Formal Verification of Replication on a Distributed Data Space Architecture

Jozef Hooman
Katholieke Universiteit Nijmegen
Centrum voor Wiskunde en Informatica

Jaco van de Pol
Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

Symposium on Applied Computing
March 2002
Overview

1. Real-time Distributed Data Space – Splice
   (a) publish/subscribe mechanism
   (b) distributed data bases
   (c) crucial role of time-stamps

2. Transparent Replication
   (a) Problem statement
   (b) Case study
   (c) Difficulties
   (d) Solution

3. Complementary Formal Methods
   (a) $\mu$CRL model (process specification)
   (b) PVS model (denotational semantics)
Splice Architecture

Background: Splice was developed at Hollandse Signaalapparaten bv, and is currently the main product of Thales Real-time Solutions.

Goal of Splice: real time and high-bandwidth distribution of volatile data, between heterogeneous processes, supporting fault tolerance.

Application area: Large scale embedded systems, such as command and control systems, process control systems, air traffic regulation systems.

Particular features:

- Publish/subscribe mechanism
- Distributed data storage
- Frequency decoupling of processes
- Automatic replication of processes
Publish/subscribe

- The Splice architecture is data oriented. All processes are seen as producers or consumers of data. Data is partitioned in sorts.
- At run-time processes can join a Splice system by announcing themselves as publisher of a certain sort, or as subscriber to a certain sort.
- Publishers can write data items of that sort, subscribers can read data items of that sort, satisfying certain queries.
- Splice addresses the data distribution.
- Advantage: decoupling between producers and consumers leads to robust and flexible systems.
Splice Agents and Data bases

The application processes are connected to Splice agents, which basically do the following:

- The agent of a publisher broadcasts all written items to the agents of the subscribers.
- The agents of the subscribers store all forwarded data in a local data base.
- The agents of the subscribers serve all read-requests from the local data base.
Structure of the Database

Accumulation of data in buffers is not a good idea: memory consuming, no time to process.

Therefore, Splice provides frequency decoupling of processes with different rates.

Most data has only temporal validity. So Splice uses selective overwriting.

The data base entries consist of three fields:

- **Key** (to identify the data)
- **Value** (to represent the most actual value)
- **Time stamp** (local clock time at writing)
Time stamps

Application processes

write(key, data)

Splice Agent

time

Splice Database

time > t_old?

(key, data, time)

key
d_old
t_old
More Splice features

• Permanent data storage (disk, main memory)
• Dynamic entering/leaving of processes
• High performance
• Fault tolerance
  – Crashed processes are automatically restarted
  – Restarts possibly happen on different nodes
  – Processes can be replicated at run-time

Does selective overwriting with time stamps support transparent replication?
Transparent Replication

A process $P$ can be replicated fully transparently, if $P = P||P$. In general, the fact whether replication of $P$ is transparent depends on

- Properties of $P$.
- What can be observed by the context.

We looked at a concrete and simple case study, which is prototypical for the application area. We compared the following systems:

$$
Producer || Transformer || Consumer
$$

$$
Producer || Transformer || Transformer || Consumer
$$
Case study

Input

Producer
write input

Transformer
read input
write output

Consumer
read output

Output

Replication

Splice layer
The Case Study

Producer:

loop

  d := Input()
  splice_write("input",d)

end

Transformer:

loop

  d := splice_read("input",_)
  splice_write("output",transform(d))

end

Consumer:

loop

  e := splice_read("output",_)
  Output(e)

end
Difficulty and Solution

The system was modeled as a formal specification and analysed, using model checking and theorem proving.

It appeared that replication of the transformer is not transparent. An error trace was produced automatically.

The problem is caused because the transformers use their own local clock, leading to duplicated items with different time stamps.

We proposed an extension of Splice with explicitly timed primitives. It has been formally checked that in this case replication is transparent.

The explicitly timed primitives are added to the newest Splice release.
Methodological issues

We used two complementary formal methods, combining model checking and theorem proving

1. \(\mu\text{CRL} \) approach:
   (a) operational model
   (b) automatic verification
   (c) concrete, finite instances only
   (d) equality between processes
   (e) “rapid prototyping”

2. PVS-approach:
   (a) denotational (compositional) semantics,
   (b) interactive verification
   (c) statements on \(n\) inputs, \(m\) processes
   (d) implementation relation
   (e) “step-wise refinement”
\(\mu\text{CRL approach (1)}\)

Model system as a bunch of processes:

1. Define the processes \(\text{Network, Agent}\).

2. Define \(\text{Splice}\) as the composition:

\[
\text{Splice} = \text{Network} \parallel \text{Agent} \parallel \cdots \parallel \text{Agent}
\]

3. Define application processes \(P_i\)

4. Define the system as the composition of:

\[
\text{System} = \text{Splice} \parallel P_1 \parallel \cdots \parallel P_n
\]
\( \mu \text{CRL approach (2)} \)

Generate state space, minimize, explore:

1. vizualisation
2. simulation
3. equality checking w.r.t. a specification
4. model checking w.r.t. certain properties

System without replication

System with replication
PVS approach (1)

Our PVS approach is based on denotational semantics.

Denotational semantics for Splice programs

1. Define all program constructs as operators on a specific mathematical domain.

2. Mix programs with specifications in pre-postcondition style.

3. Define refinement relation between “mixed programs”

```
SpliceProgs:TYPE= pred[[SemPrim,SemPrim]]
SemPrim : TYPE =
    [# state : [Vars -> Data],
      clock : real,
      db : setof[[Data,real],
      ownw : setof[[Data,real]],
      envw : setof[[Data,real]] #]
```
**PVS approach (2)**

**Application specific:**

1. Model *top level specification*
2. Refine to *parallel composition* of component specs
3. Introduce *replication* at specification level
4. Refine components specs to *programs*

![Diagram]

- **Toplevel Specification**
  - ProdSpec || TransSpec || Cons.Spec
  - TransSpec || TransSpec
    - Producer || Transformer || Transformer || Consumer
Conclusion

• Generic model of Splice in $\mu$CRL. This model supports automatic verification for Splice programs with a finite state space. Also error trace generation is possible.

• A semantics of Splice programs in PVS. This model supports interactive step-wise refinement proofs for Splice programs.

• We studied replication of programs on Splice, found problems, and proposed an extension of the Splice primitives with explicit time stamps.

• Using these new primitives we proved for certain applications that replication is transparent.

• The extension will be incorporated in the next Splice release.