State machines

Its models are state machines with F = {i} and P = {R, F}, where i is a constant, F a predicate symbol with one argument and R a predicate symbol with two arguments. A (concrete) model M contains a set of concrete elements A – which may be a set of states of a computer program. The interpretations i M ∈ A, RM ∈ A × A, and FM ⊆ A are understood to be a designated initial state, a state transition relation, and a set of final (accepting) states, respectively. Model M is concrete since there is nothing left un-specified and all checks M φ have definite answers: they either hold or they don't. In practice not all functional or other requirements of a software system are known in advance, and they are likely to change during its lifecycle. For example, we may not know how many states there will be; and some transitions may be mandatory whereas others may be optional in an implementation. Conceptually, we seek a description M of all compliant implementations Mi (i ∈ I) of some software system. Given some matching property ψ, we then want to know

- (assertion checking) whether ψ holds in all implementations Mi ∈ M; or
- (consistency checking) whether ψ holds in some implementation Mi ∈ M.

For example, let M be the set of all concrete models of state machines, as above. A possible assertion check ψ is 'Final states are never initial states.' An example of a consistency check ψ is 'There are state machines that contain a non-final but deadlocked state.' As remarked earlier, if M were the set of all state machines, then checking properties would risk being undecidable, and would at least be intractable. If M consists of a single model, then checking properties would be decidable; but a single model is not general enough. It would comit us to instantiating several parameters which are not part of the requirements of a state machine, such as its size and detailed construction. A better idea is to fix a finite bound on the size of models, and check whether all models of that size that satisfy the requirements also satisfy the property under consideration.

- If we get a positive answer, we are somewhat confident that the property holds in all models. In this case, the answer is not conclusive, because there could be a larger model which fails the property, but nevertheless a positive answer gives us some confidence.
- If we get a negative answer, then we have found a model in M which violates the property. In that case, we have a conclusive answer, and can inspect the model in question

Alma

It names the module About Alma and defines a simple signature of type Person. Then it declares a signature SoapOpera which has a cast – a set of type Person – a designated cast member alma, and a relation loves of type cast -> cast. We check the assertion OfLovers in a scope of at most two persons and at most one soap opera.

1. Expressions of the form all x : T | F state that formula F is true for all instances x of type T. So the assertion states that with S {...} is true for all soap operas S.

```
module AboutAlma
sig Person {}
sig SoapOpera {
cast : set Person,
alma : cast,
loves : cast -> cast
}
assert OfLovers {
all S : SoapOpera |
with S {
all x, y : cast |
alma in x.loves && x in y.loves => not alma in y.loves
}
}
```

2. The expression with S {...} is a convenient notation that allows us to write loves and cast instead of the needed S.loves and S.cast (respectively) within its curly brackets.
3. Its body ... states that for all x, and y in the cast of S, if alma is loved by x and x is loved by y, then – the symbol => expresses implication – alma is not loved by y.