SYNTHESIS OF REACTIVE PROGRAMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Anuchit Anuchitanukul
September 1995

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Zohar Manna
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

John C. Mitchell

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Richard Waldinger

Approved for the University Committee on Graduate Studies:

Abstract

We study various problems of synthesizing reactive programs. A *reactive* program is a program whose behaviors are not merely functional relationships between inputs and outputs, but sequences of actions as well as interactions between the program and its environment. The goal of program synthesis in general is to find an *implementation* of a program such that the behaviors of the implementation satisfy a given *specification.*

The reactive behaviors that we study are ω-regular infinite sequences and regular finite sequences. The domain of the implementation is (finite) *transition systems* for closed system synthesis, and *transition system modules* for open system synthesis. We consider various solutions, e.g. *basic*, *maximal*, *modular* and *exact*, for any particular subclasses of the implementation language and investigate how characteristics of the program such as fairness, number of processes and composition operations, affect the synthesis algorithm. In addition to the automata-theoretic algorithms, we give a synthesis algorithm which synthesizes a program directly from the *linear-time temporal logic* ETL.

Acknowledgements

I cannot find any word that would be enough to express my gratitude toward my advisor Zohar Manna. In the past few years, he has always encouraged me and believed in me, even at times when I felt I did no longer believe in myself. Without his patience, guidance and support throughout these years, I would not have finished this thesis.

I would like to thank Richard Waldinger for introducing me to the area of program synthesis. He initiated my interest in research and influenced my early research in program synthesis.

I am particularly grateful to my friends Howard Wong-Toi and Arjun Kapur. In addition to being a dear and very supportive friend, Howard was also my mentor and an invaluable source of information and advice. He seems to know everything! For Arjun, even with all his busy schedules, he is always ready to help. He is always willing to listen to my new ideas and give me fruitful discussion.

To all my friends and colleagues, Henny Sipma, Liz Wolf, Luca de Alfaro, Nikolaj Bjorner and Tomás Uribe, thank you for your help and suggestions.

Above all, I would like to thank my parents who gave me the strength that carried me through the doctoral program.

Anuchit Anuchitanukul
Stanford, California
September 18, 1995

Contents

List of Figures

Chapter 1

Synthesis Problems

In this chapter, we will describe what the synthesis problems are and give the motivation why we are interested in them. The synthesis problems in this chapter are defined abstractly and the definitions are independent of the choices of the specification and implementation languages.

To study any synthesis problem, we must make three choices: $Comp$, L_{spec} and L_{imp}.

1. $Comp$ is the underlying domain of computations. In other words, it is the domain of the behaviors in which we are interested. Ranging from linear models to branching models, $Comp$ can be simply the set of traces (finite and/or infinite sequences from some alphabet), the set of timed traces, or the set of computation trees.

2. L_{spec} is the specification language. We may, for example, study a synthesis problem from a specification given as a formula in a temporal logic, or from a specification given as an automaton. Given the semantics, each specification can be mapped into a subset of $Comp$ which is the set of all computations satisfying the specification. We denote such mapping by $[\![\cdot]\!]_{spec} : L_{spec} \to 2^{Comp}$.

3. L_{imp} is the implementation language with a semantics $[\![\cdot]\!]_{imp} : L_{imp} \to 2^{Comp}$.

1.1 Basic, Maximal, Exact and Weak Synthesis

The simplest form of synthesis problems is the *basic* synthesis problem. Given a specification, the goal is to synthesize or to constructively find an implementation such that all computations of the implementation satisfy the specification.

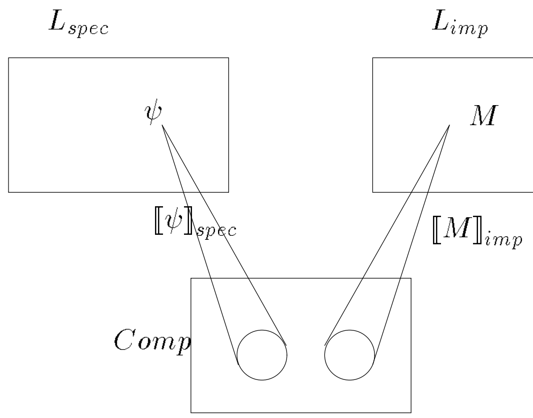

Figure 1.1: Specification and implementation languages

Definition 1.1.1 $\mathsf{SYNTH}(\psi)$ (Basic Synthesis)

Input: a specification $\psi \in L_{spec}$

Goal: find an implementation $M \in L_{imp}$ such that

$$[\![M]\!]_{imp} \subseteq [\![\psi]\!]_{spec}$$

When such M exists, we say that M is a *(basic) solution* of $\mathsf{SYNTH}(\psi)$ and write $M \models \mathsf{SYNTH}(\psi)$.

A solution of the basic synthesis problem may not be *maximal*. In other words, there may be other solutions which have a larger set of behaviors. The best analogy is the problem of implementing a program that computes square roots of a real number. An implementation which prints out only the positive square root definitely satisfies the specification. However, it is not the maximal solution because there is another implementation that prints out both the positive and negative roots.

Definition 1.1.2 $\mathsf{M\!-\!SYNTH}(\psi)$ (Maximal Synthesis)

Input: $\psi \in L_{spec}$

Goal: find $M \in L_{imp}$ such that

$$[\![M]\!]_{imp} \subseteq [\![\psi]\!]_{spec} \qquad \text{and}$$

there is no other $M' \in L_{imp}$ such that

$$[\![M]\!]_{imp} \subset [\![M']\!]_{imp} \subseteq [\![\psi]\!]_{spec}$$

Depending on the choices of L_{spec} and L_{imp}, there may not be a maximal solution to some specifications, even though those specifications have a basic solution. In addition, the maximal solution may not be unique.

In some cases, we may require that the computations of a solution must not only satisfy the specification but all possible computations that satisfy the specification must be represented. In other words, they must be *exactly* the set of all computations of the solution.

Definition 1.1.3 $\mathsf{E}-\mathsf{SYNTH}(\psi)$ (Exact Synthesis)

Input: $\psi \in L_{spec}$
Goal: find $M \in L_{imp}$ such that
$$[\![M]\!]_{imp} = [\![\psi]\!]_{spec}$$

In general, there may not even be a basic solution to a specification. Therefore, the best we can achieve is to find an implementation such that some of its computations satisfy the specification. Later, we will discuss the synthesis problems of open systems where a computation can be considered as a game between the environment and the module we want to synthesize. It may not always be the case that there is an implementation of the module that always wins, but there may be an implementation of the module that has a chance of winning a game.

Definition 1.1.4 $\mathsf{W}-\mathsf{SYNTH}(\psi)$ (Weak Synthesis)

Input: $\psi \in L_{spec}$
Goal: find $M \in L_{imp}$ such that
$$[\![M]\!]_{imp} \cap [\![\psi]\!]_{spec} \neq \emptyset$$

1.2 Modular Synthesis and Open Systems

Sometimes the objective is not to synthesize the entire program, but a module. In other words, we expect to compose the result with other modules. To study such synthesis problems, we have to define what it means to compose two modules together.

Let $|| : L_{imp} \times L_{imp} \rightarrow L_{imp}$ be a composition operation which takes two programs (modules) and composes them into another program. We can define $||$ to be a serial composition, or a parallel composition. It could also be asynchronous or synchronous, and may involve sharing some global variables and hiding some local variables. Since there is no real difference between a program and a module, we will use the word "program" and "module" interchangeably.

With the composition operation, we can define the *modular* synthesis problem. This problem is also known as the control problem. Given a specification and a module(s), we would like to find another module such that when composed with the given module(s), their combined computations satisfy the specification.

Definition 1.2.1 Mod$-$SYNTH(ψ, M_E)

Input: $\psi \in L_{spec}$ and $M_E \in L_{imp}$

Goal: find $M_S \in L_{imp}$ such that

$$[\![M_E || M_S]\!]_{imp} \subseteq [\![\psi]\!]_{spec}$$

When we say a computation of a program, we usually mean the behavior of the program alone without any outside interference, i.e., the program is considered as a *closed* system. On the contrary, considered as an *open* system, a program runs in an environment which consists of many other programs or modules, and a computation of the program in this semantics must reflect the influence from the environment. To consider the synthesis problems of open systems uniformly, we define the *open system semantics* denoted by $O[\![\cdot]\!]_{imp} : L_{imp} \rightarrow 2^{Comp}$, from the base semantics $[\![\cdot]\!]_{imp}$.

Definition 1.2.2 $O[\![\cdot]\!]_{imp}$

$$O[\![M]\!]_{imp} = \bigcup_{k\geq 0,\ M_1,\ldots,M_k \in L_{imp}} [\![M || M_1 || \ldots || M_k]\!]_{imp}$$

Since we assume that $||$ is closed and has a $||$ identity, the definition can be simplified to:

$$O[\![M]\!]_{imp} = \bigcup_{M' \in L_{imp}} [\![M || M']\!]_{imp}$$

Therefore, in addition to the synthesis problems under the original semantics $[\![\cdot]\!]_{imp}$, we can also consider the synthesis problems under $O[\![\cdot]\!]_{imp}$ and refer to them simply as *open system* synthesis problems.

For example, the basic open system synthesis problem of ψ corresponds to the *strong realizability problem*. A solution to the basic open system synthesis problem is an implementation which when composed with or put in any environment, will guarantee that the overall computation satisfies the specification ψ. If we unfold the definitions, the problem itself can be expressed as:

$$(\exists M \in L_{imp}) \;\; O[\![M]\!]_{imp} \subseteq [\![\psi]\!]_{spec}$$

or equivalently

$$(\exists M_S \in L_{imp})(\forall M_E \in L_{imp}) \;\; [\![M_S || M_E]\!]_{imp} \subseteq [\![\psi]\!]_{spec}$$

It leads us to an interesting problem which can be expressed as the dual of the above expression:

$$(\forall M_E \in L_{imp})(\exists M_S \in L_{imp}) \;\; [\![M_S || M_E]\!]_{imp} \subseteq [\![\psi]\!]_{spec}$$

In other words, it is the problem of checking whether the modular synthesis problem of ψ and M_E always has a solution for any M_E. This is clearly weaker than strong realizability (i.e. the basic open system synthesis). For strong realizability, we need to find an implementation that works universally in any environment, but in this case, we only have to come up with an implement for every environment, after knowing how the environment behaves.

Definition 1.2.3 $\forall\exists$−$\mathsf{SYNTH}(\psi)$

Input: $\psi \in L_{spec}$

Goal: check whether for any environment $M_E \in L_{imp}$,
there exists an implementation $M_S \in L_{imp}$ such that

$$[\![M_S || M_E]\!]_{imp} \subseteq [\![\psi]\!]_{spec}$$

In the same way, the weak open system synthesis problem of ψ can be expressed as:

$$(\exists M_S \in L_{imp})(\forall M_E \in L_{imp}) \;\; [\![M_S || M_E]\!]_{imp} \cap [\![\psi]\!]_{spec} \neq \emptyset$$

and its dual

$$(\forall M_E \in L_{imp})(\exists M_S \in L_{imp}) \;\; [\![M_S || M_E]\!]_{imp} \cap [\![\psi]\!]_{spec} \neq \emptyset$$

is sometimes called the *weak realizability problem.*

Definition 1.2.4 $\forall\exists$−W−$\mathsf{SYNTH}(\psi)$

Input: $\psi \in L_{spec}$

Goal: check whether for any environment $M_E \in L_{imp}$,
there exists an implementation $M_S \in L_{imp}$ such that

$$[\![M_S || M_E]\!]_{imp} \cap [\![\psi]\!]_{spec} \neq \emptyset$$

With the open system semantics, we can now define the synthesis problem of modules under an assumption of the environment, or $\mathsf{ES\!-\!SYNTH}(\cdot,\cdot)$. Unlike the modular synthesis problem, we do not have a concrete implementation of the environment beforehand and the inputs to this problem are an environment assumption and a specification. The environment assumption restricts the environment to those whose behaviors conform with the assumption. Therefore, the synthesized module is required to satisfy the specification only when the environment satisfies the assumption.

First, for any $\psi_E \in L_{spec}$, define $L(\psi_E)$ to be:

$$L(\psi_E) = \{M_E \in L_{imp} \mid O[\![M_E]\!]_{imp} \subseteq [\![\psi_E]\!]_{spec}\}.$$

i.e., the set of all M_E which satisfies ψ_E in the open system semantics, or equivalently, the set of all solutions of the basic open system synthesis problem of ψ_E.

Definition 1.2.5 $\mathsf{ES\!-\!SYNTH}(\psi_E, \psi_S)$

Input: $\psi_E, \psi_S \in L_{spec}$

Goal: find $M_S \in L_{imp}$ such that
for any $M_E \in L(\psi_E)$,
$[\![M_S || M_E]\!]_{imp} \subseteq [\![\psi_S]\!]_{spec}$.

The $\mathsf{ES\!-\!SYNTH}(\cdot,\cdot)$ synthesis problem is, in some sense, more general than the (basic) open system synthesis problem. It also corresponds more closely to the usual modular development of software. Each team responsible for developing a module is provided with both the specification of the module and the specification of the environment or the interface under which the module is expected to run.

1.3 Basic Relations Between Synthesis Problems

We can see from the definition that some synthesis problems are stronger than others.

Proposition 1.3.1 *The following sentences are true:*

1. *If* $M \models \mathsf{SYNTH}(\psi)$ *then* $M \models \mathsf{W\!-\!SYNTH}(\psi)$.
2. *If* $M \models \mathsf{M\!-\!SYNTH}(\psi)$ *then* $M \models \mathsf{SYNTH}(\psi)$.
3. *If* $M \models \mathsf{E\!-\!SYNTH}(\psi)$ *then* $M \models \mathsf{M\!-\!SYNTH}(\psi)$.

The proposition above applies to the synthesis problems under both the original and the derived open system semantics.

Suppose that for any $\psi \in L_{spec}$, there exists $\neg\psi \in L_{spec}$ such that

$$[\![\neg\psi]\!]_{spec} = Comp - [\![\psi]\!]_{spec}.$$

Proposition 1.3.2 *The following sentences are true:*

1. $\forall\exists$−$\mathsf{SYNTH}(\psi)$ *(under* $[\![\cdot]\!]_{imp}$*) iff* W−$\mathsf{SYNTH}(\neg\psi)$ *under* $O[\![\cdot]\!]_{imp}$ *has no solution.*
2. $\forall\exists$−W−$\mathsf{SYNTH}(\psi)$ *(under* $[\![\cdot]\!]_{imp}$*) iff* $\mathsf{SYNTH}(\neg\psi)$ *under* $O[\![\cdot]\!]_{imp}$ *has no solution.*

Proposition 1.3.3 *If the maximal open system synthesis problem of* ψ_E *has a unique solution* M_E*, then*

$$M_S \models \mathsf{ES}\text{−}\mathsf{SYNTH}(\psi_E, \psi_S) \quad \text{iff} \quad M_S \models \mathsf{Mod}\text{−}\mathsf{SYNTH}(M_E, \psi_S).$$

Suppose there exists a specification $\psi_E \to \psi_S \in L_{spec}$ such that

$$[\![\psi_E \to \psi_S]\!]_{spec} = [\![\neg\psi_E]\!]_{spec} \cup [\![\psi_S]\!]_{spec}.$$

Then, clearly, we can relate $\mathsf{SYNTH}(\cdot)$ with ES−$\mathsf{SYNTH}(\cdot,\cdot)$.

Proposition 1.3.4 *If* M *is a solution to the basic open system synthesis of* $\psi_E \to \psi_S$*, then* M *is also a solution to the* ES−$\mathsf{SYNTH}(\psi_E, \psi_S)$ *problem.*

Note that the converse is not always true.

An implementation language L_{imp} has a *universal* module M_U iff for any $M \in L_{imp}$, $[\![M || M_U]\!]_{imp} = O[\![M]\!]_{imp}$.

Proposition 1.3.5 *If* L_{imp} *has a universal module* M_U*, then the following sentences hold:*

1. $M \models \mathsf{Mod}$−$\mathsf{SYNTH}(\psi, M_U)$ *(under* $[\![\cdot]\!]_{imp}$*) iff* $M \models \mathsf{SYNTH}(\psi)$ *under* $O[\![\cdot]\!]_{imp}$*.*
2. $\mathsf{SYNTH}(\psi)$ *under* $O[\![\cdot]\!]_{imp}$ *has no solution iff* $M_U \models \mathsf{W}$−$\mathsf{SYNTH}(\neg\psi)$ *under* $O[\![\cdot]\!]_{imp}$*.*

Chapter 2

Closed System Synthesis

This chapter describes the synthesis methods for closed system synthesis problems.

A closed system is a system whose behaviors are generated purely by the system itself without any external influence. The implementation language which we choose to represent a closed system here is the *fair transition system*. The behavior of a fair transition system is a finite or infinite sequence of characters generated during a fair execution of the transition system.

The specification language in this case is a combination of ω-*regular* and *regular* languages, for specifying infinite and finite behaviors.

In this chapter, we first introduce the fair transition systems and the smaller classes with the restrictions on determinism, τ-determinism and the number of processes. We then show the relations between these classes with different restrictions. At the end, we conclude that synthesis problems of closed systems do not only coincide but also have a simple implementation in some special classes.

The different restrictions on these classes mean that an implementation in one class can be considered "simpler" than another implementation in another class. For example, the fewer the number of processes an implementation requires, the simpler it is. Therefore, the goal of the synthesis is not only to find a correct implementation, but also to find the simplest one.

2.1 Fair Transition Systems

2.1.1 Definitions

A *fair transition system* (TS) over a finite alphabet Σ, $\mathcal{A} = \langle \Sigma, Q, Q_0, \mathcal{T}, J, C \rangle$, consists of a finite set of *states* Q, a set of initial states $Q_0 \subseteq Q$, a finite set of transitions $\mathcal{T}$, a set of weakly fair transitions $J \subseteq \mathcal{T}$, and a set of strongly fair transitions $C \subseteq \mathcal{T}$.

Any transition $\tau \in \mathcal{T}$ is a function $\tau : Q \to 2^{\Sigma \times Q}$. In other word, a transition causes the transition system to move nondeterministically from one state to another state, and at the same time, print out a character from the alphabet Σ. We say that a transition τ is *enabled* at a state q iff $\tau(q) \neq \emptyset$. Let $Enabled(q)$ denote the set of transitions which are enabled at q.

A *weakly fair transition system* (WTS) is a fair transition system in which $C = \emptyset$. Similarly, a *strongly fair transistion system* (STS) is a fair transition system in which $J = \emptyset$.

A fair transition system is called *deterministic*, iff $|Q_0| = 1$ and for any $q \in Q$ and any $\sigma \in \Sigma$, there is at most one $\tau \in \mathcal{T}$ and $q' \in Q$ such that $\langle \sigma, q' \rangle \in \tau(q)$. A fair transition system is called *τ-deterministic* iff $|Q_0| = 1$ and for any $q \in Q$, any $\sigma \in \Sigma$ and any $\tau \in \mathcal{T}$, there is at most one q' such that $\langle \sigma, q' \rangle \in \tau(q)$. Let DTS (DWTS, DSTS) stand for deterministic (weakly, strongly, resp.) fair transition system. Similarly, we write τDTS (τDWTS, τDSTS) for τ-deterministic (weakly, strongly, resp.) fair transition system.

A fair transition system can generate both finite and infinite words. A finite *run* of a TS $\mathcal{A}$, which produces a finite word $w = \sigma_0\sigma_1 \ldots \sigma_n \in \Sigma^*$, consists of a sequence of states, $q_0 q_1 \ldots q_{n+1} \in Q^*$ and a sequence of transitions, $\tau_0\tau_1 \ldots \tau_n \in \mathcal{T}^*$, such that, $q_0 \in Q_0$, for every $0 \leq i \leq n$, $\langle \sigma_i, q_{i+1} \rangle \in \tau_i(q_i)$, and $Enabled(q_{n+1}) = \emptyset$. The last condition requires that the last state must be the terminal state, i.e., no transition remains enabled at the end of the computation.

Similarly, an infinite run of $\mathcal{A}$ produces an infinite word $w = \sigma_0\sigma_1 \ldots \in \Sigma^\omega$ and consists of a sequence of states $q_0 q_1 \ldots \in Q^\omega$ and a sequence of transitions $\tau_0\tau_1 \ldots \in \mathcal{T}^\omega$. In addition to the requirements that $q_0 \in Q_0$ and for every $0 \leq i$, $\langle \sigma_i, q_{i+1} \rangle \in \tau_i(q_i)$, we also require that an infinite run satisfies *fairness* conditions, that is,

- for all $\tau \in J$, if there is some j such that τ is enabled at q_i for all $i > j$, then $\tau_k = \tau$ for some $k > j$.

- for all $\tau \in C$, if τ is enabled infinitely often, τ is taken infinitely often.

For convenience, we will say that a run is *fair* iff either it is finite, or, it is infinite and satisfies the fairness conditions above.

A language $\mathcal{L}_{\mathcal{A}}$ generated by a TS $\mathcal{A}$, consists of all the finite and infinite words produced by some fair run of $\mathcal{A}$. Two fair transition systems are *equivalent* iff they generate the same language. For any TS $\mathcal{A}$, we will write $\mathcal{L}^*_{\mathcal{A}}$ [$\mathcal{L}^\omega_{\mathcal{A}}$] to denote the set of finite [resp. infinite] words produced by some fair run of $\mathcal{A}$.

2.1.2 Fairness

The following construction transforms a TS into an equivalent WTS, and as a special case, a DTS (which includes DSTS) into a τDWTS. We will show later that DWTS is less expressive than DTS and that is why we can only find an equivalent τDWTS for any DTS.

Theorem 2.1.1 *For any TS* $\mathcal{A} = \langle \Sigma, Q, Q_0, \mathcal{T}, J, C \cup \{\hat{\tau}\}\rangle$, *there is an equivalent TS* $\mathcal{A}' = \langle \Sigma, Q \cup (Q \times 2^{J \cup C}), Q_0, \mathcal{T}', J' \cup \{\hat{\tau}'\}, C'\rangle$ *such that* $|\mathcal{T}| = |\mathcal{T}'|$, $|J| = |J'|$ *and* $|C| = |C'|$.

Construction: For every transition $\tau \in \mathcal{T}$, there is a corresponding transition $\tau' \in \mathcal{T}'$. The sets J' and C' are the corresponding copy of J and C, respectively. Each τ' is defined as follows:

- For any $q \in Q$,
  - for the transition $\hat{\tau}'$, if $\hat{\tau} \in Enabled(q)$, then $\hat{\tau}'(q) = \hat{\tau}(q)$; otherwise,
  $$\hat{\tau}'(q) = \{\langle \sigma, \langle q', \emptyset\rangle\rangle \mid \tau \in \mathcal{T}, \langle \sigma, q'\rangle \in \tau(q)\}.$$
  - for other transitions τ', $\tau'(q) = \tau(q)$.
- For any $\langle q, S\rangle \in Q \times 2^{J \cup C}$ and for any $\tau \in \mathcal{T}$,
  - if $\hat{\tau}, \tau \in Enabled(q)$ and $\tau \in J \cup C$ and $\tau \notin S$,
  $$\tau'(\langle q, S\rangle) = \{\langle \sigma, \langle q', S \cup \{\tau\}\rangle\rangle \mid \langle \sigma, q'\rangle \in \hat{\tau}(q)\}.$$
  - otherwise,
  $$\tau'(\langle q, S\rangle) = \{\langle \sigma, \langle q', S - \{\tau\}\rangle\rangle \mid \langle \sigma, q'\rangle \in \tau(q)\}.$$

Correctness: The case of finite words is straightforward. It is easy to see that the construction does not lose or introduce any finite words.

For the case of infinite words, we first show that any word generated by a fair run r of $\mathcal{A}$ is generated by a fair run of $\mathcal{A}'$. If $\hat{\tau}$ is taken infinitely often in r then, the same r would be a fair run of $\mathcal{A}'$. On the other hand, if $\hat{\tau}$ is enabled only finitely many times, then map the portion of r after the last time $\hat{\tau}$ is enabled, to the states in $Q \times \emptyset$. The mapped image of r is a fair run of $\mathcal{A}'$.

For the other direction, consider an infinite fair run r' of $\mathcal{A}'$. If r' remains forever in Q, then $\hat{\tau}$ is taken infinitely often and it is clear that r' is also a fair run of $\mathcal{A}$. Now, consider the states $\langle q, S\rangle \in Q \times 2^{J\cup C}$. It is clear that $\mathit{Enabled}(\langle q, S\rangle) = \mathit{Enabled}(q)$. Therefore, the only two cases when the image of r' in $\mathcal{A}$ could be a non-fair run of $\mathcal{A}$ are:

- $\hat{\tau}$ is taken under the name of τ infinitely often and τ is only taken finitely many times, or
- $\hat{\tau}$ is enabled infinitely often but only taken finitely many times.

However, the use of the set S excludes both cases. ∎

The above theorem implies that any fair transition system can be transformed into a weakly fair transition system. It is easy to extend this construction to eliminate all strongly fair transitions in a single transformation with $|J \cup C| \times 2^{|J\cup C|}$ increase in the number of nodes.

The following is the reverse of the previous construction, from a TS into an equivalent STS, and as a special case, a DTS into a DSTS.

Theorem 2.1.2 *For any TS $\mathcal{A} = \langle \Sigma, Q, Q_0, \mathcal{T}, J \cup \{\hat{\tau}\}, C\rangle$, there is an equivalent TS $\mathcal{A}' = \langle \Sigma, (Q \times 2^{J\cup C}), \langle Q_0, \emptyset\rangle, \mathcal{T}', J', C' \cup \{\hat{\tau}'\}\rangle$ such that $|\mathcal{T}| = |\mathcal{T}'|$, $|J| = |J'|$ and $|C| = |C'|$.*

Construction: Let Q_e be the subset of Q where $\hat{\tau}$ is enabled.

- For the transition $\hat{\tau}'$,

$$\begin{aligned}\hat{\tau}'(\langle q, S\rangle) = \quad & \{\langle \sigma, \langle q', S\rangle\rangle \mid \langle \sigma, q'\rangle \in \hat{\tau}(q)\} \cup \\ & \{\langle \sigma, \langle q', S\cup\{\tau\}\rangle\rangle \mid \text{for any } \tau \in (J \cup C) - S, \\ & \qquad q \in Q_e \ \wedge\ q' \notin Q_e \ \wedge\ \langle \sigma, q'\rangle \in \tau(q)\}.\end{aligned}$$

- For other transition τ',
  - if $\tau \in S$,
  $$\tau'(\langle q, S\rangle) = \{\langle \sigma, \langle q', S - \{\tau\}\rangle\rangle \mid \langle \sigma, q'\rangle \in \tau(q)\}.$$
  - otherwise,
  $$\tau'(\langle q, S\rangle) = \{\langle \sigma, \langle q', S\rangle\rangle \mid \langle \sigma, q'\rangle \in \tau(q) \text{ and if } q \in Q_e \text{ then } q' \in Q_e\}.$$

Correctness: If a word is generated by a fair run r of $\mathcal{A}$, then the word must be generated by the image of r on $\mathcal{A}'$. The only cases when the image of r on $\mathcal{A}'$ could be unfair are when:

- τ is taken infinitely often in r, but in the image of r, $\hat{\tau}'$ replaces τ infinitely often and τ' is taken only finitely often,
- $\hat{\tau}'$ is enabled infinitely often without being taken.

The first case is ruled out because the use of the set S guarantees that if $\hat{\tau}'$ replaces τ infinitely often then τ' must be taken infinitely often. For the second case, consider the path through the states $\langle q, S\rangle$ where $S \neq \emptyset$ without taking $\hat{\tau}'$. Such path must eventually empty the set S and move to some states $\langle q', \emptyset\rangle$. The only way the path could visit $\langle q, S\rangle$, $S \neq \emptyset$ again, is by taking $\hat{\tau}'$. On the other hand, if the image of r visits only the states $\langle q, \emptyset\rangle$ infinitely often without taking $\hat{\tau}'$, then $\hat{\tau}$ is enabled continuously in r without being taken. This contradicts the assumption that r is fair.

For the other direction, suppose r' is a fair run of $\mathcal{A}'$. We show that if the image of r' in $\mathcal{A}$ is not fair, then r' is not fair. The only case we have to consider is when $\hat{\tau}$ is continuously enabled in the image of r' but is not taken. It is clear that $\hat{\tau}'$ must be enabled infinitely often and is not taken also. ∎

2.1.3 Degree of Parallelism

Definition 2.1.1 Degree of Parallelism

For any TS $\mathcal{A} = \langle \Sigma, Q, Q_0, \mathcal{T}, J, C\rangle$, the *degree of parallelism* of $\mathcal{A}$, denoted by $Par(\mathcal{A})$, is the number of transitions in $\mathcal{A}$, *i.e.* $Par(\mathcal{A}) = |\mathcal{T}|$.

We denote the class of TS with the degree of parallelism of n by nTS. This notation applies to other subclass as well. For example, 2τDWTS and 2τDSTS denote such class of τWTS and τSTS with the degree of parallelism of 2, respectively.

Definition 2.1.2 Minimal Degree of Parallelism

A TS $\mathcal{A}$ has the *minimal* degree of parallelism of n iff for any equivalent TS system $\mathcal{B}$, $Par(B) \geq n$.

It is easy to see that the following property holds.

Theorem 2.1.3 *If a TS $\mathcal{A} = \langle \Sigma, Q, Q_0, \mathcal{T}, J, C \rangle$, has a minimal degree of parallelism of n, then $|J \cup C| \geq n-1$.*

Proof: Suppose $|J \cup C| < |\mathcal{T}| - 1$ in a TS $\mathcal{A}$. It means there are at least two transitions $\tau_1, \tau_2 \in \mathcal{T} - (J \cup C)$. If that is the case, then we can construct another TS which has the exact same structure as $\mathcal{A}$ except that the two transitions τ_1 and τ_2 are combined into one transition. The new TS is equivalent to the original TS $\mathcal{A}$. ∎

Let $\vec{L}$ be the limit of the prefix of L, $lim(Pref(L))$ (or the safety closure of L). For any word w, let $Inf(w)$ be the set of states which are visited infinitely often by w.

Definition 2.1.3 Dynamic Degree of Parallelism

The *dynamic degree of parallelism* of a TS $\mathcal{A}$ with respect to a word $w \in \vec{L}_{\mathcal{A}}$ is the number of transitions that are enabled at some states visited infinitely often by w, *i.e.*

$$Par_{dyn}(\mathcal{A}, w) = |\bigcup_{q \in Inf(w)} Enabled(q)|.$$

The intended meaning of dynamic degree of parallelism is that it is a measure of how many serial processes are needed to carry out a computation of a transition system in certain "points" in the computation of the system. A word $w \in \vec{L}_{\mathcal{A}}$ corresponds to a loop in a TS. The dynamic degree of parallelism is thus the number of transitions enabled in the loop in a TS. It is more meaningful to use the number of transitions enabled in a loop than the number of transitions enabled at a state because we may artificially reduce the number of enabled transitions at one state by shifting and probably increasing the number of transitions at some subsequent states.

Definition 2.1.4 Minimal Dynamic Degree of Parallelism

A TS $\mathcal{A}$ has the *minimal* dynamic degree of parallelism iff for any word $w \in \vec{L}_{\mathcal{A}}$ and for any equivalent TS, $Par_{dyn}(\mathcal{B}, w) \geq Par_{dyn}(\mathcal{A}, w)$.

2.1.4 Infinite Behaviors

At this point, we will turn our attention to infinite behaviors of fair transition systems. We will show that any ω-regular language is represented by a 2τDWTS or a 2τDSTS. In other words, it means that we only need 2 deterministic programs running in parallel in order to generate any ω-regular language.

Theorem 2.1.4 *For any ω-regular language $\mathcal{L}$, there exists a 2τDWTS [or 2τDSTS] $\mathcal{A}$ such that $\mathcal{L}_{\mathcal{A}} = \mathcal{L}^{\omega}_{\mathcal{A}} = \mathcal{L}$.*

Construction: First, consider a simpler polynomial translation from a nondeterministic Büchi automaton (NBA) into a 2WTS/2STS. Given a NBA $\mathcal{A} = \langle Q, Q_0, \delta, F\rangle$, construct a 2STS $\mathcal{B} = \langle Q \times \{0,1\}, Q_0 \times \{0\}, \{\tau_0, \tau_1\}, \emptyset, \{\tau_1\}\rangle$ as follows:

1. First, we may assume that for any state $q \in Q$, there is a non-ϵ path from q to some state in F (otherwise, we can easily come up with an algorithm to eliminate all states q which violate the assumption).

2. Consider $\mathcal{A}$ as a labeled graph. Search the graph in the reverse direction from the final states F and generate a labeled DAG $\mathcal{G} \subseteq \mathcal{A}$ such that a state q is in $\mathcal{G}$ iff there is a path from q to some final states in F, and the path itself must be in $\mathcal{G}$.

3. For any $q \in Q$,

$$\tau_0(\langle q, 0\rangle) = \{\langle \sigma, \langle q', 0\rangle\rangle \mid q' \in \delta(q,\sigma) \ \wedge \ q' \notin F\}.$$

$$\tau_0(\langle q, 1\rangle) = \{\langle \sigma, \langle q', 1\rangle\rangle \mid q \xrightarrow{\sigma} q' \in \mathcal{G} \ \wedge \ q' \notin F\} \ \cup \ \{\langle \sigma, \langle q', 0\rangle\rangle \mid q \xrightarrow{\sigma} q' \in \mathcal{G} \ \wedge \ q' \in F\}.$$

$$\tau_1(\langle q, 0\rangle) = \{\langle \sigma, \langle q', 0\rangle\rangle \mid q' \in \delta(q,\sigma) \ \wedge \ q' \in F\} \ \cup \ \{\langle \sigma, \langle q', 1\rangle\rangle \mid q \xrightarrow{\sigma} q' \in \mathcal{G} \ \wedge \ q' \notin F\}.$$

Using the same construction, we can construct a 2WTS $\mathcal{B} = \langle Q \times \{0,1\}, Q_0 \times \{0\}, \{\tau_0, \tau_1\}, \{\tau_1\}, \emptyset\rangle$.

The transformation from a NBA $\mathcal{A} = \langle Q, Q_0, \delta, F\rangle$, to a 2τDSTS $\mathcal{B}' = \langle 2^Q \cup (2^Q \times 2^Q), \{Q_0\}, \{\tau_0, \tau_1\}, \emptyset, \{\tau_1\}\rangle$ is similar to the above transformation and is based on the subset construction.

- For any state $S \in 2^Q$,

$$\tau_0(S) = \{\langle \sigma, S' \rangle \mid S' = \{q' \mid q \in S \ \wedge \ q' \in \delta(q,\sigma)\} \ \wedge \ S' \neq \emptyset\}.$$

$$\tau_1(S) = \{\langle \sigma, \langle S', S' \rangle \rangle \mid S' = \{q' \mid q \in S \ \wedge \ q' \in \delta(q,\sigma)\} \ \wedge \ S' \neq \emptyset\}.$$

- For any state $\langle S_1, S_2 \rangle \in 2^Q \times 2^Q$, $\tau_1(\langle S_1, S_2 \rangle) = \emptyset$, and
  - if $S_1 \cap F \neq \emptyset$, then

$$\tau_0(\langle S_1, S_2 \rangle) = \{\langle \sigma, S' \rangle \mid S' = \{q' \mid q \in (S_1 \cap F) \ \wedge \ q' \in \delta(q,\sigma)\} \ \wedge \ S' \neq \emptyset\}.$$

  - else,

$$\tau_0(\langle S_1, S_2 \rangle) = \{\langle \sigma, \langle S', S_1 \cup S_2 \rangle \rangle \mid S' = \{q' \mid q \in S_1 \wedge q' \in \delta(q,\sigma)\} - S_2 \wedge S' \neq \emptyset\}.$$

Note also that $\mathcal{B}$ and $\mathcal{B}'$ generate only infinite words.

Correctness: First, we can show that any word accepted by a run r of $\mathcal{A}$ is generated by a fair run of $\mathcal{B}$. All the states in r can be mapped directly into states in $Q \times \{0\}$. Since all egdes going into a final state in $F \times \{0\}$ belong to τ_1, τ_1 is taken infinitely often. Therefore, the image of the run r is a fair run of $\mathcal{B}$.

For the other direction, consider a fair run r' of $\mathcal{B}$. Suppose r' visits $Q \times \{1\}$ only finitely many times. Then, τ_1 must be taken infinitely often into $F \times \{0\}$; otherwise, r' will not be fair. If r' visits $Q \times \{1\}$ infinitely often, then every time it visits a state in $Q \times \{1\}$, it will eventually visit $F \times \{0\}$ because $\mathcal{G}$ is acyclic and every path in $\mathcal{G}$ leads to a state in F.

For $\mathcal{B}'$, the proof is very similar. Every state in a fair run of $\mathcal{A}$ is mapped to a state in 2^Q except the final states which are mapped to a state in $2^Q \times 2^Q$. The subset subtraction of S_2 in the last part of the definition of τ_0 guarantees that the paths are acyclic. ∎

While every ω-regular language can be generated by such simple classes as 2τDSTS and 2τDWTS, we also know that infinite languages generated by any fair transition system are ω-regular.

Theorem 2.1.5 *For any TS $\mathcal{A}$, $\mathcal{L}^\omega_\mathcal{A}$ is ω-regular.*

Proof: When we ignore the finite words generated by a TS, the translation of a TS into a Streett automaton is straightforward. Given a TS $\mathcal{A} = \langle \Sigma, Q, Q_0, \mathcal{T}, J, C \rangle$, construct a SA $\langle \Sigma, Q \times \mathcal{T}, Q_0 \times \mathcal{T}, \sigma, F \rangle$. The transition relation is simply:

$$\sigma(\langle q, \hat{\tau}\rangle, c) = \{\langle q', \tau\rangle \mid \langle c, q'\rangle \in \tau(q)\}.$$

For the acceptance condition $F = \{(L_\tau, U_\tau\}_{\tau \in J \cup C}$, we have two cases. For each $\tau \in J$, $L_\tau = \emptyset$ and $U_\tau = Q \times \{\tau\} \cup \{q | \tau \notin Enabled(q)\} \times \mathcal{T}$. For each $\tau \in C$, $L_\tau = \{q | \tau \in Enabled(q)\} \times \mathcal{T}$ and $U_\tau = Q \times \{\tau\}$. ■

Theorem 2.1.6 *The class of ω-languages generated by WTS, STS, τDTS, τDSTS, or τDWTS is exactly the class of ω-regular languages.*

Proof: This follows from the two theorems that TS, 2τDSTS and 2τDWTS generate ω-regular languages. Therefore, any class of fair transition systems TS', such that 2τDSTS $\subseteq TS' \subseteq$ TS, must also generate ω-regular languages. ■

Theorem 2.1.7 *The class of ω-languages generated by DWTS is strictly smaller than the class of languages recognized by deterministic Büchi automaton (DBA).*

Proof: The translation from a DWTS into a deterministic Büchi automaton (DBA) is obvious but there is a language, namely $(aa)^*b^\omega$, which is recognized by a DBA but not by any DWTS.

Given a DWTS $\mathcal{A} = \langle \Sigma, Q, Q_0, \mathcal{T}, J, \emptyset\rangle$, we construct a DBA for each τ in J and then construct the product automaton from those $|J|$ DBAs. Each DBA $\langle \Sigma, Q \times \mathcal{T}, Q_0 \times \mathcal{T}, \sigma, F_\tau\rangle$, constructed for each $\tau \in J$, has the same set of states and transition function. The acceptance condition F_τ is simply $Q \times \{\tau\} \cup \{q | \tau \notin Enabled(q)\} \times \mathcal{T}$.

If the language $(aa)^*b^\omega$ were generated by a DWTS, then there must be an unfair run which generates the word a^ω. Since the run is unfair, it means that there is a transition which is continuously enabled from some point onward but has never been taken. We know that the automaton is a DWTS and thus it is deterministic. Therefore, if the transition were taken, it must print out the character b. Since the transition is enabled from some point onward, we know that it could have printed out b at any time, i.e. after printing out any number of a's. This, however, contradicts the assumption that the automaton generates b^ω only after an even number of a's. ■

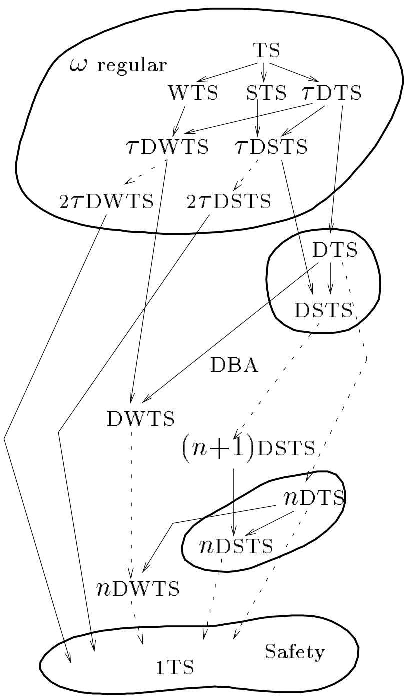

Figure 2.1: TS hierarchy

Theorem 2.1.8 *The class of ω-languages generated by DTS is the same as the class of ω-languages generated by DSTS and properly contains the languages recognized by DBA. However, there are some ω-regular languages which can not be generated by any DTS.*

Proof: Suppose $\Sigma = \{a, b, c\}$. A language in which if a word contains infinitely many b's then it must also contains a c, is generated by a DTS, but not recognized by any DBA. If there were a DBA which accepts the language, then the words of the form $(a+b)^*a^\omega$ which have only finitely many b's, must also be accepted. Therefore, for every word of that form, there must be an accepting state which is visited after the automaton reads some prefix a^{i_1}. For the same reason, there exists another prefix $a^{i_1}ba^{i_2}$ which leads the automaton into another accepting state as well. If we continue extending the prefix in this manner, we can show that the DBA accepts a word $a^{i_1}ba^{i_2}ba^{i_3}b\ldots$ which has infinitely many b's but no c's at all. However, the language is generated by a DTS $\langle \Sigma, \{q_0, q_1, q_2\}, \{q_0\}, \{\tau_0, \tau 1\}, \emptyset, \{\tau_1\}\rangle$

where

$$\begin{aligned}
\tau_0(q_0) &= \{\langle a, q_0\rangle, \langle b, q_1\rangle, \langle c, q_2\rangle\}\\
\tau_0(q_1) &= \{\langle a, q_0\rangle, \langle b, q_1\rangle\}\\
\tau_0(q_2) &= \{\langle a, q_2\rangle, \langle b, q_2\rangle, \langle c, q_2\rangle\}\\
\tau_1(q_1) &= \{\langle c, q_2\rangle\}\\
\tau_1(q_0) &= \tau_1(q_2) = \emptyset.
\end{aligned}$$

On the other hand, a language $(a+b)^*a^\omega$ can not be generated by any DTS. Suppose there were such DTS. Since any finite word $(a+b)^*b$ is a prefix of some (infinite) words generated by the DTS, we know that there is a transition which prints out b and is enabled at any state of the DTS. View the DTS as a graph. There must be a strongly connected component in the graph. The path that passes through every edge (transition) including those with b as the output label is a fair run and contains infinitely many b's, contradicting the assumption that the DTS generates $(a+b)^*a^\omega$.

The construction in theorem 2.1.2 shows that the class of languages generated by DTS and DSTS is the same. The fact that the class of languages recognized by DBA is contained in the class of languages generated by DTS is implied by theorem 2.2.1 in the following section. ∎

Theorem 2.1.9 *For any $n > 0$, the class of ω-languages generated by nDTS is the same as the class of ω-languages generated by nDSTS and is strictly smaller than the class of ω-languages generated by $(n{+}1)$DTS. The class of ω-languages generated by nDWTS is properly contained in the class of ω-languages generated by nDTS.*

Proof: First, the construction in theorem 2.1.2 shows that nDTS and nDWTS can be translated into an nDSTS. Second, the language $(aa)^*b^\omega$ is generated by a 2DSTS but not by any nDWTS, as shown in theorem 2.1.7. Finally, it is easy to show that there are some ω-languages that are generated by $(n+1)$DTS but not by any nDTS. This fact is actually implied by theorem 2.2.1 in the next section.∎

In other words, there are some languages that cannot be generated by any DTS with a degree of parallelism smaller than n. For example, a language $(a^*b)^*c^\omega$ is generated by a DTS with degree of pallelism of at least 3.

2.2 Closed System Synthesis

In this section, we combine what we has developed in this chapter into the solutions to the synthesis problems. We consider two cases depending on whether the specification specifies:

- only the infinite behaviors, or,
- both infinite and finite behaviors.

2.2.1 Only Infinite Behaviors

The solution to weak and simple synthesis problems in the closed system case can be as simple as a TS with a single sequence as its only behavior. In other words, we only need to check for satisfiability or non-emptyness of the specification. Therefore, the more interesting question is whether the maximal or exact solution exists.

When the specification is simply ω-regular, the weak, simple, maximal and exact synthesis problems coincide. That is, by theorem 2.1.4, there is an exact solution to every specification. Moreover, the solution is as simple as a 2τDTS. The construction in theorem gives such implementation which is a 2τDTS with an exponential increase in the number of nodes. Without the τ deterministic restriction, the construction yields a 2TS which has only twice as many nodes as that of the given (Büchi) specification.

For the less expressive class DTS, there may not be an exact solution but we can find a maximal solution. Even though it is not unique, the maximal solution has the minimal degree and dynamic degree of parallelism.

Theorem 2.2.1 *For any ω-regular language $\mathcal{L}$, there is a DTS $\mathcal{A}$ such that:*

1. *$\mathcal{A}$ is the maximal solution, i.e. $\mathcal{L}_{\mathcal{A}} = \mathcal{L}^*_{\mathcal{A}} \subseteq \mathcal{L}$ and there is no other DTS $\mathcal{A}'$ such that $\mathcal{L}_{\mathcal{A}} \subset \mathcal{L}_{\mathcal{A}'} \subseteq \mathcal{L}$.*
2. *$\mathcal{A}$ has the minimal degree of parallelism and minimal dynamic degree of parallelism.*
3. *If $\mathcal{L}$ is recognized by a DTS or DBA, then $\mathcal{L}_{\mathcal{A}} = \mathcal{L}$.*

Construction: For any ω-regular language $\mathcal{L}$, we can find a DSA $\mathcal{A} = \langle \Sigma, Q, \{q_0\}, \delta, F \rangle$ that recognizes $\mathcal{L}$. Suppose $F = \{(L_1, U_1), \ldots, (L_n, U_n)\}$. For each (L_i, U_i), we consider $\mathcal{A}$ as a graph and assign colors, $\{c_0, c_1, \ldots\}$, to the edges with a subroutine call to $\mathsf{coloring}(\mathcal{A}, (L_i, U_i), F, c_0)$.

Subroutine coloring(G,(L_i, U_i),F',c_j):

1. Set the states in U_i apart from the graph G and break $G-U_i$ into strongly connected components, $C_0, C_1, \ldots, C_m$.

2. Consider the graph as a pre-order of $C_0, \ldots, C_m$ and traverse the pre-order in the reverse direction. For each C_k, let $fr(C_k)$ be the states $q \in C_k$ such that there is an edge going from q to some unmarked states outside C_k, and assign the color c_j to all such edges. Then, do the following:

Case 1 $C_k \cap L_i = \emptyset$: If F' is not empty, then pick a (L_k, U_k) from F', and call coloring(C_k, (L_k, U_k), $F' - \{(L_i, U_i)\}$, c_j).

Case 2 $C_k \cap L_i \neq \emptyset$ and $fr(C_k) \neq \emptyset$: break $C_k - fr(C_k)$ into strongly connected components $C'_0, \ldots, C'_{m'}$ and recursively follow the same coloring subroutine for each $C'_{k'}$, but use the next color c_{j+1}.

Case 3 $fr(C_k) = \emptyset$ and $C_k \cap L_i = C_k$ or $|C_k| = 1$: Mark all states in C_k and all incoming edges as "deleted".

Case 4 Otherwise ($fr(C_k) = \emptyset$ and $\emptyset \neq C_k \cap L_i \neq C_k$): break C_k into strongly connected components $C'_0, \ldots, C'_{m'}$ such that either $C_k \subseteq L_i$ or $C_k \cap L_i = \emptyset$. Make $C'_0, \ldots, C'_{m'}$ into a pre-order by marking cyclic edges as "deleted" and recursively apply the algorithm to each $C'_{k'}$.

For each color c_j used, we associate a transition τ_j to the edges with the color and construct a DTS $\mathcal{A}' = \langle \Sigma, Q \times \{1, \ldots, n\}, \{\langle q_0, 1\rangle\}, \mathcal{T}, \emptyset, \mathcal{T}\rangle$ such that for each layer $Q \times \{i\}$:

- we use the coloring from the call to coloring($\mathcal{A}$,(L_i, U_i),F,c_0) to define the transitions in $\mathcal{T}$ on this layer, and,

- the transitions going out from the states $U_i \times \{i\}$ lead to the next layer $Q \times \{i+1\}$.

Correctness: The key idea is based on the fact that the edges going out from a strongly connected component must have a different color from all transitions in the strongly connected component. Otherwise, a computation can remain in the strongly connected component forever and will be an unfair computation of $\mathcal{A}$. Use this argument and induction on the number of levels of strongly connected components within strongly connected components to prove the minimality of degree and dynamic degree of parallelism. ∎

Unlike in the case of DTS, there is no maximal DWTS solution for ω-regular specifications in general. Consider, for example, the case of the ω-language $(aa)^*b^\omega$. Suppose a DWTS $\mathcal{A}$ is a solution. We know that the word a^ω is not generated by any fair or unfair run of $\mathcal{A}$. If a^ω is generated by an unfair run, then it must be the case that there is a transition which generates b and is enabled from a certain point on. But this cannot be the case because $\mathcal{A}$ can generate b only after an even number of b's. Since $\mathcal{A}$ cannot generate a^ω, then there must be a word $(aa)^n b^\omega$ generated by $\mathcal{A}$ such that n is maximal. We can easily construct another DWTS which generates not only every word $\mathcal{A}$ generates but also $(aa)^{n+1}b^\omega$.

2.2.2 Both Infinite and Finite Behaviors

Now, we consider synthesis problems where a specification specifies both infinite ω-regular words and finite regular words. Finite words may represent either terminating behaviors or deadlocks. Therefore, to satisfy the specification, an implementation must generate only good infinite behaviors and must not generate any new deadlocks. In this case, there is still an exact solution for such specification. However, the "minimal" exact solution is no longer in 2τDTS, but in 2TS or 3τDTS.

We can modify the construction in theorem 2.1.4 to give an exact solution in 2TS or 3τDTS by simply adding a transition into a terminating state whenever the finite word the transition system has already printed out satisfies the specification. To show that there is no exact 2τDTS solution, consider the exact synthesis problem of $(a+b)^*a^\omega \; + \; (a+b)^*$. Since every finite word is in the language, any state in a solution of this problem must have an outgoing transition(s) which generates a and b and moves the system into a terminating state. If we remove such transitions and if the solution were 2τDTS, what remains is a deterministic transition system. However, we have shown that $(a+b)^*a^\omega$ is not generated by any DTS. Therefore, we can conclude that there is no exact solution that is a 2τDTS.

Chapter 3

Open System Synthesis

This chapter focuses on open systems. An open system is a system whose behaviors can be affected by outside influence. The model of open systems that we consider as the implementation language is the *transition system module*. Transition system modules can be composed together into one transition system. The main goal of synthesis problems, in short, is to synthesize one of the modules so that the overall behaviors of the composed transition system satisfies certain properties specified by the specification. In detail, the synthesis problems in this chapter vary according to whether some information is given as a part of the inputs to the problems, how the outside influence, which we call the *environment*, behaves, and how the environment's behaviors are characterized.

Transition system modules are very similar to the transition systems defined in the previous chapter. The only difference is that, unlike the transition systems which have a single set of states, a transition system module has two sets of states, global and local states. The global states are shared among the modules which are composed into a transition system. Therefore, the computation of each module (including the one being synthesized), can be affected by other modules (which become the environment to the module being synthesized).

In this chapter, we first define transition system modules and study their properties. Next, a generic basic synthesis algorithm is described. We then show how this synthesis algorithm can be applied to solve various synthesis problems (e.g modular, basic, etc.) with different settings (e.g. synchronous, unobservable variables, etc). At the end, we present some results on the more complicated maximal and $\mathsf{ES{-}SYNTH}(\cdot,\cdot)$ synthesis problems.

3.1 Transition system modules

A *transition system module* (TSM) $\mathcal{M}$ is a tuple $\langle \Sigma, Q_g, Q_l, Q_0, \mathcal{T}, J, C \rangle$. Q_g is a set of *global* states, representing *shared variables* and Q_l is a set of *local* states, visible and accessible only to $\mathcal{M}$. $Q_0 \subseteq Q_g \times Q_l$ is the set of initial states. Each transition τ is now a function $\tau : Q_g \times Q_l \to 2^{\Sigma \times Q_g \times Q_l}$. If $|\mathcal{T}| = 1$, then we say that $\mathcal{M}$ is *serial.* If $|Q_l| = 1$, then we say that $\mathcal{M}$ is *simple.*

Transition system modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$, where each $\mathcal{M}_i = \langle \Sigma_i, Q_g, Q_{li}, Q_{0i}, \mathcal{T}_i, J_i, C_i \rangle$, can be composed into a transition system $\mathcal{M}_1 || \ldots || \mathcal{M}_n = \langle \Sigma, Q, Q_0, \mathcal{T}, J, C \rangle$ such that,

- Σ is $\Sigma_1 \cup \ldots \cup \Sigma_n$.
- Q is $Q_g \times Q_{l1} \times \cdots \times Q_{ln}$.
- Q_0 is the set of all states $\langle q_g, q_1, \ldots, q_n \rangle \in Q$ such that $\langle q_g, q_i \rangle \in Q_{0i}$.
- $\mathcal{T}$ is $\mathcal{T}_1 \cup \ldots \cup \mathcal{T}_n$ and each transition $\tau \in \mathcal{T}_i$ is extended over Q and Σ, *i.e.* ,

$$\langle \sigma, \langle q'_g, q_1, \ldots, q'_i, \ldots, q_n \rangle \rangle \in \tau(\langle q_g, q_1, \ldots, q_i, \ldots, q_n \rangle) \text{ iff } \langle \sigma, q'_g, q'_i \rangle \in \tau(q_g, q_i).$$

- $J = J_1 \cup \ldots \cup J_n$ and $C = C_1 \cup \ldots \cup C_n$.

Consequently, the behaviors of a transition system module depend on its *environment* which may consist of any number of other transition system modules. Moreover, since we always compose TSMs into a TS or simply consider a single TSM as a TS, we use the definitions of runs, fair runs, word generations, etc. which were developed in the previous chapter.

3.1.1 Decomposition

Since TSMs can be composed into a TS, it is only logical to consider the decomposition of a TS into TSMs. Decomposition can be a research topic in itself and we do not attempt to fully study decomposition in detail here. The decomposition considered here is just a simple decomposition of a TS into *serial* TSMs with the requirements that the decomposition preserves equivalence and respects some form of homomorphism.

Definition 3.1.1 Decomposition

A decomposition of a TS $\mathcal{A} = \langle \Sigma, Q, Q_0, \{\tau_1, \ldots, \tau_n\}, J, C \rangle$ into an equivalent TS $\mathcal{M}_1 || \ldots || \mathcal{M}_n$ of serial TSMs $\mathcal{M}_i = \langle \Sigma, Q_g, Q_{li}, Q_{0i}, \{\tau_i'\}, J_i, C_i \rangle$, is a one-to-one function $d : Q \rightarrow Q_g \times Q_{l1} \times \ldots \times Q_{ln}$ such that:

- $J_i = \{\tau_i'\}$ if $\tau_i \in J$, and $C_i = \{\tau_i'\}$ if $\tau_i \in C$.
- For any $\langle \sigma, q' \rangle \in \tau_i(q)$, if $d(q) = \langle q_g, q_1, \ldots, q_n \rangle$ and $d(q') = \langle q_g', q_1', \ldots, q_n' \rangle$, then $\langle \sigma, q_g', q_i' \rangle \in \tau_i'(q_g, q_i)$ and for all $j \neq i$, $q_j = q_j'$.
- For any $\langle \sigma, q_g', q_i' \rangle \in \tau_i'(q_g, q_i)$ and any $q \in Q$, if $d(q) = \langle q_g, q_1, \ldots, q_n \rangle$, then there exists $q' \in Q$ such that $\langle \sigma, q' \rangle \in \tau_i(q)$, and $d(q') = \langle q_g', q_1, \ldots, q_i', \ldots, q_n \rangle$.
- $q \in Q_0$ and $d(q) = \langle q_g, q_1, \ldots, q_n \rangle$ iff for all $0 < i \leq n$, $\langle q_g, q_i \rangle \in Q_{0i}$.

When we decompose a TS into TSMs, one important property we may want to see in TSMs is modular independence. In other words, those TSMs should be loosely coupled and should not have too many shared variables. The size of the set of global states, $|Q_g|$, can be considered as a measure of how many shared variables are used to control the computation of the system and how one module may affect the execution of other modules. Or, from another point of view, $log_2(Q_g)$ is the number of bits needed to represent all possible global states. Therefore, in the process of decomposition, we may want to minimize the size of $|Q_g|$ as much as possible.

Any decomposition mapping d induces an equivalence relation r_d between the states Q of the original TS that map to the same global state:

$$\begin{aligned} r_d(q, q') \ \text{ iff } \quad & \text{for some } q_g, q_1, q_1', \ldots, q_n, q_n', \\ & d(q) = \langle q_g, q_1, \ldots, q_n \rangle \ \text{ and} \\ & d(q') = \langle q_g, q_1', \ldots, q_n' \rangle. \end{aligned}$$

If we want to minimize $|Q_g|$, we have to minimize the number of equivalence classes (of the relation r_d) induced by d.

Finding the best d with the minimal number of equivalence classes of r_d is a difficult task and the complexity is inherently exponential in the size of the input TS. On the other hand, if we only look for a suboptimal d with the *largest* induced relation r_d, we can find such d within polynomial time.

A binary relation r_1 is *larger* than another relation r_2 iff $r_2(q_1, q_2)$ implies $r_1(q_1, q_2)$, for any q_1, q_2. A decomposition d has the largest induced relation r_d iff there is no other decomposition with the induced relation larger than r_d.

Theorem 3.1.1 *From a fair transition system $\mathcal{A} = \langle \Sigma, Q, Q_0, \mathcal{T}, J, C \rangle$, we can find a decomposition d of $\mathcal{A}$ into an equivalent transition system $\mathcal{M}_1 || \ldots || \mathcal{M}_n$ with the largest induced relation r_d.*

Construction: For each transition τ, consider $\mathcal{A}$ as a graph and remove all edges which belong to τ. Each cluster of connected nodes represents a local state of τ. Global states are constructed by grouping compatible states in each cluster into groups.

Formally, for any $q_1, q_2 \in Q$, construct an equivalence relation $Loc_\tau(q_1, q_2)$ such that $Loc_\tau(q_1, q_2)$ iff there is a path from q_1 to q_2, or from q_2 to q_1, without passing through τ. Each equivalence class of Loc_τ is a local state in Q_τ. Compute the relation Glb which is the largest fixed-point of the following relation g:

$$
\begin{aligned}
& g(q_1, q_2) = \\
& \quad g(q_2, q_1) \ \wedge \ (\forall q_3 \in Q) \text{ if } g(q_2, q_3) \text{ then } g(q_1, q_3) \\
& \quad \wedge \ (\forall \tau \in \mathcal{T}) \ \text{ if } Loc_\tau(q_1, q_2) \\
& \qquad\qquad \text{then } (\forall \langle \sigma, q_3 \rangle \in \tau(q_1))(\exists \langle \sigma, q_4 \rangle \in \tau(q_2)) \ Loc_\tau(q_3, q_4) \wedge g(q_3, q_4) \\
& \quad \wedge \ \text{ if } (q_1 \in Q_0) \ \wedge \ (\forall \tau \in \mathcal{T}) \ Loc_\tau(q_1, q_2) \\
& \qquad \text{then } q_2 \in Q_0.
\end{aligned}
$$

Note that the largest fixed-point of this relation may not be unique.

Each equivalence class of Glb is a global state in Q_g whereas each equivalence class of Loc_τ is a local state in Q_τ. That means that the decomposition maps all states q (of Q) in an equivalent class of Glb are mapped to single (new) global state q_g representing the equivalence class of Glb where q belongs. Similarly, for any τ, each q is also mapped to a local state q_τ which represents the equivalence class of Loc_τ which q is in.

Correctness: We can easily show that any equivalence relation r_d induced by any decomposition d must be a fixed point of g. ∎

The above decomposition algorithm can be easily extended to decompose a TSM to another m-equivalent TSM which is a composition of serial TSMs, by considering $\Sigma \times Q_g$ as the output alphabet.

3.1.2 M-equivalence

Similar to the equivalence between transition systems, we have the notion of *m-equivalence* between transition system modules.

Definition 3.1.2 M-equivalence

A TSM $\mathcal{M}_1$ is *m-equivalent* to another TSM $\mathcal{M}_2$ iff for any TSM $\mathcal{M}'$, $\mathcal{L}_{\mathcal{M}_1||\mathcal{M}'} = \mathcal{L}_{\mathcal{M}_2||\mathcal{M}'}$.

Theorem 3.1.2 *M-equivalence is decidable.*

Proof: Given a TSM $\langle \Sigma, Q_g, Q_l, Q_0, \mathcal{T}, J, C \rangle$, we can construct a SA $\langle \Sigma', Q', Q'_0, \delta, F \rangle$ as follows:

- $\Sigma' = Q_g \times (\Sigma \cup \{e\}) \cup \{h\}$,
- $Q' = Q_g \times Q_l \times (\mathcal{T} \cup \{e\}) \cup \{q_h\}$,
- $Q'_0 = Q_0 \times \{e\}$,
- $F = \{(L_\tau, U_\tau)\}_{\tau \in J \cup C} \cup \{(\emptyset, \{q_h\})\}$, and
  - for each $\tau \in J$, $L_\tau = Q'$ and $U_\tau = \{\langle q_g, q_l, t\rangle \in Q' \mid \tau(q_g, q_l) = \emptyset\} \cup Q_g \times Q_l \times \{\tau\}$,
  - for each $\tau \in C$, $L_\tau = \{\langle q_g, q_l, t\rangle \in Q' \mid \tau(q_g, q_l) \neq \emptyset\}$ and $U_\tau = Q_g \times Q_l \times \{\tau\}$.
- For any $\langle q_g, q_l, t\rangle \in Q'$, and any $\langle \bar{q}_g, c\rangle \in \Sigma'$,
  - if $q_g \neq \bar{q}_g$, then $\delta(\langle q_g, q_l, t\rangle, \langle \bar{q}_g, c\rangle) = \emptyset$.
  - if $c \in \Sigma$ (*i.e.* $c \neq e$), then $\delta(\langle q_g, q_l, t\rangle, \langle q_g, c\rangle) = \{\langle q'_g, q'_l, \tau\rangle \mid \langle c, q'_g, q'_l\rangle \in \tau(q_g, q_l)\}$.
  - otherwise, $\delta(\langle q_g, q_l, t\rangle, \langle q_g, e\rangle) = Q_g \times \{q_l\} \times \{e\}$.
- For the new symbol h (halt) and the special state q_h,
  - if $\tau(q_g.q_l) = \emptyset$ for all $\tau \in \mathcal{T}$, then $\delta(\langle q_g, q_l, t\rangle, h) = \{q_f\}$;
    otherwise, $\delta(\langle q_g, q_l, t\rangle, h) = \emptyset$.
  - $\delta(q_h, h) = \{q_h\}$.

We can transform any TSM $\mathcal{M}_1$ and $\mathcal{M}_2$ (with the same alphabet and global states), into SA $\mathcal{A}_1$ and $\mathcal{A}_2$, and then, check if $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent. ∎

The *underlying structure* of a TSM $\mathcal{M} = \langle \Sigma, Q_g, Q_l, Q_0, \mathcal{T}, J, C \rangle$, is a graph/automaton $U(\mathcal{M}) = \langle \Sigma, Q_g, \delta \rangle$ such that

$$\delta(q_g, \sigma) = \{q_g' \mid \langle \sigma, q_g', q_l' \rangle \in \tau(q_g, q_l), \text{ for some } \tau \in \mathcal{T}, \text{ and some accessible } q_l \in Q_l\}.$$

A local state $q_{ln} \in Q_l$ is *accessible* if there exists a sequence,

$$\langle q_{l0}, q_{g0} \rangle \tau_0 \langle q_{g0}', q_{l1}, q_{g1} \rangle \tau_1 \langle q_{g1}', q_{l2}, q_{g2} \rangle \tau_2 \ldots \tau_n \langle q_{g(n-1)}', q_{ln} \rangle,$$

such that for any $i \geq 0$, there is some σ, $\langle \sigma, q_{gi}', q_{l(i+1)} \rangle \in \tau_i(q_{gi}, q_{li})$.

Theorem 3.1.3 *If TSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ are m-equivalent, then $U(\mathcal{M}_1)$ and $U(\mathcal{M}_2)$ are isomorphic.*

Proof: Suppose $U(M_1) = \langle \Sigma, Q_g, \delta_1 \rangle$ and $U(M_2) = \langle \Sigma, Q_g, \delta_2 \rangle$. We only need to show that for arbitrary $q_g, q_g' \in Q_g$, if $q_g' \in \delta_1(q_g)$ then $q_g' \in \delta_2(q_g)$. Since $q_g' \in \delta_1(q_g)$, by definition, there exists a sequence

$$\langle q_{l0}, q_{g0} \rangle \tau_0 \langle q_{g0}', q_{l1}, q_{g1} \rangle \tau_1 \ldots \langle q_{ln}, q_{gn} \rangle \tau_n \langle q_l, q_g \rangle \tau \langle q_l', q_g' \rangle$$

where $q_{l0}, \ldots, q_{ln}, q_l, q_l'$ are local states of M_1. The sequence represents (the projection of) a prefix of a possible run of M_1. Since M_1 and M_2 are equivalent, there must be another sequence

$$\langle p_{l0}, q_{g0} \rangle \tau_0 \langle q_{g0}', p_{l1}, q_{g1} \rangle \tau_1 \ldots \langle p_{ln}, q_{gn} \rangle \tau_n \langle p_l, q_g \rangle \tau \langle p_l', q_g' \rangle$$

where $p_{l0}, \ldots, p_{ln}, p_l, p_l'$ are local states of M_2; otherwise, we can construct a TSM M_3 which differentiates between the runs of $M_1 || M_3$ and $M_2 || M_3$. Therefore, we know that $q_g' \in \delta_2(q_g)$. ∎

Theorem 3.1.4 *For any $n > 0$, there exists a TSM with the degree of parallelism of n such that there is no other m-equivalent TSM with the degree of parallelism smaller than n.*

Proof: Consider, for example, a TSM $\mathcal{M} = \langle \{c\}, \{q_{g1}, \ldots, q_{gn}\}, \{q_l\}, \{\langle q_{g1}, q_l \rangle\}, \{\tau_1, \ldots, \tau_n\}, \emptyset, \{\tau_1, \ldots, \tau_n\} \rangle$. For each τ_i, $\tau_i(q_{gi}, q_l) = \{\langle c, q_{gi}, q_l \rangle\}$ and $\tau_i(q_{gj}, q_l) = \emptyset$, for any $j \neq i$. Use the previous theorem on the underlying structure and induction on i to show that a fair computation that passes through $q_{g1}, \ldots, q_{gi}$ must contain an infinite number of occurrences of all transitions enabled at $q_{g1}, \ldots, q_{gi}$ and there must exist at least one transition enabled at $q_{g(i+1)}$ that is different from any transition enabled at $q_{g1}, \ldots, q_{gi}$. Therefore, any TSM which is m-equivalent to $\mathcal{M}$ must have at least n transitions.

More specifically, for each i, consider a TSM $\mathcal{M}_{\mathcal{E}} = \langle \{a, b, d\}, \{q_{g1}, \ldots, q_{gn}\}, \{q_{le}\}, \{\langle q_{g1}, q_{le}\rangle\}, \{\tau\}, \emptyset, \{\tau\}\rangle$ defined as follows:

- For $0 \leq j < i$, $\tau(q_{gj}, q_{le}) = \{\langle d, q_{g(j+1)}, q_{le}\rangle\}$.
- $\tau(q_{gi}, q_{le}) = \{\langle a, q_{g(i+1)}, q_{le}\rangle\}$.
- $\tau(q_{g(i+1)}, q_{le}) = \{\langle b, q_{gi}, q_{le}\rangle\}$.

Any fair computation of $\mathcal{M}||\mathcal{M}_E$ must include an infinite number of occurrences of the character c between a and b because τ_{i+1} must be taken infinitely often. Now, if TSM $\mathcal{M}'$ is m-equivalent to $\mathcal{M}$, then $\mathcal{M}'$ must have the same underlying structure. Consider a computation of $\mathcal{M}'||\mathcal{M}_E$ in which all transitions of $\mathcal{M}'$ enabled at q_{gj}, for all $0 \leq j < i$, are taken infinitely often but the transition(s) enabled at $q_{g(i+1)}$ is taken only finitely many times. It is easy to show that it exists. Since it does not contain an infinite number of occurrences of c between a and b, the computation must be an unfair computation. With the induction hypothesis, we can conclude that there must exist at least one transition that is enabled at $q_{g(i+1)}$ and is different from any transition enabled at $q_{g1}, \ldots, q_{gi}$. ∎

The above theorem establishes the m-equivalence hierachy of TSMs with different degrees of parallelism. Therefore, we say that a TSM $\mathcal{M}$ has the minimal degree of parallelism if there is no other m-equivalent TSM with the degree of parallelism smaller than $Par(\mathcal{M})$.

Theorem 3.1.5 *For any TSM $\mathcal{M}_1$, there exists a TSM $\mathcal{M}_2$ with $Par(\mathcal{M}_2) = 1$ such that for any TSM $\mathcal{M}'$, $\mathcal{L}_{\mathcal{M}_2||\mathcal{M}'} \subseteq \mathcal{L}_{\mathcal{M}_1||\mathcal{M}'}$.*

Proof: Given $\mathcal{M}_1 = \langle \Sigma, Q_g, Q_l, Q_0, \mathcal{T}, J, C\rangle$, construct $\mathcal{M}_2 = \langle \Sigma, Q_g, Q_l', Q_0', \{\tau'\}, \emptyset, \{\tau'\}\rangle$ as follows:

- Each $q_l' \in Q_l'$ consists of two components $\langle q_l, t\rangle$, where q_l is from Q_l and t is a transition queue $(\tau_1, \tau_2, \ldots, \tau_n)$, of transitions $\tau_i \in \mathcal{T}$.
- Q_0' includes all $\langle q_g, \langle q_l, t\rangle\rangle \in Q_g \times Q_l'$ such that $\langle q_g, q_l\rangle \in Q_0$.

- For any $q_g \in Q_g$, $q_l \in Q_l$ and any transition queue $t = (\tau_1, \tau_2, \ldots, \tau_n)$, let $1 \leq i \leq n$ be the smallest index such that $\tau_i \in Enabled(q_g, q_l)$. Then, we define τ' to be:

$$\tau'(q_g, \langle q_l, t\rangle) = \{\langle c, q'_g, \langle q'_l, (\tau_1, \ldots, \tau_{i-1}, \tau_{i+1}, \ldots, \tau_n, \tau_i)\rangle\rangle \mid \langle c, q_g, q'_l\rangle \in \tau_i(q_g, q_l)\}.$$

  If $Enabled(q_g, q_l) = \emptyset$, then $\tau'(q_g, \langle q_l, t\rangle) = \emptyset$.

The fair runs of the transition system $\mathcal{M}_2||\mathcal{M}'$ always correspond to some fair runs of the transition system $\mathcal{M}_1||\mathcal{M}'$. In short, $\mathcal{M}_2$ simulates the computation of $\mathcal{M}_1$. The fact that τ' is strongly fair ensures that if some transition of $\mathcal{M}_1$ remains enabled infinitely often, then $\mathcal{M}_2$ will eventually simulate a transition of $\mathcal{M}_1$. The transition queue guarantees that every transition enabled (infinitely often) is eventually taken. ∎

Theorem 3.1.6 *Fairness and determinism:*

1. *There are some STSMs with no m-equivalent WTSMs.*
2. *There are some WTSMs with no m-equivalent STSMs.*
3. *There are some TSMs with no m-equivalent DTSMs.*
4. *There are some TSMs with no m-equivalent τDTSMs.*

Proof: Consider the following TSMs:

1. a STSM $\mathcal{M}_1 = \langle\{a, b\}, \{q_{g1}, q_{g2}\}, \{q_l\}, \{\langle q_{g1}, q_l\rangle\}, \{\tau_1, \tau_2\}, \emptyset, \{\tau_1, \tau_2\}\rangle$ with
   - $\tau_1(q_{g1}, q_l) = \{\langle a, q_{g1}, q_l\rangle\}$,
   - $\tau_1(q_{g2}, q_l) = \emptyset$,
   - $\tau_2(q_{g2}, q_l) = \{\langle b, q_{g2}, q_l\rangle\}$, and,
   - $\tau_2(q_{g1}, q_l) = \emptyset$.
2. a WTSM $\mathcal{M}_2$ which is exactly the same as the STSM above except that both τ_1 and τ_2 are weakly fair instead.
3. a TSM $\mathcal{M}_3 = \langle\{a, b\}, \{q_{g1}, q_{g2}\}, \{q_{l1}, q_{l2}, q_{l3}\}, \{\langle q_{g1}, q_{l1}\rangle\}, \{\tau\}, \emptyset, \{\tau\}\rangle$ with
   - $\tau(q_{g1}, q_{l1}) = \{\langle a, q_{g1}, q_{l2}\rangle, \langle b, q_{g2}, q_{l3}\rangle\}$,

- $\tau(q_{g1}, q_{l2}) = \tau(q_{g2}, q_{l3}) = \{\langle b, q_{g1}, q_{l1}\rangle\}$, and,
- $\tau(q_{g2}, q_{l1}) = \tau(q_{g2}, q_{l2}) = \tau(q_{g1}, q_{l3}) = \emptyset$.

We can show that there is no WTSM [STSM] which is m-equivalent to $\mathcal{M}_1$ [resp. $\mathcal{M}_2$]. There is also no DTSM and no τDTSM which is m-equivalent to $\mathcal{M}_3$. The detail of the proof is left as an exercise to the reader. ∎

3.1.3 Modular forms

We can define a similar modular form $\langle \Sigma, Q_g, Q_l, \delta, Q_0, F\rangle$, for ω-automata with other acceptance conditions. The transition function δ is now $\delta : Q_g \times Q_l \times \Sigma \to 2^{Q_g \times Q_l}$, and Q_0 is a subset of $Q_g \times Q_l$. However, there are two possiblilities for F: F may involve local states Q_l (LOC) or the product states $Q_g \times Q_l$ (GLB). For example, using a Büchi acceptance condition, F could be either a subset of Q_l or a subset of $Q_g \times Q_l$. Since this is an excursion from our main interest, we will only briefly discuss about this topic.

A word w is accepted by a composition $\mathcal{M}||\mathcal{E}$ of a ω-automaton $\mathcal{M}$ and some composition $\mathcal{E}$ of other ω-automata and/or TSMs, iff w is an interleaved execution of all the components and satisfies the acceptance conditions of all the components including $\mathcal{M}$'s. We may require that $\mathcal{M}$ is executed infinitely often (INF). This requirement is sometimes called *impartiality.*

Theorem 3.1.7 *With (LOC) and (INF) conditions, any ω-automaton module with one of the acceptance conditions (Büchi, Rabin, Streett and Muller) is m-equivalent to some ω-automaton module with another kind of acceptance condition.*

Proof: From any ω-automaton module $\mathcal{M} = \langle \Sigma, Q_g, Q_l, \delta, Q_0, F\rangle$ with an acceptance condition of type $X \in \{$Büchi, Rabin, Streett, Muller$\}$, construct an ω-automaton $\mathcal{A} = \langle (Q_g \times \Sigma \times Q_g) \cup (Q_g \times Q_g \times \Sigma \times Q_g), Q_l \cup Q_{0A}, \delta_A, Q_{0A}, F\rangle$ where F is the same acceptance condition of the same type X, and,

- $Q_{0A} = \{\langle q_l, 0\rangle \mid \langle q_g, q_l\rangle \in Q_0,$ for some $q_g\}$,
- for any $q_l \in Q_l$, $\delta_A(q_l, \langle q_g, \sigma, q'_g\rangle) = \{q'_l \mid \langle q'_g, q'_l\rangle \in \delta(q_g, q_l, \sigma)\}$,
- for any $\langle q_l, 0\rangle \in Q_{0A}$, $\delta_A(\langle q_l, 0\rangle, \langle q_i, q_g, \sigma, q'_g\rangle) = \{q'_l \mid \langle q'_g, q'_l\rangle \in \delta(q_g, q_l, \sigma) \;\wedge\; \langle q_i, q_l\rangle \in Q_0\}$.

Next, transform $\mathcal{A}$ into another ω-automaton $\mathcal{B} = \langle (Q_g \times \Sigma \times Q_g) \cup (Q_g \times Q_g \times \Sigma \times Q_g), Q'_l, \delta_B, Q_{0B}, F' \rangle$ where F' is an acceptance condition of another type $Y \in$ {Büchi, Rabin, Streett, Muller}. Then, construct an ω-automaton module $\mathcal{M}' = \langle \Sigma, Q_g, Q'_l \cup Q'_{l0}, \delta', Q'_0, F' \rangle$ where F' is again an acceptance condition of the type Y, and,

- $Q'_{l0} = Q_g \times Q_{0B}$,
- $Q'_0 = \{\langle q_i, \langle q_i, q_l \rangle \rangle \mid \langle q_i, q_l \rangle \in Q_{l0}\}$,
- for any $q_l \in Q'_l$, $\delta'(q_g, q_l, \sigma) = \{\langle q'_g, q'_l \rangle \mid q'_l \in \delta_B(q_l, \langle q_g, \sigma, q'_g \rangle)\}$, and,
- for any $\langle q_i, q_l \rangle \in Q'_{l0}$, $\delta'(q_g, \langle q_i, q_l \rangle, \sigma) = \{\langle q'_g, q'_l \rangle \mid q'_l \in \delta_B(q_l, \langle q_i, q_g, \sigma, q'_g \rangle)\}$.

It is not hard to see that $\mathcal{M}$ and $\mathcal{M}'$ are m-equivalent. ∎

Proposition 3.1.1 *With (LOC) condition, Büchi = Rabin ⊂ Streett = Muller, that is:*

1. *any ω-automaton module with acceptance condition of type $X \in$ {Büchi, Rabin} is m-equivalent to some ω-automaton module with acceptance condition of type $Y \in$ {Büchi, Rabin, Streett, Muller}.*
2. *any ω-automaton module with acceptance condition of type $X \in$ {Streett, Muller} is m-equivalent to some ω-automaton module with acceptance condition of type $Y \in$ {Streett, Muller}.*

However, if we consider a composition of two Büchi ω-automaton submodules $\mathcal{M}_1 || \mathcal{M}_2$ where each of $\mathcal{M}_i = \langle \Sigma, Q_g, Q_{lg}, Q_{lli}, \delta_i, F_i \rangle$. Q_{lg} is the set of states shared by both submodules and observable only to (local to) both submodules, while Q_{lli} is local to only $\mathcal{M}_i$. This composition can also be considered as an ω-automaton module with a variant of Büchi condition, denoted by 2-Büchi, where F is a pair of two subsets of local states.

Proposition 3.1.2 *With (LOC) condition, any Street or Muller ω-automaton module is m-equivalent to a composition of two Büchi ω-automaton submodules.*

Proposition 3.1.3 *With (GLB) condition, Büchi ⊂ Rabin ⊂ Streett = Muller:*

1. *any Büchi ω-automaton module is m-equivalent to some ω-automaton module with another kind of acceptance condition.*

2. *any Rabin ω-automaton module is m-equivalent to some Streett/Muller ω-automaton module.*

3. *any Streett ω-automaton module is m-equivalent to some Muller ω-automaton module, and vice versa.*

3.2 Synthesis Algorithm

In this section, the generic synthesis algorithm is presented in two steps: first, a descriptive overview of the algorithm, and then the actual details of the algorithm. Later, we will show the application of this algorithm to synthesis problems within various frameworks.

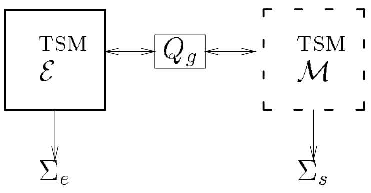

Figure 3.1: Synthesizing $\mathcal{M}$ under the environment $\mathcal{E}$ with a global state (ranging over Q_g)

3.2.1 Overview

The inputs to the algorithm consist of three parts: a specification ψ, an environment model $\mathcal{E}$, and an execution model $\mathcal{X}$. The specification defines the *good* or desired behaviors of the entire system (the composition of the environment and the module being synthesized) while the environment model and the execution model restricts the set of all possible interleaving behaviors down to a set of behaviors we are interested in. We can also think of the execution model as representing the scheduler.

The goal of the synthesis algorithm is to synthesize a module $\mathcal{M}$ such that whenever the behavior of $\mathcal{M}||\mathcal{E}$ satisfies the execution model, it must also satisfy the specification.

One of the components of the environment model $\mathcal{E}$ is a set of global states Q_g. This represents a shared variable, between the environment and the module we are going to synthesize, whose range is Q_g. This is the only means of communication and "control" between the environment and the module.

Specification

The behaviors that we want to specify are finite and infinite sequences of the global states (Q_g), the outputs of the environment (Σ_e), and the outputs of the module (Σ_s).

$$q_{g0} \xrightarrow{\sigma_{e0}} q_{g1} \xrightarrow{\sigma_{s1}} q_{g2} \xrightarrow{\sigma_{s2}} q_{g3} \xrightarrow{\sigma_{e4}} \dots \quad [\xrightarrow{\sigma_{sk}} q_{g(k+1)}]$$

We can assume without losing generality that $\Sigma_e \cap \Sigma_s = \emptyset$.

The finite behaviors can represent either a dead-lock or a termination. In order to handle finite and infinite sequences uniformly, we extend a finite sequence into an infinite sequence by concatenating the finite sequence with a new special character $h \notin \Sigma_e \cup \Sigma_s$ (halt) and an (arbitrary) infinite sequence.

Therefore, given a specification ψ which defines $\mathcal{L}_\psi \subseteq (Q_g \times (\Sigma_e \cup \Sigma_s \cup \{h\}))^\omega$, we can consider ψ as the specification of infinite behaviors $\mathcal{L}_\psi^{inf}$ and finite behaviors $\mathcal{L}_\psi^{fin}$ where $\mathcal{L}_\psi^{inf} = \mathcal{L}_\psi \cap (Q_g \times (\Sigma_e \cup \Sigma_s))^\omega$ and,

$$\mathcal{L}_\psi^{fin} = \{w \in (Q_g \times (\Sigma_e \cup \Sigma_s))^* \mid \text{for some } \alpha \in (Q_g \times (\Sigma_e \cup \Sigma_s \cup \{h\}))^\omega,\ wh\alpha \in \mathcal{L}_\psi\}.$$

An infinite (finite) behavior α (w) of $\mathcal{M}||\mathcal{E}$ satisfies a specification ψ iff $\alpha \in \mathcal{L}_\psi^{inf}$ (resp. $w \in \mathcal{L}_\psi^{fin}$). For finite behaviors, this means that the implementation does not introduce new dead-locks (or terminating behaviors).

Execution model

The execution model $\mathcal{X}$ characterizes a language $\mathcal{L}_\mathcal{X} \subseteq \{0,1\}^n((\{0,\dots,n\}\times\{0,1\}^n)\cup\{h\})^\omega$. Each word in $\mathcal{L}_\mathcal{X}$ is a representation of a "good" or "possible" scheduling sequence. In other words, we may assume a scheduling sequence that is not in $\mathcal{L}_\mathcal{X}$ may not or cannot happen. If $\mathcal{X}$ encodes fair computations, then each word in $\mathcal{L}_\mathcal{X}$ describes a fair scheduling sequence.

The special character h is used here to represent, again, the finite scheduling sequences. In each (non-h) character $\langle i, a_1, \dots, a_n\rangle$, the first component i, which ranges from 0 to n, describes who is scheduled to run and which transition is taken. If $i = 0$, it is the environment's turn. If $i \neq 0$, then it is the module's turn and the i-th transition of the module is taken. For the other components, $a_j = 1$ indicates that the j-th transition is enabled in the state after the transition has been completed. The first character of every word has the form $\langle a_1, \dots, a_n\rangle$, and similar to the previous case, each a_j in this case indicates whether the j-th transition is enabled at the initial state. Figure 3.2 summarizes the relation between scheduling sequences and behaviors.

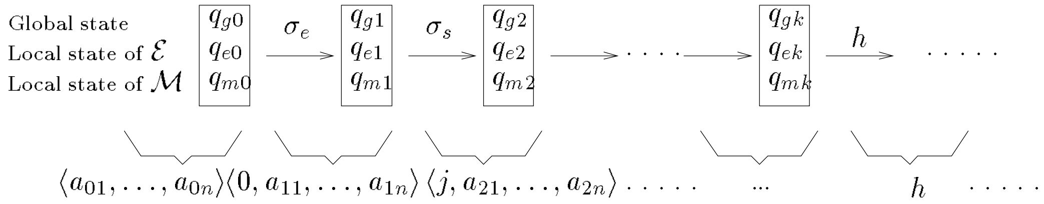

Figure 3.2: scheduling sequence

Figure 3.3 gives an example of an execution model. Using this model, the output of the synthesis algorithm is a module with one weakly fair transition ($n = 1$). The model itself is a Büchi automaton where q_{x0} is the initial state and q_{x2} is the final state. It is easy to see that this model excludes all scheduling sequences that remain infinitely at q_{x1}. Those are exactly the unfair scheduling sequences $x\langle 0, 1\rangle^\omega$ where the module (the single transition in the module) is enabled continuously from a certain point on but is not taken. For each pair $\langle 0, 1\rangle$, the character 0 in the first position means that the module is not scheduled to run and 1 indicates that the module is enabled. For some reasons that will be explained later, the synthesis algorithm will ensure that any behavior of $\mathcal{M}||\mathcal{E}$ that corresponds to an unfair scheduling sequence is not constrained to satisfy the specification.

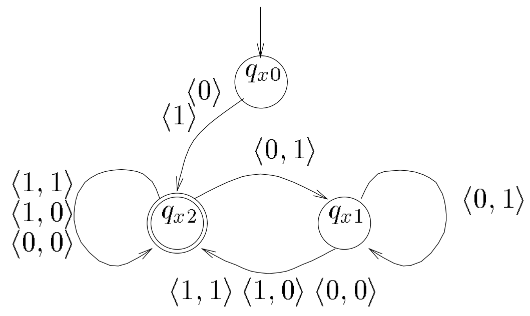

Figure 3.3: execution model example with 1 weakly fair transition

Note also that in this particular model, all scheduling sequences that contain h are rejected by the automaton. This means that any finite behavior of $\mathcal{M}||\mathcal{E}$ does not have to satisfy the specification. If we consider the execution model as representing an assumption of what we consider "good" or "possible" scheduling, we may say that any finite behavior satisfies the specification *vacuously*.

Figure 3.4 is a modified version of the previous execution model. Unlike the previous model, in this new model, scheduling sequences that result in a finite behavior are accepted. It is worth pointing out that, for a scheduling sequence with h to be accepted, the first h must occur immediately after some character of the form $\langle i, 0\rangle$. This means that an execution can stop only when the module is not enabled. Later, we will see how we can encode the other half of the requirement, that is, an execution can stop only when the environment is not enabled. Later, we will see that this technique also applies to other frameworks (e.g. synchronous composition).

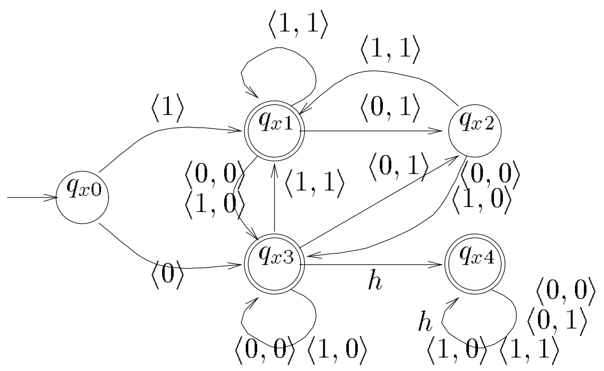

Figure 3.4: execution model example with h

Environment model

The environment model $\mathcal{E}$ is a TSM. In order to combine $\mathcal{E}$ later with the specification and the execution model, we construct an automaton that recognizes a behavior

$$q_{g0} \xrightarrow{\sigma_{e0}} q_{g1} \xrightarrow{\sigma_{s1}} q_{g2} \xrightarrow{\sigma_{s2}} q_{g3} \xrightarrow{\sigma_{e4}} \dots \quad [\ \xrightarrow{\sigma_{sk}} q_{g(k+1)}\ ] \quad \in Q_g((\Sigma_e \cup \Sigma_s) \times Q_g)^{(\omega,*)}$$

iff it is a behavior of $\mathcal{M}||\mathcal{E}$, for some $\mathcal{M}$.

Like what we have done for the specification, we augment finite behaviors with the special character h and an infinite sequence. This can be done simply by adding a transition $\langle q_g, q_{le}\rangle \xrightarrow{h}$ whenever the environment is not enabled at $\langle q_g, q_{le}\rangle$, where q_{le} is a local state of the environment model. The result is an automaton that recognizes infinite words of the form $Q_g((\Sigma_e \cup \Sigma_s \cup \{h\}) \times Q_g)^{\omega}$.

Combined specification, environment model and execution model

One of the key steps in the algorithm is to complement the specification and construct a product of the environment model, the execution model and the complement of the specification. The result is an automaton $\mathcal{A}$ that accepts a behavior and a scheduling sequence

$$\begin{array}{cccccc} q_{g0} & \xrightarrow{\sigma_0} q_{g1} & \xrightarrow{\sigma_1} q_{g2} & \ldots & \xrightarrow{h} \ldots & \ldots \\ \langle a_{00},\ldots,a_{0n}\rangle & \langle i_0, a_{10},\ldots,a_{1n}\rangle & \langle i_1, a_{20},\ldots,a_{2n}\rangle & \ldots & h & \ldots \end{array}$$

iff the behavior is a behavior of the environment model $\mathcal{E}$ but does not satisfy the specification ψ and the scheduling sequence is accepted by $\mathcal{X}$.

Projecting and Checking Enabling Condition Consistency

At this point, we consider the projection of behaviors of the entire system into behaviors of the module being synthesized while retaining the essential information of the corresponding scheduling sequence.

For convenience, let $\{a_j\}$ denote $a_{j1},\ldots,a_{jn}$ in Figure 3.5. The main idea of the projection is demonstrated in Figure 3.5 (a). A section of the behavior of the entire system ($\xrightarrow{\sigma_{sk}} q_{gk} \ldots \xrightarrow{\sigma_{em}} q_{gm}$) and the scheduling sequence ($\langle i_k, a_{k1},\ldots,a_{kn}\rangle,\ldots,\langle 0, a_{m1},\ldots,a_{mn}\rangle$) is projected into a tuple $\langle \sigma_{sk},\ q_{gk},\ i_k, Q_1,\ldots,\ Q_n, q_{gm}\rangle$. In short, the tuple encodes a move $(\sigma_{sk},\ q_{gk},\ i_k)$ by the module being synthesized, the enabling condition $(Q_1,\ldots,Q_n \subseteq Q_g)$, and the global state the module observed (q_{gm}) before it makes the next move. For the enabling condition, we require that $Q_1,\ldots,Q_n$ are *consistent* with $\{a_k\}\ldots\{a_m\}$. In other words, for any j, $1 \le j \le n$ and any i, $k \le i \le m$, $a_{ij} = 1$ iff $q_{gi} \in Q_j$. In the special case, when $k = 0$, the initial section is projected into a tuple $\langle q_{g0}, Q_1,\ldots,Q_n, q_{gm}\rangle$.

If the module is scheduled to run only finitely many times, then it must be the case that the scheduling sequence either has the form $\ldots\langle i_k,\ldots\rangle\langle 0,\ldots\rangle\ldots\langle 0,\ldots\rangle\ldots$, or contains h. Figure 3.5 (b) and (c) show the projection of such cases. Therefore, the projection of a behavior of the entire system and a scheduling sequence is a finite or infinite sequence of the tuples described above.

To handle finite and infinite projections uniformly, we simply augment any finite projection u,

$$u \in ((Q_g \times (2^{Q_g})^n \times Q_g)(\Sigma_s \times Q_g \times \{1,\ldots,n\} \times (2^{Q_g})^n \times Q_g)^*(\Sigma_s \times Q_g \times \{1,\ldots,n\} \times (2^{Q_g})^n)$$

$$\underbrace{\begin{array}{cc} \cdots & \xrightarrow{\sigma_{e(k-1)}} q_{g(k-1)} \\ \cdots & \langle 0, \{a_{k-1}\}\rangle \end{array}}_{\langle \ldots, q_{g(k-1)}\rangle} \underbrace{\begin{array}{cccc} \xrightarrow{\sigma_{sk}} q_{gk} & \xrightarrow{\sigma_{e(k+1)}} q_{g(k+1)} & \cdots & \xrightarrow{\sigma_{em}} q_{gm} \\ \langle i_k, \{a_k\}\rangle & \langle 0, \{a_{k+1}\}\rangle & \cdots & \langle 0, \{a_m\}\rangle \end{array}}_{\langle \sigma_{sk}, q_{gk}, i_k, Q_1, \ldots, Q_n, q_{gm}\rangle} \underbrace{\begin{array}{cc} \xrightarrow{\sigma_{s(m+1)}} q_{g(m+1)} & \cdots \\ \langle i_{m+1}, \{a_{m+1}\}\rangle & \cdots \end{array}}_{\langle \sigma_{s(m+1)}, q_{g(m+1)}, i_{m+1}, \ldots\rangle}$$

(a)

$$\underbrace{\begin{array}{cc} \cdots & \xrightarrow{\sigma_{e(k-1)}} q_{g(k-1)} \\ \cdots & \langle 0, \{a_{k-1}\}\rangle \end{array}}_{\langle \ldots, q_{g(k-1)}\rangle} \underbrace{\begin{array}{ccccc} \xrightarrow{\sigma_{sk}} q_{gk} & \xrightarrow{\sigma_{e(k+1)}} q_{g(k+1)} & \cdots & \cdots & \cdots \\ \langle i_k, \{a_k\}\rangle & \langle 0, \{a_{k+1}\}\rangle & \cdots & \langle 0, \ldots\rangle & \cdots \end{array}}_{\langle \sigma_{sk}, q_{gk}, i_k, Q_1, \ldots, Q_n\rangle}$$

(b)

$$\underbrace{\begin{array}{cc} \cdots & \xrightarrow{\sigma_{e(k-1)}} q_{g(k-1)} \\ \cdots & \langle 0, \{a_{k-1}\}\rangle \end{array}}_{\langle \ldots, q_{g(k-1)}\rangle} \underbrace{\begin{array}{cccccc} \xrightarrow{\sigma_{sk}} q_{gk} & \xrightarrow{\sigma_{e(k+1)}} q_{g(k+1)} & \cdots & \xrightarrow{\sigma_{em}} q_{gm} & \xrightarrow{h} \cdots & \cdots \\ \langle i_k, \{a_k\}\rangle & \langle 0, \{a_{k+1}\}\rangle & \cdots & \langle 0, \{a_m\}\rangle & h & \cdots \end{array}}_{\langle \sigma_{sk}, q_{gk}, i_k, Q_1, \ldots, Q_n, \rangle}$$

(c)

Figure 3.5: Projection

into an (infinite) *extended projection* $u\eta$,

$$u\eta \in ((Q_g \times (2^{Q_g})^n \times Q_g)(\Sigma_s \times Q_g \times \{1, \ldots, n\} \times (2^{Q_g})^n \times Q_g)^\omega$$

with an arbitrary infinite sequence $\eta \in Q_g(\Sigma_s \times Q_g \times \{1, \ldots, n\} \times (2^{Q_g})^n \times Q_g)^\omega$.

From the product automaton $\mathcal{A}$ we constructed earlier, we construct another automaton $\mathcal{B}$ that accepts a projection sequence γ,

$$\gamma \in (Q_g \times (2^{Q_g})^n \times Q_g)(\Sigma_s \times Q_g \times \{1, \ldots, n\} \times (2^{Q_g})^n \times Q_g)^\omega$$

iff it is a projection of some behaviors and scheduling sequences that are accepted by $\mathcal{A}$. Therefore, if $\mathcal{B}$ accepts γ, then there is a behavior β and a scheduling sequence α such that:

- β is a behavior of the environment $\mathcal{E}$ but does not satisfy ψ,
- α is accepted by $\mathcal{X}$ (i.e. it is a good or possible scheduling sequence), and,
- γ is a projection or extended projection of β and α.

Co-determinization and Checking Execution Consistency

The next step is to co-determinize the automaton $\mathcal{B}$. The result is an automaton $\mathcal{C}$ that accepts a projection sequence γ iff for any behavior β and scheduling α, if

- β is a behavior of the environment $\mathcal{E}$,
- α is accepted by $\mathcal{X}$, and,
- γ is a projection or extended projection of β and α,

then β satisfies ψ.

Consider any two consecutive tuples

$$\ldots\langle\sigma_{s1}, q_{g1}, i_1, Q_{11}, \ldots, Q_{1n}, q_{g1}^{obs}\rangle\langle\sigma_{s2}, q_{g2}, i_2, Q_{21}, \ldots, Q_{2n}, q_{g2}^{obs}\rangle\ldots$$

in a projection sequence. From the definitions, Q_{1i_2} are the global states in which the i_2-th transition are enabled, and q_{g1}^{obs} is the global state observed by the module before it takes the i_2-th transition. Therefore, for a projection sequence to be *consistent* with some execution of the entire system, it must be the case that for any such two tuples, $q_{g1}^{obs} \in Q_{1i_2}$.

From the automaton $\mathcal{C}$, we construct an automaton $\mathcal{D}$ that accepts a projection sequence γ iff for any behavior β and scheduling α, if

- β is a behavior of the environment $\mathcal{E}$,
- α is accepted by $\mathcal{X}$, and,
- γ is a <u>consistent</u> projection or extended projection of β and α,

then β satisfies ψ.

Synthesis

We can interpret the automaton $\mathcal{D}$ as an automaton on labeled trees where the observed global states and the index of the transitions (taken after having observed the global states) give the branch directions in the trees and the rest of the projection tuples are the labels on the trees. Figure 3.5 illustrates how a projection sequence is interpreted as a path in a tree.

Finally, we check the emptiness of the tree automaton $\mathcal{D}$. If it is not empty, then it must accept a regular tree which is generated by a transducer $\mathcal{T}$. A TSM $\mathcal{M}$ can then

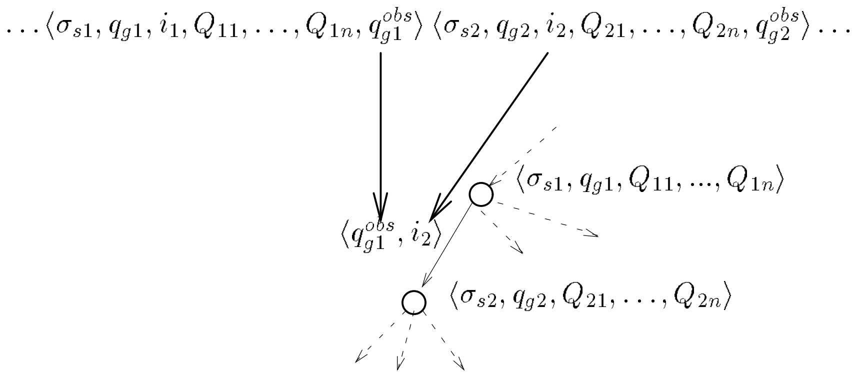

Figure 3.6: interpreting a projection sequence as a path in a labeled tree

be constructed from $\mathcal{T}$. Being constructed from a tree accepted by $\mathcal{D}$, $\mathcal{T}$ and $\mathcal{M}$ has the property that, regardless of how the directions are chosen in the tree, all the paths satisfy the acceptance condition of $\mathcal{D}$. Each path is a projection sequence and we can consider $\mathcal{M}$ as projection sequence generator. Therefore, $\mathcal{M}$ generates a projection sequence γ iff for any behavior β and scheduling α, if

- β is a behavior of the environment $\mathcal{E}$,
- α is accepted by $\mathcal{X}$, and,
- γ is a consistent projection or extended projection of β and α,

then β satisfies ψ. In other words, any behavior of $\mathcal{M}||\mathcal{E}$ with a scheduling sequence accepted by $\mathcal{X}$ satisfies ψ.

3.2.2 Detailed constructions

Input/Output

The inputs to the algorithm are:

1. a specification ψ defining $\mathcal{L}_\psi \subseteq (Q_g \times (\Sigma_e \cup \Sigma_s \cup \{h\}))^\omega$, assuming without losing generality that $h \notin \Sigma_e \cup \Sigma_s$ and $\Sigma_e \cap \Sigma_s = \emptyset$.

2. an environment model TSM $\mathcal{E} = \langle \Sigma_e, Q_g, Q_{le}, Q_{0e}, \mathcal{T}_e, J_e, C_e \rangle$.

3. an execution model, represented by an ω-automaton $\mathcal{X}$, which recognizes a language $\mathcal{L}_{\mathcal{X}}$, $\mathcal{L}_{\mathcal{X}} \subseteq \{0,1\}^n((\{0,\ldots,n\} \times \{0,1\}^n) \cup \{h\})^\omega$.

The output is a transition system module with n transitions.

Specification

From ψ, we construct a Büchi automaton $\mathcal{S}$, which recognizes the complement language $\bar{\mathcal{L}_\psi}$:

$$\bar{\mathcal{L}_\psi} = \{t\alpha \mid \text{for any } t \in \Sigma_e \cup \Sigma_s \cup \{h\} \text{ and any } \alpha \notin \mathcal{L}_\psi\}.$$

Note that an arbitrary character t is added to shift the language from $(Q_g \times (\Sigma_e \cup \Sigma_s \cup \{h\}))^\omega$ to $((\Sigma_e \cup \Sigma_s \cup \{h\}) \times Q_g)^\omega$. The only purpose of this shifting is to make the later formulation of a product automaton more convenient.

Environment model

From $\mathcal{E}$, construct a Streett automaton $\mathcal{E}' = \langle \Sigma', Q', \{q_0\}, \delta, F' \rangle$ where $\Sigma' = (\Sigma_e \cup \Sigma_s \cup \{h\}) \times Q_g$, and $Q' = \{q_0 \cup q_h\} \cup (Q_g \times Q_{le} \times (\mathcal{T}_e \cup \{s\}))$. The transition function $\delta : Q' \times \Sigma' \to 2^{Q'}$ is defined as follows:

- for q_0 and any $\langle \sigma, q_g \rangle \in \Sigma'$,

$$\delta(q_0, \langle \sigma, q_g \rangle) = \{\langle q_g, q_l, s \rangle \mid \langle q_g, q_l \rangle \in Q_{0e}\}.$$

- for q_h and any $\langle \sigma, q_g \rangle \in \Sigma'$,

$$\delta(q_h, \langle \sigma, q_g \rangle) = \{q_h\}.$$

- for any $\langle q_g, q_l, x \rangle \in Q'$ and any $\langle \sigma, q_g' \rangle \in \Sigma_e \times Q_g$,

$$\delta(\langle q_g, q_l, x \rangle, \langle \sigma, q_g' \rangle) = \{\langle q_g', q_l', \tau \rangle \mid \text{ any } \tau \in \mathcal{T},\ \langle \sigma, q_g', q_l' \rangle \in \tau(q_g, q_l)\}.$$

- for any $\langle q_g, q_l, x \rangle \in Q'$ and any $\langle \sigma, q_g' \rangle \in \Sigma_s \times Q_g$,

$$\delta(\langle q_g, q_l, x \rangle, \langle \sigma, q_g' \rangle) = Q_g \times \{q_l\} \times \{s\}.$$

- for any $\langle q_g, q_l, x \rangle \in Q'$ and any $q_g' \in Q_g$,

$$\delta(\langle q_g, q_l, x \rangle, \langle h, q_g' \rangle) = \begin{cases} \{q_h\} & \text{if } \mathit{Enabled}(q_g, q_l) = \emptyset \\ \emptyset & \text{otherwise.} \end{cases}$$

For the acceptance condition, $F' = \{(L_\tau, U_\tau)\}_{\tau \in J_e \cup C_e}$ and

- for each $\tau \in J_e$, $L_\tau = Q'$ and $U_\tau = \{q_h\} \cup \{\langle q_g, q_l, t\rangle \in Q' \mid \tau(q_g, q_l) = \emptyset\} \cup Q_g \times Q_l \times \{\tau\}$,
- for each $\tau \in C_e$, $L_\tau = \{\langle q_g, q_l, t\rangle \in Q' \mid \tau(q_g, q_l) \neq \emptyset\}$ and $U_\tau = \{q_h\} \cup (Q_g \times Q_l \times \{\tau\})$.

Then, transform $\mathcal{E}'$ into a Büchi automaton $\hat{\mathcal{E}}$.

Execution model

From $\mathcal{X}$, construct a Büchi automaton $\hat{\mathcal{X}}$ such that $\mathcal{L}_{\hat{\mathcal{X}}} = \{i\beta \mid \beta \in \mathcal{L}_{\mathcal{X}} \ \wedge \ i \in \{0, \ldots, n\}\}$. The language $\mathcal{L}_{\hat{\mathcal{X}}}$ is basically equivalent to $\mathcal{L}_{\mathcal{X}}$ except for an extra first character that is added to every word in $\mathcal{L}_{\mathcal{X}}$.

Combine Specification, Environment Model and Execution Model

From the complement of the specification $\mathcal{S} = \langle(\Sigma_e \cup \Sigma_s \cup \{h\}) \times Q_g, Q_s, Q_{s0}, \delta_s, F_s\rangle$, the environment model $\hat{\mathcal{E}} = \langle(\Sigma_e \cup \Sigma_s \cup \{h\}) \times Q_g, Q_e, Q_{e0}, \delta_e, F_e\rangle$, and the execution model $\hat{\mathcal{X}} = \langle(\{0, \ldots, n\} \times \{0,1\}^n) \cup \{h\}, Q_x, Q_{x0}, \delta_x, F_x\rangle$, we construct the product automaton of $\mathcal{S}$, $\hat{\mathcal{E}}$ and $\hat{\mathcal{X}}$. Each $\langle\sigma_e, q_g\rangle \in \Sigma_e \times Q_g$ of $\mathcal{S}$ and $\hat{\mathcal{E}}$ is matched with some characters from $\{0\} \times \{0,1\}^n$ of $\hat{\mathcal{X}}$ and each $\langle\sigma_s, q_g\rangle \in \Sigma_s \times Q_g$ with some characters from $\{1, \ldots, n\} \times \{0,1\}^n$. The special case $\langle h, q_g\rangle$ of $\mathcal{S}$ and $\hat{\mathcal{E}}$ is matched with h of $\hat{\mathcal{X}}$.

Formally, we construct a Büchi automaton $\mathcal{A} = \langle\Sigma_A, Q_A, Q_{A0}, \delta_A, F_A\rangle$ where the alphabet is

$$\begin{aligned}\Sigma_A = \ & (\Sigma_e \times Q_g \times \{0,1\}^n) \cup \\ & (\Sigma_s \times Q_g \times \{1, \ldots, n\} \times \{0,1\}^n) \cup \\ & (\{h\} \times Q_g).\end{aligned}$$

The state Q_A is $Q_s \times Q_e \times Q_x \times \{0, \ldots, 3\}$, the initial state $Q_{A0} = Q_{s0} \times Q_{e0} \times Q_{x0} \times \{0\}$ and the final state $F_A = Q_{s0} \times Q_{e0} \times Q_{x0} \times \{3\}$. The matching described above is reflected in the transition function δ_A. For example, in the case of the environment's move (when $\sigma \in \Sigma_e$), $\langle q'_s, q'_e, q'_x, j\rangle \in \delta_A(\langle q_s, q_e, q_x, i\rangle, \langle\sigma, q_g, a_1, \ldots, a_n\rangle)$ iff $q'_s \in \delta_s(q_s, \langle\sigma, q_g\rangle)$, $q'_e \in \delta_e(q_e, \langle\sigma, q_g\rangle)$, $q'_x \in \delta_x(q_x, \langle 0, a_1, \ldots, a_n\rangle)$, and,

$$j = \begin{cases} (i+1) \ mod \ 4 & \text{if} \quad i = 0 \text{ and } q'_s \in F_s \ \text{ or } \ i = 1 \text{ and } q'_e \in F_e \ \text{ or} \\ & \qquad i = 2 \text{ and } q'_x \in F_x \ \text{ or } \ i = 3 \\ i & \text{otherwise.} \end{cases}$$

In the other cases, all the conditions are similar except $q'_x \in \delta_x(q_x, \langle k, a_1, \ldots, a_n\rangle)$ when $\sigma \in \Sigma_s$, and $q'_x \in \delta_x(q_x, h)$ when $\sigma = h$.

Projecting and Checking Enabling Condition Consistency

Next, we construct, from $\mathcal{A}$, another Büchi automaton $\mathcal{B} = \langle \Sigma_B, Q_B, Q_{B0}, \delta_B, F_B \rangle$. Now, $\Sigma_B = \Sigma_s \times Q_g \times \{1, \ldots, n\} \times (2^{Q_g})^n \times Q_g$ reflects only the action of the module being synthesized ($\Sigma_s \times Q_g \times \{1, \ldots, n\}$), the enabling condition of the module $(2^{Q_g})^n$, and the global states observed by the module Q_g, respectively. The set of states Q_B is simply $(Q_A \times \{0,1\}) \cup \{q_f\}$, the initial state $Q_B = Q_{0A} \times \{0\}$, and the accepting state F_B is $(F_A \times \{0,1\}) \cup (Q_A \times \{1\}) \cup \{q_f\}$.

We say that a path in $\mathcal{A}$,

$$q_1 \xrightarrow{\langle \sigma_s, q_{g1}, i, a_{11}, \ldots, a_{1n} \rangle} q_2 \xrightarrow{\sigma_{A2}} q_3 \xrightarrow{\sigma_{A3}} \ldots q_k \xrightarrow{\sigma_{Ak}} q_{k+1}$$

where $\sigma_{Aj} = \langle \sigma_e, q_{gj}, a_{j1}, \ldots, a_{jn} \rangle$, is *consistent* with $\langle \sigma_s, q_{g1}, i, Q_1, \ldots, Q_n, q_{gk} \rangle$, if for any $1 \le j \le k$ and any $1 \le l \le n$, $a_{jl} = 1$ iff $q_{gj} \in Q_l$.

We also say that an infinite path in $\mathcal{A}$,

$$q_1 \xrightarrow{\langle \sigma_s, q_{g1}, i, a_{11}, \ldots, a_{1n} \rangle} q_2 \xrightarrow{\sigma_{A2}} q_3 \xrightarrow{\sigma_{A3}} \ldots$$

where $\sigma_{Aj} \in \Sigma_A$, *halts* the (synthesized) module with $\langle \sigma_s, q_{g1}, i, Q_1, \ldots, Q_n, q'_g \rangle$, iff it passes through some states of F_A infinitely often and either:

- for all $2 \le l$, σ_{Aj} is of the form $\langle \sigma_e, q_{gj}, a_{j1}, \ldots, a_{jn} \rangle$, and for any $j \ge 1$, any $1 \le l \le n$, $a_{jl} = 1$ iff $q_{gj} \in Q_l$.
- there is some k such that $\sigma_{Ak} \in \{h\} \times Q_g$, and for any $2 \le j < k$, σ_{A_j} is of the form $\langle \sigma_e, q_{gj}, a_{j1}, \ldots, a_{jn} \rangle$, and for any $1 \le j < k$, any $1 \le l \le n$, $a_{jl} = 1$ iff $q_{gj} \in Q_l$.

For δ_B, we have three cases

- $\langle q'_A, 0 \rangle \in \delta_B(\langle q_A, i \rangle, \sigma_B)$ iff there is a path from q_A to q'_A that is consistent with σ_B.
- $\langle q'_A, 1 \rangle \in \delta_B(\langle q_A, i \rangle, \sigma_B)$ iff there is a path from q_A to q'_A that is consistent with σ_B and passes through some state in F_A.
- $q_f \in \delta_B(\langle q_A, i \rangle, \sigma_B)$ iff there is an infinite path that starts from q_A and halts the module with σ_B.

For the special case q_f, $\delta(q_f, \sigma_B) = \{q_f\}$.

Co-determinization and Checking Execution Consistency

We can co-determinize [Saf88,EJ89] the Büchi automaton $\mathcal{B}$ into a deterministic Rabin automaton $\mathcal{C}$, which rejects all the Σ_B^ω words accepted by $\mathcal{B}$.

From $\mathcal{C} = \langle \Sigma_B, Q_C, Q_{C0}, \delta_C, F_C \rangle$, we can construct a Rabin tree automaton $\mathcal{D} = \langle \Sigma_D, Q_D, Q_{0D}, \delta_D, F_D \rangle$. The alphabet Σ_D is the product $Q_g \times \{1, \ldots, n\}$ of the (observed) global states and the index of the (taken) transitions. The set of states Q_D and initial states Q_{D0} are $(Q_C \times (\Sigma_s \times Q_g \times (2^{Q_g})^n)) \cup \{q_{imp}\}$ and $Q_{C0} \times (\Sigma_s \times Q_g \times (2^{Q_g})^n)$, respectively. Suppose $F_C = \{(L_{C1}, U_{C1}), \ldots, (L_{Ck}, U_{Ck})\}$. Then, the acceptance condition F_D is $\{(\emptyset, \{q_{imp}\}), (L_{D1}, U_{D1}), \ldots, (L_{Dk}, U_{Dk})\}$ where each L_{Di} (U_{Di}) is the product of L_{Ci} (resp. U_{Ci}) and $\Sigma_s \times Q_g \times (2^{Q_g})^n$.

For the transition funtion δ_D, if $q_g^{obs} \in Q_i$ then,

$$\begin{aligned}
&\delta_D(\langle q_c, \sigma_s, q_g, Q_1, \ldots, Q_n \rangle, \langle q_g^{obs}, i \rangle) = \\
&\qquad \{\{q_c'\} \times \Sigma_s \times Q_g \times (2^{Q_g})^n \mid \text{ for some } j,\ q_c' \in \delta_C(q_c, \langle \sigma_s, q_g, j, Q_1, \ldots, Q_n, q_g^{obs} \rangle)\}
\end{aligned}$$

Otherwise (when $q_g^{obs} \notin Q_i$), $\delta_D(\langle q_c, \sigma_s, q_g, Q_1, \ldots, Q_n \rangle, \langle q_g^{obs}, i \rangle) = \{q_{imp}\}$. For the special case q_{imp}, $\delta_D(q_{imp}, \sigma_D) = \{q_{imp}\}$.

Synthesis

As in [PR89a,PR89b], we check for the emptiness of the tree automaton $\mathcal{D}$ and construct a (deterministic) ω-transducer $\mathcal{T} = \langle \Sigma_D, Q_T, q_{T0}, \delta_T, l_T \rangle$, which is a folded representation of a regular tree accepted by $\mathcal{D}$. The labeling function l_T is a function $l_T : Q_T \to \Sigma_s \times Q_g \times (2^{Q_g})^n$. For any $q_t \in Q_T$, if $l_T(q_t) = \langle \sigma_s, q_g, Q_1, \ldots, Q_n \rangle$, then let $act(q_t)$, $gbl(q_t)$ and $enb_i(q_t)$ denote σ_s, q_g and Q_i, respectively.

We can then transform $\mathcal{T}$ into a TSM $\mathcal{M} = \langle \Sigma_s, Q_g, Q_T, \{\langle glb(q_{T0}), q_{T0} \rangle\}, \{\tau_1, \ldots, \tau_n\}, J, C \rangle$. For each τ_i, if $q_g \in enb_i(q_t)$ then

$$\tau_i(q_g, q_t) = \{\langle act(q_t'), glb(q_t'), q_t' \rangle \mid q_t' = \delta_T(q_t, \langle q_g, i \rangle))\}.$$

Otherwise, $\tau_i(q_g, q_t) = \emptyset$. The set J and C are known beforehand and depend only on the execution model $\mathcal{X}$.

3.3 Open System Synthesis

It is clear that the algorithm can be used to solve the modular synthesis problem in which the specification specifies acceptable sequences, both finite and infinite. The choice of the domain of the synthesized output (such as nSmWTSM for some integer n and m) can be encoded in the execution model $\mathcal{X}$.

For the basic open system synthesis, we recall proposition 1.3.5 and show only that there is a universal element for the domain of the environment model. We can construct, for any alphabet Σ_e and any set of global states Q_g, a TSM $\mathcal{M}_U = \langle \Sigma_e, Q_g, \{q_{l0}, q_{l1}\}, Q_g \times \{q_{l0}, q_{l1}\}, \{\tau\}, \emptyset, \emptyset\rangle$ where $\tau(q_g, q_{l0}) = \Sigma_e \times Q_g \times \{q_{l0}, q_{l1}\}$ and $\tau(q_g, q_{l1}) = \emptyset$, for any $q_g \in Q_g$. It is easy to show that $\mathcal{M}_U$ is the universal element of any nSmWTSM subclasses. For the weak open system synthesis, it is obvious that if ψ has a weak open system solution, then M_U is a solution. By proposition 1.3.5, we apply the algorithm to check whether the basic open system synthesis of ψ has a solution. The $\forall\exists-\mathsf{SYNTH}(\psi)$ and $\forall\exists-\mathsf{W}-\mathsf{SYNTH}(\psi)$ problems can be solved by transforming them into a weak or basic open system problem, according to proposition 1.3.2 for closed systems and 1.3.6 for open systems. For the modular synthesis problem under the open system semantics, we simply construct a TSM $\mathcal{E}||M_U$ and use it in the place of $\mathcal{E}$ in the algorithm.

The problems that have not been addressed are the maximal open system synthesis and the $\mathsf{ES}-\mathsf{SYNTH}(\cdot,\cdot)$ problem. We will consider each of them separately later.

3.4 Modeling

Various kinds of computational properties besides fairness can be modeled and solved within the framework. For example, the behaviors of the scheduler can be modeled into the execution model $\mathcal{X}$. If we consider each $\sigma \in \Sigma_s \cup \Sigma_e$ to be a variable assignment function $\sigma : Var \rightarrow Val$, for some variable set Var and a variable domain Val, then the characteristics of variables, such as being input, output or shared, can be encoded in either the environment model or the specification, with a careful choice of global states (Q_g). The separation of the global states (Q_g) from the local states (Q_{le}) and from the output alphabet (Σ_e) allows us to model unobservable variables and uncertainty.

Synchronous and asynchronous composition

Although the algorithm explicitly handles asynchronous compostion, it is easy to encode synchronous composition, or theoretically any kind of composition, in the execution and environment model.

Formally, given two TSMs $\mathcal{M}_a = \langle \Sigma_a, Q_g, Q_{la}, Q_{0a}, \mathcal{T}_a, J_a, C_a \rangle$. and $\mathcal{M}_b = \langle \Sigma_b, Q_g, Q_{lb}, Q_{0b}, \mathcal{T}_b, J_b, C_b \rangle$, the synchronous composition of $\mathcal{M}_a$ and $\mathcal{M}_b$, under a synchronization function $s : Q_g \times Q_g \to 2^Q_g$ is a TS $\mathcal{M}_a ||^s \mathcal{M}_b = \langle \Sigma, Q, Q_0, \mathcal{T}, J, C \rangle$ where

- Σ is $\Sigma_a \times \Sigma_b$.
- Q is $Q_g \times Q_a \times Q_b$.
- Q_0 is the set of all states $\langle q_g, q_a, q_b \rangle \in Q$ such that $\langle q_g, q_a \rangle \in Q_{0a}$ and $\langle q_g, q_b \rangle \in Q_{0b}$.
- $\mathcal{T}$ is $\{\tau' | \tau \in \mathcal{T}_a \cup \mathcal{T}_b\}$ and for each transition $\tau_a \in \mathcal{T}_a$, τ'_a is defined as follows:

$$\langle \langle \sigma_a, \sigma_b \rangle, \langle q'_g, q'_a, q'_b \rangle \rangle \in \tau'_a(\langle q_g, q_a, q_b \rangle)$$
$$\text{iff}$$
$$(\exists \tau_b \in \mathcal{T}_a) \langle \sigma_a, q_{ga}, q'_a \rangle \in \tau_a(q_g, q_a) \text{ and } \langle \sigma_b, q_{gb}, q'_b \rangle \in \tau_a(q_g, q_a) \text{ and } q'_g \in s(q_{ga}, q_{gb}),$$

- $J = \{\tau' | \tau \in J_a \cup J_b\}$ and $C = \{\tau' | \tau \in C_a \cup C_b\}$.

The role of the synchronization function is to allow us to represent the behaviors of the combined TS when both modules try to write to the same shared variables at the same time.

We can model synchronous composition and solve the synthesis problems under synchronous compostion with the same algorithm. The key idea is to consider an execution sequence of $\mathcal{M}_a ||^s \mathcal{M}_b$ as an alternating sequence of outputs of $\mathcal{M}_a$ and $\mathcal{M}_b$, and solve the synthesis problems as if asynchronous composition were used. First, such alternating execution has to be modeled with the execution model $\mathcal{X}$. Second, we have to modify the given environment module to reflect the fact that the global state that should be observed by the environment at each step is the previous global state, instead of the current global state printed out by the system.

Given the environment model $\mathcal{E} = \langle \Sigma, Q_g, Q_l, Q_0, \mathcal{T}, J, C \rangle$, we can construct $\mathcal{E}' = \langle \Sigma, Q_g, Q'_l, Q'_0, \mathcal{T}', J', C' \rangle$ as follows:

- $Q'_l = Q_l \times Q_g$,

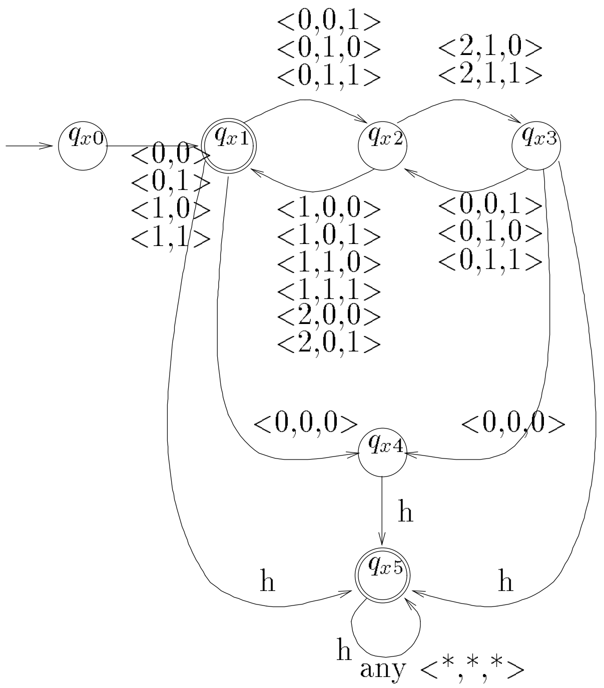

Figure 3.7: Execution model for synchronous composition

- $Q'_0 = \{\langle q_g, \langle q_l, q_g\rangle\rangle | \langle q_g, q_l\rangle \in Q_0\}$,
- $\mathcal{T}' = \{\tau' | \tau \in \mathcal{T}\}$ where each τ' is defined by:

$$\langle \sigma, q'_g, \langle q'_l, q'_g\rangle\rangle \in \tau'(q_{ga}, \langle q_l, q_g\rangle) \text{ iff } \langle \sigma, q_{gb}, q'_l\rangle \in \tau(q_g, q_l) \text{ and } q'_g \in s(q_{ga}, q_{gb}),$$

- $J = \{\tau' | \tau \in J_a \cup J_b\}$ and $C = \{\tau' | \tau \in C_a \cup C_b\}$.

The new environment model $\mathcal{E}'$ now becomes the input to the algorithm. It is easy to see that its alternating behaviors when combined with any module (the synthesized module) simulate the synchronous behaviors of $\mathcal{E}$ when combined with the same module.

Figure 3.7 shows an execution model with 2 transitions in the synthesized module. The first transition is a weakly fair transition. The good execution sequences are those alternating sequences that satisfy the weakly fair condition (when the modules are composed synchronously). As previously defined in the section 3.2.1, the first position of each tuple indicates whether it is the system module's turn (1) or the environment's turn (0), and the second and third position indicate whether the first and, respectively, the second transition are enabled (after the transition has been carried out). Note that since we map a single synchronous step into two steps in the execution model, the weakly fair condition now becomes: if the module (or the only transition in the module) is enabled in every even step from a certain point on, then it has to be taken infinitely often.

3.5 Maximal Open System Synthesis

Although finding the maximal solution for the modular synthesis and basic open system synthesis is still an open question, we can find a partial or an approximate solution, which maximizes the set of good projected behaviors.

Intuitively, a projected behavior is a projection of a behavior that the module being synthesized can observe and a good projected behavior is a projected behavior that is not a projection of an undesirable behavior.

Formally, we first define a projection relation $pr \subseteq Bhv \times ProjBhv$ where Bhv is the set $Q_g((\Sigma_e \cup \Sigma_s) \times Q_g)^{(\omega,*)}$ of finite and infinite behaviors, and $ProjBhv$ is the set $(Q_g\Sigma_s \times Q_g)^{(\omega,*)}$ of finite and infinite projected behaviors. A behavior $\alpha \in Bhv$

$$\alpha = q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} q_2 \ldots \quad [\ \xrightarrow{\sigma_{n-1}} q_n\ ]$$

and a projected behavior $\beta \in ProjBhv$

$$\beta = p_0 \xrightarrow{\sigma_{s0}} p'_0\ p_1 \xrightarrow{\sigma_{s1}} p'_1\ p_2 \xrightarrow{\sigma_{s2}} p'_2 \ldots \quad [\ p_k \xrightarrow{\sigma_{sk}} p'_k\ ]$$

satisfy the projection relation, i.e. $pr(\alpha, \beta)$ iff there is an increasing sequence of integers, $x_0 x_1 \ldots [x_k]$ such that

- $p_i = q_{x_i}$, $p'_i = q_{x_i+1}$ and $\sigma_{si} = \sigma_{x_i} \in \Sigma_s$,
- for any $\sigma_j \in (\Sigma_s - \Sigma_e)$, $(\exists i)\ j = x_i$.

For any $\beta \in ProjBhv$, let $Pr^{-1}(\beta)$ denote the set $\{\alpha \in Bhv \mid pr(\alpha, \beta)\}$.

Given an environment $\mathcal{E}$ and a specification ψ, a projected behavior β is a *good* projected behavior w.r.t. $\mathcal{E}$ and ψ iff $Pr^{-1}(\beta) \cap O[\![\mathcal{E}]\!]_{imp} \subseteq [\![\psi]\!]_{spec}$.

For a module $\mathcal{M}$, an environment $\mathcal{E}$ and a specification ψ, let $PB(\mathcal{M})$ be the set of projected behaviors of $\mathcal{M}||\mathcal{E}$:

$$PB(\mathcal{M}) = \{\beta \in ProjBhv \mid (\exists \alpha \in [\![\mathcal{M}||\mathcal{E}]\!]_{imp}) pr(\alpha, \beta)\},$$

and let $GPB(\mathcal{M})$ be the set of good projected behaviors of $\mathcal{M}||\mathcal{E}$ (w.r.t. $\mathcal{E}$ and ψ):

$$GPB(\mathcal{M}) = \{\beta \in PB(\mathcal{M}) \mid Pr^{-1}(\beta) \cap O[\![\mathcal{E}]\!]_{imp} \subseteq [\![\psi]\!]_{spec}\}.$$

Given an environment $\mathcal{E}$ and a specification ψ, our goal is to find a module $\mathcal{M}$ that is not only a solution to the modular synthesis problem of $\mathcal{E}$ and ψ but also maximizes the set $GPB(\mathcal{M})$.

Consider a Rabin tree automaton $\mathcal{D} = \langle \Sigma_D, Q_D, Q_{0D}, \delta_D, F_D \rangle$. Let $\mathcal{D}_q$ denote a variant of $\mathcal{D}$ in which the only initial state is q, i.e., $\mathcal{D}_q = \langle \Sigma_D, Q_D, \{q\}, \delta_D, F_D \rangle$. Let $Q_{ne} \subseteq Q_D$ be the set of states from which there is an infinite tree satisfying the Rabin acceptance condition F_D. The set Q_{ne} can be computed in deterministic time $O(n(nm)^{cm})$ for some constant c, where $n = |Q_D|$ and $m = |F_D|$ by using the algorithm from [VS85,Em85,PR89a] to check the non-emptiness of each $\mathcal{D}_q$, $q \in Q_D$.

Theorem 3.5.1 *If $r = q_0 q_1 \ldots$ is a path in a Q_D-tree T_{Q_D} of the automaton $\mathcal{D}$ that accepts a Σ_D tree, then $q_i \in Q_{ne}$, for all $i \geq 0$.*

Proof: Suppose r passes a state q and $q \notin Q_{ne}$. Then, D_q does not accept any tree. This implies that the subtree of T_{Q_D} which has q as its root does not satisfy the acceptance condition, contradicting the assumption that T_{Q_D} is accepted by $\mathcal{D}$. ∎

From the Rabin tree automaton $\mathcal{D}$, we can construct $Red(\mathcal{D}) = \langle \Sigma_D, Q_{ne}, Q_{0D} \cap Q_{ne}, \delta'_D, F'_D \rangle$ where δ'_D is simply the restriction of δ_D to the set Q_{ne} and $F'_D = \{(U \cap Q_{ne}, L \cap Q_{ne}) | (U, L) \in F_D\}$. It is obvious that $\mathcal{D}$ and $Red(\mathcal{D})$ accept the same tree language.

Now, we can proceed to the following theorem on finding the solution with maximal good projected behaviors.

Theorem 3.5.2 *For any given environment $\mathcal{E}$ and specification ψ, there is a module $\mathcal{M}_x$ such that $\mathcal{M}_x \models \mathsf{Mod\text{-}SYNTH}(\psi, \mathcal{E})$ and for any $\mathcal{M}'$, if $\mathcal{M}' \models \mathsf{Mod\text{-}SYNTH}(\psi, \mathcal{E})$, then $GPB(\mathcal{M}') \subseteq GPB(\mathcal{M}_x)$. Moreover, $\mathcal{M}_x$ is in 2STSM.*

Proof: We apply the synthesis algorithm (in section 3.2). Here, the execution model $\mathcal{X}$, as shown in Figure 3.4, encodes only a fairness constraint when there is only one strongly fair transition in the synthesized module. At one point in the algorithm, after co-determinizing and checking execution consistency, we have a tree automaton $\mathcal{D}$ on labeled trees. The trees branch on the alphabet Q_g, the global states observed by the module being synthesized. The labels on the nodes of a tree belong to the alphabet $\Sigma_s \times Q_g \times 2^{Q_g}$. Each label represents the output of the synthesized module, the new global state set by the module, and the set of global states which enable the module to make the next transition. We compute $Red(\mathcal{D})$ that preserves the tree language accepted by $\mathcal{D}$. Now, reconsider $Red(\mathcal{D})$ as a Rabin automaton on the alphabet $Q_g \times \Sigma_s \times Q_g \times 2^{Q_g}$. The Rabin automaton $Red(\mathcal{D})$ accepts a sequence iff it is a path in a tree accepted by the tree automaton $\mathcal{D}$. Use the algorithm in chapter 2 (for closed systems) to construct a 2STS $\mathcal{M}$ that generates the sequences from the alphabet $Q_g \times \Sigma_s \times Q_g \times 2^{Q_g}$. Remove any inconsistency between the set of enabling global states (2^{Q_g}) and the following observed global state (Q_g). We can interpret $\mathcal{M}$ as a 2STSM on the alphabet Σ_s.

Suppose a behavior β,

$$\beta = p_0 \xrightarrow{\sigma_{s0}} p'_0 \; p_1 \xrightarrow{\sigma_{s1}} p'_1 \; p_2 \xrightarrow{\sigma_{s2}} p'_2 \ldots \quad [\; p_k \xrightarrow{\sigma_{sk}} p'_k \;]$$

is a good projected behavior of some module $\mathcal{M}'$ and $\mathcal{M}'$ is a solution to the modular synthesis of $\mathcal{E}$ and ψ. We can show that β corresponds to a path in a tree accepted by the tree automaton $\mathcal{D}$. Consider a behavior α of $\mathcal{M}'$ such that $pr(\alpha, \beta)$. Suppose q_{li} is the local state of $\mathcal{M}'$ before the transition $p_i \xrightarrow{\sigma_{si}} p'_i$ is taken. Let $Q_i \subseteq Q_g$ be the set of global states that enable $\mathcal{M}'$ at q_{li}, that is,

$$Q_i = \{q_g \in Q_g \,|\, \text{for some } \tau \in \mathcal{T},\ \tau(q_g, q_{li}) \neq \emptyset\},$$

where $\mathcal{T}$ is the set of transitions of $\mathcal{M}'$. If β is finite, then $Q_{k+1} = \emptyset$. Let $\mathcal{M}'_i$ be a TSM that is exactly the same as $\mathcal{M}'$ except that the set $Q_g \times \{q_{li}\}$ is the set of initial states. From theorem 3.1.5, there exists a 1STSM $\mathcal{N}_i$ such that for any $\mathcal{E}'$, $\mathcal{L}_{\mathcal{N}_i||\mathcal{E}'} \subseteq \mathcal{L}_{\mathcal{M}_i||\mathcal{E}'}$. Let T_i be the labeled subtree (whose paths are projected behaviors) induced by $\mathcal{N}_i$.

Now, we recursively define T'_i, as illustrated in Figure 3.8 to be the same as T_i except for the branch on p_i leading to a node labeled by $\langle \sigma_{si}, p'_i, Q_i \rangle$, which becomes the root of a subtree T'_{i+1}. It is clear that β corresponds to a path in T'_0. The path that does not correspond to β satisfies the acceptance condition of D because it will eventually follow a path in some T_i and we know that $\mathcal{N}_i$ is in 1STSM and $\mathcal{M}$ is a solution to the synthesis problem of $\mathcal{E}$ and ψ. For the path that corresponds to β, we consider two cases. First, if β is infinite, then we know that for any α such that $pr(\alpha, \beta)$, α corresponds to a fair computation. We know that α corresponds to a fair computation because the execution

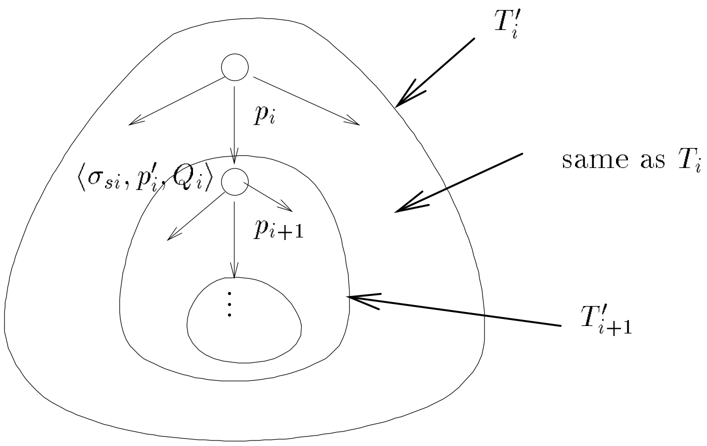

Figure 3.8: the tree T_i' with the projected behavior β embedded as a path in the tree

model$\mathcal{X}$ encodes the case when there is only one transition in the synthesized module, and since β is infinite, the transition must be scheduled infinitely often. Second, if β is finite, we can pick the enabling condition Q_{k+1} to be the empty set. Thus, for any α such that $pr(\alpha, \beta)$, α also corresponds to a fair computation in this case. Therefore, for both cases, to satisfy the acceptance condition of D, we have to show that for any $\alpha \in O[\![\mathcal{E}]\!]_{imp}$, if $pr(\alpha, \beta)$, then α satisfies ψ. We know that this is true because we assume that β is a good projected behavior. ∎

3.6 ES Synthesis

For the ES synthesis problem, our goal is to find an implementation such that when composed with any environment that satisfies the environment assumption, the overall behaviors of the whole system satisfies the specification. It is clear that this problem is closedly related to the problem of finding the maximal solution for open system synthesis. If we have a method of solving the maximal open system synthesis and the solution is unique, then the ES synthesis problem will be reduced to simply a modular synthesis problem. Unfortunately, we still do not have such method. However, there are certain cases where we can solve the ES synthesis problem.

Environment assumption = safety

An environment assumption ψ_E defines a language $\mathcal{L}_{\psi_E} \subseteq (Q_g \times (\Sigma_e \cup \Sigma_s \cup \{h\}))^\omega$. An environment assumption ψ_E is a *safety* property iff for any infinite word $q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} q_2 \xrightarrow{\sigma_2} \ldots \notin \mathcal{L}_{\psi_E}$, there exists a finite prefix $q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \ldots q_i \xrightarrow{\sigma_i}$ such that for any $w \in (Q_g \times (\Sigma_e \cup \Sigma_s))^\omega$. $q_0 \xrightarrow{\sigma_0} \ldots q_i \xrightarrow{\sigma_i} w \notin \mathcal{L}_{\psi_E}$. Note that the condition restricts infinite behaviors (or words without any occurrence of h) only. Finite behaviors do not affect whether the language is safety property.

The following theorem states that any safety property has a unique maximal open system implementation.

Theorem 3.6.1 *If ψ_E is a safety property, then there exists a module $\mathcal{E}$ such that $O[\![\mathcal{E}]\!]_{imp} \subseteq \mathcal{L}_{\psi_E}$ and for any $\mathcal{E}'$, if $O[\![\mathcal{E}']\!]_{imp} \subseteq \mathcal{L}_{\psi_E}$, then $O[\![\mathcal{E}']\!]_{imp} \subseteq O[\![\mathcal{E}]\!]_{imp}$.*

Proof: First, suppose $\mathcal{L}_{\psi_E}$ is represented as a deterministic Rabin automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \langle$ where the alphabet Σ is $Q_g \times (\Sigma_e \cup \Sigma_s \cup \{h\})$. There is no special reason why we assume that $\mathcal{L}_{\psi_E}$ is represented as an DRA. We can start with other representations, and follow the general ideas demonstrated in the constructions below. Not surprisingly, the complexity of the constructions depends on the representation we choose.

A path $q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \ldots [q_k]$, where $q_i \in Q$ and $\sigma_i \in \Sigma$ for all i, is called a h-free path if it does not pass through any $\langle q_g, h \rangle \in \Sigma$, i.e., $\sigma_i \notin Q_g \times \{h\}$ for all i. Let $Q_F \subseteq Q$ be the subset of all states s such that there is an infinite h-free path from s that satisfies the acceptance condition F. Suppose $q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \ldots$ is a run of $\mathcal{A}$, i.e., for any i, $q_i \in Q$, $\sigma_i \in \Sigma$, $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, \sigma_i)$. If $q_i \in Q_F$ for all i and $\sigma_i \notin Q_g \times \{h\}$, we can show that

$\sigma_0\sigma_1\ldots \in \mathcal{L}_{\psi_E}$. Suppose $\sigma_0\sigma_1\ldots \notin \mathcal{L}_{\psi_E}$. Since ψ_E is a safety property, then there must exist an integer i such that $\sigma_0\sigma_1\ldots\sigma_i w \notin \mathcal{L}_{\psi_E}$ for all $w \in Q_g \times (\Sigma_e \cup \Sigma_s)$. That means, $q_i \notin Q_F$, leading to a contradiction to an earlier assumption. A state $q \in Q$ is *s-reachable* from another state $q' \in Q$ iff there is a path $q \xrightarrow{\sigma_1} q_1 \ldots \xrightarrow{\sigma_i} q'$ for some $i \geq 0$, and $\sigma_j \in Q_g \times \Sigma_s$ for all $0 \leq j \leq i$.

Next, construct a graph $G = \langle \Sigma', Q', q_0', \delta' \rangle$ where

- $\Sigma' = \Sigma_g \times (\Sigma_e \cup \{h\})$,
- $Q' = 2^Q$,
- $q_0' = \{q_0\}$,
- for any $P \in Q'$ and any $\sigma \in \Sigma'$,

$$\delta'(P, \sigma) = \{q' \in Q \mid (\exists\, q \in P) q' \text{ is } s\text{−reachable from } \delta(q, \sigma)\}.$$

A node $P \in Q'$ of the graph G is called a *bad* node iff either $P \not\subseteq Q_F$ or there exists some $q_g \in Q_g$ such that for all $\sigma' \in (\Sigma_e \cup \{h\})$, $\delta'(P, \langle q_g, \sigma' \rangle)$ is a bad node.

Finally, construct a 1TSM $\mathcal{E} = \langle \Sigma_e, Q_g, Q_l, Q_{0e}, \{\tau\}, J, \{\tau\} \rangle$ where

- $Q_l = Q' \times \{0, 1\}$,
- $Q_{0e} = \{\langle q_g, \langle \{q_0\}, 0 \rangle \rangle \mid (\exists \sigma \in (\Sigma_e \cup \Sigma_s \cup \{h\}))\delta(q_0', \langle q_g, \sigma \rangle) \in Q_F\}$,
- For any $P \in Q'$, $q_g \in Q_g$ and $\sigma_e \in \Sigma_e$,

$$\tau(q_g, \langle P, 0 \rangle) = \begin{cases} \emptyset & \text{if } \delta'(P, \langle q_g, \sigma_e \rangle) \text{ is a bad node} \\ \{\delta'(P, \langle q_g, \sigma_e \rangle)\} \times \{0, 1\} & \text{otherwise} \end{cases}$$

$$\tau(q_g, \langle P, 1 \rangle) = \begin{cases} \emptyset & \text{if } \delta'(P, \langle q_g, h \rangle) \text{ is not a bad node} \\ \tau(q_g, \langle P, 0 \rangle) & \text{otherwise.} \end{cases}$$

It is easy to show that $\mathcal{E}$ is a basic open system solution of ψ. Any computation of $\mathcal{E}$ corresponds to a run r of $\mathcal{A}$. We know from the construction that r passes through the states in Q_F only and from what we have proved earlier, we can conclude that r must be accepted by $\mathcal{A}$.

Now, suppose $\mathcal{E}'$ is also a basic open system solution of ψ. We can show that any open system behavior r of $\mathcal{E}$ must pass though the states in Q_F only. If $\mathcal{E}'$ is scheduled to run infinitely often on r, then it is clear from the definition that r never visits any state outside Q_F.

For the other case, when $\mathcal{E}'$ halts at some point on r, we only have to consider the case when $\mathcal{E}'$ stops after r passes through some state outside Q_F. However, we can show that such behavior is not possible. Consider any behavior $\alpha \in \mathcal{L}_{\psi_E}$ of the form $q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \ldots q_i \xrightarrow{h} \ldots$. We can show that if α passes through some state outside Q_F, i.e. $q_0 \xrightarrow{\sigma_0} \ldots q_i \xrightarrow{\sigma} w \notin \mathcal{L}_{\psi_E}$ for all $\xrightarrow{\sigma} w \in ((\Sigma_e \cup \Sigma_s) \times Q_g)^\omega$, then $\alpha \notin O[\![\mathcal{E}']\!]_{imp}$. The reason is that the environment of $\mathcal{E}'$ can always keep extending from q_i onward and making it into an infinite sequence, regardless of what $\mathcal{E}'$ does. From the assumption, any infinite sequence of the form $q_0 \xrightarrow{\sigma_0} \ldots q_i \xrightarrow{\sigma} w$ does not satisfy ψ_E. Therefore, if α were in $O[\![\mathcal{E}']\!]_{imp}$, it would lead to a contradiction with our assumption that $\mathcal{E}'$ satisfies ψ_E.

Therefore, we can conclude that any open system behavior r of $\mathcal{E}'$ must pass though the states in Q_F only. Next, we can show that the projection of r must never pass a bad state of G; otherwise, there exists another behavior of $\mathcal{E}$ with the same projection that does not satisfy ψ. Finally, we can prove that r is also an open system behavior of $\mathcal{E}$. If $\mathcal{E}$ is scheduled infinitely often on r, we can select a computation of $\mathcal{E}$ which passes through the local states in $Q_l = Q' \times \{0\}$. If $\mathcal{E}$ is scheduled on finitely many times, then at the last point on r where $\mathcal{E}$ is schedule, we follow the transition of $\mathcal{E}$ to a local state in $Q' \times \{1\}$. The definition of τ on local states $Q' \times \{1\}$ guarantees that $\mathcal{E}$ can halt at that point. ∎

The above theorem implies that, when the environment assumption is a safety property, we can find such module $\mathcal{E}$ and apply the algorithm in section 3.2. to solve the ES synthesis problem.

Limiting the domain of the environment

If we limit the domain of the environment to some particular set of transition system modules that can be represented by the execution model $\mathcal{X}$, then we can apply the synthesis algorithm to solve the ES synthesis problem. In other words, the goal of the ES synthesis problem here is to find an implementation such that when composed with any environment that *belongs to some limited set of TSMs* and satisfies the environment assumption ψ_E, the overall behaviors of the compostion of the environment and the synthesized module satisfies the specification ψ_S. One of such sets of TSMs that can be represented by some execution model $\mathcal{X}_E$ is the class of TSMs that have n transitions among which there are k strongly fair and l weakly fair transitions, for some particular integers n, k and l.

We start by applying the algorithm in section 3.2. to solve the basic open system synthesis of the environment assumption ψ_E with the execution model $\mathcal{X}_E$. In the middle of the algorithm, we construct a tree automaton $\mathcal{D}$ recognizing all labeled trees of projected behaviors that can guarantee that ψ_E is satisfied in the corresponding complete behaviors. Even though we do not have a method to construct the unique maximal solution from $\mathcal{D}$, we can compute $Red(\mathcal{D})$ and interpret it as a usual Rabin automaton, instead of a Rabin tree automaton. $Red(\mathcal{D})$ can be used as the environment model input to the algorithm in section 3.2., but now, with ψ_S as the specification and a different execution model $\mathcal{X}$. The execution model $\mathcal{X}$ encodes not only the fairness constraint of the module being synthesized but also the fairness constraint of the environment. The output (the module synthesized by the algorithm) of is a solution of the ES synthesis problem.

Chapter 4

Direct Synthesis From Temporal Logic

In this chapter, we study the direct synthesis of open systems from a temporal logic specification. We present two algorithms, one for realizability checking and the other for synthesis. The specification language we study is the Extended Temporal Logic (ETL) described in [Wo83]. We also introduce a scheduling variable μ following the approach in [BKP84]. Although useful for expressing specifications and for extending the algorithm to handle sequential composition, μ is not essential to the algorithms. The realizability-checking algorithm is based on the tableau decision procedure described in [Wo85]. Given a specification, the first algorithm checks for *strong* realizability under fairness and random environment assumptions and generates a structure called a *realizability graph*. If it is realizable, the synthesis algorithm takes the generated realizability graph and produces a program which satisfies the specification. The transitions in the generated program may be labeled weakly or strongly fair as necessary. Since the realizability-checking algorithm is a tableau-based algorithm, it manipulates only formulas of linear temporal logic, which are subformulas of the original specification.

4.1 Definitions

A [infinite] *behavior* σ over a state space Σ is a pair $\langle \sigma_v, \sigma_s \rangle$ of two equal-length [infinite] sequences: a sequence of *states* $\sigma_v = s_0 s_1 s_2 \ldots$ where $s_i \in \Sigma$ and a *scheduling* sequence $\sigma_s = a_0 a_1 a_2 \ldots$ where $a_i \in \{0, 1\}$. We denote the set of all infinite behaviors over Σ

by $Bhv(\Sigma)$, or Bhv if Σ is clear from the context, and the set of all finite behaviors by $Bhv_{fin}(\Sigma)$.

We can represent a behavior $\langle s_0 s_1 s_2 \ldots, a_0 a_1 a_2 \ldots\rangle$ pictorially as

$$\stackrel{a_0}{\rightarrow} s_0 \stackrel{a_1}{\rightarrow} s_1 \stackrel{a_2}{\rightarrow} s_2 \ldots$$

The intended meaning is that the move from s_i to s_{i+1} is caused by the environment if $a_{i+1} = 0$, and by the system if $a_{i+1} = 1$. Since we always assume that the environment chooses the initial state, we require that the scheduling sequence always begins with 0, i.e., $a_0 = 0$.

Given a behavior $\sigma = \langle s_0 s_1 s_2 \ldots, a_0 a_1 a_2 \ldots\rangle$, we write $State(i, \sigma)$ to denote s_i and $Sched(i, \sigma)$ to denote a_i. If $\sigma = \langle \sigma_v, \sigma_s \rangle$, then $\sigma|_i$ denotes a behavior $\langle \sigma_v|_i, \sigma_s|_i \rangle$ where $\sigma_v|_i$ $[\sigma_s|_i]$ is the prefix of σ_v $[\sigma_s]$ of length i.

Let $\Pi^0 : Bhv_{fin} \mapsto \Sigma^*$ be a function that maps a finite behavior σ to a subsequence of states which are caused by the system, namely, all $State(i, \sigma)$ where $Sched(i, \sigma) = 1$. Let $\Pi^1 : Bhv_{fin} \mapsto \Sigma^*$ be a function that maps a finite behavior σ to a subsequence of states which are observed by the system (precede a system state), that is, all $State(i, \sigma)$ where $Sched(i + 1, \sigma)$ is defined and $Sched(i + 1, \sigma) = 1$, or $i + 1$ is the length of σ.

A *computer* $f : \Sigma^* \times \Sigma^* \mapsto \Sigma$ is a partial function which takes a history of all the states the system caused and all the states the system observed and selects a state as the next move of the system. A *run* of a computer f is an infinite behavior such that for all i, if $Sched(i, \sigma) = 1$ then $f(\Pi^0(\sigma|_i), \Pi^1(\sigma|_i))$ is defined and equal to $State(i, \sigma)$. Therefore, a behavior is a run of a computer if every system move is the result of f computed with the information regarding the system's own moves and all the moves the system has observed in the past.

A run σ of f is *weakly fair* iff for all j, if $f(\Pi^0(\sigma|_i), \Pi^1(\sigma|_i))$ is defined for all $i \geq j$, then $Sched(k, \sigma) = 1$ for some $k \geq j$, i.e., if f is continuously enabled beyond a certain point, it has to be taken eventually. Similarly, a run σ is *strongly fair* iff for all j, if for all $j' \geq j$ there exists $i \geq j'$ such that $f(\Pi^0(\sigma|_i), \Pi^1(\sigma|_i))$ is defined, then $Sched(k, \sigma) = 1$ for some $k \geq j$.

Let $Run_{sf}(f)$ be all possible strongly fair runs and $Run_{wf}(f)$ all the weakly fair runs of the computer f. A set B of behaviors is *realizable* (under fairness and random environment assumptions) iff there exists a computer f such that $Run_{sf}(f) \subseteq B$. If only weak fairness is assumed, B is realizable iff there exists a computer f such that $Run_{wf}(f) \subseteq B$.

4.2 Preliminaries

4.2.1 Specification Language

The specification language studied here is Extended Temporal Logic (ETL) augmented with a special predicate μ. The use of ETL and μ is not necessary for the realizability-checking and synthesis algorithms. Clearly, the algorithms can handle any subset of the language, including ordinary propositional temporal logic specifications without μ. Adding μ to the language is necessary, however, to express some common forms of specifications such as mutual exclusion. Without μ, we would have to separate the environment assumption and the system property. With μ, the whole specification can be expressed in a single formula.

In ETL, there are infinitely many temporal operators. Each corresponds to a non-terminal symbol of a right-linear grammar. A right-linear grammar G is a tuple (V_N, V_T, P) such that

- $V_N = \{\mathcal{G}_1, \ldots, \mathcal{G}_m\}$ is a finite set of non-terminal symbols.
- $V_T = \{t_1, \ldots, t_n\}$ is a finite set of terminal symbols.
- P is a finite set of production rules of the forms $\mathcal{G}_i \to t_j$ or $\mathcal{G}_i \to t_j \mathcal{G}_k$ where $\mathcal{G}_i, \mathcal{G}_k \in V_N$ and $t_j \in V_T$.

For each non-terminal symbol $\mathcal{G}_i$, the corresponding temporal operator $\mathcal{G}_i(\phi_1, \ldots, \phi_n)$ has exactly n arguments (n is the number of terminal symbols).

Given a set $\mathcal{P}$ of propositions and a truth-value assignment function $\pi : \Sigma \mapsto 2^{\mathcal{P}}$, the semantics of a formula on an infinite behavior σ is defined as follows:

- $\sigma \models \phi$ iff $\langle \sigma, 0 \rangle \models \phi$.
- $\langle \sigma, i \rangle \models p$ iff $p \in \pi(\mathit{State}(i, \sigma))$, for any proposition $p \in \mathcal{P}$.
- $\langle \sigma, i \rangle \models \mu$ iff $\mathit{Sched}(i, \sigma) = 1$.
- $\langle \sigma, i \rangle \models \bigcirc\phi$ iff $\langle \sigma, i+1 \rangle \models \phi$.
- $\langle \sigma, i \rangle \models \mathcal{G}(\phi_1, \ldots, \phi_n)$ iff there is a word (finite or infinite) $w = t_{n_0} t_{n_1} t_{n_2} \ldots$ (each $t_{n_j} \in V_T$), generated by $\mathcal{G}$, and for all $j \geq 0$, $\langle \sigma, i+j \rangle \models \phi_{n_j}$.
- Other cases ($\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, and $\neg\phi$) are standard.

Clearly, any formula ϕ defines a set of infinite behaviors B which satisfy the formula, i.e., $\sigma \models \phi$ iff $\sigma \in B$. Therefore, we define a specification to be realizable if the corresponding set of behaviors is realizable.

4.2.2 Elementary Formulas

A formula is called *elementary* if it is either

- an atomic formula, i.e., an atomic proposition (including μ) or its negation, or
- a next formula, i.e., a formula that has $\bigcirc$ as its main connective.

4.2.3 Decomposition Rules

The following decomposition rules are used in the tableau graph construction algorithm to decompose non-elementary formulas. The meaning of a decomposition rule is that in order to satisfy the formula on the left hand side, one of the sets on the right hand side must be satisfied.

- $(\phi_1 \vee \phi_2) \Longrightarrow \{\{\phi_1\}, \{\phi_2\}\}$
- $(\phi_1 \wedge \phi_2) \Longrightarrow \{\{\phi_1, \phi_2\}\}$
- $\neg(\phi_1 \vee \phi_2) \Longrightarrow \{\{\neg\phi_1, \neg\phi_2\}\}$
- $\neg(\phi_1 \wedge \phi_2) \Longrightarrow \{\{\neg\phi_1\}, \{\neg\phi_2\}\}$
- $(\neg\neg\phi) \Longrightarrow \{\{\phi\}\}$
- $(\neg \bigcirc \phi) \Longrightarrow \{\{\bigcirc\neg\phi\}\}$
- For an ETL grammar operator $\mathcal{G}(\phi_1, \ldots, \phi_n)$ with grammar productions of the form: $\mathcal{G} \rightarrow t_{a_i}\ \mathcal{G}_{b_i}$ where $1 \leq i \leq l$ is the index of the production rules of $\mathcal{G}$, $t_{a_i} \in V_T$ and $\mathcal{G}_{b_i} \in V_N$ (which may or may not be present), we have the following decomposition rules:

$$\mathcal{G}(\phi_1, \ldots, \phi_n) \Longrightarrow \bigcup_{1 \leq i \leq l} \{\{\phi_{a_i}, \bigcirc\mathcal{G}_{b_i}(\phi_1, \ldots, \phi_n)\}\}$$

$$\neg\mathcal{G}(\phi_1, \ldots, \phi_n) \Longrightarrow \{\bigcup_{1 \leq i \leq l} \{\neg\phi_{a_i} \vee \bigcirc\neg\mathcal{G}_{b_i}(\phi_1, \ldots, \phi_n)\}\}$$

4.2.4 Tableau Graph

Before we proceed to describe the realizability-checking algorithm, we will briefly explain a tableau graph construction similar to that in [Wo85]. A tableau graph is a directed graph in which each node n is labeled with a set of formulas, denoted by $\Phi(n)$.

- A node n in a tableau graph is called a *state* node iff $\Phi(n)$ contains only elementary formulas.
- A node n is *environment-compatible* iff $\mu \notin \Phi(n)$.
- Similarly, a node n is *system-compatible* iff $\neg\mu \notin \Phi(n)$.

Given a formula $\hat{\psi}$ to be checked for satisfiability, the tableau graph for $\hat{\psi}$ is created as follows:

First,

1. create a node *(root)* and label it with $\{\hat{\psi}\}$.

Repeatedly apply steps 2 and 3.

2. If a node n, with no successor, contains a non-elementary formula ϕ in its label $\Phi(n)$, and if the decomposition rule for ϕ is $\phi \Longrightarrow \{S_1, \ldots, S_t\}$, then for each set of formulas S_i, create a successor of n and label it with $(\Phi(n) - \{\phi\}) \cup S_i$. However, if there is a node with the same label already, then just connect n to the existing node.
3. For a state node n with label $\Phi(n)$, create (if no duplication occurs) a successor of n and label it with $\{\phi \mid \bigcirc\phi \in \Phi(n)\}$.

Finally,

4. Remove all inconsistent nodes (the nodes containing a proposition p and its negation $\neg p$).

A loop in a tableau graph is called a *self-supporting* loop if for any state node n in the loop, there is a finite path in the loop starting from n such that all formulas of the form $\bigcirc\neg\mathcal{G}(\ldots)$ in $\Phi(n)$ are *fulfilled* on the path. A formula $\bigcirc\neg\mathcal{G}(\ldots)$ with the decomposition rule,

$$\neg\mathcal{G}(\phi_1, \ldots, \phi_n) \Longrightarrow \{ \bigcup_{1 \le i \le l} \{\neg\phi_{a_i} \vee \bigcirc\neg\mathcal{G}_{b_i}(\phi_1, \ldots, \phi_n)\}\}$$

is *fulfilled* at a state n if the next state n' on the path contains a term from each of the disjunctions of the decomposition rule and if the term is $\bigcirc\neg\mathcal{G}_{b_i}(\ldots)$ then it is also fulfilled at n' (i.e. at the next state down the path).

4.2.5 Maximally Consistent Subsets

Given a set of state nodes N, a subset $N_{mcs} \subseteq N$ is *maximally consistent* if both of the following conditions are satisfied:

- **Consistent:** It is not the case that for some proposition p other than μ and for some nodes $n_1, n_2 \in N_{mcs}$, both $p \in \Phi(n_1)$ and $\neg p \in \Phi(n_2)$. In other words, the union of all the observable atomic formulas (which are all atomic formulas except μ and $\neg\mu$) in the labels of the nodes in N_{mcs} is consistent.

- **Maximal:** There is no other subset $N' \subseteq N$ such that N' satisfies the above condition (consistent) and $N_{mcs} \subset N'$.

4.2.6 Maximally Negation-Consistent Subsets

For a set of state nodes N, a subset $N_{mncs} \subseteq N$ is *maximally negation-consistent* if both of the following conditions are satisfied:

- **Negation-consistent:** There exists a function f which maps each node $n \in N_{mncs}$ to an atomic formula $f(n) \in \Phi(n)$ which is not μ or $\neg\mu$, and the set $P = \{\neg f(n) \mid n \in N_{mncs}\}$ is consistent. The set P is called the *falsifying* set for N_{mncs}.

- **Maximal:** There is no other subset $N' \subseteq N$ such that N' satisfies the above condition (negation-consistent) and $N_{mncs} \subset N'$.

4.2.7 Realizability Graph

The structure created by the realizability-checking algorithm is called a *realizability graph.* A realizability graph is a directed bipartite graph $(V_s, V_n, E_{sn}, E_{ns})$ where

- V_s is a set of nodes called *R-state* nodes and labeled by a *node-label* which is a set of tableau graph nodes and a *write-label* which is a set of atomic formulas.

- V_n is a set of nodes called *R-non-state* nodes and labeled by a node-label.

- V_{sn} is a set of links from R-state nodes to R-non-state nodes.
- V_{ns} is a set of links from R-non-state nodes to R-state nodes.

4.2.8 Embedding

An increasing sequence $d_0 \dots d_l$ of integers is an *embedding* of a path [loop] $n_0 \dots n_k$ in a tableau graph into a path [loop] $v_0 \dots v_l$ in a realizability graph if both of the following conditions hold:

- for all $0 \leq i \leq l$, n_{d_i} is in the node-label of v_i.
- for all n_j, if $j \neq d_i$ for all $0 \leq i \leq l$, then n_j is environment-compatible.

It is straightforward to extend the definition to allow the embedding of an infinite path in a tableau graph into a (finite or infinite) path in a realizability graph.

4.3 Realizability-Checking Algorithm

The key idea in the algorithm is that the realizability graph represents a game between the system and the environment in which the environment can make any finite number of moves after a system's move. This is represented by the alternate levels of R-state and R-non-state nodes. Given a formula ψ to be tested for realizability, the algorithm construct a tableau graph for the negation of ψ. To "win the game", the environment must try to force the execution to stay on a path in the tableau graph which falsifies ψ; whereas the system must try to push the execution out of such path.

We start constructing the realizability graph from an R-state node which contains the root node n_{root} of the tableau graph. Since the environment can make any number of moves, it may try to follow any path in the tableau graph from n_{root}. Without the complete knowledge of all the moves the environment makes, the system cannot determine which path the environment has taken. It can only use the information from the state it observes when it is scheduled to run, to determine a *set* of all state nodes accessible from n_{root} the path might have led into. In the worst case, such s set will be a maximally consistent subset of all accessible state nodes. Therefore, we construct an R-non-state successor of the R-state node, for each maximally consistent subset. For its own move, the system must try to push the execution out of any path which the environment might follow (and win) afterward.

The best move that the system can possibly make is to falsify as many successor nodes of the nodes in the node-label of the R-non-state node and in essence, to limit the possible paths left for the environment to follow. This is the reason why we compute the maximally negation-consistent subsets and the falsifying set of atomic formulas. The remaining nodes which are not falsified can be computed by subtracting the maximally negation-consistent subsets from the set of all successors of the nodes in the R-non-state node. For each best move possible, we create an R-state successor of the R-non-state successor, put the remaining nodes in its node-label and continue expanding the realizability graph from the new R-state node.

In the algorithm, at each R-state node v_s, we compute a set *Disabled* by collecting all state nodes in the labels of every deleted R-non-state successor of v_s. When an R-non-state successor v_{ns} is deleted, it means the system will not be able to satisfy the specification by making a transition from v_s through v_{ns}. Therefore, we should consider such a transition "disabled". As a result, we put every state node in the deleted R-non-state node into the set *Disabled* because the environment can choose to move into some states in which the transition through the deleted R-non-state node is disabled.

Finally, we also have to check at each R-state node that the environment cannot win by remaining in a loop containing disabled state nodes.

4.3.1 Main procedure

1. First, create a tableau graph *Glb* for the formula $\neg\psi$ where ψ is the formula to be tested for realizability.

2. Create an R-state node *(root)* and label it with the set $\{n_{root}\}$ where n_{root} is the root node of *Glb*.

3. Call the subroutine *Expand*, passing the root node as its parameter, to expand the realizability graph in a depth-first fashion.

4. Finally, check if the root node of the final realizability graph is deleted. If it is not deleted, then the formula ψ is realizable. Otherwise, it is unrealizable.

4.3.2 Subroutine *Expand* (Realizability Graph Construction)

Given an R-state node v_s with a node-label $L(v_s)$, expand the realizability graph as follows:

1. If $L(v_s)$ is empty, then do nothing and return.
2. Let N_{acc} be the set of all state nodes n_k accessible from some $n_0 \in L(v_s)$ through some path $n_0 \dots n_k$ in *Glb* such that for all $0 < i \leq k$, n_i is an environment-compatible node.
3. If there is a node in N_{acc} which contains only atomic formulas, then delete v_s and return from *Expand*.
4. Set *Disabled* to be the empty set.
5. For each maximally consistent subset N_{mcs} of N_{acc},
   (a) Create an R-non-state node v_{ns} as a successor of v_s and label v_{ns} by N_{mcs}.
   (b) Let N' be the set of all system-compatible state nodes n_k accessible from some $n_0 \in N_{mcs}$ through a path $n_0 \dots n_k$ where for all $0 < i < k$, n_i is not a state node.
   (c) For each maximally negation-consistent subset N_{mncs} of N' and the corresponding falsifying set P of atomic formulas,
      i. Create an R-state node as a successor of v_{ns} and label it by a node-label $N' - N_{mncs}$ and a write-label P. Then, recursively call *Expand* on the new node.
      ii. However, if there is an R-state node v'_s with the same node-label and write-label, and if, in addition, the node v'_s itself is marked, *"satisfied"*, then connect v_{ns} to v'_s. If v'_s is not marked *"satisfied"*, then check whether there exists a self-supporting loop in *Glb* that can be embedded into the loop $v_s \dots v'_s$. If there is no such loop in *Glb*, connect v_{ns} to v'_s.
   (d) If there is no successor to v_{ns}, delete v_{ns} and add all the nodes in N_{mcs} (the node-label of v_{ns}) to the set variable *Disabled*.
6. Check if there is a self-supporting loop in *Glb* which is accessible from a node in $L(v_s)$ through a path consisting only of environment-compatible nodes, and all state nodes in the loop are environment-compatible and in the set *Disabled*. If there is, then delete v_s. Otherwise, mark v_s *"satisfied"* and return.

   If only weak fairness is allowed in the definition of realizability that we are checking, we only have to look for a self-supporting loop with *at least one* state node in *Disabled*.

4.4 Synthesis Algorithm

To simplify the presentation, we choose to represent the synthesized module by a labeled finite automaton. A *module automaton* is a tuple $\langle S, \delta, s_0, l\rangle$ where S is a finite set of states, $\delta : S \times 2^{\mathcal{P}} \mapsto 2^S$ the transition relation, $s_0 \in S$ the initial state and $l : S \mapsto 2^{\mathcal{P}}$ the labeling function. A run r is a sequence (finite or infinite) of states from S starting with s_0. We will write $r[k]$ to denote the k-th state in the sequence r and $|r|$ to denote the length of r. A behavior σ with a truth-value assignment $\pi : \Sigma \mapsto 2^{\mathcal{P}}$ is accepted by the automaton iff there is a run r such that for every k, if $r[k+1]$ is defined then $\pi((\Pi^0(\sigma))[k]) = l(r[k+1])$ and $r[k+1] \in \delta(r[k], \pi((\Pi^1(\sigma))[k]))$. With weak fairness, a behavior σ is accepted iff in addition to the previous conditions, if r is finite then for some j, there exist infinitely many $i \geq j$, such that $\delta(r[|r|], \pi(State(i, \sigma))) = \emptyset$. A similar acceptance condition can be defined for the case of strong fairness.

Given a realizability graph, we will synthesize a module automaton which implements the specification. First, for each R-state node v, create a state $s_v \in S$ for the automaton. The initial state s_0 corresponds to the root node of the realizability graph. For the labeling function l, let $l(s_v)$ be the write-label of v.

For each s_v and each $x \in 2^{\mathcal{P}}$, recall the set N_{acc} of all state nodes accessible from the nodes in the node-label $L(v)$ of the R-state node v. Find the largest subset $N \subseteq L(v)$ such that for every state node $n \in N$, all the propositions $p \in \Phi(n)$ are in x and there is no $p \in x$ such that $\neg p \in \Phi(n)$. An R-non-state successor v_{ns} of v is said to *cover* x iff $N \subseteq L(v_{ns})$ where $L(v_{ns})$ is the node-label of v_{ns}. Let V be the set of all R-state successors of the R-non-state successor v_{ns} of v which covers x. Then $\delta(s_v, x) = \{s_{v_s} \in S \mid v_s \in V\}$.

4.5 Correctness and Completeness

Theorem 4.5.1 *(Correctness) If A is the module automaton synthesized after checking the realizability of the specification formula ψ under strong [weak] fairness, then for all behaviors σ accepted by A under strong [weak] fairness, $\sigma \models \psi$.*

Proof: Suppose there were a behavior σ accepted by A under strong fairness but $\sigma \not\models \psi$. We will prove that this leads to a contradiction. First, from $\sigma \not\models \psi$, then $\sigma \models \neg\psi$ and we can show that σ can be embedded into a path in the tableau graph *Glb*. The path starts from the root node of *Glb* and may be either finite or infinite. A behavior σ can be embedded into a path $n_0 n_1 n_2 \ldots$ in the tableau graph iff for all state nodes n_i, if n_i is the j-th state node in the path, then for all $\phi \in \Phi(n_i)$, $\langle \sigma, j \rangle \models \phi$. Next, we can show that the path $n_0 n_1 n_2 \ldots$ can be embedded into a path $v_0 v_1 v_2 \ldots$ in the realizability graph, using the assumption that σ is accepted by A. We use the fact that we compute the maximally negation-consistent subset in the realizability-checking algorithm to show (by induction) the existence of the part of the embedding from an R-non-state node to an R-state node and the fact that we compute the largest subset $N \subseteq L(v)$ for each R-state node v in the synthesis algorithm for the part of the embedding from an R-state node to an R-non-state node. If the path $n_0 n_1 \ldots$ in *Glb* is finite, then it must be the case that the last state node n_t in the path must contain only atomic formulas, because $\sigma \models \neg\psi$. It implies that n_t is accessible (in N_{acc}) from some node in the label of the last R-state node v_l of the path $v_0 v_1 \ldots$. If that is the case, v_l would have been deleted in step 3 of the realizability-checking algorithm, a contradiction.

If $n_0 n_1 \ldots$ is infinite but $v_0 v_1 \ldots v_l$ is finite, then we can also derive a contradiction by showing that for the case of strong [weak] fairness, there must be a self-supporting loop within $n_0 n_1 \ldots$ such that all [some] state nodes in the loop are environment-compatible and in the set *Disabled*. The essential step is to use the fact that σ is accepted under strong [weak] fairness and to show from the properties of maximally consistent subsets that all state nodes n_i in the loop must be in the set *Disabled* if $\delta(s_{v_l}, \pi(n_i)) = \emptyset$. However, if such a self-supporting loop exists, then v_l would have been deleted in step 6.

Similarly, we can also derive a contradiction in the case when both $n_0 n_1 \ldots$ and $v_0 v_1 \ldots$ are infinite, by showing that the loops in $v_0 v_1 \ldots$ would have been eliminated in step 5.(c).ii. ∎

Theorem 4.5.2 *(Termination) The realizability-checking algorithm always terminates.*

Proof: There are only finitely many possible R-state and R-non-state nodes. Therefore, it is not possible to keep expanding the realizability graph forever. It is also clear that there are only finitely many maximally consistent and maximally negation-consistent subsets at any time in the algorithm. ∎

Theorem 4.5.3 *(Completeness) The formula ψ is realizable iff the root node of the realizability graph is not deleted.*

Proof: One direction of the proof, showing that if the root node of the realizability graph is not deleted then ψ is realizable, is straightforward from theorem 4.5.1 (correctness).

In the other direction, we assume that ψ is realizable and show that the root node of the realizability graph is not deleted. Since ψ is realizable, there exists a function f which realizes it.

We have to define an embedding of a behavior into the realizability graph. A behavior σ can be embedded into a finite or infinite path $v_0 v_1 \ldots$ starting from the root in the realizability graph iff there exists an increasing sequence of integers $d_0 d_1 \ldots$ such that all of the following conditions are true:

- for all $i \geq 0$ and $n \in L(v_{2i})$, there is a formula $\phi \in \Phi(n)$ such that $\langle \sigma, d_i \rangle \models \neg\phi$,
- for all $i > 0$, $\langle \sigma, d_i \rangle \models \mu$,
- for all $i > 0$ and $n \in L(v_{2i+1})$, there is a formula $\phi \in \Phi(n)$ such that $\langle \sigma, d_i - 1 \rangle \models \neg\phi$.

An R-state node is called *reachable* iff there is a behavior of f which can be embedded into some path passing through the node. It is easy to see that the root node must be reachable. We want to show that some of the reachable nodes including the root are not deleted.

First, we can show that a reachable R-state node must not be deleted in step 3; otherwise, we can easily construct a behavior of f which falsifies ψ.

Next, we can show that for every reachable R-state node v and every self-supporting environment-compatible loops accessible from a node in the node-label $L(v)$ of v, there exists a node n in the loop such that for every R-non-state successor v_{ns} of v which contains n in the node-label, there is an R-state successor v' of v_{ns} which is also reachable. Again, we can prove this by showing that if such n does not exist, we can construct a fair behavior of f which falsifies ψ. We also use the fact that the label of v_{ns} is a maximally consistent set to show the existence of the embedding into a path through v_{ns}.

Finally, we can show that for any loop of reachable R-state nodes, if there is a self-supporting loop in Glb which can be embedded into it as in step 5.(c).ii, there is a reachable R-state node in the loop which is not deleted as the result of breaking the loop of R-state nodes in step 5.(c).ii. We prove this by considering a behavior of f which can be embedded into a path passing through this loop of reachable R-state nodes. Clearly, the path cannot remain within the loop forever or the behavior will not satisfy ψ. ■

Chapter 5

Related Works and Conclusion

5.1 Related Works

5.1.1 Program synthesis

The problem of automatic program synthesis has been previously studied in many different frameworks. For functional programs, the specification is a first-order formula expressing the desired relationship between inputs and outputs, where the synthesized program can be extracted from a constructive proof of the formula [MW80,Con85]. The approach used in [MW80] has been successfully applied to synthesize sequential programs.

Later, many efforts [EC82,MW84] have been made to extend the approach to synthesize reactive programs. The synthesized program is extracted from a proof of the satisfiability of the specification given in either linear temporal logic [MW84] or branching time logic [EC82]. However, the reactive programs considered in these works do not have any interaction with the environment, that is, they are *closed* systems.

The effort to synthesize reactive modules, i.e., *open* systems, was first reported in [PR89a]. In that paper, the synthesis of reactive *synchronous* modules from a specification in linear-time temporal logic is linked to the problem of checking the validity of a branching-time temporal formula obtained by transforming the original specification.

The restriction to synchronous systems (or the game of perfect information) was removed in [PR89b] where the problem of synthesizing asynchronous systems is considered. In that work, a linear-time temporal specification is transformed into a formula in branching-time temporal logic by introducing read and write variables, and by adding constraints on the variables.

At about the same time, several notions of realizability were introduced and studied in [ALW89]. For the finite case, the approach taken is similar to the automata approach of [PR89a,b]. Because of the choice of the specification, the method can check realizability in a more general sense, that is, when the behavior of the environment is restricted. However, [ALW89] only considered synchronous systems.

In [WD91], the approach of [PR89b] was extended to handle shared variables and the restriction on read and write sequences was relaxed. The paper also generalizes [ALW89] to include the asynchronous and real-time cases.

An important aspect of concurrent and reactive programming is the fairness assumption. The problem of synthesizing a reactive program under fairness assumption was first raised in [ALW89] but the solution was not provided. The first solution was reported in [AM94]. The tableau approach [MW84] was extended in [AM94] not only to handle the synthesis problem of open systems, but also address how fairness assumptions can be handled. Later, [Var95] shows that fair realizability checking and synthesis can also be carried out with the automata-theoretic approach.

5.1.2 Game and control theories

Game theory and control theory are closely related to the problem of program synthesis. In game theory, many researchers have studied games where two players take actions alternatively and infinitely. The first player wins if the infinite sequences generated by the game belong to a certain set and loses otherwise. A certain kind of games are called *Borel* games and it was shown to be related to various temporal logics and ω-languages. See [GH82,HR86,Sta87,Tho90]. The existence of a winning startegy for either players, or *determinacy*, was studied quite thoroughly [BL69,Mar75,Bu83] and it was shown that all Borel games are determined. Recently, [Tho94] gave a new recursive construction of winning strategies for finite-state games and raised new questions on finding the winning strategies for more complicated games such as games over pushdown transition graphs and games over hybrid systems. [Tho94] showed that there is a memoryless strategy for games characterized by a Rabin condition and [Les95] showed that there is a polynomial-sized strategy for games with Muller acceptatnce condition.

For control theory, [RW89] studied the control problem of *discrete event systems (DES).*

In the paper, DES is simply a finite state machine with certain subsets of the alphabet designated as control patterns. The sets of desired behaviors studied [RW89] can be characterized as safety properties. Later, [Thi92,TW94] extended the approach to study the control problem of infinite behaviors with liveness requirements. For timed and hybrid systems, [MPS94] has made an attempt to synthesize controllers for certain classes of timed and hybrid systems, and has shown that there is a memoryless controller for a certain formulation of the control problem. For automata-theoretic approach, see [ABB95].

5.1.3 Fairness

Various notions of fairness have been introduced and investigated [Fra86]. [Pa80] showed the connection between fairness and fixpoints in the semantics of data-flow languages. The notions of strong and weak fairness in transition systems were taken from [MP92]. The original idea of the notions was first proposed in [LPS81]. [QS83] proposed a notion of (strong) fairness in transition systems and studied a logic for proving properties under fairness assumption. Fairness in CCS was studied by [CS84] and its connection to regularity was investigated in [GN89,PRW87,Pri88,Pri93]. [AH94] proposed a stronger notion of fairness called finitary fairness and showed that finitary fairness is adequately abstract for the purpose of verification and leads to a simpler verification process.

5.2 Conclusion

We uniformly define and study the synthesis problems of reactive programs. We consider many synthesis problems, namely, basic, weak, maximal, exact, modular, and ES synthesis, and classify them into two groups, closed and open system synthesis, depending on whether the program being synthesized interacts with its environments.

In this work, a specification is a set of ω-regular (infinite) sequences and regular (finite) sequences. The output or the implementation is a transition system for closed system synthesis and a transition system module for open system synthesis. The specification and implementation languages we study subsume other works mentioned earlier in the previous section. In our framework, the synthesized program may be disabled, have any number of transition functions/ processes, or contain some fairness constraints. We also study the relationship between the number of processes and the solution of synthesis problems. Our framework is also general enough to handle modeling variations such as how the modules are composed, synchronously or asynchronously, or how the scheduler behaves.

In modular synthesis problem, we have a model of the environment as an input to the problem. It can be shown easily that games and control problems may be formulated as a modular synthesis problem and solve uniformly in our framework.

For closed system, we show that there is always an exact solution and the solution exists in 2τDTS – a very simple class in the hierachy of classes of transition systems – when the specification specifies only infinite behaviors, and 2TS for general specifications that specify both finite and infinite behaviors.

For open system, we solve all cases except the maximal synthesis and ES synthesis problems. Our framework allows us to specify the class of TSMs we want to consider as the implementation language. Although we still do not have a method to find the maximal solution for open systems, we show that we can find a solution which maximizes the good projected behaviors and the solution lies in 2TSM. We prove that we can solve ES synthesis, when the environment assumption is a safety property, or when the domain of the environment we consider is limited to some classes of TSMs.

Bibliography

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. *Proc. 16th Int. Colloq. Aut. Lang. and Prog*, Springer-Verlag, Berlin, LNCS 372, 1989, pp. 1–17.

[AH94] R. Alur and T.A. Henzinger, Finitary fairness, *Proc. Symp. on Logic in Comp. Sci. LICS'94*, 1994, pp. 52–61.

[AM94] A. Anuchitanukul and Z. Manna, Realizability and synthesis of reactive modules, *Computer Aided Verification, Proc. 6th CAV94 Workshop*, LNCS 818, 1994, pp. 156–169.

[ABB95] A. Aziz, F. Balarin, et. al., Supervisory Control of Finite State Machine, *Computer Aided Verification, Proc 7th CAV95 Workshop*, LNCS 939, 1995, pp. 279–292.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli, Now you may compose temporal logic specifications, *Proc. 16th ACM Symp. Theory of Comp.*, 1984, pp. 51–63.

[BPS94] D. Bruschi, G. Pighizzini, and N. Sabadini, On the existence of minimum asynchronous automata and on the equivalence problem for unambiguous regular trace languages, *Information and Computation*, vol. 108, no. 2, Feb. 1994, pp. 262–85.

[Bu83] J.R. Büchi, State-strategies for games in $F_{\sigma\delta} \cap G_{\delta\sigma}$, *Journal of Symbolic Logic*, vol. 48, no. 4, Dec 1983, pp. 1171–1198.

[BL69] J.R. Büchi and L.H. Landweber, Solving sequential conditions by finite-state strategies, *Trans Amer. Math. Soc.* vol. 138, 1969, pp. 295–311.

[CDK93] E.M. Clarke, I.A. Draghicescu, R.P. Kurshan, A unified approach for showing language inclusion and equivalence between various types of ω-automata, *Information Processing Letters*, vol. 46, 1993, pp. 301–308.

[Con85] R.L. Constable, Constructive mathematics as a programming logic I: Some principles of theory, *Ann. Discrete Math.*, vol. 24, 1985, pp. 21–38.

[CS84] G. Costa and C. Stirling, A fair calculus of communicating systems, *Acta informatica*, vol. 21, 1984, pp. 417–441.

[Em85] E.A. Emerson, Automata, tableaux, and temporal logics, *Proc. Conf. Logic of Programs*, LNCS 193, Springer, 1985, pp. 79–88.

[EC82] E.A. Emerson and E.M. Clarke, Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comp. Prog.*, vol. 2, no. 3, 1982, pp. 241–266.

[EJ89] E.A. Emerson and C.S. Jutla, On simultaneously determinizing and complementing ω-automata, *Proc. 4th IEEE Symp. Logic in Comp. Sci.*, 1989, pp. 333-42.

[Fra86] N. Francez, Fairness, Springer-Verlag, 1986.

[GN89] I. Guessarian and W. Nair-Dinedane, Fairness and regularity for SCCS processes, *Informatique Theorique et Applications*, vol. 23, 1989, pp. 59–86.

[GH82] Y. Gurevich and L.A. Harrington, Automata, trees, and games, *Proc. 14th Ann. ACM Symp. on the Theory of Computing*, 1982, pp. 60–65.

[HR86] H.J. Hoogeboom and G. Rozenberg, Infinitary languages: basic theory and applications to concurrent systems, in J. W. de Bakker et al. eds., *Current Trends in Concurrency*, LNCS 224, Springer, Berlin, 1986, pp. 266–342.

[KK91] N. Klarlund and D. Kozen, Rabin measures and their applications to fairness and automata theory, *Proc. of 6th Annual IEEE Symp. on Logic in Computer Science*, 1991, pp. 256–265.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi, Impartial, justice and fairness: the ethics of concurrent termination, *ICALP*, LNCS 115, 1981, pp. 264–277.

[Les95] H. Lescow, On polynomial-size programs winning finite-stae games, *Computer Aided Verification, Proc 7th CAV95 Workshop*, LNCS 939, 1995, pp. 239–252.

[MPS94] O. Maler, A. Pnueli, and J. Sifakis, On the synthesis of Discrete Controllers for Timed Systems, in E.W. Mayr and C. Pnuech (Eds), *Proc. of STACS95*, LNCS 900, Springer, 1995, pp. 229–242.

[MP92] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer-Verlag, 1992.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Prog. of Lang. and Sys.*, vol. 2, no. 1, 1980, pp. 90–121.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal-logic specifications. *ACM Trans. on Prog. Lang. and Sys.*, vol. 6, no. 1, 1984, pp. 68–93.

[Mar75] D.A. Martin, Borel determinacy, *Ann. Math.*, vol. 102, 1975, pp. 363–371.

[Par80] D. Park, On the semantics of fair parallelism, *Abstract Software Specification*, LNCS 86, 1980, pp. 504–524.

[PR89a] A. Pnueli and R. Rosner, On the synthesis of a reactive module. *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, 1989, pp. 179–190.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. *Proc. 16th Int. Colloq. Aut. Lang. Prog.* Lec. Notes in Comp. Sci. 372, Springer-Verlag, Berlin, 1989, pp. 652–671.

[Pri88] L. Priese, Fairness, *EATCS Bull.*, vol. 35, 1988, pp. 171–181.

[Pri93] L. Priese, Fairness, Part II. *EATCS Bull.*, vol. 50, 1993, pp. 247–259.

[PN93] L. Priese and D. Nolte, Strong fairness and ultra metrics, *Theoretical Computer Science*, vol. 99, 1992, pp. 121–140.

[PRW87] L. Priese, R. Rehrmann, and U. Willecke-klemme, An Introduction to the regular theory of fairness, *Theoretical Computer Science*, vol. 54, 1987, pp. 139–163.

[QS83] J.P. Queille and J. Sifakis, Fairness and related properties in transition systems – a temporal logic to deal with fairness, *ACTA Informatica*, vol. 19, 1983, pp. 195-220.

[RW89] P.J.G. Ramdage and W.M. Wonham, The Control of Discrete Event Systems, *Proc. of the IEEE on Control Theory*, vol. 77, 1989, pp. 81–98.

[Saf88] S. Safra, On the complexity of ω-automata, *Proceedings of the 21st ACM Symposium on Theory of Computing*, Seattle, May 1989, pp. 127–137.

[Saf92] S. Safra, Exponential determinization for ω-automata with strong-fairness acceptance condition, ACM Symp. on Theory of Computing, vol. 24, 1992, pp. 275–282.

[Seg93] R. Segala, Quiesence, fairness, testing, and the notion of implementation, *CONCUR'93* Best, E. (editor), 1993, pp. 324–338.

[Sta87] L. Staiger, Research in the theory of ω-languages, *J. Inform. Process*, Cybernet, vol. 23, 1987, pp. 415–439.

[TW94] J.G. Thistle and W.M. Wonham, Control of infinite behavior of finite automata, *SIAM Journal on Control and Optimization*, vol. 32, no. 4, July 1994, pp. 1075–1097.

[Tho94] W. Thomas, On the synthesis of strategies in infinite games, in E.W. Mayr and C. Pnuech, eds., *Proc. of STACS95*, LNCS 900, Springer, 1995.

[Tho90] W. Thomas, Automata on Infinite Objects, *Handbook of Theoretical Computer Science*, J. van Leeuwen, eds., 1990, pp. 133–191.

[Var95] M.Y. Vardi, An automata-theoretic approach to fair realizability and synthesis, *Computer Aided Verification, Proc 7th CAV95 Workshop*, LNCS 939, 1995, pp. 267–278.

[VS85] M.Y. Vardi and L.J. Stockmeyer, Improved upper and lower bounds for modal logics of programs, *Proc. 17th ACM Symp. Theory of Comp.*, 1985, pp. 240–251.

[Wo83] P. Wolper. Temporal logic can be more expressive. *Info. and Cont.*, vol. 56, 1993, pp. 72–99.

[Wo85] P. Wolper. The tableau method for temporal logic: An overview. *Logique et Anal.*, vol. 28, 1985, 119–136.

[WD91] H. Wong-Toi and D.L. Dill. Synthesizing processes and schedulers from temporal specifications, *Computer-Aided Verification (Proc. CAV90 Workshop)*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 3 (American Mathematical Society, 1991).