Proofs of Procedures

Mark Ardis and Curtis Clifton

April 2010*

1 Introduction

So far we have looked at rules for describing the behavior of simple program statements: assignment, if-then-else, and while loops. Real programs also include procedures and functions, which introduce local scopes and parameter-passing mechanisms. This paper describes the rules we need to reason about procedures. It borrows heavily from a paper by C.A.R. Hoare [1] that extended invariant assertions to handle procedures.

One of the key ideas we need to preserve about procedures is their reuse. That is, just as a programmer only writes a procedure once and then reuses it several times, we want to reason about a procedure definition only once and then reuse that work each time we encounter a call to the procedure. This reuse is sometimes termed *modular reasoning*. The rules we will use are designed to allow such modular reasoning.

2 Rules of Inference

Just as we did with simpler program statements, we define inference rules to use with each type of program construct. For procedures we will have more than one rule, as some simple procedures are easier to reason about than others. We will need rules to handle parameter passing, local variables, and procedures with side effects as well. (We could use one very complex rule for all sorts of procedures, but it seems sensible to use simpler alternative where they suffice.)

2.1 Nearly Useless Procedures

The first rule applies to very basic procedures, those without parameters, side effects, or return values. Here's the rule:

*Based on an original document by Mark Ardis, September 2000

Statement	Rule	Informal Meaning

Procedure without parameters, side effects, or return value	If
	`//@ requires` P;
	`//@ assignable \nothing;`
	`//@ ensures` Q;
	`void p() {` S; `}`
	and
	`//@ assert` P;
	S;
	`//@ assert` Q;
	and S has no side effects except on local variables, then
	`//@ assert` P;
	`p();`
	`//@ assert` Q;
		If a procedure with no parameters, no side effects, and no return value is proven to satisfy its specification, then a call to the procedure has the same pre- and post-conditions as the procedure itself.

The no-parameter rule allows us to reason about simple procedures that take no parameters. It also introduces some new Java Modeling Language (JML) syntax [2].

- The requires clause before the procedure declaration gives the pre-condition for the procedure.
- The assignable clause says what program state may be modified by the procedure (apart from the procedure's local variables). The assignable clause for a procedure is also known as its *frame condition*. A procedure with a frame condition assignable \nothing; is also known as a *pure* procedure. Pure procedures are special as they may be called from within a JML specification.
- The ensures clause gives the post-condition for the procedure.

To use the rule we need to verify two things about the declared procedure p: (i) the body of p must satisfy the procedure's pre- and post-conditions, and (ii) the procedure must have no side effects except perhaps on local variables. After we have shown that just once for the declaration, we can use the pre- and post-conditions of the procedure to reason about any call to the procedure.

For example, suppose we defined a procedure checkQuadrant as follows:

```
//@ requires true;
//@ assignable \nothing;
//@ ensures x >= 0 && y >= 0;
void checkQuadrant() {
    while (x < 0 || y < 0) {
        // infinite loop
    }
}
```

This is a strange implementation, but because the procedure cannot have side effects, we are limited in what we can do. The only way to guarantee that the post-condition holds when we get there, is to prevent getting there in other circumstances!

Despite the strange implementation, we could prove the procedure (partially) correct like this:

```
    //@ assert true;
while (x < 0 || y < 0) {
        //@ assert true && (x < 0 || y < 0);
    // ==> implication holds read down the
page
        //@ assert true;
}
    //@ assert true && x >=0 && y >= 0;
    //@ assert x >= 0 && y >= 0;
```

Now, we could use the pre- and post-conditions where ever procedure checkQuadrant was called:

```
    //@ assert true;
checkQuadrant();
    //@ assert x >= 0 && y >= 0;
```

What good is a procedure with no arguments, no side effects, and no return value? Not much, really. But this rule does serve as a gentle introduction to the pattern that later, more interesting rules will follow.

2.2 Basic Procedures with Parameters

The next rule introduces parameters, but assumes that calls to the procedure will use the same names for actual parameters as were used for formal parameters.

Statement	**Rule**	**Informal Meaning**

Procedure with actuals equal to formals, no side effects, and no return value	If
	`//@ requires` P;
	`//@ assignable \nothing;`
	`//@ ensures` Q;
	`void p(`x_1, ..., x_n`) { `S`; }`
	and
	`//@ assert` P;
	S;
	`//@ assert` Q;
	and S has no side effects except on local variables, then
	`//@ assert` P;
	`p(`x_1, ..., x_n`);`
	`//@ assert` Q;
		Suppose a procedure with no side effects and no return value is proven to satisfy its specification. Then a call to that procedure has the same pre- and post-conditions as the procedure itself, provided the call uses actual arguments that are variable references having the same names as the corresponding formal parameters.

Suppose we had defined our checkQuadrant procedure a little differently:

```
//@ requires true;
//@ assignable \nothing;
//@ ensures x >= 0 && y >= 0;
void checkQuadrant(int x, int y) {
    while (x < 0 || y < 0) {
        // infinite loop
    }
}
```

and we proved it partially correct, using the same proof as above. Then whenever we encountered a call to checkQuadrant whose actual arguments were variable references to x and y, we could reuse the procedure's pre- and post-conditions:

```
    //@ assert true;
checkQuadrant(x, y);
    //@ assert x >= 0 && y >= 0;
```

2.3 Pure Procedures with Arbitrary Arguments

Now we consider calls to side-effect-free procedures that return values. We also expand our rule to allow arbitrary expressions as the arguments.

Statement	**Rule**	**Informal Meaning**

Pure procedure	If
	`//@ requires` P;
	`//@ assignable \nothing;`
	`//@ ensures` Q;
	T p(x_1, ..., x_n) {
	S;
	return r;
	}
	and
	`//@ assert` P;
	S;
	`//@ assert` $Q[r/\backslash\text{result}]$;
	and S has no side effects except on local variables, then
	`//@ assert` $P[\vec{e}/\vec{x}]$;
	v = p(e_1, ..., e_n);
	`//@ assert` $Q[\vec{e}/\vec{x}, v/\backslash\text{result}]$;
		Suppose a procedure with no side effects is proven to satisfy its specification. Then a call to that procedure has the same pre- and post-conditions as the procedure itself, as long as the actual argument expressions are substituted appropriately for the formal parameters in the conditions.

In this rule the notation $P[\vec{e}/\vec{x}]$ means the predicate P with each occurrence of x_1 replaced with e_1, each of occurrence of x_2 replaced with e_2, and so on. The notation $Q[\vec{e}/\vec{x}, v/\backslash\text{result}]$ means the same substitutions in the predicate Q, plus substituting the variable v from the left-hand side of the assignment for any occurrences of the JML keyword \result. An example should help clarify.

Consider this specification of a procedure sum:

```
//@ requires true;
//@ assignable \nothing;
//@ ensures \result == x + y;
int sum(int x, int y) {
    int r = x + y;
    return r;
}
```

The ensures clause in this specification says that the value returned by the procedure is equal to the sum of the parameters x and y.

Using the rule as a template, we begin a proof of the correctness of the procedure like this:

```
1       //@ assert true;
2   int r = x + y;
3       //@ assert r == x + y;
4   return r;
```

where in line 3 the predicate is from the procedure's ensures clause, substituting r for \result.

We complete the proof of correctness like so:

```
    //@ assert true;
    //@ assert x + y == x + y;
int r = x + y;
    //@ assert r == x + y;
return r;
```

Having proved that sum satisfies its specification, we can now use the rule to reason about calls to the procedures. For example, we can prove the following code correct:

```
        //@ assert a == 2;
    b = 3;
    c = sum(a, b);
        //@ assert c == 5;
```

Using the rule we add assertions around the call to sum:

```
1       //@ assert a == 2;
2   b = 3;
3       //@ assert true;
4   c = sum(a, b);
5       //@ assert c == a + b;
6       // ==> Does implication hold?
7       //@ assert c == 5;
```

For line 5 to imply line 7, we need to add a clause to line 5:

```
1       //@ assert a == 2;
2   b = 3;
3       //@ assert true;
4   c = sum(a, b);
5       //@ assert c == a + b && a + b == 5;
6       //@ assert c == 5;
```

Finally, because sum is pure, any assertions about a and b after the call must have been true before the call. So, we can add a + b == 5 to the pre-condition of the call and complete the proof as follows:

```
    //@ assert a == 2;
    //@ assert a + 3 == 5;
b = 3;
    //@ assert a + b == 5;
    //@ assert true && a + b == 5;
c = sum(a, b);
    //@ assert c == a + b && a + b == 5;
    //@ assert c == 5;
```

2.4 Pure Procedures and Invariants

In our example with sum above, we argued that, because the procedure was pure, we could copy predicates between the pre- and post-conditions. The rule below formalizes that reasoning, and accounts for the special case of the variable on the left-hand side of the assignment.

Statement	**Rule**	**Informal Meaning**

Pure procedure with invariant	If
	`//@ requires` P;
	`//@ assignable \nothing;`
	`//@ ensures` Q;
	T p(x_1, ..., x_n) {
	S;
	return r;
	}
	and
	`//@ assert` P;
	S;
	`//@ assert` $Q[r/\backslash\text{result}]$;
	and S has no side effects except on local variables, then
	`//@ assert` $P[\vec{e}/\vec{x}]$ && R_1;
	v = p(e_1, ..., e_n);
	`//@ assert` $Q[\vec{e}/\vec{x}, v/\backslash\text{result}]$ && R_2;
	where R_1 ==> R_2 and R_1 and R_2 do not contain free occurrence of the variable v.
		Suppose a procedure with no side effects is proven to satisfy its specification. Then a call to that procedure has the same pre- and post-conditions as the procedure itself, as long as the actual argument expressions are substituted appropriately for the formal parameters in the conditions.
		
		Any additional predicates that do not refer to the variable on the left-hand side of the assignment are maintained across a call to the procedure ($R_1 = R_2$) or are weakened ($R_1 \Rightarrow R_2$).

Suppose we encountered a call to sum and wished to calculate its weakest pre-condition:

```
    //@ assert ???
b = 17;
c = sum(a, b);
    //@ assert c == 59 && z > 1;
```

We introduce assertions around the call to sum according to the rule:

```
1  b = 17;
2      //@ assert true && R1;
3  c = sum(a, b);
4      //@ assert c == a + b && R2;
5      // ???
6      //@ assert c == 59 && z > 1;
```

Then we manipulate the assertion in line 4 so that it implies the program's post-condition and is in the form specified in the rule. That is, we separate it into parts where c appears and parts where it doesn't.

```
b = 17;
   //@ assert true && R1;
c = sum(a, b);
   //@ assert c == a + b && 59 == a + b && z > 1;
   //@ assert c == a + b && c == 59 && z > 1;
   //@ assert c == 59 && z > 1;
```

Based on the notation in the rule, R_2 is 59 == a + b && z > 1. We can choose R_1 to be the same predicate and complete the calculation as follows:

```
   //@ assert a == 42 && z > 1;
   //@ assert 59 == 17 + a && z > 1;
b = 17;
   //@ assert 59 == a + b && z > 1;
   //@ assert true && 59 == a + b && z > 1;
c = sum(a, b);
   //@ assert c == a + b && 59 == a + b && z > 1;
   //@ assert c == a + b && c == 59 && z > 1;
   //@ assert c == 59 && z > 1;
```

2.5 General Rule for Procedures

We still need to handle procedures that have side effects. To specify such procedures, we need a bit more notation from JML. Here's an example of a procedure with side effcts:

```
//@ requires acc != 0;
//@ assignable v, x;
//@ ensures v == acc + \old(v) && x == v + \old(x);
void accel(double acc) {
   v = acc + v;
   x = v + x;
}
```

The assignable clause in this specification says that the variables v and x may be mutated. (The variables must be declared in a surrounding scope, for example as fields of the class containing the procedure.)

The ensures clause above demonstrates a new JML expression, \old. An \old expression has the value of whatever it wraps interpreted in the *pre-state* of a procedure call. So this ensures clause says that the new value of v is the same as acc plus the original value of v and the new value of x is the same as the new value of v plus the original value of x.

It is legal JML to wrap more complex expressions with \old. For example, if we reversed the order of the statements in the procedure we might have this specification:

```
//@ requires acc != 0;
//@ assignable v, x;
//@ ensures v == acc + \old(v) && x == \old(v + x);
void accel(double acc) {
    x = v + x;       // reversed order from the ...
    v = acc + v;     // ... original method
}
```

The ensures clause now says that the new value of x is the same as v + x evaluated in the pre-state, that is, the original value of v plus the original value of x. The predicate x == \old(v + x) can be written equivalently as x == \old(v) + \old(x). We'll use this more verbose form since it keeps the rules simpler.

We can also specify procedures that mutate their arguments and arrays. Here is a method that adds one to each element of an array:

```
//@ requires true;
//@ assignable a[*];
//@ ensures (\forall int k; 0 <= k && k < a.length; a[k] == \old(a[k] + 1));
void addOne(int[] a) {
    int i = 0;
    while (i < a.length) {
        a[i] = a[i] + 1;
        i = i + 1;
    }
}
```

The assignable clause above says that every element of the argument array a may be mutated. Variants of this notation let us say

- that a single array element may be mutated: assignable a[i]
- that a range of array elements may be mutated: assignable a[lo..hi]

Because of the need to reason simultaneously about values from before and after a procedure call, the general rule introduces additional variables that let us talk about pre-state values in

post-state assertions.

Statement	**Rule**	**Informal Meaning**

General procedure	If
	`//@ requires` P;
	`//@ assignable` $s_1, ..., s_m$;
	`//@ ensures` Q;
	`void p(`x_1, ..., x_n`) {`
	S;
	`return` r;
	`}`
	and
	`//@ assert` P && V;
	S;
	`//@ assert` $Q[r/\backslash\text{result}, \vec{s0}/\backslash\text{old}(\vec{s})]$;
	and S has no side effects except on locations specifed in $s_1, ..., s_m$, then
	`//@ assert` $P[\vec{e}/\vec{x}]$ && V && R_1;
	v = `p(`e_1, ..., e_n`)`;
	`//@ assert`
	`//@` $Q[\vec{e}/\vec{x}, v/\backslash\text{result}, \vec{s0}/\backslash\text{old}(\vec{s})]$
	`//@` && R_2;
	where V is
	$s_1 0 == s_1$ && ... && $s_m 0 == s_m$ and where R_1 ==> R_2 and R_1 and R_2 do not contain free occurrence of the references $s_1, ..., s_m$ or the variable v .
		Suppose a procedure is proven to satisfy its specification. Then a call to that procedure has the same pre- and post-conditions as the procedure itself, as long as the actual argument expressions are substituted appropriately for the formal parameters in the conditions.
		
		To deal with mutated locations, the pre-condition is augmented with a predicate V that introduces new variables to "store" the pre-state values. Any \old expressions in the post-condition are replaced with the appropriate new variable.
		
		Any additional predicates that do not refer to the variable on the left-hand side of the assignment or to any assignable locations are either maintained across a call to the procedure ($R_1 = R_2$) or are weakened ($R_1 \Rightarrow R_2$).

Here's an example using our accel procedure from above, where we've rewritten the post-condition to push the \old expressions down to the variable references:

```
//@ requires acc != 0;
//@ assignable v, x;
//@ ensures v == acc + \old(v) && x == \old(v) + \old(x);
void accel(double acc) {
    x = v + x;
    v = acc + v;
}
```

First we have to prove that the procedure is correct using the rule:

```
1     //@ assert acc != 0 && v0 == v && x0 == x;
2  x = v + x;
3  v = acc + v;
4     //@ assert v == acc + v0 && x == v0 + x0;
```

In line 1 above, the boldface clauses represent V from the rule. In line 4, the \old references are replaced with the appropriate pre-state variables v0 or x0. Now it's just a matter of using the rules for assignment:

```
    //@ assert acc != 0 && v0 == v && x0 == x;
    //@ assert acc + v == acc + v && v + x == v + x && v0 == v && x0 == x;
    //@ assert acc + v == acc + v0 && v + x == v0 + x0;
x = v + x;
    //@ assert acc + v == acc + v0 && x == v0 + x0;
v = acc + v;
    //@ assert v == acc + v0 && x == v0 + x0;
```

We also verify that no locations are mutated besides x and v.

Now we can use the specification of accel to reason about calls to it. Consider the following example where we wish to find the weakest pre-condition:

```
    //@ assert ???
g = -10;
accel(g);
    //@ assert x == 0 && v == 20;
```

First we annotate the call to accel using the rule and the procedure's specification. Notice the introduction of the pre-state variables v0 and x0.

```
1      //@ assert ???
2  g = −10;
3      //@ assert g != 0 && v0 == v && x0 == x;
4  accel(g);
5      //@ assert v == g + v0 && x == v0 + x0;
6      // ???
7      //@ assert x == 0 && v == 20;
```

In order for line 5 to imply line 7, we need to strenghthen the condition in line 5:

```
1      //@ assert ???
2  g = −10;
3      //@ assert g != 0 && v0 == v && x0 == x;
4  accel(g);
5      //@ assert v == g + v0 && x == v0 + x0 && x == 0 && v == 20;
6      //@ assert x == 0 && v == 20;
```

Next we need to move the constraints on the pre-state that appear in line 5 to the pre-state of the call. To do that we need to eliminate references to the post-state values from the clauses that refer to the pre-state values. We do that by plugging in 0 for x and 20 for v as allowed by the last two clauses in line 5:

```
1      //@ assert ???
2  g = −10;
3      //@ assert g != 0 && v0 == v && x0 == x;
4  accel(g);
5      //@ assert 20 == g + v0 && 0 == v0 + x0 && x == 0 && v == 20;
6      //@ assert x == 0 && v == 20;
```

Now we can augment the pre-state of the call with the clauses from the post-state that don't reference any mutated locations (i.e., 20 == g + v0 && 0 == v0 + x0).

```
1      //@ assert ???
2  g = −10;
3      //@ assert g != 0 && v0 == v && x0 == x && 20 == g + v0 && 0 == v0 + x0;
4  accel(g);
5      //@ assert 20 == g + v0 && 0 == v0 + x0 && x == 0 && v == 20;
6      //@ assert x == 0 && v == 20;
```

Finally, we back up from line 3 by eliminating the introduced variables and using our usual rule for assignment:

```
    //@ assert 30 == v && −30 == x;
    //@ assert 30 == v && 0 == 30 + x;
    //@ assert true && 30 == v && 0 == v + x;
    //@ assert −10 != 0                    && 20 == −10 + v && 0 == v + x;
g = −10;
    //@ assert g != 0                      && 20 == g + v && 0 == v + x;
    //@ assert g != 0 && v0 == v && x0 == x && 20 == g + v0 && 0 == v0 + x0;
accel(g);
    //@ assert 20 == g + v0 && 0 == v0 + x0 && x == 0 && v == 20;
    //@ assert x == 0 && v == 20;
```

2.5.1 Caveat

We're playing a bit loose with the JML syntax in our introduction of pre-state variables. JML, like its Java host language, is statically typed and requires declarations of all variables. Our pre-state variables, like x0 and v0 in the accel example, would have to be declared. Such *specification-only* variables, which do not appear in the actual code, can be introduced using "ghost" fields in JML. For manual proofs of correctness we prefer to omit this complication.

2.6 Invalid Substitutions

Thus far we've been ignoring some potential problems. We've assumed that substitution of names does not cause any conflicts. This section describes some things that can go wrong with the substitutions. We have to watch out for these problems so we can avoid them, typically by renaming variables in our proofs.

The first case where substitution of parameters causes problems is aliasing. Aliasing occurs when two names exist for the same program variable. This causes problems when one of those references is updated. For example, suppose we attempted the following call to sum:

```
a = sum(a, b);
```

If we blindly apply our rules for substitution might conclude that b must be zero:

```
    //@ assert true;
a = sum(a, b);
    //@ assert a == a + b;
```

From our knowledge of the program we know that a and b can initially have any integer values, so there must be a problem with our substitution. The trouble is that the a on the left-hand

side of the post-condition refers to the post-state, but the a on the right-hand side refers to the pre-state. Since sum was declared to be pure, we didn't worry about introducing pre-state variables, so we lost this distinction. We eliminate the problem by stating a rule about parameter substitutions:

Rule	**Informal Meaning**
No aliasing: In a procedure call v = p(e_1, ..., e_n); neither the variable v nor any of the locations assignable by p may appear in $e_1, ..., e_n$.	This rule prevents aliases to mutable locations from appearing in a procedure call.

This rule is not as restrictive as it seems, since we can always introduce new variables to the program for reasoning purposes. For example, we could rewrite our call to sum as:

```
    //@ assert true;
int a_old = a;
    //@ assert true;
a = sum(a_old, b);
    //@ assert a == a_old + b;
```

The no-aliasing rule also prevents problems when mutatable locations are used in the pre- and post-conditions and as actual arguments. For example, suppose we called our accel procedure passing v as an argument:

```
accel(v);
```

We would attempt to reason about this call by introducing pre- and post-conditions like so:

```
    //@ v != 0 && v0 == v && x0 == x;
accel(v);
    //@ v == v + v0 && x == v0 + x0;
```

From the post-condition we (wrongly) conclude that v0 is zero and from the pre-condition we conclude that v0 is not zero! The no-aliasing rule saves us from this falicy. The assignable clause for accel says that v is mutable by accel and disallows reasoning about the call accel(v). Instead we rewrite the call as:

```
v_old = v;
    //@ v_old != 0 && v0 == v && x0 == x;
accel(v_old);
    //@ v == v_old + v0 && x == v0 + x0;
```

3 Recursive Procedures

The final topic to consider is recursive procedures. The rules we have introduced so far require that something be proved about the body of a procedure before anything can be said about calls to that procedure. Unfortunately, recursive procedures have calls to themselves within their bodies.

The solution is to do a proof by induction. First, prove that a pair of pre- and post-conditions correctly describe the base cases. Then assume that the pre- and post-conditions apply to a call of the procedure. Using that assumption, show that they apply to the body of the procedure. The base cases turn out to be the paths through the procedure that do not involve any recursive calls. There has to be at least one, or the procedure would never terminate.

Here is the rule that describes this method:

Rule	**Informal Meaning**

Given a recursive procedure p:
`//@ requires` P;
`//@ assignable` $s_1, ..., s_m$;
`//@ ensures` Q;
`void p(`x_1, ..., x_n`) {`
S;
`return` r;
`}`
p satisfies its specification if we can:
1. assume the specification holds and prove
`//@ assert` P `&&` V;
S;
`//@ assert` $Q[r/\backslash\text{result}, \vec{s}0/\backslash\text{old}(\vec{s})]$;
and S has no side effects except on locations specifed in $s_1, ..., s_m$;
2. and demonstrate that there is a path through S that does not involve a recursive call.
	Our rules for if-then-else and while statements combine to make the proof in step 1 constitute a proof for both the base case and the inductive case of an induction proof. Step 2 ensures that there is a base case.

At first glance this rule appears to violate common sense. How can you assume that something is true in order to prove that it is? The answer is that part of the proof does not depend on the assumption—the paths through the body that do not include recursive procedure calls.

Here is a simple recursive procedure that computes the product of two numbers:

```
//@ requires b >= 0;
//@ assignable \nothing;
//@ ensures \result == a * b;
int mult(a, b) {
    int r;
    if (b > 0) {
        r = mult(a, b - 1);
        r = r + a;
    } else {
        r = 0;
    }
    return r;
}
```

Here's a proof of correctness for the body of the procedure. Not the assertions before and after the recursive call in boldface:

```
    //@ assert b >= 0;
int r;
if (b > 0) {
        //@ assert b >= 0 && b > 0;
        //@ assert b >= 1;
        //@ assert b - 1 >= 0;
    r = mult(a, b - 1);
        //@ assert r == a * (b - 1);
        //@ assert r == a * b - a;
        //@ assert r + a == a * b;
    r = r + a;
        //@ assert r == a * b;
} else {
        //@ assert b >= 0 && b <= 0;
        //@ assert b == 0;
        //@ assert 0 == b;
        //@ assert a * b == a * 0;
        //@ assert 0 == a * b;
    r = 0;
        //@ assert r == a * b;
}
    //@ assert r == a * b;
return r;
```

3.1 Termination

How can you show that a recursive procedure always terminates? You must show that each recursive call passes a different set of parameters than the original call, and that the parameters will eventually reach a value where no recursive call is made. For the example above we see that the second parameter, b, is strictly decreasing, and it will eventually cause the "else" branch to be taken, with no recursive calls.

References

[1] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathe-*

matics, pages 102–116, 1971.

[2] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. JML Reference Manual. Available from http://www.jmlspecs.org, May 2008.