Distributed Games

Matthew Hague

Computing Laboratory
Oxford University

Abstract. Program synthesis is the automatic construction of a correct program design for a given specification. We present a brief survey of synthesis techniques and paradigms. In particular we describe the synthesis of distributed reactive systems and the use of multiplayer games as a unifying approach. We present some open problems in the area and look briefly at further models such as knowledge-based or timed systems.

1 Introduction

Automatic verification of systems has proved a successful application of formal methods in computer science. These techniques are used in industry and are the focus of much research. However, it may be argued that verification is part of the debugging process, identifying errors rather than preventing them.

Given a specification, program synthesis is the automatic construction of a design that is guaranteed to be correct. The synthesis problem has as many variants as there are system paradigms. One such distinction lies between *open* and *closed* systems. Classical synthesis has concentrated on closed systems, where both program and user work together to find the required output. In contrast, an open system assumes a hostile environment. A correct design is one where the program is able to handle a user who may throw a spanner in the works. That is, for all possible inputs, the program is able to produce a correct output.

In the open setting the notion of a two-player game arises naturally. The two players being the environment ($\forall$) and the system ($\exists$); the environment wins if the specification is violated and the system wins otherwise. A correct design is a winning strategy for the system. In other words, no matter how the environment acts, the system will not violate the specification.

Reactive systems constitute a further variation of program synthesis. In this model systems are intended to run forever, reacting to user input. For example, a web-server is designed to serve user requests (whatever they may be) and to do so indefinitely.

Finally, reactive systems are frequently distributed — that is, a number of modules working together against a single environment. The notion of a game extends to an analogous multiplayer setting.

In this paper we present a brief survey of program synthesis, focusing on the distributed, reactive paradigm. We then explain how games are used as a unifying approach in this setting. Finally we identify open problems in the area and look at more exotic models such as timed or probabilistic systems.

2 Synthesis and Control

2.1 Church's Problem

In 1963 Church published a summary of then recent work concerning mathematics and finite automata [1]. In this paper he described automata, restricted recursive arithmetic and regular languages. He then presented three problems connecting automata and languages: the simplification problem, the decision problem and the synthesis problem.

The decision problem is known as the model checking problem today: given a specification and a proposed solution, determine whether the solution implies the specification. The synthesis problem, which is the focus of this survey, requires that, given a specification, if it is possible to construct an automaton satisfying the specification, that automaton is to be constructed; otherwise a negative answer should be returned. The simplification problem is to find the "simplest" equivalent specification for a given notion of simplicity.

Church summarises several results concerning the synthesis and decision problems that use different fragments of restricted recursive arithmetic or regular properties as their specification language. Church identifies several open problems, the last of which is known as "Church's Solvability Problem" and is described succinctly in [2].

Given an S1S relation $R \subseteq (2^I)^\omega \times (2^O)^\omega$, where I and O are sets of input and output signals respectively, find a function $f : (2^I)^\omega \to (2^O)^\omega$ such that for all $x \in (2^I)^\omega$ we have $R(x, f(x))$. If such a function does not exist, a negative result must be returned.

In 1969, Büchi and Landweber presented the first solution to this problem in the finite-state case [3]. They also link the problem to mathematical games, crediting McNaughton for the initial observation. In the game paradigm the environment (responsible for the inputs) plays against the system (responsible for outputs). The required f is then a winning strategy for the system. Although the notion of games is secondary to their result, Büchi and Landweber note that the game paradigm in this context adds "appealing flavours" to both automata theory and game theory.

However, Büchi and Landweber's solution to Church's Solvability problem is not straightforward and has a high computational complexity. In 1972, Hossley and Rackoff [4] and Rabin [5] provided new solutions to the problem using tree automata. A winning strategy is a tree whose branches correspond to the different possible inputs and prescribed outputs; therefore, an automaton can be constructed that accepts all trees whose paths ("computations") satisfy the specification. The synthesis problem is then reduced to finding a witness to the non-emptiness of the constructed tree automata.

2.2 Reactive Systems

In the early 1980s authors such as Clark and Emerson [6] and Manna and Wolper [7] considered the synthesis problem in a distributed setting. The delay in the consideration of this problem is, in part, attributable to the search for an appropriate specification language [8].

Early methods of synthesis in a concurrent setting extracted the program from a proof that a solution *could* be derived [7, 6]. That is, it is possible for the modules to work together to produce the correct answer. Such an approach does not consider the effect of a hostile environment and, as such, applies to *closed* rather than *open* systems.

In 1985 Pnueli and Harel [9] proposed the reactive/transformational dichotomy: whereas transformational systems work towards a final solution, reactive systems are designed to run forever, reacting to environment input. For example, web servers, operating systems and microwave ovens are all reactive systems. These systems are therefore *open* and Pnueli and Harel used a "black-cactus" analogy — with many inputs and outputs – as opposed to the more traditional "black-box".

Because many concurrent programs fall into the reactive category, the work of Pnueli and Hoel helped establish temporal logic as the principle specification language for both concurrent and reactive systems [8].

In 1989 Pnueli and Rosner [8] provided a solution to the synthesis problem for single processor reactive systems with specifications given in Linear Temporal Logic. In their setting a linear specification $\phi(x, y)$ — where x refers to the input and y to the output at each time step — is augmented to form the branching time formula $(\forall x)(\exists y)A\phi(x, y)$, where "A" means "on all paths". Their solution reduces the problem to a non-emptiness test for tree automata that accepts infinite strategy trees. The procedure is doubly exponential in the size of the specification.

The problem they consider is closely related to Church's Solvability problem. However, their solution is more general than the solutions provided by Büchi *et al* and Rabin in the sense that they allow infinite state solutions rather than only finite state solutions.

2.3 Implementations

Because of its high complexity, the method of Pnueli and Rosner has performed poorly in practice. In [10], Harding, Ryan and Schobbens present a new algorithm that avoids determinising Büchi automata. Although incomplete and still 2EXPTIME-complete, an implementation of their algorithm has shown promising results.

There have also been attempts to find synthesis problems with lower complexities. In [11], Alur and La Torre give complexity results for various fragments of LTL. In the case of LTL with only conjunction and diamond, complexity becomes PSPACE-complete.

2.4 Distributed Systems

In [12] Pnueli and Rosner extended their work in [8] to a synchronous distributed setting. In this setting the synthesis problem is to find finite state programs

$f_1, \ldots, f_k$ for the architecture built from processes $C_1, \ldots, C_k$ and their interconnection scheme. The joint (synchronous) behaviour of the system must then satisfy the linear temporal logic specification ϕ.

Their first result — derived from the work on multiple person alternation by Peterson and Reif [13] — is that in this distributed model the synthesis problem is undecidable. The proof is given by a reduction of the halting problem. However, in the case of hierarchical architectures such as a strict pipeline (as shown in Figure 2.4 for the five processor case) the problem is shown to be decidable in non-elementary time.

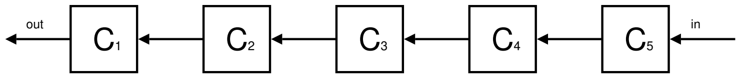

Fig. 1. A five processor pipeline.

Pnueli and Rosner present two different methods for distributed synthesis. The first method involves the synthesis of each processor in turn. The algorithm is recursive and constructs a tree automaton that accepts a strategy for the tail of a pipeline iff there exists a strategy for the head. The second method constructs a single processor strategy which is then decomposed into a distributed strategy. This method uses similar techniques to the first. The decomposition result can be extended to all acyclic architectures, however, it is not complete in the sense that a distributed strategy may exist where a decomposition does not.

2.5 Incomplete Information

Until 1997, research into program synthesis had only considered the case of complete information. This is when the program sees all inputs. In the case of incomplete information the environment may execute "hidden moves" and our strategy must operate independently.

Kupferman and Vardi addressed this problem for sequential systems in [14] and showed that for LTL, CTL and CTL* specifications incomplete information did not make the problem harder – that is, the sequential synthesis problems remain 2EXPTIME, EXPTIME and 2EXPTIME-complete respectively.

In the case of LTL the problem of incomplete information can be solved by non-deterministically guessing the hidden moves. However, in the branching case we require that nodes of the tree whose input paths co-coincide (i.e. when hidden events are removed from the history) prescribe the same moves. This is a non-regular consistency property, and cannot be evaluated directly using automata-theoretic methods. Kupferman and Vardi in effect sidestep this problem by constructing an automaton that accepts all strategies that, when padded with the possible hidden moves, satisfy the specification.

Kupferman and Vardi extended these techniques in 2001 to solve the synthesis problem in a distributed setting for a CTL* specification, with or without complete information, and with or without communication delay, in a one or two way communication paradigm and in a pipeline or ring architecture [15]. The problems are shown to be of non-elementary complexity.

2.6 The Control of Discrete Event Systems

A closely related problem to program synthesis is the control of discrete event systems. This problem was introduced by Ramadge and Wodham at the end of the eighties [16]. The control problem consists of a plant and a specification.

A plant is an automaton that describes all possible sequential behaviours of the system. Behaviours are a sequence of events, some of which are uncontrollable or unobservable. Uncontrollable and unobservable events may correspond to external input and hidden moves of the environment respectively. For example, in a microwave system, button presses may be uncontrollable events and cooking may be controllable. An example of an unobservable event may be a user placing a metallic object into the oven.

The specification is given as a set of admissible behaviours of the system. For example, we may not allow a cook event to occur whilst the microwave door is open. Additionally, we may require that it is always possible to reset the microwave to its initial state.

The controller synthesis problem is then to construct a controller, if one exists, that restricts the plant such that all of its restricted behaviours are admissible. Such a system is called a supervised system. A controller is a map from sequences of observable events to sets of allowed events. Intuitively, the controller presents a selection of possible next moves for any given history of events that can be observed. Uncontrollable events can never be restricted.

2.7 Control and Synthesis

In 2000, Kupferman, Madhusudan, Thiagajaran and Vardi presented a reduction of the program synthesis problem to the control problem [17]. This reduction is simply a case of constructing a universal plant which describes all possible sequences of inputs, outputs and hidden events. Inputs are uncontrollable and hidden events are unobservable. A controller for the plant corresponds to a program strategy for the synthesis problem.

In their paper, Kupferman *et al* consider a more general synthesis problem, where the environment may enable or disable its moves. These are called reactive environments. The solution is by reduction to the control problem, which is solved using automata-theoretic methods. The problem for reactive environments is shown to be 3EXPTIME-complete for CTL* specifications and 2EXPTIME-complete for CTL specification. This means that reactive environments lead to an exponential blow-up over the EXPTIME-complete and 2EXPTIME-complete results for maximal (non-reactive) environments.

Many results from program synthesis can be applied to the control problem. However, whilst distributed synthesis is decidable for hierarchical structures, the result only carries through for pipeline architectures. In [18], Madhusudan and Thiagajaran argue that the global nature of the specifications is unreasonably expressive. Instead they consider local specifications and show that the controller synthesis problem is decidable iff the architecture is a clean (or doubly-flanked) pipeline, or a sub-architecture of a clean pipeline. A clean pipeline is similar to a pipeline except that the environment may provide input to both extremal nodes.

2.8 Asynchronous Systems

So far we have only discussed distributed synchronous settings. The asynchronous setting was first considered by Pnueli and Rosner in [19]. However, they only considered global plants with asynchronous environments and their decision procedure was of non-elementary complexity. Madhusudan and Thiagarajan revisited this problem in 2002 using local plants with local environments who communicate via handshakes [20].

To obtain decidability results Madhusudan and Thiagarajan impose three restrictions on the problem. Firstly, the specification must be *robust*. That is, independent events may occur in any order without affecting satisfaction. Secondly the program strategy can only keep track of the time, rather than the complete history. Finally, at each time step the strategy may only recommend events that involve communication between the same set of processes.

It should be noted that Madhusudan and Thiagarajan impose very few restrictions on the architecture of the system. The model considered in [20] requires deterministic plants and single readers and writers. The authors, however, conjecture that such restrictions can be removed, and, indeed, when non-deterministic plants are allowed, a finite memory may be encoded.

The decision procedure provided runs in doubly exponential time, improving the non-elementary results of Pnueli and Rosner. The algorithm involves creating an automaton that feeds all possible linearisations of program traces into another; this second automaton checks the traces against the specification. Program strategies are those accepted by the resulting automaton.

3 Games and Synthesis

In Section 2 we gave an overview of the history of program synthesis. The techniques that have been used traditionally vary depending upon the exact nature of the problem being solved. However, in almost all cases automata-theoretic techniques are used. Recent work on synthesis has used game models. In this section we introduce the notion of mathematical games and explain some of the applications they have in program synthesis.

We begin by presenting some of the basic concepts of games and how they relate to program synthesis. We then describe recent attempts to unify the many variants of the synthesis problem using games.

3.1 Two Player Games

Intuitively, a two-player game consists of two players, a number of moves each player can make (and a set of rules governing what moves can be made, by whom and when) and a condition that needs to be met for a player to win the game. A player is said to have a *winning strategy* if he has a method of selecting his moves such that, no matter how his opponent behaves, he is always guaranteed to win.

For a simple example, consider a game of Noughts-and-Crosses (or Tic-Tac-Toe) played on a two by two grid. It is easy to see that player one (who moves first) is able to win every time and player two can do nothing to prevent this. In other words, player one has a winning strategy (whereas player two does not).

In the synthesis setting, the two players are the program and the environment. The program wins if the program run never violates the specification. The environment tries to give the program a sequence of inputs such that the program is unable to behave in an admissible fashion. If the program has a winning strategy it will specify an output for any sequence of inputs and the output will always meet the specification. If we can find a winning strategy for the system, we have a program design that is guaranteed to be correct.

More formally we represent a game as a graph (V, E) and a winning condition W. The set of vertices V of the graph is the disjoint union of a set of player one states and a set of player two states. A play proceeds from a start vertex v. If v belongs to player one, then player one makes a move — that is, player one chooses a next vertex that is connected to v by an edge. Likewise, if the vertex belongs to player two, player two chooses the next vertex. Play continues in the manner, creating a sequence $v_0, v_1, \ldots$ of vertices.

The winning condition determines who has won a play and when. A simple winning condition may be a set of goal states for each player. If player one can reach a goal state, then they win the game, and likewise for player two. In this case it is possible that each player can keep the other from winning, but cannot win themselves. Consequently, it is not always possible for either player to have a winning strategy.

If we are discussing reactive systems — which never terminate — we will be considering infinite plays. In this case it is likely that we will be interested in which vertices occur in the sequence, or which vertices occur infinitely often. For example, we may not want to reach a deadlock state, or we may want to return to the initial state infinitely often.

A strategy is a function $\rho(\widehat{v}v) = v'$ that maps any sequence of vertices ending with a v to a vertex v' that is connected to v by an edge. A strategy is a winning strategy from a given vertex iff all plays from that vertex which follow the strategy satisfy the winning condition. A memoryless strategy is a strategy where $\rho(\widehat{v}v) = \rho(\widehat{w}v)$ for all $\widehat{v}, \widehat{w}$ and v.

The link between two player games and program synthesis was first observed by McNaughton and reported by Büchi and Landweber as an aside to their solution to Church's problem [3]. However, this link was not significantly developed in a sequential setting. In the distributed case, however, there has been more interest in the link between games and synthesis.

3.2 Multiplayer Games

In a distributed setting two player games may not provide a suitable model. A more general notion of games has many players rather than just two.

In [13], Peterson and Reif introduce alternating multi-player games where the players are divided into two teams, $\exists$ and $\forall$. They show that, in general, deciding whether a team has a winning strategy is undecidable. However, they also show that in the case that the game is hierarchical – that is, all resources available to player i are available to player $i-1$ – the problem is non-elementary decidable.

Intuitively, we can interpret $\exists$ as a representation of the system, who has to find a way to win. $\forall$ represents the environment who is trying to defeat the system. The synthesised program must give a correct response for all behaviours of the environment.

This work by Peterson and Reif formed the basis of the initial work by Pnueli and Rosner on distributed synthesis [12]. They showed that for non-hierarchical architectures, the synthesis problem is undecidable, and that the problem is non-elementary decidable for hierarchical architectures. However, the notion of games was not used directly.

3.3 Games and Control

In 2002, Arnold, Vincent and Walukiewicz presented a solution to the distributed control problem with partial observation [21]. Their solution involves transforming the specification into a μ-calculus formula whose models are exactly those controllers that satisfy the problem. However, the standard μ-calculus is not expressive enough to formulate the property that the controller cannot take into account unobservable events. To overcome this problem the μ-calculus is extended with a loop proposition ($\circlearrowleft_a$) that holds at state q iff there is a transition from q to q labelled with an a.

The decision procedure for the modal-loop μ-calculus is given via a translation to “loop automata” whose semantics is given in terms of a two player game. It is then shown that an automaton has a model only in case that there is a winning strategy for player 0.

The control problem considered by Walukiewicz *et al* consists of a plant, an automaton giving the global specification and an automaton giving the local specification for each processor. A solution for each processor is found by constructing a controller that satisfies its local specification and satisfies the global specification when “quotiented” by the local specifications for the other processors, any processors that have already been created and the behaviour of the plant. In order to obtain decidability, only one local specification is allowed to use the loop proposition.

4 Unification Using Games

In section 2 we discussed several synthesis problems and their various solutions. This work has shown that synthesis is a hard and frequently undecidable task. However, because these techniques and proofs are specific to the precise nature of the problem being considered, it is difficult to see what makes synthesis so difficult and draw general conclusions. This was the motivation behind recent work (2003) by Mohalik and Walukiewicz who proposed a game framework for the formalisation of distributed synthesis [22].

We will begin by summarising the main synthesis problems. We will then introduce the games framework of Mohalik and Walukiewicz and describe how to solve the synthesis problem in this setting. Finally we will give an overview of how this framework can encode the various synthesis problems.

4.1 A Summary of Distributed Synthesis Problems

The four synthesis problems considered by Mohalik and Walukiewicz are as follows:

- **Pipeline Synthesis** — A pipeline is a sequence of processes $C_1, \ldots, C_n$ each communicating with their neighbour in a single direction. The environment provides input to one end of the pipeline, and output occurs at the other. Execution is synchronous and proceeds in communication rounds.
  A pipeline controller is a tuple $\langle f_1, \ldots, f_n \rangle$ where f_i controls process C_i. The synthesis problem is, given a specification, determine $\langle f_1, \ldots, f_n \rangle$ such that the behaviour of the pipeline meets the specification.
- **Local Specifications** — The synthesis problem with local specifications is similar to the pipeline problem. However, a specification is given for each individual process instead of a single specification for the system as a whole. As a result we are able to consider doubly flanked pipelines. That is, pipelines whose external processes both take input from the environment.
- **Communicating Machines** — In the communicating machines paradigm we fix a set of n processes, each with an alphabet of environment and controllable actions. We assume that no two processes share the same environment actions, but that they may share controllable ones. These common actions provide a method of synchronisation between processes. Each process alternates between taking environment input and producing controllable output. To make the problem decidable only *trace closed* specification are considered. That is, independent actions may occur in any order without changing the satisfaction of the specification. We also require that programs only take the clock time into account when choosing their next output, and that at each time step all next actions that the program may perform are shared by the same set of processes.
  The synthesis problem is then to find a controller for each process such that all possible execution paths satisfy the given specification.

– **Discrete Event Systems** — In the control problem for discrete event systems we are given a plant over a set of actions Σ. A plant is a deterministic finite state automaton that identifies every possible sequence of actions that a process may perform. Plants have controllable and unobservable actions. A controller can use the execution history to determine which controllable actions to restrict at a given stage, however, the controller cannot take unobservable actions into account.
  The distributed control problem is, given a plant P and languages $M, N \subseteq \Sigma^*$, find controllers $C_1, \ldots, C_n$ such that,

$$M \subseteq \mathcal{L}(P \times C_1 \times \ldots \times C_n) \subseteq N$$

4.2 A Games Model

Mohalik and Walukiewicz introduce the following notion of distributed games. A distributed game is constructed from a number of local games $\langle P, E, T \rangle$ without a winning condition, where P is the set of player moves, E the set of environment moves, and T the set of game moves. It is assumed that player turns alternate. The distributed game has a global winning condition.

For local games $G_i = \langle O_i, E_i, T_i \rangle$ for $i = 1, \ldots, n$, a distributed game is $\mathcal{G} = \langle P, E, T, Acc \subseteq (E \cup P)^\omega \rangle$, where:

1. $E = E_1 \times \ldots \times E_2$
2. $P = (P_1 \cup E_1) \times \ldots \times (P_n \cup E_n) \setminus E$
3. From a player's position we have $(x_1, \ldots, x_n) \to (x'_1, \ldots, x'_n) \in T$ iff $x_i \to x'_i \in T_i$ for all $x_i \in P_i$ and $x_i = x'_i$ for all $x_i \in E_i$.
4. From an environment position, if we have $(x_1, \ldots, x_n) \to (x'_1, \ldots, x'_n) \in T$, then for every x_i, either $x_i = x'_i$ or $x_i \to x'_i \in T_i$.
5. *Acc* is any winning condition.

There are several points to notice about distributed games. The first is that the environment does not need to move all positions during its turns, however, a player must always move immediately. In this way, environment moves may be blocked by global restrictions. Because of this we are able to assume that the environment may move to any player's position in the local games, putting any restrictions on moves into the definition of the global game.

A solution to the synthesis problem in this setting is a *distributed strategy.* A distributed strategy is a tuple of local strategies, one for each local subgame. In this sense the players do not work together against the environment, and any communication between them must go through the environment. As a result it is possible that a global strategy may exist where a distributed one does not.

4.3 Solving Distributed Games

Two tools are provided by Mohalik and Walukiewicz that enable us to determine a winning strategy for a distributed game. These are DIVIDE and GLUE. The essence of the decision procedure is that we DIVIDE and GLUE the game until we are left with a two player parity game. Decidability follows from results on two-player games.

The DIVIDE operation is predicated on players 0 and n being able to determine the global state of the game from their local view — this property is called i-deterministic, for player i. If this is the case, then, for the game $\mathcal{G}$ built from $\mathcal{G}_0, \ldots, \mathcal{G}_n$, we can build an equivalent game $\tilde{\mathcal{G}}$. This game has one less player and is built by combining $\mathcal{G}_0$ and $\mathcal{G}_n$ to form $\mathcal{G}'$ and then constructing $\tilde{\mathcal{G}}$ from $\mathcal{G}', \mathcal{G}_1, \ldots, \mathcal{G}_{n-1}$. The function $flat$ is defined to translate between positions of the two games; that is $flat((x_0, x_n), x_1, \ldots, x_{n-1}) = (x_0, x_1, \ldots, x_{n-1}, x_n)$. We then have the following theorem.

Theorem 1. *Let $\mathcal{G}$ be a 0-deterministic and n-deterministic distributed game of $n+1$ players. For every position η of $\mathcal{G}$: there is a distributed winning strategy from η iff there is one from $flat^{-1}(\eta)$ in $\tilde{\mathcal{G}}$.*

The GLUE operation is used to determinise the game for a particular player. The operation is analogous to the standard subset construction method for determinising automata. Mohalik and Walukiewicz identify two sets of conditions — I-gluable and II-gluable — under which the GLUE operation is possible; we omit the details. As shown by the following theorem, gluing a game results in an equivalent game, $\tilde{\mathcal{G}} = \text{GLUE}(\mathcal{G})$.

Theorem 2. *Let $\mathcal{G}$ be a I-gluable or II-gluable game. There is a distributed winning strategy from a position η in $\mathcal{G}$ iff there is a distributed winning strategy from the position $\tilde{\eta}$ in $\tilde{\mathcal{G}}$.*

We are then able to GLUE a game – to ensure determinism for players 0 and n – before applying DIVIDE, provided all preconditions are met along the way. This process can be repeated until we are left with a two-player game. Then we can apply known techniques to solve the game.

Theorem 3 ([23–25]). *Every (two-player) game with regular winning conditions is determined, i.e., every vertex is winning for the player or for the environment. In a parity game a player has a memoryless winning strategy from each of his winning vertices. Similarly for the environment. It is decidable to check if a given vertex of a finite game with a regular winning condition is winning for the player.*

4.4 Encoding Distributed Synthesis Problems

We now describe briefly how several synthesis problems can be encoded in this game setting.

- **Pipeline Synthesis** — we assume that the specification is given as an automaton over an alphabet that is the disjoint union of alphabets $A_1, \ldots, A_n$. A_n and A_0 represent the environment inputs and outputs respectively, and A_i is used for communication between processes C_{i+1} and C_i.

The game is constructed as follows. There are $n+1$ players; the first controls the automaton – making any non-deterministic choices – while the others represent the n processes. The environment provides the input A_n and "passes on" output from one process to its neighbour and the automaton player. The winning condition is simply a translation of the automaton acceptance condition.

– **Local Specifications** — The encoding for a doubly-flanked pipeline with local specifications is very similar to the encoding of the pipeline synthesis problem. To account for the local specifications we do not have an automaton player. Instead, given the regular languages $L_1, \ldots, L_n$ we define our acceptance condition to be that the projection of player i's inputs and outputs is in L_i.

  Additionally, we need to define player one so that the extra input is taken into account. Furthermore, to meet the conditions for *gluability* we encode the deterministic parity automaton that accepts L_1 into player one's game. The winning condition for player one is that the automaton's acceptance condition has been met.

– **Communicating Machines** — A similar technique is used to encode Madhusudan and Thiagajaran's communicating machines model. In this model each process is given as a plant that needs to be controlled. Players one to n are defined to represent these plants, with the appropriate restrictions for decidability as discussed in section 4.1.

  Player zero encodes the parity automaton representing the trace closed specification. Because, in this encoding, the environment may decide to block certain processes forever, the automaton maintains a component specifying which processes are active. It can then simply ignore the inactive processes. The environment then decides which actions should be executed, to which the active processes must respond. The winning condition is given by the acceptance condition of the specification automata.

– **Discrete Event Systems** — unfortunately Mohalik and Walukiewicz were unable to provide an adequate encoding of Ramadge and Wodham's discrete events systems in their games framework. However, they do provide a presentation of the problem as a game. The details are omitted.

5 Team Games

In 2004, Gastin, Lerman and Zeitoun [26] introduced a notion of games for asynchronous systems. This model provides a solution to the distributed control problem that subsumes much of the previous work on program synthesis. In particular, the authors cite [15, 18, 20, 22] and [12]. At the end of the paper it is shown how the work of Mohalik and Walukiewicz — described in section 4 — can be encoded in their model.

We begin by describing the notion of games put forward by Gastin *et al*, and then describe how the work of Mohalik and Walukiewicz is subsumed by the approach.

5.1 Distributed Systems

Rather than representing each process as a player of a distributed game, Gastin *et al* view a distributed system as a single asynchronous model, where actions are players divided into two teams: the actions of the system and the actions of the environment. An architecture has a finite set of actions (Σ) and a finite set of processes ($\mathcal{P}$). Each action has a read and write domain (R and W) that specify its set of read and write processes.

Previous work that the team games subsume has shown that the ability to observe properties of particular linearisations of process actions often leads to undecidability. Hence, Gastin *et al* allow only trace recognisable specifications. That is, a dependency relation between actions is given, and independent actions may occur in any order without affecting satisfaction. This results in the following restrictions on architectures,

$$\begin{array}{ll} \forall a \in \Sigma, & \emptyset \neq W(a) \subseteq R(a) \\ \forall a, b \in \Sigma, & R(a) \cap W(b) = \emptyset \iff R(b) \cap W(a) = \emptyset \end{array}$$

A distributed game over an architecture is defined via a disjoint set of states for each process and a transition relation per action. Each relation describes a transition from a product of reader process states to a product of writer process states for the action. A global state is a product of a local state for each process.

An action moves play from one global state to another iff the move preserves the local states of all processes not in the write domain of the action, and the transition relation for the action admits the product of local reader states to the product of local writer states.

A play is then a sequence of global states and action labelled transitions between them. The (trace equivalent) winning condition is defined as a set of words over tuples. Each tuple contains the action played and the projection of writer states for that action after the move had taken place.

For program strategies, Gastin *et al* use a slightly different notion of memory than other models. Instead of a history of local states, processes are able to remember information collected from other processes throughout the play. However, it can also be shown that, for all games $\mathcal{G}$ there exists an equivalent game $\mathcal{G}^\mu$ which has memoryless winning strategies.

Proposition 1. *Let $\mathcal{G}$ be a distributed game and let μ be a distributed memory on $\mathcal{G}$. One can construct a distributed games $\mathcal{G}^\mu$ such that there exists a μ-WDS (Winning Distributed Strategy) for $\mathcal{G}$ iff there exists a WMDS (Winning Memoryless Distributed Strategy) in $\mathcal{G}^\mu$. Moreover, if $\mathcal{G}$ is finite and μ is realised by a finite asynchronous automaton, then $\mathcal{G}^\mu$ is finite.*

Decidability is obtained via a translation into a "global" two-player game $\tilde{\mathcal{G}}$. In this game player 0 mimics a distributed strategy and player 1 "tests" the strategy. At each stage, player 1 can ask player 0 to play a particular action. In this way, all possible linearisations of plays may be investigated. Player 0 can either play the desired action, or refuse. To ensure that player 0 is following a distributed strategy, player 1 can reset the game. In the continuing play, player 0 must act in a manner that is consistent with the previous plays. Additionally, player 1 must act fairly; that is, he may only reset a finite number of times, and he must request that player 0 performs each action an infinite number of times — this ensures that player 0 has the opportunity to act out any strategy. Player 0 wins if the trace induced by the actions he played meet the winning condition of the original game.

Theorem 4. *The following conditions are equivalent for a distributed game $\mathcal{G}$:*

1. *There exists a WMDS for team* 0 *in the distributed game* $\mathcal{G}$.
2. *There exists a WMS for player* 0 *in the global game* $\tilde{\mathcal{G}}$.
3. *There exists a WS for player* 0 *in the global game* $\tilde{\mathcal{G}}$.

The global game $\tilde{\mathcal{G}}$ can be transformed into a parity game. Known methods can be used from then on.

5.2 Encoding Walukiewicz *et al*

Gastin *et al* provide a somewhat elegant encoding of the Mohalik and Walukiewicz's distributed games. The idea is to associate an action i with each player i of a Walukiewicz game. The players form one team, whilst the environment forms a team on its own. The set of processes in the team game matches the set of processes in the Walukiewicz game, similarly for the local states of each process. The reader and writer processes of an action i is simply $\{i\}$ and the transition relation for each process is encoded verbatim. The reader and writer sets for the environment are the set of all players and its transitions are all transitions in the Walukiewicz game that move from an environment to a player position.

The main difference between the two notions of games is the memory. Team games allow the memory to contain all information gathered from all processes throughout the history. Walukiewicz games, however, only allow the observation of local states. Consequently, the notion of memory for the corresponding team game needs to be restricted to a local view. With this restriction it follows that a Walukiewicz game is equivalent to its team game encoding. Given a Walukiewicz game $\mathcal{G}$ and its equivalent team game $\bar{\mathcal{G}}$, we have the following result.

Theorem 5. *The players have a WS in* $\mathcal{G}$ *iff team* 0 *has a local WDS in* $\bar{\mathcal{G}}$.

The absence of a satisfactory encoding of the distributed control problem was a shortcoming of the work of Mohalik and Walukiewicz. Gastin *et al* state briefly at the end of their paper that the control problem presented in [20] has a straightforward translation into team games. This is because transitions are local (in the case of the environment) or the product of local transitions.

6 Open Problems

The aim of the Walukiewicz *et al* in presenting a unifying approach to Distributed Synthesis is to identify classes of architecture-specification pairs for which the synthesis problem is decidable. The two mathematical tools, DIVIDE and GLUE, are enough to solve the problems considered. However, further tools may be required for different architecture-specification pairs. Additionally, the current decidability results are specific to particular problems; we may use the framework to identify more general decidability results.

Presently, the Team Games approach of Gastin *et al* has been shown to be decidable for trace recognisable winning conditions and undecidable for rational winning conditions. Classical winning conditions, such as Büchi, liveness and parity conditions have not been studied in general [27]; however, recent work by Gastin *et al* has shown that for series-parallel systems, *controlled reachability conditions* are decidable. These include reachability and safety conditions as well as recognisable ones [27].

Additionally, the proof of decidability given by Gastin *et al* does not present a clear method of strategy construction. Therefore, a more direct method of constructing program strategies for a specification is an open problem.

Finally, Gastin *et al* identify the distributed synthesis problem for non-co-graph alphabets. Broadly speaking, in a co-graph alphabet, if an action a is dependent on an action b, then b is also dependent on a (the formal definition is more subtle). The problems considered in the team paradigm so far have all used a co-graph model of traces.

7 Different Models

The synthesis problem has many different flavours. In this section we will briefly discuss the synthesis of timed systems, probabilistic systems and knowledge based systems.

7.1 Timed Systems

The controller synthesis problem for a given plant can be augmented with a notion of time. This can be achieved by presenting the plant using the timed automata of Alur and Dill [28, 29]. We can then reduce the synthesis problem, in the sequential setting, to a timed game between two players. We then search for a winning strategy for the controller.

Even in the sequential setting, many problems are undecidable. (See [30] for an introduction to the techniques and decidability results). Decidability can be obtained by imposing restrictions on the specification or by limiting the resources (such as the number of clocks) available to the controller. However, there is very little work available that studies the synthesis of timed systems in a distributed setting.

7.2 Probabilistic Systems

In [31] Walukiewicz observes that some distributed communication problems are well-known to only have probabilistic solutions; therefore we may wish to study randomised strategies in a distributed setting.

Recent work by Luca de Alfaro *et al* [32, 33] considers this problem in a two player setting. In this model, we extend the usual two player game with probabilistic transitions and probabilistic strategies — that is, the next state or move is given by a probability distribution.

7.3 Knowledge-Based Specifications

In [34], van der Meyden and Vardi argue that temporal logics are not enough to specify certain program properties. In particular they identify properties reflecting that a program may not have complete information about its environment. They give the following example, "send an acknowledgement as soon as you *know* that the message has been received". Consequently they study specifications given in a linear temporal logic with epistemic operators.

Automata-based techniques are used to show that the synthesis problem in this setting the single agent case is solvable. In the multi-agent case the problem is decidable in broadcast environments or when the specification is restricted to positive knowledge. Settings including asynchronous observation, branching time or conservative approximations have not been studied [35]. Additionally, there is currently no game interpretation of the knowledge-based synthesis problem.

References

1. Church, A.: Logic, arithmetic and automata. In: Proceedings of the International Congress of Mathematicians. (1962) 23–35
2. O. Kupferman, M.Y. Vardi: Church's problem revisited. In: The Bulletin of Symbolic Logic, 5(2). (1999) 245–263
3. J.R. Buchi, L.H. Landweber: Solving sequential conditions by finite-state strategies. In: Transactions of the American Mathematics Society. Volume 138. (1969) 295–311
4. R. Hossley, C. Rackoff: The emptiness problem for automata on infinite trees. In: Proceedings of the 13th IEEE Symposium on Switching and Automata Theory. (1972) 121–124
5. Rabin, M.: Automata on infinite objects and church's problem. In: Regional Conference Series in Mathematics. Volume 13. American Mathematics Society (1972)
6. E.M. Clark, E.A. Emerson: Design and synthesis of synchronisation skeletons using branching time temporal logic. In: Proceedings IBM Workshop on Logics of Programs. Volume 131., Lecture Notes in Computer Science, Springer (1981) 52–71
7. Z. Manna, P. Wolper: Synthesis of communicating processes from temporal logic specifications. In: ACM Transactions on Programming Language Systems. Volume 6. (1984) 68–93

8. A. Pnueli, R. Rosner: On the synthesis of a reactive module. In: Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages, Austin (1989)
9. D. Harel, A. Pnueli: On the development of reactive systems. In: Logics and Models of Concurrent Systems. Springer (1985) 477–498
10. Aidan Harding, Mark Ryan, Pierre-Yves Schobbens: A new algorithm for strategy synthesis in ltl games. In: Proceedings of TACAS'05, Lecture Notes in Computer Science, Springer (2005)
11. Alur, R., Torre, S.L.: Deterministic generators and games for ltl fragments. ACM Trans. Comput. Logic **5** (2004) 1–25
12. A. Pnueli, R. Rosner: Distributed reactive systems are hard to synthesize. In: Proceedings of the 31st IEEE Symposium on Foundations of Computer Science. (1990) 746–757
13. G.L. Peterson, J.H. Reif: Multiple person alternation. In: Proceedings of the 20th IEEE Symposium on Foundations of Computer Science. (1979) 348–363
14. O. Kupferman, M.Y. Vardi: Synthesis with incomplete informatio. In: 2nd International Conference on Temporal Logic. (1997) 91–106
15. O. Kupferman, M.Y. Vardi: Synthesizing distributed systems. In: Proceedings of the 16th IEEE Syposium on Logic in Computer Science. (2001)
16. P.J.G. Ramadge, W.M. Wonham: The control of discrete event systems. In: Proceedings of the IEEE. Volume 77. (1989)
17. O. Kupferman, P. Madhusudan, P.S. Thiagajaran, M.Y. Vardi: Open systems in reactive environments: Control and synthesis. In: Proceedings of CONCUR'00, 11th International Conference. Volume 1877., Lecture Notes in Computer Science (2000)
18. P. Madhusudan, P.S. Thiagajaran: Distributed control and synthesis for local specifications. In: Proceedings of ICALP'01. Volume 2076., Lecture Notes in Computer Science (2001)
19. A. Pnueli, R. Rosner: On the synthesis of an asynchronous reactive module. In: Proceedings of 16th ICALP. Volume 372., Lecture Notes in Computer Science (1989) 652–671
20. P. Madhusudan, P.S. Thiagajaran: A decidable class of asynchronous distributed controllers. In: Proceedings of CONCUR'02. Volume 2421., Lecture Notes in Computer Science, Springer (2002) 145–160
21. A. Arnold, A. Vincent, I. Walukiewicz: Games for synthesis of controllers with partial observation. In: Theoretical Computer Science. Volume 303(1). (2003) 7–34
22. S. Mohalik, I. Walukiewicz: Distributed games. In: FSTTCS'03. Lecture Notes in Computer Science, Springer (2003) 145–160
23. Martin, D.A.: Borel determinacy. Ann. of Math. (2) **102** (1975) 363–371
24. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: Proceedings of the 32nd annual symposium on Foundations of computer science, Los Alamitos, CA, USA, IEEE Computer Society Press (1991) 368–377
25. Mostowski, A.W.: Games with forbidden positions. Technical report, University of Gdansk (1991)
26. Gastin, P., Lerman, B., Zeitoun, M.: Distributed games and distributed control for asynchronous systems. In Farach-Colton, M., ed.: Proceedings of the 6th Latin American theoretical Informatics Symposium (LATIN'04). Volume 2976 of Lect. Notes in Comp. Science., Springer (2004)

27. Gastin, P., Lerman, B., Zeitoun, M.: Causal memory distributed games are decidable for series-parallel systems. In: Proceedings of FSTTCS'04. LNCS, Springer (2004) 257–286
28. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: Proceedings of the seventeenth international colloquium on Automata, languages and programming, New York, NY, USA, Springer-Verlag New York, Inc. (1990) 322–335
29. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
30. Bouyer, P.: Synthesis of timed systems. Talk given at the GAMES Spring School on Infinite Games and Their Applications, Bonn, 15–19 March 2005, http://www.games.rwth-aachen.de/Events/ (2005)
31. Walukiewicz, I.: A landscape with games in the background. In: Proceedings of IEEE LICS. (2004) 356–366
32. L. de Alfaro, R. Majumdar: Quantitative solution of omega-regular games. In: STOC'01: 33rd Anual ACM Symposium on Theory of Computer Science. (2001) 675–683
33. de Alfaro, L.: Quantitative verification and control via the mu-calculus. In: CONCUR'03, volume 2761 of Lecture Notes in Computer Science. (2003) 142–156
34. Sangiorgi, D., de Simone, R., eds.: Synthesis from Knowledge-Based Specifications (Extended Abstract). In Sangiorgi, D., de Simone, R., eds.: CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings. Volume 1466 of Lecture Notes in Computer Science., Springer (1998)
35. Wilke, T.: Synthesis from knowledge-based specifications. Talk given at the GAMES Spring School on Infinite Games and Their Applications, Bonn, 15–19 March 2005, http://www.games.rwth-aachen.de/Events/ (2005)