Using Simulation Relations for Synthesis

Satrajit Chatterjee (satrajit@cs.berkeley.edu)

May 21, 2004

Abstract

In this report we present an overview of using simulation relations for synthesis. We describe the generalization of the notion of simulation relations to the case when the FSMs have different inputs and present a construction for the solution to the unknown FSM component problem. We present an implementation in the MVSIS environment and discuss some modifications to the algorithm in order to accommodate the hybrid representation scheme for automata in MVSIS.

1 Introduction

In the FSM synthesis problem, we are given two FSMs: a fixed part \( F \) and a specification \( S \), and are asked to synthesize an FSM \( X \) (called the unknown component) such that \( F \) composed with \( X \) has the same behavior as \( S \). This problem which is a central problem of logic synthesis has received a lot of attention and many different solutions to this problem have been proposed in the literature.

Of late the theory of language equation solving [1] has emerged as a fairly comprehensive theoretical framework for sequential synthesis. In this scheme, the FSM synthesis problem is reduced to the problem of solving a language equation. Languages are represented by automata, and the automaton corresponding to the unknown component is constructed. From this automaton, the required FSM is derived. The use of this method for FSM synthesis has two major shortcomings. First, language equation solving solves a very general problem, and hence is computationally expensive. This makes the scheme impractical for even moderately large examples. Second, the solutions to a language equation form a (finite) lattice, and the procedure to solve the equation computes the largest solution. Translated to the FSM synthesis problem, this means that the FSM solution constructed from the solution of the language equation problem has many sub-solutions in it. It is an open question as to how to derive a good particular solution from the larger solution, since in the context of synthesis, we are ultimately interested in a specific implementation.

The limitations of language equation solving described above lead to the main thesis of this paper: It is beneficial to think of FSM synthesis directly in terms of FSMs (as opposed to languages and regular automata). By not solving the language equation derived from the FSM synthesis
problem, but by trying to solve the FSM synthesis problem directly, we can avoid the computational complexity of constructing a very general solution only to first trim it down to obtain an FSM and then further trim it down to obtain a realizable particular solution. Towards this end we study the idea of using simulation relations on FSMs proposed by Khatri et al. [2] in the context of Plant-Controller Synthesis. We extend this idea to a more general topology, and describe an implementation of the synthesis algorithm in MVSIS [3].

2 Terminology

An FSM $M$ is a 5-tuple $\langle S, I, O, T, r \rangle$ where $S$ is the set of states of the FSM, $I$ the input alphabet, $O$ the output alphabet, $T \subseteq S \times I \times O \times S$ is the transition relation and $r \in S$ is the initial state. $L(M) \subseteq (I, O)^\omega$ the trace language of $M$ is the set of infinite traces of the FSM. An element $w \in L(M)$ is a map from $Z_{\geq 0}$ to $(I, O)$. $L(M)$ is also called the behavior of $M$.

Consider the relation $R = \{(s, o) \mid \exists i \in I, \exists s' \in S \text{ s.t. } (s, i, o, s') \in T\}$. If $R$ is a function then $M$ is called a Moore machine.

3 FSM Synthesis using Simulation Relations

3.1 Simulation Relation for FSMs

Let $M_1 = \langle S_1, I, O, T_1, r_1 \rangle$ and $M_2 = \langle S_2, I, O, T_2, r_2 \rangle$. $\psi \subseteq S_1 \times S_2$ is a simulation relation from $M_1$ to $M_2$ iff

1. $(r_1, r_2) \in \psi$
2. $(s_1, s_2) \in \psi \implies$
   \[
   \forall i \forall o \forall s'_1 [(s_1 \xrightarrow{i/o} s'_1) \implies \exists s'_2 [(s_2 \xrightarrow{i/o} s'_2) \land (s'_1, s'_2) \in \psi]]
   \]

If there is a simulation relation from $M_1$ to $M_2$, then we say that $M_2$ simulates $M_1$ or $M_1$ has a simulation in to $M_2$. This is written as $M_1 \preceq M_2$. 

![Figure 1: The topology for the FSM synthesis problem considered in this paper.](image-url)
3.2 The FSM Synthesis Problem

In the FSM synthesis problem we consider, we are given a fixed FSM $F = \langle S_F, I \times V, U \times O, T_F, r_F \rangle$ and a overall specification FSM $S = \langle S_S, I, O, T_S, r_S \rangle$. Our task is to construct an FSM $X = \langle S_X, U, V, T_X, r_X \rangle$ such that $F \circ X \leq S$. Figure 1 shows the topology. We assume that $F$ and $S$ are deterministic machines. Furthermore we assume that $F$ is a Moore machine and require the solution $X$ to be Moore as well.

3.3 Generalized Simulation Relation

In order to solve the FSM synthesis problem, we follow the approach of Khatri et al. [2] in defining a generalized simulation relation from $F$ to $S$. (Note that simulation as defined in Section 3.1 does not work since $F$ and $S$ have different inputs and outputs.) A relation $\psi \subseteq S_F \times S_S$ is a generalized simulation relation from $M_F$ to $M_S$ iff

1. $(r_F, r_S) \in \psi$
2. $(s_F, s_S) \in \psi \Rightarrow \{ \forall i \exists v \forall u \forall o \exists s'_F$
   
   $[(s_F \xrightarrow{i/v} s_F) \Rightarrow \exists s'_S((s_S \xrightarrow{i/o} s'_S) \land (s'_F, s'_S) \in \psi_{max})]\}$

In what follows we shall abuse notation and say simulation even when we mean generalized simulation.

We claim (without proof) that the FSM problem has a solution iff there exists a simulation relation from $F$ to $S$. Intuitively this corresponds to the fact that $F$ can be controlled if and only if for all $i$ the environment can produce $X$ can give $F$ an input $v$ such that both $F$ and $S$ produce same output $o$ and go to next states $(s'_F, s'_S)$ such that the same is true at $(s'_F, s'_S)$.

3.4 Synthesis Algorithm

The synthesis is done in two steps. In the first step the largest simulation relation $\psi_{max}$ from $F$ to $S$ is found. The state pairs in $\psi_{max}$ comprise the states of $X$, i.e. $S_X = \psi_{max}$. In the second step, edges are added between these states and thus $T_X$ is determined. The initial state of the new machine is $r_X = (r_F, r_S)$. Note that $(r_F, r_S) \in \psi_{max}$ by virtue of $\psi_{max}$ being a simulation relation.

3.4.1 Computation of $\psi_{max}$

This is done as a greatest-fixed point computation. The algorithm is shown as Algorithm 1. Initially we start with all pairs of states from $F$ and $S$. At each stage, we use the condition for simulation to prune pairs of states which cannot be in the simulation relation.

3.4.2 Construction of $X$

Let $X = \langle S_X, U, V, T_X, r_X \rangle$. Set $S_X = \psi_{max}$, $r_X = (r_F, r_S)$. Let $s = (s_F, s_S)$ and $s' = (s'_F, s'_S)$ be two states of $X$. Now $(s, u, v, s') \in T_X$ iff $\exists \exists o$ s.t. $(s_F, (i, v), (u, o), s'_F) \in T_F$ and $(s_S, (i, o), s'_S) \in T_S$. 3
### Algorithm 1 Algorithm to compute $\psi_{\text{max}}$, the largest simulation relation.

<table>
<thead>
<tr>
<th>INPUT: $F$, $S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUT: $\psi_{\text{max}}$ if one exists or NO SOLUTION</td>
</tr>
<tr>
<td>$i \leftarrow 0$</td>
</tr>
<tr>
<td>$R_0(s_F, s_S) = 1$</td>
</tr>
<tr>
<td><strong>repeat</strong></td>
</tr>
<tr>
<td>$R_{i+1}(s_F, s_S) \leftarrow R_i(s_F, s_S) \land$</td>
</tr>
<tr>
<td>$\forall i \exists v \forall u \forall s' \quad ((s_F \xrightarrow{iv/uo} s_F') \Rightarrow \exists s'_S ((s_S \xrightarrow{i/o} s'_S) \land R_i(s'_F, s'_S)))$</td>
</tr>
<tr>
<td><strong>if</strong> $(r_F, r_S) \notin R_{i+1}$ <strong>then</strong></td>
</tr>
<tr>
<td><strong>return</strong> NO SOLUTION</td>
</tr>
<tr>
<td><strong>end if</strong></td>
</tr>
<tr>
<td>$i \leftarrow i + 1$</td>
</tr>
<tr>
<td><strong>until</strong> $R_{i+1}(s_F, s_S) = R_i(s_F, s_S)$</td>
</tr>
<tr>
<td><strong>return</strong> $R_{i+1}(s_F, s_S)$</td>
</tr>
</tbody>
</table>

### 4 Implementation of the Synthesis Algorithm

In this section we describe an implementation of the synthesis algorithm from Section 3.4 in the MVSIS logic synthesis environment. It is implemented as the `srsolve` command which takes in two automata corresponding to the state transition graphs of the specification and the fixed part, and generates the transition graph for the solution FSM.

The `srsolve` command starts by computing $\psi_{\text{max}}$ and then constructing the solution in the manner described in Section 3. Algorithm 1 cannot be directly used for computing $\psi_{\text{max}}$ because MVSIS uses a hybrid representation for FSMs where the states are explicit and the transition conditions are implicit in terms of the variables for the inputs and outputs. In order to compute with this hybrid representation, we need to simplify Condition (2) from Section 3.3.

**Lemma 1** $f(x) \Rightarrow \exists y \ g(x, y) \iff \exists y (f(x) \Rightarrow g(x, y))$

**Proof**

\[
\begin{align*}
  f(x) & \Rightarrow \exists y \ g(x, y) \\
  \iff & \ f(x) \Rightarrow (g(x, Y_0) \lor g(x, Y_1)) \\
  \iff & \ f(x) \lor (g(x, Y_0) \lor g(x, Y_1)) \\
  \iff & \ f(x) \lor g(x, Y_0) \lor \overline{f(x)} \lor g(x, Y_1) \\
  \iff & \ f(x) \Rightarrow g(x, Y_0) \lor f(x) \Rightarrow g(x, Y_1) \\
  \iff & \ \exists y \ (f(x) \Rightarrow g(x, y))
\end{align*}
\]
Algorithm 2 Algorithm to compute \( \psi_{\text{max}} \) when states are stored explicitly.

\[
\text{INPUT: } F, S \\
\text{OUTPUT: } \psi_{\text{max}} \text{ if one exists or NO SOLUTION} \\
i \leftarrow 0 \\
R_0 = S_F \times S_S \\
\text{repeat} \\
R_{i+1} \leftarrow \text{filter keep } R_i, \\
\text{if } (r_F, r_S) \notin R_{i+1} \text{ then} \\
\quad \text{return NO SOLUTION} \\
\text{end if} \\
i \leftarrow i + 1 \\
\text{until } R_{i+1} = R_i \\
\text{return } R_{i+1}
\]

Using this lemma, Condition (2) may be re-written as

\[
(s_F, s_S) \in \psi \Rightarrow \{ \forall i \exists v \forall u \forall s'_F \exists s'_S[(s_F \xrightarrow{iv/uo} s'_F) \\
\Rightarrow [(s_S \xrightarrow{i/u} s'_S) \land (s'_F, s'_S) \in \psi_{\text{max}}]] \}
\]

Now since \( F \) is deterministic, for a given \((i, v)\) there is only one next state. Hence the \( \forall s'_F \) can be replaced with an \( \exists s'_F \) to get the following condition:

\[
(s_F, s_S) \in \psi \Rightarrow \{ \forall i \exists v \forall u \exists s'_F \exists s'_S[(s_F \xrightarrow{iv/uo} s'_F) \\
\Rightarrow [(s_S \xrightarrow{i/u} s'_S) \land (s'_F, s'_S) \in \psi_{\text{max}}]] \}
\]

Note that now the innermost quantifiers are contiguous and that they are for the explicit variables. This enables us to explicitly iterate through the states and collect the inner relation into an implicit variable which can then be used for the quantification of the remaining variables which are all implicit. The overall algorithm is presented as Algorithm 2. The function keep takes a state pair \((s_F, s_S)\) and return true or false depending on whether the above condition is satisfied. It is shown as Algorithm 3.

Note that after the above condition is evaluated in the keep function, it is necessary to restrict it to the domain of definition of the local conditions for the transitions. This is done by existentially quantifying out the outputs so that the resultant function gives the domain (in terms of \( I \) and \( V \)) over which the transition conditions (and consequently the similarity condition derived from them) is valid.

5 Example

We applied the synthesis technique presented here to a simple example. The fixed FSM \( F \) is a two-bit counter that takes an input \( v \) which if true causes the counter to skip the next state; otherwise it behaves as a conventional counter. \( u_1 \) and \( u_0 \) are the output bits of this counter which are available to the unknown component. The input \( i \) of the counter is a
Algorithm 3 Algorithm for \texttt{keep} function from Algorithm 2.

**INPUT:** $F$, $S$, $(s_F, s_S) \in S_F \times S_S$

**OUTPUT:** true or false

$bDomain \leftarrow \phi$

\begin{itemize}
  \item for each transition $t_F$ from $s_F$ do
    \begin{itemize}
      \item for each transition $t_S$ from $s_S$ do
        \begin{itemize}
          \item $bIsInR \leftarrow (\text{dest}(t_F), \text{dest}(t_S)) \in R_i$
          \item $bCond \leftarrow \text{cond}(t_F) \Rightarrow (\text{cond}(t_S) \land bIsInR)$
          \item $bCond \leftarrow bCond \land (\exists u \exists o (\text{cond}(t_F) \land \text{cond}(t_S)))$
        \end{itemize}
      \item $bDomain \leftarrow bDomain \lor bCond$
    \end{itemize}
  \end{itemize}

\textbf{return} $\forall i \exists v \forall u \forall o bDomain$

don’t care. Figure 2 shows the state transition diagram. The specification is a one bit counter which is obtained by setting the input $v$ of the fixed to true (Figure 3). Figure 4 shows the unknown component that is synthesized using simulation relation. As expected the 00 state in $S$ simulates the 00 state in $F$, and the 11 state in $S$ simulates the 11 state in $S$. The result is a two state automaton which can be minimized to get the minimum solution. Figure 5 shows the solution obtained by language equation solving. Note that that solution contains the simulation relation solution, though the minimum solution cannot be directly inferred. (Don’t care minimization (\texttt{dcmin}) is needed to obtain that solution.)

6 Conclusion and Future Work

Simulation relations are an interesting and efficient technique for sequential synthesis. Currently most of the code is in place in MVSIS permitting experimentation with larger benchmarks. An interesting addition to the \texttt{srsolve} command would be the ability to take an user specified relation as input (in the form of a MV-network) and use that as the starting point for Algorithm 3 (instead of using all state pairs). Practically this may lead to faster solutions for some class of synthesis problems by avoiding the quadratic step. Another useful speed-up technique might be some sort of pre-filter for the states in the specification and the fixed part to decide similarity. While this might not prevent the quadratic runtime behavior, it would serve to improve the capacity of the tool since all state pairs need not be stored during the computation. Finally, reformulating this computation in terms of circuit-based manipulations might be a fruitful area of research.

References

Figure 2: The state transition graph for Fixed $F$ from Section 5. The variable order for the transitions is $i, v, u_1, u_0$.

Figure 3: The state transition graph for Specification $S$ from Section 5. The variable order for the transitions is $i, u_1, u_0$. 
Figure 4: The state transition graph for the solution $X$ obtained by synthesis based on simulation relations. The variable order for the transitions is $u_1, u_0, v$.

Figure 5: The state transition graph for the solution $X$ obtained by language equation solving. The variable order for the transitions is $u_1, u_0, v$. 