Model Checking of High Level Specifications: The LfP Experience

Frédéric GILLIERS, François Bréant, Denis Poitrenaud, Fabrice Kordon
Goal: develop reliable distributed systems
- Distributed => asynchronous
- Reliable => well defined, predictable behavior

Our proposal:
- A specification language: LfP (Language For Prototyping)
- An associated model based methodology

This work is part of the MORSE project
- Industry / academic cooperation project
• **Separation** of control and computational aspects
  
  • **We focus on the control aspect**

![Diagram](image)
LfP Main characteristics

- Structured
  - Based on hierarchical automata
  - Component types (media / classes)
  - Instruction set dedicated to modeling protocols

- Transparent to distribution
  - The specification’s behavior does not rely on the deployment

- Three kinds of diagrams
  - Architecture diagram
    - Static aspect => components declaration
  - Behavioral diagram
    - Dynamic aspect => components behavior
  - Property diagram
A simple LfP example (1)

```
server

handle_request(num: in out integer)

client

RPC

type simple_port is port (integer);
c1 : client with (id => 1);
c2 : client with (id => 2);
c3 : client with (id => 3);
c4 : client with (id => 4);

s1 : server with ();
s2 : server with ();

all

1

Server

RPC

1

Client

server.itf
6
fifo

Client.itf
8
fifo
```
A simple LfP example (2): the class SERVER

- One method “handle_request”
  - Formal parameter: numn value updated on method return (copy restore)
    ```
    itf: simple_port;
    procedure handle_request (num: inout integer);
    ```

- Infinite loop:
  - Wait for method call, execute method, back to initial state

- One method: handle_request -> increments the actual parameter
  ```
  procedure handle_request(num: inout integer) is end;
  ```
A simple LfP Example (3):
The class CLIENT

- Creates the communication media instance
- Calls handle_request until the parameter is greater than 5
Communication between client and server

- Read activation message in the client’s binder
- Send it to the server
- Read the corresponding return message (among other messages)
- Send it back to the client
• Shared decision tree
• A priori unbounded integer domain for variables
• No variable ordering constraint
• Variables may be repeated
• Three terminals:
  • 0 (unaccepted): not represented unless empty set
  • T (top: approximation): introduced to resolve variable ordering conflicts
  • 1 (accepted): all paths in a well-defined DDD lead to 1
• Usual set theoretic operations offered
**DDD Operations: Inductive Homomorphisms**

- Algebraic properties: union: $h + h'$, intersection: $h \times h'$, concatenation $h \cdot h'$, composition $h \circ h'$

- Defined by:
  - Evaluation on terminal $1 \rightarrow$ constant DDD
  - Evaluation on a `<variable, value>` pair, i.e. an arc of the structure $\rightarrow$ homomorphism to apply on son

- By definition:
  - $h(T) = T$
  - $h(O) = 0$
  - $h($node$) = \text{sum of evaluation \ on all arcs}$

- An Example:
Set Constant Homomorphism
Example (1/2)
Set Constant Homomorphism

Example (2/2)

\[ \text{SetCst}(a,1,2) \]

\[ \text{DDD}(a,1) + \text{DDD}(b,2) \]

\[ \text{DDD}(a,2) + \text{SetCst}(a,1,2) \]

\[ \text{DDD}(a,2) + \text{DDD}(b,2) \]

\[ \text{SetCst}(a,1,2) \]

\[ \text{DDD}(a,2) + \text{DDD}(b,2) \]

\[ \text{SetCst}(a,1,2) \]

\[ \text{DDD}(a,1) + \text{DDD}(b,2) \]

\[ \text{SetCst}(a,1,2) \]

\[ \text{DDD}(a,1) + \text{DDD}(b,2) \]

\[ \text{SetCst}(a,1,2) \]

\[ \text{Cache Hit} \]
Verifying LfP Specifications

- Verification requires:
  - Coding LfP States as DDDs => canonical representation
  - Coding LfP instruction as inductive homomorphisms

- Canonical representation of states
  - Unique representation of a given state
  - Avoids incompatibilities and allows fast comparison of states

```
Global variables | Shared binders | Component Instances | End Of State
```
```
Begin Instance | Instance marker | Local variables | Program Counter | End Of Instance
```
Computing the reachable states

- Define an homomorphism for every LfP instruction
- LfP transitions are implemented as composition of these homomorphisms
  - Fire a transition => apply the corresponding homomorphism

computation of the reachable states:
- A specific homomorphism: MarkFireable
  - Marks all instances of a selected transition that can be fired
- Every state reached by firing an instance of the selected transition are found
- When all transitions have been tested, all states reachable from the current state have been computed

- Deadlocks are easily detected (no fireable transition found)
- Invariants /accessibility properties can be verified
• We have achieved reachable state computation of LfP Specification
  • First step to CTL model checking
• Due to the constrained environment, there's a need for detailed models (for code generation)
  • There is a need for scalability
    • Hierarchical DDDs improve data sharing
  • There is a need for abstraction to reduce the complexity of the verification
    • Observation graph (only track states relevant to the property under verification)
• Related work (MORSE project)
  • Java code generator (prototype)
  • UML2 profile suitable for automatic generation of LfP specifications