CMU 15-671	Models of Software Systems	Fall 1997

Relating State Machines: Satisfies

Garlan	Handout 10	8 October 1997

This handout discusses the problem of whether one machine *satisfies* (in some sense) another. It should be read simultaneously (if that's possible!) with Handouts 10 and 11, which give specific examples.

Just as there is not one standard notion of equivalence, there is not one standard notion of "satisfies." In this handout we give a definition that is reasonable, representative, and used in practice.

There are three key ideas that this handout presents.

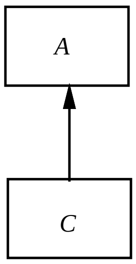
- A definition of *satisfies* in terms of a subset relationship between behavior sets of two machines.
- The notion of a mapping, called an *abstraction function*, that relates states of one machine to states of another.
- A proof technique that uses the abstraction function in a "commuting" diagram (see Section ??) to relate state transitions of one machine to state transitions of another.

Although our notion of *satisfies* and the proof technique that I present here may seem specific to the model of state machines that I have described so far, what's the most important idea for you to learn from these handouts is the notion of an abstraction function. You will also see it again, in a different guise, when we cover refinement in Z.

1 Why Would You Care About "Satisfies"?

Suppose your favorite programming language does not support the notion of a set (i.e., *set* is not a built-in type) and I asked you to implement a set type in terms of the built-in types of your language, e.g., sequences (or arrays, more likely). This is a standard exercise in programming with data abstraction. For example, you might (1) represent sets in terms of sequences and then (2) implement each set operation in terms of operations on sequences and other built-in types. After you hand in your solution, I ask you to prove that your representation using sequences satisfies the abstract set type. How would you respond?

What we really care about is the relationship between a *concrete* state machine, C, and an *abstract* state machine, A:



where we usually read the arrow in one of many ways:

C satisfies A.
C implements A.
C is a refinement of A.
The program *C is correct with respect to* the specification A.

C and A can be interpreted in many ways:

C and A are both state machines.
C is a program and A is a specification.
C is an implementation and A is a specification.
C is a state machine and A is a predicate.
C and A are both programs.
C and A are both specifications.
C is an implementation and A is an interface specification.
C is a concrete data type and A is an abstract data type.
C is an Ada package body and A is a package specification.
C is a C function definition and A is a C function declaration.
C is a C++ class definition (implementation) and A is a class declaration (interface).
C is a high-level design and A is a set of customer's requirements.
C is a low-level design and A is a high-level design.
C is an implementation and A is a low-level design.

2 What Does *Satisfies* Mean?

2.1 Binary Relations

Consider a simpler problem of determining when one binary relation *satisfies* another. Suppose we have a specification for a *square_root* procedure:

$$\begin{array}{l} \textit{square\_root(x: int)/ok(int)} \\ \quad \textbf{pre } \exists\, i : \textit{int} \bullet x = i * i \\ \quad \textbf{post } x = \textit{result} * \textit{result} \end{array}$$

that denotes the binary relation:

$$R_A = \{(4,2),(4,-2),(9,3),(9,-3),(16,4),(16,-4)\ldots\}$$

I could choose to implement this procedure such that I always return the positive square root of an integer. Its relation is:

$$R_C = \{(4,2),(9,3),(16,4)\ldots\}$$

Informally, my implementation *satisfies* the specification because R_C just narrows the possible choice of the integer returned allowed by R_A and the implementation defines some value for each input integer that is defined for the specification.

More formally, we have, given an abstract relation, R_A, and a concrete relation, R_C:

Definition 1 R_C **satisfies** R_A *iff:*

- $R_C \subseteq R_A$ *and*
- $\textit{dom}\ R_C = \textit{dom}\ R_A$.

The first property says that any pair in the concrete relation is also an element of the abstract relation. The second property says that for each input that is related by R_A, we want R_C to be defined. Without the second property, R_C could be empty and the *satisfies* relation would hold[1].

2.2 State Machines

Consider the two state machines,

$$A = (S_A, I_A, A_A, \delta_A)$$
$$C = (S_C, I_C, A_C, \delta_C)$$

each denoting a behavior set, $Beh(C)$ and $Beh(A)$, respectively. We take an event-based view of trace: Each trace is a sequence of (invocation, response) pairs and each pair represents a single execution of one of the actions provided by the machine. We assume for simplicity that there is a one-to-one correspondence between the action names in the concrete machine to those in the abstract machine and that we use a renaming function, α, to define the relationship:

$$\alpha : A_C \rightarrow A_A$$

Using α we can relate each state transition involving a C action to a state transition involving an A action.[2]

We define the *satisfies* relation as follows between two state machines as follows:

Definition 2 *C* **satisfies** *A iff $Beh(C) \subseteq Beh(A)$.*

Since $Beh(C)$ is a set of traces and $Beh(A)$ is a set of traces, the *satisfies* relation is satisfied if every trace in $Beh(C)$ is in $Beh(A)$. This means that A's set can be larger; C's set reduces the choices of possible acceptable behaviors.

Why does this definition of *satisfies* make sense? Viewing C as an implementation of a design specification A, this definition says that an implementor makes decisions that restrict the scope of the freedom *allowed by* a design. In other words, the specification is saying what *may* or *is permitted* to occur at the implementation level, not what *must* occur. The implementation narrows down the choice of what is allowed to happen. For example, an implementation might reduce the amount of nondeterminism allowed by the specification.

Thus, certainly having the behavior sets equal ($Beh(C) = Beh(A)$) is too strong. Having the subset relation go the other way ($Beh(A) \subseteq Beh(C)$) cannot be right either; otherwise there may be executions of the concrete machine that are not permitted by the abstract one.

According to this definition of correctness the concrete machine with the empty behavior would be a perfectly acceptable implementation of an abstract machine. Since the empty behavior is not very satisfying, we normally assume that the set of initial states and the state transition function for the concrete machine are both non-empty; thus its behavior is non-empty.

The empty behavior case is the extreme case where the concrete machine does not do anything bad (a safety property)[3] since it does not do anything at all; however, the machine also does not do anything good (a liveness property). Our definition of *satisfies* does not require that our machine do anything; the definition requires only that the machine does only allowed things.

3 Showing One Machine Satisfies Another

How do we show the *satisfies* relationship holds between two state machines? Given our two state machines, $A = (S_A, I_A, A_A, \delta_A)$ and $C = (S_C, I_C, A_C, \delta_C)$, in general it is not so straightforward to determine if C *satisfies* A because their state sets, S_A and S_C, may be different. The proof technique we present uses an *abstraction function* to relate these state sets[4].

[1] A very "unsatisfying" relation!

[2] In all the examples that you will see the renaming function will be obvious. I'm being overly pedantic here.

[3] See end of Handout on Reasoning About State Machines.

[4] Much like we use the renaming function α to relate different actions sets.

3.1 A Proof Technique

There are two major steps in the proof technique for showing one machine satisfies another.

1. The Creative (Intellectually Hard) Step.

   - Define an *abstraction function*

     $$AF : S_C \rightarrow S_A$$

     to relate concrete states with abstract states.

   - Define a *representation* (sometimes called *concrete*) invariant,

     $$RI : S_C \rightarrow bool$$

     that characterizes the domain of AF. This predicate prunes down the set of concrete states, S_C, to only those of interest (only those that represent some abstract state in S_A). After you define this predicate, you must prove that it indeed is an invariant. You may use the technique presented in the Reasoning About State Machines handout to show this.

2. The Checklist (Tedious) Step.

   - For each $i_C \in I_C$ (for all initial states in I_C) show:

     $$AF(i_C) \in I_A$$

     This requires showing that each initial state of the concrete machine is some initial state of the abstract machine.

   - For each state transition $(y, c, y') \in \delta_C$ of the concrete machine C where y satisfies RI, there must exist a state transition $(AF(y), \alpha(c), AF(y')) \in \delta_A$ of the abstract machine A. To show this, it suffices to show the following commutative relationship holds:

     $$\begin{tikzcd} x \arrow[r, bend left, "a"] & x' \\ y \arrow[u, "AF"] \arrow[r, bend right, "c"'] & y' \arrow[u, "AF"'] \end{tikzcd}$$

     where $x = AF(y), x' = AF(y')$ and $a = \alpha(c)$. Here's how to read the diagram: Put both index fingers at the concrete state y; move your right one to the right (performing the concrete action c) and then up (applying AF to the new concrete state y'); move your left one up (applying AF to y) and then to the right (performing the corresponding abstract action $a = \alpha(c)$; you should end up in the same place: the abstract state, x'.

Please brand this commuting diagram on your brain.

3.2 Rationale

Why does this proof technique make sense? The technique should smell familiar. It's inductive in nature. There's a base case ("for each initial state ...") and an inductive step, defined in terms of all possible action instances ("for each state transitions ..."). As before, because the action sets are finite, we can do a big case analysis, one action per case, in the inductive step.

- In the base step, notice there is no requirement that all initial states of the abstract machine be covered. So, there may be some abstract executions that have no corresponding concrete execution since we could not even get started. This is okay since we only need to show a subset relationship between behavior sets.
- The intuition behind the inductive step is that if a state transition can occur in the concrete machine then it must be *allowed to* occur in the abstract machine. If we show the inductive step for all state transitions in the concrete machine, then we will have shown it for all of its possible executions. (Notice that in showing the commuting diagram for each action of the concrete machine, we are really showing it for each state transition that involves that action.)

After showing the base case and inductive step, we will have shown that each trace in the behavior set of the concrete machine has a corresponding (modulo the abstraction function) trace in the behavior set of the abstract machine. Hence, the behavior set of the concrete machine is a subset of the behavior set of the abstract machine (modulo the abstraction function), which is what we needed to prove.

Aside: In our proof technique, because of our simplistic way of associating actions in one machine to another (through the one-to-one function α) we are associating a single state transition in C to a single state transition in A. More generally, a state transition in C might map to a *sequence* of transitions in the abstract machine A. This generalization is especially needed for proving concurrent state machines correct and/or dealing with state machines with internal as well as external actions. We won't be using this generalization at all.

3.3 Abstraction Functions and Representation Invariants

An abstraction function maps a state of the concrete machine to a state of the abstract machine. It explains how to interpret each state of the concrete machine as a state of the abstract machine. It solves the problem of the concrete and abstract machines having different sets of states.

Since the abstraction function is a function, we can rely on the substitution property: If AF is a function and we know $x = y$ then we know $AF(x) = AF(y)$. If AF were a more general kind of relation, this property would not necessarily hold.

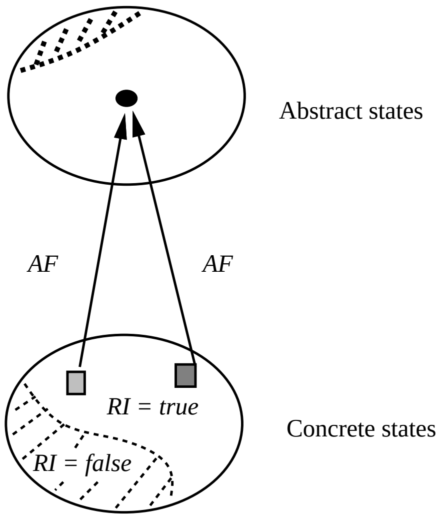
You might think that the abstraction function should map the other way, explaining how to represent each state of the abstract machine. This is usually not a good idea, in large part because there is often more than one way of representing each state of the abstract machine. For example, suppose I represent a set by a sequence. Then many *different* sequences, e.g., $\langle 3, 5, -1\rangle$, $\langle 5, 3, -1\rangle$, $\langle -1, 5, 3\rangle$, $\langle 3, 5, 5, -1\rangle$, could (given the appropriate definition of AF and RI) all represent the *same* set $\{3, 5, -1\}$.

In other words, AF, in general, is many-to-one.

AF may be partial. Not all states of the concrete machine may represent a "legal" abstract state. For example, in the integer-modulo7 example, integers not within 0..6 are not "legal" representations of days of the week. The representation invariant serves to restrict the domain of the abstraction function. We may assume that for any concrete state for which the representation invariant does not hold, AF is undefined.

Finally, AF is not necessarily onto. There may be states of the abstract machine that are not represented by any state of the concrete machine. This can be true of initial states as well. In the context of showing that one machine adequately implements another, this may sound strange to you; we say more on *adequacy* in Section ??.

Putting everything together, we have the final diagram:



where the bottom *unshaded* region represents the domain of AF and the top *unshaded* region represents its range.

3.4 Variations on a Theme

Adequacy

In this handout we explicitly stated that AF need not be surjective (onto). Were we to require AF to be surjective, then we would require that every abstract state have some concrete representation, i.e., that AF is *adequate*. Requiring AF to be adequate makes very good sense since you might like to know that every abstract state you have modeled is implemented by some concrete state. Some refinement methods like VDM require that AF be adequate. And, in proving the correctnesss of an abstract data type, we usually require AF to be adequate for the concrete representation type.

As mentioned, we also do not require adequacy in the sense of having every state transition of A be implemented in terms of one in C. Rather, we only require that every state transition in C relate to some state transition of A. (C cannot do anything not permitted by A.) This laxity is in contrast to how we defined whether one binary relation, R_C, *satisfies* another, R_A; we required that for each input related by R_A, R_C should be defined. This requirement gets at the *adequacy* of R_C viewed as an implementation of R_A.

Taking this last point to an extreme, we do not even require that every action in A actually be "implemented" by some action (or sequence of actions) in C. In other words there can be state transitions, all associated with a particular action of A, that have no correspondence are not adequately represented) in C.

You will see later in the course when we cover Z and CSP that other state machine-based models impose different kinds of adequacy restrictions in defining a *refinement/correctness* relation between two machines.

Abstraction Relations

Some people prefer to use abstraction *relations* or abstraction *mappings* more generally than functions. There are examples where it is easier, more convenient, or more natural to map a concrete state to a set of abstract states. As mentioned, however, you would lose the substitution property of abstraction functions.

Auxiliary Variables

Sometimes it is not so straightforward to prove a concrete machine satisfies an abstract machine in terms of just the state variables of the concrete machine. In this case, you need to introduce *auxiliary variables* (sometimes called *dummy variables* or *history variables* in the literature). The need for auxiliary variables in such proofs is especially common for reasoning about concurrent programs.