, the T registers of all cells form a circular shift register. The token is shifted up one place every clock cycle.

Register W is set to "1" when the client asserts the request input and the token is present. It remains set as long as the request persists.

A cell asserts its acknowledge output only in two cases:

- the client is requesting and the cell has the grant from its predecessor;
- the client is requesting, and the client is already waiting, and the token is returned to this cell. In this case, the "override out" is set and is passed backwards to its predecessor cells til cell 0 which resets the "grant out" signal and passes forwards so that the "round-robin" policy replaces the "priority" policy.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

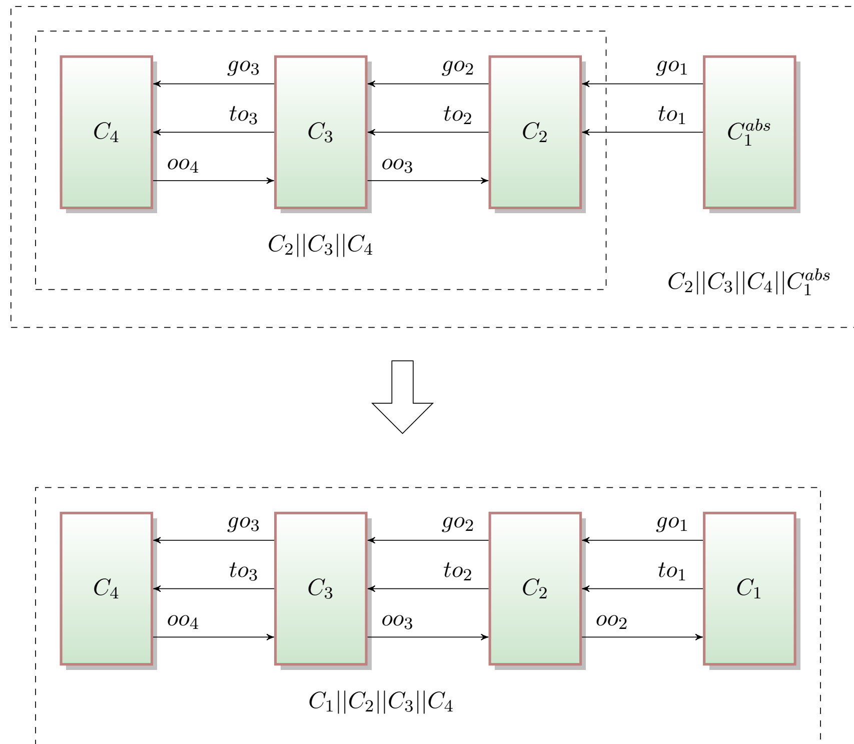


FIGURE 3.15: Incremental synthesis procedure

The bus arbiter is initialized so that all the W registers are reset to "0" and only one T register is set to "1". This initial configuration leads the arbiter to operate in "priority" mode as long as the serving requests are not many. When the requests become frequent and one of the cell finally enters the "waiting" stage, the arbiter will fall back to the "round-robin" policy to guarantee that every requesting client would eventually be acknowledged.

Control specification. Again, the property we wish to enforce is mutual exclusion: no two clients are acknowledged simultaneously. This is written in CTL as:

$$\bigwedge_{i \neq j} AG \quad \overline{ack_i \cdot ack_j}$$

As explained before, we choose ti_1 as controllable variable. The incremental synthesis as well as the global synthesis have been performed on this example with a configuration of 4 cells: $C_1, \cdots, C_4$. Each cell is modeled by 3 automata.

Incremental DCS. IDCS is applied to this example, as illustrated in Figure 3.15. We begin with $C_2||C_3||C_4$ and an abstract model C_1^{abs} for cell C_1. An abstract global model is constructed as $C_1^{abs}||C_2||C_3||C_4$. The refinement step replaces C_1^{abs} with C_1. By applying this procedure, IDCS outperforms direct DCS, as shown in section 3.7.6.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

3.7.2 Fault-tolerant task scheduling problem

Inspired from [39], this example models the fault-tolerant scheduling of 2 tasks executing on a 3 processor architecture.

Functional description. Each task starts from the "idle" state I_i; it enters the "ready" state R_i as a result of the occurrence of event r_i; it then gets to one of the "execution phase A on the j-th processor" states A_{ij} according to the combination of control signals a_{i1}, a_{i2}; phase A, when a processor failure f_j occurs, the task must migrate to one of the other processors, which are assumed to be operational; a checkpoint event chk_i models the safe termination of a computation phase. At this point, the task starts a new computation phase B, and for that purpose can be rescheduled to one of the available processors.

The control signals $\{a_{i1}, a_{i2}\}$ actually encode the activation control commands with the following significations:

- $a_{i1}\bar{a}_{i2}$: execute on processor 1;
- $\bar{a}_{i1}a_{i2}$: execute on processor 2;
- $a_{i1}a_{i2}$ or $\bar{a}_{i1}\bar{a}_{i2}$: execute on processor 3 .

The task part and the processor part of the model are shown in Figure 3.16 and Figure 3.17 respectively.

The entire system is the synchronous product of 2 task models $T_i, i = 1, 2$ and 3 processor models $P_j, j = 1, 2, 3$ as well as an environment model which models a fault assumption: at most one processor can fail. The control objective is to guarantee that no task can be scheduled on a faulty processor. This specification can be expressed in CTL:

$$AG\,\overline{\bigvee_{j=1,2,3}\Big(\bigvee_{i=1,2}(A_{ij}+B_{ij})\Big)\cdot err_j}.$$

The environment model describing the fault assumption is a FSM which emits the signals f_i. Processor faults are modeled by the uncontrollable $\{e_1, e_2, e_3\}$. The unicity of a processor failure is modeled as follows:

- if $e_1\bar{e_2}\bar{e_3}$ then f_1 occurs;
- if $\bar{e_1}e_2\bar{e_3}$ then f_2 occurs;
- if $\bar{e_1}\bar{e_2}e_3$ then f_3 occurs.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

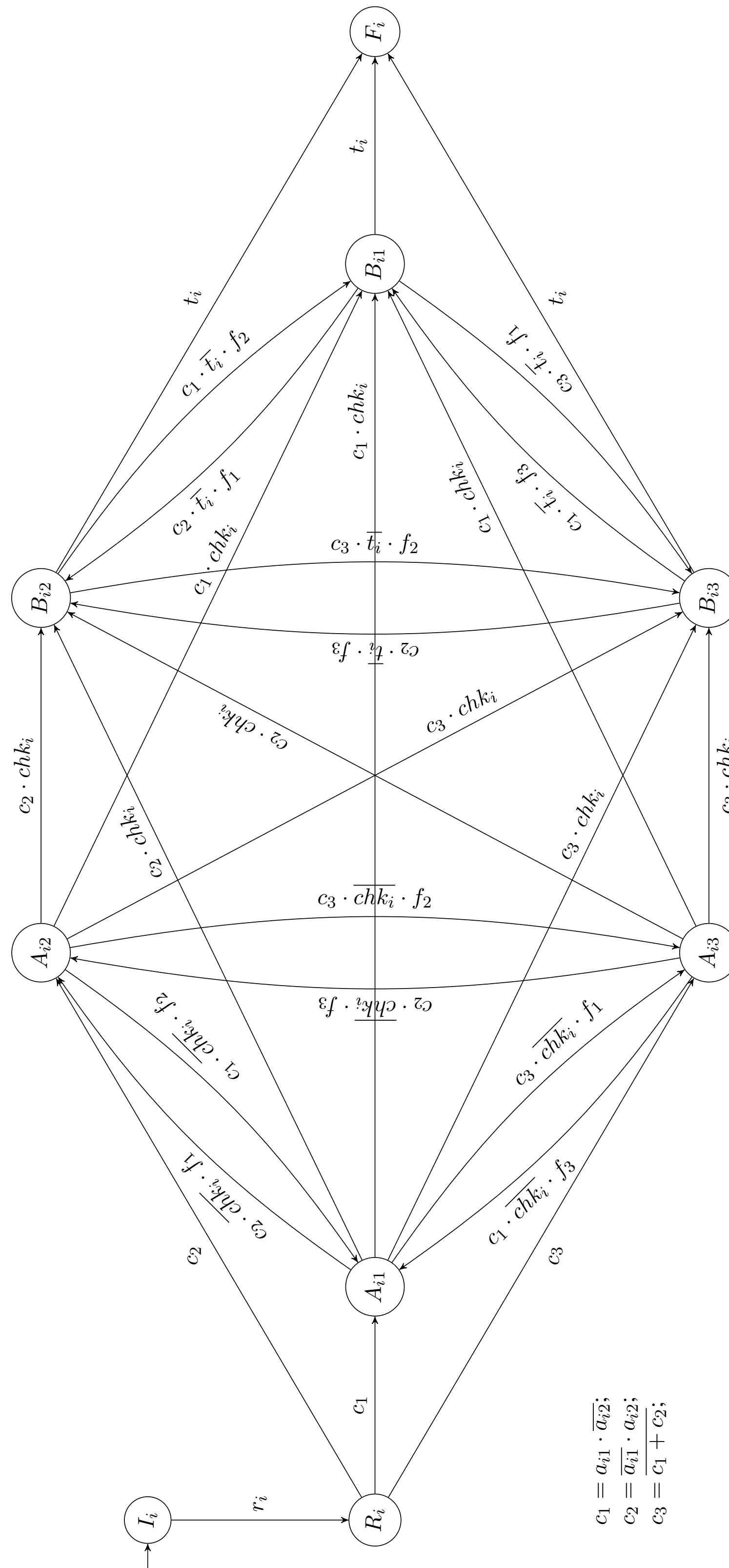


FIGURE 3.16: Task model with $i = \{1, 2\}$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

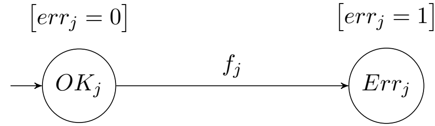


FIGURE 3.17: Processor model with $j = \{1, 2, 3\}$

Whichever values $\{e_1, e_2, e_3\}$ take, only one failure can occur at a time.

The system's inputs are partitioned as follows:

- controllable input set: $\{a_{11}, a_{12}\}$;
- uncontrollable input set: $\{r_1, t_1, chk_1\}$.

Incremental DCS. The architecture of the abstract global system is shown in Figure 3.18.

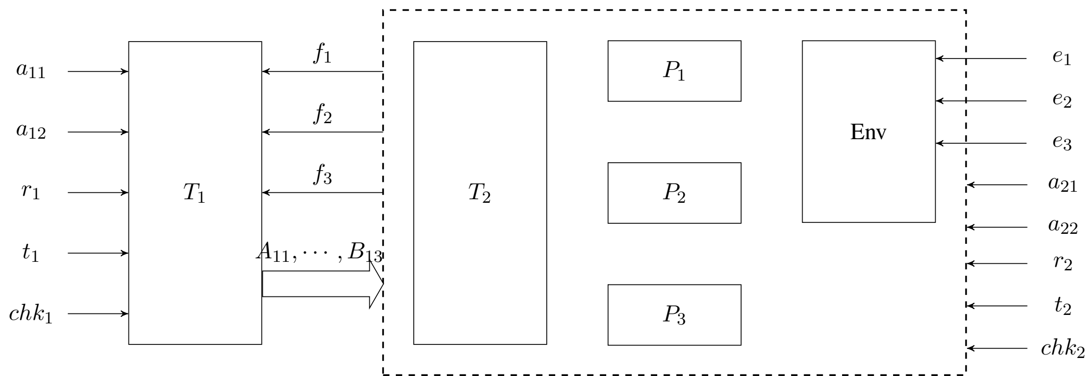


FIGURE 3.18: System architecture

IDCS is applied to the task scheduling model by starting with the abstraction:

$$M^{abs} = T_1^{abs}||T_2||P_1||P_2||P_3||Env.$$

The refinement step adds the previously abstracted module T_1, i.e.:

$$M = T_1||T_2||P_1||P_2||P_3||Env.$$

The IDCS application using the above decomposition outperforms direct DCS as shown in section 3.7.6.

3.7.3 The "Cat and mouse" problem

The cat and mouse control problem has been presented in [6] [3].

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Functional description A cat and a mouse are placed in a maze with 5 rooms numbered from 0 to 4, as shown in Figure 3.19. There are doors between two adjacent rooms, named from c_1 to c_7, and m_1 to m_6. The animals can move through these doors with the following restrictions. The cat can only pass through doors c_1 to c_7, while the mouse can only pass through doors m_1 to m_6. Each doorway can be passed in only one direction, with the exception of the door c_7 which is bi-directional. All moves of the cat and the mouse are observable. Each door is controllable to be open or close, except for the door c_7 which always stays open. As the initial state, the cat and the mouse are in room 2 and room 4 respectively. Automaton models of cat and mouse are presented in Figure 3.20.

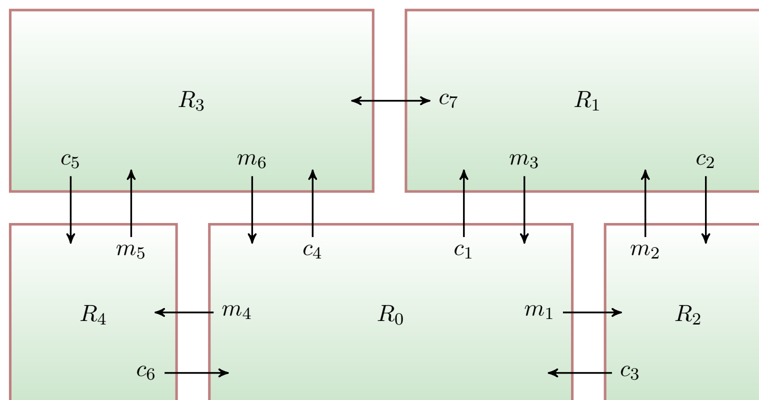


FIGURE 3.19: Cat and mouse example

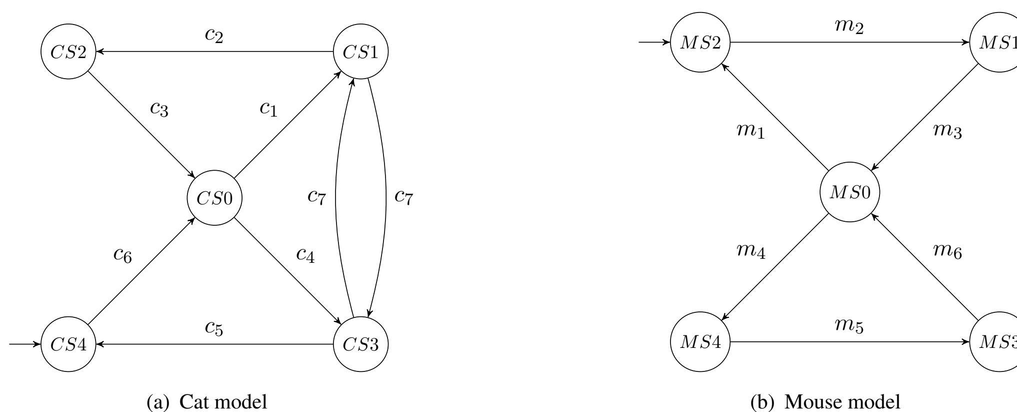


FIGURE 3.20: Cat and mouse model

The control specification we wish to enforce expresses a safety requirement: the cat and the mouse should never be in the same room simultaneously. In CTL this is expressed by:

$$AG \; \overline{\bigvee_{i=0,\cdots,4} (CS_i \cdot MS_i)}.$$

Incremental DCS. Again, in this example there is no explicit communications between the components. The synchronization is forced by the satisfaction of the global specification, as in the arbiter example shown in section 3.4.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

The problem instance we have studied features two cats C_1, C_2 and two mice M_1 and M_2. The IDCS technique is applied with the following abstraction/refinement process:

$$M^{abs1} = M_1||M_2||C_1^{abs}||C_2^{abs}$$

then

$$M^{abs2} = M_1||M_2||C_1||C_2^{abs}$$

and finally

$$M = M_1||M_2||C_1||C_2$$

The IDCS application outperforms direct DCS, as shown in section 3.7.6.

3.7.4 The "Dining philosophers" problem

Functional description. The dining philosophers problem is a famous problem of several users sharing a set of common resources. It is summarized as five philosophers sitting at a round table doing one of two things: eating or thinking, as shown in Figure 3.21. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each pair of adjacent philosophers, and as such, each philosopher has one fork to his left and one fork to his right. The behavior of each philosopher is as follows. The philosopher may be thinking or he may want to eat. In order to go from the "thinking" state to the "eating" state, the philosopher needs to pick up both forks on the table, one at a time, in either order. After the philosopher is done eating, he places both forks back on the table and returns to the "thinking" state. Each philosopher can only use the forks on his immediate left and immediate right.

In our experiments, we only consider 3 philosophers with 3 forks. Figure 3.22(a) and Figure 3.22(b) show the automaton model of the philosopher $P_i, i = 1, 2, 3$ and the fork $F_j, j = 1, 2, 3$. Philosopher P_i has two adjacent forks, the left one noted as fork F_j and the right one noted as fork F_k. When the left fork F_j is available ($a_j = 1$), the philosopher P_i can pick up it and outputs $l_j = 1$ to announce its status. The same rule applies to the right fork. One fork can be used by only one philosopher at a time. When two persons try to pick up the same fork j, it will enter in an error state E_j and outputs $e_j = 1$. The events are denoted by f_{ij} for "philosopher i picks up fork j"and t_i for "philosopher j finishes eating and puts both forks down". We consider the pick up action $f_{ij}, i = 1 \cdots 3, j = 1 \cdots 3$ are controllable, and the put down action $t_i, i = 1 \cdots 3$ are uncontrollable.

Control specification. If each philosopher picks the fork on his left side, each philosopher will be waiting for the philosopher on his right side to drop the fork. A deadlock will happen as no one would like to give up the competition. Similar situation can happen when each philosopher holds the fork on

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

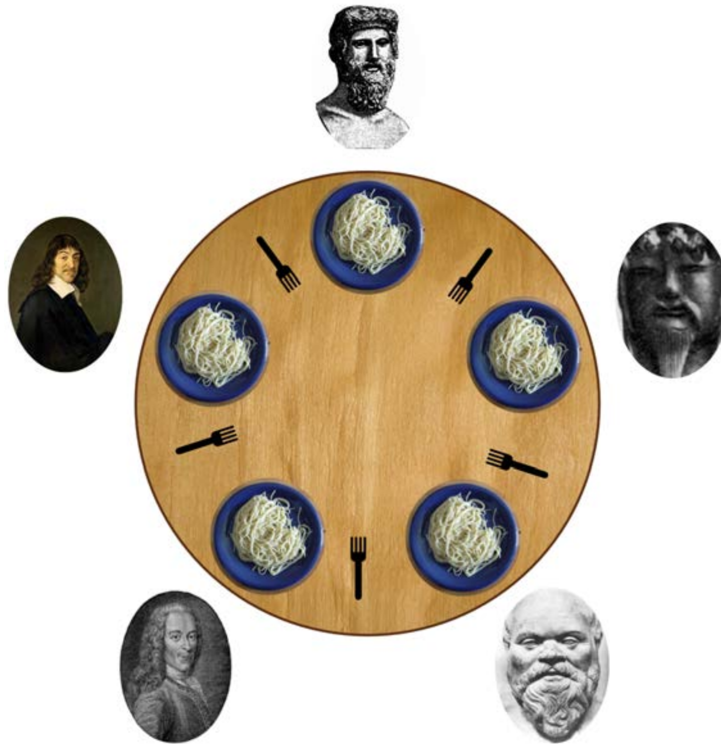
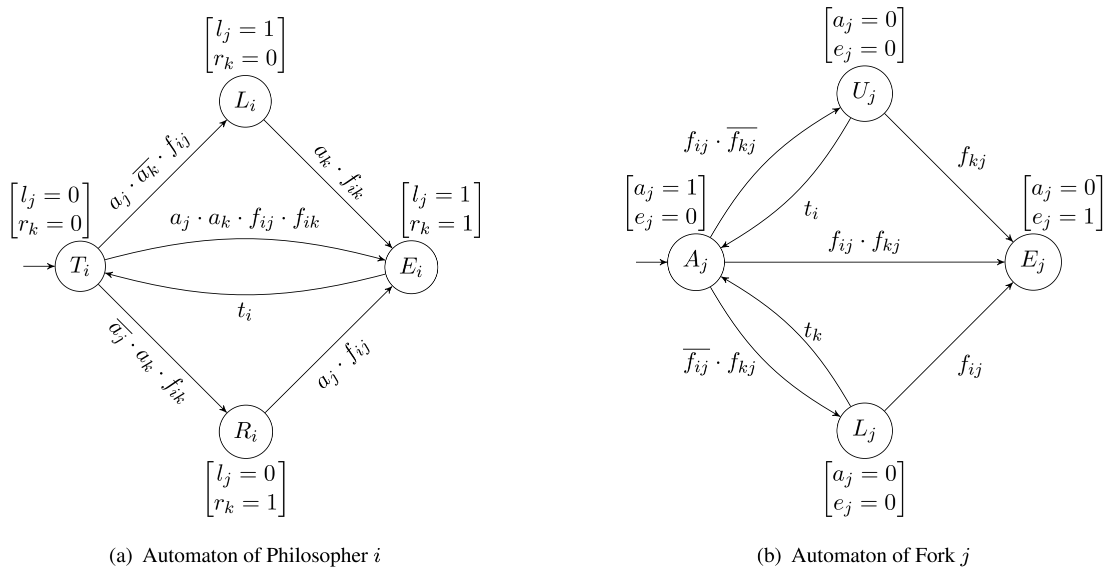


FIGURE 3.21: Dining philosophers problem



(a) Automaton of Philosopher i

(b) Automaton of Fork j

FIGURE 3.22: FSM model of Dining philosophers problem

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

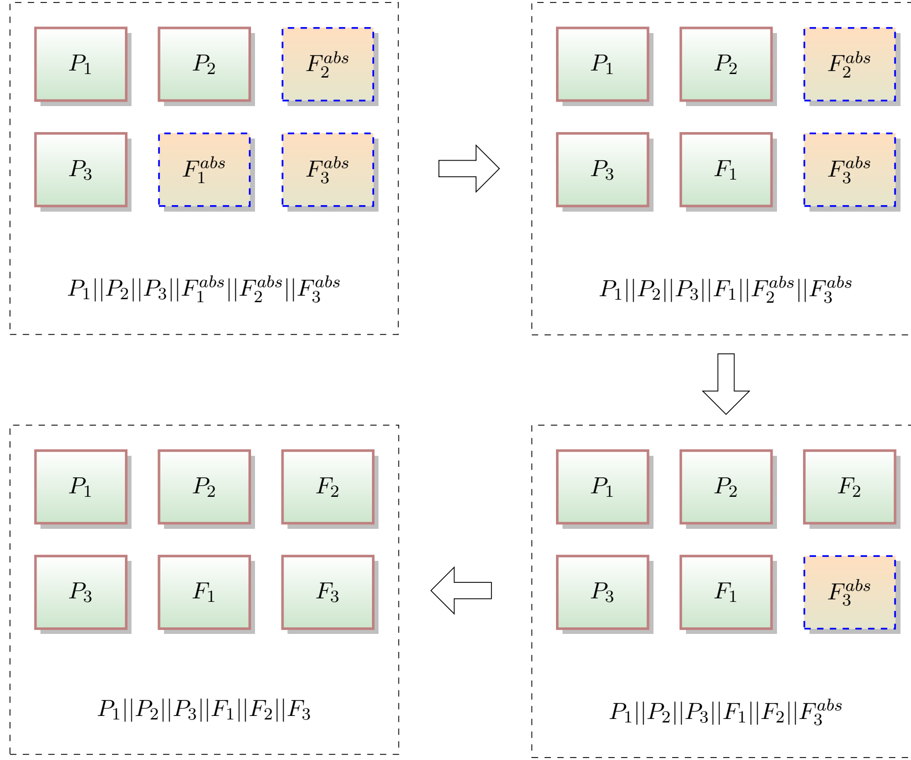


FIGURE 3.23: Incremental DCS procedure

his right side. The control objective is to prohibit such deadlock, stated as: the philosophers should not hold the fork on their right side or the fork on their left side simultaneously. In CTL formula:

$$AG \quad \overline{(l_1 \cdot l_2 \cdot l_3)} \cdot \overline{(r_1 \cdot r_2 \cdot r_3)} \cdot \overline{e_1 + e_2 + e_3}.$$

Incremental DCS. In this example, philosophers and forks are connected through variables a_j. They are also synchronized by the global specification. The incremental synthesis procedure is illustrated in Figure 3.23.

We start with the abstract model

$$P_1||P_2||P_3||F_1^{abs}||F_2^{abs}||F_3^{abs},$$

i.e.: we replace $F_j, j = 1, 2, 3$ by their corresponding abstraction F_j^{abs}. This abstract model leads to an approximate intermediate result $\mathcal{IUC}^{abs1}$. We then refine it incrementally: we refine F_1^{abs} by F_1 and F_2^{abs} by F_2, one after another. This procedure gives two approximate result $\mathcal{IUC}^{abs2}$,$\mathcal{IUC}^{abs3}$. When the last component F_3^{abs} is refined to F_3, a final synthesis is performed to obtain the final result $\mathcal{IUC}$. The performance figures are shown in section 3.7.6 as PH3.

We also performs different orders on this example. In the configuration PH1, we begins with

$$P_1||P_2||P_3||F_1||F_2||F_3^{abs},$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

i.e. we only replace F_3 with F_3^{abs}. In this case, we have only one refinement. The performance figures are shown in section 3.7.6 as PH1.

We then starts with

$$M^{abs1} = P_1||P_2||P_3||F_1||F_2^{abs}||F_3^{abs},$$

and performs two steps of refinement:

$$M^{abs2} = P_1||P_2||P_3||F_1||F_2||F_3^{abs},$$

and

$$M = P_1||P_2||P_3||F_1||F_2||F_3.$$

This is noted as PH2 in section 3.7.6.

All three above procedures of IDCS application outperform direct DCS, as shown in section 3.7.6.

3.7.5 PI-Bus arbiter

The Open Microprocessor systems Initiative (OMI) was founded in October 1991 with the goal of providing Europe with strong capabilities in microprocessor systems [40]. In 1994, OMI published a Peripheral Interconnect Bus (PI-Bus) protocol which is aimed to be used in modular, highly integrated microprocessors and micro-controllers for the purpose of on-chip module communication. In a PI-Bus system, processors or DMA controllers initiate date read or write operations through an interface called PI-Bus *master* agent; The addressed devices which perform the actual read or write operation are connected via the interface PI-Bus *slave* agent. The ownership of the bus is managed by the *bus arbiter*.

Functional description. An embedded system using PI-Bus as on-chip interconnect is shown in 3.24. It has three parts: *bus arbitrator* with one or more *master interface* and one or more *slave interface*. The bus arbitrator takes care of the bus usage. Whenever a "master" device wants to use the bus, it has to ask to the bus arbitrator. The bus can be used by a "master" device only when that device is granted an access privilege.

Figure 3.25 shows the interconnection of the master interfaces, slave interfaces and the bus arbiter. A master can initiate an operation via signal RQ_i. It can also send request to "lock" the bus. The operation type is determined by the variable OPC. One of the slaves is selected at a time by the SEL signal. As a response to the master device, a slave sends its acknowledgment via ACK, which encodes **IDLE**, **RDY**, **RDM**, **WAT**, **ERR** and **RTR**.

We are mainly interested in the bus arbiter. Details on *master* and *slave* interfaces can be found in [41]. The bus arbiter is responsible for three main functions:

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

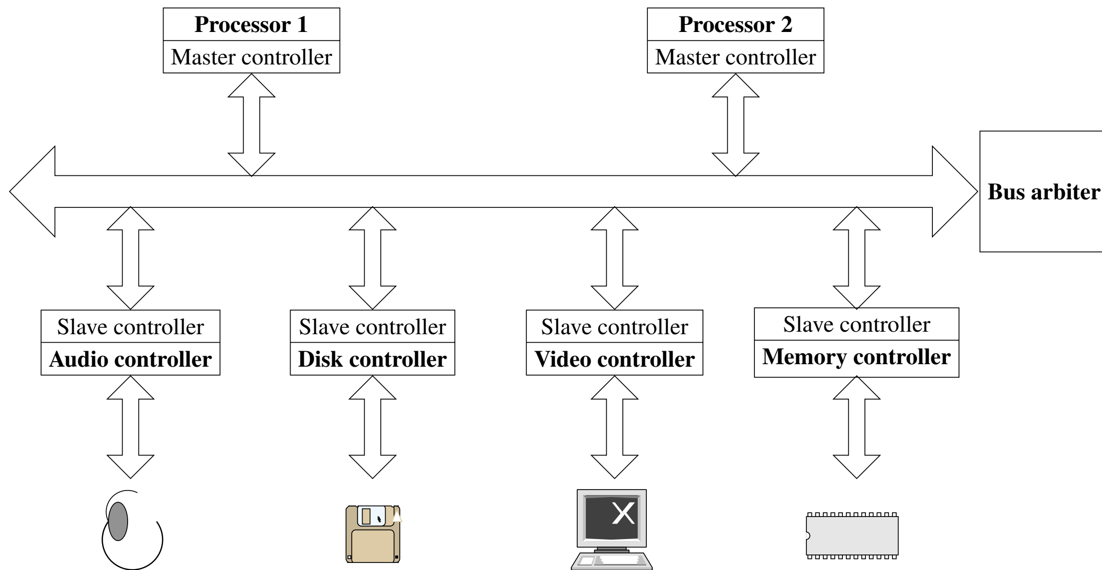


FIGURE 3.24: System architecture of PI Bus

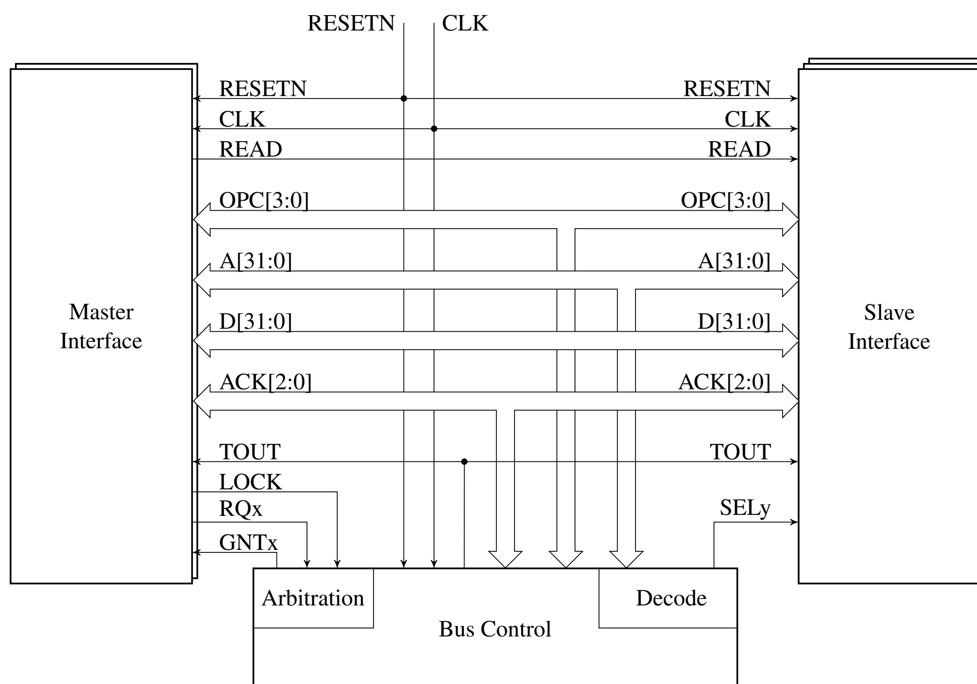


FIGURE 3.25: Internal interaction

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- **arbitration:** Bus control needs to arbitrate which master is granted the requested bus. If no bus requests the bus, grant may be given by default to one master known as the default master. A grant may only be given if:
  - the bus is idle, or
  - during the data cycle of a no-operation and the lock signal is not set, or
  - during the data cycle, the slave is returns acknowledge code **RDY** or **RDM** and the lock signal is not set.
- **address decoding:** In order to determine the target slave of a bus operation, bus control decodes in the address cycle upper bits of the address issued by the granted master. The corresponding slave select signal is set in the same address cycle. No slave select is generated if
  - the master starts a no-operation;
  - in the case of a pipelined bus operation, if the slave is returning a wait acknowledge code (**WAT**) in the data cycle of the previous bus operation.
- **timeout control:** The timeout mechanism is intended for bus operations which are not completed by the selected slave with a **ready** acknowledge code. The bus control flags a timeout after an implementation dependent number of clock cycles.

The state transition diagram for the bus controller is shown in Figure 3.26.

Incremental DCS. A Verilog HDL implementation of PI-Bus protocol can be found in the *Texas97 verification benchmarks* package [42]. We've applied the incremental synthesis on a configuration with two masters. The desired property is a mutual exclusion of the access to the bus: the grant signals GNT_0 and GNT_1 should not be set simultaneously, or in CTL formula:

$$AG\overline{(GNT_0 \cdot GNT_1)}.$$

There is an internal counter which counts clock cycles and emits a "time out" signal to the arbiter. This count is implemented as a 256-bit counter composed of 8 full adders. The counter is reset to 0 on receiving a "clear" signal CLR; it continues to add one on the arrival of the "increment" signal INC. When it counts to 255, it emits an "time out" event by raising its output signal $TOUT$.

The counter is connected to the main logic of the arbiter via variables CLR, INC and $TOUT$, as illustrated in Figure 3.27.

An error is intentionally introduced into the main logic of the bus arbiter so that the desired mutual exclusion property could be violated under certain circumstances. Thus a supervisor is needed to correct this "bug". We choose RQ_0 and RQ_1 as controllable variables.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

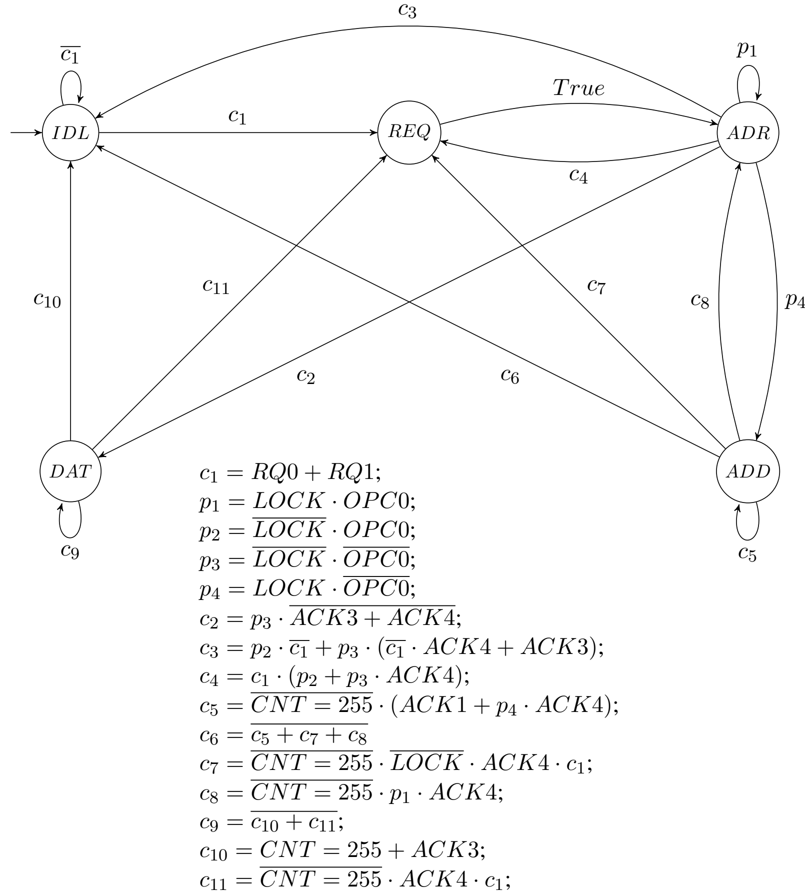


FIGURE 3.26: Automaton of bus arbitrator

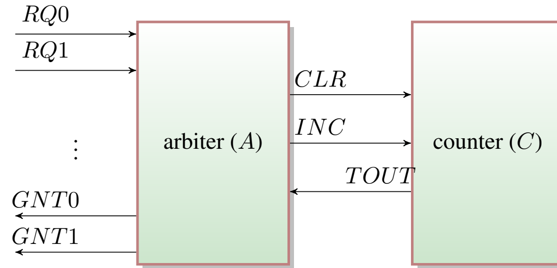


FIGURE 3.27: Arbiter structure

We start incremental synthesis procedure by a composition of the arbiter and the abstraction of the counter C^{abs}:

$$M^{abs} = A||C^{abs}.$$

An approximate result $\mathcal{IUC}^{abs}$ is calculated from this abstract global model; the abstract model is then refined by replacing C^{abs} with C; the actual $\mathcal{IUC}$ is then calculated on the basis of the approximate $\mathcal{IUC}^{abs}$; a supervisor is finally calculated from the actual $\mathcal{IUC}$. The incremental DCS application outperforms direct DCS, as shown in section 3.7.6.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

TABLE 3.1: Experiment results

		PB	BA	MA	TA	CM	PH1	PH2	PH3
glob.	mem.	3.8	4.2	1.3	6.1	17.2	57	57	57
	BDD	0.05	0.05	7728	0.08	0.65	2.4	2.4	2.4
	time	5m20s	7s	-	100s	1m	4m44s	4m44s	4m44s
incr.	mem.	2.0	1.9	1.1	3.3	6.1	51	9.4	2.4
	BDD	0.02	0.01	5785	0.04	0.12	1.9	0.26	0.014
	time	0.16s	2s	-	37s	47s	4m25s	19s	1s
gain	mem.	47%	54%	15%	46%	65%	11%	84%	96%
	BDD	60%	80%	25%	50%	82%	21%	89%	99%
	time	-	71%	-	63%	22%	6%	93%	100%

3.7.6 Results

The quantitative figures obtained show that for the examples presented, IDCS improves both computation time and memory usage. Results are shown in Table 3.1, where memory usage is measured in megabytes, and numbers of BDD nodes are shown in millions except for example MA. All experiments are performed on a computer with an Intel Core 2 T7100 CPU and 2GB memory. Note that for the example MA, the two synthesis procedures finish instantaneously. Thus in Table 3.1, they are not marked.

Among these examples, PB stands for PI-BUS, the bus controller example presented in section 3.7.5 which manages shared resources for several devices; The abbreviation BA stands for the distributed arbiter shown in section 3.7.1. It is composed of 4 cells synchronized by a token ring; The example MA is the two-machine arbiter example presented and illustrated in section 3.4 and 3.3; Example TA models the fault-tolerant scheduling of 2 tasks executing on 3 processors, which is presented in section 3.7.2; CM is the “cat and mouse” problem presented in section 3.7.3. It has 2 mice and 2 cats in 5 rooms; Examples PH1, PH2 and PH3 model the 3 dining-philosophers problem presented in section 3.7.4. Their difference is described in section 3.7.4.

The figures obtained show that the IDCS technique achieves very interesting performance improvements over classical DCS. Besides, the figures obtained for the example PH3 strongly suggest that generalization of IDCS to n modules can bring spectacular improvements. Also note that, our main goal is to reduce memory usage, but most of our experiments also show improvements in running time.

3.8 Conclusion

The incremental Discrete Controller Synthesis (IDCS) technique presented exploits the modularity of a system in order to construct a control solution progressively. Abstraction/refinement operations are

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

performed in sequence on a progressively growing system. This process follows a user-specified ordering which must be provided.

The key advantage of IDCS is its ability to exploit modularity *with communication* between the different modules. It improves the performance of the classical BDD-based DCS, for systems featuring concurrent communicating modules. The time/memory efficiency of IDCS is illustrated with quantitative figures.

The IDCS algorithm does not perform any qualitative analysis on the system. However, such an analysis could be a useful step. We believe that the connectivity between modules can be a criterion for choosing an abstraction order. This is one interesting future research direction for this work. Another important research direction is the application of incremental/compositional techniques for accessibility and liveness control specifications.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Chapter 4

Implementation of supervisors

4.1 Introduction

In Chapter 2 and Chapter 3, we have investigated the symbolic discrete supervisor synthesis technique, which we have extended to an incremental approach, with better performance figures. In this chapter, we address the supervisor implementation issue.

Both DCS and IDCS produce a symbolic supervisor which is an equation. Under that form, it is not suitable for hardware design. Our objective is to *implement* the supervisor by solving the control equation symbolically. For each controllable input, we produce a Boolean expression which always evaluates to the admissible control value for that input. Thus, the supervisor is transformed into a set of Boolean expressions, whose hardware implementation is straightforward.

Thus, the symbolic resolution of the control equation calls for the resolution of two main problems, one behavioral and one structural:

1. the *control non-determinism*: Most synthesized supervisors are control non-deterministic, in the sense that if at a given time several control solutions are possible, none of them are chosen a priori. To make such choice, it is necessary to either define a heuristic method to automatically remove the non-determinism, or rely on the intervention from a human operator. In general, the removal of the non-determinism amounts to restrain the control solutions to a strict subset. The deterministic result obtained is commonly designated by the name of *controller*.

2. the *structural incompatibility* between the supervisor and the system to control. This problem only concerns symbolic DCS. Indeed, symbolic supervisors do not feature input/output signals. They are in represented by a Boolean equation encoding all acceptable control solutions. A straight-forward supervisor implementation would involve performing continuous on-line resolution of the

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

supervisor equation at the run-time. This requires a Boolean solver software tool, physically running on a microprocessor (dedicated or not). The target architecture of the implemented supervisor is thus a software architecture, which has limitations in terms of run-time efficiency.

4.1.1 Contributions

We propose to solve the control non-determinism and find an adequate solution to the structural incompatibility problem. Solving the control non-determinism guarantees an autonomous implementation for the choice of control solutions. Solving the structural incompatibility problem amounts to finding a representation of the supervisor equation as a set of control functions. This has multiple advantages:

- a hardware implementation, in the form of an electronic circuit, realized by FPGA or ASIC technology, can be obtained;
- the supervisor implementation represented as a set of control functions appears to be much more compact than the initial supervisor.

Besides, we propose a control loop architecture suitable for the hardware design context. We identify a class of hardware design problems for which our control architecture is suitable. This is illustrated in section 4.5.

4.1.2 State of the art

The supervisor implementation generally amounts to solving the control non-determinism. The deterministic control solution obtained is known as the *controller* and is generally a subset of the synthesized supervisor.

Supervisor implementation has been investigated in the past, with a special concern about execution on a programmable logic controller [43] [44][45] [46].

In [47] the approach proposed aims at minimizing protocol latency. It solves non-determinism such that the progress speed towards a target set is maximized. However, once this optimality criterion is satisfied, any remaining non-determinism is solved by making arbitrary choices. This work has been generalized and improved in [48] [49].

Most research we are aware of have achieved supervisor implementation by developing hints in order to successively refine the control objective and finally yield a deterministic supervisor (i.e. a controller). Other implementation approaches treat non-determinism by achieving a random choice. The same problem can also be addressed by choosing either fixed or dynamic priority mechanisms over the controllable input set.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Our technique achieves deterministic supervisor implementation by applying a decomposition procedure to the supervisor while conserving the control solutions obtained by synthesis. Similar approaches have been proposed and used in different contexts besides controller synthesis. In [50], a parametric decomposition technique is applied over Boolean predicates in the context of formal verification of hardware designs. This technique applies a decomposition procedure quite close to ours. In [51], a controller implementation technique is presented. Controllers are obtained by optimal synthesis (with quantitative criteria); the technique is also symbolic, but it produces a strict subset of the supervisor. In [52] the authors study the triangulation of a polynomial equation over ternary values, with the same objective of achieving a supervisor implementation. The same objective is also investigated in [53], where the control synthesis technique operates directly from specifications (by contrast to DCS, which needs a system). A supervisor hardware implementation approach starting from Petri nets is presented in [54].

The outline of this chapter is the following:

- Section 4.2 presents two principle problems to solve;
- Section 4.3 presents our supervisor implementation algorithm;
- In Section 4.4, a simple manufacture example is studied to show the significance of the implementation method.
- In Section 4.5, we discuss the application of DCS and supervisor implementation method in two hardware design contexts;
- Section 4.6 gives information about the tools that have been used or developed.

4.2 The supervisor implementation problem

We recall that the studied system is assumed to be fully observable. As illustrated in chapter 2, the symbolic DCS approach handles sets of states and/or transitions, instead of languages. For a given system M and a desired specification $spec$, called a *control objective*, symbolic DCS computes a supervisor $\mathcal{SUP}$ guaranteeing that M always satisfies $spec$. The control architecture is illustrated in Figure 4.1.

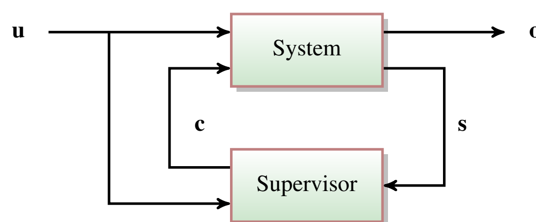


FIGURE 4.1: Traditional control loop

The symbolic supervisor computation proceeds following two steps:

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

1. compute the *invariant under control* ($\mathcal{IUC}$) subset of states of M. As long as M remains inside $\mathcal{IUC}$ the violation of $spec$ can be indefinitely postponed. The computation details of $\mathcal{IUC}$ is listed in Chapter 2.

2. compute the supervisor $\mathcal{SUP}$: the set of all transitions $\mathbf{s} \stackrel{\mathbf{u},\mathbf{c}}{\longmapsto}$ of M leading to $\mathcal{IUC}$. The exact expression of the supervisor is:

$$\mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) = \exists \mathbf{s}' : \mathcal{T}(\mathbf{s}, \mathbf{u}, \mathbf{c}, \mathbf{s}') \cdot \mathcal{IUC}(\mathbf{s}').$$

In the above equation, $\mathbf{s} = \langle s_1, \cdots, s_n \rangle$ is the vector of state variables; $\mathbf{u} = \langle u_1, \cdots, u_m \rangle$ and $\mathbf{c} = \langle c_1, \cdots, c_p \rangle$ are the vector of uncontrollable and controllable variables respectively. In the following, we use s, u, c to represent an element of vector $\mathbf{s}$, $\mathbf{u}$ and $\mathbf{c}$ respectively.

A control solution exists iff the initial state $s_0 \in \mathcal{IUC}$. From a dynamic point of view, $\mathcal{SUP}$ is said to "play" with the controllable inputs C, against the environment, "playing" with the uncontrollable inputs U of M, with the objective of never reaching a state violating $spec$. Thus, for any state $\mathbf{s} \in \mathcal{IUC}$ and for any uncontrollable input vector $\mathbf{u} \in \mathbb{B}^m$, the supervisor $\mathcal{SUP}$ computes adequate values for the controllable inputs $\mathbf{c}$ so that the transition fired by M leads into $\mathcal{IUC}$.

The control non-determinism problem The symbolic supervisor $\mathcal{SUP}$ has two important properties: (1) it is *maximally permissive* i.e. all transitions of M leading to $\mathcal{IUC}$ are contained in $\mathcal{SUP}$ and (2) it is *control non-deterministic* i.e. for a given current state and a given uncontrollable input value, there may exist more than one possible successor states in $\mathcal{IUC}$. In other words, for some value combinations of $s \in \mathbb{B}^n$, $u \in \mathbb{B}^m$, there exists $c, c' \in \mathbb{B}^p$ such that:

$$\begin{aligned} \mathcal{SUP}(s, u, c) &= 1 \\ \mathcal{SUP}(s, u, c') &= 1 \end{aligned}$$

and $c \neq c'$.

Supervisors are usually "passive" in that they don't actively choose from the possible control solutions, but rather provide an admissible event set. Some tools provide a mechanism to solve this non-determinism to some extent. The *Sigalsimu* [3] simulator provides an user interface so that user can choose value of controllable variables when necessary. It can also make random choice. The *hes* [55] simulator used by *BZR* language [35][56] tries to valuate each controllable variable to true if possible [55].

The structural compatibility problem Besides non-determinism, symbolic DCS also raises an architectural problem: the supervisor $\mathcal{SUP}$ is represented by a Boolean characteristic function encoding

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

acceptable transitions with respect to the control objective. The Boolean expression of $\mathcal{SUP}$ is structurally incompatible with respect to the system M and thus with the control architecture represented in Figure 4.1. This structural incompatibility is usually eliminated by solving the Boolean equation:

$$\mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) = 1 \tag{4.1}$$

either "on-line" by a solver, as illustrated in Figure 4.2, or "off-line".

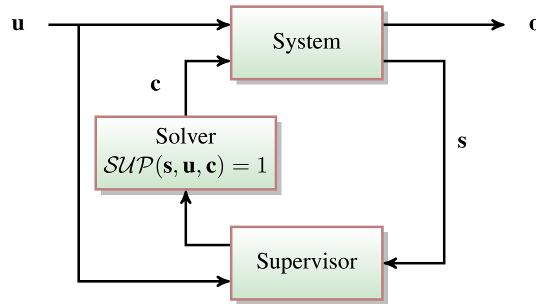


FIGURE 4.2: Control loop with a symbolic supervisor $\mathcal{SUP}$

4.3 Symbolic Supervisor Implementation

The implementation technique we present addresses both of these two issues. The control non-determinism issue is made explicit by adding supplementary environment variables $\mathbf{c}^{env}$, with each $c_i^{env}, i = 1, \ldots, p$ corresponding to c_i. In the example of Figure 4.3, the control non-determinism in state S is expressed by the possibility to choose between $a = 1$ and $a = 0$ in order to stay inside the set $\mathcal{IUC}$. This non-determinism is removed by introducing the auxiliary input variable a^{env}. When M is in state S, the value of the controllable a can be any among the values that can be taken by the auxiliary input a^{env}.

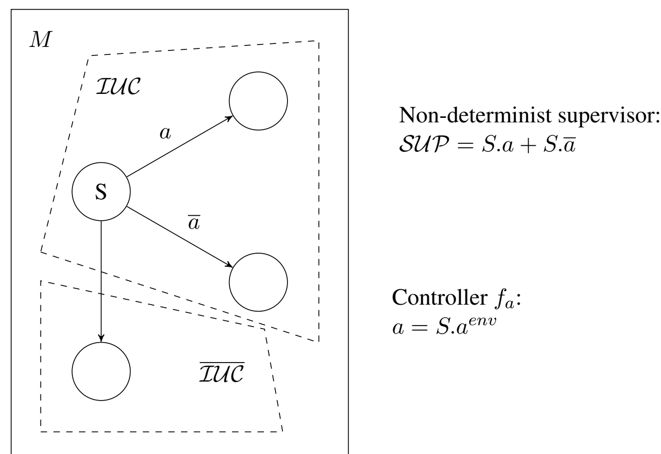


FIGURE 4.3: Removal of control non-determinism

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

The structural incompatibility issue is solved by computing individual expressions for each controllable variable c_i of M. This amounts to solving equation (4.1) off-line. By achieving this transformation, we wish to obtain the architecture shown in Figure 4.4.

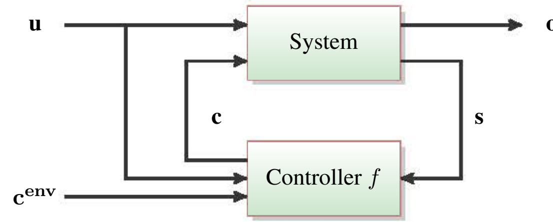


FIGURE 4.4: Target control architecture

Starting from the the supervisor $\mathcal{SUP}$ we construct a controller as a tuple of p control functions:

$$f_i : \mathbb{B}^n \times \mathbb{B}^{p+m} \rightarrow \mathbb{B},\ i = 1, \ldots, p, \text{ such that:}$$

$$c_i = f_i(s_1, \ldots, s_n, u_1, \ldots, u_m, c_1^{env}, \ldots, c_p^{env}) \tag{4.2}$$

The relationship between $\mathcal{SUP}$ and f is expressed by the following equation:

$$\mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) = \exists c_1^{env}, \ldots, c_p^{env} : \prod_{i=1}^{p} (c_i \Leftrightarrow f_i(\mathbf{s}, \mathbf{u}, \mathbf{c^{env}})) \tag{4.3}$$

In other terms, we wish that (1) any control solution allowed by $\mathcal{SUP}$ can be reproduced by f and (2) any control solution computed by f is also accepted by $\mathcal{SUP}$. Finding f satisfying equation (4.3) also amounts to making explicit the control non-determinism of $\mathcal{SUP}$ by introducing the auxiliary input variables $\mathbf{c^{env}}$ as displayed in Figure 4.4.

Supervisor implementation algorithm Note that a major condition prior to implementing $\mathcal{SUP}$ is that it must be satisfiable:

$$\forall \mathbf{s} \in \mathcal{IUC}, \forall \mathbf{u} \in \mathbb{B}^m, \exists \mathbf{c} : \mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) \tag{4.4}$$

In other terms, the predicate $\mathcal{SUP}$ must be satisfiable on the whole set of invariant under control $\mathcal{IUC}$ states no matter what value the uncontrollable variables $\mathbf{u}$ may take. This is also the case in that a synthesis solution exists.

The proposed implementation algorithm relies on the Shannon decomposition of $\mathcal{SUP}$, which is applied to the controllable variables $c_i, i = 1, \ldots, p$:

$$\mathcal{SUP} = \overline{c_i} \cdot \mathcal{SUP}|_{c_i=0} + c_i \cdot \mathcal{SUP}|_{c_i=1}$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

The impact of variable c_i on the satisfiability of $\mathcal{SUP}$ is summarized by the truth Table 4.1. Our objective is to find an expression yielding the values of c_i such that $\mathcal{SUP}$ is satisfied.

TABLE 4.1: Truth table of the Boole decomposition of $\mathcal{SUP}$ with respect to c_i

$\mathcal{SUP}\|_{c_i=1}$	$\mathcal{SUP}\|_{c_i=0}$	c_i	$\mathcal{SUP}$
0	0	0	**0**
0	0	1	**0**
0	1	0	**1**
0	1	1	**0**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

TABLE 4.2: Values of c_i when $\mathcal{SUP}$ is satisfiable

$\mathcal{SUP}\|_{c_i=1}$	$\mathcal{SUP}\|_{c_i=0}$	c_i
0	1	**0**
1	0	**1**
1	1	**0 or 1** (c_i^{env})

Table 4.2 summarizes the values that a controllable variable c_i may take, so that the supervisor equation (4.1) is satisfied. As pointed out by this table, the simultaneous satisfaction of both co-factors $\mathcal{SUP}|_{c_i=0}$ and $\mathcal{SUP}|_{c_i=1}$ has a special meaning: regardless of the value of c_i, all transitions allowed by $\mathcal{SUP}$ lead to $\mathcal{IUC}$. This characterizes the control non-determinism with respect to the controllable variable c_i. Each time c_i has no impact on the satisfaction of $\mathcal{SUP}$, its value is driven by c_i^{env}. Structurally, variables c^{env} are auxiliary environment (input) variables as pointed out in Figure 4.4.

According to Table 4.2, the following Boolean expression computes the value of f_i, associated to the controllable variable c_i:

$$f_i = \overline{\mathcal{SUP}|_{c_i=0}} \cdot \mathcal{SUP}|_{c_i=1} + c_i^{env} \cdot \mathcal{SUP}|_{c_i=0} \cdot \mathcal{SUP}|_{c_i=1} \tag{4.5}$$

The expression (4.5) represents the basic step of the controller implementation algorithm presented in Algorithm 5: f_0 is computed by assuming that $\mathcal{SUP}$ holds. c_0 is substituted by f_0 inside $\mathcal{SUP}$, yielding $\mathcal{SUP}_1$. Then the algorithm computes f_1 assuming that $\mathcal{SUP}_1$ holds, and so on.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Algorithm 5 Supervisor implementation algorithm

Require: $\mathcal{SUP}$, a satisfiable supervisor
1: {starts with $\mathcal{SUP}$ and computes $f_1, \ldots, f_p$}
2: {intermediate results: $\mathcal{SUP}_1, \ldots, \mathcal{SUP}_{p+1}$}
3: $\mathcal{SUP}_1 \leftarrow \mathcal{SUP}$
4: **for** $i = 1 \rightarrow p$ **do**
5: $\quad f_i \leftarrow \overline{\mathcal{SUP}_i|_{c_i=0}} \cdot \mathcal{SUP}_i|_{c_i=1} + c_i^{env} \cdot \mathcal{SUP}_i|_{c_i=1} \cdot \mathcal{SUP}_i|_{c_i=0}$
6: $\quad \mathcal{SUP}_{i+1} \leftarrow$ substitute c_i by f_i in $\mathcal{SUP}_i$
7: **end for**

Algorithm 5 produces the tuple of functions:

$$f = \begin{pmatrix} f_1(\mathbf{s}, \mathbf{u}, c_1^{env}, f_2, \ldots, f_p) \\ f_2(\mathbf{s}, \mathbf{u}, c_2^{env}, f_3, \ldots, f_p) \\ \ldots\ldots \\ f_p(\mathbf{s}, \mathbf{u}, c_p^{env}) \end{pmatrix}$$

as well as a residual expression $\mathcal{SUP}_p(\mathbf{s}, \mathbf{u})$ obtained by successive substitutions of f_i for c_i into $\mathcal{SUP}$. In order to guarantee that all control solutions generated by f are accepted by $\mathcal{SUP}$, we must show that $\mathcal{SUP}_p$ is a tautology: $\forall \mathbf{s} \in \mathcal{IUC}, \forall \mathbf{u} \in \mathbb{B}^m : \mathcal{SUP}_p$.

Theorem 1. $\mathcal{SUP}_p$ is a tautology.

Proof. By applying the successive substitutions of f_i into $\mathcal{SUP}_i$ the following expression is obtained for $\mathcal{SUP}_p$:

$$\mathcal{SUP}_p = \exists c_p, \ldots, \exists c_1 : \mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c})$$

Thus, as long as the system M remains inside its invariant under control set $\mathcal{IUC}$ and as long as the initial assumption expressed in (4.4) also holds, the predicate $\forall \mathbf{s} \in \mathcal{IUC}, \forall \mathbf{u} \in \mathbb{B}^r : \mathcal{SUP}_p$ is true.

□

This theorem shows that by substituting f for $\mathbf{c}$ in $\mathcal{SUP}$ we obtain a tautology and hence that all control solutions generated by f are contained in $\mathcal{SUP}$. Conversely, we need to prove that:

Theorem 2. All control solutions contained in $\mathcal{SUP}$ can be reproduced by f.

Proof. The above statement is equivalent to:

$$\forall \mathbf{s}, \forall \mathbf{u}, \forall \mathbf{c} : \big(\mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) \implies \exists \mathbf{c}^{\mathbf{env}} : \mathbf{c} \Leftrightarrow f(\mathbf{s}, \mathbf{u}, c^{env})\big)$$

By applying the existential quantification and by substituting the expression of f given in (4.5), the right term of the implication is equivalent to $\mathcal{SUP} + \prod_{i=1}^{p} \overline{c_i} \cdot \overline{\mathcal{SUP}|_{c_i=0}} \cdot \overline{\mathcal{SUP}|_{c_i=1}}$. Thus, the above implication is a tautology. □

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

It is important to note that the controller f we obtain depends on the variable order used when applying the Shannon decomposition. This order establishes an evaluation priority: f_p is evaluated first, and its value is used to evaluate $f_{p-1}, \ldots, f_1$.

4.4 Application of the supervisor implementation algorithm

In this section, a simple example inspired from the manufacturing control area is studied, in order to illustrate the advantages the implementation method described above.

4.4.1 Functional description

Consider a manufacture line with two machines connected by one buffer, as shown in Figure 4.5.

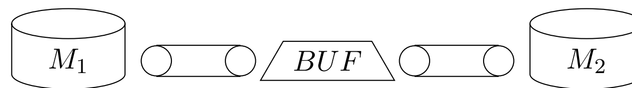
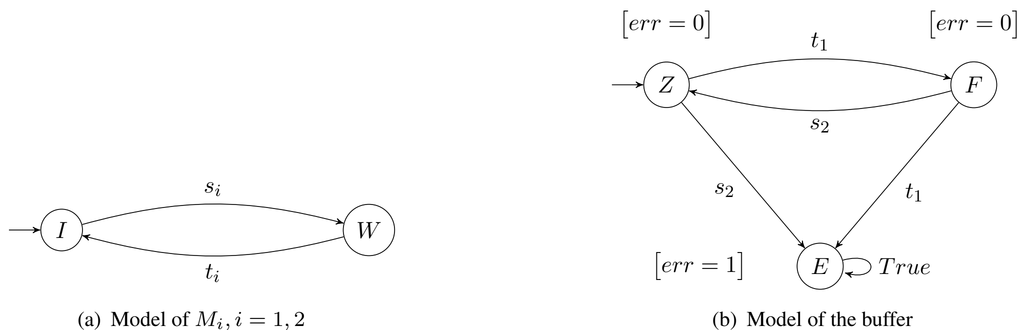


FIGURE 4.5: A manufacture line

Each workpiece has to go through two processing procedures carried out by two machines M_1 and M_2 respectively. When a workpiece finishes in M_1, it then enters into the buffer to wait for the availability of M_2. The two machines work as shown in Figure 4.6(a).



(a) Model of $M_i, i = 1, 2$

(b) Model of the buffer

FIGURE 4.6: FSM models of the system

A machine fetches a new workpiece and starts processing by receiving a *starting* event s_i; it finishes processing and outputs the workpiece by receiving a *terminating* event t_i from a dedicated sensor. We assume that the s_1 and s_2 input events of the two machines are controllable while events t_1 and t_2 are uncontrollable. This assumption is justified by the fact that the start time can be chosen by the environment, but the moment when the processing terminates cannot be known a priori.

Assume that the buffer can only hold one workpiece at a time. When it holds a workpiece and another one is coming, an overflow will occur; when the buffer is empty and the machine M_2 is trying to fetch

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

a workpiece from it, another error will occur. When any of the two errors occurs, it will enter the error state E and send $err = 1$. The buffer model is shown in Figure 4.6(b).

4.4.2 Supervisor implementation

The control objective amounts to making the error state E inaccessible. In CTL, given $spec = \overline{err}$, we wish to enforce the property:

$$AG(spec).$$

The entire system model is the synchronous product of the three individual models, which contains 12 composed states. For readability reason, we name the composed states by concatenating state names of the three components in the order M_1, M_2 and BUF. For example, the composed state I_1W_2F means M_1 in state I, M_2 in state W and BUF in state F. This mechanism assumes a one-hot state encoding of each FSM, which is more readable than a Boolean encoding.

Among the 12 composed states, 4 states should be forbidden by $\lambda^{-1}(spec)$: $\{I_1I_2E, I_1W_2E, W_1I_2E, W_1W_2E\}$. Furthermore, from states W_1I_2F and W_1W_2F, machine M_1 can issue an uncontrollable event t_1 which forces the system into the forbidden states, and thus these two states should also be forbidden. The 6 states left constitute the *invariant under control* $\mathcal{IUC}$. A symbolic supervisor is extracted from the $\mathcal{IUC}$ and the transition relation, in the form of an equation:

$$\begin{aligned} (W_1W_2Z + W_1I_2Z + I_1W_2F)\cdot\overline{s}_1\cdot\overline{s}_2 &\quad + \qquad (4.6)\\ (I_1W_2Z + I_1I_2Z)\cdot\overline{s}_2 &\quad + \qquad (4.7)\\ I_1I_2F\cdot(\overline{s}_1 + s_2) &\quad = \quad 1 \qquad (4.8)\end{aligned}$$

Any combination satisfying the above equation is a solution of the control problem. For instance, if the system is in state I_1I_2F, this means that both machines are "idle" and there is one workpiece already in the buffer. A control choice exists between not starting M_1 or starting M_2 or do both actions simultaneously. We can either prevent M_1 from passing a new workpiece to the buffer by issuing $s_1 = 0$, or start M_2 to fetch the workpiece by issuing $s_2 = 1$. This choice is left to the user.

In the following the supervisor generated above is implemented by applying the algorithm 5. For readability reasons, we note the state set $\{W_1W_2Z, W_1I_2Z, I_1W_2F\}$ as A, set $\{I_1W_2Z, I_1I_2Z\}$ as B and set $\{I_1I_2F\}$ as C.

The supervisor equation 4.6 can be rewritten as:

$$A\cdot\overline{s}_1\cdot\overline{s}_2 + B\cdot\overline{s}_2 + C\cdot(\overline{s}_1 + s_2) = 1.$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

By applying out supervisor implementation algorithm 5 according to the variable order s_1, s_2 we obtain two control functions:

$$
\begin{aligned}
f_1 &= s_1^{env} \cdot B + s_1^{env} \cdot s_2^{env} \cdot C \\
f_2 &= s_2^{env} \cdot C
\end{aligned}
\tag{4.9}
$$

where s_1^{env} and s_2^{env} are two environment variables introduced for controllable variables s_1 and s_2 respectively; f_1 and f_2 are control functions for s_1 and s_2 respectively. These two functions actually generate value to feed to the s_i event for two machines.

4.4.3 Interpretation of the implemented supervisor

According to equations 4.9, when the system is in one of the states of set A, we have:

$$
\begin{aligned}
f_1 &= 0 \\
f_2 &= 0
\end{aligned}
$$

In this case, there is no control non-determinism. Both s_1 and s_2 should be forbidden. The environment values s_i^{env} are ignored by the controller.

When the system is in one of the states in set B, we have:

$$
\begin{aligned}
f_1 &= s_1^{env} \\
f_2 &= 0
\end{aligned}
$$

In this case, the buffer is *empty* and machine M_1 is *idle*. While M_2 should be prevented from starting, M_1 is free to fetch a new workpiece and start processing. This choice is left to the environment, which can assign the variable s_1^{env}.

When the system is in state set C, or specifically in state $I_1 I_2 F$, we have:

$$
\begin{aligned}
f_1 &= s_1^{env} \cdot s_2^{env} \\
f_2 &= s_2^{env}
\end{aligned}
$$

This means that M_2 can start anytime, but that M_1 can only start if M_2 is also started. Indeed, the buffer is full and should be emptied by M_2 before M_1 can operate.

According to the control architecture achieved by implementing the supervisor, the values of s_1^{env} and s_2^{env} can be assigned at any time by the environment. All combinations are safe because they are always filtered by the controller.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

4.5 Applying DCS to hardware design

In this section, we show the application of the DCS technique to two hardware design contexts: component-based design and automatic error correction.

4.5.1 Component-based design

In hardware design, the reuse of "on-the-shelf" components is an important design methodology. Under this paradigm, designers usually start by choosing a bunch of components from a hardware "library", with respect to the given design specification; these components are assembled/interconnected to form an entire design; simulations and debug are performed to ensure the respect of the specification. This last step can be tedious and error-prone.

The (incremental) DCS and the supervisor implementation techniques can be very useful during a component-based design process: DCS can automatically generate a correct-by-construction supervisor which, once composed with the component assembly, yields a correct controlled system. Thus, in this context, DCS is used for *finalizing a partially designed system.*

As shown in figure 4.7, the hardware designer uses one or more libraries of reusable components. He/she picks the interesting components with respect to the functionality to build. These components are then connected with each other, according to their respective interface. Most frequently, the target functionality is not obtained by simply connecting such components together. On the one hand, corner-case configurations, violating the initial specification, can remain, or can be introduced by the component assembly which has been achieved. To avoid such erroneous configurations, additional logic usually needs to be added. On the other hand, the designer can choose to partially build his system, and terminate the design process with respect to a given specification by synthesizing a supervisor.

In both cases, a supervisor can either eliminate erroneous, corner-case configurations, or simply be the component terminating the design. To achieve this, the designer needs to designate a set of controllable input variables $\mathbf{c}$. The choice of the variables to be made controllable strongly depends on the designer's knowledge of the design at hand and of the specification to be enforced.

A supervisor is then generated, implemented and is finally connected to the system M. The designer may need to manually add additional logic, specifying how the auxiliary control inputs c^{env} can be used by the environment of the controlled system. The resulting system is then simulated: as DCS enforces a given specification by constraining the behavior of the controlled system, it is important to see if interesting behaviors have not been pruned. If this happens, then a different control attempt can be made, by choosing another set of controllable inputs.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

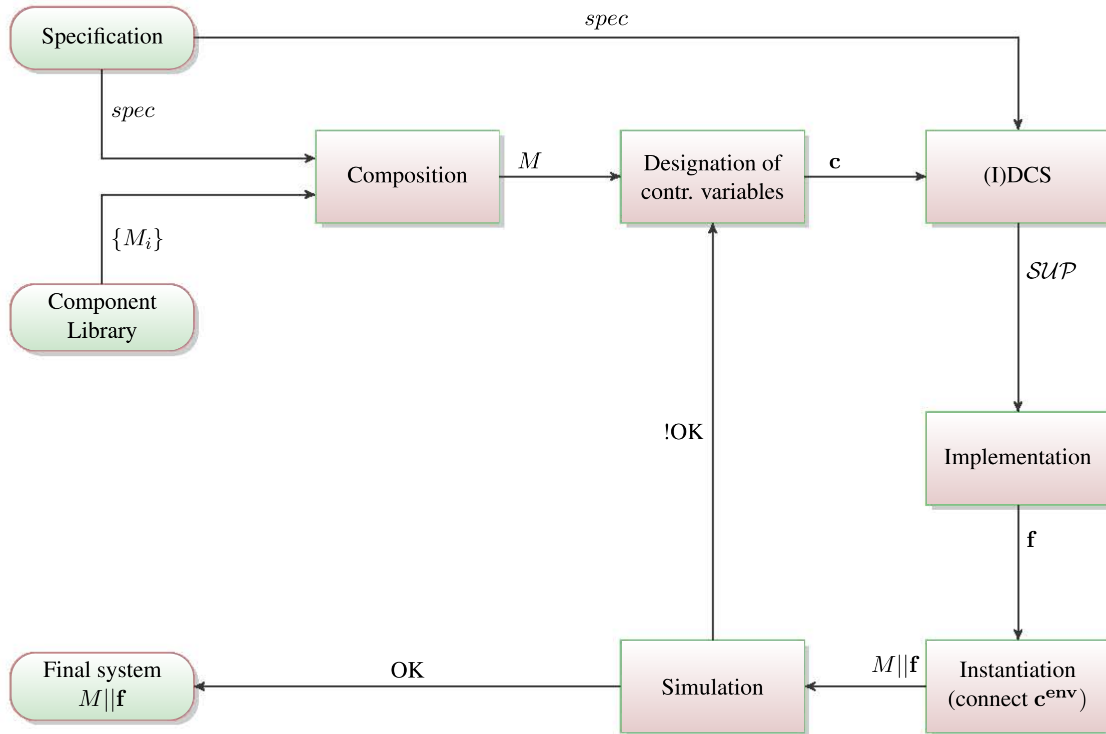


FIGURE 4.7: Component-based design flow

Example of the 3-way arbiter with token Let us apply our implementation approach to the 3-way arbiter with token, presented in section 3.5. The designer disposes of a collection cells $M_i, i = 1, 2, 3$. He wishes to connect them together to compose a 3-way arbiter satisfying the specification of mutual exclusion. The composition $M_1||M_2||M_3$ leaves the token management completely unmodeled. As the token mechanism is fundamental in achieving mutual exclusion, synthesizing a supervisor which controls the arrival of input token tin_1 can be a solution for automatically finalizing the design of the arbiter. We want to guarantee the mutual exclusion property by enabling token insertion only at appropriate moments.

The generated control function is too voluminous to be presented here. Figure 4.8 shows the entire design that satisfies the mutual exclusion specification. The supervisor is generated, and an auxiliary input is added for the controllable variable tin_1.

At any moment, the environment of the 3-way arbiter can *try* to insert a token via tin_1^{env}. The supervisor allows this token or filters it out, if its insertion may violate mutual exclusion.

Several possibilities exist for connecting tin_1^{env}. It can be required that a token is always inserted when possible. This is achieved by the assignment $tin_1^{env} = 1$. However, if there is no request, why should a token be inserted in the first place? Thus, another possibility would be to insert a token only when it is needed and when its insertion is possible:

$$tin_1^{env} = req_1 + req_2 + req_3,$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

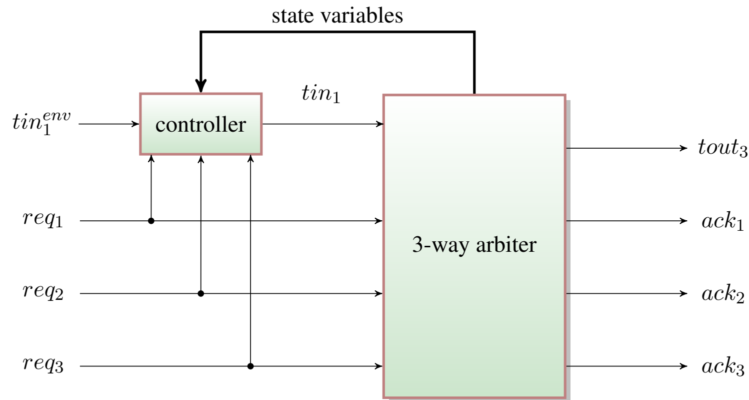


FIGURE 4.8: Implementation of the controlled 3-way arbiter

Any usage made of tin_1^{env} is safe with respect to mutual exclusion. However, an assignment such as $tin_1^{env} = 0$ would disable the client acknowledgment forever (which is still perfectly safe with respect to mutual exclusion). This is why we advocate the use of simulation in order to ensure the validity of the behavior of the controlled arbiter.

4.5.2 Automatic error correction

In the previous section, we discussed the application of DCS and supervisor implementation in the hardware design phase. However, if an error is discovered after the design or even after the manufacturing phase, it is often hard or even impossible to correct. We show that DCS can be successfully used for easily finding a correcting patch.

The design method is represented in Figure 4.9. The "suspected design" exhibits an unexpected behavior, which can be represented as a CTL accessibility property $\overline{spec} = EFp$. This property can be formally verified by model checking. The designer can even search for a witness for $\overline{spec}$ by evaluating $spec = AG\overline{p}$ and by generating a counter-example which shows that $\overline{spec}$ can happen. Then, he can attempt to enforce $spec$ by DCS. This method is illustrated by the following example.

4.5.2.1 A serial-parallel converter

The serial-parallel converter shown in figure 4.10 is composed of two existing components: a Serial Buffer (SB) and a Word Construction Buffer (WCB) which are composed by a synchronous product.

The two components operate as follows:

- the SB receives serial data as four-bit words $dwi_0, \cdots, dwi_3$ plus one parity bit dwi_4. After parity checking, it sends them to WCB;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

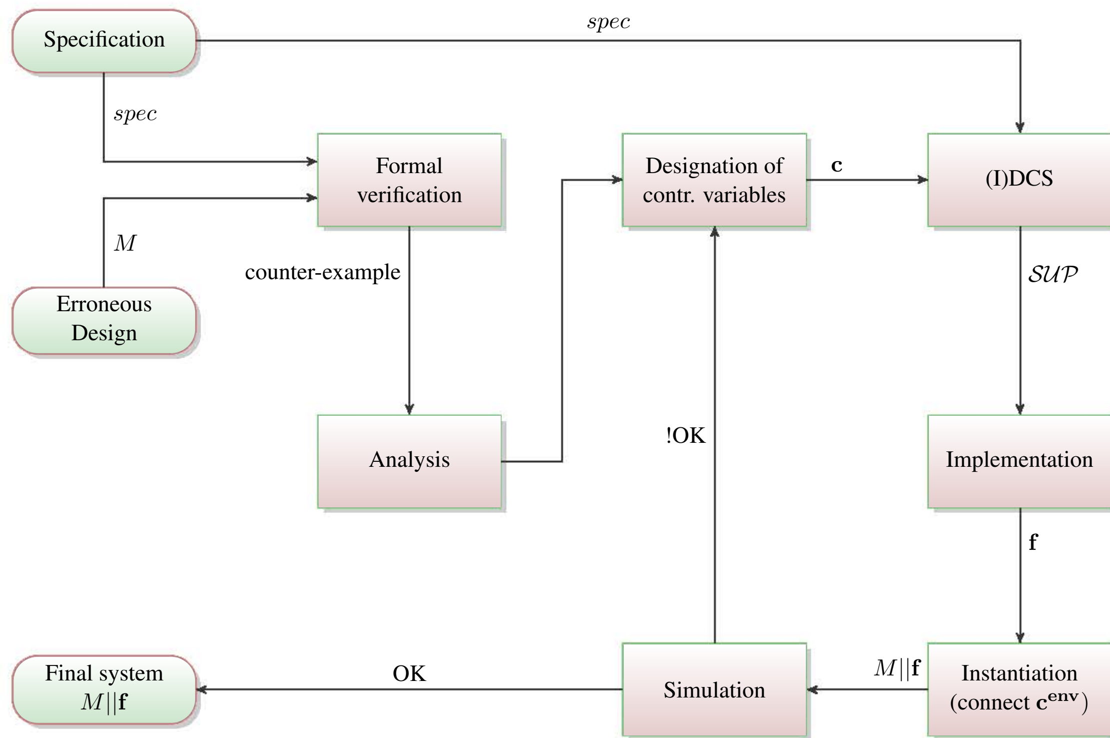


FIGURE 4.9: Error correction design flow

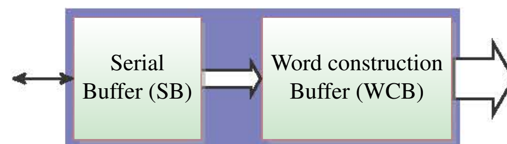


FIGURE 4.10: Serial-parallel converter

- the WCB receives 4-bit words and makes them up in bytes;
- as soon as four pieces of serial data have been received, the WCB outputs them in burst;

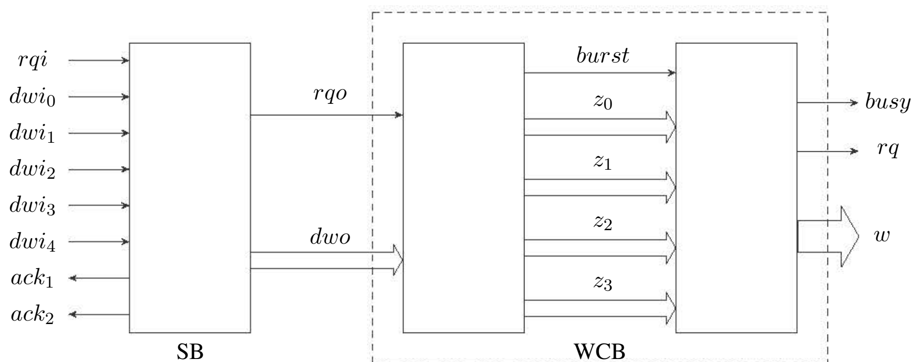


FIGURE 4.11: Architecture of the serial-parallel converter

This whole circuit can be decomposed into three function blocks as depicted in Figure 4.11. Block A represents the serial buffer (SB), and block B together with block C represents the word construction buffer (WCB). Block B accumulates incoming data, and constructs bytes out of 4-bit words. Block

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

C flushes the accumulated data as soon as block B is full. Signals burst and z_0, z_1, z_2, z_3 are internal variables of the WCB, thus we cannot observe or directly change their values from the outside.

Signal ack_1 and ack_2 are used to do the synchronization with the environment. If SB receives the data correctly, it will set $ack_1 = 1$ to signal the environment; if the parity check fails, SB sets $ack_2 = 1$ to signal the error so that the environment will resend. Otherwise, $ack_1 = ack_2 = 0$ meaning that no data has been received. As long as ack has not been set, the environment should maintain the request rqi and the data.

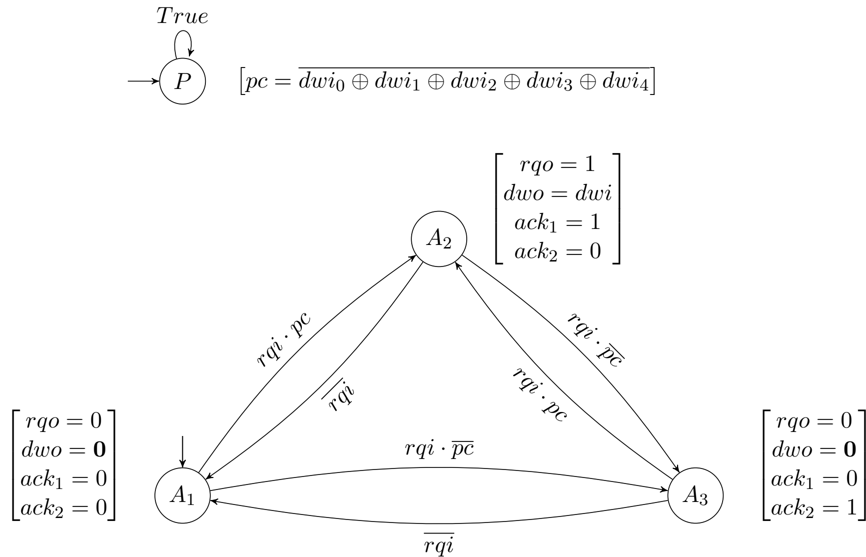


FIGURE 4.12: FSM model of block A

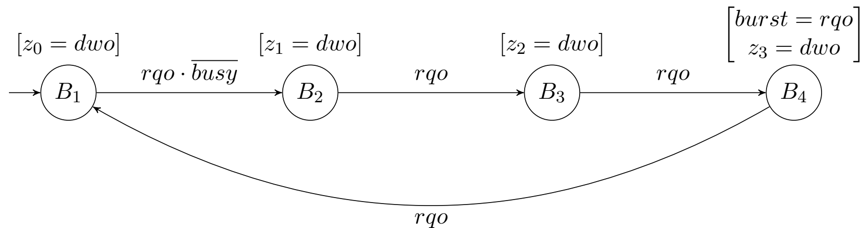


FIGURE 4.13: FSM model of block B

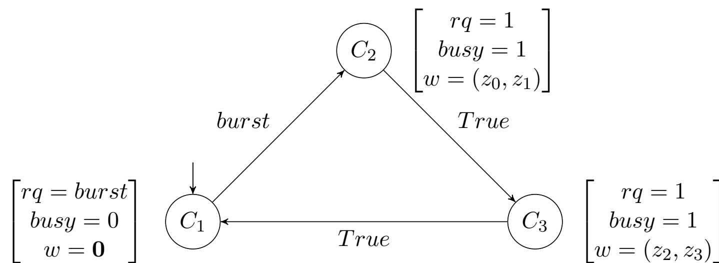


FIGURE 4.14: FSM model of block C

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Figure 4.12, 4.13, 4.14 show their corresponding FSM models. Output assignments are associated to states. The initial states are A_1, B_1 and C_1 respectively. The global FSM model of the serial-parallel converter is the synchronous product of modules A, B and C. Its behavior is the following:

- Block A (the serial buffer). When an input request rqi occurs, the input data is checked for parity errors. We model this functionality by an 1-state automaton which calculates and emits the parity check result pc. If the input data are correct, they are sent to the next component. Thus, the output request rqo signals the WCB the presence of valid data. If a parity error is found ($pc = 0$), the input data is not forwarded to the WCB, and an error code is sent to the external environment: $ack_1 = 0$, $ack_2 = 1$.
- Block B (the data accumulator). It maintains four internal buffers $z_0, \cdots, z_3$. Each time it receives an output request rqo from SB, it stores the data into one of the buffers in the order. New serial data can only be received if the system is not currently flushing parallel data. Otherwise, data may be lost. When in state $B4$, since buffer z_0, z_1 and z_2 are already filled with data, it generates a burst signal to tell block C to start flushing the accumulated data.
- Block C (the data-burst block). This is the part that sends the parallel data to the environment. In the initial state C_1, no data is output. On receiving a burst signal, it becomes busy and outputs the first byte (z_0, z_1). At the next time step, it outputs the second byte (z_2, z_3) and returns back to state C_1.

4.5.2.2 Correction analysis

One important property the serial-parallel converter should satisfy is that all pieces of serial data must be eventually retransmitted i.e. a piece of serial data is never lost. This property is expressed by the following temporal logic formula written in CTL [57]:

$$\textit{No data loss: } AG\ \overline{busy \cdot rqo}.$$

Thus, SB should never send a piece of data to the WCB during a flush. This property holds except for the following scenario:

- at some time t, WCB has almost filled its internal buffer except for one free slot of serial data;
- at the same moment, SB sends a piece of serial data to WCB and acknowledges a piece of input serial data simultaneously;
- at time $t + 1$, WCB is full and starts a flush.
- at the same moment SB sends the piece of serial data it has previously acquired;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- as WCB is busy and flushing, this piece of data is lost.

This scenario is confirmed by symbolic model-checking, which proves that property *"No data loss"* is false with the above counter-example.

Our goal is to ensure that property *"No data loss"* holds without rewriting neither SB nor WCB. We use symbolic DCS in order to synthesize a correcting "patch" [58] [59] that ensures the satisfaction of the above property.

4.5.2.3 Discrete Controller Synthesis

In order to apply DCS to the converter the designer must indicate which input variables are controllable. There is no a priori indication about the input to be controlled. However, it can be observed that by delaying the arrival of the input serial data adequately, the data loss can always be avoided. Technically, delaying the arrival of an incoming serial data amounts to delaying the arrival of the incoming request rqi. Thus, we choose rqi to be the controllable input variable. The remaining input variables of the system are considered uncontrollable.

The supervisor has been synthesized by the *Sigali* DCS tool [60]. Then, we have implemented the supervisor by generating a controller according to the Algorithm 5. The controlled converter architecture is represented in Figure 4.15 and is is easy to check that it corresponds to the generic target architecture shown in Figure 4.4.

The implementation algorithm has introduced the auxiliary environment variable rqi^{env}, corresponding to the controllable variable rqi. Note that the incoming request is now rqi^{env}, which is driven by the environment and read by the controller. According to the value of rqi^{env} and the internal state of the system (i.e. the serial-parallel converter), the controller assigns adequate values to rqi.

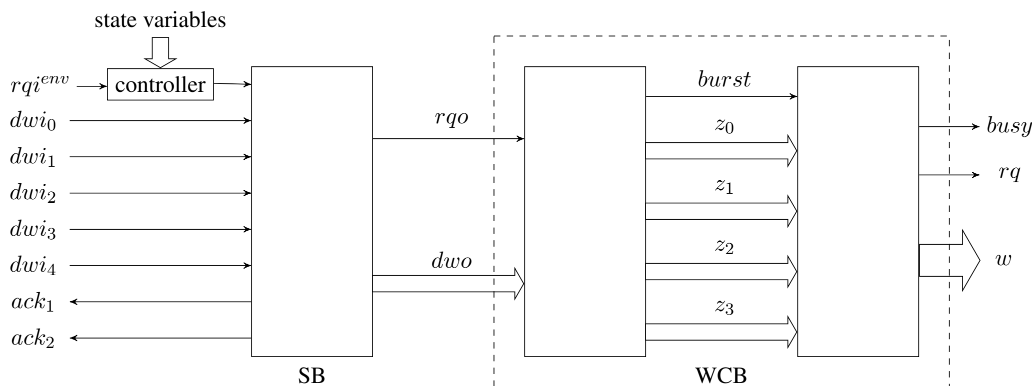


FIGURE 4.15: Error correction by DCS

Hence, the controller we obtain acts like a filter on the rqi^{env} input. When an incoming piece of serial data arrives, the controller either transmits it to the SB, or blocks it if that piece of data is likely to be lost. At any time t, if an incoming piece of serial data arrives, rqi^{env} is raised. Two scenarios are possible:

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- there remains only one free memory slot inside WCB, and this slot is about to be filled by SB with a piece of previously acquired data, i.e. data received at $t-1$. At this moment, if rqi^{env} is acknowledged, the data will be lost at $t+1$. However, upon reception of rqi^{env} the controller sets $rqi = 0$. Thus, incoming requests are not transmitted to the SB, and the environment will be acknowledged;

- there remains more than one free memory slot inside WCB. In that case when receiving rqi^{env} the controller sets $rqi = 1$. The incoming requests are transmitted to the SB so that they can be acknowledged.

Thus, the input transaction is delayed, in the sense that SB has no knowing about it as long as the data cannot be safely stored and retransmitted. Initially, incoming transactions were only used to ensure parity correction; by applying DCS, these transactions ensure in addition that the serial-parallel converter is ready.

Our controller shifts the control non-determinism towards the environment. When a non-deterministic choice is left on the value of rqi, this non-determinism is solved by the environment through rqi^{env}. By construction of our controller, the value assigned to rqi^{env} by the environment is directly fed to rqi.

Note that the input filtering mechanism established by addition of a bug correcting controller heavily relies on control non-determinism. Indeed, if the synthesized controller was completely deterministic then the functional decomposition would produce a closed-loop system with respect to input rqi, in which rqi would never be assigned by the environment.

4.6 Implementation

All examples presented in this document have been modeled using the Mode Automata language [61]. Models are then translated by the Matou compiler [61] into z3z, the Sigali input language. The supervisor was synthesized by the *Sigali* [60] symbolic DCS tool. Interactive simulations have been performed using the dedicated tool *SigalSimu*. The design flow used is illustrated in Figure 4.17.

Note that the synthesized supervisor constructed by *Sigali* is a Boolean equation, similar to equation 4.1. Due to its non-determinism, this supervisor can only be used for interactive simulation. This is achieved by the tool *Sigalsimu*. During simulation, whenever control non-determinism is present, a manual choice must be made by the user, as shown in Figure 4.16.

The supervisor implementation algorithm has been coded on the top of the CUDD library[62]. The supervisor generated by *Sigali* is exported in text format. The implementation algorithm reads the supervisor and produces a controller which is composed of several BDDs, each representing a control function. After the implementation step, the controlled system obtained can be verified using simulation tools or

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

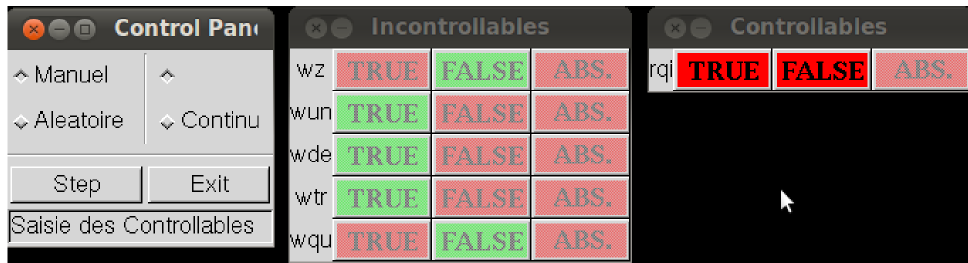


FIGURE 4.16: Manually solving non-determinism during interactive simulation: controllable rqi is clickable.

formal verification tools, and can also be fed to hardware synthesis tools for implementation on a hardware target such as an ASIC or a FPGA. This can be easily achieved by means of a simple syntactic translation of the control functions into any standard hardware description language.

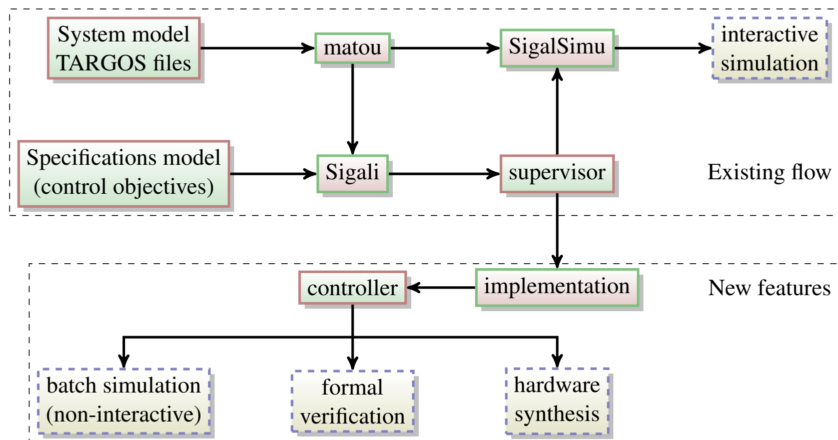


FIGURE 4.17: Design flow with supervisor implementation

4.7 Conclusion

Discrete Controller Synthesis has been shown to be useful as a complementary design tool besides simulation and formal verification. We have also presented and illustrated a symbolic supervisor implementation technique. This technique tackles the control non-determinism issue as well as the structural incompatibility of the supervisor with respect to the modular structure expected for it. A very important advantage is that our supervisor implementation conserves all the control solutions produced by DCS. This feature is very interesting with respect to straightforward determinization of a synthesized supervisor, which always produces a strict subsets of $\mathcal{SUP}$. Besides, it is to be noted that the implemented supervisor f is generally more compact than $\mathcal{SUP}$, which makes it easier to handle.

Still it may be argued about the introduction of the auxiliary variables c^{env} and about their physical correspondence in general. Actually, the control architecture we implement is a rather particular interpretation of the traditional control loop shown in Figure 4.1. The main difficulty comes from the translation of this control architecture to the hardware design context. The distinction between controllable/uncontrollable

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

variables is not natural. Input variables are generally dedicated to the environment; their encapsulation inside a control loop is not natural. However, the control architecture we propose leaves the input/output interface of the controlled system unchanged.

While symbolic DCS is less subject to state space explosion than enumerative DCS, its complexity still remains exponential in the number of state variables encoding the state of the system. Moreover, our symbolic implementation algorithm adds its own exponential complexity, which is not negligible. This is why, future research directions include developing heuristics for enhancing its performance.

The most important advantage of this technique is its ability to generate correcting patches automatically. Bug correction by rewriting code is sometimes physically impossible, especially when a hardware implementation has been produced and cannot be altered anymore. In such situations, the only architectural possibility is attempting a bug correction from “outside” of the physical component. This is what designers actually do manually, which is time-consuming, unsafe and error-prone. We believe that in such situations the method we propose can be extremely useful.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Chapter 5

Conclusion

This work advocates the introduction of the symbolic Discrete Controller Synthesis technique in the embedded hardware design flow, besides formal verification and simulation. Like formal verification, BDD-based DCS faces the same exponential theoretical complexity. We have proposed an incremental DCS (IDCS) technique which tackles the exponential complexity problem and is shown to give interesting performance results. IDCS is entirely based on the classical DCS algorithm and exploits the modular structure of the system to control.

As supervisors generated by DCS are not structurally compatible with hardware designs, a control architecture suitable for hardware design is proposed. A supervisor is built to control a system by acting through a set of inputs which are *designated* as *controllable*. Controlling an input actually amounts to filtering its values coming from the environment, according to the current state of the controlled system and to the control specification. Control non-determinism occurs when a choice is possible for the value of one or more controllable inputs. In our control architecture, this choice is always made by the environment, by assigning the auxiliary input variables.

In order to make this control architecture realizable, the supervisor is systematically transformed from its equational representation to a functional one. A design method has been proposed, based on this approach, and its specificities have been highlighted: either finalizing a design with respect to a specification, or generating a correcting patch for a design containing an error. This design method integrates DCS together with formal verification and traditional simulation. It has been illustrated on simple yet realistic hardware designs. A separate and more extensive study has also been conducted on the application of this method in the design process of an AHB bus (not represented in this document).

These contributions have been implemented and validated on a set of representative benchmarks, composed of a collection of modular systems, some of which exhibit explicit communication between modules. The performance results show interesting enhancements for both memory usage and execution time. However, the order in which IDCS should consider each module of a system must be user-specified. This

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

choice is important for the performance of IDCS. Finding a good order for IDCS is left as a future research direction for this work. Besides, like most BDD-based symbolic techniques, the performance of IDCS is sometimes impacted by fine-tuning strategies related to BDD manipulations, such as the variable ordering technique.

It ought to be noted that the DCS technique we have used only enforces invariance of a set of states. Hence, the only control specifications that can be handled are sets of states to enforce or to prohibit. Such control specifications encompasses CTL formulae, as each CTL formula has a corresponding set of states where it holds, and which can be computed symbolically by mu-calculus. Besides, CTL has been chosen because it is both familiar to hardware designers and richer than LTL. Desired behaviors can also be represented operationally, by means of finite state machines, also known as monitors, featuring a set of prohibited states (which are prohibited by DCS).

In this work we have not studied accessibility and liveness. Enforcing accessibility by DCS would be an interesting feature. This could be used to guarantee, for instance, that some desired functionalities can always be activated, or, as it is the case for manufacturing control systems, that a termination state can always be reached.

Liveness is an even more important aspect in hardware design. Unlike accessibility, which guarantees possibility, enforcing liveness would enforce the functionality of the system to control: a desired behavior could be forced to occur within finite time, which makes its occurrence certain instead of possible. Such a property is of great interest in hardware design, just like liveness coupled with fairness constraints. To the best of our knowledge, few DCS techniques are able to enforce liveness. Actually, the "program synthesis" technique has been developed independently of the supervisory control theory, and also applied in the electronic design automation area [63] in order to generate designs directly from their LTL logical specification. This technique is designed for rapid system prototyping and cannot achieve control of an existing system. Besides, it is possible to implement liveness by applying *optimal synthesis* [64]. However, this technique does not take into account fairness.

Thus, future research directions for this work include developing a DCS algorithm able to enforce liveness coupled to fairness constraints on an existing system, and studying the possible enhancements that can be achieved by developing a suitable incremental or compositional DCS approach. The supervisor implementation algorithm proposed in this document is equally subject to combinational explosion, due to its exponential complexity related to BDD manipulation. Another important future research direction wold be to enhance its performance.

Finally, it seems clear that IDCS can easily cooperate with existing compositional DCS techniques. Within a compositional DCS process, IDCS can accelerate the computation of local control solutions. On the other hand, IDCS benefits from a compositional approach by operating only on modules of limited size. Developing a DCS approach integrating incremental and compositional techniques can lead to important enhancements of DCS performance.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Appendix A

Résumé

A.1 Introduction à la SCD Incrémentale

Les efforts déployés pour appliquer la SCD aux conceptions de taille industrielle affronte la même difficulté: système mène à l'explosion en temps et/ou en mémoire pendant le calcul du superviseur. La SCD est implémentée à la base des algorithmes symboliques qui ont la complexité spatiale exponentielle du nombre de variables Booléennes ils manipulent. Cette complexité est inhérente aux manipulations des Diagrammes de Décision Binaire, qui sont fondamentales dans l'implémentation des algorithmes de parcours symbolique. Parcours symbolique basé sur BDD est une technique générique utilisée pour résoudre formellement beaucoup de problèmes liés aux systèmes d'état/transition Booléenne. En particulier, model-checking symbolique de CTL consiste d'une collection d'algorithmes de parcours symbolique qui produisent l'ensemble d'états où une formule est vraie, et vérifient si le résultat contient l'état initial. Problèmes de la SCD profite aussi de la technique de parcours symbolique: comme illustré dans Chapter 2, un superviseur est construit (si il existe) comme un ensemble de toutes les solutions possibles de contrôle satisfaisant la spécification de contrôle.

A.1.1 Evolution de l'usage de mémoire pendant la SCD.

Il peut être observé que pendant la synthèse, un pic de l'usage de mémoire apparaît souvent pendant le calcul des résultats intermédiaires. Nous le montrons par un exemple. Le modèle est décrit en détail dans la section 3.7.2. Dans l'illustration, la procédure de synthèse se fait comme suivant:

1. identifier l'ensemble d'états qui satisfont la propriété de contrôle;
2. calculer l'ensemble d'invariant sous contrôle. Ce processus est itératif. Il calcule les prédécesseurs contrôlables de l'ensemble obtenu dans la itération précédente et continue jusqu'à un point fixe;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

3. extraire un contrôleur depuis cet invariant sous contrôle.

Pendant chaque calcul d'un ensemble contrôlable d'états prédécesseur, le nombre de BDDs est enregistré. L'évolution de l'usage de mémoire en termes de nombre de BDD pendant la synthèse est illustrée dans la Figure A.1. Nous démarrons avec un petit modèle qui a seulement 4000 noeuds BDD. Le premier calcul de prédécesseur aboutit à un pic qui atteint 35.000 de noeuds BDD. Les 4 calculs suivants tombent à des nombres moindres de noeuds. Le calcul final est la génération d'un superviseur du $\mathcal{IUC}$, qui utilise 15.000 noeuds. Depuis cet exemple, nous notons qu'un modèle de taille petite peut mener un usage haut de mémoire et le pic de l'usage de mémoire apparaît souvent dans le milieu du calcul du $\mathcal{IUC}$. Ce pic de l'usage de mémoire est causé par la complexité exponentielle de la SCD basé sur BDD. En général, ce pic est critique vis-à-vis la faisabilité de la SCD pour les conceptions réelles. Tout ordinateur a une ressource bornée de mémoire, alors qu'il est important d'attaquer l'explosion exponentielle, pour que cette borne soit jamais traversée, ou au moins retardée.

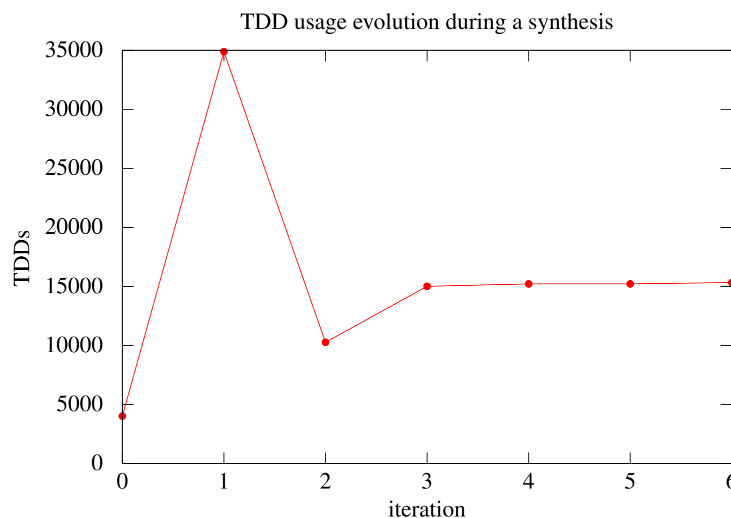


FIGURE A.1: Pic de l'usage de mémoire pendant la SCD

Afin d'éviter une explosion combinatoire, une solution alternative a été récemment proposée. Il préconise l'utilisation de algorithmes basés sur SAT. Ainsi, pour établir la vérité d'une formule, il suffit de rechercher d'une solution satisfaisante, au lieu de calculer l'ensemble des toutes les solutions possibles. Cette solution est très efficace dans la vérification formelle, mais pas possible dans la SCD. Par définition, un superviseur devrait être maximalement permissive: il doit inclure toutes les solutions possibles de contrôle. Ainsi, les algorithmes symboliques restent fondamentales pour la SCD.

La méthode largement utilisée pour surmonter l'explosion combinatoire lors de parcours symbolique est la décomposition, à la suite de l'approche " diviser pour régner". En particulier, la décomposition modulaire selon la structure modulaire d'un système, est une approche naturelle et intuitive exploitée dans plusieurs contributions de la recherche sur la SCD.

Cette technique suit deux étapes: (1) calculer une solution approximative de contrôle pour chaque module. Cela est efficace, que la SCD agit localement sur chaque module, (2) recombiner des solutions

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

approximatives locales dans une solution globale exacte. Les techniques SCD modulaires dont nous sommes conscients s'appliquent aux modèles basés sur les événements qui n'ont pas de communication explicite entre modules: le mécanisme de synchronisation entre les modules ne se fait par les entrées partagées contrôlables/incontrôlables.

Notre contribution est une procédure *incrémentale*, plutôt que modulaire, de synthèse de superviseur. Cette technique profite également de la structure modulaire du système, et prend en compte les communication explicite: chaque module a une interface d'entrée / sortie, et les interconnexions sont possibles selon un modèle de communication présentés ci-dessous.

Pour un système comportant deux ou plusieurs modules, la SCD incrémentale est un processus itératif. Il commence par une abstraction initiale: tous, sauf un modules sont abstraits . Ils sont réincorporés progressivement au cours des itérations suivantes:

- les SCD approximatives. Cela constitue un invariant sous contrôle qui est une sur-approximation (un sur-ensemble) de l'invariant exact sous contrôle;
- raffinement incrémental: rajouter un module au système abstrait, parmi ceux précédemment abstraits.

Lorsque le raffinement progressif ajoute le dernier module, une dernière étape de SCD est appliquée à l'ensemble du système et produit une solution exacte. Cette procédure est illustrée à la Figure A.3. Dans le ci-après, la technique incrémentale de SCD (SICD) est présentée en détail. Deux exemples sont utilisés pour illustrer cette technique. Un petit exemple est présenté dans la section A.4. Il dispose de deux modules et est utilisé pour illustrer toutes les étapes SICD sur un système simple. Pour la raison de lisibilité, cet exemple n'a pas de communication interne entre les modules. Puis, un plus grand exemple, présenté dans la section A.5, illustre l'application de la SICD sur une conception plus réaliste concernant trois modules communicants.

A.2 Définitions

Dans cette section, nous définissons la notion de FSM contrôlable communiquant, basé sur le modèle FSM échantillonné booléenne avec des sorties, présentés dans le chapitre 2.

A.2.1 Machines d'état fini contrôlable (CFSM)

Le modèle FSM introduit dans le chapitre 2 n'est pas suffisant pour tenir compte de la communication entre les modules. Un attribut appelé "entrée de l'interface" est ajouté au modèle FSM.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Une FSM Booléenne contrôlable modulaire (CFSM) M est définie comme un tuple:

$$M = (Q, I, L, \delta, q_0, PROP, \lambda),$$

tel que:

- Q : un ensemble fini de n états $\{q_1, q_2, \ldots, q_n\}$;
- I : un ensemble de variables Booléennes, tel que $I = U \cup C$ et $U \cap C = \emptyset$:
  - $U = \{u_1, \cdots, u_m\}, m \geq 0$: l'ensemble de variables d'entrée incontrôlable;
  - $C = \{c_1, \cdots, c_p\}, p \geq 0$: l'ensemble de variable d'entrée contrôlable;
- $L = \{l_1, \cdots, l_r\}, r \geq 0$: un ensemble d'entrée d'interface;
- $\delta : Q \times \mathbb{B}^{m+p+r} \to Q$ est la fonction de transition de M;
- $q_0 \in Q$ est l'état initial de M;
- $PROP = \{p_1, \cdots, p_v\}, v \geq 0$: un ensemble de v propositions Booléennes atomiques;
- $\lambda : Q \to \mathbb{B}^v$ est la fonction d'étiquetage.

Remarque Les fonctions λ et λ^{-1} sont identiques à celles définies dans le Chapter 2. Dans la suite, nous notons $\mathbf{u}, \mathbf{c}, \mathbf{l}$, les vecteurs contenant les variables de U, C, L, respectivement.

A.2.2 Composition synchrone

Le produit synchrone entre deux FSM contrôlables communiquants est définie par rapport au problème de SCD à résoudre. Un système de M dispose d'une interface d'entrée/sortie. Les entrées sont soit contrôlables ou incontrôlables. Cependant, M peut avoir une structure hiérarchique *interne* : il peut être composé de deux ou plus de modules communiquants. Chacun d'entre eux peut présenter des entrées contrôlables et/ou incontrôlables , mais aussi des entrées *d'interface* dédiées aux communication internes avec les autres modules de M. Cette structure est illustré dans la figure A.2.

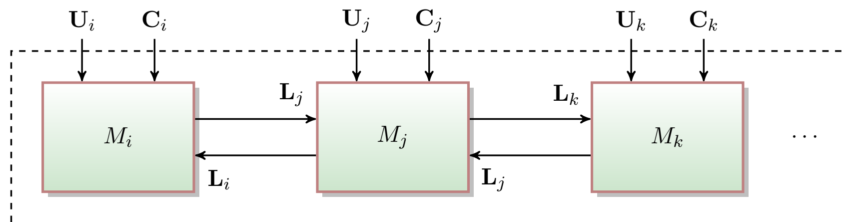


FIGURE A.2: Structure modulaire d'un système contrôlable

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Notons $M = ||_{i=1}^{K} M_i$ la composition synchrone de K modules communiquants $M_i = (Q_i, I_i, L_i, \delta_i, q_{0i}, PROP_i, \lambda_i), i = 1, \cdots, K$. Dans la suite, nous supposons qu'une sortie ne peut pas être conduite par plus d'un CFSM à l'intérieur d'un produit synchrones:

$$\forall i, j = 1, \cdots, K, i \neq j : PROP_i \cap PROP_j = \emptyset.$$

Nous supposons également que les ensembles de variables contrôlables et incontrôlables sont cohérents par rapport à la composition synchrone: si une entrée est partagée entre M_i et M_j, elle ne peut pas être contrôlable dans M_i et incontrôlable dans M_j.

Selon la structure hiérarchique de M, l'ensemble des entrées de l'interface du module M_i peut être partitionné en $K-1$ sous-ensembles disjoints $\{L_i^j | j = 1, \ldots, K, i, j \neq\}$ avec

$$L_i^k \cap L_i^j = \emptyset, (\forall k, j = 1, \cdots, K, j \neq k)$$

et

$$\bigcup_{j=1,\cdots,K, j\neq i} L_i^j = L_i$$

et

$$0 \leq |L_i^j| \leq |L_i|.$$

L'ensemble des entrées de l'interface L_i^j appartient à M_i et est connecté à des sorties de M_j.

Soit $PROP_i^{out_j} \subseteq PROP_i$ l'ensemble des propositions de sortie de M_i qui sont connectées au module M_j via L_j^i. De même, les fonctions de sortie λ_i de M_i sont partitionnées en fonction de leur connectivité. Soit $(\lambda_i^{j,k}), 0 \leq k \leq |L_j^i|$ les sorties de M_i reliées à M_j.

Le produit synchrone des CFSMs $M = ||_{i=1}^{K} M_i$ est défini comme ci-dessous:

- $Q = Q_1 \times \cdots \times Q_K$;
- $I = I_1 \cup \cdots \cup I_K$, tel que $I_i = U_i \cup C_i$;
- $L = \emptyset$;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- $\delta : Q \times \mathbb{B}^{m_1+p_1+r_1} \times \cdots \times \mathbb{B}^{m_K+p_K+r_K} \to Q$ est défini pas:

$$\delta = \Big(\delta_1\big(q_1, \mathbf{i_1}, \quad \lambda_2^{1,1}(q_2), \cdots, \lambda_2^{1,|L_1^2|}(q_2), \lambda_3^{1,1}(q_3), \cdots, \lambda_3^{1,|L_1^3|}(q_3), \cdots\big), \quad \cdots, \quad \delta_K\big(q_K, \mathbf{i_K}, \quad \lambda_1^{K,1}(q_1), \cdots, \lambda_1^{K,|L_K^1|}(q_1), \lambda_2^{K,1}(q_2), \cdots, \lambda_2^{K,|L_K^2|}(q_2), \cdots\big) \Big);$$

- $q_0 = (q_{01}, \cdots, q_{0K})$;
- $PROP = PROP_1 \cup \cdots \cup PROP_K$;
- $\lambda : Q \to \mathbb{B}^{\sum_{i=1}^{K} |PROP_i|}$ est défini par:

$$\lambda\big((q_1, \cdots, q_K)\big) = \big(\lambda_1(q_1), \cdots, \lambda_K(q_K)\big).$$

La définition et les propriétés de la fonction inverse λ^{-1} reste inchangées.

Dans la suite, nous définissons d'abord la procédure de SCD incrémentale , et ensuite illustrer chacune de ses démarches auprès d'un arbitre simple à deux voies comme un exemple dans la section A.4. Les résultats de SICD sont comparés à ceux obtenus par la application direct de SCD.

A.3 Technique de SCD incrémentale

La procédure de la SCD incrémentale (SICD) fonctionne sur les mêmes entrées que la technique SCD direct, mais nécessite une indication complémentaire spécifiée par l'utilisateur: un ordre des modules qui constituent le système global. La SICD se fait itérativement: pour un système contenant $K \geq 2$ modules, il faut K étapes, un pour chaque module, avec $K!$ ordres d'application possible. Nous soutenons qu'un ordre des modules peut être spécifié par l'utilisateur , et déterminé en fonction de la structure et de la connectivité du système global.

Tout comme la SCD direct, la SICD renforce l'assertion $AG(spec)$, où $spec$ est une proposition Booléenne exprimée sur l'ensemble $PROP_1 \cup \cdots \cup PROP_K$ de M en utilisant les connecteurs classiques Booléennes " $\cdot$" (conjonction), " $+$ " (disjonction) et " $\overline{p}$ " (négation). La SCD incrémentale fonctionne comme ci-dessous.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Tout d'abord, il construit un modèle abstrait $M_1||M_2^{abs}||\cdots||M_K^{abs}$. L'abstraction remplace $M_2,\cdots,M_K$ par un plus permissive FSM modèle abstrait, qui modélise tous les comportements possibles pour les sorties pilotées initialement par $M_2\cdots,M_K$, et ayant une influence soit sur le comportement de M_1 ou sur la satisfaction de la $spec$. Un tel modèle FSM abstrait pour une variable donnée x est un automate non-déterministe de deux états, permettant toutes les transitions et affirmant $x=0$ ou $x=1$, selon son état actuel. Modules $M_2^{abs},\cdots,M_K^{abs}$ sont dits être un *loose*, à savoir l'environnement non-restrictif pour M_1, dans le sens où il permet plus de comportements des sorties de $M_2,\cdots,M_K$ en réalité.

Le nombre de machines abstraites à états finis obtenues dépend du nombre des variables de sortie partagées avec M_1 et $spec$. Pour m variables partagées, l'environnement non-restrictif $M_i^{abs}, i=2,\cdots,K$ modélise toutes les valeurs possibles de ces variables, et a donc m^2 états. Ainsi, le gain obtenu par l'abstraction dépend fortement de la connectivité entre M_i avec M_1 et $spec$, et le modèle concret de M_i lui-même. L'opération d'abstraction est formalisée dans la section suivante.

Deuxièmement, une solution intermédiaire approximative de contrôle $\mathcal{IUC}_1^{abs}$ est synthétisée pour $spec$ sur le modèle abstrait du système $M_1||M_2^{abs}\cdots M_K^{abs}$. Ce résultat est obtenu en appliquant la SCD et profite d'une diminution du coût de calcul, car elle opère sur une plus petite modèle, en fonction de la taille de l'abstraction précédemment atteint.

Finalement, l'abstraction est partiellement raffinée: M_2^{abs} est remplacé par M_2. Le résultat intermédiaire $\mathcal{IUC}_1^{abs}$ est utilisé comme le point de départ pour la synthèse d'une nouvelle solution de contrôle pour $spec$ et $M_1||M_2||M_3^{abs}\cdots||M_K^{abs}$. Modules $M_3\cdots M_K$ sont ajoutés progressivement, et les résultats intermédiaires successifs sont calculés.

La dernière étape fonctionne sur le modèle du système entier. Il est souhaité de bénéficier des calculs réalisés sur les modèles abstraits. La technique SICD est illustrée à la figure A.3 pour $K=2$, par rapport à l'application directe de la SCD.

A.3.1 Abstraction

Etant donné une machine à états finis M, et une proposition correspondante Booléenne $p\in PROP$. L'abstraction de M par rapport à p est une FSM non-déterministe dont les états satisfont soit p ou $\overline{p}$, mais pas les deux en même temps:

$$abs(M,p)=(Q^{abs},I^{abs},L^{abs},\delta^{abs},Q^{abs},PROP^{abs},\lambda^{abs})$$

où

- $Q^{abs}=\{q_p,\tilde{q_p}\}$, $I^{abs}=\emptyset$ et $L^{abs}=\emptyset$;
- $\delta^{abs}:Q^{abs}\rightarrow 2^{Q^{abs}}$ est défini par: $\delta^{abs}(q_p)=\delta^{abs}(\tilde{q_p})=Q^{abs}$;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

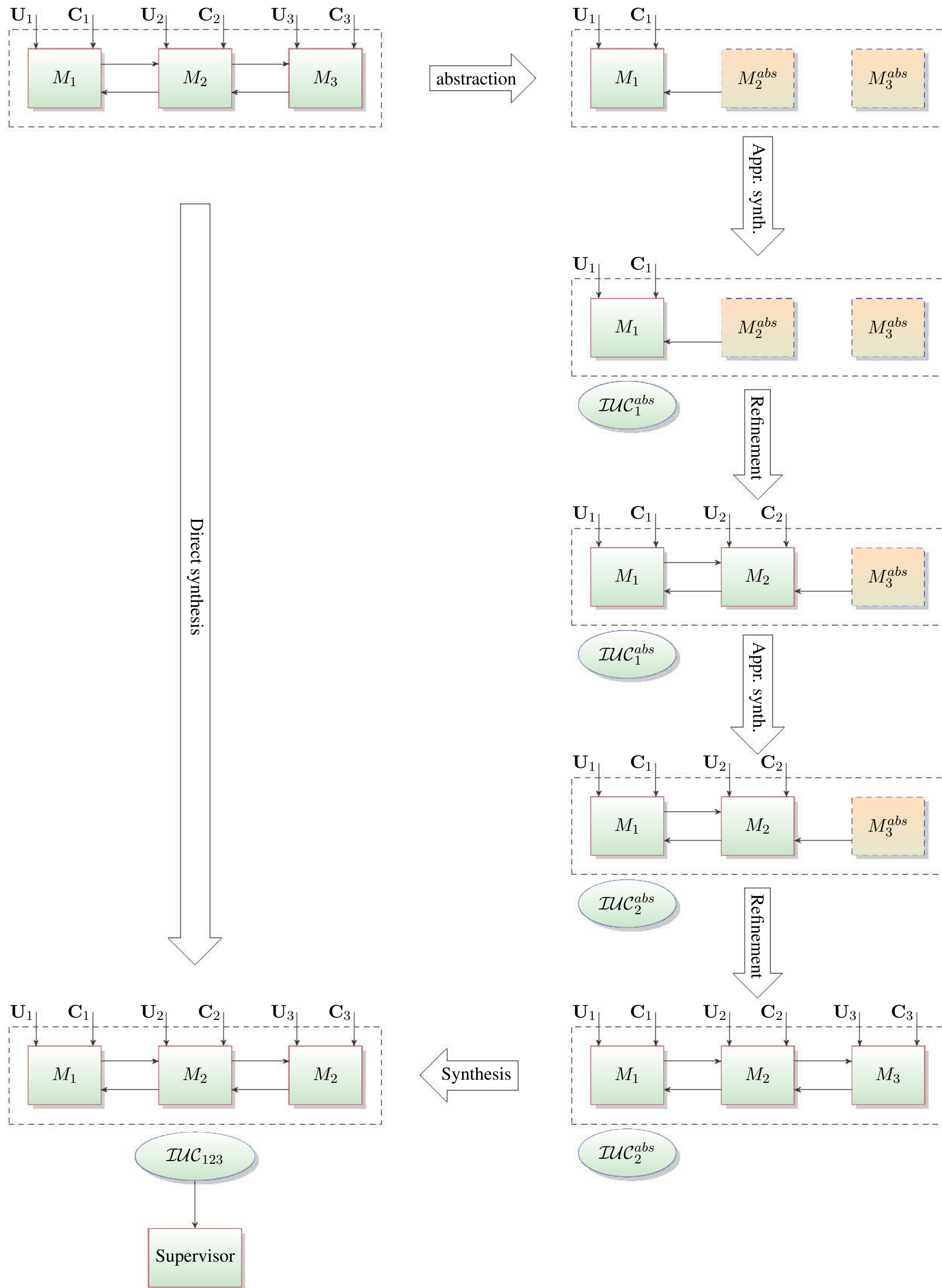


FIGURE A.3: La SCD incrémentale versus la SCD direct

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- l'état initial peut être n'importe quel entre Q^{abs};
- $PROP^{abs} = \{p\}$;
- $\lambda^{abs} : Q^{abs} \rightarrow \mathbb{B}$ est définie par:
  $\lambda^{abs}(q_p) = 1$ and $\lambda^{abs}(\tilde{q_p}) = 0$.

L'abstraction de M avec l'égard d'un sous-ensemble $PROP^m \subset PROP$, avec $1 \leq m \leq |PROP|$ est définie par le produit synchrone des abstractions individuels définies sur $p_i \in PROP^m$:

$$Abs(M, PROP^m) = \|_{i=1,\cdots,m}\ abs(M, p_i).$$

Cette opération abstrait à l'extérieur tous les comportements de M, et produit un modèle FSM qui reproduit tous les 2^m assertions possibles pour les propositions de $PROP^m$ ainsi que toutes les séquences possibles des assertions.

Comme indiqué ci-dessous, la SICD applique l'opération d'abstraction itérativement. Pour un système $M = ||_{i=1}^{K} M_i$, chaque itération j de la technique incrémentale de la SCD abstrait les modules $M_j, \cdots, M_K$ selon leur connectivité aux modules $M_1, \cdots, M_{j-1}$ et aux propositions atomiques $SPEC_j \in PROP_j$ utilisées pour exprimer $spec$. Cette opération est définie par:

$$\mathcal{ABS}(M, j, spec) = ||_{k=1}^{j-1} M_k || \big(||_{k=j}^{K} Abs(M_k, \bigcup_{l=1}^{k-1} PROP_k^{out_l} \cup SPEC_k) \big).$$

A.3.2 Raffinement d'ensemble d'états abstraits

L'opération de raffinement projette un ensemble d'états abstraits aux états d'origine précédemment abstraits par $\mathcal{ABS}$. Le raffinement des états abstraits par rapport au module M_j est une fonction qui projette un sous-ensemble d'états Q^{abs_j} de $\mathcal{ABS}(M, j, spec)$ à l'ensemble $Q^{abs_{j+1}}$ de $\mathcal{ABS}(M, j+1, spec)$ par rapport à la satisfaction d'un ensemble de prédicats PROP.

Elle est définie par:

$$ref(q^{abs_j}, PROP, Q^{abs_{j+1}}) = \{q^{abs_{j+1}} \in Q^{abs_{j+1}} | \forall p \in \text{PROP}, \\ q^{abs_j} \in \lambda^{-1abs_j}(p) \rightarrow q^{abs_{j+1}} \in \lambda^{-1abs_{j+1}}(p)\},$$

où $\lambda^{-1abs_j}(p)$ et $\lambda^{-1abs_{j+1}}(p)$ sont les fonctions inverses d'étiquetage de $\mathcal{ABS}(M, j, spec)$ et $\mathcal{ABS}(M, j+1, spec)$ respectivement.

Le raffinement d'un ensemble d'états abstraits Q^{abs} est défini par:

$$Ref(Q^{abs_j}, PROP, Q^{abs_{j+1}}) = \bigcup \{ref_j(q^{abs_j}, PROP, Q^{abs_{j+1}}) | q^{abs} \in Q^{abs}\}.$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

A.3.3 L'algorithme de la SCD incrémentale

L'algorithme de la SICD commence par la description de M et la spécification $spec$ à respecter. Il faut aussi un ordre entre les modules de $M = M_1||M_2||\cdots||M_K$, qui doit être appliqué pendant les étapes de raffinement successif. A chaque itération j, une abstraction $\mathcal{IUC}(M, j, spec)$ est calculée par rapport à:

- les sorties de $M_{j+1}\cdots M_K$ qui sont connectées aux entrées locales de $M_1\cdots M_j$;
- l'ensemble de propositions atomiques de $M_{j+1}\cdots M_K$ utilisées pour exprimer $spec$.

Algorithm 6 L'algorithme de la SICD

1: {entrée:

- $M = M_1||M_2||\cdots||M_K, (K \geq 2)$, le système à contrôler;
- $spec$ la spécification à renforcer;

sortie: $\mathcal{IUC}^{inc}$, l'ensemble d'invariant sous contrôle pour M et $spec$ }

2: $M^{abs} \leftarrow \mathcal{ABS}(M, 2, spec)$

3: $\mathcal{I} \leftarrow \lambda^{-1abs}(spec)$

4: $\mathcal{I} \leftarrow \mathcal{IUC}(M^{abs}, \mathcal{I})$

5: $j \leftarrow 3$

6: **while** $\mathcal{I} \neq \emptyset$ **and** $j \leq K$ **do**

7: $M^{abs} \leftarrow \mathcal{ABS}(M, j, spec)$

8: $\mathcal{I} \leftarrow Ref(\mathcal{I}, \{spec\}, Q^{abs})$

9: $\mathcal{I} \leftarrow \mathcal{IUC}(M^{abs}, \mathcal{I})$

10: $j \leftarrow j + 1$

11: **end while**

12: **if** $\mathcal{I} \neq \emptyset$ **then**

13: {L'ensemble final d'invariant sous contrôle est projeté aux états Q de M:}

14: $\mathcal{I} \leftarrow Ref(\mathcal{I}, \{spec\}, Q)$

15: $\mathcal{IUC}^{inc} \leftarrow \mathcal{IUC}(M, \mathcal{I})$

16: **end if**

Algorithme 6 présente l'algorithme de la SICD. Chaque itération j calcule une solution approximative de contrôle à un système abstrait $\mathcal{ABS}(M, j, spec)$. Chaque itération produit un résultat intermédiaire utilisé à l'itération suivante. A la fin, une étape finale de la SCD est effectuée sur l'ensemble du système.

Les informations sur les performances et la mise en œuvre de la technique de SICD sont commentées en détail dans la section A.6. Dans la suite, nous établissons que l'algorithme de SICD produit également un superviseur maximalement permissive.

Theorem A.1. *Les algorithmes de la SCD et de la SICD produisent le même résultat.*

Proof. La déclaration ci-dessus est vraie ssi la synthèse incrémentale et la synthèse directe produisent les même ensembles d'invariant sous contrôle.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Rappelons que pour tout problème de SCD, nous avons $\mathcal{IUC} \subseteq \lambda^{-1}(spec)$.

Soit $\mathcal{IUC}$ le résultat produit par la SCD directe et $\mathcal{IUC}^{inc}$ le résultat produit par l'algorithme SICD. On peut observer que $\mathcal{IUC}^{inc} \subseteq \mathcal{IUC}$. En effet, la dernière étape de la SICD opère sur l'ensemble du système M, il tente de faire invariant l'ensemble $\mathcal{I}$, raffiné à l'itération $j = K$ par rapport à $spec$. Par construction, tous ces états sont inclus dans Q^M et satisfont $spec$. Ainsi, $\mathcal{I} \subseteq \lambda^{-1}(spec)$. La dernière application de la SCD calcule $\mathcal{IUC}^{inc} \subseteq \mathcal{I}$ en faisant invariant l'ensemble $\mathcal{I}$.

Donc, d'une part, la SCD directe rend invariant l'ensemble $\lambda^{-1}(spec)$ sur M et produit l'ensemble $\mathcal{IUC} \subseteq \lambda^{-1}(spec)$. D'autre part, la dernière étape de la SICD commence par un sous-ensemble de $\lambda^{-1}(spec)$: il fait invariant l'ensemble $\mathcal{I} \subseteq \lambda^{-1}(spec)$ sur M et produit $\mathcal{IUC}^{inc}$.

Maintenant, considérons l'état $q \in \mathcal{IUC}^{inc}$; q est contenu dans $\lambda^{-1}(spec)$. Supposons que q n'est pas un élément de $\mathcal{IUC}$. Cela signifie que la SCD directe a enlevé l'état q. La dernière étape de la SICD effectue une SCD directe ordinaire sur M, ce qui rend invariant l'ensemble des états $\mathcal{I}$, contenant q. Ainsi, l'état q devrait aussi être enlevé par la SICD et ne devrait pas être inclus dans $\mathcal{IUC}^{inc}$. Donc, il est vrai que $\forall q \in Q^M : q \in \mathcal{IUC}^{inc} \implies q \in \mathcal{IUC}$. Nous pouvons conclure que $\mathcal{IUC}^{inc} \subseteq \mathcal{IUC}$.

Supposons maintenant que l'inclusion ci-dessus est stricte. Cela signifie que il existe au moins d'un état $q \in Q^M$ tel que $q \in \mathcal{IUC}$ et $q \notin \mathcal{IUC}^{inc}$. L'état q a été enlevé par l'algorithme de la SCD incrémental: il existe une transition sortant q et menant à l'extérieur de $\mathcal{IUC}^{inc}$ pour une valeur incontrôlable étant donnée et pour toute valeur prise pour les variables contrôlables. Toutefois, si une telle transition existe, l'état q serait également être enlevé par la SCD depuis $\mathcal{IUC}$. Par conséquent, nous concluons que $\mathcal{IUC}^{inc} = \mathcal{IUC}$.

□

A.4 Premier exemple: un modèle d'arbitre

L'exemple présenté est inspiré de [36]. Il modélise un arbitre qui gère l'accès exclusif à une ressource partagée par deux clients indépendants. Ce modèle se concentre uniquement sur la fonctionnalité de la gestion des l'accès. La ressource partagée réelle ainsi que son interaction avec chaque client sont laissées non modélisées.

La gestion de d'accès est modélisée par deux machines concurrent à états, $M1$ et $M2$, comme montré par la Figure A.4. Chaque machine M_i, $(i = \{1, 2\})$, reçoit une demande req_i d'un client et émet un accès correspondant gnt_i au client. Quand elle reçoit une demande d'accès req_i, M_i soit entre dans l'état *waiting* W_i ou dans l'état *granting* G_i. Chaque client a la possibilité de rétracter sa demande, par l'émission de $stop_i$ avant qu'il ne soit accordé. La décision d'accès est prise par la valeur des entrées go_i; chaque machine M_i peut rester à l'état *waiting* aussi longtemps que go_i est *false* et aussi longtemps que le client demandeur ne se rétracte pas sa demande, par l'émission de $stop_i$. A l'état G_i, M_i emet

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

gnt_i à son demandeur. L'accès reste acquis pour le client i, jusqu'à l'issu de $stop_i$ et il retourne à l'état *idle* I_i.

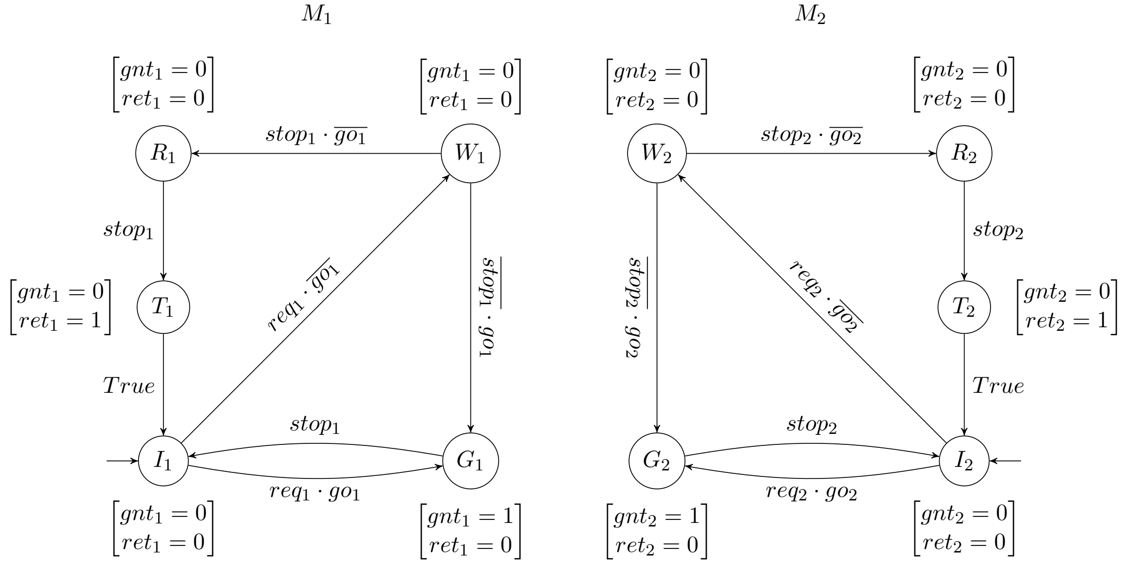


FIGURE A.4: Le modèle de l'arbitre

La propriété souhaitée de l'accès est une *exclusion mutuelle*: machines $M_i, (i = \{1, 2\})$ ne doit jamais émettre leur signaux gnt_i au même moment. Cette exigence peut être exprimée par la proposition:

$$spec_1 : \overline{gnt_1 \cdot gnt_2}.$$

En outre, il est supposé que le mécanisme de rétraction n'a pas encore été implémenté, et il est donc nécessaire que l'état de rétraction T_i sera interdit:

$$spec_2 : \overline{ret_1 + ret_2}.$$

La conjonction de trois exigences sont exprimée par la proposition:

$$spec : \overline{gnt_1 \cdot gnt_2} \cdot \overline{ret_1} \cdot \overline{ret_2}.$$

La proposition $spec$ doit être rendu invariant sur le modèle global de l'arbitre. Cela est spécifié formellement en CTL [37] comme:

$$renforcer : \ AG(spec).$$

Dans cet exemple, les variables d'entrée $\{req_1, req_2, stop_1, stop_2\}$ sont incontrôlables, qui sont affectées par les clients. Les variables d'entrée $\{go_1, go_2\}$ sont contrôlables. Cet exemple simple ne dispose pas de variables de l'interface. La seule synchronisation entre M_1 et M_2 est assurée par la satisfaction de *spec*, comme indiqué ci-dessous.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Les ensembles de propositions Booléennes associées à $M_i, i = 1, 2$ sont $PROP_i = \{gnt_i, ret_i\}$. Les fonctions d'étiquetage λ_i associent les propositions atomiques aux états de M_i:

$$\lambda_i^{gnt_i,ret_i}(I_i) = (0, 0),$$
$$\lambda_i^{gnt_i,ret_i}(G_i) = (1, 0),$$
$$\ldots$$

Inversement, l'ensemble d'états de M_i satisfaisant la proposition $gnt_i \cdot \overline{ret_i}$ est donné par:

$$\begin{aligned}\lambda_i^{-1}(gnt_i \cdot \overline{ret_i}) &= \lambda_i^{-1}(gnt_i) \cap \lambda_i^{-1}(\overline{ret_i}) \\ &= \{G_i\} \cap Q_i \setminus \{T_i\} = \{G_i\}.\end{aligned}$$

Un superviseur doit être construit par la SCD pour mettre en œuvre la politique d'accès *spec* par adéquatement attribuant des valeurs aux variables contrôlables.

A.4.1 Appliquer la SCD globale au modèle de l'arbitre

Avec la synthèse globale directe, nous avons d'abord construit la composition synchrone M_{12} de M_1 et M_2. Pour illustrer la fonction inverse d'étiquetage λ_{12}^{-1}, nous l'appliquons à la proposition Booléenne $spec$ que nous souhaitons à renforcer. Nous obtenons les résultats suivants:

$$\begin{aligned}\lambda_{12}^{-1}(\overline{gnt_1 \cdot gnt_2} \cdot \overline{ret_1} \cdot \overline{ret_2}) &= \\ \lambda_{12}^{-1}(\overline{gnt_1 \cdot gnt_2}) \cap \lambda_{12}^{-1}(\overline{ret_1}) \cap \lambda_{12}^{-1}(\overline{ret_2}).\end{aligned}$$

En appliquant successivement les propriétés citées dans la section 2.2.3 et la définition de λ_{12}, nous obtenons:

$$\begin{aligned}\lambda_{12}^{-1}(\overline{gnt_1 \cdot gnt_2}) &= \\ Q_1 \times Q_2 \setminus \lambda_{12}^{-1}(gnt_1 \cdot gnt_2) &= \\ Q_1 \times Q_2 \setminus (\{G_1\} \times Q_2 \cap Q_1 \times \{G_2\}) &= \\ Q_1 \times Q_2 \setminus \{G_1\} \times \{G_2\}.\end{aligned}$$

Une application similaire des même calculs conduit à un résultat global:

$$\begin{aligned}\lambda_{12}^{-1}(\overline{gnt_1 \cdot gnt_2} \cdot \overline{ret_1} \cdot \overline{ret_2}) &= \\ Q_1 \times Q_2 \setminus (\{G_1\} \times \{G_2\} \cup \{T_1\} \times Q_2 \cup Q_1 \times \{T_2\}).\end{aligned}$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

L'algorithme de SCD commence avec l'ensemble d'états satisfaisant *spec* : $\mathcal{I}^0 = \lambda_{12}^{-1}(spec)$. L'ensemble d'invariant sous contrôle $\mathcal{IUC}$ par rapport à *spec* contient tous les états de $M_1||M_2$ tel que pour tout tuple incontrôlable $(req_1, req_2, stop_1, stop_2) \in \mathbb{B}^4$ il existe toujours $(go_1, go_2) \in \mathbb{B}^2$ tel que la transition obtenue conduit à $\mathcal{IUC}$. Ainsi, nous obtenons un ensemble d'invariant sous contrôle qui enlève les états R_1, T_1, R_2, T_2 ainsi que les état global $\{G_1\} \times \{G_2\}$, comme indiqué dans Figure A.5.

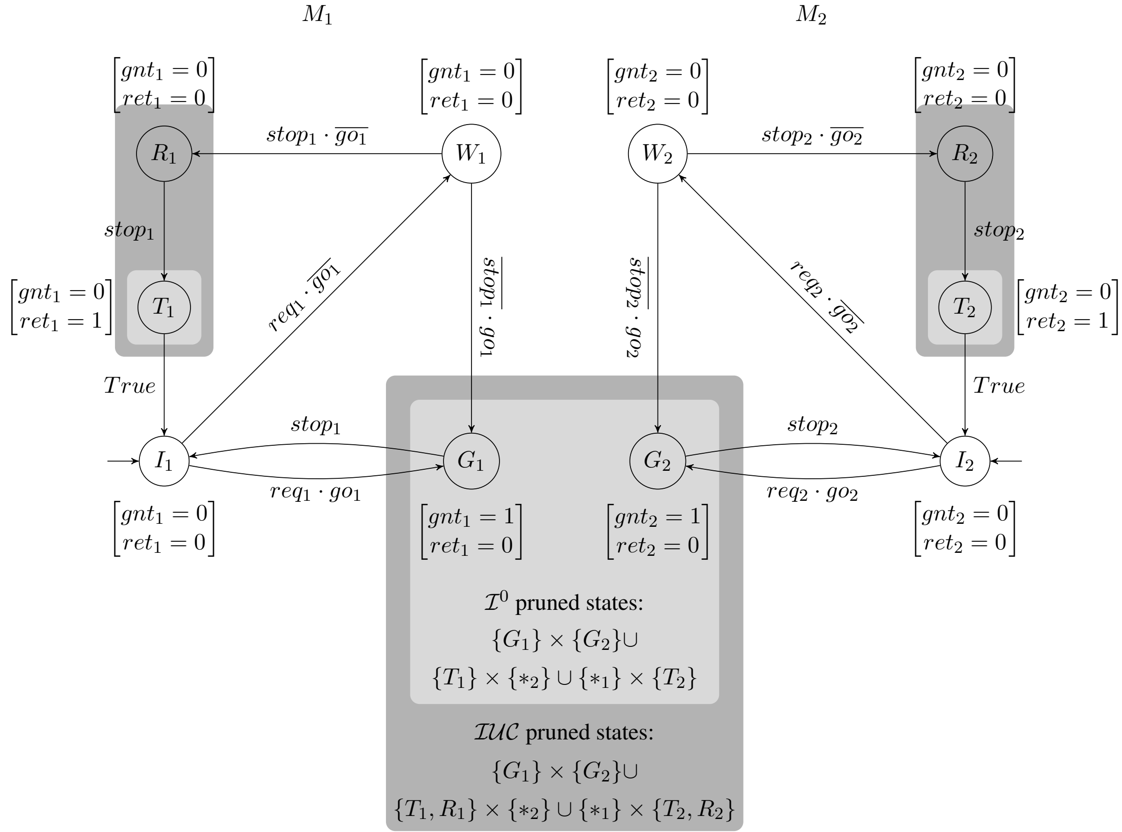


FIGURE A.5: SCD globale pour $M_1||M_2$ et *spec*

Le superviseur correspondant est représente par:

$$
\begin{aligned}
(I_1, I_2) &\Rightarrow (\overline{req_1 \cdot go_1} + \overline{req_2 + go_2})+ \\
(I_1, W_2) &\Rightarrow (\overline{req_1 \cdot go_1} \cdot \overline{stop_2} + stop_2 \cdot go_2 + \overline{stop_2} \cdot \overline{go_2})+ \\
(I_1, G_2) &\Rightarrow (\overline{req_1 \cdot go_1} + stop_2)+ \\
(W_1, I_2) &\Rightarrow (\overline{go_1} + \overline{req_2 \cdot go_2})+ \\
(W_1, W_2) &\Rightarrow (\overline{go_1} + \overline{go_2})+ \\
(W_1, G_2) &\Rightarrow (\overline{go_1} + stop_2)+ \\
(G_1, I_2) &\Rightarrow (stop_1 + \overline{req_2 \cdot go_2})+ \\
(G_1, W_2) &\Rightarrow (stop_1 + \overline{go_2}).
\end{aligned}
$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Il représente toutes les transitions de $M_1||M_2$ conduisant à $\mathcal{IUC}$. Par exemple, dans l'état (I_1, I_2), si req_1 et req_2 sont à la fois activées, le superviseur permet soit $go_1 = 1$ ou $go_2 = 1$ mais pas les deux à la fois.

A.4.2 Appliquer la SICD au modèle de l'arbitre

Dans la suite, nous notons $\{*_i\}$ pour l'ensemble d'états du modèle M_i.

A.4.2.1 Abstraction

Nous abstrayons M_2 en le remplaçant par M_2^{abs}. L'abstraction appliquée à M_2 concerne ses variables partagées avec M_1 et $spec$. Nous avons $PROP_2^{out} = \emptyset$ et $SPEC_2 = \{gnt_2, ret_2\}$. Figure A.6 représente le modèle abstrait obtenu. Pour la raison de lisibilité, les états abstraits sont directement étiquetés par leur propositions booléennes correspondantes. Notons que le modèle abstrait M_2^{abs} est entièrement non-déterministe: à tout moment, toutes les transitions sont activées.

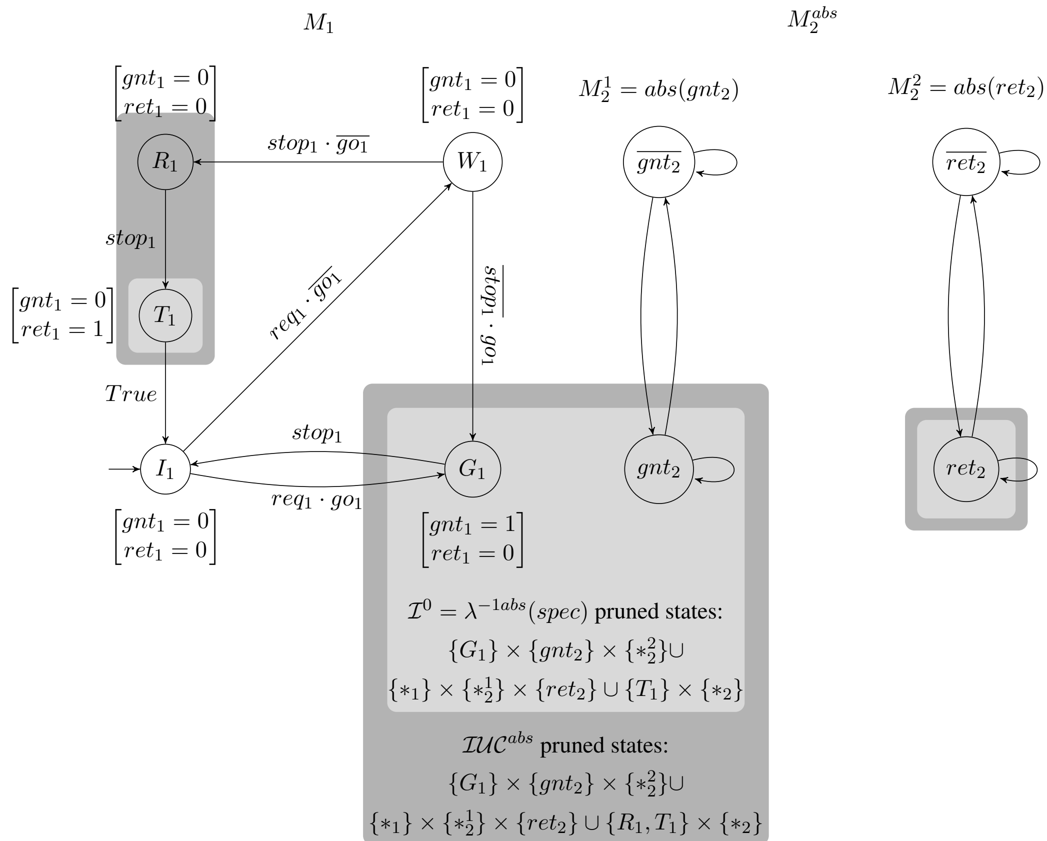


FIGURE A.6: Modèle abstrait $M_1||M_2^{abs}$ et le calcul du $\mathcal{IUC}^{abs}$ approximatif

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

A.4.2.2 Le calcul du $\mathcal{IUC}$ approximatif

L'ensemble approximatif d'invariant sous contrôle pour la propriété *spec* est calculé sur le modèle abstrait $M_1||M_2^{abs}$. Comme illustré dans la Figure A.6, $\mathcal{IUC}^{abs}$ enlève les états suivants : $\{G_1\} \times \{gnt_2\}$, R_1, T_1 et l'état abstrait ret_2.

A.4.2.3 Raffinement et la synthèse finale

Nous remplaçons M_2^{abs} par M_2 et affinons les états enlevés depuis $\mathcal{IUC}^{abs}$ aux états de $M_1||M_2$ par rapport à $spec$. Cela se fait en appliquant $Ref(\mathcal{IUC}^{abs}, spec, Q^{M_1||M_2})$. Ainsi, l'état de M_2 satisfaisant $gnt_2 = 1 \wedge ret_2 = 0$ est G_2, et l'état satisfaisant $ret_2 = 1$ et $\{T_2\}$. L'ensemble raffiné obtenu est représenté dans la Figure A.7.

Le calcul du superviseur final tente de faire invariant l'ensemble $Ref(\mathcal{IUC}^{abs}, spec, Q^{M_1||M_2})$. Cette dernière étape enlève l'état R_2 depuis la solution finale, comme le montre dans la Figure A.7. Ce résultat final est exactement le même que celui obtenu par la SCD directe.

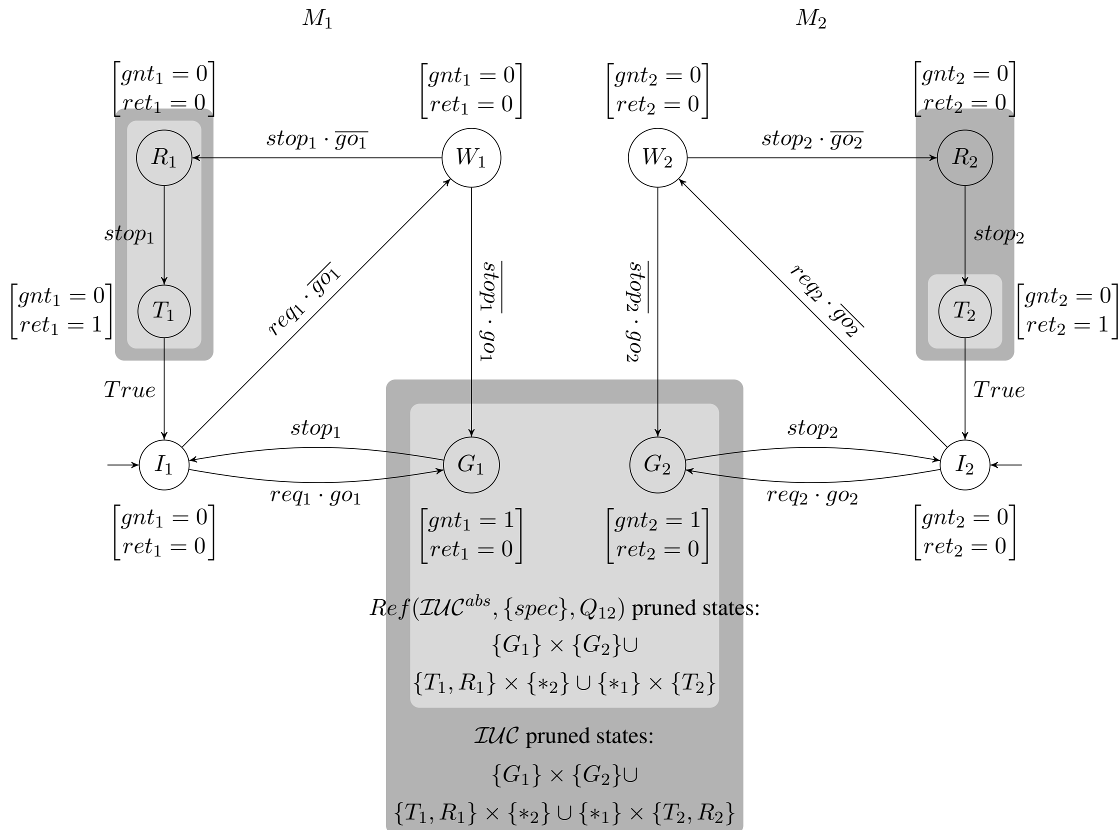


FIGURE A.7: La SCD finale assurant *spec*

Cet exemple illustre les principes de base de la technique de synthèse incrémentale. Toutefois, les deux composantes ci-dessus n'ont pas de communication explicite entre eux. Leur synchronisation est renforcée par la satisfaction de la spécification globale $spec$.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

A.5 Deuxième exemple: un arbitre avec jeton

C'est un système un peu plus complexe avec la communication entre ses composantes. Il met en œuvre un arbitre qui gère l'accès exclusif à une ressource partagée par trois clients indépendants. Le modèle présenté ci-dessous se concentre uniquement sur la fonctionnalité de la gestion de l'accès. La ressource partagée réelle ainsi que son interaction avec chaque client sont laissées non modélisées.

La gestion de l'accès est modélisée par trois cellules concurrentes, $M_i, (i = \{1, 2, 3\})$. Ils partagent une fonction et structure identique. Chaque cellule i reçoit une demande req_i de son client et émet un accord correspondant d'accès ack_i à ce client. Le contrôle d'accès est appliqué par un mécanisme de jeton. Une cellule peut servir son client que si elle maintient le jeton.

Cellules M_i sont modélisées par deux automates: M_i^1 reçoit le jeton. M_i^2 implémente l'issu de l'accès à la ressource partagée selon la disponibilité du jeton. Il transmet également le jeton, une fois qu'il a été effectivement utilisé. Les automates d'une seule cellule sont indiqués dans la Figure A.8. M_i^1 et M_i^2 communiquent via le signal go_i, qui est affecté à 1 chaque fois qu'un jeton a été reçu et 0 autrement. L'état N_i signifie "pas de jeton reçu". L'état T_i signifie "un jeton a été reçu". L'état I_i est un état de "idle", en attente d'une demande du client. L'état G_i est un état actif, où un client vient d'obtenir l'accès et le jeton est transmis.

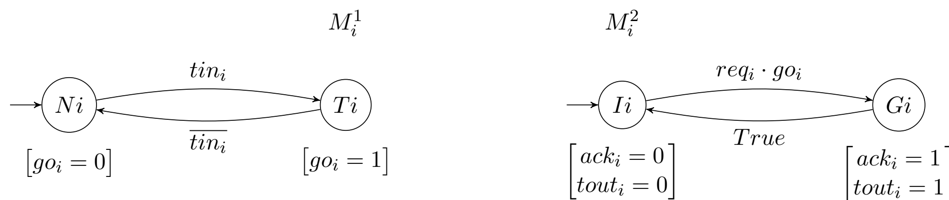


FIGURE A.8: Modèle d'une seule cellule i

L'arbitre est construit par l'instanciation des trois cellules $M_i, i = 1, 2, 3$ et en connectant leur interfaces et les variables de sortie, comme le montre la Figure A.9. Les jetons sont insérés à M par l'environnement via l'entrée tin_1.

Selon la définition de la composition synchrone des CFSMs communicants, modules M_i disposent des entrées de l'environnement et des entrées locales. Les entrées de l'environnement sont dédiées à la communication avec composants situés en dehors de M. Les entrées locales sont utilisées pour modéliser la communication entre les modules M_i de M. Ici, nous avons $L_1 = \emptyset$, $L_2 = \{tin_2\}$, $L_3 = \{tin_3\}$.

La politique souhaitée de l'accès est une *exclusion mutuelle*: machines M_i ne doivent jamais émettre leur signal ack_i au même moment. Cette exigence peut être exprimée par le prédicat suivant:

$$spec : \overline{ack_1 \cdot ack_2 + ack_2 \cdot ack_3 + ack_1 \cdot ack_3}$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

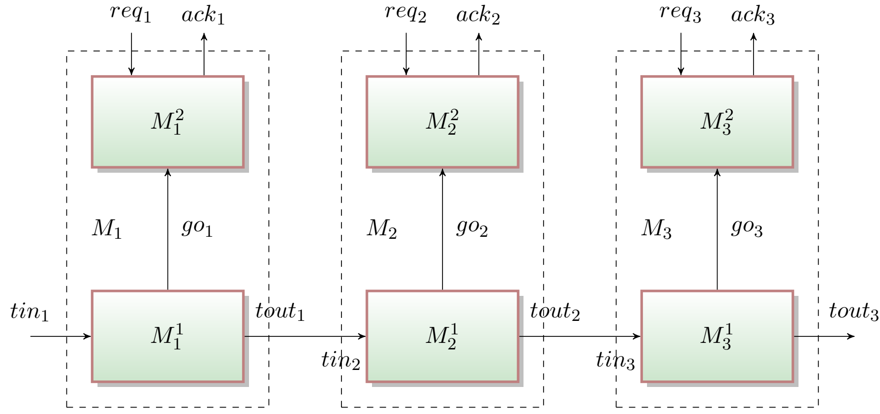


FIGURE A.9: Connexion interne de l'arbitre avec jeton

A.5.1 Définition du problème de la SCD

Modules M_i sont des blocs de construction pour la conception d'un arbitre correct. Ils sont assemblés selon la Figure A.9. Dans le modèle obtenu, le mécanisme de jeton n'est que partiellement défini. Un jeton peut être inséré par l'environnement, via l'entrée tin_1 qui est affectée à 1. Une fois inséré, le jeton est passé d'une cellule au suivant dès qu'il est utilisé. Toutefois, afin de satisfaire $spec$, il ne doit jamais avoir plus d'un jeton présent à l'intérieur de l'arbitre. Par conséquent, la valeur de tin_1 ne peut pas être choisi au hasard. Il ne devrait pas être mis à 1 jusqu'à aucun jeton est présent à l'intérieur de M.

Nous essayons de rendre invariant la proposition $spec = 1$ par l'application de la SICD. Comme tin_1 semble essentielle pour garantir l'unicité de jeton à l'intérieur de M, nous l'avons choisi comme *contrôlable*. Les demandes entrantes $req_i, i = 1, 2, 3$ peuvent arriver à tout moment, et elles sont considérées comme *incontrôlables*.

Ainsi, le superviseur doit contrôler M via son entrée de jeton tin_1, en fonction des entrées incontrôlables $\{req_1, req_2, req_3\}$ et l'arbitre contrôlée devrait satisfaire la propriété CTL [37]:

$$renforcer: \ AG(spec).$$

A.5.2 Appliquer la SICD à l'arbitre

La première étape consiste de calculer $\lambda_{123}^{-1}(spec)$. Nous obtenons les résultats suivants:

$$\lambda_{123}^{-1}(\overline{ack_1 \cdot ack_2 + ack_2 \cdot ack_3 + ack_1 \cdot ack_3}) =$$
$$\lambda_{123}^{-1}(\overline{ack_1 \cdot ack_2} \cdot \overline{ack_2 \cdot ack_3} \cdot \overline{ack_1 \cdot ack_3}) =$$
$$\lambda_{123}^{-1}(\overline{ack_1 \cdot ack_2}) \cap \lambda_{123}^{-1}(\overline{ack_2 \cdot ack_3}) \cap \lambda_{123}^{-1}(\overline{ack_1 \cdot ack_3}).$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

En appliquant successivement les propriétés citées dans section 2.2.3 et la définition de λ_{123}^{-1} nous obtenons:

$$\begin{aligned}\lambda_{123}^{-1}(\overline{ack_1 \cdot ack_2}) =\\ Q_1 \times Q_2 \times Q_3 \setminus \lambda_{123}^{-1}(ack_1 \cdot ack_2) =\\ Q_1 \times Q_2 \times Q_3 \setminus (\{*_1^1\} \times \{G_1\} \times Q_2 \times Q_3 \cap Q_1 \times \{*_2^1\} \times \{G_2\} \times Q_3) =\\ Q_1 \times Q_2 \times Q_3 \setminus (Q_1^1 \times \{G_1\} \times Q_2 \times Q_3 \cap Q_1 \times Q_2^1 \times \{G_2\} \times Q_3).\end{aligned}$$

Une application similaire des même calculs produit le résultat global:

$$\begin{aligned}\lambda_{123}^{-1}(\overline{ack_1 \cdot ack_2 + ack_2 \cdot ack_3 + ack_1 \cdot ack_3}) =\\ Q_1 \times Q_2 \times Q_3 \setminus (Q_1^1 \times \{G_1\} \times Q_2 \times Q_3 \cap Q_1 \times Q_2^1 \times \{G_2\} \times Q_3\\ \cup Q_1 \times Q_2^1 \times \{G_2\} \times Q_3 \cap Q_1 \times Q_2 \times Q_3^1 \times \{G_3\}\\ \cup Q_1^1 \times \{G_1\} \times Q_2 \times Q_3 \cap Q_1 \times Q_2 \times Q_3^1 \times \{G_3\}).\end{aligned}$$

A.5.2.1 Abstraction

Il existe plusieurs possibilités d'abstraction-raffinement d'après l'ordre défini par l'utilisateur. Une possibilité est de commencer par le modèle abstrait $M_1||M_2^{abs}||M_3^{abs}$ et incrémentalement remplacer M_2^{abs} avec M_2 et M_3^{abs} avec M_3. Ici, nous commençons par le modèle abstrait $(M_1||M_2)||M_3^{abs}$, donnée par $\mathcal{ABS}(M, 3, spec)$, comme le montre la Figure A.10.

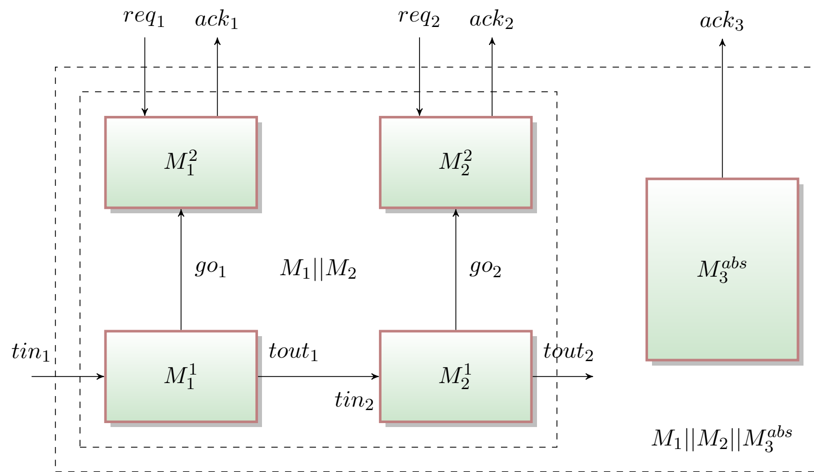


FIGURE A.10: Ordre d'abstraction de l'arbitre

L'opération d'abstraction appliquée à M_3 concerne ses variables de sortie partagées avec $M_1||M_2$ et $spec$. Nous avons $PROP_3^{out} = \{ack_3\}$ et $SPEC_3 = \{ack_3\}$. Notons que ack_3 et $tout_3$ ont toujours des valeurs identiques. Figure A.11 représente le modèle abstrait obtenu. Pour la raison de lisibilité, états abstraits sont directement marquées avec leurs propositions booléennes correspondantes.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

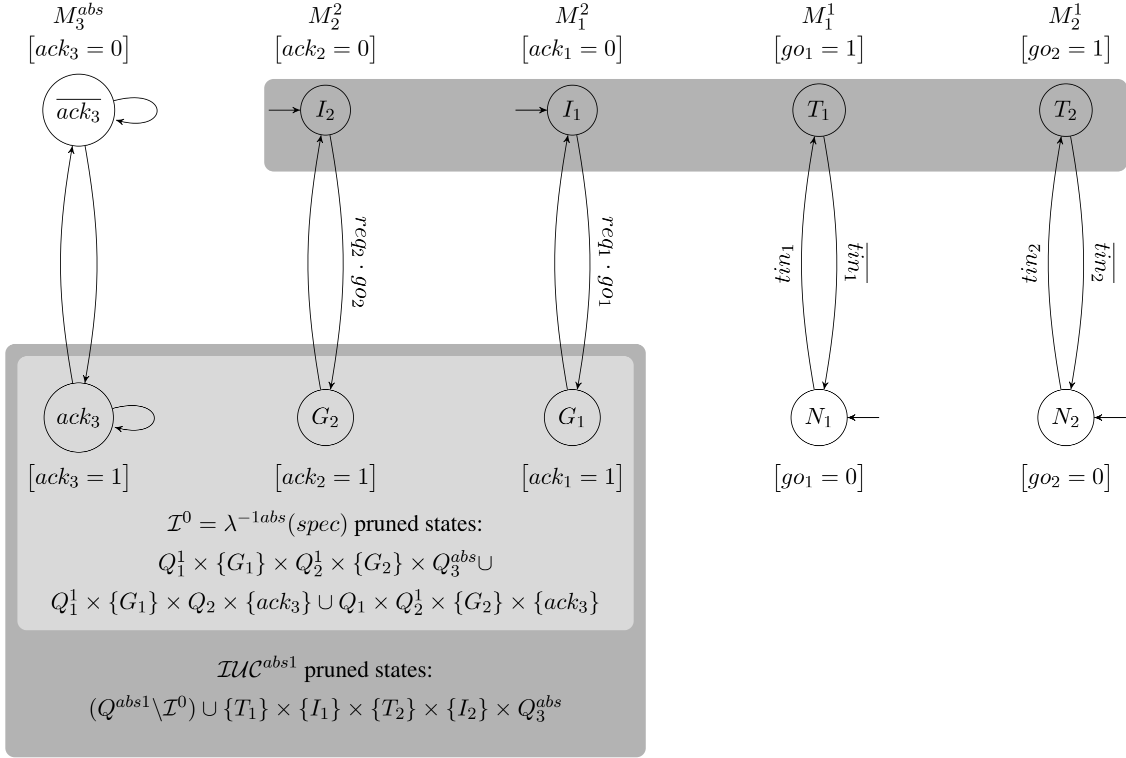


FIGURE A.11: Modèle abstrait $M_1||M_2||M_3^{abs}$ et le calcul de $\mathcal{IUC}$ approximatif

A.5.2.2 Calcul du $\mathcal{IUC}$ approximatif.

L'ensemble approximatif d'invariant sous contrôle pour la propriété *spec* est construit sur le modèle abstrait $M^{abs1} = M_1||M_2||M_3^{abs}$. Comme illustré dans la Figure A.11. Le calcul d'abord identifie l'ensemble initial d'états à enlever $\lambda^{-1abs1}(spec)$. La première itération conduit à $\mathcal{IUC}^{abs1}$ qui enlève les états suivants depuis l'ensemble $Q_1 \times Q_2 \times Q_3^{abs}$:

$$\mathcal{IUC}^{abs1} = Q_1 \times Q_2 \times Q_3^{abs} \backslash \\ Q_1^1 \times \{G_1\} \times Q_2^1 \times \{G_2\} \times Q_3^{abs} \cup Q_1^1 \times \{G_1\} \times Q_2 \times \{ack_3\} \\ \cup Q_1 \times Q_2^1 \times \{G_2\} \times \{ack_3\} \cup \{T_1\} \times \{I_1\} \times \{T_2\} \times \{I_2\} \times Q_3^{abs}.$$

A.5.2.3 Raffinement

Le modèle abstrait M^{abs1} est raffiné en remplaçant M_3^{abs} avec M_3, et en projetant $\mathcal{IUC}^{abs1}$ aux états $Q = Q_1 \times Q_2 \times Q_3$ comme le montre dans la Figure A.12.

L'invariant approximatif sous contrôle $\mathcal{IUC}^{abs1}$ calculé dans l'étape précédente est alors projeté en arrière sur ce modèle:

$$\mathcal{I}^0 = Ref(\mathcal{IUC}^{abs1}, \{spec\}, Q)$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Par exemple, l'ensemble d'états abstraits $Q_1^1 \times \{G_1\} \times Q_2 \times \{ack_3\}$ est raffiné à $Q_1^1 \times \{G_1\} \times Q_2 \times Q_3^1 \times \{G_3\}$ comme illustré dans la Figure A.12.

A.5.2.4 Synthèse finale et résultats finaux

Le résultat final de la SICD est obtenu en appliquant une dernière étape de la SCD sur le modèle global, en utilisant comme point de départ le résultat obtenu au cours du processus itératif. Ici, nous calculons l'invariant final sous contrôle $\mathcal{IUC}$, à partir de $\mathcal{I}^0$, comme le montre dans la Figure A.12.

Après cette étape, nous obtenons:

$$\begin{aligned}\mathcal{IUC} = Q_1 \times Q_2 \times Q_3 \backslash \\ \{T_1\} \times \{I_1\} \times Q_2 \times \{T_3\} \times \{I_3\} \cup Q_1 \times \{T_2\} \times \{I_2\} \times \{T_3\} \times \{I_3\} \cup \\ \{\{T_1\} \times \{G_1\} \times \{T_2\} \times \{I_2\} \times \{N_3\} \times \{I_3\}, \{T_1\} \times \{I_1\} \times \{T_2\} \times \{G_2\} \times \{N_3\} \times \{I_3\}\}.\end{aligned}$$

La construction du superviseur final depuis $\mathcal{IUC}$ est simple. En simulant le système globale contrôlé, il peut constater que le superviseur contrôle l'entrée tin_1 comme suit. Lorsque la cellule 3 a servi une demande entrante ($ack_3 = 1$), le jeton est transmis par la sortie $tout_3$, et est donc perdu. Pas plus de jetons sont présents à l'intérieur de M. En ce moment, le superviseur permet d'insérer un nouveau jeton par l'environnement à tout moment. Cela se fait en attribuant $tin_1 = 1$. Une fois qu'un jeton a été inséré, il reste à l'intérieur M, passant de M_1 à M_2 et puis à M_3, tant que l'accusé prochain ack_3 est fait. Pendant ce temps, le superviseur désactive l'insertion d'un nouveau jeton. En effet, si un nouveau jeton est inséré, alors plus d'une cellule peuvent être titulaires d'un jeton à un moment donné, et donc, deux ou plusieurs clients peuvent être servis en même temps.

Cependant, M ne peut pas être directement composé avec son superviseur. Parce que les superviseurs générés par la SCD symbolique sont fonctions caractéristiques (i.e. équations), qui ne peuvent pas être manipulées en tant que composants, ayant des entrées et des sorties. Ainsi, chaque superviseur doit être *implémenté* avant de pouvoir être "branché" dans le système à contrôler. Ce problème est développé dans le chapitre suivant. L'architecture matérielle finale de l'arbitre contrôlé est affiché et commenté dans la Figure 4.8.

A.6 Mise en oeuvre et la performance

La performance de l'algorithme SICD dépend fortement de la réelle mise en œuvre de la technique SCD. Les possibilités techniques disponibles sont la *SCD énumérative* et la *SCD symbolique basée sur BDD*.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

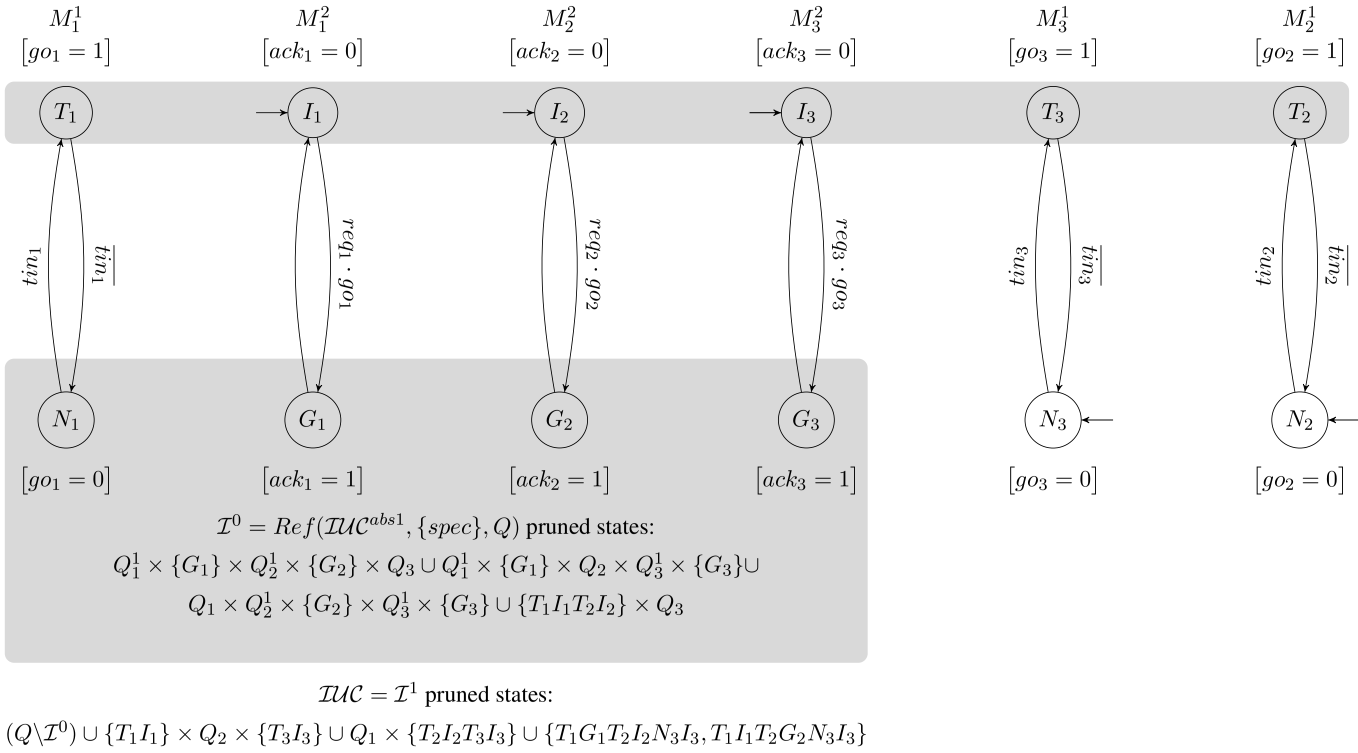


FIGURE A.12: Raffinement de $M_1||M_2||M_3^{abs}$ et le calcul du $\mathcal{IUC}$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

La technique énumérative de la SCD représente explicitement l'ensemble des états du système. La complexité de la SCD est $O(n|\Sigma|)$ où n est le nombre total d'états du système composé, et $|\Sigma|$ représente la taille de l'alphabet d'entrée.

La SCD symbolique basé sur BDD manipule des ensembles d'états, plutôt que des états, et utilise des diagrammes de décision binaire (BDD) [5] pour les représenter. La performance de cette technique est prometteuse, mais reste lié par la complexité spatiale pour construire une BDD, qui est exponentielle du nombre des variables booléenne représentant le système. Ainsi, la mémoire est une ressource critique pour la SCD symbolique.

Indépendamment de la technique de SCD utilisée, le calcul d'un $\mathcal{IUC}$ abstrait est certainement plus rapide, car il fonctionne sur une modèle réduit. L'accélération dépend principalement de la décomposition structurelle atteinte et donc sur la taille finale du modèle abstrait obtenu. Toutefois, l'application finale de la SCD globale à partie d'une solution intermédiaire produite incrémentalement offre toujours la même complexité théorique que la SCD classique, ainsi que les frais engendrés par le calcul de la solution abstraite.

Toutefois, la capacité de la SCD symbolique pour manipuler efficacement ensembles d'états au lieu des états individuels est un avantage important. Par conséquent, nous avons choisi de construire l'algorithme SICD en haut de la SCD symbolique développée dans [38], qui est la seule plate-forme de SCD symbolique appropriée et disponible. Malgré les limites théoriques rappelés ci-dessus, nous nous attendons à une amélioration *moyenne* de la performance de SICD symbolique pour les raisons suivantes.

Tout d'abord, le calcul du $\mathcal{IUC}$ abstrait fonctionne sur le modèle réduit $\mathcal{ABS}(M, j, spec)$. Comme la plupart des états sont enlevés par l'abstraction de M, l'impact sur la taille de la BDD construite pour le parcours symbolique de l'espace d'état de $\mathcal{ABS}(M, j, spec)$ est indubitable.

Deuxièmement, le calcul de $\mathcal{I}$ produit une solution *intermédiaire, approximative* du problème SCD. Il s'appuie sur une représentation plus compacte, car il est construit sur un système abstrait contenant moins de variables. En outre, l'ensemble $\mathcal{I}$ est un sous-ensemble de $\lambda^{-1}(spec)$. Ainsi, un certain nombre d'états de $\mathcal{ABS}(M, j, spec)$ sont taillés à un moindre coût de calcul et ne doivent pas être réexaminée au cours des applications ultérieures de SCD. Ainsi, nous nous attendons à la dernière étape de SCD converge plus rapidement (avec des moindre itérations de point fixe) vers la solution finale. D'ailleurs, si un problème SCD n'a pas de solution, cela peut être détectés sur le système abstrait à un coût beaucoup plus faible.

Cependant, ces facteurs de performance ne peuvent être théoriquement généralisés. On peut s'attendre à de mauvais résultats pour certains problèmes pour lesquels il n'y a pas de représentation réelle efficace de BDD. En générale, les systèmes comportant des opérations arithmétiques tels que les multiplications au niveau du bit sont connus pour avoir représentations de BDD insolubles. En outre, l'exécution de la SCD symbolique exige des réglages supplémentaires liés aux manipulations de BDD. La SCD globale et incrémentale peuvent profiter considérablement d'un bon choix initial de l'ordre des variables. En

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

effet, certaines expressions booléennes peuvent avoir une représentation de BDD très compacte, ou un énorme, en fonction de l'ordre des variables choisi pour construire leur BDD. En général, trouver un tel bon ordre heuristique est fait, le plus souvent en exploitant la connaissance des propriétés structurelles du système. Le problème d'optimisation de trouver la meilleure ordre des variables est NP-complet. Deuxièmement, la performance de la manipulation de BDD peut être améliorée par l'application de *réordonnance* dynamique des variables. C'est un outil fondamental mise en œuvre par la plupart des paquets BDD. Au moment de l'exécution, il réalise permutations locales entre les variables liées, qui donne également des améliorations *moyenne* de la mémoire, si ce calcul est intensive. Encore une fois, l'utilisation de cette technique pourrait offrir une amélioration, mais peut aussi conduire à des résultats de mauvaise performance. Application de cette technique demande une expertise mature du système à contrôler.

En conclusion, la SICD peut être utilisée comme un outil de tirer parti de la complexité de la SCD, en combinaison avec un ordonnance statique et dynamique des variables. En outre, cette technique peut être facilement automatisée.

Les chiffres de performances de la SICD symbolique sur la SCD symbolique sont présentés dans la section 3.7.

A.7 Introduction à l'implémentation du superviseur

Le problème de mise en œuvre auquel nous nous intéressons est spécifique au contexte de la Synthèse de Contrôleurs Discrets (SCD) [6, 38, 65]. Plus précisément, on s'intéresse à la variante symbolique de cette technique [38, 65], car les performances du parcours symbolique utilisant des Diagrammes de Décision Binaire (BDD) sont à ce jour incontestables.

Les avantages de la SCD ont été peu exploités dans le contexte de la conception de systèmes matériels. Les concepteurs utilisent pourtant des techniques de vérification formelle telles que le "model checking", très apparentées à la SCD. Notre objectif est d'obtenir un contrôleur, sous forme d'un réseau de portes logiques, permettant une réalisation physique immédiate grâce à des technologies cibles telles que les FPGAs.

La mise en œuvre d'un superviseur, représenté par son modèle logique (graphe d'états ou équation caractéristique), est une transformation, avec prise en compte de contraintes de la plateforme physique ciblée. Ainsi, si la cible est une plateforme logicielle, le superviseur doit être un programme exécutable. Si en revanche on cible une plateforme matérielle, ce même superviseur doit être mis sous forme d'un réseau de portes. Selon l'application visée, d'autres contraintes de mise en œuvre peuvent encore s'ajouter : rapidité, taille, déterminisme, tolérance aux fautes, distribution, etc. Cependant, quelle que soit la cible visée, il y a deux problèmes nécessitant un traitement en amont.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

D'une part, le *non-déterminisme de contrôle*. La plupart des superviseurs synthétisés sont non-déterministes : si à un instant donné plusieurs solutions de contrôle sont possibles, aucun choix n'est fait a priori. Pour effectuer ce choix il est nécessaire de définir une méthode pour lever le non-déterminisme, ou à défaut l'intervention de l'opérateur humain. La levée du non-déterminisme de contrôle revient généralement à restreindre les solutions de contrôle à un sous-ensemble strict. Le résultat déterministe obtenu est communément désigné sous le nom de *contrôleur*.

D'autre part, il y a *l'incompatibilité structurelle* entre le superviseur et le procédé à contrôler. Ce problème ne concerne que la SCD symbolique. En effet, les superviseurs symboliques n'ont pas d'interface d'entrée/sortie. Ils sont générés sous la forme d'une équation Booléenne caractéristique, encodant l'ensemble des solutions de contrôle calculées. Une approche directe d'implémentation serait de résoudre continuellement l'équation du superviseur durant l'exécution du système contrôlé. Ceci requiert un solveur d'équation Booléennes, exécuté physiquement sur un microprocesseur (dédié ou pas). L'architecture cible du superviseur implémenté serait ainsi une architecture logicielle, ce qui pose des limitations en termes d'efficacité pendant l'exécution.

Contributions

Pour cela, nous proposons de résoudre l'indéterminisme de contrôle et trouver une solution adéquate au problème de l'incompatibilité structurelle. La résolution de l'indéterminisme de contrôle garantit une implémentation autonome vis-à-vis du choix des solutions de contrôle. La résolution de l'incompatibilité structurelle revient à trouver une représentation du superviseur sous forme d'un ensemble de fonctions de contrôle. Cette solution présente un double avantage. Premièrement et le plus important, une implémentation matérielle peut être obtenue, sous forme de circuit électronique. Ceci est intéressant lorsque les performances d'exécution du système contrôlé ont de l'importance. Deuxièmement, l'implémentation d'un superviseur par un ensemble de fonctions de contrôle semble beaucoup plus compacte que le superviseur initial.

Par ailleurs, nous tentons d'adapter l'architecture en "boucle de contrôle" classique au contexte de la conception matérielle. Nous identifions une catégorie de problèmes de conception matérielle pour laquelle notre architecture de contrôle est appropriée.

Etat de l'art

L'implémentation de superviseurs a déjà été examinée avec une attention particulière accordée à l'exécution de contrôleurs sur un automate programmable industriel [43–46].

Dans [47], l'approche proposée a pour objectif de minimiser la latence d'un protocole. Elle résout le non-déterminisme de manière à ce qu'un ensemble d'états cible soit atteint le plus rapidement possible. Toutefois, une fois ce critère de vitesse satisfait, tout non-déterminisme restant est levé en faisant des choix arbitraires. Cette contribution a été généralisée et améliorée en [48, 49].

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

La plupart des travaux de recherche à notre connaissance ont abordé le problème de l'implémentation de superviseurs à travers une démarche de raffinements successifs de l'objectif de contrôle aboutissant sur un superviseur déterministe (i.e. un contrôleur). D'autres approches préconisent la mise en place d'un choix aléatoire en cas de non-déterminisme. Les mécanismes de priorités fixe ou dynamique ont également été utilisés.

Notre technique permet d'obtenir un contrôleur déterministe, à travers une étape de décomposition structurelle appliquée au superviseur, décomposition qui conserve l'ensemble des solutions de contrôle obtenues par synthèse. Des approches similaires ont été proposées et utilisées dans divers contextes, pas toujours en relation avec la synthèse de contrôleurs. Dans [66], une technique de décomposition paramétrique est appliquée sur des prédicats Booléens dans le contexte de la vérification formelle de systèmes matériels. Plus proche de notre travail, dans [51] on présente une technique de mise en oeuvre de contrôleurs obtenus par synthèse optimale ; la technique est également symbolique, mais engendre seulement un sous-ensemble strict du superviseur. Dans [52] les auteurs étudient la triangulation d'une équation polynômiale, avec le même objectif d'atteindre une implémentation de superviseurs, en utilisant une représentation en logique ternaire. Le même objectif d'implémentation est examiné dans [63], où la technique de synthèse de contrôleurs opère directement depuis un ensemble de spécifications formelles (contrairement à la SCD, qui a besoin d'un modèle de procédé). Une autre approche d'implémentation matérielle de superviseurs, basée sur des réseaux de Pétri, est présentée dans [67].

L'organisation de ce papier est la suivante : La section II présente les définitions de concepts basiques et des notations. Notre méthode d'implémentation de superviseurs est présentée dans la section III. La section IV illustre l'application de notre méthode sur un système matériel. Enfin, la section V présente l'ensemble des outils que nous avons utilisés ou développés.

A.8 Définitions

A.8.1 Machines d'états finis contrôlables

Soit $\mathbb{B} = \{0, 1\}$ l'ensemble des valeurs Booléennes. Soit P une machine à états finis définie par le n-uplet :

$$P = \langle I, S, \delta, s_0, O, \lambda \rangle$$

tel que :

- I est l'ensemble des variables Booléennes d'entrée, tel que $I = U \cup C$, et $U \cap C = \emptyset$;
- $U = \{u_0, \cdots, u_{r-1}\}, r > 0$ est l'ensemble des variables incontrôlables d'entrée. Nous notons $\mathbf{u} = (u_0, \cdots, u_{r-1})$ un tuple des éléments de U;

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- $C = \{c_0, \cdots, c_{p-1}\}, p > 0$ est l'ensemble des variables contrôlables d'entrée. Nous notons $\mathbf{c} = (c_0, \cdots, c_{p-1})$ un tuple des éléments de C;
- $S = \{s_0, \cdots, s_{n-1}\}, n > 0$ est l'ensemble des variables Booléennes d'états. Nous notons $\mathbf{s} = (s_0, \cdots, s_{n-1}), n > 0$ un tuple des éléments de S;
- $\delta : \mathbb{B}^{p+r} \times \mathbb{B}^n \to \mathbb{B}^n$ est la fonction de transition de P;
- $s_0 \in \mathbb{B}^n$ est l'état initial de P;
- $O = \{o_0, \cdots, o_{m-1}\}, m > 0$ est l'ensemble des variables Booléennes de sortie;
- $\lambda : \mathbb{B}^{p+r} \times \mathbb{B}^n \to \mathbb{B}^m$ est la fonction de sortie associée à O.

Dans la suite, nous appelons P le *procédé* i.e. le système à contrôler par SCD. Les ensembles d'états de P sont manipulés par l'intermédiaire de leur fonction caractéristique. Soit $E \subset \mathbb{B}^n$. La fonction caractéristique de E est définie par:

$$\mathcal{C}_E : \mathbb{B}^n \to \mathbb{B} \text{ et } \mathcal{C}_E(e) = 1 \Leftrightarrow e \in E \tag{A.1}$$

Les opérations ensemblistes ordinaires ont des opérateurs Booléens correspondants: $''+''$ ("ou" logique) réalise l'union d'ensembles et $''.''$ ("et" logique) réalise l'intersection d'ensembles. La négation logique $\overline{\mathcal{C}_E}$ exprime le complément de E par rapport à $\mathbb{B}^n$.

Soit $S' = \{s'_0, \cdots, s'_{n-1}\}$ un ensemble de variables dites de *"prochain état"*. La relation de transition $\mathcal{T}$ de P est définie par l'ensemble de toutes les transitions possibles $\mathbf{s} \xrightarrow{\mathbf{u},\mathbf{c}} \mathbf{s}'$ tel que $\mathbf{s}' = \delta(\mathbf{s}, \mathbf{u}, \mathbf{c})$:

$$\mathcal{T}(\mathbf{s}, \mathbf{u}, \mathbf{c}, \mathbf{s}') = \prod_{i=0}^{n-1} s'_i \Leftrightarrow \delta_i(\mathbf{u}, \mathbf{c}, \mathbf{s}) \tag{A.2}$$

où l'opérateur "produit" réalise un "et" logique sur n opérandes.

A.8.2 Synthèse de contrôleurs discrets

La SCD a été développée dans [6], utilisant la théorie des langages. Parallèlement, des travaux de recherche en conception de circuits ont démontré le grand intérêt de la représentation symbolique d'ensembles d'états utilisant les BDDs[5], pour traiter le problème de l'explosion combinatoire de l'espace d'états [16]. Des développements de la SCD basés sur des techniques symboliques ont été proposés dans [38, 65, 68].

L'approche symbolique de SCD manipule des ensembles d'états et/ou transitions, au lieu des langages. Pour un procédé donné P et une spécification désirée Q, nommée un *objectif de contrôle*, la SCD symbolique calcule un superviseur $\mathcal{SUP}$ garantissant que P satisfait toujours Q. L'architecture de contrôle est illustrée dans la figure A.13 et A.14. Le calcul procède en deux étapes:

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

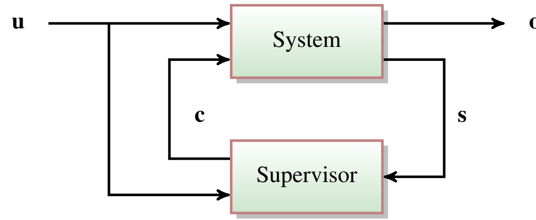


FIGURE A.13: Boucle de contrôle classique

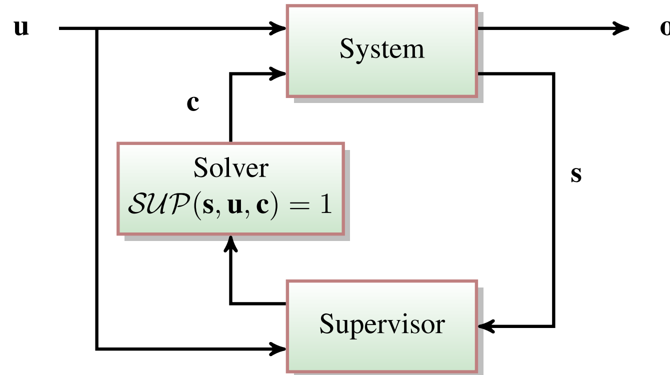


FIGURE A.14: Boucle de contrôle avec un superviseur $\mathcal{SUP}$ symbolique

- calculer le sous-ensemble dit de l'*invariant sous contrôle* ($\mathcal{IUC}$) d'états de P. Tant que P reste dans $\mathcal{IUC}$, le non respect de Q peut être constamment évité. Les détails du calcul de $\mathcal{IUC}$ dépassent le cadre de ce papier. Ils peuvent être consultés dans [38, 68];
- calculer le superviseur $\mathcal{SUP}$: l'ensemble de toutes les transitions $\mathbf{s} \xmapsto{\mathbf{u},\mathbf{c}}$ de P aboutissant à $\mathcal{IUC}$. L'expression précise du superviseur est:

$$\mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) = \exists \mathbf{s}' : \mathcal{T}(\mathbf{s}, \mathbf{u}, \mathbf{c}, \mathbf{s}').\mathcal{IUC}(\mathbf{s}') \quad \text{(A.3)}$$

Une solution de contrôle existe ssi $s_0 \in \mathcal{IUC}$. D'un point de vue dynamique, on dit que $\mathcal{SUP}$ "joue" avec les entrées contrôlables C, contre l'environnement, "jouant" avec les entrées incontrôlables U de P, avec l'objectif de ne jamais atteindre un état violant Q. Ainsi, pour tous les états $\mathbf{s} \in \mathcal{IUC}$ et pour tous les vecteurs d'entrées incontrôlables $\mathbf{u} \in \mathbb{B}^r$, le superviseur $\mathcal{SUP}$ calcule les valeurs adéquates pour les entrées contrôlables $\mathbf{c}$ afin que la transition tirée par P aille vers un état de $\mathcal{IUC}$.

Le superviseur symbolique $\mathcal{SUP}$ a deux propriétés importantes: (I) il est *maximalement permissif* i.e. toutes les transitions de P allant vers $\mathcal{IUC}$ sont comprises dans $\mathcal{SUP}$ et (II) il est *non déterministe du point de vue du contrôle* i.e. pour un état donné et un valeur d'entrée incontrôlable, il peut exister plus d'un successeur possible dans $\mathcal{IUC}$. Notre objectif est d'*implémenter* $\mathcal{SUP}$, ce qui revient à résoudre l'indéterminisme de contrôle, tout en restant maximalement permissif. Cette démarche est développée dans la section suivante.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

A.9 Implémentation Symbolique du Superviseur

A.9.1 Conditions pour la compatibilité structurelle

En général, l'implémentation de superviseurs se traduit par une résolution de l'indéterminisme de contrôle. La solution de contrôle obtenue ainsi obtenue s'appelle un *contrôleur* et est généralement un sous-ensemble du superviseur initialement synthétisé.

Cependant, en plus du non-déterminisme, la SCD symbolique introduit aussi un problème architectural: le superviseur $\mathcal{SUP}$ est représenté par une fonction caractéristique Booléenne encodant les transitions acceptables par rapport à l'objectif de contrôle. L'expression Booléenne de $\mathcal{SUP}$ est structurellement incompatible avec P et ainsi avec l'architecture de contrôle représentée dans la figure A.14(a). Cette incompatibilité structurelle est en général éliminée en résolvant l'équation Booléenne:

$$\mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) = 1 \tag{A.4}$$

soit "en-ligne", grâce à un solveur, comme le montre la figure A.14(b), soit "hors-ligne".

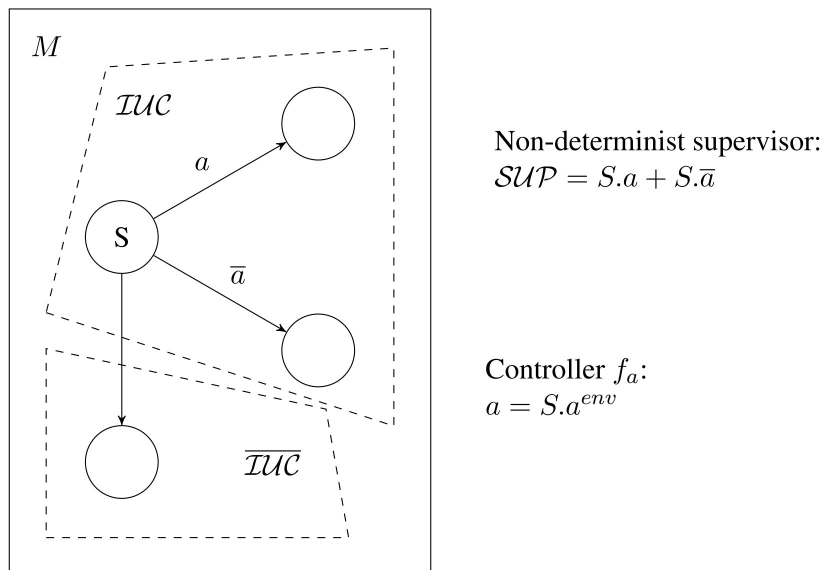


FIGURE A.15: levée du non-déterminisme de contrôle

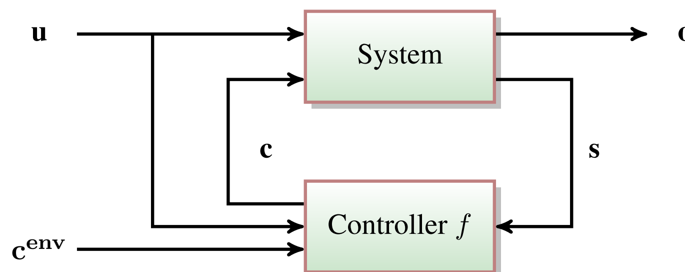


FIGURE A.16: architecture de contrôle souhaitée

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

La technique d'implémentation que nous présentons traite ces deux aspects. Ainsi, l'indéterminisme de contrôle est rendu explicite en ajoutant des variables supplémentaires d'environnement. Dans l'exemple de la figure A.16a, le non-détermnisme de contrôle dans l'état S se traduit par la possibilité de choisir soit $a = 1$ soit $a = 0$ pour rester dans l'ensemble $\mathcal{IUC}$. Cet non-determinisme est levé par l'introduction de la variable d'environnement a^{env}.

L'incompatibilité structurelle est résolue en calculant une expression individuelle pour chaque variable contrôlable $\mathbf{c}$ de P. Cela revient à résoudre l'équation (A.4) hors-ligne. On obtient ainsi l'architecture de la figure A.16b. A partir du superviseur $\mathcal{SUP}$, nous construisons un contrôleur sous la forme d'un vecteur de p fonctions de contrôle:

$$f_i : \mathbb{B}^n \times \mathbb{B}^{p+r} \to \mathbb{B},\ i = 0 \dots p-1,\ \text{tel que:}$$

$$c_i = f_i(s_0, \dots, s_{n-1}, u_0, \dots, u_{r-1}, c_0^{env}, \dots, c_{p-1}^{env}) \tag{A.5}$$

La relation entre $\mathcal{SUP}$ et f est exprimée par l'équation suivante :

$$\mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) = \exists c_0^{env}, \dots, c_{p-1}^{env} : \prod_{i=0}^{p-1} (c_i \Leftrightarrow f_i(\mathbf{s}, \mathbf{u}, \mathbf{c^{env}})) \tag{A.6}$$

Ainsi, notre objectif est de trouver une implémentation f telle que (I) toutes les solutions de contrôle permises par $\mathcal{SUP}$ peuvent être reproduites par f et (II) toutes les solutions de contrôle données par f sont aussi acceptées par $\mathcal{SUP}$. De plus, trouver f satisfaisant l'équation (A.6) revient également à rendre explicite l'indéterminisme de contrôle de $\mathcal{SUP}$ en utilisant les variables $\mathbf{c^{env}}$, comme le montre la figure A.16b.

A.9.2 Algorithme d'implémentation symbolique

Notons au préalable que la condition suffisante pour pouvoir implémenter un superviseur $\mathcal{SUP}$ est la satisfaisabilité :

$$\forall \mathbf{s} \in \mathcal{IUC}, \forall \mathbf{u} \in \mathbb{B}^r, \exists \mathbf{c} : \mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) \tag{A.7}$$

En d'autres termes, le prédicat $\mathcal{SUP}$ doit être satisfaisable sur tout l'ensemble invariant sous contrôle $\mathcal{IUC}$ et quelles que soient les valeurs des variables incontrôlables u. C'est d'ailleurs le cas dès lors qu'une solution de synthèse existe.

Notre algorithme d'implémentation utilise la décomposition de Boole appliquée à $\mathcal{SUP}$, qui est appliquée récursivement aux variables $c_i, i = 0 \dots p-1$:

$$\mathcal{SUP} = \overline{c_i}.\mathcal{SUP}|_{c_i=0} + c_i.\mathcal{SUP}|_{c_i=1} \tag{A.8}$$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

L'impact de la variable c_i sur la satisfaisabilité de $\mathcal{SUP}$ est résumé par le tableau de vérité A.1a. Notre objectif est de trouver une expression donnant les valeurs de c_i telles que $\mathcal{SUP}$ est satisfait. Les valeurs de c_i satisfaisant la décomposition de Boole et telles que $\mathcal{SUP}$ reste satisfaisable sont résumées par la table de vérité présentée dans Table A.1b. Ce tableau résume les valeurs que la variable contrôlable c_i peut prendre, afin que l'équation de superviseur (A.4) soit satisfaisable. Comme montré par cette table, la satisfaction simultanée des deux cofacteurs $\mathcal{SUP}|_{c_i=0}$ et $\mathcal{SUP}|_{c_i=1}$ a une signification particulière: quelle que soit la valeur de c_i, toutes les transitions permises par $\mathcal{SUP}$ vont vers $\mathcal{IUC}$. Ceci caractérise l'indéterminisme de contrôle par rapport à la variable contrôlable c_i. A chaque fois que c_i n'a pas d'influence sur la satisfaisabilité de $\mathcal{SUP}$, sa valeur est déterminée par c_i^{env}. Structurellement, les variables c^{env} sont des variables auxiliaires d'environnement (entrée) comme montré dans la figure A.16.

$\mathcal{SUP}\|_{c_i=1}$	$\mathcal{SUP}\|_{c_i=0}$	c_i	$\mathcal{SUP}$
0	0	0	**0**
0	0	1	**0**
0	1	0	**1**
0	1	1	**0**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

a)

$\mathcal{SUP}\|_{c_i=1}$	$\mathcal{SUP}\|_{c_i=0}$	c_i
0	1	**0**
1	0	**1**
1	1	**0 or 1** (c_i^{env})

b)

TABLE A.1: a) décomposition de Boole appliquée à $\mathcal{SUP}$ par rapport à la variable c_i; b) Valeurs de c_i lorsque $\mathcal{SUP}$ est satisfaisable

D'après le tableau A.1b, l'expression Booléenne suivante calcule le valeur de f_i, associé à la variable contrôlable c_i:

$$f_i = \overline{\mathcal{SUP}|_{c_i=0}}.\mathcal{SUP}|_{c_i=1} + c_i^{env}.\mathcal{SUP}|_{c_i=1}.\mathcal{SUP}|_{c_i=0} \tag{A.9}$$

L'expression (A.9) représente les étapes de base de l'algorithme d'implémentation de contrôleur présenté dans algorithme 7: f_0 est calculé en supposant que $\mathcal{SUP}$ soit vrai. c_0 est substitué par f_0 dans $\mathcal{SUP}$, produisant $\mathcal{SUP}_1$. Ensuite l'algorithme calcule f_1 assumant que $\mathcal{SUP}_1$ soit vrai, etc.

Algorithm 7 Algorithme de mise en œuvre du contrôleur

Require: $\mathcal{SUP}$, un superviseur satisfaisable
1: {démarre avec $\mathcal{SUP}$ et calcule $f_0, \ldots, f_{p-1}$}
2: {résultats intermédiaires: $\mathcal{SUP}_0, \ldots \mathcal{SUP}_p$}
3: $\mathcal{SUP}_0 \leftarrow \mathcal{SUP}$
4: **for** $i = 0 \rightarrow p - 1$ **do**
5: $\quad f_i \leftarrow \overline{\mathcal{SUP}_i|_{c_i=0}}.\mathcal{SUP}_i|_{c_i=1} + c_i^{env}.\mathcal{SUP}_i|_{c_i=1}.\mathcal{SUP}_i|_{c_i=0}$
6: $\quad \mathcal{SUP}_{i+1} \leftarrow$ substituer f_i à la place de c_i dans $\mathcal{SUP}_i$
7: **end for**

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

L'algorithme 7 produit le vecteur de fonctions:

$$f = \begin{pmatrix} f_0(\mathbf{s}, \mathbf{u}, c_0^{env}, f_1, \ldots, f_{p-1}) \\ f_1(\mathbf{s}, \mathbf{u}, c_1^{env}, f_2, \ldots, f_{p-1}) \\ \ldots\ldots \\ f_{p-1}(\mathbf{s}, \mathbf{u}, c_{p-1}^{env}) \end{pmatrix} \tag{A.10}$$

ainsi qu'une expression résiduelle $\mathcal{SUP}_p(\mathbf{s}, \mathbf{u})$ obtenue par les substitutions successives de c_i par f_i dans $\mathcal{SUP}$. Afin de garantir que toutes les solutions de contrôle produites par f sont acceptées par $\mathcal{SUP}$, nous devons montrer que $\mathcal{SUP}_p$ est une tautologie: $\forall \mathbf{s} \in \mathcal{IUC}, \forall \mathbf{u} \in \mathbb{B}^r : \mathcal{SUP}_p$

Theorem 3. $\mathcal{SUP}_p$ est une tautologie.

Proof. En appliquant les substitutions successives de f_i dans $\mathcal{SUP}_i$, l'expression suivante est obtenue pour $\mathcal{SUP}_p$:

$$\mathcal{SUP}_p = \exists c_{p-1}, \ldots, \exists c_0 : \mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) \tag{A.11}$$

Ainsi, le prédicat $\forall \mathbf{s} \in \mathcal{IUC}, \forall \mathbf{u} \in \mathbb{B}^r : \mathcal{SUP}_p$ est identique à notre hypothèse initiale exprimée dans (A.7). □ □

Ce théorème montre qu'en substituant f à la place des $\mathbf{c}$ dans $\mathcal{SUP}$, nous obtenons une tautologie et par conséquent toutes les solutions de contrôle produites par f sont contenues dans $\mathcal{SUP}$. A présent il faut démontrer que:

Theorem 4. Toutes les solutions de contrôle comprises dans $\mathcal{SUP}$ peuvent être reproduites par f.

Proof. L'affirmation ci-dessus est équivalente à:

$$\forall \mathbf{s}, \forall \mathbf{u}, \forall \mathbf{c} : (\mathcal{SUP}(\mathbf{s}, \mathbf{u}, \mathbf{c}) \implies \exists \mathbf{c^{env}} : \mathbf{c} \Leftrightarrow f(\mathbf{s}, \mathbf{u}, c^{env})) \tag{A.12}$$

En appliquant la quantification existentielle et en substituant l'expression de f donnée dans (A.9), le terme à la droite de l'implication est équivalent à $\mathcal{SUP} + \overline{c_i}.\overline{\mathcal{SUP}|_{c_i=0}}.\overline{\mathcal{SUP}|_{c_i=1}}$. Ainsi, l'implication ci-dessus est une tautologie. □ □

Il est important de souligner que le contrôleur f obtenu dépend de l'ordre des variables utilisé lors de l'application de la décomposition de Boole. Cet ordre établit une priorité d'évaluation: f_{p-1} est évalué d'abord, et son valeur est utilisée pour évaluer $f_{p-2} \ldots f_0$.

La complexité de notre algorithme dépend fortement de la taille de $\mathcal{SUP}$ en termes de noeuds de BDD. La plus coûteuse opération est le cofacteur généralisé, qui est utilisée pour substituer une expression pour une variable de BDD. Cette opération est exponentielle au nombre de variables de $\mathcal{SUP}$.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

La génération de contrôleurs est illustrée dans la prochaine section sur un composant matériel modélisant un convertisseur série-parallèle.

A.10 Application de la SCD à la conception de matériel

Dans cette section, nous montrons l'application de la technique de SCD à deux contextes de conception de matériel: la conception basée sur composants et la correction automatique des erreurs.

A.10.1 Conception basée sur composants

Dans la conception de matériel, la méthodologie de conception de réutilisation des composants est importante. Selon ce paradigme, les concepteurs habituellement commencent par choisir un groupe de composants à partir d'une bibliothèque matérielle, par rapport à la spécification proposée de la conception, ces éléments sont assemblés/interconnectés pour former une conception globale; les simulations et les correction sont effectuées pour s'assurer le respect de la spécification. Cette dernière étape peut être fastidieux et une source d'erreurs.

La SCD (incrémentale) et les techniques d'implémentation de superviseur peuvent être très utiles au sein d'un processus de conception basée sur composants: la SCD peut générer automatiquement un superviseur correcte par construction qui, une fois composé avec l'assemblage des composants, on obtient un système contrôlé correct. Ainsi, dans ce contexte, la SCD est utilisée pour *finaliser une conception partiellement construite*.

Comme le montre la Figure A.17, le concepteur de matériel utilise une ou plusieurs bibliothèques de composants réutilisables. Il/elle prend les éléments intéressants par rapport à la fonctionnalité à construire. Ces éléments sont ensuite reliés les uns aux autres, selon leur interface respective. Le plus souvent, la fonctionnalité ciblée n'est pas obtenue par une connexion simple des composants. D'une part, les configurations de cas pathologique, qui violent la spécification initiale, peuvent rester, ou être introduites par l'assemblage des composants. Pour éviter une telle configuration erronée, une logique supplémentaire doit généralement être ajoutée. D'autre part, le concepteur peut choisir de construire son système partiellement, et mettre fin au processus de conception par rapport à une spécification donnée par synthèse d'un superviseur.

Dans les deux cas, un superviseur peut soit éliminer des configurations erronée, ou tout simplement être la composante qui finalise la conception. Pour ce faire, le concepteur a besoin de désigner un ensemble de variables contrôlables d'entrée $\mathbf{C}$. Le choix des variables contrôlables dépend fortement de la connaissance du concepteur de la conception à la main et de la spécification à renforcer.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Un superviseur est alors généré, implémenté, et est enfin relié au système M. Le concepteur peut avoir besoin d'ajouter manuellement des logiques supplémentaires, en précisant comment les entrées auxiliaires de contrôle c^{env} peuvent être utilisées par l'environnement du système contrôlé. Le système est alors simulé: la SCD applique une spécification donnée en limitant le comportement du système à contrôler, il est important de vérifier si les comportements intéressants n'ont pas été enlevés. Si cela se produit, une tentative de contrôle peut être réalisé, en choisissant un autre ensemble des entrées contrôlables.

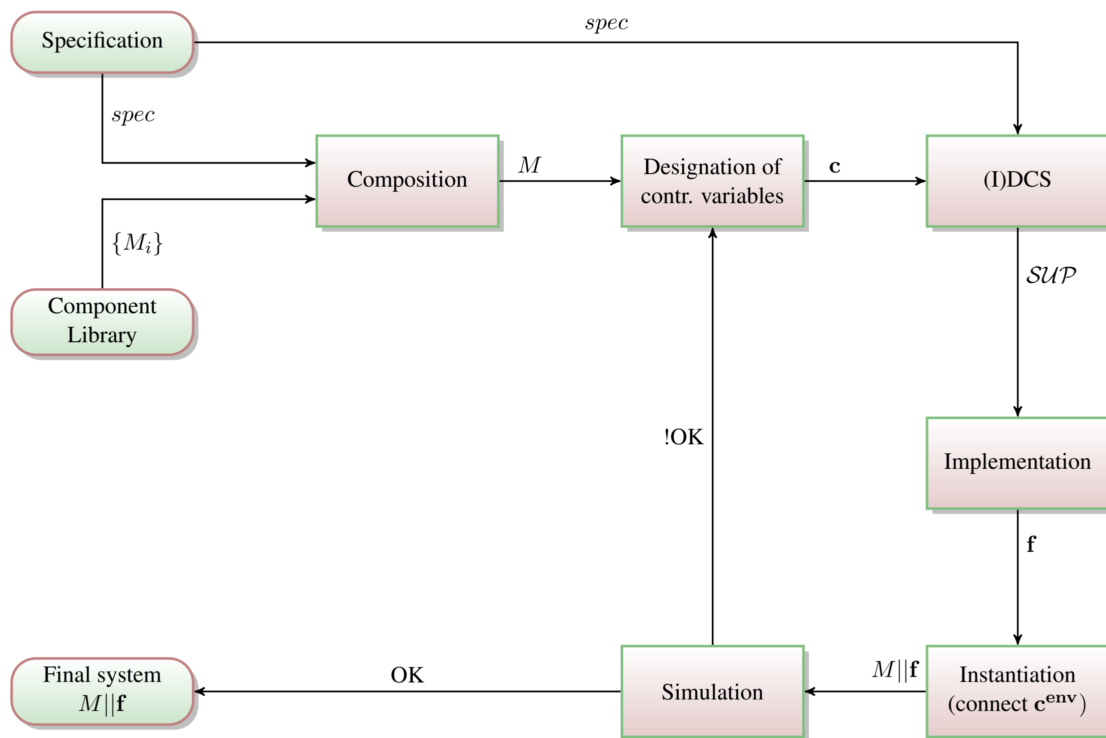


FIGURE A.17: Flot de conception basée sur composants

Exemple de l'arbitre avec jeton Nous appliquons notre approche d'implémentation à l'arbitre avec jeton, présenté dans la section A.5. Le concepteur dispose d'une collection de cellules $M_i, i = 1, 2, 3$. Il tient à les relier entre eux pour composer un arbitre satisfaisant la spécification de l'exclusion mutuelle. La composition $M_1||M_2||M_3$ laisse la gestion du jeton complètement non modélisé. Comme le mécanisme de jeton est fondamental dans la réalisation de l'exclusion mutuelle, la synthèse d'un superviseur qui contrôle l'arrivée du jeton en entrée tin_1 peut être une solution pour finaliser automatiquement la conception de l'arbitre. Nous voulons garantir la propriété d'exclusion mutuelle par l'insertion de jeton qu'aux moments appropriées.

La fonction de contrôle générés est trop volumineux pour être présentée ici. Figure A.18 montre toute la conception qui satisfait la spécification de l'exclusion mutuelle. Le superviseur est généré, et une entrée auxiliaire est ajoutée pour la variable contrôlable tin_1.

A tout moment, l'environnement de l'arbitre peut *essayer* d'insérer un jeton via tin_1^{env}. Le superviseur permet ce jeton ou le filtre, si son insertion peut constituer une violation de l'exclusion mutuelle.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

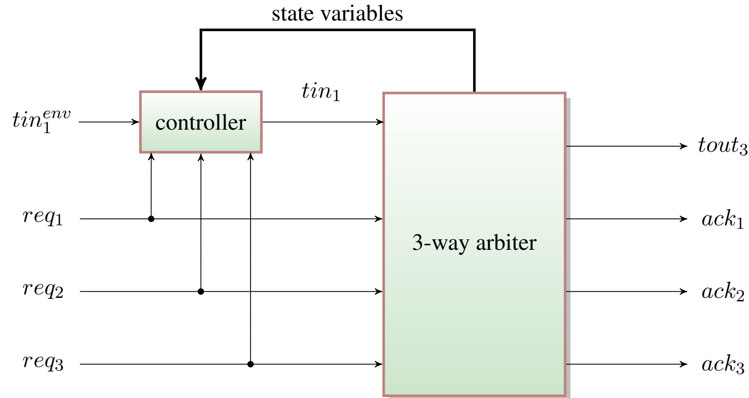


FIGURE A.18: Implémentation de l'arbitre contrôlé

Plusieurs possibilités existent pour la connexion de tin_1^{env}. Il peut être nécessaire qu'un jeton est toujours inséré lorsque cela est possible. Cela s'agit d'affecter $tin_1^{env} = 1$. Toutefois, s'il y a aucune demande, pourquoi un jeton est inséré en premier temps? Ainsi, Une autre possibilité serait d'insérer un jeton que lorsque cela est nécessaire et quand son insertion est possible:

$$tin_1^{env} = req_1 + req_2 + req_3,$$

Toute configuration de tin_1^{env} est sûre en ce qui concerne l'exclusion mutuelle. Toutefois, une configuration tel que $tin_1^{env} = 0$ désactiverait le service pour tous les clients pour toujours (ce qui est encore parfaitement sécurité par rapport à l'exclusion mutuelle). C'est pourquoi nous recommandons l'utilisation de la simulation en vue d'assurer la validité du comportement de l'arbitre contrôlé.

A.10.2 Correction automatique des erreurs

Dans la section précédente, nous avons discuté l'application de la SCD et de l'implémentation de superviseur dans la phase de conception de matériel. Toutefois, si une erreur est constatée après la conception ou même après la phase de fabrication, il est souvent difficile, voire impossible, de la corriger. Nous montrons que la SCD peut être facilement utilisée avec succès pour trouver un "patch" corrigeant.

La méthode de la conception est représentée à la Figure A.19. La "conception soupçonnée" dispose un comportement inattendu, qui peut être représentée comme une propriété d'accessibilité CTL $\overline{spec} = EFp$. Cette propriété peut être formellement vérifiée par la vérification du modèle. Le concepteur peut même chercher un témoin de $\overline{spec}$ par l'évaluation de $spec = AG\overline{p}$ et en générant un contre-exemple qui montre que $\overline{spec}$ peut arriver. Ensuite, il peut tenter de renforcer $spec$ par la SCD. Cette méthode est illustrée par l'exemple suivant.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

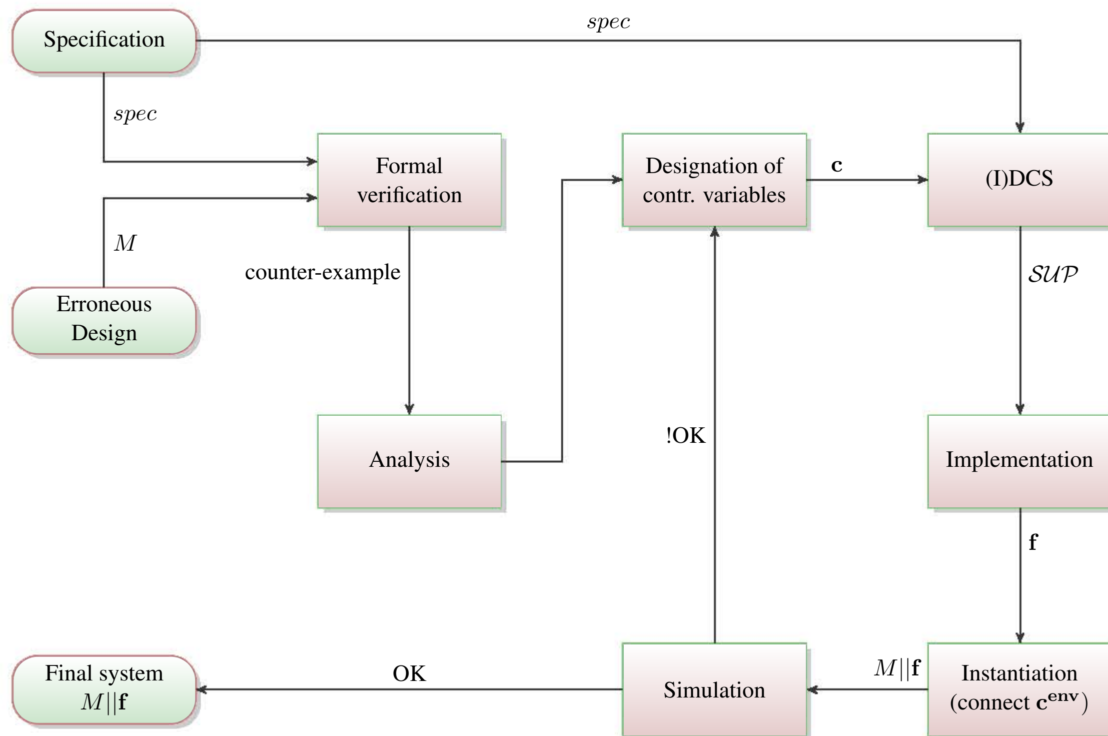


FIGURE A.19: Flot de conception de la correction des erreurs

A.10.3 Le convertisseur série-parallèle

Nous illustrons l'application de notre technique de génération de contrôleurs dans un contexte de conception matérielle, avec le but de construire un convertisseur série-parallèle correct par construction.

La figure A.21(a) présente l'architecture d'un convertisseur série-parallèle. Il est composé par deux composants: un tampon série (SB) et un accumulateur (WCB) qui interagissent par l'intermédiaire d'un produit synchronisé. De plus amples détails de la modélisation peuvent être trouvés dans [69].

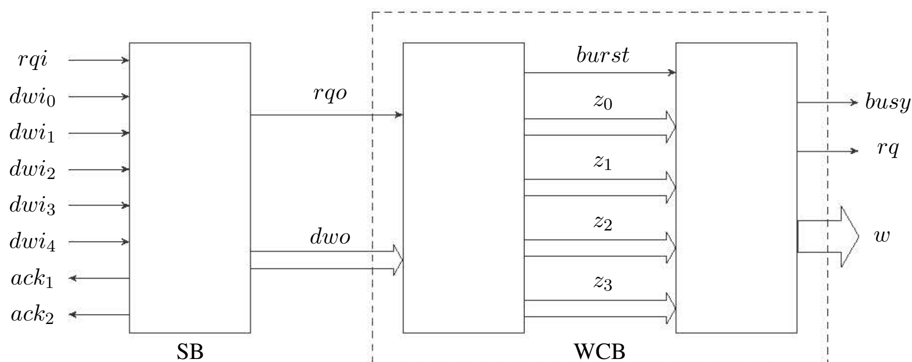


FIGURE A.20: architecture du convertisseur série-parallèle

Le composant SB reçoit des données série en 4-bits et ensuite fait un contrôle de parité. Selon le résultat, SB envoie un acquittement ou un code d'erreur par akn à l'environnement.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

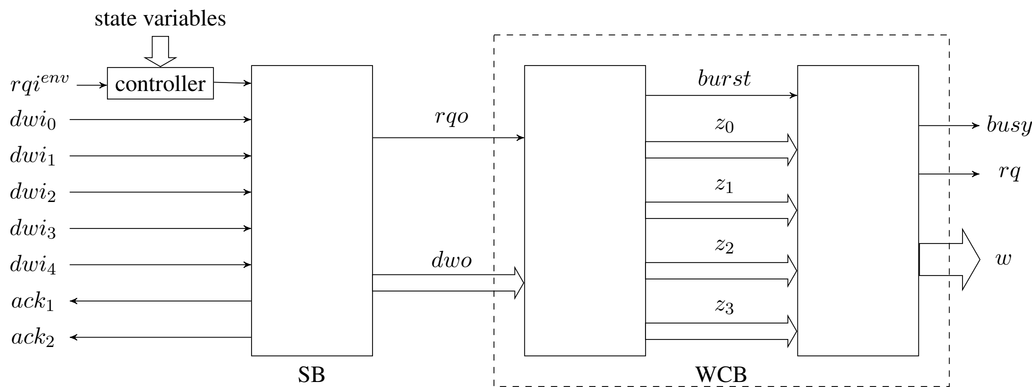


FIGURE A.21: correction d'erreurs de conception par SCD

Les données série arrivent selon une transaction bien établie: l'environnement envoie les données sur dw et positionne une requête d'entrée *rqi*, signifiant que des données d'entrée valides sont disponibles. Après une transition, correspondant à un moment de temps synchrone, SB acquitte l'envoyeur en envoyant le résultat du contrôle de parité. Ainsi, toute transaction entrante dure toujours deux moments de temps synchrone. A noter que le mécanisme de "poignée de main" n'est utilisé que pour garantir la correction des données entrantes par rapport au contrôle de parité. De plus, l'environnement est supposé maintenir sa requête et les données jusqu'à ce qu'il soit acquitté.

Le SB renvoie toutes les données série correctes au WCB. Ce composant implémente deux comportements:

- il accumule les données série dans un tampon interne, en construisant des mots de 8 bits;
- lorsque le tampon interne est plein, il est purgé en renvoyant en parallèle toutes les données accumulées. Pendant l'envoi, WCB est *occupé* et il ne peut plus recevoir de données entrantes. Il devient disponible à nouveau dès que l'envoi est fini.

SB implémente un comportement de "pipeline", i.e. à un moment donné t il peut simultanément envoyer données séries correctes au WCB et acquitter une donnée série entrante.

A.10.4 Analyse de correction

Les composants SB et WCB sont des composants sur étagère, qui ont été vérifiés complètement et indépendamment l'un de l'autre. Ils sont corrects par rapport à leur spécifications fonctionnelles. Pourtant, ils n'ont pas été conçus pour fonctionner ensemble.

Nous souhaitons obtenir un convertisseur série-parallèle comme une nouvelle fonctionnalité obtenue en combinant deux composants existants et corrects. Pourtant, lorsque SB et WCB sont combinés, le composant résultant ne fonctionne pas comme attendu. Une propriété importante que le convertisseur

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

doit avoir est que toutes les données série doivent être retransmises, i.e. une donnée série n'est jamais perdue. Cette propriété est exprimée par la formule temporelle CTL [37] suivante:

$$\textit{Pas de perte: } AG\ \overline{busy \wedge rqo}$$

Ainsi, SB n'enverra jamais une donnée au WCB pendant un flush. Cette propriété est vraie sauf pour le scénario suivant: à certain moment t, WCB a presque rempli tous ses tampons internes sauf un emplacement disponible pour une donnée série. En même temps, SB renvoie une données série au WCB et acquitte une donnée série simultanément. Au moment $t+1$, WCB est plein et commence à purger les données accumulées. Pourtant, en même temps, SB lui envoie la donnée série qu'il vient d'acquitter. Comme WCB est occupé durant la purge, cette donnée est perdue. Ce scénario est confirmé par "model-checking" symbolique, qui démontre que la propriété *"Pas de perte"* est fausse.

Notre but est de garantir la satisfaction de la propriété *"Pas de perte"* sans réécrire ni SB ni WCB. Nous utilisons la SCD symbolique afin de synthétiser un "patch" [58, 59] qui assure la satisfaction de la propriété ci-dessus.

A.10.5 Correction de l'erreur par SCD

Afin d'appliquer la SCD à l'exemple du convertisseur, il faut préalablement déterminer les variables d'entrée contrôlables. Il n'y a pas d'indication a priori sur la ou les entrées à contrôler. Pourtant, on observe qu'en retardant judicieusement l'arrivée des données série, il est possible d'éviter la perte de données. Retarder l'arrivée d'une donnée série entrante revient à retarder l'arrivée de la requête entrante de rqi. Ainsi, nous choisissons rqi comme la variable contrôlable d'entrée. Les variables d'entrée restantes sont considérées comme étant incontrôlables.

Le superviseur a été synthétisé par l'outil de SCD *Sigali* [60]. Ensuite, nous avons implémenté le superviseur en générant un contrôleur grâce à l'algorithme 7 présenté dans la section III. L'architecture du convertisseur contrôlé est représentée dans la figure A.21(b), et correspond donc à notre objectif d'architecture représenté dans la figure A.16.

L'algorithme de mise en œuvre a introduit la variable auxiliaire de d'environnement $rqi - env$, correspondant à la variable contrôlable rqi. Il est à noter que la requête entrante devient $rqi - env$, qui est pilotée par l'environnement et lue par le contrôleur synthétisé. Selon la valeur de $rqi - env$ et l'état interne du procédé (i.e. le convertisseur série-parallèle), le contrôleur affecte les valeurs adéquates à rqi.

Par conséquent, le contrôleur que nous obtenons fonctionne comme un filtre sur l'entrée $rqi - env$. Lorsqu'une donnée série arrive, soit le contrôleur la transmet au SB, soit il la retarde si la donnée peut être perdue. A tout instant t, si une série donnée arrive, $rqi - env$ est activé par l'environnement. Deux scénarios sont possibles:

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

- il ne reste qu'un emplacement mémoire libre dans WCB, et cet emplacement va être rempli par SB avec une donnée acquise précédemment, à l'instant $t-1$. En ce moment, si $rqi-env$ est acquitté, la donnée sera perdue à l'instant $t+1$. Cependant, dès la réception de $rqi-env$ le contrôleur affecte $rqi=0$. De cette manière, la requête entrante n'est pas transmise au SB et de ce fait, son acquittement sera différé;

- il reste plus d'un emplacement mémoire libre dans WCB. Dans ce cas, lors de la réception de $rqi-env$ le contrôleur affecte $rqi=1$. La requête entrante est transmise au SB pour qu'elle soit acquittée.

Ainsi, la transaction d'entrée est retardée, dans le sens où SB ne la reçoit pas, tant que la retransmission de la donnée série entrante n'est pas garantie sans perte. A noter qu'initialement, les transactions entrantes n'étaient utilisées que pour assurer la correction de parité; en appliquant la SCD, ces transactions garantissent en plus que le convertisseur série-parallèle est réellement prêt pour le traitement complet d'une donnée série.

L'indéterminisme de contrôle induit par la synthèse du superviseur est "déplacé" vers l'environnement. Lorsqu'un choix non-déterministe existe sur la valeur de rqi, toute valeur affectée par l'environnement à $rqi-env$ se répercute sur rqi.

Le mécanisme de filtrage des entrées s'appuie sur l'existence du non-déterminisme de contrôle. En effet, si le contrôleur était déterministe, alors la décomposition en fonctions engendrerait un système en boucle fermée.

Le contrôleur que nous obtenons est un BDD comprenant les entrées et les variables d'état du convertisseur. Il comprend environ 200 noeuds, et ne peut de ce fait pas être représenté.

A.11 Mise en œuvre

Le convertisseur série-parallèle a été modélisé en utilisant le langage des Automates de Mode et le compilateur Matou [70]. Le superviseur a été synthétisé par l'outil de SCD symbolique *Sigali* [60]. Les simulations intéractives ont été réalisées en utilisant l'outil *SigalSimu*. Le flot de conception est illustré dans la figure A.22.

Notre algorithme d'implémentation a été codé avec l'aide de la bibliothèque CUDD [62]. Il lit le superviseur produit par *Sigali* et produit un contrôleur qui est composé de plusieurs BDDs, chacun représentant une fonction de contrôle. Après l'étape d'implémentation, le procédé contrôlé par le contrôleur implémenté peut être vérifié en utilisant des outils de simulation ou de vérification formelle, et peut également être synthétisé vers une cible matérielle.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

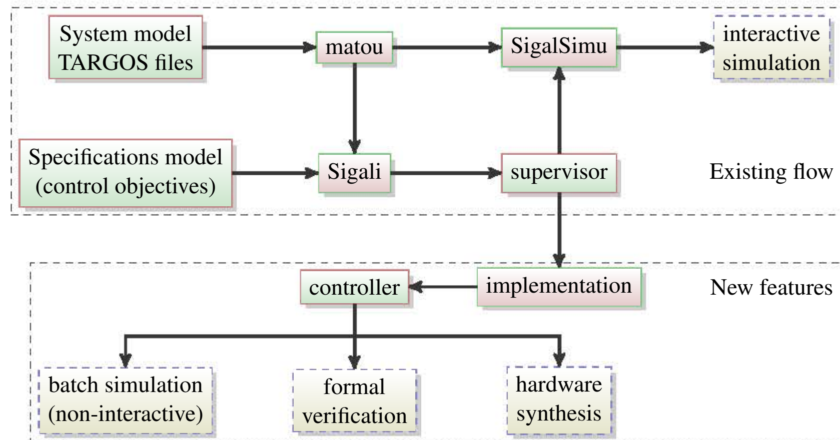


FIGURE A.22: Flot de conception avec implémentation de contrôleurs

Nous avons présenté et illustré une technique d'implémentation de superviseurs obtenus par SCD symbolique. Cette technique résout le non-déterminisme de contrôle ainsi que l'incompatibilité structurelle du superviseur par rapport au procédé. L'avantage très important de notre méthode est la conservation de l'intégralité des solutions de contrôle. Cette propriété est très intéressante par rapport à une déterminisation simpliste du superviseur synthétisé, qui produit presque toujours un sous-ensemble strict de $\mathcal{SUP}$. De plus, on note que le superviseur implémenté f est généralement plus compact que $\mathcal{SUP}$, ce qui rend son utilisation plus efficace.

Par ailleurs, l'introduction des variables auxiliaires c^{env} peut sembler discutable, notamment en ce qui concerne la signification physique de ces variables. L'architecture de contrôle que nous proposons est une interprétation particulière de la boucle de contrôle traditionnelle, illustrée dans la figure A.14. Nous avons adapté cette architecture de contrôle au contexte de la conception matérielle. Dans ce contexte, la distinction entre variables contrôlables et incontrôlables n'est pas naturelle. Les variables d'entrée sont généralement dédiées à l'environnement du procédé. C'est pourquoi l'architecture de contrôle que nous proposons ne change pas l'interface d'entrée/sortie du procédé. Les entrées contrôlables sont en réalité *filtrées* par le contrôleur. Il faut noter cependant que le fait de filtrer les entrées d'un composant peut se révéler dangereux par rapport à son comportement recherché. Dans la section IV nous avons identifié une classe d'applications où le filtrage des entrées peut être effectué sans risque.

Notons que l'implémentation des superviseurs produits par des techniques énumératives est algorithmiquement plus simple, car ils n'ont pas d'incompatibilité structurelle par rapport au procédé à contrôler. Implémenter un superviseur explicite revient à résoudre l'indéterminisme de contrôle. Pourtant, la SCD énumérative est coûteuse lors de son application à des systèmes de taille réelle. Notre choix pour les techniques de SCD symbolique est motivé par leur capacité de traiter des systèmes de taille réelle: bien que toujours sujettes à l'explosion combinatoire, elles restent bien plus efficaces que les techniques de SCD énumérative basées sur [6].

Cependant, alors que la SCD symbolique est moins sensible au problème de l'explosion combinatoire, sa complexité dépend toujours du nombre de variables d'état présentes dans le procédé. De plus, notre

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

algorithme de mise en œuvre ajoute sa propre complexité à la génération d'un contrôleur. C'est pourquoi une perspective importante de recherche sera de développer un algorithme plus efficace pour la décomposition structurelle du superviseur ainsi que l'étude de méthodes de synthèse compositionnelle.

A.12 Conclusion

Ce travail préconise l'introduction de la technique de la Synthèse symbolique de contrôleurs discrets dans le flot de conception de matériel embarqué, outre la vérification formelle et la simulation. Comme la vérification formelle, la SCD basée sur BDD est confrontée aux mêmes complexité théorique exponentielle. Nous avons proposé une technique de SCD incrémentale (SICD) qui traite le problème de complexité exponentielle et tend à donner des résultats intéressants de performance. La SICD est entièrement basée sur l'algorithme de SCD classique et exploite la structure modulaire du système à contrôler.

En tant que les superviseurs générés par la SCD ne sont pas structurellement compatible avec la conception de matériel, une architecture appropriée de contrôle pour la conception de matériel est proposée. Un superviseur est construit pour contrôler un système en agissant à travers un ensemble d'entrées qui sont *désigné* comme *contrôlables*. Contrôler une entrée s'élève actuellement à filtrer ses valeurs provenant de l'environnement, selon l'état actuel du système à contrôler et à la spécification de contrôle. Le non-déterminisme de contrôle se produit lorsque les choix sont possibles pour la valeur d'une ou plusieurs entrées contrôlables. Dans notre architecture de contrôle, ce choix est toujours fait par l'environnement, en assignant les variables auxiliaires d'entrée.

Afin de rendre réalisable cette architecture de contrôle, le superviseur est systématiquement transformé depuis sa représentation équationnelle à une représentation fonctionnelle. Une méthode de conception a été proposée, basée sur cette approche, et ses spécificités ont été mises en évidence: soit finaliser une conception par rapport à une spécification, ou générer un "patch" pour une conception comporte une erreur. Cette méthode de conception intègre la SCD avec la vérification formelle et la simulation traditionnelle. Elle a été illustrée sur les conceptions simples mais réalistes de matériel.

Ces contributions ont été mises en œuvre et validées sur un ensemble de benchmarks. Les résultats montrent des améliorations des performances intéressantes pour la mémoire usagée et le temps d'exécution. Toutefois, l'ordre dans lequel la SICD considère chaque module d'un système doit être spécifié par l'utilisateur. Ce choix est important pour la performance de la SICD. Trouver un bon ordre pour la SICD est parti comme un axe de recherche future ce ce travail. En outre, comme la plupart des techniques symboliques basées sur BDD, la performance de la SICD est parfois touchée par les manipulations de BDD, telles que la technique d'ordonnancement des variables.

Il devrait être noté que la technique de SCD que nous avons utilisée seulement applique invariance d'un ensemble d'états. Dans ce travail, nous n'avons pas étudié l'accessibilité et la vivacité. Renforcer l'accessibilité par la SCD serait une fonctionnalité intéressante. Cela pourrait être utilisé pour garantir,

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

par exemple, que certaines fonctionnalités souhaitées peuvent toujours être activées, ou, comme c'est le cas pour le contrôle des systèmes de fabrication, où l'état de terminaison peut toujours être atteint.

Ainsi, les directions de recherche future de ce travail incluent l'élaboration d'un algorithme de SCD capable de faire respecter la vivacité couplée avec des contraintes d'équité sur un système existant, et d'étudier les améliorations possibles qui peuvent être atteintes

Enfin, il semble clair que la SICD peut facilement coopérer avec les techniques de SCD compositionnelle. Dans un processus de la SCD compositionnelle, la SICD peut accélérer le calcul des solutions locales de contrôle. D'autre part, la SICD bénéficie d'une approche compositionnelle en exploitant uniquement sur les modules de taille limitée. Développer une approche intégrant la SCD incrémentale et les techniques compositionnelles peut entraîner d'importantes améliorations de la performance.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

Bibliography

[1] Nicely Thomas R. Pentium fdiv flaw faq. http://www.trnicely.net/pentbug/pentbug.html, 2010.

[2] A. Pnueli. The temporal logic of programs. **In** : Proceedings of the 18th IEEE Symposium on Foundations of Computer Science. 1977, pp. 46–67.

[3] Marchand H., Bournai P., Borgne M. Le, et al. Synthesis of discrete-event controllers based on the signal environment. Discrete Event Dynamic System: Theory and Applications, vol. 10, #4, 2000, pp. 325–346.

[4] Ramadge P. J.G Wonham W. M. The control of discrete event systems. Proceedings of the IEEE, vol. 77, #1, 1989, pp. 81–98.

[5] Bryant R. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, 1986.

[6] Ramadge P.J.G. Wonham W.M. The control of discrete event systems. Proceedings of the IEEE, vol. 77, #1, 1989, pp. 81–98.

[7] Wonham W.M. Supervisory control of Discrete-Event systems, 2007.

[8] Cassandras Christos G. Lafortune Stéphane. Introduction to Discrete Event Systems (Second Edition). Springer, 2008.

[9] Büchi J. Richard. Symposium on decision problems: On a decision method in restricted second order arithmetic. **In** : Ernest Nagel Patrick Suppes Tarski Alfred, eds., Logic, Methodology and Philosophy of Science, Proceeding of the 1960 International Congress, vol. 44 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1966, pp. 1 – 11.

[10] Milner R. Calculi for synchrony and asynchrony. Theoretical Computer Science, vol. 25, #3, 1983, pp. 267–310.

[11] Lind-Nielsen Jørn. Buddy - a binary decision diagram package @ONLINE, 2010. Http://sourceforge.net/projects/buddy/.

[12] Whaley John. Javabdd: a java library for manipulating bdds, 2007. Http://javabdd.sourceforge.net/.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

[13] Sanghavi Jagesh V., Ranjan Rajeev K., Brayton Robert K., et al. High performance bdd package by exploiting memory hierarchy. **In** : Proceedings of the 33rd annual Design Automation Conference, DAC '96. New York, NY, USA: ACM, 1996. ISBN 0-89791-779-0, pp. 635–640.

[14] Sasao Tsutomu. Ternary decision diagrams survey. **In** : Proc. ISMVL '97. 1997, pp. 241–250.

[15] Clarke E. M., Emerson E. A., Sistla A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst., vol. 8, #2, 1986, pp. 244–263.

[16] McMillan Kenneth L. Symbolic Model Checking - An approach to the state explosion problem. Ph.D. thesis, Carnegie Mellon University, 1992.

[17] Shimizu Kanna, Dill David L., Hu Alan J. Monitor-based formal specification of pci. **In** : In Formal Methods in Computer-Aided Design. Springer-Verlag, 2000, pp. 335–352.

[18] Bloem Roderick, Galler Stefan, Jobstmann Barbara, et al. Specify, compile, run: Hardware from PSL. Electronic Notes in Theoretical Computer Science, vol. 190, #4, 2007, pp. 3–16.

[19] Coussy Philippe Morawiec Adam, eds. High-Level Synthesis. Dordrecht: Springer Netherlands, 2008. ISBN 978-1-4020-8587-1.

[20] Wonham W. Ramadge P. Modular supervisory control of discrete-event systems. Mathematics of Control, Signals, and Systems (MCSS), vol. 1, #1, 1988, pp. 13–30.

[21] Vaz A. F. Wonham W. M. On supervisor reduction in discrete-event systems. Int. J. Control, vol. 44, 1986, pp. 475–491.

[22] Su R. Wonham W. M. Supervisor reduction for discrete-event systems. Discrete Event Dynamic Systems, vol. 14, #1, 2004, pp. 31–53.

[23] Schmidt Klaus, Marchand Hervé, Gaudin Benoit. Modular and Decentralized Supervisory Control of Concurrent Discrete Event Systems Using Reduced System Models. **In** : Workshop on Discrete Event Systems, WODES'06. Ann-Arbor United States: IEEE Computer society, 2006, pp. 149–154.

[24] Feng Lei Wonham Murray. Computationally efficient supervisor design: Abstraction and modularity. **In** : Proceedings of the 8th International Workshop on Discrete Event Systems, WODES'06. 2006, pp. 8, 3.

[25] Feng Lei. Computationally Efficient Supervisor Design For Discrete-Event Systems. Ph.D. thesis, University of Toronto, 2007.

[26] Rong Su Jan H. van Schuppen Rooda Jacobus E. Supervisor synthesis based on abstractions of nondeterministic automata. **In** : Discrete Event Systems, 2008 9th International Workshop on. Goteborg, Sweden, 2008, pp. 412–418.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

[27] Flordal H. Malik R. Supervision equivalence [supervisor synthesis]. **In** : Discrete Event Systems, 2006 8th International Workshop on. 2006, pp. 155 –160.

[28] Flordal Hugo, Malik Robi, Fabian Martin, et al. Compositional synthesis of maximally permissive supervisors using supervision equivalence. Discrete Event Dynamic Systems, vol. 17, #4, 2007, pp. 475–504.

[29] Malik Robi Flordal Hugo. Yet another approach to compositional synthesis of discrete event systems. **In** : Discrete Event Systems, 2008 9th International Workshop on. Goteborg, Sweden, 2008, pp. 16–21.

[30] Hill D.M. R.C. Tilbury. Modular supervisory control of discrete-event systems with abstraction and incremental hierarchical construction. **In** : Discrete Event Systems, 2006 8th International Workshop on. Ann Arbor, MI, 2006, pp. 399–406.

[31] Hill R.C., Tilbury D.M., Lafortune S. Modular supervisory control with equivalence-based conflict resolution. **In** : American Control Conference, 2008. 2008, pp. 491 –498.

[32] Gaudin B. Marchand H. Modular supervisory control of a class of concurrent discrete event systems. **In** : Workshop on Discrete Event Systems, WODES'04. 2004, pp. 181–186.

[33] Gaudin Benoît. Synthèse de contrôleurs sur des systèmes à événements discrets structurés. Ph.D. thesis, Université de Rennes I, 2004.

[34] Back Ralph-Johan Seceleanu Cristina. Contracts and games in controller synthesis for discrete systems. **In** : 11th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS'04). IEEE Computer Society, 2004, pp. 307–315.

[35] Delaval Gwenaël, Marchand Hervé, Rutten Éric. Contracts for modular discrete controller synthesis. **In** : ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010). Stockholm, Sweden, 2010, pp. 57–66.

[36] Marchand Hervé Rutten Éric. Managing multi-mode tasks with time cost and quality levels using optimal discrete control synthesis. **In** : Proceedings of the 14th Euromicro Conference on Real-Time Systems. Washington, DC, USA: IEEE Computer Society, 2002. ISBN 0-7695-1665-3, pp. 241–.

[37] Clarke Edmund M. Emerson E. Allen. Design and synthesis of synchronization skeletons using branching-time temporal logic. **In** : Logic of Programs, Workshop. London, UK: Springer-Verlag, 1982. ISBN 3-540-11212-X, pp. 52–71.

[38] Marchand Hervé. Méthode de Synthèse d'automatismes décrits par des systèmes à événements discrets finis. Ph.D. thesis, Université de Rennes I, 1997.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

[39] Dumitrescu E., Girault A., Marchand H., et al. Optimal discrete controller synthesis for modeling fault-tolerant distributed systems. **In** : IFAC Workshop on Dependable Control of Discrete-event Systems , DCDS'07. Cachan, France, 2007.

[40] Peire J. The open microprocessor systems initiative (OMI). the european largest microprocessor industry challenge. **In** : Industrial Electronics, Control, and Instrumentation, 1996., Proceedings of the 1996 IEEE IECON 22nd International Conference on. Taipei, 1996.

[41] Siemens. OMI 324: PI-Bus peripheral interconnect bus, draft standard, rev. 0.3d, 1994.

[42] Jas Abhijit, Sen Alper, Akturan Cagdas, et al. Examples of hardware verification using VIS, 2008.

[43] Fabian M. Hellgren A. PLC-based implementation of supervisory control for discrete event systems. **In** : Decision and Control, 1998. Proceedings of the 37th IEEE Conference on, vol. 3. 1998, pp. 3305–3310.

[44] Flordal H., Fabian M., Akesson K., et al. Automatic model generation and plc-code implementation for interlocking policies in industrial robot cells. Control Engineering Practice, vol. 15, #11, 2007, pp. 1416–1426.

[45] Silva D.B., Santos E.A.P., Vieira A.D., et al. Application of the supervisory control theory to automated systems of multi-product manufacturing. **In** : Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on. 2007, pp. 689–696.

[46] de Queiroz Max H. Cury Jose E. R. Synthesis and implementation of local modular supervisory control for a manufacturing cell. **In** : Proc. 6th International Workshop on Discrete Event Systems (WODES'02). Zaragoza, Spain: IEEE Computer Society, 2002, pp. 377–382.

[47] Passerone Roberto, Rowson James A., Sangiovanni-Vincentelli Alberto L. Automatic synthesis of interfaces between incompatible protocols. **In** : Design Automation Conference. 1998, pp. 8–13.

[48] Passerone Roberto, de Alfaro Luca, Henzinger Thomas A., et al. Convertibility verification and converter synthesis: two faces of the same coin. **In** : ICCAD'02. 2002, pp. 132–139.

[49] Raclet Jean-Baptiste. Residual for component specifications. Electronic Notes in Theoretical Computer Science, vol. 215, 2008, pp. 93 – 110. Proceedings of the 4th International Workshop on Formal Aspects of Component Software (FACS 2007).

[50] Aagaard M. D, Jones R. B, Seger C. -J.H. Formal verification using parametric representations of boolean constraints. **In** : Design Automation Conference, 1999. Proceedings. 36th. 1999, pp. 402–407.

[51] Tronci Enrico. Automatic synthesis of controllers from formal specifications. **In** : ICFEM. 1998, pp. 134–143.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

[52] Marchand H. Le Borgne M. Note sur la triangulation d'une équation polynomiale. Tech. rep., IRISA, 2004.

[53] Bloem R., Bloem R., Galler S., et al. Automatic hardware synthesis from specifications: A case study. **In** : Galler S., ed., Proc. Design, Automation & Test in Europe Conference & Exhibition DATE '07. 2007, pp. 1188–1193.

[54] Bulach S., Brauchle A., Pfleiderer H.-J., et al. Design and implementation of discrete event control systems: A petri net based hardware approach. Discrete Event Dynamic Systems, vol. 12, #3, 2002, pp. 287–309.

[55] An Xin, Delaval Gwenaël, Marchand Hervé, et al. A bzr user manual, 2011.

[56] Aboubekr Soufyane, Delaval Gwenaël, Rutten Éric. A programming language for adaptation control: Case study. **In** : 2nd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES 2009). ACM SIGBED Review, vol. 6. Grenoble, France, 2009, pp. 11:1–11:5.

[57] Clarke Edmund Emerson E. Design and synthesis of synchronization skeletons using branching time temporal logic. **In** : Logics of Programs. Springer Berlin / Heidelberg, 1982, pp. 52–71.

[58] Tivoli Massimo, Fradet Pascal, Girault Alain, et al. Adaptor synthesis for real-time components. **In** : Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'07. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN 978-3-540-71208-4, pp. 185–200.

[59] Sinha Roopak, Roop Partha S., Basu Samik. A module checking based converter synthesis approach for SoCs. **In** : Proceedings of the 21st International Conference on VLSI Design. IEEE Computer Society, 2008. ISBN 0-7695-3083-4, pp. 492–501.

[60] Marchand Hervé, Bournai Patricia, Borgne Michel Le, et al. Synthesis of Discrete-Event controllers based on the signal environment. Discrete Event Dynamic System: Theory and Applications, vol. 10, #4, 2000, pp. 325–346.

[61] Maraninchi Florence Rémond Yann. Mode-Automata: a new domain-specific construct for the development of safe critical systems. Science of Computer Programming, vol. 46, #3, 2003, pp. 219–254.

[62] Somenzi F. CUDD: CU decision diagram package release, 1998.

[63] Bloem Roderick, Galler Stefan, Jobstmann Barbara, et al. Interactive presentation: Automatic hardware synthesis from specifications: a case study. **In** : DATE '07: Proceedings of the conference on Design, automation and test in Europe. New York, NY, USA: ACM Press, 2007. ISBN 9783981080124, pp. 1188–1193.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés

[64] Dumitrescu Emil, Girault Alain, Marchand Hervé, et al. Multicriteria optimal reconfiguration of fault-tolerant real-time tasks. **In** : Workshop on Discrete Event Systems, WODES'10. Berlin, Allemagne: IFAC, 2010, pp. 366–373.

[65] Vahidi Arash, Fabian Martin, Lennartson Bengt. Efficient supervisory synthesis of large systems. Control Engineering Practice, vol. 14, #10, 2006, pp. 1157–1167.

[66] Aagaard M.D., Jones R.B., Seger C.-J.H. Formal verification using parametric representations of boolean constraints. **In** : Design Automation Conference, 1999. Proceedings. 36th. 1999, pp. 402–407.

[67] Bulach S., Brauchle A., Pfleiderer H.-J., et al. Design and implementation of discrete event control systems: a petri net based hardware approach. Discrete Event Dynamic Systems: Theory and Applications, , #12, 2002, pp. 287–309.

[68] Asarin Eugene, Maler Oded, Pnueli Amir. Symbolic controller synthesis for discrete and timed systems. **In** : Hybrid Systems. 1994, pp. 1–20.

[69] Dumitrescu E. Ren M. Automatic error correction based on discrete controller synthesis. **In** : The 4th International Federation of Automatic Control Conference on Management and Control of Production and Logistics, IFAC MCPL'07. 2007.

[70] Maraninchi F. Rémond Y. Mode-automata: a new domain-specific construct for the development of safe critical systems. Science of Computer Programming, vol. 46, #3, 2003, pp. 219–254.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2011ISAL0071/these.pdf
© [M. Ren], [2011], INSA de Lyon, tous droits réservés