MAINTAINING XML DATA INTEGRITY IN PROGRAMS

AN ABSTRACT DATATYPE APPROACH

Patrick Michel
Arnd Poetzsch-Heffter

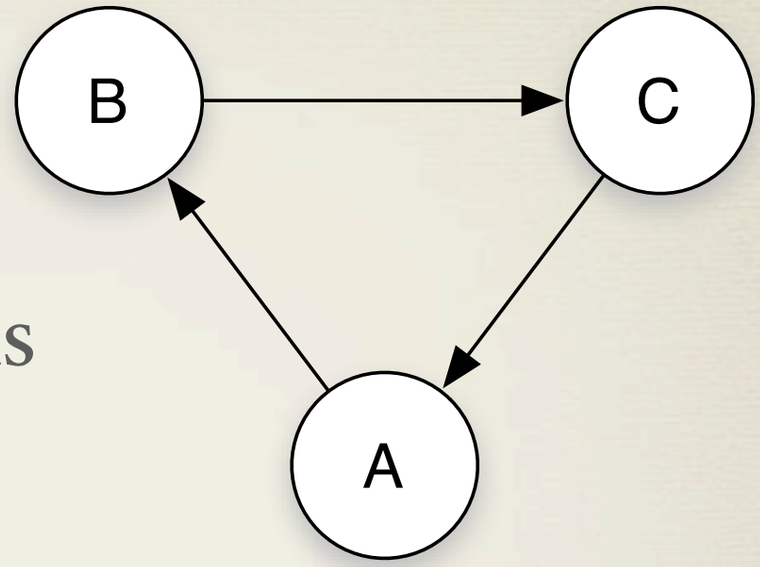
Technische Universität Kaiserslautern

SoFTeX Software Technology Group

Outline

* Overview of the Problem Domain
* Abstract Datatype Approach
* Implementation of the Approach
* Tool Demo
* Conclusion

Scenario



- loosely coupled distributed systems
- collaborate (workflows)
- exchange XML data
- data is schema-constrained
- applications have to keep the data valid (invariant)
- applications are written in languages like Java or C#

Simple Example Schema

```
start =
  element bin {
    attribute capacity {
      xs:integer [. > 0] [ sum(//size) <= . ]
    },
    element item * {
      attribute size { xs:integer [. > 0] }
  } }
```

* typical integrity constraints:
  * range constraints
  * value comparisons
  * contain aggregates like `sum`, `count`, etc.
  * contain references (e.g. an `item` could reference a `type`)

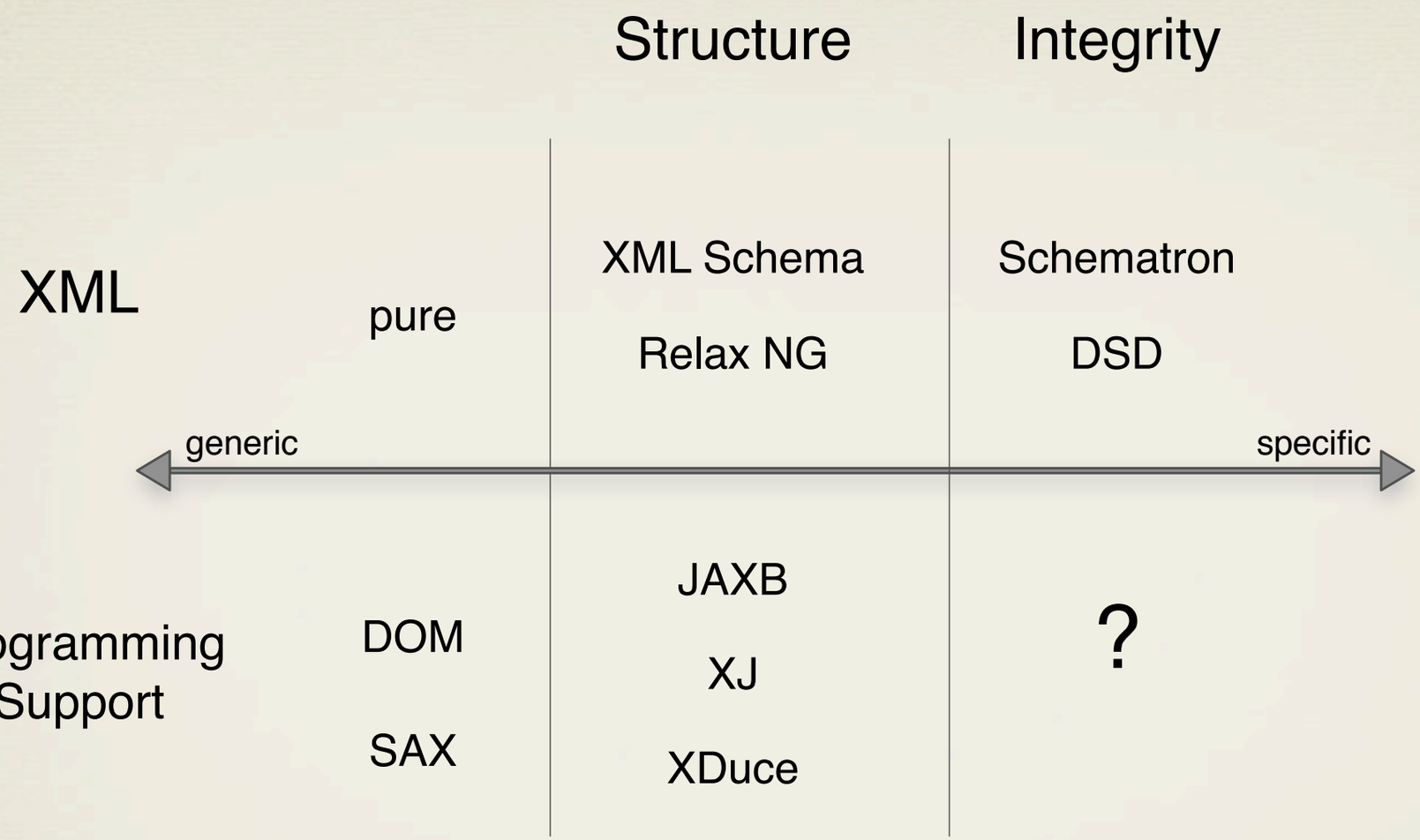
Integrity Constraints

- structural and base types are not enough
  - e.g. tax declaration forms
  - value consistence, value relations

```
//capacity > 0
sum(//size) <= /bin/capacity
sum(//salary[//employee/level]/amount) <= //budget
```

- integrity constraints are inherent to datatypes
  - failures are fatal
  - constraints have to be invariant
  - modifications have to be correct

XML Support



* Validating ✓, Reading ✓ (even gets easier), Modifying ?

Maintaining Data Integrity

- consider a **Java** method `addItem`
  - implemented using e.g. **DOM** or **XJ**
  - modifies data constrained by `bin`-schema (**Relax NG**)
  - does it violate any integrity constraints? (e.g. **XPath**)
- combinations of complex languages
  - hard to know in advance if invariants are violated
  - expensive to check if invariants are violated
  - hard to recover from a detected error
  - verification is next to impossible

Abstract Datatype Approach

* XML datatype with integrity constraints
  * declarative definition (like `bin` example)
  * with a set of interface procedures
    * written in a restricted language with XML support
    * e.g.

```
proc addItem(ident id, int size) {
  insert /bin <item id=(id) size=(size) />
}

proc remItem(ident id) {
  free //item[id];
}
```

  * prove that all procedures maintain the invariant
  * proof is done on the schema + procedures alone

Abstract Datatype Approach

* generate abstract type with these methods
  * invariant, structure and implementations hidden
  * modifications through interface procedures
  * all language features can be used
  * ok to allow introspections for reading (with any language)

* e.g.: class `Bin`, with interface procedures:
  * `addItem(Identifier id, Integer size)`
    * `DuplicateItemException`
    * `InvalidSizeException`
    * `CapacityExceededException`
  * `remItem(Identifier id)`
    * `NoSuchItemException`

Using the ADT in Java

- Backtracking Bin-Packing Algorithm
  - using recursion and loops on interface procedures
  - exploiting the fact that no invariant can be violated

```
public static boolean pack(Bin source, Bin[] target) {
    if(source.bin().item().empty()) return true;
    itemElement item = source.bin().item().first();
    for(int i = 0; i < target.length; i++) {
        try { target[i].addItem(item.Id(), item.size()); }
        catch(CapacityExceededException e) { continue; }
        source.remItem(item.Id());
        if (pack(source, target)) return true;
        target[i].remItem(item.Id());
        source.addItem(item.Id(), item.size());
    }
    return false;
}
```

Implementation

- approach is more general
- we focus on automated methods
  - Java programmers can use this!
  - trying to support as many features as possible
- prototype system
  - schemata lead to path-based propositions (invariant)
  - weakest precondition technique for procedures
  - simplification technique to get smallest incremental check, using an SMT solver in the process
  - remaining preconditions become exceptions

Conclusion

- Integrity constraints are essential to datatypes.
- To be able to maintain them, XML data is made available as ADT, with a set of interface procedures.
  - The constraints are defined and maintained without involving the host language semantics.
  - Still, all host language features can be used to create complex algorithms on top of interface procedures.
  - Correctness proofs can be automated for useful invariants combined with local manipulation procedures.
- The technique is usable by Java programmers, as no background in theory is needed.

Invariant

Structure:

```
/bin
/bin/capacity
{ /bin/item{x} } /bin/item{x}/size
```

Typing:

```
/ is complex
/bin is complex
/bin/capacity is int
{ /bin/item{x} } /bin/item{x} is complex
{ /bin/item{x}/size } /bin/item{x}/size is int
```

Inegrity:

```
/bin/capacity > 0
sum (/bin/item*/size) <= /bin/capacity
{ /bin/item{x}/size } /bin/item{x}/size > 0
```

Preconditions

Preconditon:

```
not /bin/item[id]                          DuplicateItemException
size > 0                                   InvalidSizeException
size + sum (/bin/item*/size) <= /bin/capacity
                                           CapacityExceededException
```

```
proc addItem(ident id, int size) {
  new /bin/item[id];
  new /bin/item[id]/size;
  set /bin/item[id]/size size;
}
```