SYNTHESIS OF COMMUNICATING PROCESSES FROM TEMPORAL LOGIC SPECIFICATIONS

Zohar Manna
Computer Science Department
Stanford University
Stanford, CA
and
Applied Mathematics Department
The Weizmann Institute
Rehovot, Israel

Pierre Wolper
Computer Science Department
Stanford University
Stanford, CA

Abstract: In this paper, we apply Propositional Temporal Logic (PTL) to the specification and synthesis of the synchronization part of communicating processes. To specify a process, we give a PTL formula that describes its sequence of communications. The synthesis is done by constructing a model of the given specifications using a tableau-like satisfiability algorithm for PTL. This model can then be interpreted as a program.

1. Introduction

Most concurrent programs can easily be separated into two parts: a *synchronization* part that enforces the necessary constraints on the relative timing of the execution of the different processes and a *functional* part that actually manipulates the data and performs the computation required of the program. For example, the part of a concurrent program that ensures mutual exclusion between sections of code is in the "synchronization part" of that program whereas the code that is made mutually exclusive is in the "functional part".

The synchronization part of a concurrent program is rarely deep, but it is nevertheless frequently complicated. That is, writing it requires a lot of attention to intricate details but does not require insight into a variety of underlying mathematical theories. These characteristics make the development of tools for specifying and automatically synthesizing synchronization code a highly desirable and yet manageable task.

In this paper, we propose to use Propositional Temporal Logic (PTL) as a specification language for the synchronization part of CSP-like programs and we present a corresponding synthesis algorithm based on the decision procedure for PTL.

This research was supported in part by the National Science Foundation under grant MCS80-06930, by the Office of Naval Research under Contract N00014-76-C-0687, by the United States Air Force Office of Scientific Research under Grant AFSOR-81-0014 and by an IBM Predoctoral Fellowship.

CSP, the language of Communicating Sequential Processes, was developed by Hoare [Ho78] as a tool for describing distributed processes. It views distributed processes as interacting exclusively through well defined inter-process input/output (I/O) operations. This makes it quite easy to separate the "synchronization part" of a CSP program from its "functional part". Indeed, the "synchronization part" can be viewed as the program abstracted to its I/O operations. To describe the synchronization part of a CSP program it is then usually sufficient to give the temporal relations that have to exist between the execution of specific I/O operations.

Propositional Temporal Logic ([Pr67], [RU71]) is especially well suited for this task. Indeed, it is an extension of classical propositional logic geared towards the description of *sequences*. Moreover, PTL is decidable and has the finite model property. That is, given a PTL formula it is decidable if that formula is satisfiable, and if it is satisfiable, it has a finite model. This will be the basis of our synthesis method. Indeed, given specifications in PTL, we will use a tableau-like method ([Sm68], [BMP81]) to test for satisfiability and construct a model of the specifying formula. We then extract from that model the synchronization part of a CSP-like program.

2. The CSP Framework

The framework in which we specify and synthesize synchronization problems is that of Hoare's language of Communicating Sequential Processes (CSP) [Ho78]. A program in that language is a collection of (possibly nondeterministic) sequential processes each of which can include inter-process I/O operations. These I/O operations are the only interaction between the processes. Syntactically, an inter-process I/O operation names the source (input) or destination (output) process and gives the information to be transmitted. In Hoare's notation, the operation "output s to process P" is written

$$P!s$$

and the operation "input s from process P" is

$$P?s$$

Semantically, when a process reaches an input (output) operation, it waits for the corresponding process to reach the matching output (input) operation. At that point, the operation is performed and both processes resume their execution. There is no queuing or buffering of messages.

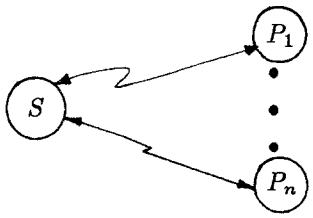
We will use CSP with the following modifications:

a) We consider systems of non-terminating processes. Terminating processes can be accomodated if they are considered to end with a dummy I/O operation that is repeated forever.

b) As we are interested in pure synchronization problems, we will assume that the only information exchanged between processes is a finite set of signals s_i.

c) We assume that when several I/O operations are possible, the one to be executed is chosen fairly. More specifically, we assume that if an I/O operation is infinitely often enabled (both sender and receiver are ready to perform it) it will eventually be executed.

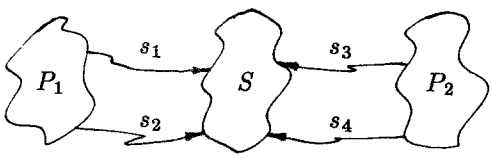
We will specify systems of processes where one process, the synchronizer S, communicates with a set of other processes P_i, $1 \leq i \leq n$.



Thus, the only communications taking place are between the synchronizer S and each of the processes P_i.

To specify the synchronization part of such a system, we will look at the infinite *sequence* of I/O operations executed by each of the processes (S and P_i's) that we assume to be non-terminating.

Example: Consider the following system:



where S receives signals s_1 and s_2 from P_1 and signals s_3 and s_4 from P_2. The sequence of I/O operations executed by S will be some interleaving of the four operations $P_1?s_1$, $P_1?s_2$, $P_2?s_3$, $P_2?s_4$. For instance it could be

$$P_1?s_1 \quad P_2?s_4 \quad P_2?s_3 \quad P_1?s_1 \quad . \quad . \quad .$$

Similarly, the sequence of I/O operations executed by P_1 will be some interleaving of $S!s_1$, $S!s_2$. ∎

The specifications will, for each process independently, characterize those sequences of I/O operations that are acceptable. The synthesis algorithm will then generate a program that when executed generates a sequence of I/O operations satisfying the specifications.

3. The Specification Language

As a specification language, we use Propositional Temporal Logic (PTL). Temporal Logic was initially developed as a branch of philosophical logic dealing with the nature of time and of temporal concepts ([Pr67], [RU71]). Recently it has been adapted to the task of reasoning about the execution sequences of programs and was found especially useful in proving properties of concurrent programs ([Pn77], [MP81]). Here, we use Temporal Logic in a similar framework; the specific formal PTL system we use is a variant of the one appearing in [GPSS80].

Intuitively, PTL is a logic oriented towards reasoning about sequences. It is a classical propositional logic extended with four *temporal* operators: $\bigcirc$, $\Diamond$, $\Box$ and U; the first three are unary, the last binary. For a sequence and a given state in that sequence,

$\bigcirc f$ is true iff f is true in the next state in the sequence;

$\Box f$ is true iff f is true in all future states of that sequence;

$\Diamond f$ is true iff f is true in some future state (i.e., it is eventually true); and

$f_1 \, U \, f_2$ is true iff f_1 is true for all states until the first state where f_2 is true.

More formally, PTL has the following syntax and semantics:

Syntax:

PTL *formulas* are built from

- A set $\mathcal{P}$ of atomic propositions: $p_1, \; p_2, \; p_3, \ldots$
- Boolean connectives: $\wedge$, $\neg$.
- Temporal operators: $\bigcirc$ ("next"), $\Box$ ("always"), $\Diamond$ ("eventually"), U ("until").

The formation rules are:

- An atomic proposition $p \in \mathcal{P}$ is a formula.
- If f_1 and f_2 are formulas, so are
$$f_1 \wedge f_2, \; \neg f_1, \; \bigcirc f_1, \; \Box f_1, \; \Diamond f_1, \; f_1 \, U \, f_2.$$

We will also use $\vee$ and $\supset$ as the usual abreviations.

Semantics:

A *structure* for a PTL formula (with set $\mathcal{P}$ of atomic propositions) is a triple $\mathcal{A} = (S, N, \pi)$ where

- S is an enumerable set of states.

- N: $(S \to S)$ is an accessibility function that for each state gives a unique next state.

- π: $(S \to 2^P)$ assigns truth values to the atomic propositions of the language in each state.

For a structure $\mathcal{A}$ and a state $s \in S$ we have

$$\langle \mathcal{A}, s \rangle \models p \quad \text{iff} \quad p \in \pi(s)$$

$$\langle \mathcal{A}, s \rangle \models f_1 \wedge f_2 \quad \text{iff} \quad \langle \mathcal{A}, s \rangle \models f_1 \text{ and } \langle \mathcal{A}, s \rangle \models f_2$$

$$\langle \mathcal{A}, s \rangle \models \neg f \quad \text{iff} \quad \text{not } \langle \mathcal{A}, s \rangle \models f$$

$$\langle \mathcal{A}, s \rangle \models \bigcirc f \quad \text{iff} \quad \langle \mathcal{A}, N(s) \rangle \models f$$

In the following definitions, we denote by $N^i(s)$ the i^{th} state in the sequence

$$s,\ N(s),\ N(N(s)),\ N(N(N(s))),\ \ldots$$

of successsors of a state s.

$$\langle \mathcal{A}, s \rangle \models \Box f \quad \text{iff} \quad (\forall i \geq 0)(\langle \mathcal{A}, N^i(s) \rangle \models f)$$

$$\langle \mathcal{A}, s \rangle \models \Diamond f \quad \text{iff} \quad (\exists i \geq 0)(\langle \mathcal{A}, N^i(s) \rangle \models f)$$

$$\begin{aligned} \langle \mathcal{A}, s \rangle \models f_1 \, U \, f_2 \quad \text{iff} \quad & (\forall i \geq 0)(\langle \mathcal{A}, N^i(s) \rangle \models f_1) \text{ or} \\ & (\exists i \geq 0)\big(\langle \mathcal{A}, N^i(s) \rangle \models f_2 \ \wedge \\ & \qquad \forall j (0 \leq j < i \ \supset \ \langle \mathcal{A}, N^j(s) \rangle \models f_1)\big) \end{aligned}$$

An *interpretation* $I = \langle \mathcal{A}, s_0 \rangle$ for PTL consists of a structure $\mathcal{A}$ and an initial state $s_0 \in S$. We will say that an interpretation $I = \langle \mathcal{A}, s_0 \rangle$ *satisfies* a formula f iff $\langle \mathcal{A}, s_0 \rangle \models f$. Since an interpretation I uniquely determines a sequence

$$\sigma = s_0,\ N(s_0),\ N^2(s_0),\ N^3(s_0),\ \ldots$$

we will often say "the sequence σ satisfies a formula" instead of "the interpretation I satisfies a formula".

Note: The temporal operators we have defined differ from those in [GPSS80] in the following way:

- They are reflexive. That is, a state is included in its own sequence of successors.

- The Until operator does not have an "eventuality component". That is, according to our definitions, $f_1 \, U \, f_2$ does not imply $\Diamond f_2$.

Our purpose in using PTL is to describe processes by specifying their allowable sequences of I/O operations. To do this, we consider PTL formulas where the atomic

propositions stand for I/O operations. And, to reflect the fact that we are looking at sequences where only one I/O operation occurs at a time, we systematically add to our specifications for each process the following *single event condition*:

$$\Box\Big(\big(\bigvee_{1\leq i\leq n} p_i\big) \wedge \big(\bigwedge_{1\leq i<j\leq n} \neg(p_i \wedge p_j)\big)\Big) \tag{3.1}$$

where $p_1, \ldots, p_n$ are all the atomic propositions (I/O operations) appearing in the specifications of that process. In other words, a state of our temporal logic corresponds to the execution of exactly one I/O operation (the atomic proposition true in that state) and the "next" state corresponds to the execution of the next I/O operation.

Example:

For a process P that sends signals s_1 and s_2 to a process S,

$$S!s_1$$

specifies that all its sequences of I/O operations start with $S!s_1$. And,

$$\Box(S!s_1 \supset \bigcirc S!s_2)$$

specifies that $S!s_1$ is always immediately followed by $S!s_2$, with no other I/O operation being performed by P in between.

4. Examples of Specifications

Let us first recall that when we give the specifications for a synchronization problem, we independently give the specifications for each of the processes involved (the synchronizer S and synchronized processes P_i). That means that for each process we give a PTL formula that, in conjunction with the single event condition (3.1), has to be satisfied by the sequences of I/O operations executed by *that* process. Thus, for instance, $\bigcirc$ means "next" in the particular process we are specifying.

Example 1: Mutual Exclusion

Suppose we have two processes, P_1 and P_2, that communicate with a synchronizer S. The signals sent to the synchronizer by P_i $(i = 1, 2)$ are $S!begin_i$ (begin critical section) and $S!end_i$ (end critical section). The synchronizer should ensure that processes P_1 and P_2 are never simultaneously in their respective critical sections that start with $S!begin_i$ and end with $S!end_i$. What the specifications for a process P_i should say is that P_i alternately sends $begin_i$ and end_i signals, starting with a $begin_i$. This is expressed by the conjunction of the following formulas:

$$S!begin_i$$

(the first signal sent is *begin critical section*)

$$\Box(S!begin_i \supset \bigcirc S!end_i)$$

(after a *begin critical section* signal, the next signal sent is *end critical section*)

$$\Box(S!end_i \supset \bigcirc S!begin_i)$$

(after an *end critical section* signal, the next signal sent is *begin critical section*).

The specifications for the synchronizer are:

$$\Box(P_1?begin_1 \supset ((\neg P_2?begin_2)\ U(P_1?end_1)))$$

(after letting P_1 proceed into its critical section by accepting a $begin_1$ signal, do not let P_2 enter its own critical section until P_1 has finished)

$$\Box(P_2?begin_2 \supset ((\neg P_1?begin_1)\ U(P_2?end_2)))$$

(after letting P_2 proceed into its critical section by accepting a $begin_2$ signal, do not let P_1 enter its own critical section until P_2 has finished).

One would expect that it is also necessary to specify absence of starvation:

$$\Box(\Diamond P_1?begin_1 \vee \Diamond P_1?end_1)$$

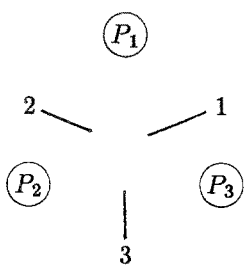
(do not neglect P_1 indefinitely)

$$\Box(\Diamond P_2?begin_2 \vee \Diamond P_1?end_2)$$

(do not neglect P_2 indefinitely). But as we will see later, in section 6, we do not have to write these conditions explicitly since they will always be systematically introduced during the synthesis. ∎

Example 2: Dining Philosophers

We specify the classical dining philosophers problem for three philosophers. Three philosophers are sitting at a round table in a Chinese restaurant alternatively thinking and eating. Between two philosophers there is only one chop stick and a philosopher needs to pick up both the chop stick at his left and the one at his right before he can eat.



The problem is to synchronize the eating of the philosophers. We have a process P_i per philosopher and a synchronizer (or "chop sticks" process) S. Each philosopher P_i communicates with the synchronizer S by four operations:

$S!pick_i$	pick up chop stick i
$S!pick_{i\oplus 1}$	pick up chop stick $i \oplus 1$
$S!put_{i\oplus 1}$	put down chop stick $i \oplus 1$
$S!put_i$	put down chop stick i

($\oplus$ designates addition modulo 3; we will also use $\ominus$ for subtraction modulo 3).

The specifications for each philosopher P_i, $i = 1, 2, 3$ are:

$$S!pick_i$$

(the first signal sent is $pick_i$)

$$\Box(S!pick_i \supset \bigcirc S!pick_{i\oplus 1})$$

$$\Box(S!pick_{i\oplus 1} \supset \bigcirc S!put_{i\oplus 1})$$

$$\Box(S!put_{i\oplus 1} \supset \bigcirc S!put_i)$$

$$\Box(S!put_i \supset \bigcirc S!pick_i)$$

Again, these specifications say that each philosopher repeatedly picks up one chop stick, picks up the second, puts the second chop stick down and puts the first chop stick down.

The specifications for the synchronizer are

$$\Box\big(P_i?pick_i \supset ((\neg P_{i\ominus 1}?pick_i)\ U(P_i?put_i))\big)$$

$$\Box\big(P_i?pick_{i\oplus 1} \supset ((\neg P_{i\oplus 1}?pick_{i\oplus 1})\ U(P_i?put_{i\oplus 1}))\big)$$

for $i = 1, 2, 3$. These essentially say that a chop stick cannot be picked up by two philosophers simultaneously.

5. Overview of the Synthesis

As described in Section 3, when we specify a system of processes, we specify each of the processes involved separately. This makes the specification task much easier. However, to deal with some properties of the system like absence of deadlock or starvation, we have to look at the combination of the specifications of all the processes involved. But, as the specifications refer to the sequence of I/O operations of each process separately, we first have to modify these specifications so that they refer to the global

sequence of I/O operations, that is the merge of the sequences of I/O opèrations of the individual processes.

Thus, the first step of our synthesis is the *relativization procedure* that takes the specifications of each process (the *local specifications*) and transforms them into specifications for the global system of processes (the *global specifications*). After the relativization, we proceed to do the synthesis with the global specifications of the system of processes.

The second step is then to apply a tableau-like satisfiability algorithm for PTL to these global specifications. The tableau decision procedure we use is essentially the one described in [BMP81] restricted to linear time and modified to use our assumption that exactly one atomic proposition is true in each state.

The decision procedure can have two possible outcomes: either it declares that the specifications are unsatisfiable and in that case it means that there is no program that can satisfy the synchronization problem as specified. Or, it produces a *model graph* from which all possible models of the specifications can be extracted.

This model graph could almost be transformed into the programs we are synthesizing except for the fact that there could be some paths in the graph that never satisfy some eventualities (properties of the form $\Diamond f$). In other words, though all models of the specifications can be generated from that graph, not all paths generated by the graph are models of the specifications. Our next step will thus be to unwind the graph to obtain an actual model of the specifications. Unfortunately, this unwinding usually gives a graph that, though it generates only models of the specifications, generates only one or a few of the possible models. In programming terms, this means that our processes will be restricted to only a few of the possible execution sequences satisfying the specifications, which clearly is undesirable.

In the special case where the eventualities are "non temporal" (*i.e.*, of the form $\Diamond f$ where f does not contain temporal operators) we are able to avoid unwinding by relying on our fairness hypothesis on the execution of CSP programs. We then synthesize our programs from a model graph that not only generates only models of the specifications (given the fairness hypothesis) but also can generate all possible models.

The final step in the synthesis will be to extract the processes from the model graph. This is rather straightforward as the model graph itself can be viewed as the synchronizer process and the other processes can be obtained as restrictions of that graph.

In summary, the steps of our synthesis will be

1) relativize the specifications (to obtain the global specifications).

2) apply the satisfiability algorithm (to obtain the model graph).

3) unwind if necessary (to satisfy eventualities).

4) generate the individual processes.

6. Relativization

Our purpose here is to take the local specifications of the processes and transform them into global specifications for the sequence of I/O operations executed by the whole system of processes. At first glance it might seem that the global specifications would simply be the conjunction of the specifications of all the processes involved. However before taking that conjunction there are three problems that have to be dealt with:

(1) At the global level, the sending and receiving of a given message is a single action. Thus, we have to make explicit the correspondence between pairs of matching I/O operations; that is, pairs of operations consisting of an output operation that sends a given message (*e.g.* $S!s$ appearing in P_i) and the corresponding operation that receives that message(*e.g.* $P_i?s$ appearing in S).

(2) The local specifications for a process describe its sequence of I/O operations. But, that sequence is only a subsequence of the global sequence of I/O operations. The local specifications have to be modified to reflect this fact. *Note:* we are reasoning under our assumption that only one I/O operation happens at a time (locally and globally).

(3) The subsequence of the global sequence corresponding to each process is infinite. This has to be made explicit in the global specifications.

These considerations lead us to the following three steps of our relativization procedure.

(1) Rename matching I/O operations to a unique new appellation. For example we would, in our preceeding example, rename $S!begin_1$ and $P_1?begin_1$ to $begin_1$. .

(2) Define inP_i to be $p_1 \vee \ldots \vee p_n$ where $p_1, \ldots, p_n$ are the I/O operations appearing in P_i. Then, to refelect the fact that the specifications for P_i concern a subsequence of the global sequence, we transform these specifications using the two following rules:

$$p \rightarrow (\neg inP_i \; U \; p) \tag{6.1}$$

where p is an atomic proposition, and

$$\bigcirc f \rightarrow (\neg inP_i \; U (inP_i \wedge \bigcirc f)) \tag{6.2}$$

That is, the right-hand side of (6.1) is substituted for all the atomic propositions in the specifications of P_i and the right-hand side of (6.2) for all occurences of $\bigcirc$.

Note: in our specific framework, all I/O operations occur between the synchronizer S and some other process P_i. Thus for the synchronizer $inS = true$ and its specifications need not be modified.

(3) For each process P_i we add the following *infinite subsequence requirement.*

$$\Box \Diamond(inP_i) \tag{6.3}$$

That is, some operation of process P_i has to occur infinitely often in the global sequence.

The global specifications are then the conjunction of the specification for the synchronizer, the specification for the processes P_i modified using (6.1) and (6.2) and the requirements (6.3).

The only non-trivial step is step (2). Let us call the local specifications for a process P_i transformed by using rules (6.1) and (6.2) the *modified specifications* for P_i. We have the following result:

Proposition 6.1: A sequence satisfies the modified specifications for P_i if and only if its subsequence consisting of all the I/O operations of P_i satisfies the original specifications for P_i.

The proposition can be easily proved by induction on the structure of the specifications for P_i.

Before we give an example, let us first note that for a formula relative to a process P_i that is of the form

$$\Box(p \supset \bigcirc q)$$

(*i.e.*, if p then q in the next state) the relativized version is

$$\Box((\neg inP_i \; U \; p) \supset (\neg inP_i \; U(inP_i \wedge \bigcirc(\neg inP_i \; U \; q))))$$

This can be simplified, using PTL equivalences to

$$\Box(p \supset \bigcirc(\neg inP_i \; U \; q))$$

(*i.e.*, if p then, from the next state on, we are not in P_i until q).

Example: Mutual exclusion problem

Let us recall that the specifications for the mutual exclusion problem are:

For the processes P_i, $i = 1, 2$:

$$S!begin_i$$

$$\Box(S!begin_i \supset \bigcirc S!end_i)$$

$$\Box(S!end_i \supset \bigcirc S!begin_i)$$

For the synchronizer S:

$$\Box(P_1?begin_1 \supset ((\neg P_2?begin_2)\, U (P_1?end_1)))$$

$$\Box(P_2?begin_2 \supset ((\neg P_1?begin_1)\, U (P_2?end_2)))$$

Then, if

$$inP_1 \equiv begin_1 \vee end_1$$

$$inP_2 \equiv begin_2 \vee end_2,$$

the global specifications for the mutual exclusion problem are:

From the specifications of P_1:

$$\neg inP_1\ U\ begin_1$$

$$\Box(begin_1 \supset \bigcirc(\neg inP_1\ U\ end_1))$$

$$\Box(end_1 \supset \bigcirc(\neg inP_1\ U\ begin_1))$$

From the specifications of P_2:

$$\neg inP_2\ U\ begin_2$$

$$\Box(begin_2 \supset \bigcirc(\neg inP_2\ U\ end_2))$$

$$\Box(end_2 \supset \bigcirc(\neg inP_2\ U\ begin_2))$$

From the specifications of S:

$$\Box(begin_1 \supset \neg begin_2\ U\ end_1)$$

$$\Box(begin_2 \supset \neg begin_1\ U\ end_2)$$

The infinite subsequence requirements:

$$\Box \Diamond inP_1$$

$$\Box \Diamond inP_2$$

Remark: The relativization procedure can be viewed as a semantic rule for the execution in parallel of communicating processes. Indeed, if we view the meaning of a communicating process as its possible sequences of I/O operations as described by a PTL formula,

then the relativization procedure gives the meaning of the concurrent execution of the processes.

7. The Satisfiability Algorithm

In this section we will describe the tableau method we use to test for satisfiability and construct a model of the global specifications. We will first briefly review the tableau method for propositional calculus, then indicate how it can be extended to handle temporal logic and finally give in detail the exact algorithm we have developed for our specific purpose.

A set of formulas $\{f_1, \ldots, f_n\}$ is satisfiable if there is an interpretation that simultaneously satisfies all the formulas in that set. The tableau method for propositional calculus is based on the following relations between satisfiablility of sets of formulas:

T1: A set of formulas $\{f_1, \ldots, f_{i_1} \wedge f_{i_2}, \ldots, f_n\}$ is satisfiable if and only if the set of formulas $\{f_1, \ldots, f_{i_1}, f_{i_2}, \ldots, f_n\}$ is satisfiable

T2: A set of formulas $\{f_1, \ldots, \neg(f_{i_1} \wedge f_{i_2}), \ldots, f_n\}$ is satisfiable if and only if the set $\{f_1, \ldots, \neg f_{i_1}, \ldots, f_n\}$ or the set $\{f_1, \ldots, \neg f_{i_2}, \ldots, f_n\}$ is satisfiable

T3: A set of formulas $\{f_1, \ldots, \neg\neg f_i, \ldots, f_n\}$ is satisfiable if and only if the set $\{f_1, \ldots, f_i, \ldots, f_n\}$ is satisfiable

To test a formula f for satisfiability, one thus starts with the singleton $\{f\}$ and uses rules T1–T3 to decompose f into sets of its subformulas. If the decomposition is carried on until the sets contain only atomic formulas (atomic propositions or their negation), satisfiability can easily be decided. Indeed, a set of atomic formulas is satisfiable if and only if it does not contain a proposition and its negation. This procedure actually corresponds to transforming the formula into disjunctive normal form. An extensive study of tableau methods for propositional and predicate calculus appears in [Sm68].

For PTL we also have to deal with the temporal operators. This can be done with the following three identities

$$\Box f \equiv f \wedge \bigcirc \Box f \tag{7.1}$$

$$\Diamond f \equiv f \vee \bigcirc \Diamond f \tag{7.2}$$

$$f_1 U f_2 \equiv f_2 \vee (f_1 \wedge \bigcirc(f_1 U f_2)) \tag{7.3}$$

These identities will enable us to decompose a formula into sets containing atomic formulas (atomic propositions and their negation) and PTL $\bigcirc$-formulas (formulas having $\bigcirc$ as their main connective). The achievement of such a decomposition is to separate the requirements expressed by the formula into a requirement on the "current state" (the

atomic formulas) and into a requirement on "the rest of the sequence" (the O-formulas). One then checks that the set of formulas concerning the "current state" is satisfiable and then repeats the whole process with the O-formulas, after having removed their outermost O operator. In other words, one tests for satisfiability by trying to build a model state by state. As all the formulas appearing in the process are subformulas of the initial formula, one will eventually reach a state that has already occurred, thus the process terminates.

There is, however, at that point one more step to do. The identity (7.2) allows us to satisfy $\Diamond f$ by always postponing it ($\bigcirc \Diamond f$). Thus, before declaring a formula satisfiable, we have to check that all the formulas of the form $\Diamond f$ can be effectively satisfied; that is, that there is a possible future state in which f is true.

Let us now describe our algorithm in more detail. The central part of the algorithm is the *decomposition procedure* that separates the requirements expressed by a set of formulas S into requirements on the "current state" and on the "rest of the sequence". In that procedure, we use our assumption that exactly one atomic proposition is true in each state. That assumption makes it much more efficient to check all possible assignments of truth values to the atomic propositions in the current state (the number of such assignments is the same as the number of atomic propositions in the language) than to brutally apply the decomposition to a set of formulas including the single event condition (3.1). Indeed, the latter could lead to examining a number of cases that is exponential in the number of atomic propositions, but that would eventualy be restricted to a linear number.

To do this, we decompose the set of formulas S separately for each atomic proposition in the language. That is, for each proposition p, we decompose the set of formulas under the assumption that p is true and the other atomic propositions false. The decomposition procedure thus takes as inputs a set of PTL formulas S and a proposition p. It outputs a set Σ_p of sets S_i of formulas f_{ij}, *i.e.* $\Sigma_p = \{S_i\}$ where each $S_i = \{f_{ij}\}$. Each formula $f_{ij} \in S_i$ either is a O-formula or is "marked", *i.e.* it is a formula that already has been used in the decomposition and is only kept for reference. Under the assumption that p is true, the original set of formulas S is satisfiable if and only if, for some i, all the unmarked formulas in S_i are satisfiable. In other words, the O-formulas in each set S_i give one of the possible requirements on the "rest of the sequence" if p is the proposition true in the current state.

The decomposition procedure initializes Σ_p with the set of sets of formulas $\{S\}$ and then repeatedly transforms it until all the elements S_i of Σ_p contain only marked formulas or O-formulas. It is the following:

(1) (Initialize): start with $\Sigma_p = \{S\}$.

(2) (Expand): repeat steps (3)–(5) until for all $S_i \in \Sigma_p$, all the formulas $f_{ij} \in S_i$ are marked formulas or O-formulas.

(3) Pick a formula $f_{ij} \in S_i \in \Sigma_p$ that is not marked and not a O-formula.

(4)(Simplify): In the formula f_{ij}, replace all the occurrences of p that are not in the scope of a temporal operator by *true* and all similar occurrences of the other atomic propositions by *false*. Perform boolean simplification. This yields a formula f'_{ij}, called "f_{ij} simplified for p".

(5) (a) if $f'_{ij} \equiv true$ replace S_i by $S_i - \{f_{ij}\}$. Given that p is true, f_{ij} is identically true and can thus be removed from S_i.

(b) if $f'_{ij} \equiv false$ replace Σ_p by $\Sigma_p - \{S_i\}$. In this case, f_{ij} is false and the set S_i is unsatisfiable. It can thus be removed.

(c) if f'_{ij} is a O-formula, replace S_i by $(S_i - \{f_{ij}\}) \cup \{f'_{ij}\}$. As we have obtained a O-formula, no more decomposition is necessary.

(d) if f'_{ij} is of type α (see table below), replace S_i by

$$(S_i - \{f_{ij}\}) \cup \{f'_{ij}*, \alpha_1, \alpha_2\}$$

where $f'_{ij}*$ is f'_{ij} marked. Since a formula of type α is satisfiable iff both α_1 and α_2 are satisfiable, we replace f_{ij} by α_1 and α_2. We also keep a record of f'_{ij} by marking it.

(e) if f'_{ij} is of type β (see table below), replace S_i by the two following sets:

$$(S_i - \{f_{ij}\}) \cup \{f'_{ij}*, \beta_1\}, \quad (S_i - \{f_{ij}\}) \cup \{f'_{ij}*, \beta_2\}$$

where $f'_{ij}*$ is f'_{ij} marked. Since a formula of type β is satisfiable iff either β_1 or β_2 are satisfiable, we replace S_i by two sets: one containing β_1 and one containing β_2.

The formulas of type α and β are given in the following two tables. Notice the correspondence between the entries in the tables concerning temporal operators and the identities (7.1)–(7.3).

α	α_1	α_2
$f_1 \wedge f_2$	f_1	f_2
$\neg\neg f_1$	f_1	f_1
$\neg \bigcirc f_1$	$\bigcirc \neg f_1$	$\bigcirc \neg f_1$
$\Box f_1$	f_1	$\bigcirc \Box f_1$
$\neg(f_1 \, U \, f_2)$	$\neg f_2$	$\neg f_1 \vee \bigcirc \neg(f_1 \, U \, f_2)$
$\neg \Diamond f_1$	$\neg f_1$	$\bigcirc \neg \Diamond f_1$

β	β_1	β_2
$\neg(f_1 \wedge f_2)$	$\neg f_1$	$\neg f_2$
$\Diamond f_1$	f_1	$\bigcirc \Diamond f_1$
$(f_1 \, U \, f_2)$	f_2	$f_1 \wedge \bigcirc(f_1 \, U \, f_2)$
$\neg \Box f_1$	$\neg f_1$	$\bigcirc \neg \Box f_1$

Example: Let us apply the decomposition procedure for q to the set of formulas

$$S = \{\Box(q \supset \neg(p \, U \, r))\}$$

Σ_q first gets initialized to

$$\Sigma_q = \{\{\Box(q \supset (\neg p \, U \, r))\}\}$$

At that point, the only formula we can choose in step (3) is $\Box(q \supset (\neg p \, U \, r))$. As all its atomic propositions occur within the scope of a temporal opeartor ($\Box$), step (4) does not modify it. Step (5d) splits $\Box(q \supset (\neg p \, U \, r))$ into $q \supset (\neg p \, U \, r)$ and $\bigcirc \Box(q \supset (\neg p \, U \, r))$, therefore, we get

$$\Sigma_q = \{\{q \supset (\neg p \, U \, r), \quad \bigcirc \Box(q \supset (\neg p \, U \, r)), \quad \Box(q \supset (\neg p \, U \, r))*\}\}.$$

Step (3) then chooses $q \supset (\neg p \, U \, r)$ which is simplified by step (4), after replacing q by *true*, to $(\neg p \, U \, r)$. This is a formula of type β, we thus split the set that contains it into two sets: one containing r and the other containing $\neg p \wedge \bigcirc(\neg p \, U \, r)$.

$$\begin{aligned}\Sigma_q = \{\{&r, \quad (\neg p \, U \, r)*, \quad \bigcirc \Box(q \supset (\neg p \, U \, r)), \quad \Box(q \supset (\neg p \, U \, r))*\}, \\ &\{\neg p \wedge \bigcirc(\neg p \, U \, r), \quad (\neg p \, U \, r)*, \quad \bigcirc \Box(q \supset (\neg p \, U \, r)), \quad \Box(q \supset (\neg p \, U \, r))*\}\}.\end{aligned}$$

Then, as r simplified for q is false, by (5b) the first set is removed and we get

$$\Sigma_q = \{\{\neg p \wedge \bigcirc(\neg p \, U \, r), \quad (\neg p \, U \, r)*, \quad \bigcirc \Box(q \supset (\neg p \, U \, r)), \quad \Box(q \supset (\neg p \, U \, r))*\}\}.$$

And, finally, as $\neg p \wedge \bigcirc(\neg p \, U \, r)$ simplified for q is $\bigcirc(\neg p \, U \, r)$ (p is replaced by $false$), we get by (5c)

$$\Sigma_q = \{\{\bigcirc(\neg p \, U \, r), \quad (\neg p \, U \, r)*, \quad \bigcirc \Box(q \supset (\neg p \, U \, r)), \quad \Box(q \supset (\neg p \, U \, r))*\}\}.$$

∎

We can now proceed to describe the *satisfiability algorithm*. This algorithm uses the decomposition procedure to build a *model graph* that is a search for all potential models

of the formula. From that graph, we will be able to decide satisfiability and to construct a model. Each node and edge in the graph is labeled with a set of formulas. The sets of formulas labeling an edge always contain exactly one of the atomic propositions of the language. The edges of the graph will correspond to the "states" of the interpretation of PTL.

The graph is constructed as follows:

(1) Start with a graph containing just one node labeled by a set S containing the formulas f_i to be tested (the *initial formulas*),*i.e.* $S = \{f_i\}$.

(2) Repeatedly apply step (3) to the nodes of the graph until it has been applied to all nodes.

(3) For every atomic proposition p in the language:

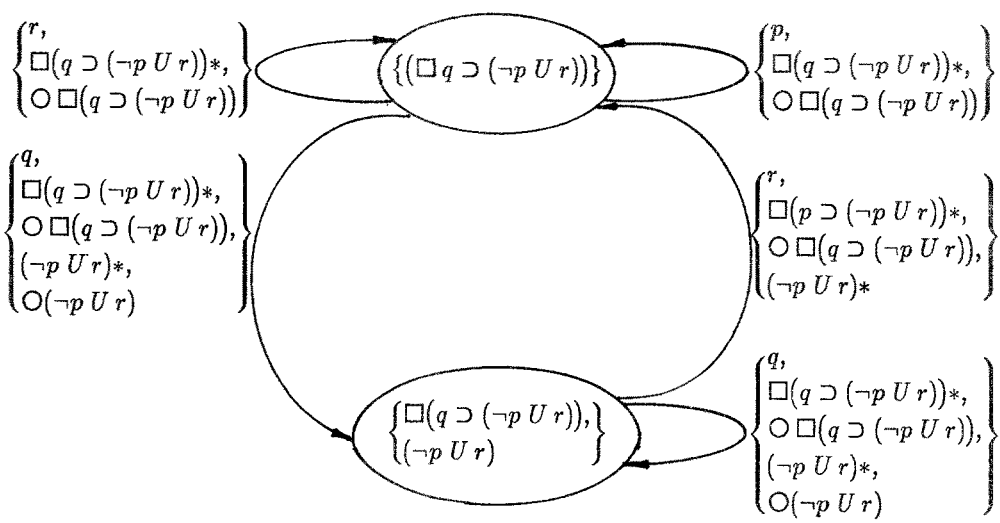
(a) Apply the decomposition procedure for p to the set S of formulas labeling the current node.

(b) For each set S_i in the set Σ_p generated by the decomposition procedure, create an edge labeled by $\{p\} \cup S_i$ leading to a node labeled by the set of all formulas f such that $\bigcirc f \in S_i$ or to a node that can be determined to be labeled by an equivalent set of formulas. If there is no such node, create one.

Example 1: For the formula

$$f_0 = \Box(q \supset (\neg p \; U \; r)),$$

the graph is:



This graph was constructed by starting with a node labeled by $\{\Box(q \supset (\neg p\ U\ r))\}$. Then, applying the decomposition procedure for q to that set of formulas we obtain, as described previously

$$\Sigma_q = \{\{\bigcirc(\neg p\ U\ r),\quad (\neg p\ U\ r)*,\quad \bigcirc\Box(q \supset (\neg p\ U\ r)),\quad \Box(q \supset (\neg p\ U\ r))*\}\}.$$

Thus we create an edge labeled by

$$\{q,\quad \bigcirc(\neg p\ U\ r),\quad (\neg p\ U\ r)*,\quad \bigcirc\Box(q \supset (\neg p\ U\ r)),\quad \Box(q \supset (\neg p\ U\ r))*\}.$$

Since this set contains two $\bigcirc$-formulas ($\bigcirc(\neg p\ U\ r)$ and $\bigcirc\Box(q \supset (\neg p\ U\ r))$), the edge leads to a node labeled by

$$\{(\neg p\ U\ r),\quad \Box(q \supset (\neg p\ U\ r))\}.$$

The other edges are constructed similarly. ∎

Example 2: Mutual exclusion problem.

Let us recall that the global specifications for the mutual exclusion problem are:

$$\neg in P_1\ U\ begin_1 \tag{7.4}$$

$$\Box(begin_1 \supset \bigcirc(\neg in P_1\ U\ end_1)) \tag{7.5}$$

$$\Box(end_1 \supset \bigcirc(\neg in P_1\ U\ begin_1)) \tag{7.6}$$

$$\neg in P_2\ U\ begin_2 \tag{7.7}$$

$$\Box(begin_2 \supset \bigcirc(\neg in P_2\ U\ end_2)) \tag{7.8}$$

$$\Box(end_2 \supset \bigcirc(\neg in P_2\ U\ begin_2)) \tag{7.9}$$

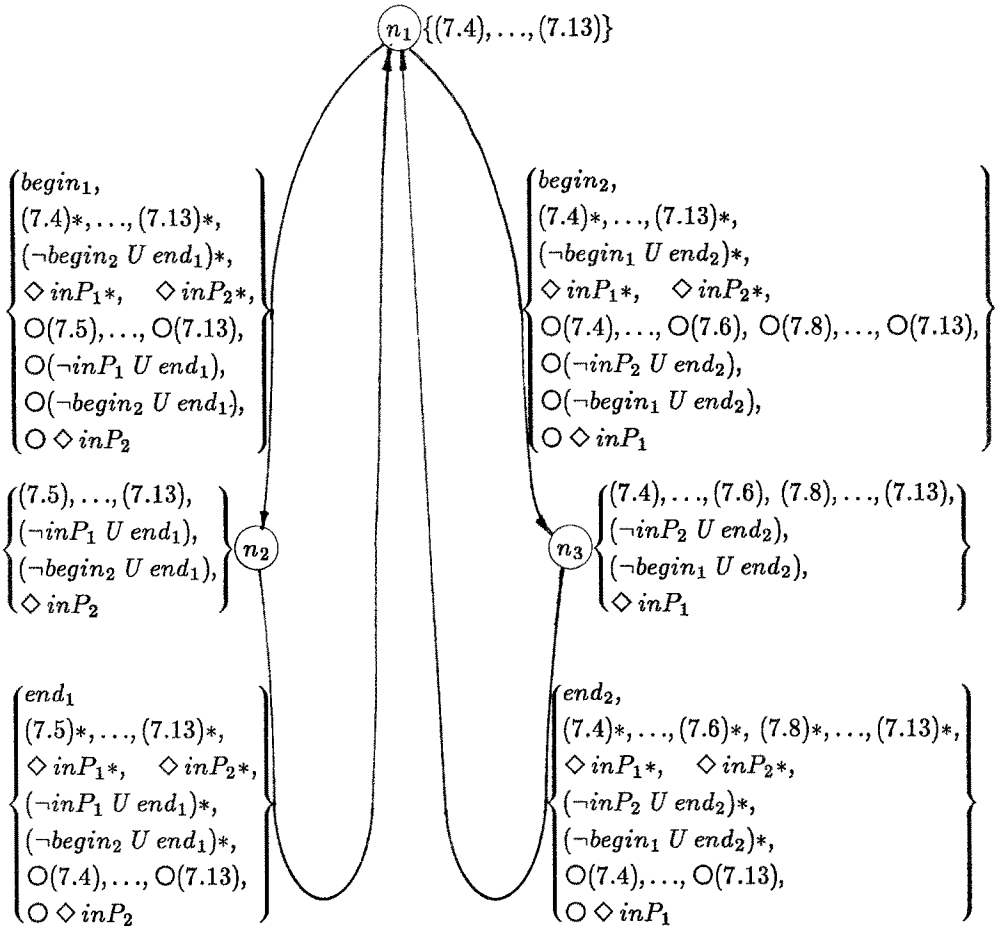
$$\Box(begin_1 \supset \neg begin_2\ U\ end_1) \tag{7.10}$$

$$\Box(begin_2 \supset \neg begin_1\ U\ end_2) \tag{7.11}$$

$$\Box \Diamond in P_1 \tag{7.12}$$

$$\Box \Diamond in P_2 \tag{7.13}$$

The graph the satisfiability algorithm yields for these specifications is then:



Note that the end_1 edge from n_2 is supposed to lead to a node labeled by

$$\{(7.4), \ldots, (7.13), \quad \Diamond\, inP_2\}.$$

But, as (7.13) is $\Box \Diamond inP_2$ and as $\Box \Diamond p \equiv \Box \Diamond p \wedge \Diamond p$, this set is equivalent to

$$\{(7.4), \ldots, (7.13)\}$$

and the edge can lead to n_1. Similarly, the end_2 edge from n_3 also leads to n_1. ∎

It is straightforward to give an upper bound on the size of the graph. The number of nodes in the graph is at most 2^{4c+2} where c is the number of temporal operators in the formula to be tested. Indeed, given the α and β rules, the formulas appearing in a node are either the initial formula, a subformula of the initial formula with a temporal operator as its main connective (there are exactly c such formulas), a subformula of the initial formula appearing in the immediate scope of a O operator (there are at most c such formulas) or the negation of any of the above. There are clearly at most $4c + 2$ such

formulas and as each node is characterized by a subset of these formulas, a bound on the number of distinct nodes is 2^{4c+2}.

The last step of satisfiability algorithm is to check that all the nodes are satisfiable and that all eventualities can effectively be realized. For this, we apply the following nodes and edges *elimination procedure*:

Repeatedly apply the following two rules until no longer possible.

(1) If a node has no edge leaving it, eliminate that node and all edges leading to it.

(2) If an edge contains an *eventuality formula*, that is a formula of the form

$$\Diamond f_1, \quad \neg\Box\neg f_1 \quad \text{or} \quad \neg(\neg f_1 \, U \, f_2)$$

then, delete that edge if there is no path from that edge leading to an edge containing $\{p, f_1'\}$ for some atomic proposition p in the language, where f_1' is f_1 simplified for p.

Note: In the preceeding examples, no elimination is necessary.

We have the following result:

Proposition 7.1: The initial formula, in conjunction with the single event condition (3.1), is satisfiable if and only if the result of the elimination process is not the empty graph.

We will not give here a proof of this result as such a proof would follow very closely the one presented in [BMP81] for a branching time PTL and in [Wo81] for an extension to PTL.

8. Eventualities and Unwinding

If the specifications are satisfiable, the decision procedure described in the previous section has provided us with a non-empty graph. This graph describes the models of the specifications in the sense that every sequence that is a model is a path in the graph and that every finite path obtained from the graph is the prefix of some model. This latter property simply follows from the fact that the decision procedure ensures that the sets of formulas associated with each edge or node of the graph are indeed satisfiable. Unfortunately, it is not always the case that all *infinite paths* obtainable from the graph satisfy the specifications. Indeed, some of these paths could leave some eventuality formula unsatisfied. However, it is always possible to modify the graph so that every infinite path satisfies the specifications.

The construction basically proceeds by *unwinding* the graph up to states where the eventualities are actually realized. The new graph is finite and can be used to generate the

program we are trying to synthesize. This unwinding has the disadvantage that it forces the processes to execute one specific path among all those that satisfy the specifications; clearly, this can lead to undesirable inefficiencies.

Example: If the specifications are

$$\Box\Diamond a \wedge \Box\Diamond b, \tag{8.1}$$

the unwinding algorithm could, for instance, give the sequence $a, b, a, b, a, b, \ldots$ as a model. In other words it would require that in order to satisfy (8.1) we alternatively execute a and b. This is correct but could be unacceptable in a situation where a can be repeated substantially faster than b. ∎

In the next section, we will see that under some conditions, the unwinding can be avoided. In the meantime, let us examine the unwinding procedure we use.

Given a graph $G = (N, E)$ with nodes N and edges E, produced by the satisfiability algorithm, we build a new graph $G' = (N', E')$ as follows.

(1) Initially G' consists of a set $N'_0 = N$ of nodes. We will call N'_0 the *initial nodes*.

(2) For each node $n'_0 \in N'_0$ do the following:

(a) Select an edge $e \in E$ leaving the node $n \in N$ corresponding to n'_0.

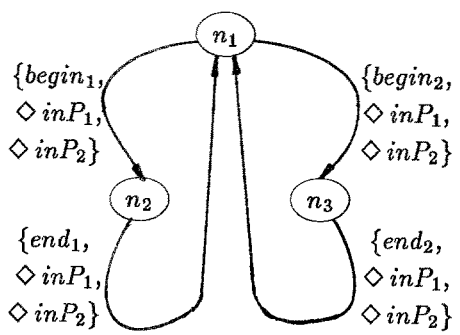
(b) Build a path starting with $e'_0 = e$ such that all eventualities in e'_0 are satisfied on that path. Given the fact that in the decision procedure we have eliminated all edges containing eventualities that could not be satisfied, we are guaranteed that such a path always exists.

(c) Let e'_f be the last edge in the path built in (b). If the corresponding edge $e_f \in E$ leads to a node $n \in N$ then connect e'_f to the corresponding $n'_0 \in N'_0$.

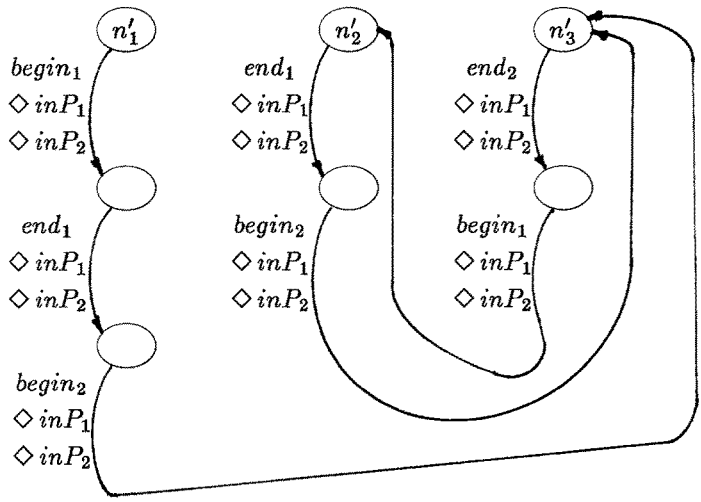
The result of the construction is a structure that satisfies the specifications.

Example:

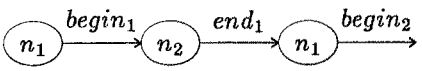
For the mutual exclusion problem we specified earlier, the graph G we obtained from the decision procedure is of the form:



For the sake of simplicity we have only annoted the edges with atomic propositions and eventuality properties. If we apply the unwinding algorithm to this graph, we get the following graph G' where $N'_0 = \{n'_1, n'_2, n'_3\}$:



To build the path starting from n'_1, we select the $begin_1$ edge leaving n_1 in G. This edge contains two eventualities: $\Diamond inP_1$ and $\Diamond inP_2$. A path that satisfies both these eventualities is



as $begin_1$ satisfies $\Diamond inP_1$ and $begin_2$ satisfies $\Diamond inP_2$. We thus incorparate this path into G' and connect its last edge to n'_3. ∎

9. Dynamic Satisfiability

As we pointed out in the last section, unwinding can lead to very inefficient programs. What we would really like is to be able to avoid the unwinding and decide dynamically, during the execution, which path through the graph we are going to take, but still do this in a way that satisfies the eventualities.

This is possible when the following three conditions are satisfied.

(1) the CSP program generated is executed fairly; that is, if a communication is infinitely often possible it is eventually executed.

(2) all eventualities are non-temporal, *i.e.* in all eventuality formulas

$$\Diamond f_1, \quad \neg\Box\neg f_1 \quad \text{or} \quad \neg(\neg f_1 \; U \; f_2)$$

labeling edges, f_1 does not contain any temporal operators.

(3) The graph satisfies the following dynamic satisfiability criterion.

Dynamic Satisfiability Criterion:

Let us denote by Π_i the set of atomic propositions corresponding to the I/O operations performed between the scheduler S and a process P_i. A model graph is said to satisfy the *dynamic satisfiability criterion* if for each edge containing an eventuality formula of the form

$$\Diamond f_1, \quad \neg\Box\neg f_1 \quad \text{or} \quad \neg(\neg f_1 \, U \, f_2)$$

(where f_1 is non-temporal) all maximum acyclic paths starting from that edge either

(1) contain an edge labeled by a proposition p that satisfies f_1

or

(2) contain a node that has an outgoing edge labeled by a proposition $p \in \Pi_i$ satisfying f_1, provided that either

(a) the edge leaving that node and included in the path is labeled by an atomic proposition $q \in \Pi_i$, *i.e.* an atomic proposition representing an I/O operation performed by the same process P_i as the one performing p

or

(b) No atomic proposition q labeling an edge of that path or any other maximum acyclic path on which f_1 has to be satisfied and conditions (1) or (2a) do not hold is in Π_i.

Essentially, the criterion checks that on *all* infinite paths, either the eventuality is realized or it is infinitely often "possible" and thus will be realized due to the fairness assumption. That means that any "fair" path in the graph is a model of the specifications and, as we will see, will be a potential execution sequence of the synthesized programs. The precise justification of the criterion involves the way we obtain the individual processes and the assumptions we make about their execution. We will discuss these issues in the next section and thus postpone our proof of the criterion until then.

Note: In the mutual exclusion example the three conditions are satisfied. We therefore do not need to unwind that graph. ∎

10. Generating the processes

The processe we generate will look very much like the model graphs we have been dealing with in the preceeding sections. If one takes such a graph and eliminates all the labeling except for the I/O operations labeling edges, the result can be interpreted as a CSP-like program. Indeed, executing such a program is traversing the graph while

performing the I/O operations on the edges. A node with several outgoing edges is viewed as a guarded command that has as guards the I/O operations appearing on those edges. Thus, according to the definition of CSP, when such a node is reached, one of the operations that is enabled (*i.e.*, such that the matching process is also ready to execute it) is chosen and the corresponding edge is followed.

The easiest process to obtain is the one for the synchronizer S. As we explained in section 2, all I/O operations are between the synchronizer and some other process P_i. This implies that the model graph we have obtained from the global specifications can be taken as the program for the synchronizer. The only (trivial) transformation that needs to be done is to rename the I/O operations back to their local name (*e.g.*, $begin_1$ becomes $P_1?begin_1$).

Each of the other processes will be obtained by restricting the model graph to the I/O operations of that process.

For a model graph $G = (N, E)$ and a process P_i, we thus build a *restricted graph* $G_i = (N_i, E_i)$. Each node of G_i ($n_i \in N_i$) corresponds to sets of nodes of the graph G. For a node n_i, we denote its corresponding set of nodes of G as $\mathcal{N}_{n_i} \subset N$. If the I/O operations of P_i are $\Pi_i = \{p_1, \ldots, p_n\}$, the construction proceeds as follows:

(1) Initially, G_i contains one node; this node corresponds to an initial node of G and all nodes accessible from that node in G through a path containing no edge labeled by a proposition $p \in \Pi_i$.

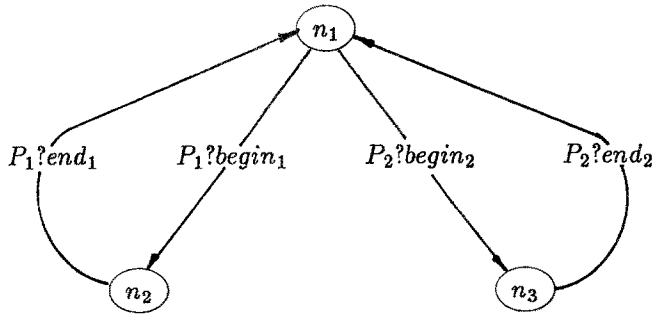
(2) Repeat step (3) until it has been applied to all nodes in G_i.

(3) Select an unprocessed node $n_i \in N_i$. For all propositions $p \in \Pi_i$ create an edge from n_i to a node $n'_i \in N_i$ such that the set $\mathcal{N}_{n'_i}$ is the set of all nodes accessible in G from any node in $\mathcal{N}_{n_i}$ through a path containing exactly one occurrence of p and no occurrence of any other member of Π_i (we call such a path a p-path). A new node n'_i is created only when G_i does not already contain a node characterized by the set $\mathcal{N}_{n'_i}$. If $\mathcal{N}_{n'_i} = \phi$ no edge is added.

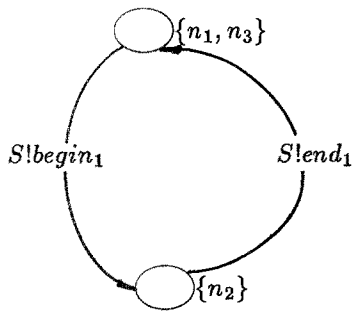
We then just have to rename the I/O operations back to their local name to obtain the process P_i.

Example:

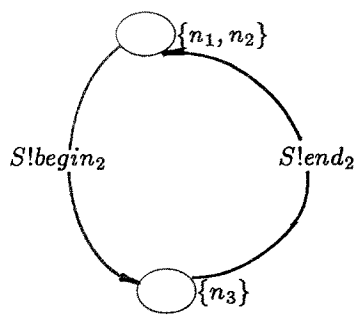
For the mutual exclusion problem specified in section 4, the program for S is:



for the processes P_1 we have



and for the process P_2



To obtain the graph for P_1, we start with the set of nodes in the model graph accessible from n_1 by a path not labeled by any operation of process P_1. This set is $\{n_1, n_3\}$. The only node accessible from either n_1 or n_3 through a $begin_1$-path is n_2. Thus we have a path labeled by $begin_1$ leading to a node labeled by $\{n_2\}$. There are no nodes accessible from either n_1 or n_3 through an end_1-path, thus no edge labeled by end_1 will leave the node $\{n_1, n_3\}$ of the graph for process P_1. The edges leaving $\{n_2\}$ are constructed similarly. ∎

We view the execution of such a system of processes as it is defined in CSP. That is, the processes have to execute matching I/O operations simultaneously. Note that even though our processes consist solely of I/O operations, we do not assume anything about the relative speed of their execution. This means that after a process executes an

I/O operation, there could be an arbitrary finite delay before it is ready to execute the following one. This delay could for, instance, correspond to the execution of a purely sequential piece of code.

The last step now is to derive actual CSP programs from the graphs. A simple way to do this is to assign a number to each node of the graph and use a variable N to keep track of the location in the graph. The program is then just one repetitive command where the *guards* are composed of a test on the value of N followed by an I/O operation, and where the *bodies* are just an updating of N.

Example:

For the synchronizer S in the mutual exclusion example, the CSP program is:

$$
\begin{array}{lll}
*[\ N = 1; P_1?begin_1 & \rightarrow & N := 2 \\
\ [] N = 1;\ P_2?begin_2 & \rightarrow & N := 3 \\
\ [] N = 2;\ P_1?end_1 & \rightarrow & N := 1 \\
\ [] N = 3;\ P_2?end_2 & \rightarrow & N := 1\ ]
\end{array}
$$

The program repeatedly checks at which location in the graph it is, then waits for the corresponding inputs and finally updates its location variable.

For the process P_1, the program is:

$$
\begin{array}{lll}
*[\ N = 1;\ S!begin_1 & \rightarrow & N := 2 \\
\ [] N = 2;\ S!end_1 & \rightarrow & N := 1\ ]
\end{array}
$$

and for the process P_2, the program is:

$$
\begin{array}{lll}
*[\ N = 1;\ S!begin_2 & \rightarrow & N := 2 \\
\ [] N = 2;\ S!end_2 & \rightarrow & N := 1\ ]
\end{array}
$$

In these programs a purely sequential piece of code can be inserted immediately after the updating of the location variable N. ∎

From the way the processes were obtained, it is clear that any concurrent execution of the system of processes (more precisely the sequence of I/O operations performed during the execution) will correspond to a path through the global graph. Thus in the case where we have unwound the graph, the synthesized processes satisfy the specifications. However, we still have to prove that if the global graph satisfies the dynamic satisfiability criterion, then any fair execution of the extracted program will satisfy all eventualities. Recall that in a fair execution every I/O operation that is infinitely often possible (both sender and receiver are ready to perform it) will eventually be executed.

Proposition 9.1: If the model graph satisfies the dynamic satisfiability criterion, then every fair execution of the extracted programs satisfies the specifications.

Proof: In view of the preceeding remarks, it is sufficient to show that all eventualities are satisfied. Let us assume that there is some eventuality formula $(\Diamond f)$ that is not satisfied for some fair computation. We will show that some operation that realizes the eventuality (satisfies f) is infinitely often possible during that computation. Hence, due to our fairness assumption that operation will be executed, and we have a contradiction. Actually, all we need to show is that for such a computation, some operation satisfying the eventuality will be possible in a finite number of steps. Indeed, the same argument can then inductively be applied to the computation starting after the point where the operation was possible. And, as we only have a finite number of possible I/O operations, one of those satisfying f will be infinitely often possible.

Let us consider the path through the global graph corresponding to our computation. Clearly, no operation p satisfying f appears on that path. Thus either condition (2a) or (2b) of the dynamic satisfiability criterion is satisfied on every maximal acyclic part of the path.

(1) If condition (2a) is satisfied somewhere on the path we have a node on the path that has an outgoing edge labeled by an operation p satisfying f. Thus, at that point the synchronizer S is ready to perform p. As the operation on the path is in the same process P_i as p, that process must also be ready to perform p. Thus p is possible.

(2) If condition (2a) is never satisfied, then (2b) has to be satisfied on every maximum acyclic part of the path. Thus some operation p will repeatedly appear as an alternative branch on the path. As no operation in the process P_i containing p appears on the path, when P_i becomes ready to execute p it will remain in that state. Then, when the synchronizer reaches the next node where p is an alternative, p will be possible. ∎

12. Conclusions and Comparison with Other Work

We have shown how the "synchronization part" of processes could be specified and synthesized. The main techniques we have used are:

(1) abstracting concurrent computations to sequences of "events" (in our case I/O operations)

(2) describing these sequences using Propositional Temporal Logic

(3) using the tableau decision procedure for PTL to synthesize the processes.

Clearly there are some limitations to our approach. The most fundamental one is that the synthesized processes are intrinsically finite state. However, this does not

exclude practical use of the method since many synchronization problems have finite state solutions. Getting rid of this limitation would most likely eliminate the decidability property of our specification language. We would then no longer be able to guarantee a correct solution to the problem whenever the specifications are satisfiable.

The PTL we have used in this paper, though it has been called *expressively complete* since it is as expressive as the first order theory of linear order [GPSS80] cannot describe all finite-state behaviors. However, an extension to PTL that would allow the description of all such behaviors has been recently developed [Wo81]. Incorporating it in our specification language would let us describe a wider class of synchronization problems. We also plan to apply the techniques we developed here to the synthesis of network protocols and sequential digital circuits.

Among related work, we should first mention that Clarke and Emerson [CE81] have been independently investigating the use of similar model building techniques for synchronization code synthesis. Their approach is, however, based on a branching time temporal logic and is oriented towards the synthesis of shared memory programs.

Earlier work on the synthesis of synchronization code includes that of Griffiths [Gr75] and Habermann [Ha75]. Griffiths' specification language is rather low-level in the sense that it is procedural in nature. In Habermann's "path expressions", the specification language is regular expressions. This has the disadvantage of requiring a global description instead of a collection of independent requirements, as in PTL. Also, regular expressions cannot describe eventualities explicitly and in [Ha75] no attention is given to the problems of deadlock and starvation.

Among later work on the subject one finds the work of Laventhal [La78], and the one of Ramamritham and Keller [RK81]. Here, the specification language is quite expresssive. In the former approach it is based on first-order predicate calculus with an ordering relation and in the latter on Temporal Logic. However, in both cases the synthesis method is rather informal and does not rely on a precise underlying theory.

Acknowledgements: We wish to thank Yoni Malachi, Joe Weening and Frank Yellin for a careful reading of a draft of this paper.

13. References

[BMP81] M. Ben-Ari, Z. Manna, A. Pnueli, "The Logic of Nexttime", *Eighth ACM Symposium on Principles of Programming Languages*, Williamsburg, VA, January 1981, pp. 164–176.

[CE81] E. M. Clarke, E. A. Emerson, "Synthesis of Synchronization Skeletons from Branching Time Temporal Logic", *Proceedings of the Workshop on Logics of Programs*, Yorktown-Heights, NY, Springer-Verlag Lecture Notes in Computer Science, 1981

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah and J. Stavi, "The Temporal Analysis of Fairness", *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, January 1980, pp. 163–173.

[Gr75] P. Griffiths, "SYNVER: A System for the Automatic Synthesis and Verification and Synthesis of Synchronization Processes", Ph. D. Thesis, Harvard University, June 1975.

[Ha75] A. N. Habermann, "Path Expressions", Computer Science Report, Carnegie-Mellon University, 1975.

[Ho78] C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No 8 (August 1978), pp. 666–677.

[La78] M. Laventhal, "Synthesis of Synchronization Code for Data Abstractions", Ph. D. Thesis, MIT, June 1978.

[MP81] Z. Manna, A. Pnueli, "Verification of Concurrent Programs: the Temporal Framework", *The Correctness Problem in Computer Science* (R. S. Boyer and J S. Moore, eds.), International Lecture Series in Computer Science, Academic Press, London, 1981.

[Pn77] A. Pnueli, "The Temporal Logic of Programs", *Proceedings of the Eighteenth Symposium on Foundations of Computer Science*, Providence, RI, November 1977, pp. 46–57.

[Pr67] A. Prior, **Past, Present and Future**, Oxford University Press, 1967.

[RU71] N. Rescher, A. Urquart, **Temporal Logic**, Springer-Verlag, 1971

[RK81] K. Ramamritham, R. M. Keller, "Specification and Synthesis of Synchronizers", *Proceedings International Symposium on Parallel Processing*, August 1980, pp. 311–321.

[Sm68] R. M. Smullyan, **First Order Logic**, Springer-Verlag, Berlin, 1968.

[Wo81] P. Wolper, "Temporal Logic Can Be More Expressive", *Proceedings of the Twenty-Second Symposium on Foundations of Computer Science*, Nashville, TN, October 1981.