Verifying Properties beyond Contracts of SCOOP Programs

Jonathan Ostroff, Faraz Ahmadi Torshizi, and Hai Feng Huang

Department of Computer Science and Engineering, York University,
4700 Keele St.,Toronto, ON M3J 1P3, Canada
{jonathan, faraz, hhuang}@cse.yorku.ca

Abstract. SCOOP and Spec# are programming languages that aim to extend Design by Contract to concurrent and reactive systems. In this paper we discuss how appropriate theorem provers (using Hoare-like verification) can be used to statically check that the contracts are obeyed in concurrent executions, as well as discussing the syntactic and semantic differences between SCOOP and Spec#. We provide a formal model for SCOOP programs as a fair transition system and we use temporal logic for describing system properties beyond contractual correctness. We show that verified contracts provide only a certain measure of correctness, but may not be able to guarantee additional safety and liveness system properties without global reasoning. We show how Microsoft Research's SpecExplorer tool can be used to test SCOOP programs for system properties beyond contracts.

1 Introduction

Concurrent and reactive systems are hard to write and even harder to test. In industrial settings, software verification consists almost entirely of testing. Testing is one of the costliest and most laborious aspects of commercial software development, especially given the lack of systematic engineering methodology, clear semantics and adequate tool support. Concurrency and the need to develop software for reactive systems introduces a level of complexity beyond that of sequential programming. Object-oriented code with dynamic thread creation also introduces additional levels of complexity.

Formal methods using model-checkers and theorem provers have not been considered practical for software applications, but this situation is slowly changing. Until relatively recently, the majority of the work carried out by the formal methods community for proving programs correct has been devoted to special languages that differ from industrial strength programming languages [21]. This is a useful phase as it allows the formal methods community to experiment with new methods.

Recently, more steps have been taken to work with real programs written in modern programming languages. The B-method was used to produce the control system for the Paris driverless metro [3]. In this system, the specification was written and refined from the B specification language into Ada code with all the refinements checked via a theorem prover. Abstract interpretation has been used in [8] to analyze some C programs of up to 100K lines of code, although there are difficulties dealing with rich data structures and dynamic threads. Java PathFinder (JPF) is a verification environment for Java for detecting deadlocks and assertion violations integrating program analysis and model checking [9]. The following quote from [21] is instructive:

> Although it is hard to quantify the exact size of program that JPF can currently handle - "small" programs might have "large" state-spaces - we are routinely analyzing programs in the 1000 to 5000 line range. ... it is naive to believe that model checking will be capable of analyzing programs of 100000 lines or more ...

Undoubtedly, these new methods will be scaled up to handle larger and more realistic examples. Even the ability to analyze small critical chunks of realistic code is a welcome addition to bug detection. Nevertheless, it appears that we will still need to rely on testing for the foreseeable future, with formal verification as a helpful technique for finding additional bugs.

The authors of [4] investigate the use of contracts in object oriented code. The authors state that contracts are known to be a useful technique to specify the precondition and postcondition of operations and class invariants, thus making the definition of object-oriented analysis or design elements more precise. The paper shows how to reuse and instrument contracts to ease testing. A thorough case study is run where they define contracts, instrument them using a commercial tool, and assess the benefits and limitations of doing so to support the isolation of faults. They show that Design by Contract (DbC) has proven to be a powerful lightweight method for documenting contracts in object oriented code as well as for detecting bugs.

The object oriented Eiffel programming language is an industrial strength language with a mature contracting mechanism [13]. ESC/Java [19] shows how to add and check contracts for Java and Spec# is a superset of C# which has a contracting mechanism as well as static verification of contracts [2].

The Simple Concurrent Object-Oriented Programming (SCOOP; hereafter "Scoop") mechanism was proposed as a way to introduce inter-object concurrency into the Eiffel programming language [13]. The mechanism extends the Eiffel language by adding one keyword **separate** that can be applied to entities (attributes and formal routine arguments). If entity `e` is declared **separate** then any call `e.f` is executed in its own thread of control; application of **separate** to entities or arguments indicate that these constructs are points of synchronization.

Part of the Scoop mechanism was implemented by Compton [6] by building upon the GNU SmartEiffel compiler and runtime system, and a Scoop translator using the Eiffel Software compiler was reported in [7]. Scoopli is currently the most up-to-date implementation of Scoop. Using a library approach and the Eiffel Software compiler, code runs as a C or .NET executable [15].

In this paper we will describe the Scoop mechanism via a simple example called *Zero-One* and compare Scoop and Spec# especially with respect to static verification of contracts using theorem provers. We will contrast runtime Assertion Checking versus static Verification, and we will show that contracts can be used to detect certain classes of errors. However, we will also show that there are system properties that contracts alone (without global reasoning) may not detect. We provide an outline of how to convert Scoop code to fair transition systems and we use temporal logic for writing system specifications. We show how to build reduced models and how to use SpecExplorer for testing system properties beyond contracts. The combination of contracts and reduced model testing provide lightweight formal verification that scales up to large systems.

2 Sequential and Concurrent Computation

Object oriented computation (sequential or concurrent) is performed via the mechanism of the feature call $t.r(x)$ to a target t attached to some object obj. A processor invokes the routine call r with argument x to the object obj. In the sequential case, there is only one processor.

In the concurrent case, we have two or more processors. A processor is an autonomous thread of control capable of supporting the sequential execution of instructions for one or more objects. This definition assumes that the processor is some device, which can be implemented either in hardware (e.g. a computer equipped with its own central processor), or as software (e.g. a thread, task or stream). Hence, a processor in this context is an abstraction and we may assume the availability of an unlimited number of processors.

A *subsystem* is a processor together with the set of objects it performs actions on. Within a subsystem, communication is synchronous, and execution follows the usual Eiffel sequential model. Communication between subsystems is asynchronous and processing is in parallel. This potential parallelism is the result of different processors handling each subsystem [6].

A separate object is any object that from the viewpoint of the current object is in a different subsystem. At run time, a separate object can only be referenced (if reachable at all) through a separate entity. An entity is either an attribute of a class, a formal argument of a routine, or a local variable of a routine. A separate reference is a reference to a separate object. This reference must be through a separate entity that is not void, and not attached to a local object. A separate call is any routine call $t.r(x)$, from the current object in which the call is made, where the target t is a separate object. A subsystem is created with the creation of a separate object.

2.1 A Simple Sequential Example

To motivate the main discussion we describe a simple Scoop program – the Zero-One example which uses a sequential class DATA (Fig. 1) written in standard Eiffel. The contracts (preconditions, postconditions and class invariants) document the specification and may also be used to find implementation bugs and demonstrate the correctness of the code.

Correctness of the implementation can be demonstrated either by run-time *Assertion Testing* or by static compile-time *Formal Verification* via the use of a theorem prover. Consider the code in class TEST (Fig. 2) which uses class DATA.

The `create d` instruction (in routine `r`) does a default initialization of all the attributes as shown in the immediately following check statement. Will the feature call `d.one` in the above code succeed without contract violations? The correctness rule for a general feature call $t.r(x)$ is:

$$\frac{\{pre_r \wedge I\}do_r\{post_r \wedge I\}}{\{pre'_r\}t.r(x)\{post'_r\}} \quad \text{[CR1 -- Sequential Correctness Rule]}$$

where pre_r, do_r and $post_r$ are the precondition, body and postcondition of routine r respectively and I is the invariant of the class in which r occurs. The primed notation used in the consequence of rule [CR1] refers to the contracts suitably qualified to the target t. For example, for routine `one` of class DATA, rule CR1 reduces to

$$\frac{\{x = 0 \wedge y = 0\}x := 1;\ y := 1;\ c1 := c1 + 1\{x = 1 \wedge y = 1 \wedge Q\}}{\{d.x = 0 \wedge d.y = 0\}d.one\{d.x = 1 \wedge d.y = 1 \wedge Q'\}}$$

where $Q \overset{\text{def}}{=} c0 = \textbf{old}\ c0 \wedge c1 = \textbf{old}\ c1 + 1 \wedge b = \textbf{old}\ b$ and $Q' \overset{\text{def}}{=} d.c0 = \textbf{old}\ d.c0 \wedge d.c1 = \textbf{old}\ d.c1 + 1 \wedge d.b = \textbf{old}\ d.b$.

In Formal Verification, we can use a theorem prover to check each routine for the verification conditions generated by rule [CR1]. Such a static check guarantees that the code will run correctly without contract violations at runtime. We have implemented such a theorem prover for a significant subset of sequential Eiffel [18, 20]. This theorem prover trivially verifies the correctness of DATA (Fig. 1) and the correctness of the routine `r` in class TEST (Fig. 2). The theorem prover is putatively sound (on the assumption that it is constructed correctly) but not complete. The theorem prover will issue a warning if a verification condition fails to prove with some debugging information as to the source of the problem. The warning could indicate a real bug, but could also mean that the verification condition is true, but that the theorem prover was unable to prove it. Manual intervention would then be required to achieve full certification.

In Assertion Testing, we enable run-time assertion checking and the compiler then generates code that checks the contracts at each feature call (such as `d.one`). Assertion Testing is much weaker than Verification, as [CR1] is only checked for the executions in our testing suite. However, any code of any size can be automatically checked in this manner without the need to provide complete contracts. Testing is thus a successful totally automated lightweight method for documenting and automatically checking specifications.

2.2 A Simple SCOOP Example Using DATA

Classes `ZERO` and `ONE` show some of the main Scoop properties (Fig. 3). For the purposes of this discussion, we assume that a single instance of `ONE` is running

```
class DATA feature
  x,y,c0,c1: INTEGER
  b: BOOLEAN

  zero is
    require x = 1 and y = 1
    do
       x:=0; y:=0; c0 := c0 + 1
    ensure
      x = 0 and y = 0
      c0 = old c0 + 1 and b = old b and c1 = old c1
    end

  one is
    require
      r1: x = 0 and y = 0
    do
       x:=1; y:=1; c1 := c1 + 1
    ensure
      e1: x = 1 and y = 1
      e2: c1 = old c1 + 1 and c0 = old c0 and b = old b
    end

  stop is
    do
      b := true; x := 2
    ensure
      b and x = 2
      y = old y and c0 = old c0 and c1 = old c1
    end
invariant
    inv_data: ((x = 0 and y = 0) or (y = 1 and x = 1)) or b
end -- class DATA
```

Fig. 1. Class DATA

```
class TEST feature
  d: DATA

  r is
    do
      create d
      check
        d.x = 0 and d.y = 0 and d.b = false and d.c1 = 0 and d.c0 = 0
      end
      d.one
    end
end
```

Fig. 2. Class TEST

in a subsystem under the control of processor π_1. Likewise an instance of ZERO is running under the control of processor π_0.

Class ROOT is shown in the listing in Fig. 4. A system execution is initiated when the constructor ROOT.make is called. The constructor creates and initiates the execution of the three subsystems π_0, π_1 and π_d.

In class ONE, attribute data of type DATA is declared **separate**. This means that the object attached to data at runtime runs in its own subsystem (e.g. under the control of processor π_d) and thus under a different processor than the one handling the current object. The responsibility of routine run is to invoke the separate call data.one repeatedly. Scoop requires that such calls be wrapped in a routine such as do_one (see lines 16 and 20).

It is instructive to follow an execution that has arrived at line 16 (which we denote as $\pi_1 = 16$). Control transfers to line 20 where processor π_1 waits to get a lock on the data object under control of π_d. If deadlock does not occur and the lock is obtained (with unique access to data.b), the non-separate precondition $count \leq 1000$ is immediately checked at line 23. A failure generates a precondition exception, and success means that π_1 *waits* for the separate precondition $\neg\ data.b$ at line 22 to become true. It is thus possible for this subsystem to deadlock at line 22 if the condition never becomes true. Assuming the wait condition $\neg\ data.b$ becomes true, execution continues at line 27. An asynchronous feature call data.one is sent to subsystem π_d, and execution continues until 29 where π_1 waits for all asynchronous calls to terminate including the query data.x = 1, at which point the assignment can be executed (this is called wait by necessity [13]).

There is another danger. The asynchronous separate feature call data.one at line 27 may fail when it is finally executed by π_d because the non-separate precondition of DATA.one (i.e. $data.x = 1 \wedge data.y = 1$) may fail to hold (this condition was not checked by the π_1 client prior to the feature call). There are thus a variety of reasons why this Scoop program may fail:

1. Deadlocks may occur at lines 20 [call this failure F1] and 22 [F2].
2. The non-separate precondition may fail at line 23 (a client check is not performed at line 16) [F3].
3. The non-separate precondition of DATA.one may fail at line 27 (or more correctly, the failure will occur when the precondition is checked in subsystem π_d) [F4].

Although we described the execution in terms of acquiring and releasing locks, it is the job of the Scoop compiler to enforce the atomicity described above. The compiler will automatically detect where the Scoop **separate** keyword is missing or inappropriately used, and properly enforce the appropriate behaviour. Thus many race conditions are automatically eliminated. This does not mean that all race conditions are eliminated. The claim that, by using the Scoop model, we eliminate many bugs that come from race conditions, is like the claim that functional languages eliminate side-effect bugs. We may still write code such that the same kind of interference occurs in both cases, but the language leads you naturally away from it.

```
class ONE create
  make
feature
  data: separate DATA
  count: INTEGER

  make(d: separate DATA) is
    do
      data := d
    end

  run is
    do
      from
       until false -- later changed to count > 1000
      loop
        do_one(data)
      end
    end

  do_one(d: separate DATA) is
    require
      separate_pre: not d.b
      non_separate_pre: count <= 1000
    local
      test: BOOLEAN
    do
     d.one
     count := count + 1
     test := d.x = 1
    ensure
      non_separate_post: count = old count + 1
      separate_post: d.x = 1 and d.y = 1 and d.b = old d.b
    end
end -- class ONE
```

Fig. 3. Class ONE (similarly for class ZERO)

```
class ROOT create
    make
feature
  d:  separate DATA
  p0: separate ZERO
  p1: separate ONE

  make is
    do
      create d
      create p0.make(d)
      create p1.make(d)
      run(p0, p1)
    end

  run(z: separate ZERO; o: separate ONE) is
    do
      z.run
      o.run
    end
end
```

Fig. 4. Class ROOT – initiates the three subsystems

3 Detecting Contract Failures

How can we detect deadlocks [F1, F2] and contract failures [F3, F4] as described in the previous section? Due to interference from other subsystems, our formal condition for class correctness must now change to the following [13] (page 1023):

$$\frac{\{pre_S \wedge pre_{NS} \wedge I\}do_r\{post_S \wedge post_{NS} \wedge I\}}{\{pre'_{NS}\}t.r(x)\{post'_{NS}\}} \quad [\text{ CR2 Rule }]$$

where pre_S and $post_S$ are separate pre/postconditions and pre_{NS} and $post_{NS}$ are non-separate pre/post conditions. There is a significant difference between the sequential rule [CR1] and the [CR2] rule. The sequential rule is a full correctness condition – if the antecedent holds, then not only is the call partially correct, but it is also guaranteed to terminate. By contrast, the [CR2] rule only checks partial correctness as it does not incorporate any checks that would catch deadlocks such as [F1] and [F2]. To detect such deadlocks, we need information about other subsystems. We will examine safety properties such as system deadlock detection and liveness properties in a later section.

We may use the [CR2] rule to detect contractual errors such as [F3] and [F4]. Consider first [F3]. If we change our theorem prover to use [CR2] rule instead of [CR1], then we obtain a warning at line 16 (Fig. 3) because we are calling `do_one` without satisfying its precondition $count \leq 1000$ at line 23. Likewise for

[F4], we obtain a warning at line 27 because we are calling `DATA.one` without guaranteeing its precondition.

We can eliminate these warnings from the theorem prover by strengthening the code. [F3] can be fixed by changing the loop guard at line 14 to `(until count > 1000)`. [F4] can be fixed by using a stronger separate precondition at line 22: `not d.b and d.x = 0 and d.y = 0`. This strengthened precondition means that π_1 waits at line 22 until some other processor (e.g π_0) sets the data variables to zero. With these changes, a theorem prover using [CR2] rule will pass without warnings.

We tested the original and revised code in Scoopli. The original code failed with contract exceptions [F3] and [F4], and the revised code passed, thus illustrating Assertion Testing. However, neither Formal Verification nor Assertion Testing were able to guarantee detection of deadlocks such as [F1] and [F2].

4 Comparison of Spec# and Scoop

The Spec# programming system[1] extends C# with contracts (like Eiffel), while it also aims to support concurrency based on object ownership [11]. Spec# extends the type system of C# to include non-null types and checked exceptions. It provides method contracts in the form of pre/postconditions as well as object invariants. The Spec# compiler is integrated into the Microsoft Visual Studio development environment for the .NET platform. The compiler statically enforces non-null types, and emits run-time checks for method contracts and invariants [2].

The Spec# static program verifier (Boogie) generates logical verification conditions from a Spec# program. Internally, it uses an automatic theorem prover (currently *Simplify*) that analyzes the verification conditions to prove the correctness of the program or to find errors in it. Spec# aims to maintain invariants in object-oriented programs in the presence of callbacks, threads, and inter-object relationships [11].

According to [1] there is a problem with the normal rule for invariants especially for concurrent programs. The authors of [1] write that a popular view is that an object invariant is simply a shorthand for a postcondition on every constructor and a pre/postcondition on every public method. The idea behind this view is that an object's invariant should hold whenever the object is publicly visible. This view in itself is appropriate, but is often combined with the following faulty regime. Callers of the methods of a class T do not need to be concerned with establishing the implicit precondition associated with the invariant. For the invariant of a class T to hold at entries to its public methods, it is sufficient to restrict modifications of the invariant to methods of T and for each method in T to establish the invariant as a postcondition. This regime permits a method to violate an object invariant for the duration of the call, as long as it is re-established before returning to the caller. But, unless every method body is

[1] http://research.microsoft.com/specsharp

```
 public sealed class One {
    [LockProtected]
    public Data ! data;
    public int count=0;

    invariant data != null;

    public One([LockProtected] Data d)
       requires d != null;
    {
       data = d;
    }

    public void Run()
        ensures (data.x == 0 && data.y == 0 && !data.b && count <= 1000)
            ==> (count == old(count) + 1);
        ensures (data.x == 0 && data.y == 0 && !data.b && count <= 1000)
            ==> (data.x == 1 && data.y == 1 && data.b == old(data.b));
    {
        while (count <= 1000)
           invariant this.IsExposable;
        {
          expose (this)
          {
              assume data.IsLockProtected;
              acquire (data)
              {
                  if (data.x == 0 && data.y == 0 && !data.b)
                  {
                      assume data.IsPeerConsistent;
                      //Console.WriteLine("1 count: " + count);
                      count++;
                      data.One();
                      assert data.x == 1;
                  }
              }
            }
        }
    }
}
```

Fig. 5. Spec# Code similar to ONE

atomic, this is a problem as illustrated in the paper for a routine that calls itself at a point where the invariant has not yet been re-established. For Spec# the recommendation is made for a construct that declares that the invariant may be temporarily violated (see for example the `expose` construct in Fig. 5).

The problem identified by [1] needs to be examined in the Scoop model. A call to a separate routine is atomic, thus ensuring that no other subsystem will interfere. Second, the Scoop (and Eiffel) model only require the invariant to hold on entry to a *qualified* call (see [13], page 366). There is no such rule for unqualified calls which are not directly executed by clients but only serve as auxiliary tools for carrying out the needs of qualified calls. In such cases it is in order to temporarily violate the invariant provided it is re-established at the end of the routine. We refer the reader to [14] for further discussion.

The Spec# static program verifier uses a theorem prover to prove rules such as [CR1]. The verifier is interesting but still very much in the experimental stage. For example, verification of genericity and inheritance are not yet fully implemented, some primitive types such as reals are not checked, and pure methods in postconditions do not verify. Spec# code (approximately) equivalent to the revised version of class `ONE` (Fig. 3) is shown in Fig. 5. There is not yet much documentation available to enable us to fully evaluate the tool. As far as we were able to determine, there are a number of differences when compared to Scoop.

1. Atomicity is enforced by explicit acquire statements, and various constructs such as sealed assertions and lock protection must be declared.
2. In Spec# all calls are synchronous. Scoop offers a mix of synchrony and asynchrony.
3. To get the theorem prover to work, various assumptions must be added (see `assume` clauses).
4. Preconditions are correctness conditions, not *wait* conditions as in Scoop. This means that wait conditions must be explicitly programmed in via wait constructs.
5. Basic Spec# (without the extensions of [11]) allows object sharing between multiple threads, resulting in potential intra-object concurrency and races that Scoop would prohibit.

Impressively, the Spec# verifier was able to prove the correctness of the contracts and catch incorrect implementations such as those associated with [F3] and [F4]. Also, the ability to statically check for non-null types is significant. However, we still lack full automated capabilities to detect system properties such as complete deadlock detection and liveness properties.

5 SCOOP semantics for contracts

As mentioned earlier, the [CR2] rule, while being correct, is too weak to allow us to argue about separate postconditions. Further, while the Scoop model treats separate preconditions as wait conditions (rather than correctness conditions), we have not explained how invariants and postconditions are treated. Recently, Piotr Nienaltowski and Bertrand Meyer have proposed that postconditions be treated as wait conditions (similar to that of preconditions) and that invariants be disallowed from referring to separate entities [17]. In the sequel we follow the lead of [17] with respect to the intuitions behind postcondition and invariant semantics but provide a temporal logic description of the semantics. Rule 1.5 in [17] is not strong enough to allow for fully compositional proofs of correctness and liveness. This paper will provide a temporal logic version of the semantics based on recent discussions with the authors of [17] and to be reported more fully in [16].

It is convenient to use temporal logic to describe system properties (beyond contracts between a client object and a supplier object). So as to describe the contracting semantics and system properties we provide a schema of how to translate Scoop programs into fair transition systems, which can then be used as the basis for expressing temporal logic system properties. We provide below the main features of a fair transition system in the sense of Manna and Pnueli [12], and we provide a sketch of how to adapt fair transition systems for Scoop programs.

Fair Transition Systems

A fair transition system M is a 5-tuple $M = (V, I, T, J, F)$;

1. The *system variables* V is a finite set of typed variables. The creation of a new subsystem (e.g. `create p1.make(d)` in Fig. 4) corresponds to extending V with a corresponding control variable (e.g. π_1 which is the handler for this instance of class `ONE`). A control variable for a subsystem can be used to indicate which line of code in that subsystem is currently being executed (it is never used in actual program text). A state s of the system is a mapping that assigns to each variable $v \in V$ a value in $\mathit{type}(v)$. The set of all states is denoted by Σ.
2. The *initial condition* I is a boolean valued expression in the variables that characterizes the states at which the execution of the system can begin. A state s satisfying I, i.e. $s \models I$, is called an *initial state.*
3. T is a finite set of transitions. Each transition τ in T is a function $\tau : \Sigma \rightarrow 2^{\Sigma}$ that maps a prestate s in Σ to a (possibly empty) set of τ*-successor* poststates $\tau(s)$ which are obtained when τ is taken. Each state s' in the set $\tau(s)$ is defined to be a τ*-successor* of s. The transition relation $\rho(\mathbf{old}\ V, V)$ describes a set of 2-tuples (consisting of a prestate s and poststate s') that relates the prestate s to its τ-successor $s' \in \tau(s)$, and where $\mathbf{old}\, V$ (by which we mean any of the variables in V) is evaluated in the prestate s, and V is evaluated in the successor state s'.
4. $J \subseteq T$ is a set of *just* transitions. If a just transition $\tau \in J$ is continually enabled it must eventually be taken. Likewise F is a set of fair transitions, i.e. a fair transition that is enabled infinitely often must eventually be taken.

Executions of Fair Transition Systems

An *execution* of a model $M = (V, I, T, J, F)$ is any infinite sequence of states:

$$\sigma = s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} s_2 \xrightarrow{\tau_2} \ldots$$

with $\tau_0, \tau_1, \tau_2 \ldots$ elements of T, so that the following three requirements are satisfied:

1. **Initialization:** The first state of the execution satisfies the initial condition, i.e. $s_0 \models I$.
2. **Succession:** For all positions i in the execution, $s_{i+1} \in \tau_i(s_i)$, i.e. state s_{i+1} is a τ_i-successor of state s_i using the transition relation. This also means that $s_i \models e_{\tau_i}$ where e_{τ_i} is the enabling condition (conjunction of separate and non-separate preconditions and locations) of transition τ_i. We say that τ_i is *taken* at position i in the execution σ, and we may write *taken*(τ_i) to express this.
3. **Justice and Fairness:** For each τ in the justice set, it is not the case that τ is continually enabled beyond some position in the trajectory, but taken at only finitely many positions in the execution. A similar constraint applies for fair transitions.

Scoop code can be converted to a fair transition system using the techniques in [12]. Each construct such as assignments and alternatives are translated into transitions, and the transitions of different subsystems are interleaved with each other, with the fairness constraints removing the non-fair executions. The considerations below may be used to translate the feature call into appropriate transitions.

Postconditions and Invariants

As mentioned earlier, the Scoop model treats preconditions as wait conditions. We now need to consider invariants and postconditions.

There is a major difference between the feature call `Current.do_one(data)` at line 16 and feature call `d.one` at line 27 in Fig. 3.

– At line 16, the argument `data` is in an *unlocked* context because the enclosing routine `run` does not declare `data` as separate.
– At line 27, the feature call `d.one` is in a *locked* context because `d` is locked by the enclosing routine `do_one`. Even if routine `one` would have an argument, e.g. `d.one(x)`, it would be considered as executing in a locked context if both `x` and `d` are declared separate in the formal argument list of `do_one`. The locked case can use the standard sequential [CR1] rule.

Consider, now, the unlocked case at line 16 in Fig. 3, at which point the handler π_1 must execute the routine `do_one(data)` where `data` is an attribute declared as `separate DATA`. This routine "wraps" all accesses to `data` within one call so that no other processor may interfere. Handler π_1 waits to acquire a lock on `data` and for the precondition of the routine `do_one` to become true. Who manages lock acquires and releases and who is responsible for executing the body of the routine `do_one`?

We may assume that the Scoop runtime has a global handler π that manages an action queue for servicing separate calls such as `do_one`. Separate feature calls are queued in the order received, and the global handler guarantees that calls are handled in that order. Provided that all calls can be shown to terminate[2], the global handler guarantees that $\mathit{Acquire}(\texttt{data})$ eventually becomes true at line 16. The global handler is solely responsible for managing all locks on seperate subsystems, granting them and releasing them as required. If, also, the precondition $\mathit{Pre}(\texttt{do\_one})$ subsequently becomes true, then π can mark `do_one` as currently executing and then initiate execution of the body of `do_one`.

In the mean time, handler π_1, having (asynchronously) handed responsibility for the routine call `do_one` off to global handler π, may continue executing at line 17. In the actual example, it just loops back, but in general it could do some local processing before returning to line 16.

Now, routine `do_one` has a postcondition (see lines 31 and 32). When is the postcondition evaluated and by whom? It cannot be evaluated by π_1 immediately after dispatching `do_one` to π because then there would be unnecessary blocking at line 16, which would undermine our attempts at being able to process locally while `do_one` is executed elsewhere.

The logical candidate to choose is handler π (who is anyway responsible for handling `do_one` and lock acquisitions and releases). Handler π checks that `do_one` has completed processing, checks the postcondition, flags any contractual exceptions, and releases the lock on `data`. Consider handler π_1 executing the wrapped routine `do_one` as follows:

```
15:
16: do_one(data)
17:
```

We use our temporal logic framework to describe the behaviour of routine `do_one`.

$$\begin{aligned}
\Box[\,(\pi_1 = 16) \wedge P \;\; &\rightarrow \;\; (\pi_1 = 16)\,\mathcal{U}\,(\pi_1 = 17) \;\wedge\; \Diamond(Q)\,] \qquad (1)\\
P \;&\overset{\text{def}}{=}\; \Diamond[\mathit{Acquire}(\texttt{data}) \wedge \mathit{Pre}(\texttt{do\_one}) \wedge \mathit{Inv}]\\
Q \;&\overset{\text{def}}{=}\; \mathit{Acquire}(\texttt{data})\,\mathcal{U}\,[\mathit{Post}(\texttt{do\_one}) \wedge \mathit{Release}(\texttt{data}) \wedge \mathit{Inv}]
\end{aligned}$$

where $\mathit{Acquire}(\mathit{data})$ and $\mathit{Release}(\mathit{data})$ are functions of the global handler π. *Pre* and *Post* stand for precondition and postcondition respectively of `do_one` and *Inv* is the invariant of class `ONE`. $\Diamond$ is the standard *eventually* operator, $\Box$ the

[2] e.g. using the sequential rule [CR1]. This would require one of the lock passing mechanisms to be implemented so that callbacks do not cause deadlocks.

henceforth operator, and $p\mathcal{U}q$ the *until* operator ("*p until q*" means eventually q, and p holds continuously at least until the first occurrence of q).

As in [17], it is not necessary for a separate postcondition to hold immediately after the execution of the routine's body. The wait semantics applies to postconditions but waiting happens on the supplier side, or more precisely it is managed by the global handler π and the data supplier. The separate target `data` is not released until the postcondition is satisfied.[3]

Note that (1) reduces to a much simpler form for the feature call `d.one` at line 27 because `one` (and any arguments of `one` should there be any) are all already in a locked context. Thus we may drop the lock acquisitions and releases and for such a routine call we may use the standard sequential rule [CR1].

What about invariants? As stated in [17], invariants play an important role in the Design by Contract methodology. They are the primary tool for ensuring the consistency of objects. To prove the correctness of a routine, we assume the invariant before the execution of the body and we must guarantee that it holds again when the body terminates. Rule (1) follows this pattern. Scoop's separate call rule requires that the target of a separate call must appear as formal argument of the enclosing routine. But calls appearing in invariants have no enclosing routines! Therefore, we prohibit the use of separate calls in invariants. Conceptually, we still consider that a violated invariant causes waiting but in practice, since all its clauses only contain non-separate calls, we may reduce the wait semantics to a correctness semantics. As in the case of preconditions and postconditions, the run-time system is able to react to a violated invariant by raising an exception.

Temporal Logic System Specifications

With a fair transition model of Zero-One in place we may document system liveness and safety properties using temporal logic, e.g.

Specification S1: $\Box\Diamond(zero \wedge \bigcirc one)$ where $zero \overset{\text{def}}{=} (\pi_d.x = 0 \wedge \pi_d.y = 0)$ and $one \overset{\text{def}}{=} (\pi_d.x = 1 \wedge \pi_d.y = 1)$. [S1] asserts that we alternate between *zero* and *one* infinitely often. This liveness property is false because the loop only executes 1000 times.

Specification S2: $\Box(\pi_d.x = 0 \wedge \pi_d.y = 0 \quad \vee \quad \pi_d.x = 1 \wedge \pi_d.y = 1)$ – Henceforth, a stronger version of the DATA class invariant holds. This safety property is true because the routine $\pi_d.stop$ is never invoked in Zero-One.

Specifications such as [S1] and [S2] are valid iff they hold in all executions of Zero-One model. The invariant `inv_data` for class DATA (Fig. 1) can be checked either statically by the theorem prover or by run-time assertion checking. However, class ROOT (Fig. 4) never sets in motion any subsystem that triggers

[3] If the postcondition is divided into individual separate clauses, they may be checked and released separately.

routine `DATA.stop`. This is easy to see by inspection in our simple system. However, routine `stop` could occur in much bigger systems or be invoked within one of those hard to read structures such as an alternative within a loop where it is hard to decide whether it actually happens or not. If `stop` never occurs, we should actually be able to prove a stronger system invariant than `inv_data` given by:

$$\Box(x = 0 \land y = 0 \ \lor \ x = 1 \land y = 1) \tag{2}$$

Formal Verification via a theorem prover of the type discussed earlier is unable to prove this stronger property as the [CR1] rule must hold for all routines in DATA (including `stop`).

To prove properties such as S1 and S2, we note that (1) will be needed. (1) is a good rule for explaining the semantics of Scoop to compiler writers. Nevertheless, (1) as a reasoning rule is insufficient for system properties such as S1 and S2. The problem is that the postcondition of the `do_one` routine is not sufficiently projected into the future so that when execution returns to line 16, we can make use of it to argue that eventually the data will be set to zero, which would allow for the precondition of the `one` routine to be re-enabled. This will require global reasoning as discussed in [17, 16].

Testing

If we are allowed to change the code (e.g. by adding new subsystems) then runtime Assertion Testing could be used. We could allow a high priority subsystem to constantly test property (2). However, changing the implementation code is not recommended. What we need is a method to test system properties of concurrent systems without changing the code. How shall we do this?

To check system properties beyond the ones that the theorem prover for contracts can handle, we could rely on model checking and theorem proving techniques for fair transition systems. For example, we could envisage using the SPIN tool [10] or other such efficient state exploration tools.

As discussed in the introduction, formal method tools for software have steadily improved. However, they still do not fully scale up to systems of realistic size, and testing is still required for the foreseeable future. In the next section we discuss testing methods that are able to deal with Scoop programs of arbitrary size.

6 Testing, Reduced Models and SpecExplorer

In this section we show how to test for system properties beyond contracts. Essential to the method is the idea of a reduced model M_r corresponding to an original model M. The reduced model can stand in place of M for certain properties provided that $M_r \sim M$, i.e. M_r is behaviourally equivalent to M on a set of observable variables $\mathcal{O} \subset V_M \cap V_{M_r}$ which is in the intersection of the variables set of M and M_r. The definition of behavioural equivalence is defined in [12] (page 45-47). A stronger relation – congruence of statements – is also provided in [12]. We can strengthen the notion of behavioural equivalence to also include in the observable set not only variables, but transitions (associated with feature calls) as well. Likewise, we could check for behavioural equivalence with respect to properties rather than variables.

If we want to pursue fully formal Verification, we could proceed as follows. Given a full model M for a Scoop program and a temporal logic specification S, automatically construct an appropriate reduced model M_r so that $M_r \sim M$ for the specification S (this construction could be done via abstract interpretation [8] if possible). We may then apply our analysis methods (e.g. model checking) to the reduced model M_r with a greater chance of not running into the problem of combinatorial explosion of states.

As already mentioned, these methods have been used on quite large programs, but there are still problems dealing with data and threads. Until such time that fully formal methods are capable of scaling up to deal with large programs automatically, it is still appropriate to look for Assertion Testing methods for system properties beyond contracts (see previous section).

6.1 SpecExplorer

Model-based testing is one of the most promising approaches for addressing these deficits [5]. In this paper we explore testing Scoop programs via SpecExplorer[4]. SpecExplorer is a software development tool for advanced model-based specification and conformance testing and is now used on a daily basis by Microsoft product groups for testing operating system components and .NET framework components [5]. The description below is taken from [5] and the tool website. The tool can be used to test reactive, object-oriented software systems. The inputs and outputs of such systems can be abstractly viewed as parameterized action labels, that is, as invocations of methods with dynamically created object instances and other complex data structures as parameters and return values. Thus, inputs and outputs are more than just atomic data-type values, like integers.

From the tester's perspective, the system under test is controlled by invoking methods on objects and other runtime values and monitored by observing invocations of other methods. As explained in detail in [5], this is similar to the invocation and call back and event processing metaphors familiar to most programmers. The outputs of reactive systems may be unsolicited, for example, as in the case of event notifications.

The core idea is that the developer encodes the system's intended behaviour (its specification) in machine-executable form (as a "model program"). The model program typically does much less than the implementation; it does just enough to capture the relevant states of the system and shows the constraints that a correct implementation must follow. The goal is to specify from a chosen

[4] http://research.microsoft.com/SpecExplorer

viewpoint what the system must do, what it may do and what it must not do. It can be used to explore the possible runs of the specification-program as a way to systematically generate test suites.

Discrepancies between actual and expected results are called conformance failures and may indicate a variety of problems. An implementation bug is a code defect in the implementation under test. A modeling error is a code defect in the model program itself. A specification error is a mistake or ambiguity in the system's specification (in other words, a misrepresentation of the intended system behaviour). A design error is a logical inconsistency in the system's intended behaviour.

SpecExplorer consists of an explicit-state model explorer, which allows the user to search the (possibly infinite) space of all possible sequences of method invocations that do not violate the pre/postconditions and invariants of the system's contracts and are relevant to a user-specified set of test properties. The tool has a traversal engine, which unwinds the resulting finite state machine to produce behavioural tests that cover all explored transitions. A binding mechanism allows users to associate actions of the model with methods of an implementation written in .NET languages.

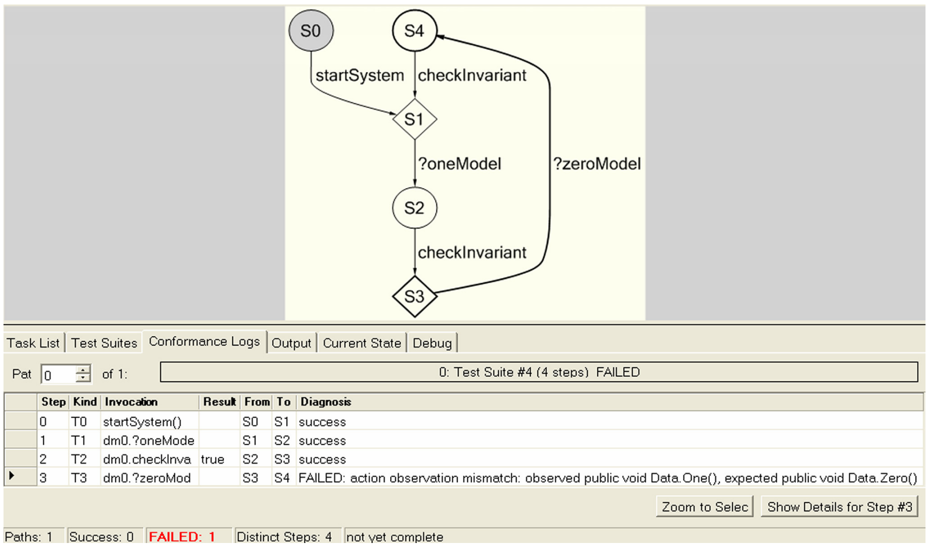
6.2 Method for Testing Scoop Programs

We assume that we are provided with a Scoop program P and a system specification S. The specification does not need to be a formal temporal logic property. It may be a UML style scenario, provided we have a precise idea of what it is. For the sake of concreteness, we let P be Zero-One and the specification S is the strong system invariant [S2] of the previous section.

As stated in [5], reactive systems are inherently nondeterministic. No single agent (component, thread, network node, etc.) controls all state transitions. Network delay, thread scheduling and other external factors can influence the system behaviour. SpecExplorer handles nondeterminism by distinguishing between *controllable* actions invoked by the tester and *observable* actions that are outside of the tester's control.

We use the terms "input" and "output" relative to the system to be tested P. The terms "observable" and "controllable" will be used with respect to the inputs and outputs (respectively) of a model M_P that is used to test P for S. We proceed as follows:

1. We are provided with a Scoop program P and a system specification S. We want to know if P satisfies S, i.e. do all executions of P satisfy S? Since we are dealing with Testing and not Verification, our method will be to run a number of test executions to show that P satisfies S.
2. Use Scoopli to convert P to a .NET component.
3. Use SpecExplorer to manually construct a reduced model M_P of P. The reduced model for Zero-One is shown in Fig. 7. SpecExplorer conveniently and automatically draws the state transition graph shown in Fig. 6. We note that SpecExplorer models are close to the fair transition systems as outlined



0: Test Suite #4 (4 steps) FAILED

	Step	Kind	Invocation	Result	From	To	Diagnosis
	0	T0	startSystem()		S0	S1	success
	1	T1	dm0.?oneMode		S1	S2	success
	2	T2	dm0.checkInva	true	S2	S3	success
▸	3	T3	dm0.?zeroMod		S3	S4	FAILED: action observation mismatch: observed public void Data.One(), expected public void Data.Zero()

```
Paths: 1   Success: 0   FAILED: 1   Distinct Steps: 4   not yet complete
```

Fig. 6. SpecExplorer discovers a bug

in the previous section. The system specifications S1 and S2 of the previous section provide guidance for constructing the reduced model, e.g. that zero and one must alternate (S1), while preserving the invariant (S2).

4. We need to check behavioural equivalence, i.e. in some sense we would like to show that $M_P \sim P$. Although we cannot do this formally, we can test for behavioural equivalence using SpecExplorer's binding mechanism and graph exploration algorithms.
5. For P given by Zero-One, we bind the actions `zeroModel` and `oneModel` in Fig. 7 to the routines `DATA.zero` and `DATA.one` (respectively) as observable actions. We bind `checkInvariant` in the model to the `check_invariant` query in the ROOT class as a controllable action. The addition of this side-effect free query to ROOT is the only change that must be made to P. This query is used to check for the stronger system invariant [S2].
6. Let SpecExplorer automatically generate test cases to explore the model, run the tests and check for conformance.

The model M is written in the SpecExplorer modeling language as shown in Fig. 7 which is very close in concept to the fair transition systems (reduced or full) described in the previous section. After effecting the bindings the tool automatically generates the state exploration graph in Fig. 6. Actions (or transitions) in the model may have pre/postconditions and invariants. In the model, actions may be declared *observable* or *controllable*.

The model actions `zeroModel` and `oneModel` are declared observable and bound to `DATA.zero` and `DATA.one` respectively. This means that these model actions are triggered whenever routines `zero` and `one` occur in the system under test P.

The model action `checkInvariant` is declared controllable and is bound to a new side-effect free query `check_invariant` in ROOT which returns true precisely when the stronger system invariant (2) holds. The round states S2 and S4 in Fig. 6 represent controllable states. When these states are reached, the controllable model action is taken thus triggering the occurrence of the bound routine in P (in this case `checkInvariant`).

```
bool systemStarted;
DataModel sharedData;

public void startSystem() requires systemStarted == false; {
    systemStarted = true;
    sharedData = createData();
}

public DataModel createData() requires systemStarted == true; {
    return (new DataModel());
}

int v = 0;

public class DataModel {
    public DataModel()
    {v = 0;}

    public void zeroModel()
    requires v == 2;
    { v = 3;}

    public void oneModel()
    requires v == 0;
    {v = 1;}

    public bool checkInvariant()
    requires v == 1 || v == 3;
    {
        if (v == 1) v = 2; else v = 0;
        return true;
    }
}
```

Fig. 7. SpecExplorer model

The model describes a system in which `zero` and `one` must alternate. If the preconditions of `zero` and `one` in DATA are removed, then the SpecExplorer model will detect such a failure as shown by the execution to the state FAILED at the bottom of Fig. 6. This is because we expect to observe `one` and instead we saw `zero`. With the preconditions re-inserted and with the revised code fixes to ONE (section 2), the model suitably extended with a timeout will detect that the alternations occur only 1000 times and hence property [S1] of the previous section will be shown not to hold. Obviously, if we had a system that alternated for an infinite amount of time, we would be able to check [S1] for that system only for a limited period. However since [S1] failed in a finite period, the model was able to detect this.

The model is also able to show that the stronger invariant specified by [S2] holds due to the fact that `checkInvariant` is a controllable action which is invoked after each observation of the system under test.

The model can detect the deadlock failures [F1] and [F2] in section 2, which would occur if one of the DATA routines did not terminate hence not releasing the lock or if the `stop` routine is invoked. This is done by activating timeouts in the model. Failures [F3] and [F4] are detected by SpecExplorer because the system under test generates signals that are not expected in the model.

7 Conclusion

Design by contract can be appropriately extended to concurrent languages such as Scoop and Spec#. In this paper, we have compared Scoop and Spec# and shown that a contracting methodology is helpful for detecting bugs in concurrent programs. We showed how theorem provers verify that the contracts are satisfied and can be used to help detect bugs statically at compile time. Assertion Testing at run time can also detect many errors. Scoop, in particular, helps the designer avoid certain classes of race conditions by enforcing atomicity at the level of feature calls.

However, we have also shown that contracts cannot be used to detect many system level properties. We have provided the outline of a method to model Scoop programs as fair transition systems. This also allows us to describe system properties in temporal logic. These models could be used to verify Scoop programs using emerging model checking tools. Nevertheless, we expect these tools to work on systems of moderate size or on small critical components only, for the foreseeable future. Therefore, we also present an Assertion Testing methodology for Scoop programs that will scale up to programs of any size using the SpecExplorer tool.

In future work, we hope to explore better mathematical models for Scoop semantics. We are also working on equipping Scoop with powerful theorem proving tools that can be used statically to verify the contracts. We also hope to investigate the use of abstract interpretation to automatically generate SpecExplorer reduced models that are safe with respect to classes of system properties.

Acknowledgements

We thank Wolfram Schulte, Rustan Leino and Wolfgang Grieskamp of Microsoft Research for help with SpecExplorer and Spec#. We also thank Bart Jacobs for help with Spec# on the newsgroup. We gratefully acknowledge the useful feedback from the referees. This work was conducted under an NSERC Discovery grant.

References

1. Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. Verification of object-oriented programs with invariants.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume LNCS 3362. Springer Verlag, 2004.
3. Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor: A Successful Application of B in a Large Project. volume LNCS 1708, pages 369–387. Springer-Verlag, 1999. Meteor: A Successful Application of B in a Large Project.
4. L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object oriented code. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 70–80, New York, NY, USA, 2002. ACM Press.
5. Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In *Microsoft Research Technical Report (MSR-TR-2005-59)*. 2005. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer.
6. M. Compton. SCOOP: an Investigation of Concurrency in Eiffel. Master's thesis, Department of Computer Science, The Australian National University, 2000.
7. Oleksandr Fuks, Jonathan S. Ostroff, and Richard F. Paige. SECG: The SCOOP-to-Eiffel Code Generator. *JOT Journal of Object Technology*, 11(3), 2004. SECG: The SCOOP-to-Eiffel Code Generator.
8. Arie Gurfinkel, Ou Wei, and Marsha Chechik. Systematic Construction of Abstractions for Model-Checking. VMCAI'06, 2006. Systematic Construction of Abstractions for Model-Checking.
9. K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.
10. Gerard Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
11. Bart Jacobs, K. Rustan M. Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 137–146. IEEE, 2005.
12. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995. Temporal Verification of Reactive Systems: Safety.
13. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

14. Bertrand Meyer. The dependent delegate dilemma. volume 195 of *NATO Science Series, II: Mathematics and Physics and Chemistry*. Springer-Verlag, June 2005.
15. Piotr Nienaltowski. Efficient data race and deadlock prevention in concurrent object-oriented programs. In *Doctoral Symposium, OOPSLA 2004 Companion*, pages 56–57, 2004.
16. Piotr Nienaltowski, B. Meyer, and J.S. Ostroff. Reasoning about concurrent object-oriented programs. 2006 (to be submitted).
17. Piotr Nienaltowski and Bertrand Meyer. Contracts for concurrency. In *First International Symposium on Concurrency, Real-Time and Distribution in Eiffel-like Languages (CORDIE'06)*. Artist2 Workshop at the University of York, UK, 2006.
18. Jonathan S. Ostroff, Chen wei Wang, Eric Kerfoot, and Faraz Ahmadi Torshizi. Automated model-based verification of object-oriented code. Technical Report CS-2006-05, York University, Toronto, 2006.
19. K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. ESC/Java User's Manual. Technical report, 2000. http://research.compaq.com/SRC/esc/papers.html.
20. Faraz Ahmadi Torshizi and Jonathan S. Ostroff. ESpec – a Tool for Agile Development via Early Testable Specifications. Technical Report CS-2006-04, York University, Toronto, 2006.
21. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10, 2003.