Concurrent Program Synthesis from a specification in Conditional Interval Temporal Logic

7X-3

Kazunori MATSUMOTO, Naoshi UCHIHIRA and Shinichi HONIDEN

Systems & Software Engineering Lab., TOSHIBA Corp.

1. INTRODUCTION

We are developing a software prototyping system named MENDELS. In the system, we construct a concurrent program as combinations of some software components stored in a components library. Moreover a synchronization program for such the components is automatically generated from a specification given as a formula in conditional interval temporal logic(CITL). CITL is an extension of usual interval temporal logic. It can describe a specification which depend on dynamic situations. In this paper, we discuss on CITL and outline of MENDELS.

2. MENDELS : A prototyping system

To improve productivity of software development, it is believed that a software must be widely reused. To do so, software should be written in a adequate language which has high modularity and high readability. For these reasons, we at first develop a language named MENDEL[1]. MENDEL is an object-oriented concurrent language based on Prolog. Objects are executed concurrently passing messages to other objects, but any execution in each object is sequential.

In MENDELS, programs are generated following way;
(1) objects(software components) written in MENDEL are stored in a components library.
(2) objects are retrieved and interconnected from a functional specification using a attributes-matching method.
(3) a control specification for synchronizing objects are given as a formula in CITL.
(4) a synchronization program is automatically generated from a specification given at (3), and
(5) determine whether generated program satisfies user's requirements, if requirements aren't satisfied then back to any previous stages.
Specifications(functional and control) must be easy to write, easy to read and enough expressive power, so we choose attribute names as a functional specification and a temporal logic formula as a control specification. More detailed discussions on software reusing are in [1].

3. ITL and CITL

ITL and local ITL was developed originally by Moszkowski[2]. He discussed many hardware properties using it, and proved that ITL with locality conditions is decidable.
Local ITL consists of usual symbols of propositional logic and two temporal operators @(next) and &&(chop).

3.1. DECISION PROCEDURE FOR ITL

We can easily construct a decision procedure for ITL[3] using tableau method similar to the procedure for PTL[4];
A propositional variable P is decomposed into

$$(P \wedge \text{empty}) \vee (P \wedge @\text{true})$$

A formula F && G is decomposed into

$$(F \wedge G) \vee (F \wedge @(F \,\&\&\, G))$$

Using these decomposition rules and other usual rules for logical connectives($\wedge$, $\vee$), we can transform a given formula F into

$$(F \wedge \text{empty}) \vee (F \wedge @F')$$

where empty is a formula defined by

$$\text{empty} = @\neg\text{true}$$

We can define other convenient formulas similar to Moszkowski. Intuitively a formula is decomposed into the end of interval and the continuation of interval.
A state transition graph, which is a collection of all models to a given formula, is constructed by the procedure. We show an example of state transition graph in Fig.1.

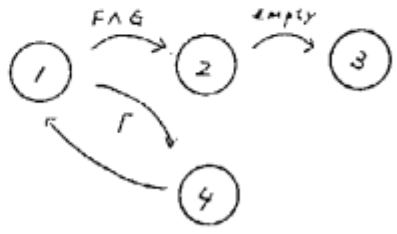

Fig.1 state transition graph for F && G

3.2. CITL

To extend ITL, we introduce a conditional formula. A conditional formula is any expression constructed from system variables, =, ≠, <, >, ∧ and ∨.

Example.

P && Q(num(Q) > 10)

is a formula in CITL and (num(Q) > 10) is called conditional formula.

In some cases, it is necessary to synchronize among programs depending on their dynamic situations. However it is difficult to give such the specification in ITL. CITL is developed to apply such the cases, i.e., we evaluate conditional formulas dynamically but other parts are the same to the ITL.

Example.
Select P or Q depending on their value can be expressed as
P(val(P) = val1) ∨ Q(val(Q) = val2)

We construct a state transition graph ignoring conditional formulas and conditional formulas are labeled on the graph(Fig.2). A transition is made only when the labeled conditional formulas are satisfied.
We can see a synchronization program as 'transition graph interpreter', and it is realized as a controller of message passing[1].

3.3. SYSTEM VARIABLES

We construct a CITL formula using system variables. Now we prepare following system variables that
(1) val(attribute name) : means the message received at or sent from an specified attribute name.
(2) num(attribute name) : means the number of messages which is received to or sent from an specified attribute name.
(3) priority(integer) : if some transitions are possible then execute the transition which has highest priority.

Example.

Consider a keycheck program which receive a list of keywords and a word. It returns YES if a word is in the list, otherwise it returns NO. In this case, checking should not be carried out before all keywords are sent. Such a restriction can be described as:

keyword((val(keyword) ≠ eof)^ priority(1))

&& word(priority(2)) && empty

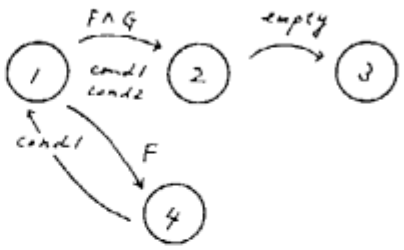

Fig.2 state transition graph
for F(cond1) && G(cond2)

4. CONCLUSIONS

We develop a new specification language CITL, which is an extension of ITL.
Our approach is similar to predicate path expression[5]. But our specification is based on temporal logic and a synchronizer is generated using a decision procedure, so verification of a given specification is simultaneously carried out in the decision procedure. In CITL, a decision procedure is applied ignoring conditional formulas but other parts are verified by the decision procedure.
Wolper's work[4] is closely related to ours. He constructs a synchronization program from a specification in ETL(extended propositional temporal logic) which is an extension of PTL. However his synthesized program is CSP and no special synchronization mechanism is proposed so there left difficulty to read his program. Furthermore ETL cannot express a specification complying with dynamic situations.

ACKNOWLEDGEMENTS

The authors thank to ICOT and Dr. Hideo Nakamura for their supports and useful advices.

REFERENCES

[1] N. Uchihira et al. Concurrent program synthesis with reusable components using temporal logic. LPC'87.
[2] B.C. Moszkowski. Reasoning about Digital Circuits. Ph.D thesis. Stanford University. 1983.
[3] M. Fujita. et al. TOKIO : Logic programming language based on temporal logic and its compilation to Prolog. ICLP. 1986.
[4] P. Wolper. Synthesis of communicating processes from temporal logic specification. Ph.D thesis. Stanford University. 1982.
[5] S. Andler. Predicate path expressions. ACM 6th POPL. 1979.