Verification Methods for Behavioural Specifications

MICHIHIRO MATSUMOTO

School of Information Science,
Japan Advanced Institute of Science and Technology

February 13, 1998

Keywords: concurrent system, algebraic specification, verification, refinement, term rewriting system.

The purposes of our research are to clear problems of previous verification methods for behavioural specifications and to propose improved verification methods. We want to decrease costs of developments of concurrent systems. As a candidate of solutions, we selected verification methods for behavioural specifications.

Concurrent systems are constructed from many objects that communicate with other objects. Because possible states and transitions are huge, numbers of the necessary tests to ensure reliability are also huge. Therefore, costs of these tests is high. On the other hand, logical verifications can find bugs of the logical level and costs of logical verifications is lower than those of the tests. Consequently, we can expect the costs of developments to decrease by exchanging the tests for a combination of tests and logical verifications.

From this expectation, logical verification methods have been studied in process algebras. We think abstract data types (abbreviate ADT) have key roles when we verify data flows over concurrent systems. But, most of process algebras can not deal with ADT. In behavioural semantics (hidden algebras), concurrent systems are treated as black boxes. So, behavioural semantics (hidden algebras) can be seen as a generalization of process algebras which can deal with ADT. So, we can adapt the techniques provided in process algebras for behavioural semantics (hidden algebras) and we can deal with ADT in behavioural semantics (hidden algebras). Therefore, we selected behavioural semantics (hidden algebras) as the foundations of our research. Behavioural specifications are specifications whose semantics are behavioural semantics.

In behavioural specification, we specify interactions between a concurrent system and a user. Operations which observe the states (of the concurrent system) are called attributes and operations which change the states are called methods. Attributes and methods

are called behavioural operators. We can only recognize the current state by observing states changed by methods through attributes. So, we can regard method sequences with an attribute as observation tools. these observation tools are called visible contexts. Behavioural equivalence $\equiv$ between states s, s' are defined as follows:

$(s \equiv s') = \wedge_{ct \in VisCt}(ct[s] == ct[s'])$

where $VisCt$ is the set of all visible contexts. In behavioural specifications, we verify behavioural properties that are behavioural equivalence relations between states of concurrent systems.

As to verification methods for behavioural specifications, there are coinduction and induction over length of contexts. The main application of these verification methods is stepwise refinement of behavioural specifications as restriction of possible implementations.

Coinduction is a verification method based on the following fact:

behavioural equivalence is the largest hidden congruence,

where a hidden congruence is a congruence such that: identity relation on data values. Consider to verify a behavioural property $s \equiv s'$. The algorithm of coinduction is as follows:

1. find a candidate R of hidden congruences,
2. check whether R is a hidden congruence, and
3. verify whether $s \equiv s'$ holds, by proving $s\ R\ s'$.

So, to use coinduction, users must find a hidden congruence. Until now, this hidden congruence should be given by hand.

Note that relations which can be defined on verification systems are relations defined by syntax — we call these relations **syntactically definable hidden congruences**. Firstly, we show that the only useful syntactically definable hidden congruence for verifications is behavioural equivalence. Therefore, R should be behavioural equivalence. Behavioural equivalence is the conjunction over all visible contexts. Consequently, a selection of hidden congruences corresponds to a selection of the set of visible contexts which construct behavioural equivalence. We let R denote the form of behavioural equivalence defined by syntax — conjunction over visible contexts — and we let $\#(R)$ denote the numbers of these visible contexts. We regard a verification method which use R as an **efficient method** if $\#(R)$ is small. We regard R as a **simple form** if $\#(R)$ is small. So, to verify behavioural properties efficiently, we need a simple form of behavioural equivalence. By eliminating redundant visible contexts, we get this simple form. We provide the algorithm which generates this simple form. That is *GSB*-**algorithm** (**test set coinduction**).

Consider to verify a behavioural property $s \equiv s'$. The algorithm of test set coinduction is as follows:

1. generate a simple form R of behavioural equivalence (by *GSB*-algorithm), and
2. verify whether $s \equiv s'$ holds, by proving $s\ R\ s'$.

By analysing the structure of the set of all visible contexts, we show the sufficient condition that *GSB*-algorithm can eliminate all redundant visible contexts.

Until now, coinduction was regarded more efficient than induction over length of contexts. By analysing the structure of the set of all visible contexts, we show the case that coinduction (test set coinduction) coincides with induction over length of contexts.

As to research of stepwise refinements of behavioural specifications as restriction of possible implementations, there are researches by Dr.Goguen and Dr.Malcolm. But, these are not satisfactory. Firstly, they give the original specification (for example, a stack). Then, they construct it from primitive modules (for example, an array and a pointer) in the refined specification. Finally, they prove that the composed module (for example, a stack constructed from the array and the pointer) satisfy the original specification. In the last process, they treat the composed module as data. But, in behavioural specification, specifications of concurrent systems must be treated as black boxes. Concretely, the problem is that there are states of the composed module which do not correspond to states of primitive modules.

We provide **projection operators** which specify correspondences between states of composed module and states of primitive modules. We provide the method which construct a composed module from primitive modules using these projection operators. We call the specifications which are written under the above method **object-oriented specifications**. Specifying concurrent systems by using object-oriented specifications, we solved the above problem. Moreover, we provide the method to verify stepwise refinement of object-oriented specifications.

As to the previous version of projection operators — we call these operators **pseudo-projection operators** in this paper —, there is a co-operative research with Mr.Iida, Dr.Diaconescu, and Dr.Lucanu. We only wrote our contribution in this paper. In this co-operative research, we specify dynamic systems using pseudo-projection operators. By changing contents of *ObjId* dynamically, we can specify dynamic systems.

The difference between projection operators and pseudo-projection operators is that projection operators are ordinary operators but pseudo-projection operators are behavioural operators. Consider to construct a stack from an array and a pointer. If we use pseudo-projection operators, we get just an array with a pointer. We can observe all contents of the array through visible contexts. On the other hand, if we use projection operator, we get a stack. We can only observe contents under a pointer. By using projection operator, we can restrict the set of visible contexts. To compose modules, this kind of restriction is necessary.