Synthesis of Asynchronous Systems

Prepared by: Christine Rizkallah
Supervisor: Sven Schewe
Date: 22, July 2008

1 Introduction

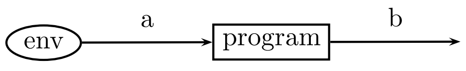
A synthesis algorithm takes a specification and generates an implementation that is guaranteed to satisfy this specification [1].

Asynchronous systems are important because they are a natural way to model many problems, for example, distributed software modules or processes running at different speeds.

Synthesis of asynchronous systems is more difficult than that of synchronous systems. This is because we are not only interested in a single program, but rather in how the whole system works correctly provided that the program we are considering is usually not scheduled all of the time. Thus, the program does not always see changes to the environment that might affect its output. To ensure that the program satisfies a certain specification we need to ensure that it satisfies the specification regardless of the environment's input.

2 Asynchronicity

Programs take inputs from the environment. Those inputs are invisible to the program until the program is scheduled. The output of the program may vary according to the input. To ensure that a certain specification or property is satisfied by the program, we need to ensure that it is satisfied regardless of the input of the environment and the scheduler. The following figure shows a program having an input variable a from the environment and an output variable b.



Consider the following sequence of states where a is the boolean input variable to the program which is chosen by the environment, b is the boolean output variable, and the boolean variable s states whether or not the program is scheduled. Note that if the program is not scheduled, then the environment is scheduled.

(¬a, ¬b, ¬s), (a, ¬b, ¬s), (a, b, s), (a, b, ¬s) ...

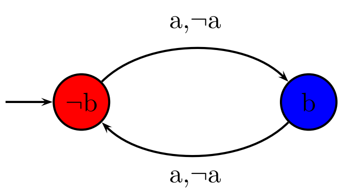
We assume that the environment cannot change the input of the program while the program is scheduled, therefore, the input of the program at step two and three is the same. The program is only scheduled at the third step (a, b, s). Thus, that is the only "visible" state to the program, the rest is "invisible".

3 Computational Model

In an asynchronous system we use two types of trees. The *behaviour tree* describes the behaviour of a program under the assumption that the program is always scheduled. The *computation tree* captures all possible system behaviours.

3.1 Example

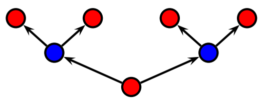
As an illustrative example, consider the following program which has only two states, input variable a and output variable b. In each step when it is scheduled it moves to the other state and gives the opposite output, regardless of its input [1].



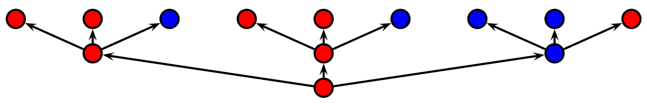
We now want to illustrate the difference between the behaviour tree and the computation tree of this program[2].

We start by the behaviour tree which describes how a program behaves when it is always scheduled. Each level of this tree is a step where the program is scheduled. The left branch represents that the program is scheduled with input $\neg a$ and the right branch represents that the program is scheduled with input a.

The behaviour tree of this program changes the output at each level, that is, everytime the program is scheduled, regardless of the input. For this program it looks as follows:



Now we show the computation tree which is more general and captures all possible system behaviours. It has three directions. The leftmost represents that the program is not scheduled and environment sends input $\neg a$, the middle one that the program is not scheduled and environment sends input a, and the rightmost represents that the program is scheduled. The following is the computation tree of the previous program:



The computation tree can be constructed from the behaviour tree. This is because the behaviour tree establishes all possible system behaviours: from the behaviour tree we know what output the program gives when scheduled, and

[1]Note that we assume from now on that initially the environment sends input a.

[2]Red and blue nodes represent nodes with output $\neg b$ and output b respectively.

we know that when the program is not scheduled its state in the behaviour tree does not change and hence, output does not change either. Thus, we know what output the computation tree has at each node.

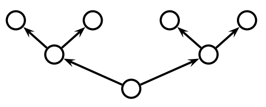
4 Translating between Trees

4.1 Synchronous Setting

In a synchronous setting the behaviour tree and the computation tree are essentially the same since the program is always scheduled. The only difference is that in the computation tree directions are added to the labels of the nodes. Thus, it is very easy to transform each of them to the other.

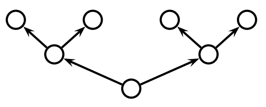
Checking whether the computation tree corresponds to a behaviour tree could be done locally by checking that the directions are correct. So this check is easy to make.

Behaviour Tree:



$\checkmark \Downarrow \Uparrow \checkmark$

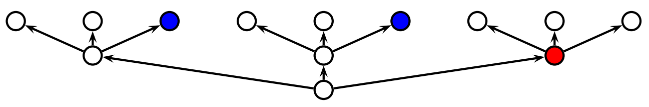
Computation Tree:



4.2 Asynchronous Setting

In an asynchronous setting this transformation is harder. As we have discussed in Section 3 it is possible into transform from a behaviour tree to a computation tree. But the question is, whether the other direction is also valid.

The answer is no, not every computation tree has a corresponding behaviour tree. Consider the following computation tree[3]:



It has no corresponding behaviour tree, since after the first scheduling of the program given input a, we get two different outputs at two different parts in the computation tree. Thus, at the same node in the behaviour tree the program outputs two different outputs, that is, the program behaves differently in the same situation. In the asynchronous setting, checking whether a computation tree has a corresponding behaviour tree cannot be done locally, since

[3]White nodes are either red or blue, we do not care about their value.

every node in the behaviour tree has infinitely many corresponding nodes in the computation tree. This is of course computationally infeasible.

5 Synthesis Algorithm

A synthesis algorithm takes a specification and generates an implementation that is guaranteed to satisfy this specification. Our aim is to present a synthesis algorithm for asynchronous systems. We will first describe the procedure for deterministic safety tree automata (*DSTA*), because of their simple acceptance condition, then show some extensions to other more general types of automata.

Initially we have an automaton that reads a computation tree and accepts if the tree satisfies a certain specification, that is, it is a singleton game going through the computation tree and trying to find a branch which does not satisfy the specification. The first step is to transform this automaton to a similar one which realizes the acceptance game on the behaviour tree.

In Subsection 5.1 we will explain how to do this automata transformation. The output of this transformation will be a universal ε safety tree automaton ($U\varepsilon STA$) which runs on a behaviour tree and accepts if the specification is satisfied. In Subsection 5.2 we will describe how to eliminate the ε transitions in the UεSTA automaton to get a corresponding universal safety tree automaton (*USTA*).

At this stage we have reduced our problem to a non-emptiness test for USTA, then generating a program that is guaranteed to satisfy the specification of the automaton.

This is an overview of the whole procedure:

DSTA A^{CT} $\Rightarrow$ UεSTA A^{BT} $\Rightarrow$ USTA A^{BT} $\Rightarrow$ NET $\Rightarrow$ Program

5.1 Automata Transformation

In this subsection we will describe how to transform a DSTA reading the computation tree to a UεSTA accepting the behaviour tree.

The automaton running on the computation tree A_{CT} is a four tuple $(2^I \times 2^S \times 2^O, Q^{dir}, q_0, \delta)$ where I is the input variable to the program, S is a boolean variable representing the scheduling decision (0 means environment scheduled and 1 program scheduled), and O is the output variable of the program. Q^{dir} are the states of the automaton labeled by directions. $q_0 \in Q^{dir}$ is the start state. $\delta(q, i, s, o) = (q_1, false, 0) \wedge (q_2, true, 0) \wedge (q_3, i, 1)$ is a function representing the transition relations. It is a partial function that is not defined on inputs that the safety automaton should reject on.

The corresponding automaton running on the behaviour tree A_{BT} is a four tuple $(Q \times 2^I \times 2^S \times 2^O, dir, q_0, \delta')$ where I, S, and O mean the same as in A_{CT}. They are added to the label of the states Q. $q_0 \in Q$ is the initial state. $\delta'(q, i, s, o) = (q_1, false, 0, \varepsilon) \wedge (q_2, true, 0, \varepsilon) \wedge (q_3, i, 1, i)$ is a partial function representing the transition relations.

5.2 Eliminating ε Transitions

In the last subsection we discussed how to transform a DSTA that reads the computation tree to a UεSTA that reads the behaviour tree. In this section we will explain how to remove those ε transitions and transform the obtained $U\varepsilon TA$ to an equivalent USTA that reads the behaviour tree. Removing the ε transitions could be done expanding the δ function using fixed point iteration. This is done as follows:

$$\delta'(q, i, s, o) = (q_1, false, 0, o) \wedge (q_2, true, 0, o) \wedge (q_3, i, 1, i)$$

6 Extensions

In the last section we have described a synthesis algorithm for DSTA. In this section we will briefly describe how this could be extended to other types of automata. Universal Safety Tree Automata are obtained for free, the only difference is that there will be more conjuncts in the transition relation. Universal Co-Büchi Automata are obtained by searching for ε cycles containing rejecting states. This is done during the process of eliminating ε transitions. If a rejecting state is found in a cycle, this information is saved and eventually the automata rejects. Note that LTL formulas could be transformed into Universal Co-Büchi Automata.

References

[1] S. Schewe and B. Finkbeiner. Synthesis of asynchronous systems. In *16th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR 2006)*, pages 127–142. Springer Verlag, 2006.