行政院國家科學委員會專題研究計畫 成果報告

從假設保證式規格自動合成反應式模組之研究
研究成果報告(精簡版)

計 畫 類 別 ：個別型
計 畫 編 號 ：NSC 95-2221-E-002-127-
執 行 期 間 ：95 年 08 月 01 日至 96 年 08 月 31 日
執 行 單 位 ：國立臺灣大學資訊管理學系暨研究所

計 畫 主 持 人：蔡益坤

計畫參與人員：博士班研究生-兼任助理：蔡明憲

報 告 附 件 ：出席國際會議研究心得報告及發表論文

處 理 方 式 ：本計畫可公開查詢

中 華 民 國 96 年 12 月 17 日

Synthesis of Reactive Modules from Assumption/Guarantee Specifications
從假設保證式規格自動合成反應式模組之研究
(NSC 95-2221-E-002-127)

Yih-Kuen Tsay
蔡益坤
Department of Information Management
National Taiwan University

Contents

Abstract

Synthesis and verification both concern formal correctness of a system. While verification checks whether a given system is correct with respect to its specification, synthesis attempts to automatically construct a correct one. A central concept in the problem of synthesizing an open system or reactive module is the *realizability* of a specification by a module that does not control or foresee the future of its environment. This concept of realizability corresponds to the existence of a *winning strategy* for the protagonist representing the module in a two-player game with the specification as the winning condition.

In this project, we investigate the synthesis problem with the specification given by a linear temporal formula in the assumption/guarantee style. This was motivated by the high complexity of synthesis with arbitrary linear temporal formulae, which is double exponential. We show that, for assumption/guarantee formulae, realizability and hence synthesis are solvable in polynomial space. It is possible to decompose the specification of a composite system, if realizable, into the assumption/guarantee specifications of its constituent modules and the synthesis of a reactive module from its assumption/guarantee specification then serves as a building block for the synthesis of the composite system.

Keywords: Assumption/Guarantee, Automata, Games, Open Systems, Reactive Systems, Specification, Synthesis, Temporal Logic, Verification, Winning Strategies.

1 Introduction

We are concerned with correct construction of systems that are the parallel composition of several concurrent modules. The kind of systems or modules that interest us are *open* in that they interact with their environment (consisting of other systems or modules) and *reactive* in that the interaction is ongoing and may never terminate. As a system may serve as a module of another larger system, we will not distinguish between a system and a module unless necessary.

Instead of verifying the conformance of a system to its specification after the construction, one may try to synthesize, by applying some systematic transformation to the given specification, a system that is correct by way of construction. However, not every specification is realizable. As a reactive module cannot control or predict its environment, at any moment in time the module can rely only on the inputs it has observed from the environment so far to decide its next output in such a way that the resulting interaction will meet the specification. This dependency of a module's reaction only on the history of its environment's behavior is reminiscent of a strategy in games.

Indeed, a reactive module may be modeled as an infinite two-player game where one player (the protagonist) represents the module and the other (the adversary) represents its environment. With a given game arena and a set of designated initial game positions, the winning strategy for the protagonist represents an abstract program of the module, while the winning set (from the perspective of the protagonist) specifies the requirements of the module. The realizability of a specification by a reactive module then corresponds to the existence of a winning strategy for the protagonist that represents the module and the synthesis of the module to the computing of the winning strategy. Though essentially the same problem and its solution may be traced

back to an earlier time, realizability/synthesis in this context was first studied more extensively by Pnueli and Rosner [14] and by Abadi *et al.* [1]. Follow-up works, e.g., [15, 8, 10], have gone beyond to take into account architectural constraints among the modules of a system.

Realizability and synthesis may also be understood in logical terms. As an illustration, consider a specification $\varphi(x, y)$ for a sequential program with x as the input variable and y the output variable. The specification $\varphi(x, y)$ can be realized by a program if and only if $\forall x \exists y \varphi(x, y)$, or simply $\exists y \varphi(x, y)$, is valid. With a constructive logic, a witness $f(x)$ such that $\varphi(x, f(x))$ holds for every x would represent an abstract program that realizes the specification. For reactive modules, the variables x and y would need to be interpreted as sequences of values; in other words, they are flexible variables in temporal logic.

There are mainly two types of temporal logic: linear-time and branching-time. They both are designed specifically for expressing assertions about infinite computations. Linear(-time) temporal logic assumes an implicit universal quantification over all branches (from the root) of the computation tree of a system, while branching(-time) temporal logic permits explicit existential and universal quantifications over the branches (either from the root or any other internal node). Though we will need to speak at times of satisfaction of a temporal formula over all branches from the root of a computation tree, we choose to use a linear temporal logic and treat this particular occurrence of universal branching quantification in an ad hoc manner.

One primary example of linear temporal logic is the temporal logic of Manna and Pnueli [12], referred to as LTL here.[1] LTL formulae express properties of an infinite computation and are suitable for specifying the behaviors of a module. The behaviors of a module are specified by an LTL formula in the sense that the executions of the module are *included* in the set of those satisfying the formula. A particularly interesting subset of LTL is QPTL, where each variable is restricted to be boolean. While LTL is most general but undecidable, QPTL is quite expressive and yet decidable. The subset of QPTL without quantifiers is referred to as PTL.

Synthesis, even when decidable, is generally of extremely high complexity. This is in sharp contrast with verification which, though of high complexity, is still amenable to practical tools. Most works on synthesis consider *arbitrary* specifications in a given logic. This generality is too costly and may not be necessary. It is plausible, in the sense of reducing the complexity, to consider the problem of synthesizing a reactive module with specifications of restricted forms.

A module is normally meant for particular contexts or environments and will behave properly only if its environment does. Regardless of the language used, when specifying properties of a module, one should therefore include (1) assumed properties about its environment and (2) guaranteed properties of the module if the environment obeys the assumption [13, 5, 9]. This type of specifications are essentially a generalization of the pre and post-conditions for sequential programs [4]. They are referred to as *assumption/guarantee* specifications.

[1] Most researchers in the model checking community use LTL to refer to the propositional linear temporal logic (which will be called PTL here), considering it as a sub-logic of the general Computation Tree Logic (CTL*) [2]. We reserve the acronym LTL for the full linear temporal logic as defined by Manna and Pnueli in [12].

In this project, we investigate the synthesis problem with the specification given by a temporal-logic formula in the A/G style. PTL and QPTL will be our main temporal logics for specifying the behaviors of a module. We have developed in the past a temporal logic formulation of assumption-guarantee specifications, A/G specifications for short, as well as proof rules for composing such specifications [6, 23]. The formulation was defined in LTL and can obviously be specialized for PTL and QPTL.

Not only may they help in reducing the complexity, A/G specifications also enable us to treat maximal environments, which provide all possible inputs to the module, and non-maximal ones uniformly. And, with QPTL as the specification language, information hiding may also be expressed as existential quantification over flexible variables. The decidable cases in synthesizing distributed systems are of extremely high (non-elementary) complexities, which suggests that fully algorithmic solutions are not practical. However, it may be possible to decompose the specification of a composite system, if realizable, into the A/G specifications of its constituent modules. The preceding solutions for synthesizing a reactive module from its A/G specification then serves as a building block for the synthesis of the composite system.

2 Related Work

A brief survey has been provided in the project proposal, except the following two more recent works. Schewe and Finkbeiner [17] proved that, for Alternating Time Logics, distributed synthesis is decidable if and only if the architecture is hierarchical (where for every two modules, one sees all inputs of the other). Hierarchical architectures form a strict subset of architectures that do not contain an information fork, for which the synthesis problem is decidable for weaker linear and branching-time logics. In [16], the same authors consider the problem of synthesizing a system with a given upper bound on its size. For linear temporal logic, they showed that the problem can be reduced to SAT solving. The translated constraint system seems to be very large, with a size exponential in the length of the original temporal formula.

We note that these works concern synthesis of an entire distributed system, while our main result concerns the synthesis of a single module with restricted forms of specification, in the hope of reducing the complexity.

3 Preliminaries

3.1 Basic Structures

We will be dealing with sequences and trees labeled with symbols from some alphabet, which serve as models of a temporal formula, inputs to an automaton, runs of an automaton, etc. Sequences are simply the special case of unary trees or paths (branches) of a tree. Let Z_m denote the set $\{0, 1, \cdots, m-1\}$ and ϵ denote the *empty* word (string). A k-ary *tree* is a prefix-closed subset of Z_k^*. That is, each node of a k-ary tree is represented by a word in Z_k^* and if xa

is a node in a tree, then its parent x is also a node in the tree. The set Z_k^* itself represents the *full* k-ary tree and the special case of $k = 1$, namely $\{0\}^* = \{\epsilon, 0, 00, \cdots\}$, is isomorphic to the *sequence* of natural numbers $0, 1, 2, \cdots$. A word $w \in (Z_k^* \cup Z_k^\omega)$ is called a *path* of a k-ary tree T if, for every proper prefix w' of w ($w' \prec w$), $w' \in T$, and if w is finite, $w \in T$, but for all $a \in Z_k$, $w \cdot a \notin T$. In other words, a path of a tree goes from the root and passes through a particular thread of descendants completely till it reaches, in the finite case, a leaf of the tree.

Given an alphabet Σ, a Σ-labeled tree, or simply Σ-*tree*, is a mapping $t : T \mapsto \Sigma$ that maps each node in the tree T to a symbol in Σ. To avoid an extra layer of mapping, we allow the symbols in an alphabet to have a structure. For our purpose, the alphabet Σ is often the powerset of a set of boolean variables such as $2^{\{x,y\}} = \{\{\}, \{x\}, \{y\}, \{x, y\}\}$, where each element is a 'structured' symbol representing one of the four possible truth assignments to the two boolean variables x and y. To comply with the usual notation for infinite sequences of states, we will often write a labeled sequence $\sigma : \{0\}^* \mapsto \Sigma$ as $\sigma = s_0, s_1, s_2, \cdots$ where $s_i \in \Sigma$ corresponds to $\sigma(i)$ $(=\sigma(0^i))$.

3.2 Temporal Logic

The Quantified Propositional Temporal Logic (QPTL) is a restriction of Linear Temporal Logic (LTL) to the boolean domain or, equivalently, an extension of Propositional Temporal Logic (PTL) with quantification. PTL is simple and strong enough for expressing many common, interesting safety and liveness properties. Yet, situations had arisen that called for a more expressive but still decidable logic. For example, PTL is incapable of expressing the fact that a proposition holds exactly at every other state of a sequence [26] and, more importantly, it is inconvenient for treating abstraction. QPTL is one of the (decidable) extensions to PTL, which was originally introduced by Sistla [18] and further studied by Sistla *et al.* [20] and by Kesten and Pnueli [7]. As an extension of PTL, QPTL formulae are constructed by applying boolean connectives and temporal operators to propositions (boolean variables) drawn from a predefined universe. QPTL additionally allows quantification over boolean variables. In the context of synthesis, we will also need a particular form of temporal formulae with a single universal branching quantification that is outside the scope of any temporal operator.

Given a (countable) set of boolean variables V, the syntax of QPTL formulae is defined as follows:

- Every variable in V is a formula.
- If p and q are formulae, then so are $\neg p$, $p \vee q$, $\bigcirc p$ (*next* p), $\Box p$ (*henceforth* p or always p), $\ominus p$ (*before* p), and $\boxminus p$ (*so-far* p).
- If $x \in V$ and p is a formula (not necessarily containing x), then $\exists x : p$ is a formula.

Additional operators can be defined in terms of $\neg$, $\vee$, $\bigcirc$, $\Box$, $\ominus$, and $\boxminus$. Below are a few useful

ones.

$$\begin{aligned} p \wedge q &\triangleq \neg(\neg p \vee \neg q) \\ p \rightarrow q &\triangleq \neg p \vee q \\ \Diamond p &\triangleq \neg\Box\neg p \quad (\textit{eventually } p \text{ or sometime } p) \\ \Diamond\!\!-\!\!p &\triangleq \neg\boxminus\neg p \quad (\textit{once } p \text{ or sometime in the past } p) \\ \forall x : p(x) &\triangleq \neg\exists x : \neg p(x) \end{aligned}$$

The familiar $p\,\mathcal{U}\,q$ (p *until* q) and $p\,\mathcal{S}\,q$ (p *since* q) can also be defined (with the help of auxiliary variables), but these derived operators are not useful for our purposes.

A QPTL formula is interpreted over an infinite sequence of states $\sigma = s_0, s_1, s_2, \cdots$, relative to a position in the sequence; a state is simply a subset of the boolean variables (representing a truth assignment to the boolean variables). Below is the semantics of QPTL formulae:

- For a variable $x \in V$, $(\sigma, i) \models x$ iff $x \in s_i$.
- $(\sigma, i) \models \neg p$ iff $(\sigma, i) \models p$ does not hold.
- $(\sigma, i) \models p \vee q$ iff $(\sigma, i) \models p$ or $(\sigma, i) \models q$.
- $(\sigma, i) \models \bigcirc p$ iff $(\sigma, i+1) \models p$. In other words, $\bigcirc p$ holds at a position (of some sequence) if p holds at the next position (of that sequence).
- $(\sigma, i) \models \Box p$ iff $\forall k \geq i : (\sigma, k) \models p$. In other words, $\Box p$ holds at a position if p holds at that position and all following positions.
- $(\sigma, i) \models \circledcirc p$ iff $(i > 0) \rightarrow ((\sigma, i-1) \models p)$. In other words, $\circledcirc p$ holds at a position if either the position is the first position (i.e., $i = 0$) or p holds at its immediate preceding position. $\mathit{first} \triangleq \circledcirc \mathit{false}$, which is true only at position 0.
- $(\sigma, i) \models \boxminus p$ iff $\forall k : 0 \leq k \leq i : (\sigma, k) \models p$. In other words, $\boxminus p$ holds at a position if p holds at that position and all preceding positions.

A sequence of states σ' is called a *x-variant* of σ if σ' differs from σ in at most the interpretation given to x in each state.

- $(\sigma, i) \models \exists x\colon p$ iff $(\sigma', i) \models p$ for some *x-variant* σ' of σ.

The truth of $\exists x\colon p$, for a boolean variable x, depends on the *existence of an infinite sequence of boolean values* for x (one for each state, rather than just a single value) such that p can be satisfied. Like in classical logic, existentially quantified variables are internal (local) to a formula and their values are inaccessible from outside of the formula.

We say that a sequence σ *satisfies* a formula p if $(\sigma, 0) \models p$, often abbreviated as $\sigma \models p$. A formula p is *valid*, denoted $\models p$, if p is satisfied by every sequence. A temporal formula can be

used to specify some property of a reactive system in the sense that every sequence of states generated from executing the system satisfies the temporal formula.

We will also use temporal formulae of the form $\forall x \exists y : \mathrm{A}\varphi(x, y)$, where A is the universal branching quantifier and $\varphi(x, y)$ is a QPTL formula with two free variables x and y. Suppose V is the set of all variables occurring in $\varphi(x, y)$, including x and y. The formula $\forall x \exists y \colon \mathrm{A}\varphi(x, y)$ is interpreted over a 2^V-tree that represents all computations of a system. A 2^V-tree t' is called a *x-variant* of t if t' differs from t in at most the interpretation given to x in each node.

- $t \models \exists x \colon p$ iff $t' \models p$ for some *x-variant* t' of t.
- $t \models \forall x \colon p$ iff $t' \models p$ for every *x-variant* t' of t.
- $t \models \mathrm{A}p$ iff $(\sigma, 0) \models p$ for every path σ of t.

3.3 Automata on Infinite Words

We briefly review finite-state automata on infinite words, namely ω-automata [3, 22], which are closely related to PTL and QPTL. As computations produced by a reactive system can be viewed as infinite words, i.e., infinite sequences of symbols, ω-automata and associated decision procedures have served as a popular tool for specifying and verifying reactive systems. Among the many equivalent classes of ω-automata, Büchi automata are the most well-studied and have the same expressive power as QPTL [18].

In terms of complexity, the size of the equivalent Büchi automaton of a PTL formula is in the worst case exponential in the size of the formula, while the size of the equivalent Büchi automaton of a QPTL formula is non-elementary (cannot be bounded by $2^{2^{\cdot^{\cdot^{2^n}}}}$ with a fixed height of the exponent stack) with respect to the size of the formula. It has been shown that the satisfiability problem for QPTL is non-elementary (which explains why QPTL has not been used often). On the other hand, the high complexity also suggests that QPTL is very succinct.

An ω-automaton is a tuple $\langle Q, \Sigma, \delta, Q^{init}, Acc \rangle$, where

- Q is a finite set of states.
- Σ is a finite alphabet.
- $\delta : Q \times \Sigma \mapsto 2^Q$ is the state transition function.
- $Q^{init} \subseteq Q$ is a non-empty set of initial states.
- Acc is the *acceptance* set. Different structures of the acceptance set are required for different classes of automata.

The automaton as defined is *nondeterministic*. It becomes *deterministic* if $|Q^{init}| = 1$ and, for all $(q, a) \in Q \times \Sigma$, $|\delta(q, a)| \leq 1$.

Let $A = \langle Q_A, \Sigma_A, \delta_A, Q_A^{init}, Acc_A \rangle$ be an ω-automaton. A *run* of A over an infinite word $w = a_0 a_1 a_2 \cdots \in \Sigma_A^\omega$ is an infinite sequence of states $r = q_0 q_1 q_2 \cdots \in Q_A^\omega$ such that:

- $q_0 \in Q_A^{init}$ and
- $q_i \in \delta(q_{i-1}, a_{i-1})$, for all $i \geq 1$.

Let $\mathrm{Inf}(r)$ be the set of states that appear infinitely many times in a run r. We now define the different classes of automata by giving the corresponding acceptance sets and conditions.

- An ω-automaton $A = \langle Q_A, \Sigma_A, \delta_A, Q_A^{init}, F \rangle$ with $F \subseteq Q$ is called a **Büchi** automaton if it uses the following acceptance condition (Büchi acceptance): a word $w \in \Sigma_A^\omega$ is accepted by A iff there exists a run r of A over w such that $\mathrm{Inf}(r) \cap F \neq \emptyset$.
- An ω-automaton $A = \langle Q_A, \Sigma_A, \delta_A, Q_A^{init}, \Omega \rangle$ with $\Omega = \{(R_1, P_1), (R_2, P_2), \cdots, (R_k, P_k)\}$, where $R_i, P_i \subseteq Q$, is called a **Streett** automaton if it uses the following acceptance condition (Streett acceptance): a word $w \in \Sigma_A^\omega$ is accepted by A iff there exists a run r of A over w such that, for all i, $(\mathrm{Inf}(r) \cap R_i \neq \emptyset) \vee (\mathrm{Inf}(r) \subseteq P_i)$, or equivalently, $(\mathrm{Inf}(r) \cap R_i \neq \emptyset) \vee (\mathrm{Inf}(r) \cap \bar{P}_i = \emptyset)$.
- An ω-automaton $A = \langle Q_A, \Sigma_A, \delta_A, Q_A^{init}, \Omega \rangle$ with $\Omega = \{(R_1, P_1), (R_2, P_2), \cdots, (R_k, P_k)\}$, where $R_i, P_i \subseteq Q$, is called a **Rabin** automaton if it uses the following acceptance condition (Rabin acceptance): a word $w \in \Sigma_A^\omega$ is accepted by A iff there exists a run r of A over w such that, for some i, $(\mathrm{Inf}(r) \cap R_i = \emptyset) \wedge (\mathrm{Inf}(r) \not\subseteq P_i)$, or equivalently, $(\mathrm{Inf}(r) \cap R_i = \emptyset) \wedge (\mathrm{Inf}(r) \cap \bar{P}_i \neq \emptyset)$.

  (There are other variants.)

We have deliberately used a slightly different convention in giving the acceptance sets and conditions for Streett and Rabin automata. To convert a usual definition to ours, each pair (E, F) in the acceptance set of the usual definition should be changed into $(E, \bar{F})$. Each pair has been named as (R, P) here, following [11], to suggest that R represents a *Recurrent* set and P a *Persistent* set, which would be more convenient when we want to show the correspondence between an ω-automata and a temporal formula.

Nondeterministic Büchi automata, deterministic/nondeterministic Streett automata, deterministic/nondeterministic Rabin automata are expressively equivalent and are all closed under union, intersection, and complementation. Deterministic Büchi automata are strictly weaker.

3.4 Automata on Infinite Trees

We now turn to automata that operate on infinite (labeled) trees, namely tree automata. Though invented for quite a different purpose, tree automata have a natural interpretation as two-player games and have found applications in the synthesis of reactive modules. In one such application, the labeled input tree may be seen as representing the strategy (control code)

of a module in the sense that the branches of a node in the input tree correspond to possible moves of the environment while the label at the end of each branch represents the response of the module to that move.

For the ease of exposition, we strict ourselves here to automata for full binary trees; extensions for other types of trees should be straightforward.

A *tree automaton* is a tuple $\langle Q, \Sigma, \delta, Q^{init}, Acc \rangle$, where

- Q is a finite set of states.
- Σ is a finite alphabet.
- $\delta : Q \times \Sigma \mapsto 2^{Q \times Q}$ is the state transition function.
- $Q^{init} \subseteq Q$ is a non-empty set of initial states.
- Acc is the *acceptance* set. Different structures of the acceptance set are required for different classes of automata.

Let $A = \langle Q_A, \Sigma_A, \delta_A, Q_A^{init}, Acc_A \rangle$ be a tree automaton. A *run* of A over a full binary Σ_A-tree t is a binary Q_A-labeled tree r such that:

- $r(\epsilon) \in Q_A^{init}$ and
- $(r(w0), r(w1)) \in \delta(r(w), t(w))$, for all $w \in \{0,1\}^*$.

Büchi, **Streett**, and **Rabin** tree automata are defined in the same way as their word-counterparts, except that the acceptance condition now is applied to the *paths* of the run tree.

For tree automata, being deterministic or nondeterministic makes a greater difference than for ω-automata. Nondeterministic tree automata (that read the input tree from the top, which our definitions do) are more expressive than their deterministic counterparts. In addition, while Streett and Rabin tree automata remain expressively equivalent, nondeterministic Büchi tree automata are weaker than Streett or Rabin tree automata.

3.5 Games

We review turn-based games. Though we will also speak of concurrent games, analogous concepts may be drawn from the simpler turn-based games. A game consists of a game arena (structure) and a winning condition. The game arena defines moves allowed for the players, while the winning condition defines how the winner is determined according to the played moves. We will mainly be concerned with two-player games.

An *arena* is a tuple $\langle V^0, V^1, E \rangle$ where V^0 and V^1 are disjoint sets of 0-vertices and 1-vertices respectively and $E \subseteq V \times V$ with $V = (V^0 \cup V_1)$ is a set of (legal) moves. The two players are called Player 0 and Player 1 respectively. We will use Player $\sigma \in \{0,1\}$ to refer to one player

and Player $\bar{\sigma} = 1 - \sigma$ to the other player or the opponent of the former player. A game is played by initially placing a token on some node in V. The two players then take turns (not necessarily alternating) to move the token to some other node as allowed by E. If the token is currently in node v which is a σ-vertex, then it is the turn of Player σ who may move the token to any node in the set $vE = \{v' \mid (v, v') \in E\}$ of successors of v. This is repeated indefinitely or until a dead-end node without successors is reached.

Formally, a play in an arena $A = \langle V_A^0, V_A^1, E_A \rangle$ is a sequence $\pi \in (V_A^+ \cup V_A^\omega)$ such that either of the following conditions holds:

- $\pi = v_0 v_1 v_2 \cdots v_l \in V_A^+$ is finite and, for all $i < l$, $(v_i, v_{i+1}) \in E$ and $v_l E = \emptyset$.
- $\pi = v_0 v_1 v_2 \cdots \in V_A^\omega$ is infinite and, for all i, $(v_i, v_{i+1}) \in E$.

A game is a pair $\langle A, \mathit{Win} \rangle$, where A is the game arena and Win is the winning set. The winning set contains only infinite plays; the winner (or loser) of a finite play is naturally determined by the last node in the play. Player 0 *wins* (and Play 1 *loses*) a play if the play belongs to the winning set. For our purposes, we will be interested in winning sets that can be described by the acceptance conditions we have seen earlier for ω-automata and tree automata. Different classes of games then arise and acquire their names from the different acceptance conditions.

While the various acceptance conditions make sense only for finite sets of states/nodes, the number of nodes in a game arena may be infinite. So, before the acceptance conditions can be applied, we introduce one layer of abstraction over the game arena, namely a coloring function $\chi : V \mapsto C$ that maps every node in V to an element in a finite set C of colors. The coloring χ is extended to a coloring of a play in the obvious way: if $\pi = v_0 v_1 v_2 \cdots$ is a play, then $\chi(\pi) = \pi(v_0)\pi(v_1)\pi(v_2)\cdots$.

A game $G = \langle A, W_\chi(\mathit{Acc}) \rangle$ is called a **Büchi** or **Rabin** game if its winning set $W_\chi(\mathit{Acc})$ is defined by a corresponding acceptance condition Acc, which is applied to the χ-colored plays (that is, a play is in the winning set if its colored version satisfies the acceptance condition). These games are usually referred to collectively as *regular* games. When a game G has a fixed starting node v^{init}, it is called an *initialized* game and is denoted as $\langle G, v^{\mathit{init}} \rangle$.

To win a play, either player would follow certain strategy when selecting the next move from the set of legal moves. Formally, for a given arena $\langle V^0, V^1, E \rangle$, a strategy is a partial function $f_\sigma : V^* \times V^\sigma \mapsto V$ which is defined for every $\pi v \in V^+$ ending with a node $v \in V^\sigma$ that has at least one successors and, in this case, $f_\sigma(\pi v) \in vE$. A prefix of a play $v_0 v_1 \cdots v_l$ is said to *conform* to f_σ if, for every i with $0 \leq i < l$ and $v_i \in V^\sigma$, $v_{i+1} = f_\sigma(v_0 \cdots v_i)$. A play conforms to a strategy if each of its prefixes conforms to the strategy. A strategy f_σ is said to be a *winning strategy for Player σ on U* ($\subseteq V$) if every play that starts from a vertex in U and conforms to f_σ is a win for Player σ.

3.6 Synthesis of a Reactive Module

We review an approach to synthesizing a module that satisfies a given PTL formula. For the ease of exposition but without loss of generality, let $\varphi(x, y)$ be the given PTL formula containing two boolean variables x and y, where x is an input variable and is controlled by the environment while y is an output variable and is controlled by the module.

As modules cannot control or predict the outputs of their environment, $\varphi(x, y)$ can be satisfied by a module, or realized, only if it prescribes sequences of values of x and y such that the current value of y depends only on the preceding values of x. Modeled as a two-player game, the value changes of y are the moves by the module (Player 0), the value changes of x are the moves by the environment (Player 1), and $\varphi(x, y)$ is the winning set. The dependence of y's values on the past values of x is exactly the requirement of a strategy for Player 0. *Realizability* of $\varphi(x, y)$ can therefore be formally defined as the existence of a winning strategy for Player 0 (which represents the module). And, the synthesis of a reactive module amounts to computing a winning strategy for Player 0 (as a strategy can be seen as an abstract program of the module).

Pnueli and Rosner [14] showed that $\varphi(x, y)$ is implementable if and only if $\forall x \exists y \colon \mathrm{A}\varphi(x, y)$, called the implementability formula, is valid (over all computation trees), which was proven equivalent to the *satisfiability* of $\mathrm{A}\varphi(x, y)$ by some full binary $2^{\{y\}}$-tree (where the two branches of each node represent the two possible values of x). Their implementability checking and synthesis algorithm proceeds in stages: First, a nondeterministic Büchi automaton A over $2^{\{x,y\}}$-sequences is constructed such that $L(A)$ contains exactly those sequences that satisfy $\varphi(x, y)$. Second, Büchi automaton A is then converted into a deterministic Rabin tree automaton B such that B will accept a full binary $2^{\{y\}}$-tree if and only if $\varphi(x, y)$ holds on all paths of the tree. Finally, automaton B is tested for non-emptiness, whose by-product is a winning strategy for Player 0 in the form of a deterministic transducer.

In the above, $\varphi(x, y)$ is arbitrary and no restrictions were imposed aside from the formula being a PTL formula.

3.7 A/G Specifications

In this section, we describe how temporal logic may be used to fully characterize a reactive module or specify its abstract properties, focusing on a formulation of A/G specifications that we have developed in the past [6, 23] and some of the associated composition rules.

A reactive module is equipped with a set of input/output variables for communication and local variables for storage. Its behaviors are prescribed by an initial condition on the output and local variables and a set of transitions that specify how it may change the values of its output and local variables in one execution step. Additionally, there are fairness conditions that dictate how often each transition should be performed. Semantically, a module is associated with a set of computations or sequences of states, each of which represents a possible 'open execution' of the module, i.e., execution of the module together with an arbitrary but compatible environment

that may arbitrarily change the values of its input variables but not its output or local variables.

With temporal logic as the modeling language, we distinguish two kinds of specification: system specification and requirement specification. The system specification of a module gives a full characterization of the module in the sense that the sequences satisfying the specification are exactly those produced from open execution of the module. Assuming a single local variable, its safety part can be put in the form of $\Box(\exists x\colon \boxminus((\mathit{first} \rightarrow \mathit{Init}) \wedge N))$, where x is the local variable, *Init* is a state formula, and N is the disjunction of transition formulae that relate the current values of the variables to their previous values. We omit the formula for fairness. When QPTL is used, values in a finite domain can be encoded by boolean values; for modules with variables of infinite domains, LTL is needed.

Requirement specifications are the usual type of temporal-logic specification, describing abstract properties of a module. A module M is said to satisfy a formula p if every computation of M satisfies p. Let Φ_M denote the system specification of M. We will regard $\Phi_M \rightarrow p$ as the formal definition of the fact that M satisfies p, denoted as $M \models p$.

Parallel composition corresponds to conjunction in temporal logic. It follows that, if M is a module of system S ($\equiv M \wedge M'$, for some M'), then $M \models p$ implies $S \models p$. In other words, every property one deduces (in a modular way) from a particular module will hold for the entire system.

We next review our temporal-logic formulation of A/G specifications. The specification $E \triangleright M$ of a module, where E is the assumption about its environment and M the guarantee by the module, essentially asserts that M holds at least one step longer than E does, i.e., if the environment has satisfied E up to the previous state then the module will satisfy M up to the current state. If E and M are expressible respectively as $\Box H_E$ and $\Box H_M$, where H_E and H_M are past formulae (whose truth depends only on the current and past states), $E \triangleright M$ (i.e., $\Box H_E \triangleright \Box H_M$) can be succinctly expressed as $\Box(\circledcirc \boxminus H_E \rightarrow \boxminus H_M)$.

Hiding is a common technique for making specifications more abstract and corresponds to existential quantification over flexible variables. A temporal formula φ will be written as $\varphi(\vec{x})$ to emphasize that the free (flexible) variables of φ are among the tuple $\vec{x}$ of variables. An A/G formula with assumption $\Box(\exists w\colon \boxminus H_E(x,y,w))$ and guarantee $\Box(\exists z\colon \boxminus H_M(x,y,z))$ is defined below; variable x is the input (output) variable of the module (environment) and y the output (input) variable of the module (environment).

$$\Box(\exists w\colon \boxminus H_E(x,y,w)) \triangleright \Box(\exists z\colon \boxminus H_M(x,y,z)) \stackrel{\Delta}{=} \Box[\circledcirc(\exists w\colon \boxminus H_E(x,y,w)) \rightarrow (\exists z\colon \boxminus H_M(x,y,z))]$$

Like in the case without hiding, the defining formula $\Box[\circledcirc(\exists w\colon \boxminus H_E(x,y,w)) \rightarrow (\exists z\colon \boxminus H_M(x,y,z))]$ says that $\exists z\colon \boxminus H_M(x,y,z)$ holds at least one step longer than $\exists w\colon \boxminus H_E(x,y,w)$ does.

The following lemma states that A/G specifications can be composed in a straightforward way:

Lemma 3.1 (Simple Composition) *Assuming that $w_1, \ldots, w_n, z_1, \ldots, z_n$ are distinct and*

no free variables become bound,

$$\models \bigwedge_{i=1}^{n} (\Box(\exists w_i \colon \boxminus H_{E_i}) \rhd \Box(\exists z_i \colon \boxminus H_{M_i}))$$
$$\rightarrow \Box(\exists w_1 \ldots w_n \colon \boxminus \bigwedge_{i=1}^{n} H_{E_i}) \rhd \Box(\exists z_1 \ldots z_n \colon \boxminus \bigwedge_{i=1}^{n} H_{M_i}).$$

Below is a more general rule for composition.

Theorem 3.2 (Composition) *Assuming that w, z, w_1, ..., w_n, z_1, ..., z_n are distinct and no free variables become bound,*

$$\frac{\begin{array}{ll} 1. & \models \Box\Big[(\exists w \colon \boxminus H_E) \wedge (\exists z_1 \cdots z_n \colon \boxminus \bigwedge_{i=1}^{n} H_{M_i}) \rightarrow (\exists w_1 \cdots w_n \colon \boxminus \bigwedge_{i=1}^{n} H_{E_i})\Big] \\ 2. & \models \Box\Big[\circledcirc(\exists x \colon \boxminus H_E) \wedge (\exists z_1 \cdots z_n \colon \boxminus \bigwedge_{i=1}^{n} H_{M_i}) \rightarrow (\exists z \colon \boxminus H_M)\Big] \end{array}}{\models \bigwedge_{i=1}^{n} (\Box(\exists w_i \colon \boxminus H_{E_i}) \rhd \Box(\exists z_i \colon \boxminus H_{M_i})) \;\rightarrow\; (\Box(\exists w \colon \boxminus H_E) \rhd \Box(\exists z \colon \boxminus H_M))}$$

Intuitively, Premise 1 of the above composition rule says that the assumption about the environment of a module should follow from the guarantees of other modules and the assumption about the environment of the entire system; while, Premise 2 says that the guarantee of the entire system should follow from the guarantees of individual modules and the assumption about its environment.

Our A/G formulation also permits expression of liveness properties in the guarantee part. Typically, they would resemble a Streett acceptance condition.

4 Summary of Research Results

We have obtained two main results from this research; one was planned in the proposal, while the other was not. The planned one states that, for assumption/guarantee formulae in PTL, realizability and hence synthesis are solvable in polynomial space (and therefore in exponential time). This to certain extent achieves our goal of reducing the high complexity of synthesis with arbitrary linear temporal formulae, which is double exponential. The unplanned result is a tool [25], which is still being improved and extended [24], for learning and researching ω-automata and temporal logics. During the course of this research, we came to realize that a deeper understanding of the relation between automata and temporal logic is needed. We developed the tool to facilitate our research. We anticipated that the tool would be useful for other researchers as well; its acceptance to TACAS 2007 was a clear evidence. The tool is freely available on the Web (`http://goal.im.ntu.edu.tw/`) and now has over 100 registered users.

We next give a bit more details of the main result on synthesis with A/G specifications. To simplify the presentation, we assume the module M shares two variables x and y with its environment E; variable x is the input (output) variable of the module (environment) and y the output (input) variable of the module (environment). Suppose the specification is given in the

form of $\Box H_E(x,y) \rhd \Box H_M(x,y)$, where $H_E(x,y)$ and $H_M(x,y)$ are past PTL formulae. Recall that $\Box H_E(x,y) \rhd \Box H_M(x,y)$ can be succinctly expressed as $\Box[\circledcirc \boxminus H_E(x,y) \rightarrow H_M(x,y)]$. The formula $H_M(x,y)$ in the guarantee part often would be a transition formula that relates the current value of y to the previous value of x, which would make realizability trivially true. However, $H_M(x,y)$ can be more complicated.

We say that $\Box H_M(x,y)$ constrains E if $\Box(\exists y : \boxminus H_M(x,y))$ is not a valid formula, i.e., its truth value depends on the remaining free variable x which is controlled by the environment E; otherwise, we say $\Box H_M(x,y)$ does not constrain E.

We claim that, if $\Box H_M(x,y)$ does not constrain E, then realizability of $\Box H_E(x,y) \rhd \Box H_M(x,y)$ is equivalent to satisfiability of the same formula, which is in PSPACE [19].

5 Concluding Remarks (Self Evaluation)

It seems very natural to assume that the specification for a module is given in the A/G style. We have just touched on a promising direction for synthesis. Efficient algorithms for system construction remain to be developed.

We had planned to exploit the program extraction feature of the Coq [21] proof assistant in this context. However, we were unable to complete this investigation due to rather involved technical difficulties, e.g., the need of a constructive version of temporal logic. We will leave this for future research. We still hope to become the first to successfully use a proof assistant to extract a reactive module from the proof of validity of the implementability formula corresponding to the given specification of the module. For specifications that defy fully algorithmic solutions for synthesis, the machine-assisted approach would provide the only alternative for the automation of open system construction.

This project involved an extensive set of fundamental mathematical and logic tools, including various classes of automata on infinite objects, games, and temporal logic. It helped the participating students get an early exposure to these foundational tools that should be very useful for their future research careers.

References

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specification. In *Proceedings of the 16th International Colloquium on Automata, Languages, and Programming, LNCS 372*, pages 1–17, 1989.

[2] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* The MIT Press, 1999.

[3] E. Grädel, W. Thomas, and T. Wilke. *Automata, Logics, and Infinite Games (LNCS 2500).* Springer, 2002.

[4] C.A.R. Hoare. An axiomatic basis for computer programs. *Communications of the ACM*, 12(10):576–580, 1969.

[5] C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.

[6] B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167:47–72, October 1996. An extended abstract appeared earlier in TAPSOFT '95, LNCS 915.

[7] Y. Kesten and A. Pnueli. Complete proof system for QPTL. *Journal of Logic and Computation*, pages 97–109, June 2001.

[8] O. Kupferman and M.Y. Vardi. Synthesizing distributed systems. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 389–398, 2001.

[9] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[10] P. Madhusudan and P.S. Thiagarajan. Distributed controller synthesis for local specifications. In *Proceedings of the 28th International Colloqium on Automata, Languages, and Programming (ICALP), LNCS 2076*, pages 396–407, 2001.

[11] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 377–408. ACM, 1990.

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

[13] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[14] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[15] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science (FOCS '90)*, pages 746–757, 1990.

[16] S. Schewe and B. Finkbeiner. Bounded synthesis. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA 2007), LNCS 4762*, pages 474–488, 2007.

[17] S. Schewe and B. Finkbeiner. Distributed synthesis for alternating-time logics. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA 2007), LNCS 4762*, pages 268–283, 2007.

[18] A.P. Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, Harvard, 1983.

[19] A.P. Sistla and E.M. Clarke. Complexity of propositional linear temporal logics. *Journal of the ACM*, 32:733–749, 1985.

[20] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Bc̈hi automata with applications to temporal logic. *Throretical Computer Science*, 49:217–237, 1987.

[21] The Coq Development Team. *The Coq Proof Assistant: Reference Manual*, 2004.

[22] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 133–191. MIT Press, 1990.

[23] Y.-K. Tsay. Compositional verification in linear-time temporal logic. In J. Tiuryn, editor, *Proceedings of the Third International Conference on Foundations of Software Science and Computation Structures, LNCS 1784*, pages 344–358. Springer, March 2000.

[24] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, W.-C. Chan, and C.-J. Luo. GOAL extended: Towards a research tool for omega-automata and temporal logic. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. To appear as an LNCS volume.

[25] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, K.-N. Wu, and W.-C. Chan. GOAL: A graphical tool for manipulating büchi automata and temporal formulae. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), LNCS 4424*, pages 466–471. Springer, March 2007.

[26] P. Wolper. Temporal logic can be more expressive. *Information and Computation*, 56(1-2):72–99, 1983.

Conference Trip Report
11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)
Braga, Portugal, March 26–30, 2007
(NSC 95-2221-E-002-127出席國際學術會議心得報告)

Yih-Kuen Tsay
蔡益坤
Department of Information Management
National Taiwan University

1 Conference Overview

The 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) was held in Braga, Portugal, March 26–30, 2007. TACAS has been one of the five main conferences of the European Joint Conferences on Theory and Practice of Software (ETAPS) since the inception of ETAPS in 1998. ETAPS is now the primary European forum for academic and industrial researchers working on topics relating to Software Science. ETAPS 2007 (March 24–April 1, 2007) also included 16 satellite workshops. The total number of participants was over 450.

TACAS and CAV are probably the two most important conferences on formal verification. The technical program of TACAS 2007 consisted of 2 invited talks and 54 (45 research and 9 tool) papers selected from 204 (170 research and 34 tool) submissions. Apparently, this was a very selective program. The conference proceedings was published as LNCS 4424 of Springer.

One nice thing about a federated conference like ETAPS is that you get many invited talks, one or two for each main conference plus several unifying ones. Below are some highlights. In his TACAS invited talk "There and Back Again: Lessons Learned on the Way to the Market", Rance Cleaveland from University of Maryland and Fraunhofer USA Center for Experimental Software Engineering and Reactive Systems Inc. talked about his experience in commercializing his research results and tools. As he contended, one should always try to package a tool in a way that suits the user's daily practice. In another TACAS invited talk "Verifying Object-Oriented Software: Lessons and Challenges", K. Rustan M. Leino from Microsoft Research, Redmond, USA introduced the Spec# system. He also did a demo of the system. In a unifying invited talk "Contract-Driven Development", Bertrand Meyer from ETH Zürich, Switzerland reflected on the development of the Eiffel approach and tool. Eiffel had just won the prestigious

ACM Software System Award (former winners include UNIX, TeX, TCP/IP, SMALLTALK, WWW, Tcl/Tk, Java, etc.). Meyer jokingly dubbed the Spec# system as Eiffel in braces. Aspect-oriented programming has emerged as a new paradigm in the design and development of software systems. In his FOSSACS invited talk "Formal Foundations for Aspects", Radha Jagadeesan from DePaul University, USA argued that aspects are no more intractable than stateful higher order programs.

2 Presentation of My Paper

I presented the following paper which I co-authored with four of my students:

Title: GOAL: A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae
Authors: Yih-Kuen Tsay, Yu-Fang Chen, Ming-Hsien Tsai, Kang-Nien Wu and Wen-Chin Chan

The paper describes a tool, named GOAL, that we have been developing for learning and researching omega-automata and linear temporal logics. The tool had earlier been introduced in an informal workshop FMEd affiliated with FM 2006. My Ph.D. student Yu-Fang Chen also attended the conference. To allow him to gain some experience and to get exposed, I asked him to do the demo part of the 30-minute presentation.

Our tool was well received by the participants. Several of them came to greet us right after our presentation. One of them even suggested an efficient procedure, though not difficult to conceive, for testing the equivalence between a Büchi automaton and a temporal formula. The equivalence test was not directly supported by our tool at that time. His suggestion prompted us to implement it in the next version.

3 Meeting Old Friends

I met Parosh Abdulla, a former colleague at Uppsala University back in 1993-5 while I was a postdoctoral researcher there. We had worked together with Čerāns and Jonsson to develop fundamental theorems for the verification of infinite-state systems. Our joint work turned to be an influential one, accumulating a considerable number of citations over the years. Abdulla is still very actively pursuing various directions of the same research topic. Through Abdulla, I also got to know Ahmed Bouajjanni from Paris 7 during the conference. Bouajjanni is also very active in the same area and a frequent co-author of Abdulla.

I was also very happy to meet for the third time Ridha Khedri. He visited Taipei for the ATVA conference in 2003. I met him again in FM 2006, of which his institute was the host, just a few months prior to TACAS. Khedri has been a faculty member at McMaster University, Canada for some time. He is an expert in algebras and their applications. It is always very nice to have someone showing you things, technical or non-technical, from a different perspective. We had a dinner and a few walks around the host town. Khedri originally was from Tunisia and from his story telling I realized how little I knew about North Africa.

Finally, I must mention Moshe Vardi, who is considered by many father of the automata-theoretical approach to model checking. Vardi visited Taipei for the ATVA conference in 2004 and subsequently helped me to get the proceedings of ATVA 2005 (of which I was PC co-chair) published in the LNCS series of Springer. He seemed to be very interested in the GOAL tool that I presented and pointed out a noteworthy Büchi complementation algorithm by him, Kupferman, and two other authors. The algorithm has now been implemented in GOAL.