Interoperability of Run-time Tools: Requirements and Concepts

Roland Wismüller

Lehrstuhl für Rechnertechnik und Rechnerorganisation, LRR-TUM
Technische Universität München, Germany
e-mail: wismuell@in.tum.de
www: www.in.tum.de/~wismuell
1. Introduction
   ➞ Run-time tools, interoperability

2. A Typical Scenario
   ➞ debugger with random access to program states
   ➞ required individual tools

3. Interactions Between Tools
   ➞ requirements for a supporting infrastructure
   ➞ existing infrastructures

4. Implementation of the Requirements in the OCM

5. Conclusions and Future Work
1. Introduction

Run-Time Tools (On-line Tools)

- Monitor, analyze und control the execution of a distributed target system (HW+SW)

- E.g. debuggers, performance analyzers, load balancers, ...

Common properties:

- event based scheme of operation
- distributed structure:
  - distributed monitor/control components
    (monitoring system)
  - (usually) centralized analysis component
  - centralized user interface (for interactive tools)
1. Introduction (ctd.)

Typical Structure of a Tool
1. Introduction (ctd.)

Interoperability

- Dazzling term with lots of different definitions:
  - polylingual software (e.g. C and Fortran)
  - communication of distributed programs (e.g. CORBA)
  - data bases (relational vs. OO)
  - ...

- An attempt of a more general definition:

  *Interoperability is the ability of independent software components (not specifically designed for that purpose) to cooperate at a syntactic and semantic level.*
1. Introduction (ctd.)

Interoperability of On-line Tools

- Multiple tools cooperate in the monitoring and control of the same target system

Motivation:

- For the user:
  - concurrent use of tools for different tasks
  - combined use can lead to additional benefits

- For the tool developer:
  - enhanced modularity
  - “factor out” target system dependencies
2. A Typical Scenario

Goal:

- Efficient debugging of long running programs
- Debugger with random access to (past) program states

Idea:

- Periodically create global checkpoints
- Any desired program state can be reached by re-executing from the immediately preceding checkpoint
2. A Typical Scenario (ctd.)

Implementation

- Combine a debugger and a checkpointer
- Plus:
  - a visualizer as a means to specify the target state
  - a deterministic execution controller

receipt of a message
different execution after restart
2.1. The Debugger DETOP

Goal:

- Debugging of process sets

Implementation:

- Hierarchical, distributed design
- Local components use OS interfaces
  - process monitoring (e.g. signals, exceptions)
  - process control (e.g. stopping)
  - read / write process memory
- No support of programming model (PVM)
  - exception: use of PVM task identifiers
2.2. Checkpointing with CoCheck

Goal:

- Consistent checkpointing of communicating processes

Basis:

- Checkpointer for independent processes (Condor)
- Protocol to flush communication links
  - store all messages is receiver’s address space

Implementation:

- Intercept most PVM calls
  - receive stored messages
  - translate task identifiers
2.3. The Visualizer VISTOP

Goal:
- State based visualization of program execution

Implementation:
- Record calls to PVM library routines
- Model state of PVM
- Display a selected state in detail
2.4. Deterministic Execution with Codex

**Goal:**

- Deterministic execution according to specified access patterns for communication objects

**Implementation:**

- Intercept PVM receive calls
  - check which message should be received according to the specified pattern
  - receive this message using a PVM call
3. Tool Interactions

Example 1: Concurrent Accesses

- CoCheck, VISTOP, and Codex monitor PVM calls in the same processes
- CoCheck and Codex modify parameters of these calls

Lesson learned from OS’s and DBS’s

- We need a dedicated layer that coordinates the accesses (voluntary coordination doesn’t work!)

Requirement 1: Common Monitoring System

- We need a common open monitoring interface that coordinates accesses to the target system
3. Tool Interactions (ctd.)

Multiple Tools Monitoring the Same Target System

Node 1

Node 2

Local Comp.

Process 1

Semaphore 1

Central Component

Local Comp.

Process 2

Central Component

Local Comp.

Process 3

Local Comp.

Process 4

GUI
3. Tool Interactions (ctd.)

Common Interface to Target System

GUI -> Central component

Local comp. of Node 1: Process 1, Semaphore 1

Local comp. of Node 2: Process 2, Process 3, Process 4

Central component -> GUI

Common distributed monitoring interface
Example 2: GUI Consistency

- Tools should provide a consistent view of the system
- But:
  - DETOP always displays the current state
  - VISTOP can display an arbitrary (past) state
- Solution:
  - when DETOP is active: VISTOP shows current state
  - when scrolling back in VISTOP: inactivate DETOP
- In addition (due to event buffering):
  - when a process is stopped: VISTOP must read event buffer
3. Tool Interactions (ctd.)

Requirement 2: *Direct Interactions*

- Notification of events occurring in other tools
- Execution of actions in other tools

Requirement 3: *Indirect Interactions*

- Monitoring of transitions of object states (caused by other tools)
3. Tool Interactions (ctd.)

Example 3: *Transparency*

- Restart by CoCheck leads to new task identifiers
- Other tools should still see the old ones
- Identifiers are determined via monitoring interface

Requirement 4: *Interception Object Accesses*

- Mechanism to intercept object accesses of other tools
- Modification of requests and/or results
3. Tool Interactions (ctd.)

Example 4: *The Really Bad Case*

- VISTOP models the PVM receive queues
- Codex violates the FIFO semantics of these queues
  - order of receipt $\neq$ order of entry in queue
- Violates a basic assumption of VISTOP
- Requires extensive modifications of at least one tool

**Recommendation:**

- Avoid implicit assumptions
- Anticipate the presence of other tools
3. Tool Interactions (ctd.)

Summary: Requirements for an Environment Supporting Interoperable On-Line Tools

- “Multi-user” interface for monitoring
  - coordinates concurrent accesses to target system
- Events and actions in other tools
  - enable direct tool interactions
- Monitoring of object state transitions
  - enable indirect tool interactions
- Interception and modification of object accesses
  - maintain views of other tools
# A Short Look at Existing Infrastructures

<table>
<thead>
<tr>
<th></th>
<th>Corba</th>
<th>ToolTalk</th>
<th>PCTE</th>
<th>DPCL</th>
<th>DAMS</th>
<th>OCM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Monitoring Interface</strong></td>
<td>-</td>
<td>-</td>
<td>o</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Multiple Tools</strong></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Coordination</strong></td>
<td>(+)</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td><strong>Direct Interactions</strong></td>
<td>+</td>
<td>+</td>
<td>o</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Events</strong></td>
<td>o</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Actions</strong></td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Access Notification</strong></td>
<td>-</td>
<td>-</td>
<td>o</td>
<td>-</td>
<td>+</td>
<td>o</td>
</tr>
<tr>
<td><strong>Access Interception</strong></td>
<td>-</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
The Distributed Monitoring System OCM

Basic concepts:
- object model of the target system
- event / action paradigm
- services: detection of events, access to objects
- can be combined in conditional requests, e.g.:
  
  ```
  thread_has Been_stopped([p_1]) :
  thread_get backtrace([$thread], 0)
  ```

Implementation:
- distributed server processes with a hierarchical structure
- event detection and execution of actions are also performed in the context of the target processes
4. Implementation in the OCM (ctd.)

“Multi-user” Interface for Monitoring

- OCM operates as a server
- Each tool has its private “environment”
  - e.g. definition of the target system
- Synchronization:
  - single actions on a single object are atomic
  - atomic multicast communication
  - requests can be globally locked
4. Implementation in the OCM (ctd.)

Events and Actions in Other Tools

- OCM supports event detection / action execution in the context of target processes
- Available events / actions detected at run-time (incl. interfaces)
- Necessary extension
  - include tools into object model of target system
    - tool object is derived from process object
    - services on processes are now also available on tools
  - removes distinction between tools and monitored system
4. Implementation in the OCM (ctd.)

Monitoring of Object State Transitions

- Can be realized via event services
- Currently only special cases, e.g. `thread_hasBeenStopped`

Interception and Modification of Object Accesses

- Not yet available in the OCM

Common Feature

- Abstract: monitoring activities of the monitoring system
- Problems: implementation, appropriateness of interface
- More specific: detection of object accesses
Detection of Object Accesses

Goal:
- generic detection mechanism
- e.g. “p_1 is stopped”, “TID of p_2 is read”

Requirements:
- (unique) identification for object components
- components must be part of OMIS object model
- components must not overlap (i.e. no aliasing)
- protected access mechanism to object components
- ensure that all accesses are detected
- manipulation of objects only by writing components
4. Implementation in the OCM (ctd.)

Detection of Object Accesses (ctd.)

- Problems:
  - implicit change of object components in OS calls
  - dynamic extensibility of OMIS/OCM
  - efficient implementation

- Implementation:
  - object = dynamic array of pointers to static structures
  - access via C macros that include matching and notification
  - matching: compare with specifications in object class

- Overhead per access:
  - 6 memory reads, 3 conditional jumps (best case)
5. Conclusions and Future Work

- Combination of run-time tools is a promising way of implementing advanced tool environments

- Interoperability is impossible without a proper support environment (monitoring system)

- This is being more and more recognized
  - e.g. DPCL (IBM), DAMS (Univ. Lisbon, Portugal), ...

- Current work in OCM:
  - direct tool interactions
  - extended object model and detection of object accesses

- Future work: full implementation of the tool scenario