Formal absence of implementation bugs in web applications:
A case study on indirect data sharing

Lieven Desmet
DistriNet Research Group
Katholieke Universiteit Leuven
Lieven.Desmet@cs.kuleuven.be
+32 16 32 79 53

OWASP
BeLux Chapter
May 10th,2007

The OWASP Foundation
http://www.owasp.org

Overview

- Introduction
- Problem statement
- Static verification of indirect data sharing
- Static and dynamic verification
- Conclusion

Overview

- Introduction
- Problem statement
- Static verification of indirect data sharing
- Static and dynamic verification
- Conclusion

Background

- DistriNet Research group (K.U.Leuven)
  - Software engineering group with focus on distributed software applications
  - Large taskforce on software security (+- 25p)
    - Identity management and privacy
    - Security at the language level
    - Security at the application and middleware level
    - Secure software engineering processes
- Try to find a balance between:
  - Basic and applied research
  - Practical hands-on

Background (2)

- Research on applying formal techniques in (web) application security
  - Concurrency control & deadlock prevention
  - Code Access Security
  - Buffer overflow protection
  - Indirect data sharing
  - ...
- *"We try to improve software security by a.o. improving the reliability of the software system"*

Formal verification in web applications research

- Protection against injection attacks and XSS
  - Run-time tainting
    - Pietraszek and Vanden Berghe (2005), Nguyen-Tuong et al. (2005), Halder et al. (2005), ...
  - Static analysis
    - Livshits and Lam (2005), Jovanovic et al. (2005)
  - Combination of static information flow analysis and run-time guards:
    - Huang et al. (2004)
- Firewall configuration analysis
  - Consistency between different firewalls and IDS configurations
    - Uribe and Cheung (2004)
  - Rule consistency and reduction
    - Golnabi et al. (2006)

Interesting overview: *http://suif.stanford.edu/~livshits/work/griffin/lit-topic.html*

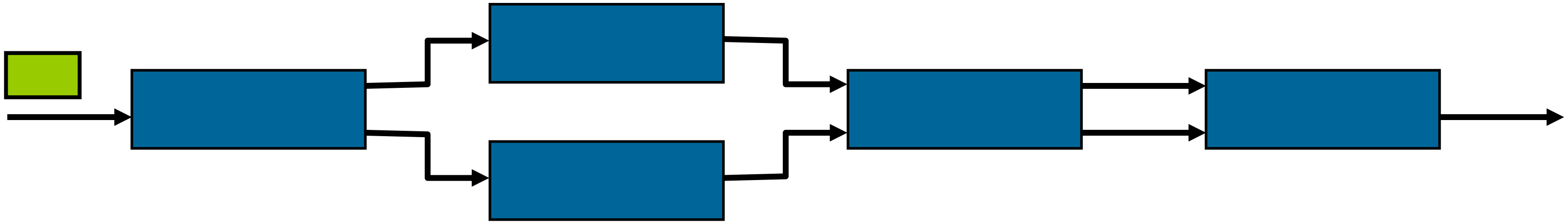
Context of this presentation

- Modern software systems:
  - Quite complex
  - Composed of reusable components

- Common architectural patterns to achieve loose coupling:
  - Pipe-and-filter style
  - Data-centered style

Pipe-and-filter style

- The software is composed as a chain of components (filters), connected to each other by means of pipes

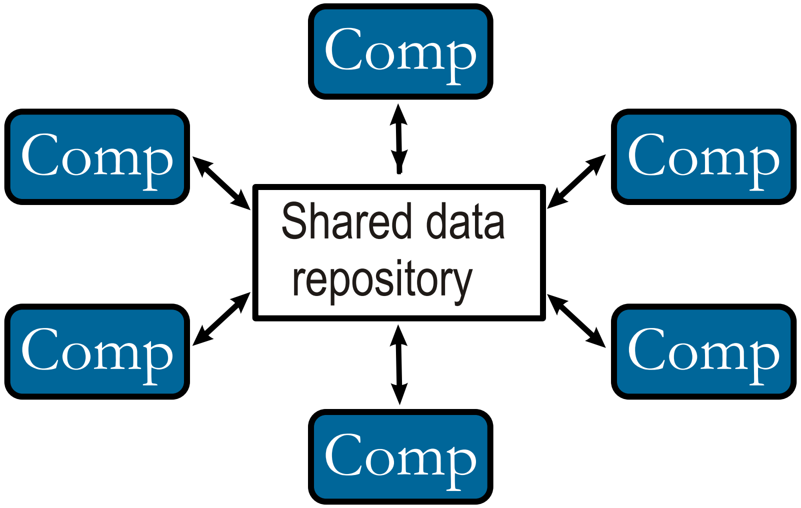


  - The invocation chain (control flow) follows the pipe
  - The dataflow follows the invocation chain by passing parameters at each invocation

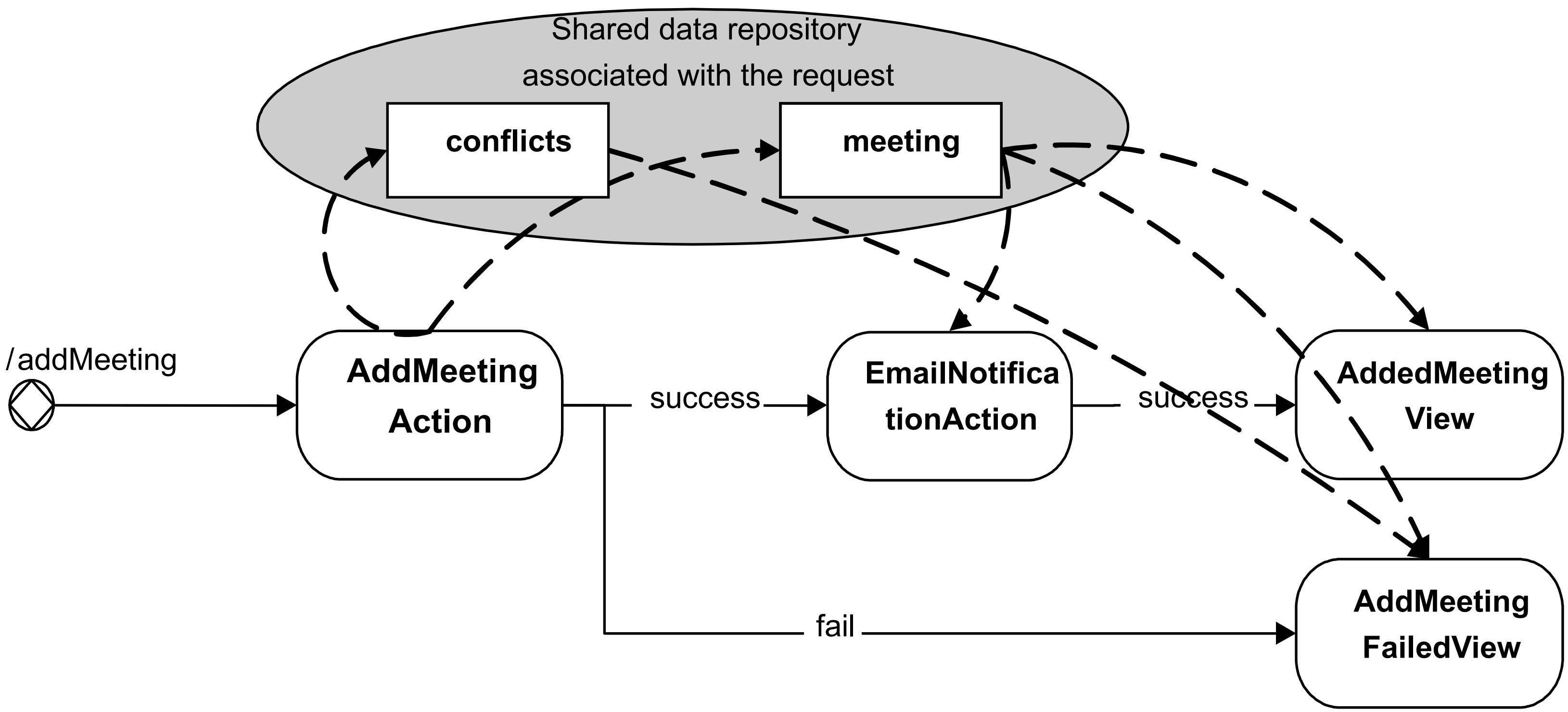
- To ease the composition, uniform interfaces are often used

Indirect data sharing

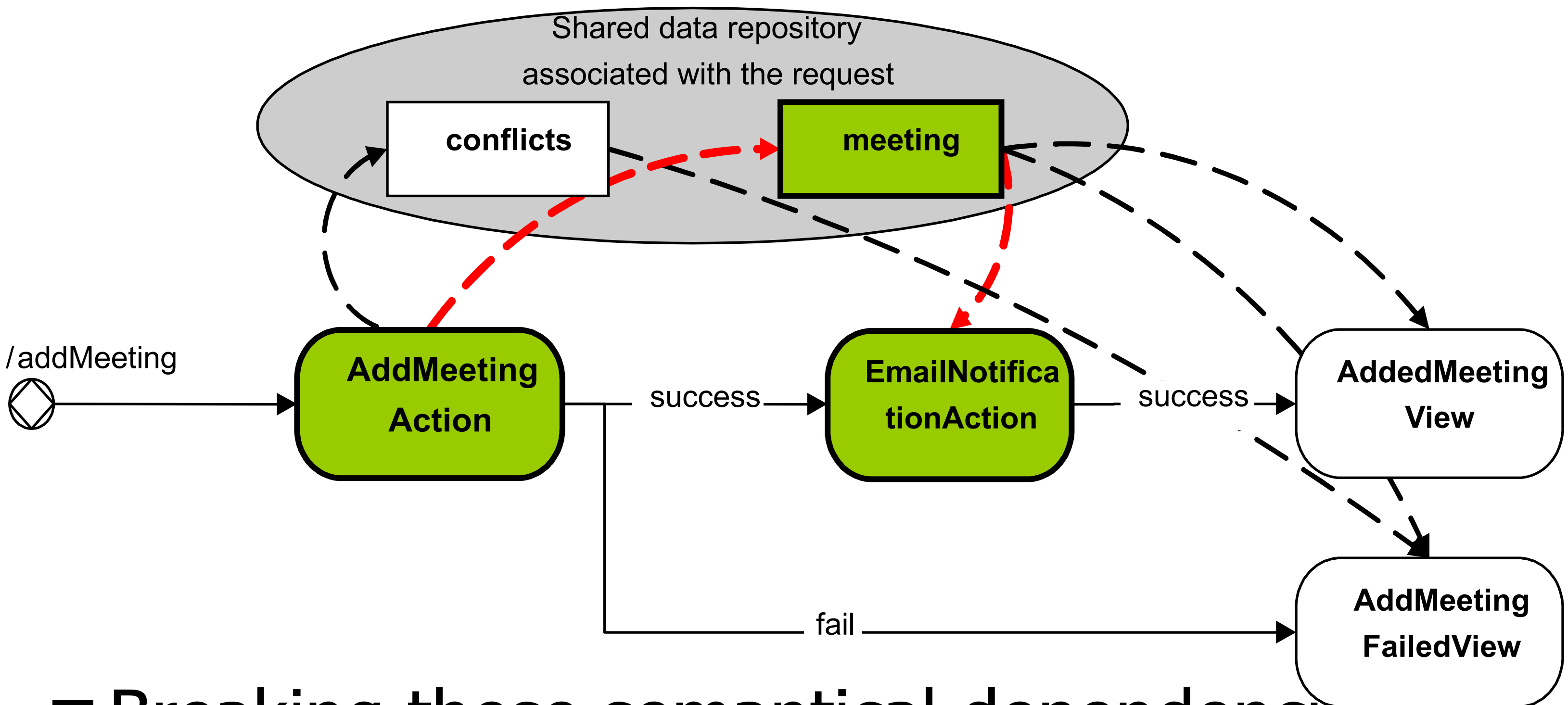
- Data-centered style:
  - Central data repository
  - Components can read and write data to the repository
  - Components share data through the shared data repository



Calendar composition example



Semantical dependencies



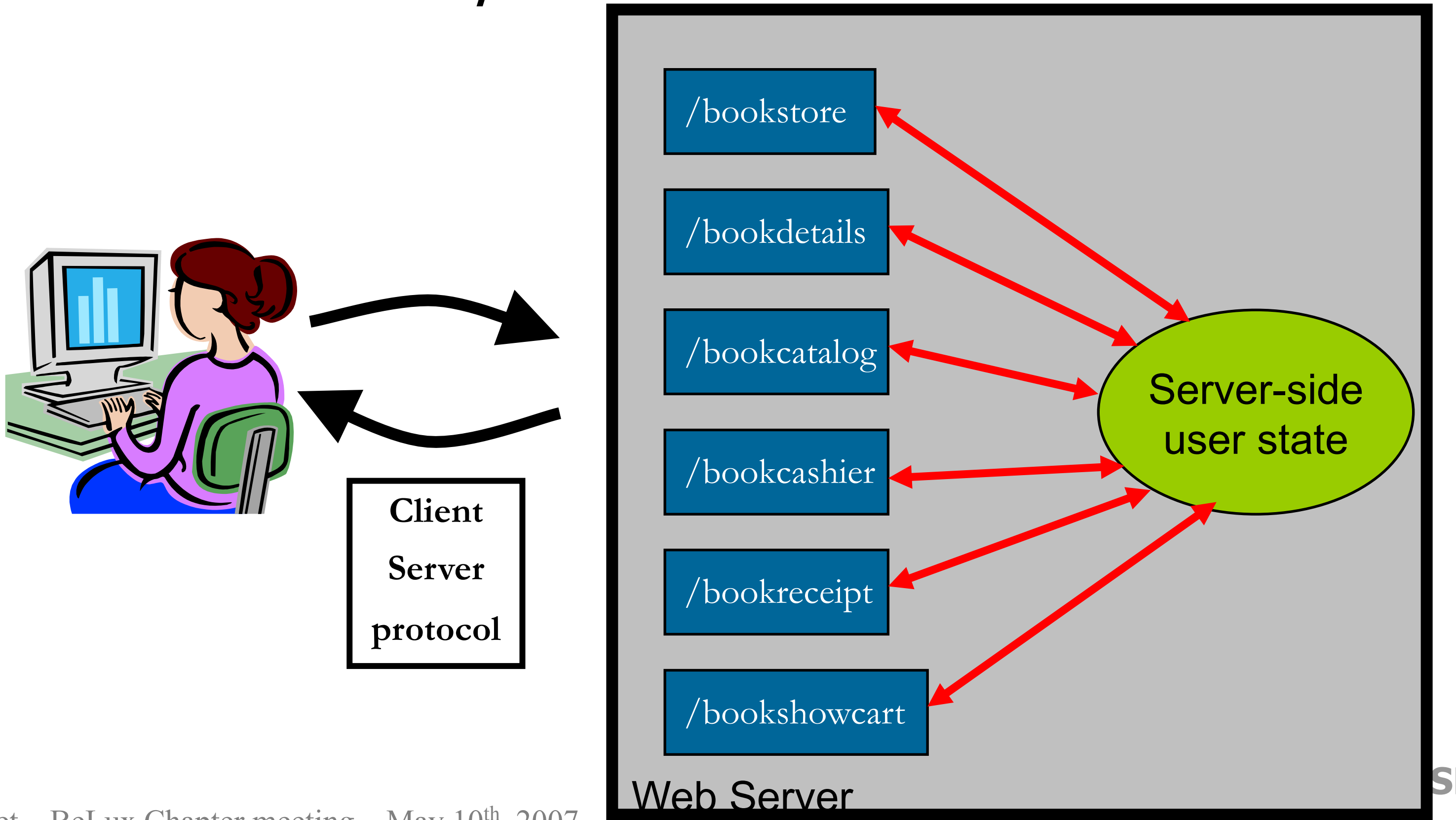
- Breaking these semantical dependencies typically leads to run-time errors!

Overview

- Introduction
- Problem statement
  - Duke's BookStore application
  - Goal and scope of the presented research
- Static verification of indirect data sharing
- Static and dynamic verification
- Conclusion

Duke's BookStore application

- E-commerce site bundled with the J2EE 1.4 tutorial
- Reactive client/server interaction



Shared data interactions

- Session repository with 3 data items:
  - messages (*ResourceBundle*)
  - cart (*ShoppingCart*)
  - currency (*Currency*)

BookDetailsServlet:
ResourceBundle messages (read)
Currency currency (cond. def. read/write)

BookStoreServlet :
ResourceBundle messages (def. read/write)

ReceiptServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)

OrderFilter:
ShoppingCart cart (read)
Currency currency (read)

CashierServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)
Currency currency (def. read/write)

CatalogServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)
Currency currency (def. read/write)

ShowCartServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)
Currency currency (cond. def. read/write)

  - cond. def. read/write

Identified problems

BookDetailsServlet:
ResourceBundle messages (read)
Currency currency (cond. def. read/write)

BookStoreServlet :
ResourceBundle messages (def. read/write)

ReceiptServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)

OrderFilter:
ShoppingCart cart (read)
Currency currency (read)

CashierServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)
Currency currency (def. read/write)

CatalogServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)
Currency currency (def. read/write)

ShowCartServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)
Currency currency (cond. def. read/write)

- BookStoreServlet is not executed first:
  - NullPointerException on retrieval of ‘messages’ data item
- OrderFilter/ReceiptServlet are executed before cart and currency are stored to the repository
  - NullPointerException on retrieval of ‘cart’ and ‘currency’ data items

Desired composition property

- ***<u>No broken data dependencies on the shared repository</u>***
  - A shared data item is only read after being written on the shared repository

  - For each read interaction, the data item present on the shared repository is of the type expected by the read operation

Goal and scope of the presented research

- Goal:
  - Eliminate run-time errors by formally guaranteeing the '*no broken data dependencies*' property
- Scope:
  - Component-based software with indirect data sharing
  - Deterministic and reactive software compositions
- Important non-functional criteria:
  - Reasonable overhead
  - Applicable to real-life applications

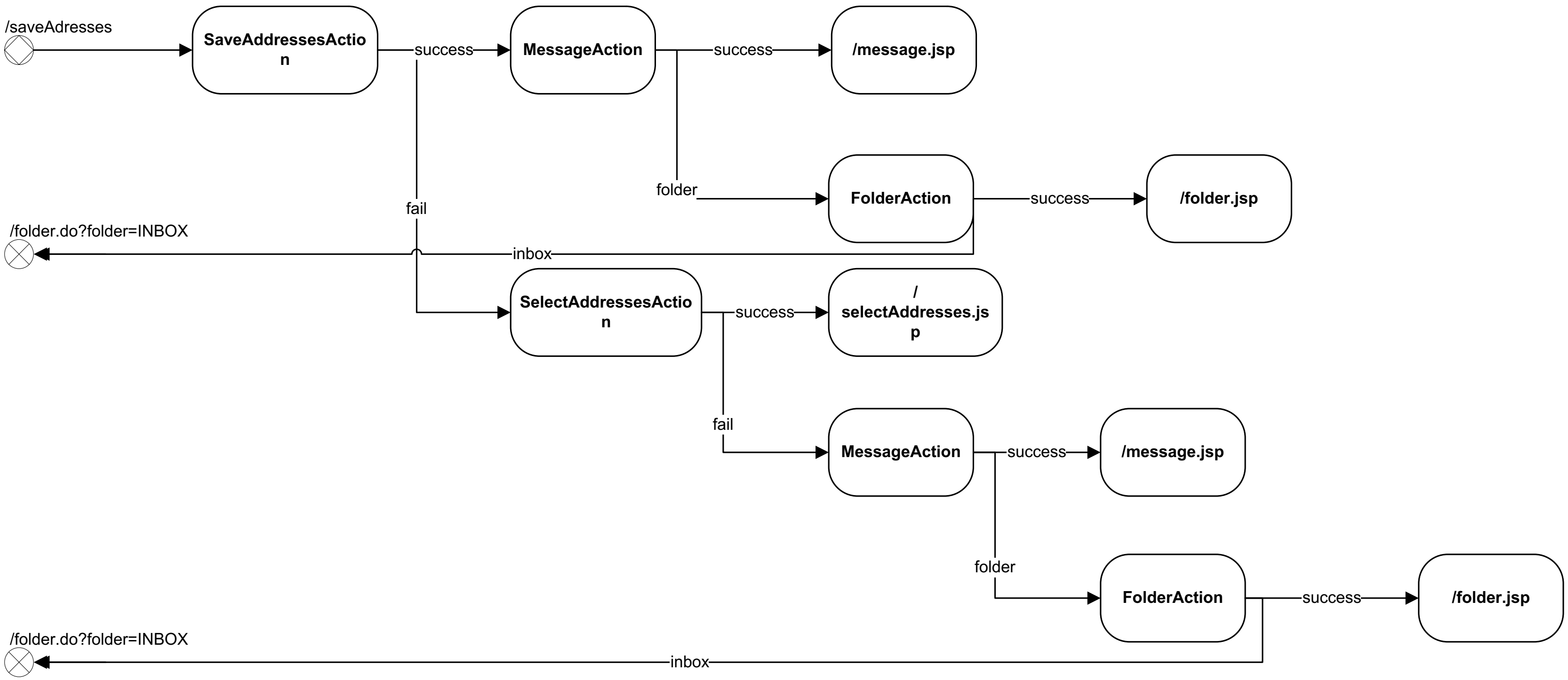
Dependency analysis in GatorMail

- GatorMail
  - Open-source webmail application built upon Struts
  - 20K lines of code
  - 65 components

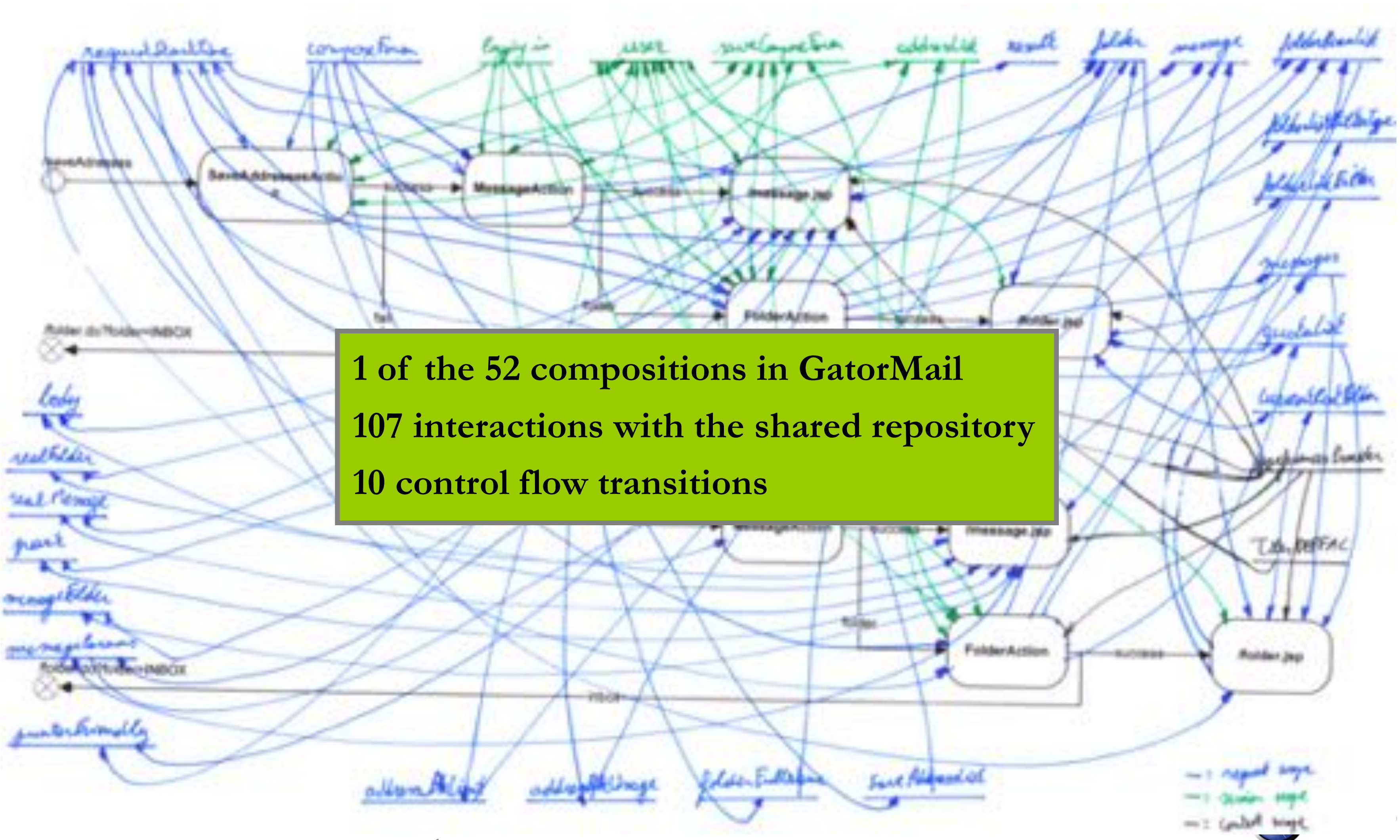
- Analysis results:
  - 65 components reused in 52 request processing flows
  - 1369 hidden interactions with the shared repository
  - 147 declarative control flow transitions

Complex dependency management

- Composition: /saveAddresses.do



Complex dependency management



1 of the 52 compositions in GatorMail

107 interactions with the shared repository

10 control flow transitions

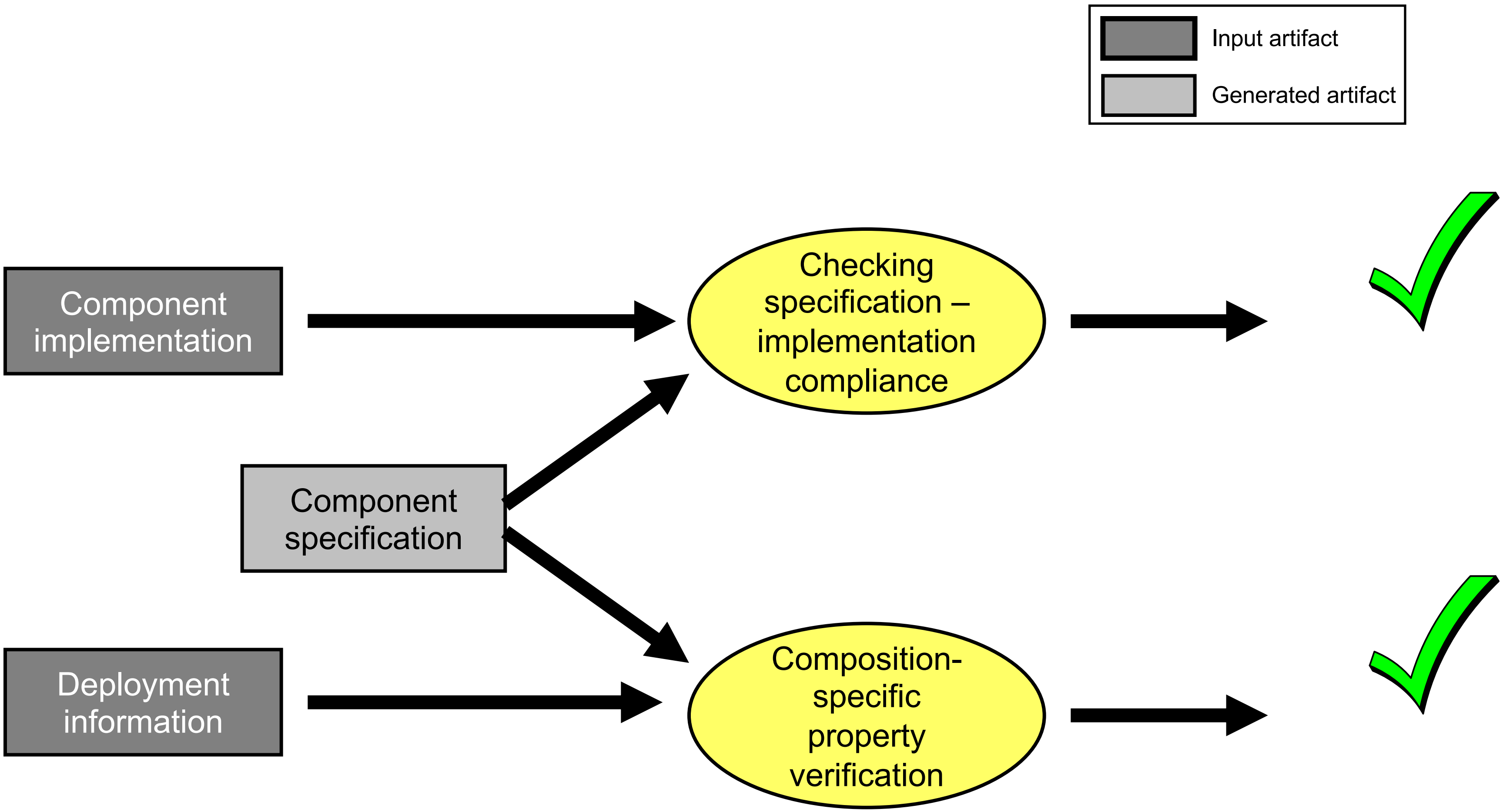
Overview

- Introduction
- Problem statement
- **Static verification of indirect data sharing**
  - Solution overview
  - GatorMail validation experiment
- Static and dynamic verification
- Related work
- Conclusion and future work

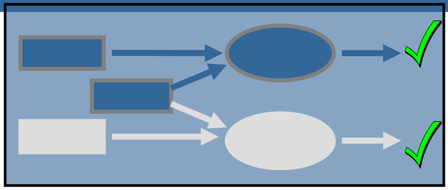
Solution

- Our approach uses static verification to guarantee that the *no broken data dependencies property* holds in a given composition
- Verification is based on component contracts instead component implementations
- 2 steps:
  - Identify interactions
  - Statically verify composition property

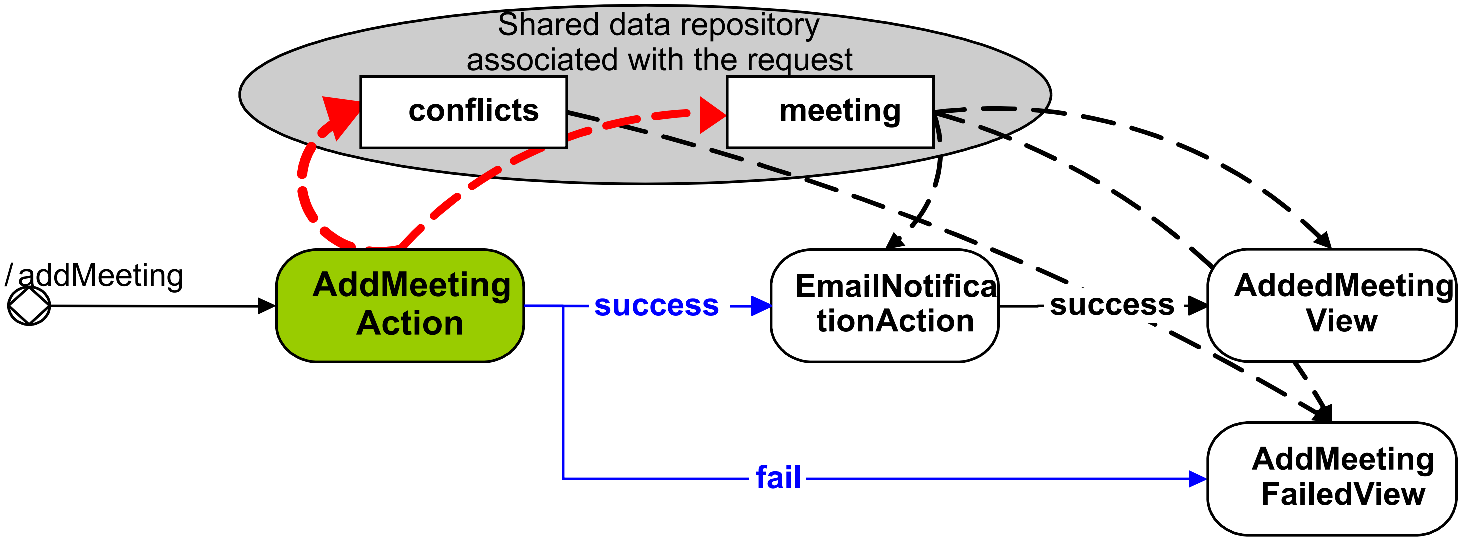
Solution overview



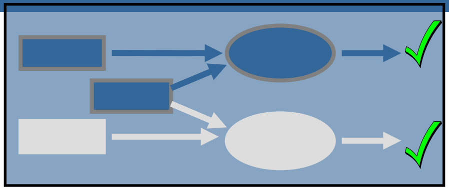
Component contracts



- Specify the component's interactions with the shared repository
- Specify the possible declarative forwards



AddMeetingAction contract



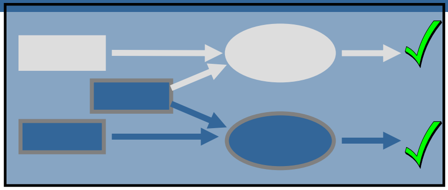
```
//spec: forwards {"success", "fail"};
//spec: writes {Meeting meeting};
//spec: on forward == "fail" also writes {Vector conflicts};
```

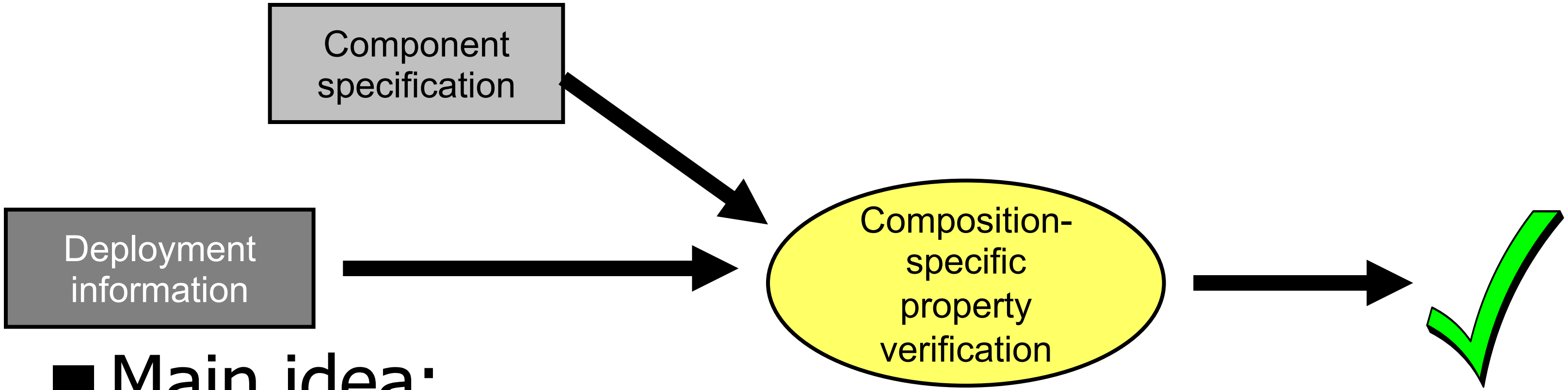
Automatically translated into Java Modeling Language (JML)

```
public class AddMeetingAction extends Action {
    //@ also
    //@ requires request != null;
    //@ ensures request.getDataItem("meeting") instanceof Meeting;
    //@ ensures \result == "fail" ==> request.getDataItem("conflicts") instanceof Vector;
    //@ ensures \result == "success" || \result == "fail";
    public String execute(Request request, Form form);
}
```

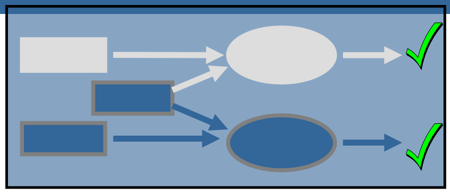
in order to be verified by existing verification tools

Composition-specific verification



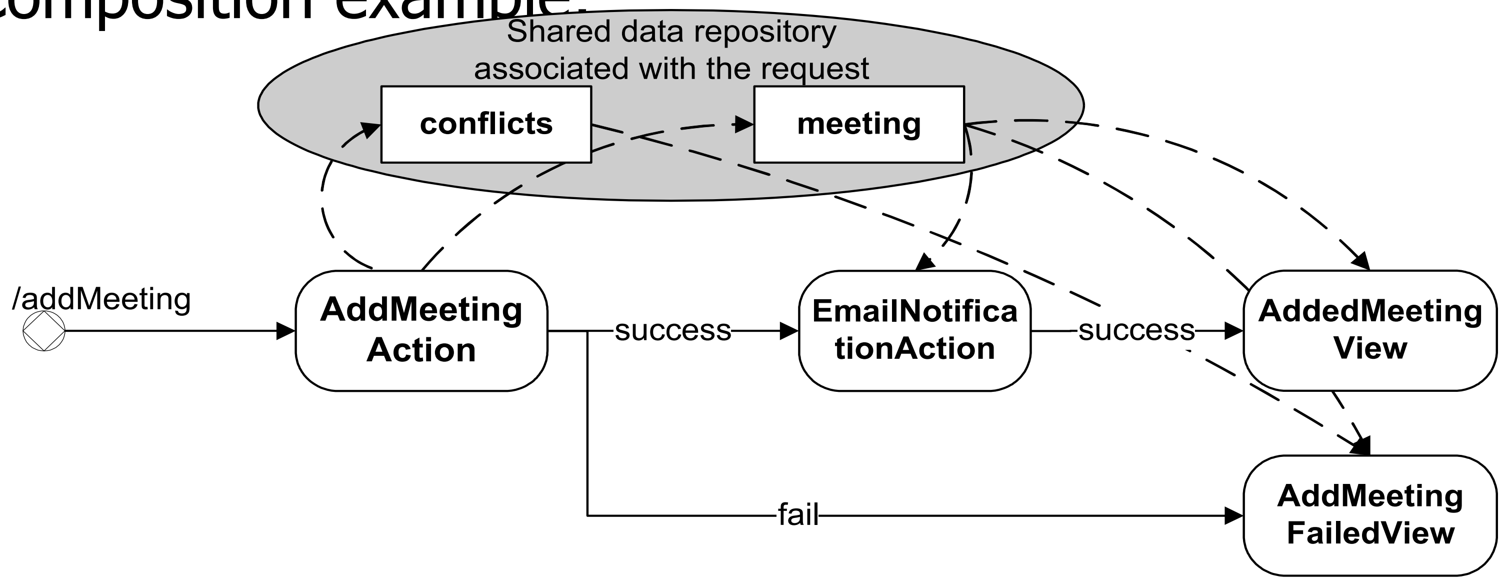


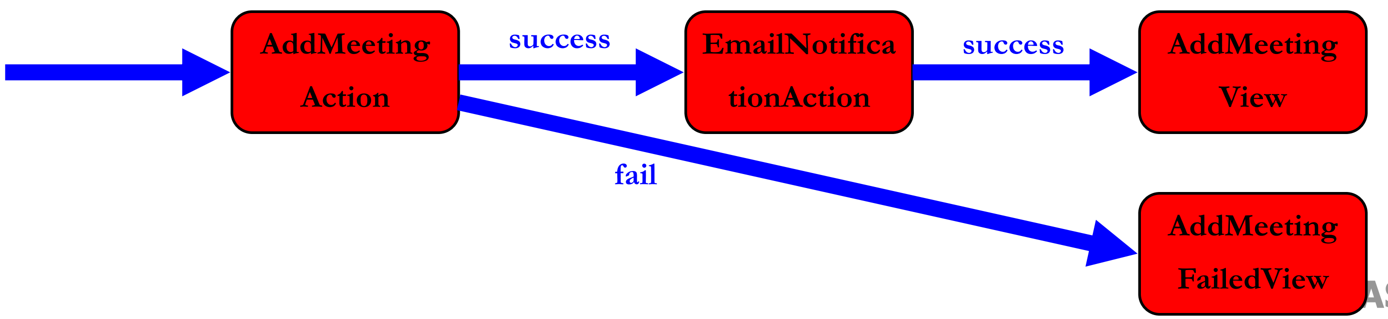
- Main idea:
  - Verify if the composition property holds for each possible execution path in the composition
- Concrete:
  - Generate a composition-specific check method, enrolling the possible run-time execution paths
  - Use existing verification tools to verify the composition property for each execution path



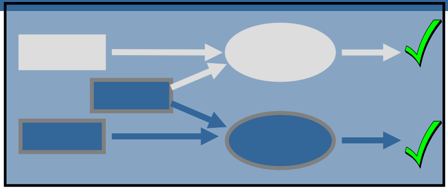
Enrolling the execution paths

- Composition example:





Enrolling the execution paths



```
//@ requires request != null;
public void check_addMeeting(Request request, Form form){
    AddMeetingAction addMeetingAction = new AddMeetingAction();
    EmailNotificationAction emailNotificationAction = new EmailNotificationAction();
    AddedMeetingView addedMeetingView = new AddedMeetingView();
    FailedAddedMeetingView failedAddedMeetingView = new FailedAddedMeetingView();

    String forward1 = addMeetingAction.execute(request,form);
    if(forward1.equals("success")){
        String forward2 = emailNotificationAction.execute(request,form);
        if(forward2.equals("success")){
          addedMeetingView.execute(request,form);
        } else { //@ unreachable; }
    } else if(forward1.equals("fail")){
        failedAddedMeetingView.execute(request,form);
    } else { //@ unreachable; }
}
```

Evaluation

- Prototype implementation:
  - Step1:
    - JML as intermediate specification language
    - Our problem-specific contracts are automatically translated into JML
    - ESC/Java2 as static verification tool
  - Step 2:
    - Composition-specific verification is automatically generated from the deployment information
    - ESC/Java2 as static verification tool
- Evaluation on the GatorMail webmail application
- Presented approach was applicable with only some slight refinements

Experiment results

- JML annotation overhead
  - At most 4 lines of problem-specific annotation

- Verification performance:
  - Modular verification
  - The verification takes up at 700 seconds per component

Conclusion

- We are able to guarantee the desired composition properties in a given composition
  - With minimal formal specification
  - Using existing reasoning tools
  - In a reasonable amount of time
- Proposed solution
  - Applicable to real-life applications
  - Scalable to larger applications (if the complexity of the individual components remains equivalent)

Overview

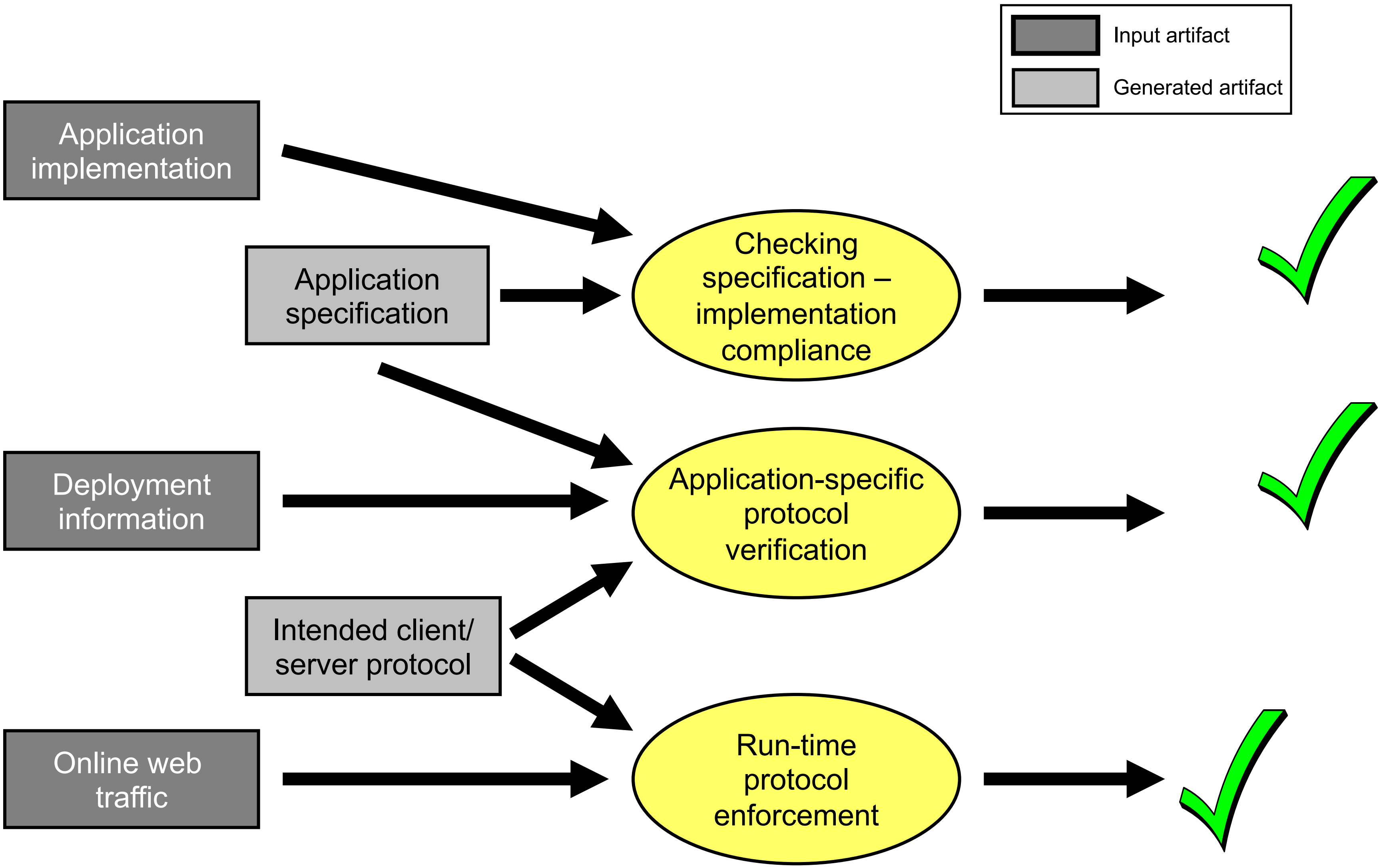
- Introduction
- Problem statement
- Static verification of indirect data sharing
- Static and dynamic verification
    - Solution overview
    - Duke's BookStore validation experiment
- Conclusion

Solution

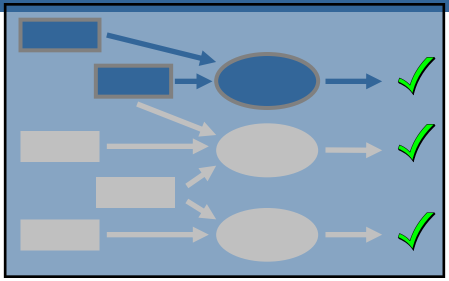
- Our approach uses static and dynamic verification to guarantee that the *no broken data dependencies property* holds in a given, reactive composition

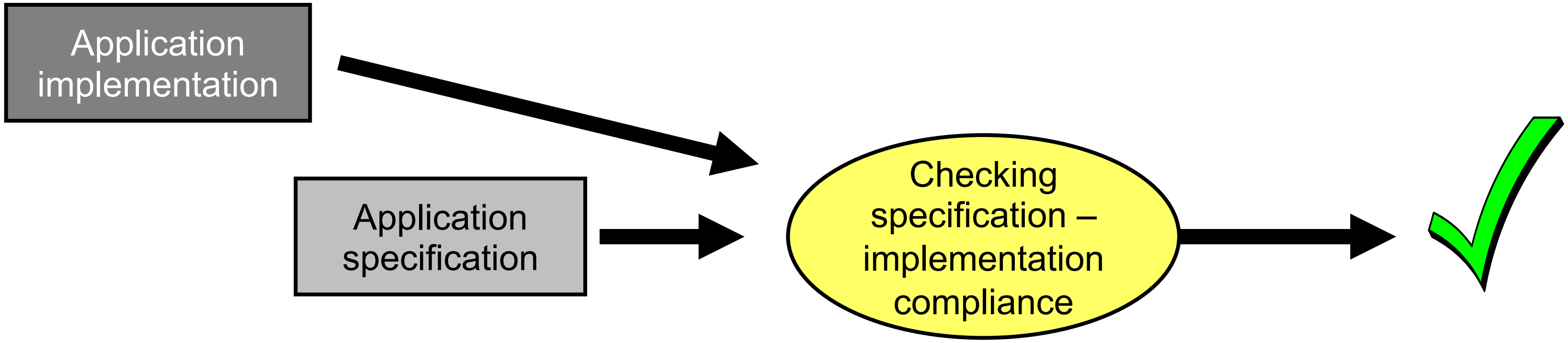
- 3 steps:
  - Identify interactions
  - Statically verify composition property
  - Enforce underlying assumptions at run time

Solution overview



Step 1

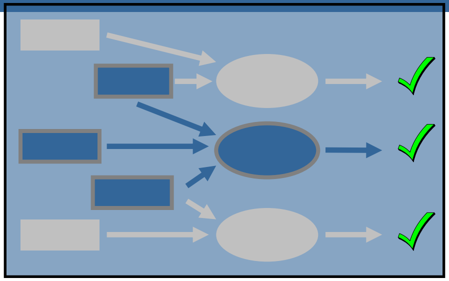


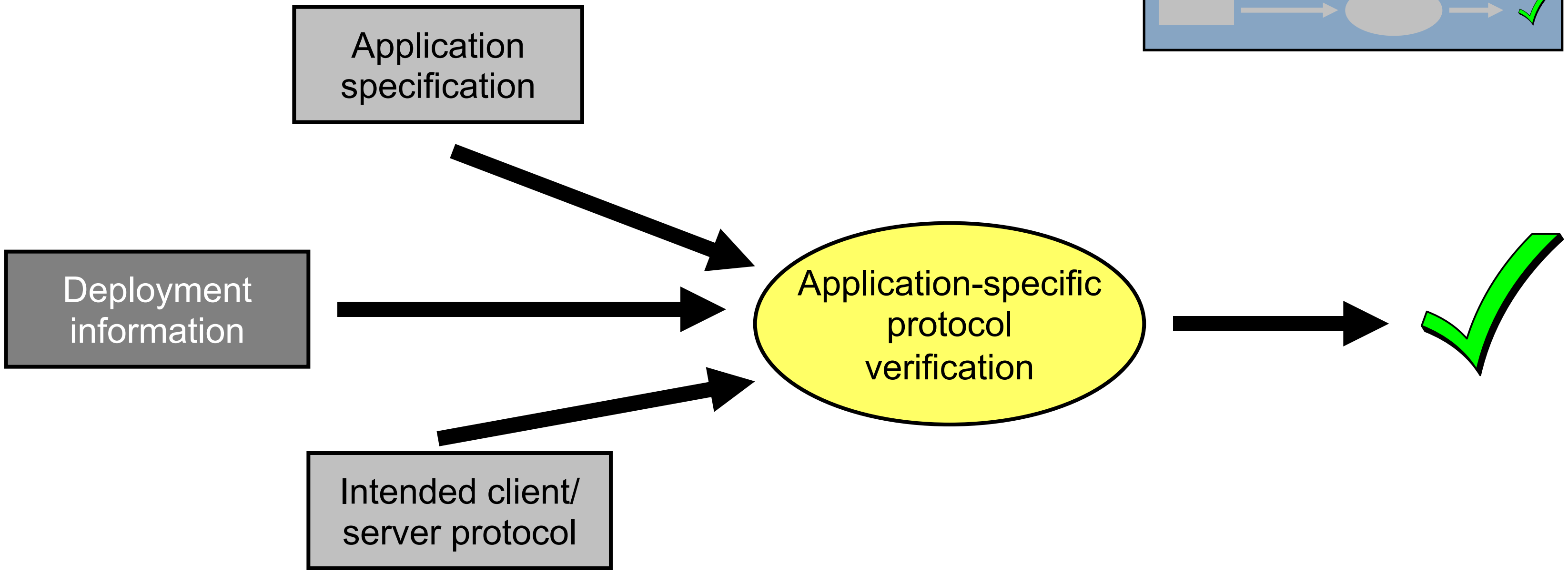


- Component contracts specify interactions with the shared repository:

```
//spec: reads {ResourceBundle messages, Nullable<ShoppingCart>cart,
                                        Nullable<Currency> currency} from session;
//spec: writes {cart == null => ShoppingCart cart} on session;
//spec: possible writes {currency == null => Currency currency} on session;
```

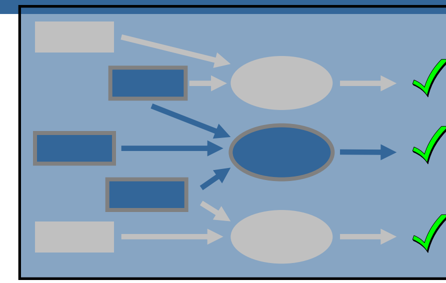
Step 2

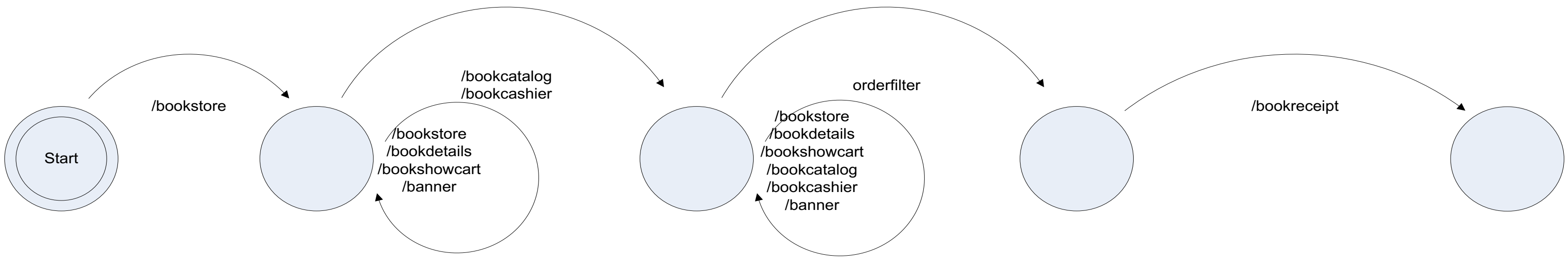




- Simulate all possible client-server interactions that comply to the intended client/server protocol
- Use static verification to formally guarantee that the *no broken data dependency property* is not violated

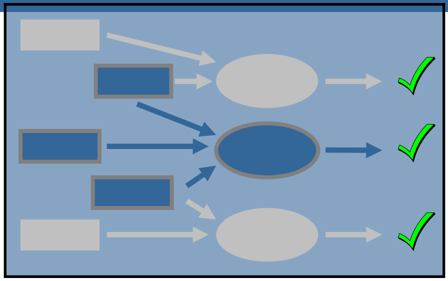
Intended client/server protocol





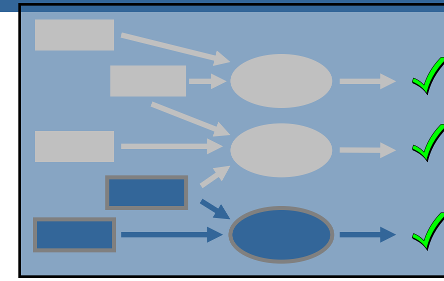
PROTOCOL := /bookstore + SERVLET A + RECEIPT
RECEIPT := (SERVLET B + SERVLET + /orderfilter + /bookreceipt) | nil
SERVLET := SERVLET A | SERVLET B
SERVLET A := /bookstore | /bookdetails | /bookshowcart | /banner | nil
SERVLET B := /bookcatalog | /bookcashier

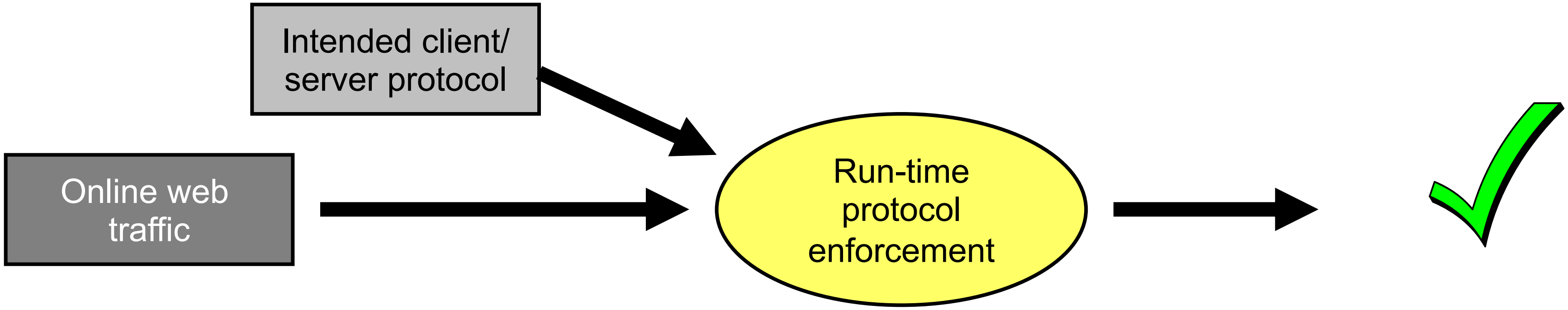
Application-specific verification



```
// …
if (random.nextBoolean()){
  switch(random.nextInt()){
    case 0: cashier .doGet(request,response); break;
    default: catalog.doGet(request,response); break;
  }
  while(random.nextBoolean()){
    switch(random.nextInt()){
      case 0: showcart.doGet(request,response); break;
      case 1: catalog.doGet(request,response); break;
      case 2: cashier .doGet(request,response); break;
      case 3: bookstore.doGet(request,response); break;
      case 4: bookdetail.doGet(request,response); break;
      default: break;
    }
  }
  // …
}
```

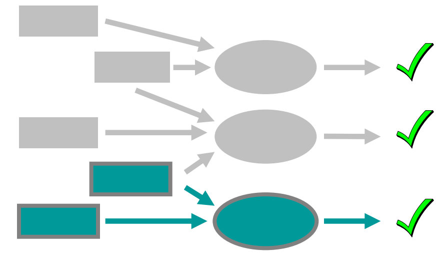
Step 3

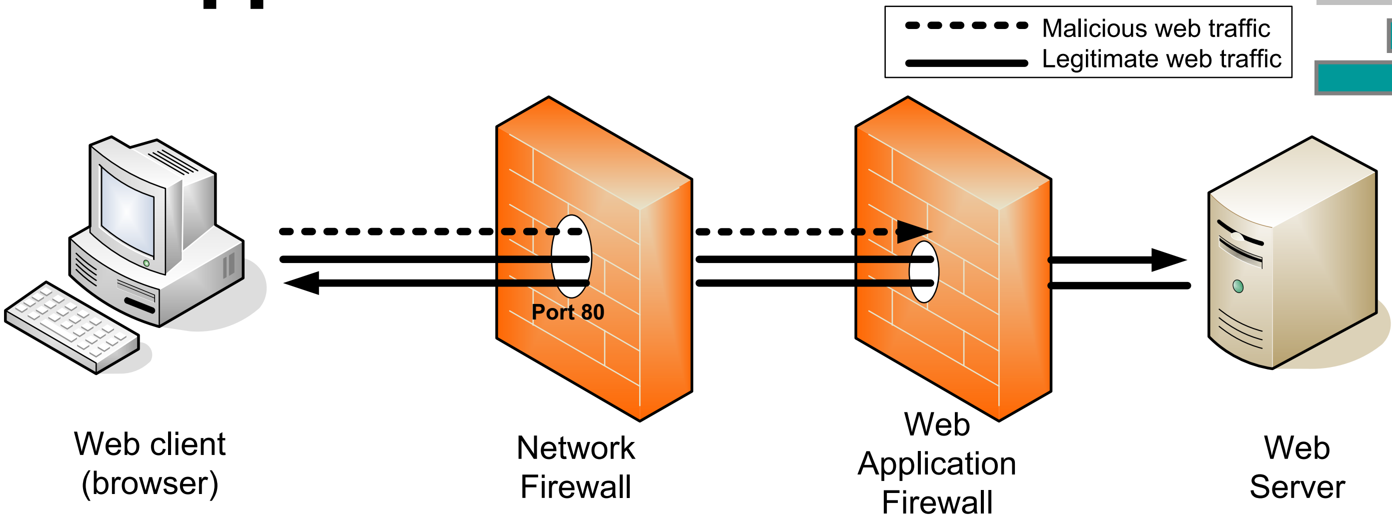




- Limit traffic to the intended client/server protocol
- Typical use of a Web Application Firewall (WAF) in protecting against forceful browsing

Web Application Firewalls





- Protect web applications a.o. against forceful browsing (cf. WAFEC)
- Typically implementation-agnostic
- No formal guarantee that they protect against exploits targeting implementation bugs

Evaluation

- Prototype implementation:
  - Step1:
    - JML as intermediate specification language
    - Our problem-specific contracts are automatically translated into JML
    - ESC/Java2 as static verification tool
  - Step 2:
    - Application-specific verification is automatically generated from the EBNF protocol specification
    - ESC/Java2 as static verification tool
  - Step 3:
    - J2EE filter as a proof-of-concept flow enforcement WAF
- Evaluation on the Duke's BookStore application from the J2EE 1.4 tutorial

Experiment results

- Annotation overhead:
  - At most 4 lines in our problem-specific annotation

- Verification performance:
  - Static verification took at most 4 minutes per component

Experiment results

- Run-time overhead:
  - Experiment:
    - sequence of 1000 visitors
    - on average 6 requests per session
    - 2% of the users applied forceful browsing
  - Measured run-time overhead of 1.3%

- In comparison:
  - In a previous prototype without static verification, a run-time overhead of approximately 20% was measured

Conclusion

- We are able to guarantee the desired composition properties in a given, reactive composition
  - With minimal formal specification
  - Using existing reasoning tools
  - In a reasonable amount of time
- Proposed solution
  - Applicable to real-life applications
  - Scalable to larger applications (if the complexity of the individual components and the protocol remains equivalent)
- We leverage WAFs to protect application-specific implementation bugs

Overview

- Introduction
- Problem statement
- Static verification of indirect data sharing
- Static and dynamic verification
- **Conclusion**
  - Contributions
  - Future work

Contributions

- Contributions:
  - We improved the reliability and security of web applications by:
    - Guaranteeing the *no broken data dependencies property*
    - Applying static verification in deterministic software compositions
    - Combining of static and dynamic verification in reactive software compositions
- Validations:
    - Validation in both deterministic and reactive software compositions
    - Low annotation cost
    - Reasonable verification time (static & dynamic)
    - Applicable to real-life applications

Future work: short term

- Support concurrent server processing by adding a fine-grained concurrency model
  - Simple model: introduce lock per user session
  - More fine-grained: maximise parallelism based on disjunct interactions with the repository
- Enrich the intended client/server protocol by incorporating input parameters and cookies
  - Formally verify the effectiveness of applied input validation checks, e.g. in WAFs

Future work: longer term

- Valorise research in a developer's tool
  - Specification inference !
  - Protocol inference !
  - Useful feedback to the developer
  - Integration into IDE
- Generalise the approach of problem-specific annotation and verification
  - Application to other composition properties
  - Composability of different properties
  - Compare to alternative approaches, such as pluggable type systems

Thank you!

Formal absence of implementation bugs in web applications:
A case study on indirect data sharing

Lieven Desmet
DistriNet Research Group
Katholieke Universiteit Leuven
Lieven.Desmet@cs.kuleuven.be
+32 16 32 79 53

OWASP
BeLux Chapter
May 10th,2007

The OWASP Foundation
http://www.owasp.org