Web-crawling Testing using TTCN-3

Bernard Stepien

University of Ottawa

bernard@site.uottawa.ca
Recursive hyperlink checking

web-crawling

• Principle: start with the root web page, test each link found on this page and in turn test each link found on the page a given link is pointing to and so on.

• Appropriate for generated web sites like for example on-line newspapers

• Is a kind of self testing procedure

• Risk of state explosion, thus needs constraints.
web application FSM

Web page 1

Web page 2.1

Web page 2.2

Web page 3
Recursive hyperlink checking
TTCN-3

Solution: Test case invokes a recursive function and appropriate list type definitions.

```tcc

type record of charstring HrefListType;

type record hyperlink_response_type {
    charstring status,
    HrefListType hrefList
}

template hyperlink_response_type hyperlink_response :={
    status := "HTTP/1.1 200 OK",
    hrefList := ?
}

testcase Link_Check_TC() runs on MTCTYPE system SystemType {
    ...
    theOverallVerdict := CheckChildLink(main_page_web_url, 1);
    setverdict(theOverallVerdict);
}
```
function CheckChildLink(charstring theURLLink)
runs on MTCType return verdicttype {

    web_port.send(theURLLink);
    web_port.receive(hyperlink_response) -> value theHyperlink_response {

        for(i:=0; i < numOfLinks; i:=i+1) {
            oneVerdict := CheckChildLink(theResponseHrefList[i]);
            if(oneVerdict == fail) { theFinalVerdict := fail; }
        }

    }

    return theFinalVerdict;
}

Problem: this design cycles and the test would never stop
Solutions to recursive cycles

• Limit the recursion depth
• Limit the domain to avoid testing the world

• Allow some links outside of the domain
• Handle load-balancing domain diversification
Recursive hyperlink checking
maximum depth

```plaintext
function CheckChildLink(charstring theURLLink, integer theDepth) runs on MTCTType return verdicttype {
    web_port.send(theURLLink);
    web_port.receive(hyperlink_response) -> value theHyperlink_response {
        ...
        if (theDepth <= maxDepth) {
            ...
            theNewDepth := theDepth + 1;

            for(i:=0; i < numOfLinks; i:=i+1) {
                oneVerdict := CheckChildLink(theResponseHrefList[i], theNewDepth);
                if (oneVerdict == fail) { theFinalVerdict := fail; }
            }
        }
    }
    return theFinalVerdict;
}
```
Maximum recursion depth approach evaluation summary

- avoids cycles
- does not ensure that the entire web application has been tested if maximum depth is unknown
- does not necessarily avoid cycles when cycle depth < maximum recursion depth
- redundancy of testing links to the same target page
- tests web pages that are external to the web site of the application's domain (google)
- does not ensure that all specified links exist. Only these found on pages are checked (discovery mode)
Recursive tests with cycle detection

• avoiding cycles completely
• does ensure that the entire web application discovered links have been tested
• no redundancy of testing links to the same target page
TTCN-3 solution for cycle detection

```tcl
type record length (0 .. infinity) of charstring VisitedUrls;

type component MTCType {
    port web web_port;
    var VisitedUrls theVisitedLinks;
}

function UpdateVisitedLinks(charstring theURLLink) runs on MTCType {
    var integer numOfLinks := sizeof(theVisitedLinks);
    theVisitedLinks[numOfLinks] := theURLLink;
}

function IsNotInVisitedLinks(charstring theURLLink) runs on MTCType return boolean {
    var integer numOfLinks := sizeof(theVisitedLinks);
    var integer i;
    for(i:=0; i < numOfLinks; i:=i+1) {
        if(theVisitedLinks[i] == theURLLink) {
            return false;
        }
    }
    return true;
}
```
TTCN-3 solution for cycle detection

Test case modifications

testcase RecursiveLink_Check_TC(charstring theURLLink)
   runs on MTCType system SystemType {
      var verdicttype theOverallVerdict;

      theVisitedLinks := {};

      map(mtc:web_port, system:system_web_port);

      theOverallVerdict := CheckChildLink(theURLLink);

      setverdict(theOverallVerdict);
   }
function CheckChildLink(charstring theURLLink) runs on MTCType

return verdicttype {

... for(i:=0; i < numOfLinks; i:=i+1) {
    if(NotInVisitedLinks(theResponseHrefList[i])) {
        oneVerdict := CheckChildLink(theResponseHrefList[i]);
        if(oneVerdict == fail) {
            theFinalVerdict := fail;
        }
    }
    else {
        log("detected recursion to page: " & theResponseHrefList[i]);
    }
}
...
}
Cycle avoidance approach evaluation summary

- tests web pages that are outside of the application's domain also recursively.
- Risk of testing the world.
- does not ensure that all specified links exist. Only these found on pages are checked.
Recursive link checking with domain restrictions

- Does not test web pages that are outside of the application's domain:
  - Web pages that are outside an enterprise’s web application.
  - Within an enterprise, ensure that a sub-application does not have links to another sub-application. Important with language issues. French version should not point to English version and vice versa.
Solution for domain restrictions
Test case modifications

testcase RecursiveDomainLink_Check_TC(charstring theURLLink, charstring theDomain) runs on MTCType system SystemType {
    var verdicttype theOverallVerdict;

    theVisitedLinks := {}; 

    map(mtc:web_port, system:system_web_port);

    theOverallVerdict := CheckChildDomainLink(theURLLink, theDomain);

    setverdict(theOverallVerdict);
}
Solution for domain restrictions

function modifications

function CheckChildDomainLink(charstring theURLLink, charstring theDomain)
runs on MTCType return verdicttype {
    ...
    for(i:=0; i < numOfLinks; i:=i+1) {
        if(IsNotInVisitedLinks(theResponseHrefList[i])) {
            if(IsInDomain(theResponseHrefList[i], theDomain)) {
                oneVerdict := CheckChildDomainLink(theResponseHrefList[i], theDomain);
                if(oneVerdict == fail) {
                    theFinalVerdict := fail;
                }
            } else {
                log("URL: " & theResponseHrefList[i] & " IS NOT IN DOMAIN: ");
                theFinalVerdict := fail; // optional depending on kind of test
            }
        } else {
            log("DETECTED RECURSIVE LINK: " & theResponseHrefList[i]);
        }
    }
    ...
}
Domain checking approaches

• Using functions
• Using templates with the TTCN-3 matching mechanism
Solution for domain restrictions

function approach

```plaintext
function IsInDomain(charstring theURL, charstring theDomain) return boolean {
  var integer theURLLength := sizeof(theURL);
  var integer theDomainLength := sizeof(theDomain);
  var integer i;

  for(i:=0; i < (theURLLength - theDomainLength); i:=i+1) {
    if(substr(theURL, i, theDomainLength) == theDomain) {
      return true
    }
  }

  return false;
}
```
Solution for domain restrictions

template approach

- So far URLs have been defined as simple strings
- Filtering URLs using strings required convoluted pattern matching routines
- A better way to filtering is by breaking down a URL in its components
- Filtering can be achieved with various templates on each URL component
URL data typing example

type record URLtype {
    charstring protocol,
    charstring host,
    charstring path,
    charstring fileName,
    charstring fileExtension
}

template URLtype isOnEtsiDomain := {
    protocol := “http://”,
    path := ?,
    fileName := ?,
    fileExtension := ?
}
function CheckChildDomainLink(charstring theURLLink, charstring theDomain)
    runs on MTCType return verdicttype {
        ...
        for(i:=0; i < numOfLinks; i:=i+1) {
            if(IsNotInVisitedLinks(theResponseHrefList[i])) {
                if(match(theResponseHrefList[i], theDomain)) {
                    oneVerdict := CheckChildDomainLink(theResponseHrefList[i], theDomain);
                    if(oneVerdict == fail) {
                        theFinalVerdict := fail;
                    }
                }
            }
        }
        ...
    }

    CheckChildDomainLink(myLink, isOnEtsiDomain);
Domain restriction approach evaluation summary

- does not ensure that all specified links exist. Only these found on pages are checked
TTCN-3 Missing links detection

• A self-testing approach that discovers links as it recurses through a web applications can not know which links are missing.
• Usually, an enterprise knows what web pages it should have.
• Solution is to provide a list of web pages that should exist.
• TTCN-3 solution is based on set matching
Solution for missing links detection
Data type/templates modifications

type set of charstring HrefSetType;

type set of charstring VisitedUrls;

template HrefSetType theSetOfWebPages := {
    "file:web_page_1.html", "file:web_page_2_1.html",
    "file:web_page_2_2.html", "file:web_page_3.html"
    //, "file:some_isolated_page.html"
}
Solution for missing links detection

Test case modifications

testcase RecursiveLink_Check_TC(charstring theURLLink) runs on MTCType system SystemType {
    var verdicttype theOverallVerdict;

    theVisitedLinks := {};
    ...
    theOverallVerdict := CheckChildLink(theURLLink); // recursive checking

    if(theOverallVerdict == fail) {
        setverdict(inconc)
    } else {
        if(match(theSetOfWebPages, theVisitedLinks)) {
            setverdict(pass)
        } else {
            log("failed to match theSetOfWebPages: ");
            listSetContent(theSetOfWebPages);
            log("vs theVisitedLinks");
            listSetContent(theVisitedLinks);
            setverdict(fail)
        }
    }
}
Missing links testing summary

• Even if the missing links test passes this does not mean that the navigation structure of the web application is correct

• You may want to check if the sequence of pages is correct
Recursive testing with link paths navigation test

- It is a different kind of recursive checking
  - No longer a discovery principle
  - More a list traversal principle
- verifies navigation paths including these that go through the same page several times
- avoiding cycles is no longer an issue, going through the same page may even be a requirement
- redundancy of testing links to the same target page is allowed and even required
Link path navigation test
types and templates

template HrefListType links_path_1 := {
    "file:web_page_1.html", "file:web_page_2_2.html",
    "file:web_page_1.html"
}
Link path navigation test
Test case modifications

testcase RecursivePath_TC(HrefListType theURLLinkPath) runs on MTCType
system SystemType {

    var verdicttype theOverallVerdict;
    NumPages := sizeof(theURLLinkPath);

    log("testing PATH of " & int2str(NumPages) & " Elements");
    map(mtc:web_port, system:system_web_port);

    theOverallVerdict := CheckChildLink(theURLLinkPath, 0);

    setverdict(theOverallVerdict);
}
function CheckChildLink(HrefListType theURLLinkPath, integer position) … {
    …
    web_port.send(theURLLinkPath[position]);
    …
    web_port.receive(hyperlink_response) -> value theHyperlink_response {
        …
        if(position+1 == NumPages) {
            return theFinalVerdict;
        }
    }
    oneVerdict := CheckChildLink(theURLLinkPath, position + 1);
    if(oneVerdict == fail) {
        theFinalVerdict := fail;
    }
    …
}
conclusions

• TTCN-3 is a very flexible language
• TTCN-3 allows the creation of any custom test
• TTCN-3 matching mechanism saves substantial amounts of coding
• TTCN-3 parametrization allows maximum code re-usability