JSS 7461
14 March 2003 Disk used
ARTICLE IN PRESS
No. of Pages 9, DTD = 4.3.1
SPS-N, Chennai

ELSEVIER

Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

The Journal of Systems and Software xxx (2003) xxx–xxx

The Journal of Systems and Software

www.elsevier.com/locate/jss

Temporal logic properties of Java objects

Radu Iosif *, Riccardo Sisto

Dipartimento di Automatica Politecnico di Torino, corso Duca degli Abruzzi 24, 10129 Torino, Italy

Received 23 December 2002; accepted 27 December 2002

Abstract

Applying finite-state verification techniques to software systems looks attractive because they are capable of detecting very subtle defects in the logic design of these systems. Nevertheless, the integration of existing formal verification tools within programming environments is not yet easy, mainly because of the semantic gap between widely used programming languages and the languages used to describe system requirements. In this paper, we propose a formal requirement specification notation based on linear temporal logic, with regard to object oriented program elements, such as classes and interfaces. The specification is inherently object oriented and is meant for the verification of concurrent and distributed software systems.

Keywords: Source code verification; Linear temporal logic; Object orientation; Java

1. Introduction

Thanks to the recent advances in tool support, finite-state verification (FSV) techniques such as model checking (Holzmann, n.d.) can now be applied with interesting results to the verification of concurrent software systems. Nevertheless, much work is still needed to enable the transition of these techniques from research to actual practice. On one hand, verification techniques are generally difficult to use and not yet well integrated in common programming environments programmers are used to. On the other hand, most of these techniques adhere to a monolithic and rather static model of software, which is no longer adequate to the new programming paradigms in use today. It is a matter of fact that object-oriented (OO) languages and middleware like for example Java and CORBA, providing concurrent, distributed and even mobile objects, are becoming one of the most common tools for building applications. Despite this fact, the new features of this kind of software, mainly dynamicity and object-orientation, are not well tackled by the existing verification tools.

In this paper we focus attention on model checking techniques for concurrent and distributed object oriented source programs, and address the problem of specifying temporal logic properties related to this kind of software. Our specific objective is defining a formal but user-friendly specification technique for expressing properties which follows the object oriented approach, is well integrated in the source code the programmer is familiar with, and can easily express what is typically needed. The Java language is taken as a reference for developing the proposed specification technique, even though the method deals with common OO ideas, which makes it suitable for other similar languages, like C++ and CORBA IDL.

As we are considering OO software, it is of crucial importance to be able to associate properties with the language elements used by the programmer, i.e. classes and interfaces, and to exploit the mechanisms of object orientation such as inheritance as much as possible and consistently with the common programming practice. OO programs are actually collections of classes, possibly grouped into packages, and, in the OO philosophy, all these are reusable software modules. This implies that it is important to be able to assess and verify not only properties related to an application as a whole, but also properties that each single reusable class or package should satisfy, and these are typically required to hold

* Corresponding author.
E-mail addresses: iosif@athena.polito.it (R. Iosif), sisto@polito.it (R. Sisto).

0164-1212/03/$ - see front matter © 2003 Published by Elsevier Science Inc.
doi:10.1016/S0164-1212(03)00062-1

somewhat independently of the way the classes or packages will actually be used. If we consider distributed OO software systems, it is even possible that the whole program does not yet exist when verification is to be done, because in such systems server objects are generally made available to unknown clients. Since clients may eventually be developed later on, the verification of servers has to be done without having a complete application at hand, but only having the source code of some classes.

Expressing properties associated with classes, and not with elements of the global program state as in the classical approach, opens new problems. As long as only static, i.e. global, variables are involved in the properties, the meaning of temporal logic formulae is exactly the same as with other non-object-oriented programs, because the lifetime of static variables coincides with the program lifetime. Instead, if for example formulae are associated with classes, the meaning is different, because in the program lifetime each class can be instantiated many times and not necessarily at program startup. Moreover, the inheritance of properties must be conveniently defined. Of course, these new kinds of specifications generally have some intuitive meaning, related with the common understanding of OO concepts, which programmers can easily learn. Nevertheless, a formal definition of their semantics is needed. This paper addresses the above problems and proposes a consistent solution in the Java environment.

The paper is organized as follows: Section 2 emphasizes the distinction between interface and implementation code, along with its implication regarding object oriented property specification, Section 3 introduces the formal execution model used as the basis for the semantics of our temporal logic formulae, Section 4 describes the notation used in property specification, Section 5 addresses some problems related to the verification of the specified properties, and Section 7 concludes.

2. Interface and implementation properties

According to the common object oriented understanding, an interface is an abstract specification of functionalities, without going through implementation details. Its purpose is to transfer information from the class developer to the class user. These two roles entail two different viewpoints. The user of an already existing class tends to see the class instances as black boxes that can be accessed via a particular interface and implement a particular functionality in a way that in principle can be ignored. By contrast, the developer of a class works on the class internals and sees how the class functionalities are implemented. In this paper we use the pure OO concept of interface, i.e. the interface of a class is represented by public methods that can be invoked, with their prototypes, whereas the implementation is all the rest. This means that all the attributes are considered encapsulated. Such assumption simplifies our work, but is not a real restriction, because non-encapsulated attributes can always be represented by means of appropriate get and set methods. As an example, let us consider the Java code in Fig. 1. The `C` class represents a possible implementation of the `AbstractContainer` interface. Under the assumption that the interface has been written separately from the class, one might need to specify abstract properties related to the interface functionality, disregarding the way it can be actually achieved. In our example, such a requirement may be that the `read()` method, always when called, returns a positive value. The implementor of the `C` class must ensure that this formula holds for the specific implementation, and may impose a sufficient condition, requiring that, in every object state, `_mod` has a value greater than zero. It can be noted at first sight that the `C` implementation of the interface respects both requirements, the first one referring to the value returned by an abstract method, while the second one involving also a class field, defining an instance variable.

We can now divide the properties that can be expressed about a class into two distinct subsets, according to the point of view under which they are formulated:

1. *interface properties* expressed by the class user point of view and involving only interface elements (i.e., abstract methods);
2. *implementation properties* expressed by the class developer point of view and involving at least an imple-

```
interface AbstractContainer {
  void set(int data);
  int get();
  int update();
  int read();
}
class C implements AbstractContainer {
  int _data;
  int _mod;
  void set(int data)  {_data = data;}
  int get()           {return _data;}
  int update() {
      if  (_data > 0) _mod = _data;
      else _mod = - _data;
  }
  int read()          {return _mod;}
}
```

Fig. 1. Interface and implementation.

mentation element of the class (i.e., encapsulated class attributes and method statements).

A class user who only knows the class interface can only specify interface properties, and expect that every class implementation will satisfy them. A class developer can instead specify both kinds of properties and verify them. In particular, the developer can specify additional implementation properties, for example to express some internal consistency requirements, and can verify that the resulting implementation satisfies all interface and implementation properties before delivering it.

3. Behavioral semantics

As we specify requirements using linear temporal logic (LTL) formulae, we need to define their formal semantics with respect to a sequential model of computation. Formally, such a model can be represented as a labeled transition system $\mathrm{LTS} = \langle \Sigma, S, \rho, s_0 \rangle$ where:

- Σ is the *alphabet* (a finite set of *symbols* representing computation events),
- S is a set of *states*,
- $\rho : S \times \Sigma \mapsto S$ is the *transition mapping* giving, for each state-symbol pair, the next state reached after the occurrence of the corresponding event.
- $s_0 \in S$ is the *initial state*.

Given the alphabet Σ, an *infinite word* is an infinite sequence of symbols of Σ. An execution of the LTS on an infinite word $w = a_0 a_1 \cdots$ is an infinite sequence of states $\pi = s_0 s_1 \cdots$ with the following properties:

- s_0 is the initial state of the LTS,
- $s_i = \rho(s_{i-1}, a_{i-1})$ for every $i \geqslant 1$ that is, every state of the sequence is obtained from the previous one in agreement with the transition mapping.

LTL (Manna and Pnueli, 1992) is a language for reasoning about sequences of states a program goes through during its execution. This language is that of propositional calculus augmented with the following four symbols representing temporal operators. The interested reader is referred to (Manna and Pnueli, 1992) for a formal definition of these operators' semantics.

(1) $\circ$ which is read 'at the next time';
(2) $\square$ which is read 'always in the future';
(3) $\diamond$ which is read 'eventually in the future';
(4) $\mathscr{U}$ which is read 'until'.
(5) $\mathscr{W}$ which is read 'weak until'.

The notation $\pi \models_i A$ is read 'A is true for the execution sequence π starting with its ith state'. One says that a temporal formula A is true for a sequence π, and one writes $\pi \models A$, if $\pi \models_0 A$ (A is true in the initial state of π).

Thanks to the modularity of the object model, when reasoning about the execution of an object-oriented program such as a Java program it is possible to consider the execution of each object separately. The execution of an object can be modeled at two different abstraction levels (i.e., interface and implementation), according to which point of view (class user or developer) is considered. The intuition is that interface properties, being more abstract, could be interpreted only considering sequences of method invocation and method return events. On the other hand, implementation properties involve program variables, whose actual values need to be explicitly represented in the model. Formally, we separate the LTS model needed to describe interface behaviors from the one concerning strict implementation details that is, actual object states. This separation is very important in order to be able to define the semantics of interface properties independently of how interfaces are implemented. In this way, formal reasoning about interface properties is possible even if implementations are not known, according to the object oriented paradigm.

Let us consider first the interface-level execution model. Since the internals of the object are not known to the class users, state information clients can be aware of is represented at most by the sequence of method call and return events that have occurred since the object creation. In other words, a user cannot distinguish two objects in which the same sequence of interface events has taken place, but the two objects could well be in two different states from the developer point of view. Based on this consideration, we define the interface-level state of an object as the ordered sequence of interface-level events that have occurred in its past. Formally, the interface-level execution model is a labeled transition system $\mathrm{LTS}^n = \langle \Sigma^n, S^n, \rho^n, s_0^n \rangle$ where:

- Σ^n is the set of all possible method call and return events (including the constructor call and returns).
- S^n is the set of interface-level states, which includes all the finite sequences of events $s = \langle e_1, e_2, \ldots, e_k \rangle$, such that $e_{1,2,\ldots,k} \in \Sigma^n$. The empty sequence is denoted by ϵ.
- $\rho^n(s, e) = s.e$ where $s.e$ denotes the concatenation of symbol e to sequence s.
- $s_0^n = \epsilon$.

Let us now consider the implementation-level object execution model. It can be defined as $\mathrm{LTS}^m = \langle \Sigma^m, S^m, \rho^m, s_0^m \rangle$ where:

- Σ^m is the set of implementation-level events. In general, they represent computation actions corresponding to the execution of statements, and include also method call and return events.

- S^m is the set of implementation-specific object states; an object state includes the state of each instance variable, and any other state information related to the object.
- $\rho^m(s, e)$, where $s \in S^m$ and $e \in \Sigma^m$ is the new state reached after the occurrence of event e.
- s_0^m is the initial object state, representing the state immediately following the object creation event.

How this kind of model can actually be extracted from the Java source code is outside the scope of this paper. A possible solution is presented for example in (Iosif and Sisto, 2000).

For technical reasons, we introduce a modified version of this LTS, which can be conveniently used as a formal basis for reasoning from the class developer point of view, and for verifying properties. This modified LTS is obtained joining state information of the two previously defined models. In practice, to stress the fact that the implementation-level model is a refinement of the interface-level one, the object state is defined as a pair of state components $s = \langle s_n, s_m \rangle$, where $s_n \in S^n$ coincides with the interface-level state, whereas $s_m \in S^m$ encompasses all the additional state information such as the current state of object attributes and the current state of the method calls that are in progress. Formally, the joint LTS is defined as $\mathrm{LTS}^i = \langle \Sigma^i, S^i, \rho^i, s_0^i \rangle$ where:

- $\Sigma^i = \Sigma^m$ is the set of implementation-level events, which is a superset of interface-level events i.e., $\Sigma^n \subseteq \Sigma^m$.
- $S^i \subseteq S^n \times S^m$ is the set of implementation-level states.
- $\rho^i(\langle s_n, s_m \rangle, e) = \begin{cases} \langle \rho^n(s_n, e), \rho^m(s_m, e) \rangle & \text{if } e \in \Sigma^n, \\ \langle s_n, \rho^m(s_m, e) \rangle & \text{otherwise.} \end{cases}$
- $s_0^i = \langle \epsilon, s_0^m \rangle$.

In LTS^i, a transition is fired either by a method call/return event, in which case both state components change, or by other implementation-level events, with a change in the implementation-level state component only. Fig. 2 shows a graphical representation of an interface-level execution sequence and a corresponding implementation-level sequence, related to the sample Java code in Fig. 1, where call(set,data) represents the event corresponding to the issue of a call of method set with argument data, whereas ret(set) is the event corresponding to a return from method set. It can be noticed how interface-level states are mapped onto sequences of implementation-level states. This correspondence is formally defined in Section 5.

4. Property specification

In this paper we consider only verification of properties associated with the program source code (source

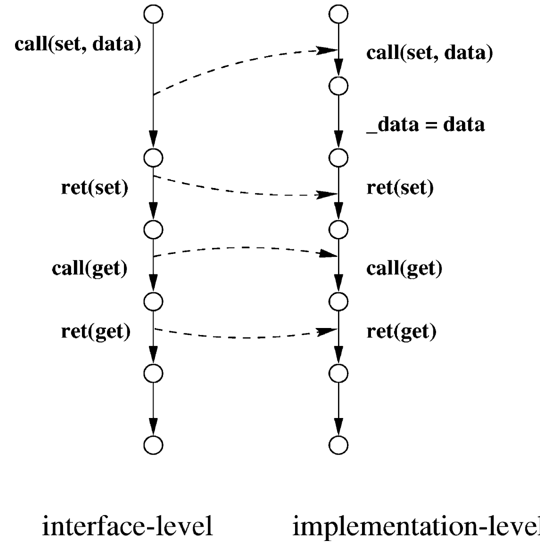

Fig. 2. Sample execution path.

code verification) and not verification of already compiled code (binary code verification). Source code verification can be integrated into the software development process more easily and facilitates the programmer in specifying correctness requirements, since such requirements can be associated directly with the program elements manipulated by the programmer, such as packages, classes and methods.

Before presenting the notation, some general principles that have been followed to define it are illustrated. As interface and implementation properties play different roles, they are treated differently. First of all, implementation properties can only be associated with class implementation definitions (i.e. Java class environments), and not with interface definitions (such as Java interface environments). Moreover, given that each instance of a class that implements an interface can be seen as an instance of the interface itself, and that each instance of a class is also an instance of all its superclasses, it is required that both interface and implementation properties hold in all the classes derived by inheritance or implementation. An equivalent form of the requirement is that interface and implementation properties are inherited by the derived classes. However, following the general assumption that classes derived by inheritance can override implementation details but not interface characteristics, we admit overriding of implementation properties only. In this way, it is guaranteed that inheritance preserves the class interface, along with all the associated properties, and clients can rely on the fact that interface properties are satisfied by all the derived classes. At the same time, the developer of a derived class is free to override not only part of the class implementation, but also some of the implementation properties, the only firm requirement being the preservation of interface properties.

4.1. Syntax and semantics

In order to define the syntax of our property specification language, we first define a set of atomic propositions that are composed into property expressions (formulae). The set of formulae is the core of the specification language. Users can rely directly on it for writing new properties or apply already written patterns (Dwyer et al., 1999) that come as a standard library. Atomic propositions in property formulae can be:

(1) Java boolean expressions; they are evaluated atomically, yielding information which regards the object state.
(2) Special atomic propositions, usually yielding information concerning the flow of control.

Let us now consider special atomic propositions. The specification of interface properties makes use of the following two special atomic propositions:

- `calling(m [, argument_list])`, which is true in all object states where some call to method `m` with actual arguments `argument_list` is being executed (i.e. is pending). Square brackets indicate optionality: if method `m` has no arguments, the argument list is void. In practice, this atomic proposition becomes true whenever a call to method `m` with actual arguments `argument_list` is issued, and it remains true until the corresponding method execution terminates. Of course, in a concurrent environment it is possible to have time-overlapping executions of `m`, in which case the predicate remains true until the current number of concurrent executions of `m` with actual arguments `argument_list` becomes 0.
- `returns(m [, argument_list] [, x])` is true in a certain object state provided that the last interface-level event occurred in the object is a return of value `x` from a call of method `m` with actual arguments `argument_list`. As with the previous predicate, the argument list and the return value can be missing.

Although omitted here for brevity, the above propositions can be formally defined by giving their truth value as a function of the interface-level state defined in Section 3.

This is the minimum core feature to express interface properties. However, to facilitate the task of specifying properties, it is useful to extend the language with some more propositions. For example, in some circumstances it is useful to refer to the number of pending calls to a method `m` with actual arguments `argument_list` in a certain program state, and this is denoted as `#calls(m [, argument_list])`.

Property specifications take the syntax given in Fig. 3. For brevity, we present here only the top-most grammar rules, the rest of them being described informally. The upper-case symbols denote terminals, while the lower-case ones are non-terminal symbols. The symbols enclosed in square braces are optional. Properties can be parameterized with respect to a number of free variables introduced by the `formal_parameter_list` symbol, that takes the same syntactical form of a Java method parameter list. A parameterized property is denoted also as an *open* property. All other properties are denoted also as *closed* properties. Open properties are not meant for actual verification, rather they are introduced as patterns for further specialization or simply for being reused. Indeed, in many instances, specializing already written properties proves to be a useful feature. The substitution of formal parameters with actual arguments in open properties is literal. Closed properties can be quantified over Java types. The `expression` symbol denotes a Java boolean expression used to restrict the quantification domain. An `ltl_formula` is usually obtained from any number of basic propositions connected with the standard LTL operators. In Fig. 3, the

```
closed_property:
 IDENTIFIER '=' [quantifier_expression]
                ltl_formula

open_property:
 IDENTIFIER '(' [formal_parameter_list] ')'
            '=' ltl_formula

quantifier_expression:
 quantifier formal_parameter_list
            ['(' expression ')']

quantifier: one of
 'forall' 'exists'

ltl_formula:
 atomic_proposition
 | property_reference
 | '(' ltl_formula ')'
 | ltl_formula binary_operator ltl_formula
 | unary_operator ltl_formula

binary_operator: one of
 'U' 'W' '->' '<->' '||' '&&'

unary_operator: one of
 '[]' '<>' 'X' '!'
```

Fig. 3. Properties syntactic grammar.

binary_operators U and W denote the temporal $\mathscr{U}$ and $\mathscr{W}$ operators respectively, while the unary_operators [], <> and X denote the temporal operators $\Box$, $\Diamond$ and $\circ$ respectively.

Properties can be referenced by means of their name. The property reference is literally substituted with the referenced property LTL formula. Of course, it is an error to specify mutually recursive properties. Moreover, there is another well-formedness condition that involves referencing quantified properties. The meaning of a quantifier which occurs other than at the beginning of a formula is undefined. The syntax of property specification ensures that this requirement is respected, imposing that the LTL formulas describing properties can only refer to open properties, which as said, cannot be quantified. In this way, a property reference can be used also as an actual argument to specialize an open property. Let us consider the following open property which expresses an overall truth:

```
Always(boolean P) = [](P)
```

It can be specialized with respect to any boolean expression, including a reference to a property:

```
myTruth() = (var == 0) -> <>(var > 0)
myAlways() = Always(myTruth())
```

First, the reference to myTruth() literally substitutes each occurrence of the formal parameter P. Then each reference is literally substituted by its LTL formula. In the end, the meaning is exactly the same as if myAlways() were defined []((var == 0) -> <>(var > 0)).

Open properties parameterized by formulae introduce patterns, used to cover a broad range of requirements for real systems, in terms of parameters that must be filled with descriptions of specific system states or events. These descriptions can be more complex than just a boolean proposition or event e.g., they can be also properties given in terms of LTL formulae. Users can be provided with a specification pattern (Dwyer et al., 1999) library written as a Java interface declaring a collection of open properties. Fig. 4 shows part of such a library. Informally, the GlobalAbsence, BeforeAbsence, AfterAbsence and BetweenAbsence open properties express the absence of an event P overall, before event R, after event Q and between Q and R, respectively. In a similar way, it is even possible for users to define their own specification patterns. An example of library use in coding actual properties is given in the next section.

```
public interface SPL {
/*@
 GlobalAbsence(P) = [](!(P))

 BeforeAbsence(P,R) =
   <>(R) -> (!(P) U (R))

 AfterAbsence(P, Q) = []((Q) -> [](!(P)))

 BetweenAbsence(P,Q,R) =
   [](((Q) && !(R) && <>(R))
   -> (!(P) U (R)))
*/
   ...
}
```

Fig. 4. Specification patterns library.

4.2. Property specification example

The interface-level atomic propositions enable us to define several interface properties of interest. As an example, let us consider the interface of a concurrently accessible integer element stack object:

```
interface IntegerStack {
  void push(int x);
  int pop();
  /*@
    lifo = forall int x, y, z (x != y)
    []((returns(push,x) U (!returns(push,y)
      U returns(pop,z))) -> x == z)
  */
}
```

The interface property lifo informally says that if a push(x) is followed by a pop() with no intermediate other push(y), then the return value of pop() is x. It is a way to specify the LIFO (last in first out) behavior of a stack. A semantically equivalent way to specify this property is using a library pattern property from the collection shown in Fig. 4:

```
lifo = forall int x, y, z (x != y)
[] (BetweenAbsence(returns(push, y),
    returns(push, x),
    returns(pop, z))
  -> x == z)
```

Let us now consider a possible implementation of the Stack interface:

```
class VectorStack implements Stack {
  Vector data = new Vector();
```

```
    int top;
    public synchronized void push(int info) {
      data.add(top++, new Integer(info));
      notifyAll();
    }
    //@ popPre = [] (calling(pop) -> top >= 0)
    public synchronized int pop() {
      while (top == 0)
       try {
        wait();
       } catch (InterruptedException e) {}
      Object info = data.remove(--top);
      return ((Integer) info).intValue();
    }
  }
```

The `lifo` property is automatically inherited by the `VectorStack` class. It can be easily seen that the `VectorStack` implementation respects the `lifo` property because both `push` and `pop` are synchronized, preventing multiple threads to access the stack internal data. As it can be noticed, the code of the `VectorStack` class is also annotated with an implementation property, named `popPre`. This property expresses a pre-condition of the `pop` method, ensuring that, whenever `pop` is called, the instance variable `top` has a positive value. In our case, the implementation of the class meets the requirement, because of the `wait-notifyAll` protocol used in the synchronized `pop` and `push` methods, respectively.

5. Property verification

By *verifying* a property in the context of a given program, we mean deciding if it holds in every execution sequence of the program. For class properties, the decision resumes to proving that the property holds in every execution path of each possible instance of the class.

As said, execution paths are formally described by an LTS. Generally, the decision of LTL formulae in the frame of an LTS is possible algorithmically (Manna and Pnueli, 1992), given that the set of states is finite. This is typically achieved by means of program slicing and abstraction-based specializations (Corbett et al., 2000a,b). Decision of temporal logic formulae in the frame of a finite LTS was made cost effective by the development of model checking techniques (Holzmann, n.d.), i.e. algorithms that attempt to exhaustively explore the state space generated by a specification in order to find counterexamples of the requirements. LTL model checking tools generally do not deal directly with quantified temporal logic formulae, because quantification increases the complexity of verification tasks and, if quantification domains are infinite, verification becomes undecidable. Nevertheless, we decided to introduce quantification in our notation, because it makes the specification of many properties of interest more direct. Of course, to make model checking of quantified formulas possible and viable, it is necessary to have sufficiently small quantification domains, which can be achieved by using abstract representations for quantification variables.

The verification of object properties ideally follows a top-down model. The intuition is that an interface property must be verified for every class that implements the interface. Moreover, a property associated with a class must be verified for every possible instance of the class. In practice, the mechanism whereby interface properties are inherited by class implementations ensures that any interface property is automatically associated also with all the classes that implement the interface. Thus, the verification task regards only properties directly or indirectly associated with classes, that must hold for all their instances. In other words, implementation properties can be seen as implicitly quantified over the domain of all existing class instances. In what follows, we explain the condition under which interface properties can be verified within implementation frames.

As discussed in Section 3, the meaning of an interface property is defined with respect to an interface-level model, denoted as LTS^n, but it should be verified considering the implementation-level object behavior described by a more detailed LTS, denoted as LTS^i. In what follows, we denote by $\mathscr{L}(\mathrm{LTS})$, the language of an LTS that is, the set of all paths it can generate. Since LTS^i was defined as a refinement of LTS^n, it is always possible to extract, from an implementation-level path π^i the corresponding interface-level path. Let $h : \mathscr{L}(\mathrm{LTS}^i) \to \mathscr{L}(\mathrm{LTS}^n)$, be a function defined as follows:

$$h\big(\langle s_k^n, s_k^m\rangle\langle s_{k+1}^n, s_{k+1}^m\rangle\cdots\big) = \begin{cases} h\big(\langle s_{k+1}^n, s_{k+1}^m\rangle\cdots\big) & \text{if } s_k^n = s_{k+1}^n, \\ s_k^n h\big(\langle s_{k+1}^n, s_{k+1}^m\rangle\cdots\big) & \text{otherwise} \end{cases} \quad \forall k \geqslant 0$$

Informally, the h function extracts, from an implementation-level path generated by a program, the corresponding interface-level path, on which we can interpret an interface property. It can be proven that the expression above defines indeed a functional relation on $\mathscr{L}(\mathrm{LTS}^i) \times \mathscr{L}(\mathrm{LTS}^n)$, but we will omit the proof for brevity reasons. Taking into consideration this relation, it is now necessary to show under what conditions the outlined verification procedure for interface properties is sound.

Specifically, soundness requires that if the property holds on all the implementation-level execution paths $\mathscr{L}(\mathrm{LTS}^i)$, then it holds also on the corresponding interface-level paths, which are the image of function h, here denoted as $h(\mathscr{L}(\mathrm{LTS}^i))$. First of all, let us remind that any interface property can be directly interpreted in the frame of LTS^i, because the state of this LTS includes the interface-level state. Of course, this interpretation is based on the fact that any atomic proposition p defined on the interface-level state can also be defined on the implementation-level state in the obvious way:

$$p(\langle s^n, s^m\rangle) = p(s^n) \tag{1}$$

If p is a predicate, and s' is the successor of state s in some execution sequence, state s' is said to be a *p-stuttering* of state s if p has the same truth value in both states. An LTL formula f is said to be *closed under stuttering* when, for every predicate p that occurs in f, its truth value remains the same under state sequences that differ only by p-stuttered states. In the following, we denote the kth element of a sequence σ by σ_k. We can now express the soundness claim:

Theorem 1. *Let ϕ be an interface property. Then*

$$\sigma \models \phi \iff h(\sigma) \models \phi, \quad \forall \sigma \in \mathscr{L}(\mathrm{LTS}^i) \tag{2}$$

holds if ϕ is closed under stuttering.

Proof. Let $\sigma = s_0, s_1, \ldots \in \mathscr{L}(\mathrm{LTS}^i)$. Then for each $k \geqslant 0$, $s_k = \langle s_k^n, s_k^m\rangle$, and for each atomic proposition p that occurs in ϕ we have $p(s_k) = p(s_k^n)$, from (1). For each $k \geqslant 0$, we have $s_{k+1} \in \rho^i(s_k, \tau_k)$. If, for some $k \geqslant 0$, $\tau_k \notin \Sigma$, then we have $s_{k+1} = \langle s_k^n, s_{k+1}^m\rangle$ from the definition of LTS^i, which implies $p(s_{k+1}) = p(s_k)$. Consequently, $h(\sigma)$ differs from σ by at most p-stuttered states. As ϕ was supposed to be closed under stuttering, its truth value over σ remains unchanged over $h(\sigma)$. ∎

This result gives us the decision criterion for interface properties. Since all next-free LTL formulae are closed under stuttering, this limitation preserves a good expressive power, enough to describe many meaningful properties.

6. Related work

The problem of using an object-oriented approach to the formal specification of temporal logic properties to be model checked on object-oriented source code has not been considered so much up to now. Indeed, the main research projects about source-level model checking of object-oriented software (Corbett et al., 2000a,b; Havelund and Skakkebaek, 1999; Young, 1994; Demartini et al., 1999) have focused attention on other problems, such as abstraction and slicing techniques, and have always used classical non-object-oriented techniques to express properties. Instead, object-oriented temporal logic techniques have been proposed for behavioral specification of object-oriented concurrent systems (for example in Denker et al., 1997), which is quite different from source-level property specification.

The first FSV tools for Java that have appeared so far follow the typical approach of considering only properties related to the global program scope. In the current version of the JPF tool (Havelund and Skakkebaek, 1999), properties can be specified in the source code, but with reference to static variables only, whereas the JCAT tool (Demartini et al., 1999) does not provide property specification at all, because it deals with deadlock detection only. Recently Corbett et al. (2000a,b) have proposed BSL (Bandera specification language) to specify the properties that can be verified by their tool (Corbett et al., 2000a,b). This language was designed to cover a broad range of notations including assertions, pre- and post- conditions for methods, and temporal logic specifications, and makes it possible to associate properties with classes and methods. However, the semantics of such notations is given only informally and an underlying formal model to enable mathematical reasoning is not defined. As their notation is intended for specifying properties of complete applications, rather than independent classes, the problem of specifying behavioral properties of interfaces is not addressed, while expressing class properties can be done by explicit quantification over the domain of all existing class instances.

An object-oriented property specification technique in part related to our one has been proposed in the context of C++ (Cline and Lea, 1990) to annotate classes and methods with expected properties. In this case, however, the annotated properties are not in the temporal logic form, and they are not intended for verification by FSV techniques. They are rather assertions to be checked at run time.

7. Conclusions

A formal specification technique has been introduced to specify properties related to object-oriented source code, and particularly concurrent and distributed code, taking as a reference the Java language. Specifications generated according to the presented approach can be used to drive source code verification tools such as the ones already delivered for Ada and Java, but also other kinds of software validation tools.

Specifications use an intuitive and simple notation, well integrated in the source code, which makes it possible to associate properties with specific program modules (classes, interfaces, packages) and not only with whole programs, thus enabling an easy object-oriented specification of properties related to open or component-based systems. In particular, interface properties make it possible to express the expected behavior of interfaces independently of how they will actually be implemented and such properties can soundly be verified on the corresponding implementation level models, provided that next-free formulae are used.

The specification of class properties is inherently object oriented. They annotate classes as a whole, thus avoiding the live-code dead-data notion common in program verification strategies, but antithetical to the object-oriented programming paradigm. Moreover, the use of specification patterns can easily be incorporated in the specification task by means of inheritance and parameterized (open) properties.

In this paper we have presented a 'core language' containing only the essential features. This notation can easily be extended with more elements (e.g. assertions related to implementations, or more kinds of atomic propositions). The practical goal of having a specification language with a formal semantics is to make proofs automatically possible. It is the authors' intention to incorporate the property specification notation presented here in a future version of the JCAT tool (Demartini et al., 1999).

References

Cline, M.P., Lea, D., 1990. The Behavior of C++ classes. In: Proceedings of Symposium on Object Oriented Programming Emphasizing Practical Applications, Marist College.

Corbett, J., Dwyer, M., et al., 2000a. Bandera: extracting finite-state models from Java source code. In: Proceedings of 22nd International Conference on Software Engineering.

Corbett, J., Dwyer, M., et al., 2000b. A language framework for expressing checkable properties of dynamic software. In: Proceedings of 7th SPIN International Workshop on Model Checking of Software.

Demartini, C., Iosif, R., Sisto, R., 1999. A deadlock detection tool for concurrent Java programs. Software: Practice & Experience 29 (7), 577–603.

Denker, G., Ramos, J., et al., 1997. A Linear Temporal Logic Approach to Objects with Transactions Algebraic Methodology and Software Technology. In: Lecture Notes in Computer Science, vol. 1349. Springer-Verlag., pp. 170–184.

Dwyer, M., Avrunin, G., Corbett, J., 1999. Patterns in property specifications for finite-state verification. In: Proceedings of 21st International Conference on Software Engineering. ACM Press, pp. 411–421.

Havelund, K., Skakkebaek, J., 1999. Applying Model Checking in JAVA Verification. In: Lecture Notes in Computer Science, 1680, pp. 216–231.

Holzmann, G., n.d. The model checker SPIN. IEEE Transactions on Software Engineering, SE-23 (5), 279–295.

Iosif, R., Sisto, R., 2000. A formal execution model for Java programs. Technical Report, DAI-ARC, Politecnico di Torino. Available from <http://www.dai-arc.polito.it/dai-arc/manual/papers/jtl.ps>.

Manna, Z., Pnueli, A., 1992. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag.

Young, M., 1994. A concurrency analysis tool suite for Ada. SERC Technical Report TR-1288-P.