Program Synthesis

William Schultz

February 18, 2023

In general, we can present the *synthesis* problem in contrast to the verification problem as follows:

- The **verification problem**: given system M and spec φ, check that $M \models \varphi$.
- The **synthesis problem**: given spec φ, find M such that $M \models \varphi$.

0.1 Functional Synthesis

The classic, *functional synthesis* problem, is defined with respect to programs that take some input x and transform it to output y. In this setting, if we are given a specification φ that prescribes the desired input/output relation, we can construct a program by means of establishing validity of the theorem

$$\forall x \, \exists y : \varphi(x, y)$$

Note that this is equivalent to the second order formula

$$\exists f, \forall x : \varphi(x, f(x))$$

where f is a concrete function that takes input x and returns the correct output y satisfying specification φ [PR89, WL69]. If we have a constructive way to prove this theorem, then we can construct f, from which we can construct a program that satisfies φ.

0.2 Reactive Synthesis

The above approach is suitable for sequential programs, but if we want to move to concurrent programs, then we need a more expressive specification lagnauge to express φ. Temporal logic became the natural choice for this and works of [CE82, MW84] essentially do concurrent program synthesis by showing satisfiability of of a particular temporal formula specification φ, and use the model satisfying φ to construct a program implementing φ.

In [PR89], however, they claim that the approach taken in [CE82, MW84] is not quite sufficient, because it assumes that we are trying to synthesize *closed* systems. That is, systems for which we have full control over every component of the system. They claim that if, for example, we aree synthesizing a system with two components, C_1 and C_2, the "hidden assumption" in [CE82] is that we have the power to construct both C_1 and C_2 in a way that will ensure the needed cooperation. But, if we are in a so-called *open system* setting, then C_1, for example, may represent the *environment* over which the implementor has no control, while C_2 is the body of the system itself (which we may refer to as a *reactive module*). In this case, we instead have to synthesize C_2 in such a way that it will work correctly in response to any possible behaviors of the environment C_1. For example, if C_1 is a module that can only modify x (a shared variable for communication), and C_2 can only modify y, then they claim the synthesis problem should instead be stated as

$$\forall x \, \exists y : \varphi(x, y)$$

which they refer to as the *reactive synthesis* problem. Note that in the formal statement above, we should now interpret x and y as being quantified over behaviors of the computation, since we are now interpreting it over temporal logic. So, the statement is saying that for any possible sequence of values x (that can be produced by the environment C_1), there exists a sequence of values y (produced by the controllable system C_2) such that $\varphi(x, y)$ holds. They note that the approach of [CE82] can be viewed as a solution to the alternate problem statement $\exists x \, \exists y : \varphi(x, y)$.

0.3 Deductive Synthesis

The *deductive approach* [MW80] tries to synthesize an input/output program by extracting it from a realizability proof.

0.4 Temporal Synthesis

Temporal synthesis considers specifications given in the form of LTL (or CTL), for example. An initial approach was to use satisfiability of a temporal formula as a way to derive M [CE82]. See also [MW84].

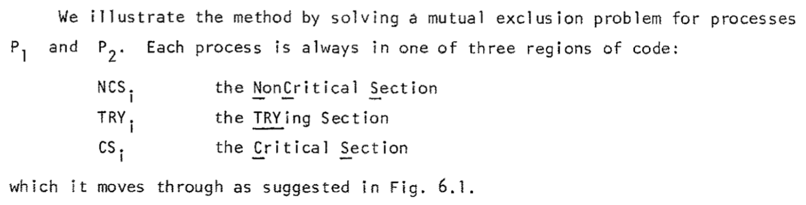
In [CE82] they consider concurrent systems consisting of a finite number of fixed processes $P_1, \ldots, P_m$ running in parallel. They treat parallelism in the usual sense i.e. nondeterministic interleaving of the sequential atomic actions of each process. They use CTL as a specification language, and consider the semantics of CTL with respect to a (Kripke) structure $M = (S, A_1, \ldots, A_k, L)$, where

- S: countable set of system states
- $A_i \subseteq S \times S$: transition relation of process i
- L: assignment of atomic propositions to each state

They use a decision procedure for satisfiability of CTL formulae (similar to one described in [BAMP81]) as part of their synthesis procedure. Given a CTL formula f_0, the decision procedure returns either "Yes, f_0 is satisfiable or "No, f_0 is unsatisfiable". If f_0 is satisfiable, then a finite model (structure) is also constructed. Their overall synthesis algorithm consists of the following high level steps:

1. Specify the desired behavior of the concurrent system using a CTL formula φ.
2. Apply the decision procedure to the formula φ to obtain a finite model of the formula.
3. Factor out the synchronization skeletons of the individual processes from the global system flowgraph defined by the model.

They demonstrate this procedure on a simple, 2 process mutual exclusion example. Below is shown the description of the abstract states of each process, $\{NCS_i, TRY_i, CS_i\}$:



We illustrate the method by solving a mutual exclusion problem for processes P_1 and P_2. Each process is always in one of three regions of code:

NCS_i	the NonCritical Section
TRY_i	the TRYing Section
CS_i	the Critical Section

which it moves through as suggested in Fig. 6.1.

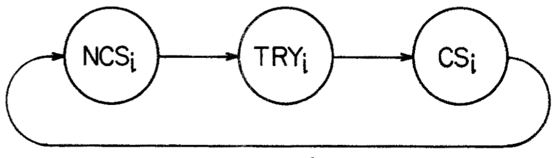


Figure 6.1

and they give the specification of the mutual exclusion problem in CTL as follows:

1. start state

   $NCS_1 \wedge NCS_2$

2. mutual exclusion

   $AG(\sim(CS_1 \wedge CS_2))$

3. absence of starvation for P_i

   $AG(TRY_i \rightarrow AF\ CS_i)$

4. each process P_i is always in exactly one of the three code regions

   $AG(NCS_i \vee TRY_i \vee CS_i)$
   $AG(NCS_i \rightarrow \sim(TRY_i \vee CS_i))$
   $AG(TRY_i \rightarrow \sim(NCS_i \vee CS_i))$
   $AG(CS_i \rightarrow \sim(NCS_i \vee TRY_i))$

5. it is always possible for P_i to enter its trying region from its non-critical region

   $AG(NCS_i \rightarrow EX_i TRY_i)$

6. it is always the case that any move P_i makes from its trying region is into the critical region

   $AG(TRY_i \wedge EX_i True \rightarrow AX_i CS_i)$

7. it is always possible for P_i to re-enter its noncritical region from its critical region

   $AG(CS_i \rightarrow EX_i NCS_i)$

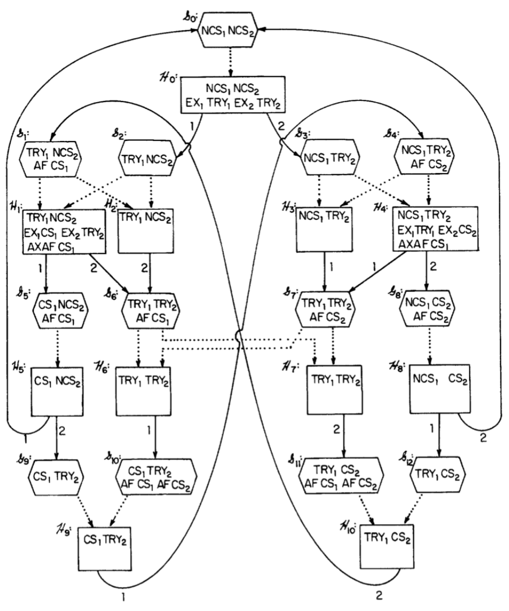
8. a transition by one process cannot cause a move by the other

   $AG(NCS_i \rightarrow AX_j NCS_i)$
   $AG(TRY_i \rightarrow AX_j TRY_i)$
   $AG(CS_i \rightarrow AX_j CS_i)$

9. some process can always move

   $AG(EX\ true)$

From this they then construct a tableau T using their decision procedure:



$\mathcal{S}_0$: $NCS_1\ NCS_2$
$\mathcal{H}_0$: $NCS_1\ NCS_2$ $EX_1\ TRY_1\ EX_2\ TRY_2$
$\mathcal{S}_1$: $TRY_1\ NCS_2$ $AF\ CS_1$
$\mathcal{S}_2$: $TRY_1\ NCS_2$
$\mathcal{S}_3$: $NCS_1\ TRY_2$
$\mathcal{S}_4$: $NCS_1\ TRY_2$ $AF\ CS_2$
$\mathcal{H}_1$: $TRY_1\ NCS_2$ $EX_1 CS_1\ EX_2\ TRY_2$ $AXAF\ CS_1$
$\mathcal{H}_2$: $TRY_1\ NCS_2$
$\mathcal{H}_3$: $NCS_1\ TRY_2$
$\mathcal{H}_4$: $NCS_1\ TRY_2$ $EX_1 TRY_1\ EX_2 CS_2$ $AXAF\ CS_1$
$\mathcal{S}_5$: $CS_1\ NCS_2$ $AF\ CS_1$
$\mathcal{S}_6$: $TRY_1\ TRY_2$ $AF\ CS_1$
$\mathcal{S}_7$: $TRY_1\ TRY_2$ $AF\ CS_2$
$\mathcal{S}_8$: $NCS_1\ CS_2$ $AF\ CS_2$
$\mathcal{H}_5$: $CS_1\ NCS_2$
$\mathcal{H}_6$: $TRY_1\ TRY_2$
$\mathcal{H}_7$: $TRY_1\ TRY_2$
$\mathcal{H}_8$: $NCS_1\ CS_2$
$\mathcal{S}_9$: $CS_1\ TRY_2$
$\mathcal{S}_{10}$: $CS_1\ TRY_2$ $AF\ CS_1\ AF\ CS_2$
$\mathcal{S}_{11}$: $TRY_1\ CS_2$ $AF\ CS_1\ AF\ CS_2$
$\mathcal{S}_{12}$: $TRY_1\ CS_2$
$\mathcal{H}_9$: $CS_1\ TRY_2$
$\mathcal{H}_{10}$: $TRY_1\ CS_2$
1
2

Figure 6.2

and then from T they extract a finite model of the global program behavior:

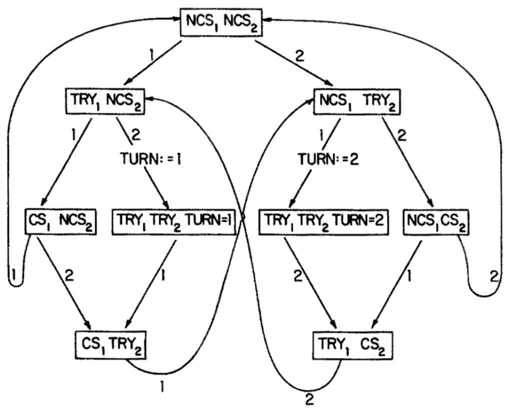


Figure 6.6

Note they manually introduced an auxiliary variable $TURN$ in order to distinguish states H_6 and H_7 in the tableau, which carries over into the extracted model.

After constructing the model representing the global program behavior, they extract "skeletons" for each individual process, which they seem to describe in a somewhat ad hoc manner i.e. they don't give a formal algorithmic procedure for this. Note that this is pointed out in [AE01], which appears to give a more formal treatment of this extraction procedure. The final, extracted skeletons for process P_1 and P_2 are shown as follows:

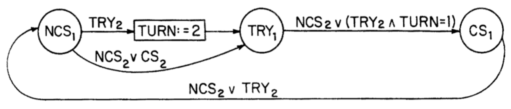


Figure 6.7 (a)

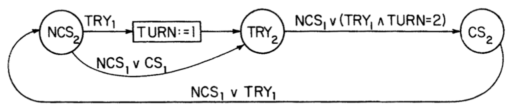


Figure 6.7 (b)

References

[AE01] Paul C. Attie and E. Allen Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Trans. Program. Lang. Syst.*, 23(2):187–242, mar 2001.

[BAMP81] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 164–176, New York, NY, USA, 1981. Association for Computing Machinery.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[MW80] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

[MW84] Zohar Manna and Pierre Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, jan 1984.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 179–190, New York, NY, USA, 1989. Association for Computing Machinery.

[WL69] Richard J. Waldinger and Richard C. T. Lee. Prow: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, page 241–252, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.