FIDE Tycoon Environment Demonstration

2nd Annual FIDE Review Meeting, Glasgow

J.W. Schmidt, F. Kirch, F. Matthes, S. Müssig
Hamburg University

September 1991

Objectives of the Demo

- In addition to traditional database system services (data storage, data sharing, access optimization and integrity management), novel demands of future database applications have to be addressed by FIDE languages and supported by their language processors:
  - Generic, reusable application software;
  - Safe, generic interfaces to services in open systems;
  - Non-standard data structures;
  - Non-standard execution paradigms (functions, methods, rules, triggers).
- The Tycoon project complements the DBPL project efforts by focussing on these novel, highly generic aspects of data-intensive applications. The first phase of the Tycoon project exploits existing state-of-the-art language technology to investigate

  Linguistic Support
  - types as first-class language entities
    (naming, kinding, parameterization, recursion, scoping)
  - module and binding mechanisms for database environments
  - higher-order functions
  - exception handling

  Add-On Library Support
  - user-defined types, type operators and kinds
  - iteration abstractions
  - generalized bulk structures
  - external call interfaces

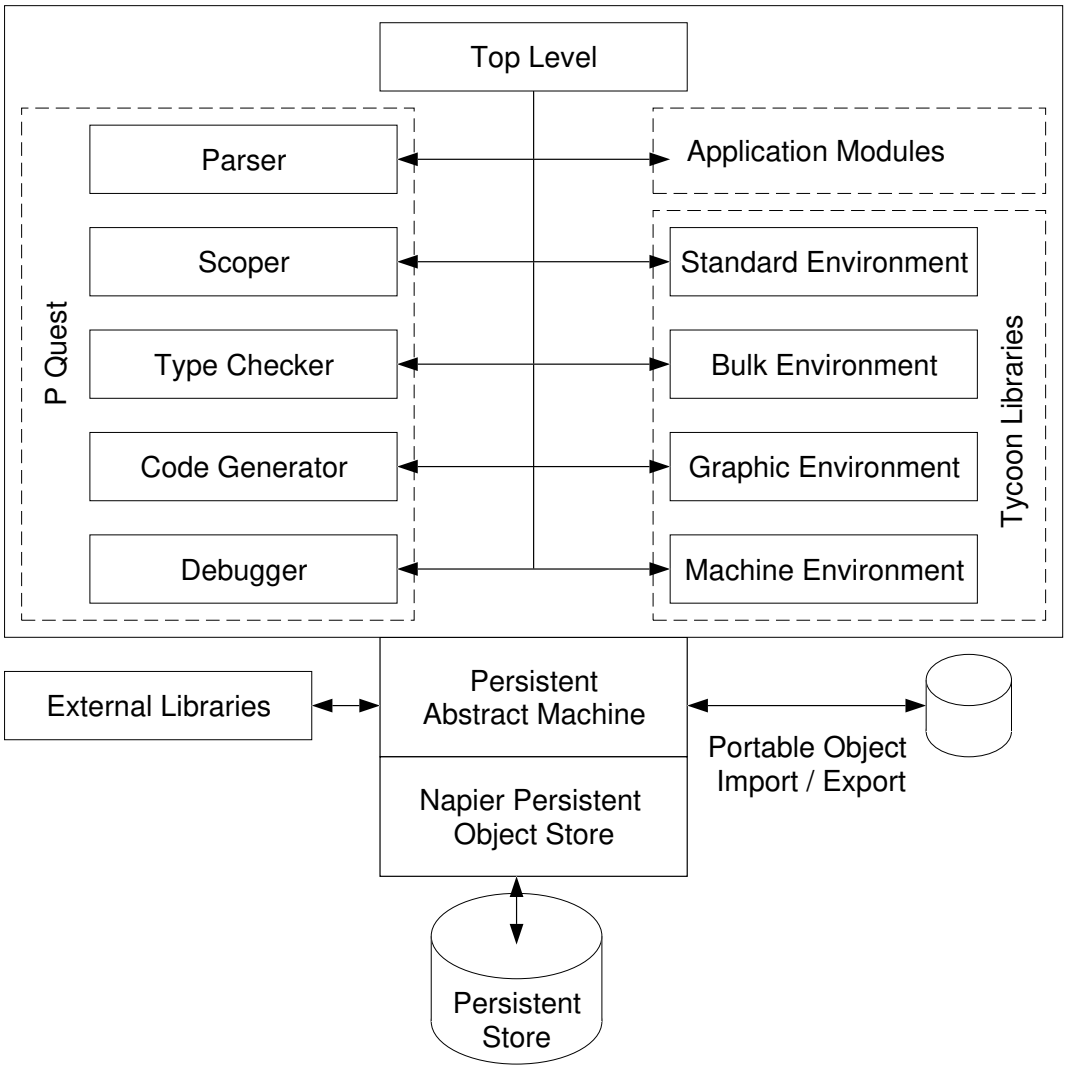
  Built-In System Support
  - persistence management and object identification
  - garbage collection
  - stability mechanisms
  - interactive interfaces and dynamic binding

Structure of the Demo

1. Tycoon Environment Overview
2. Strongly-Typed, Generic User Interface Construction
   (a) Display example data
   (b) Show database schema
   (c) Update, insert and delete example data
   (d) Instantiate a generic display routine
   (e) Extend the example display program
   (f) Introduce integrity checking code
3. Built-In Persistence Management
   (a) Checkpointing the database
   (b) Garbage collection
   (c) Program recovery

1 Tycoon Environment Overview

- Tycoon application programming environment:
  - standard environment (27 modules)
  - bulk environment (18 modules)
  - graphic environment (6 modules)
  - (machine environment (3 modules))
- Quest language processor: interactive top-level, compiler front-end, compiler back-end generating portable byte code, run-time debugger (19K lines Quest code);
- Portable persistent abstract machine (10K lines Modula-3 code);
- Persistent object store (C library provided by the University of St. Andrews).



Current Tycoon system architecture

2 Strongly-Typed, Generic User Interface Construction

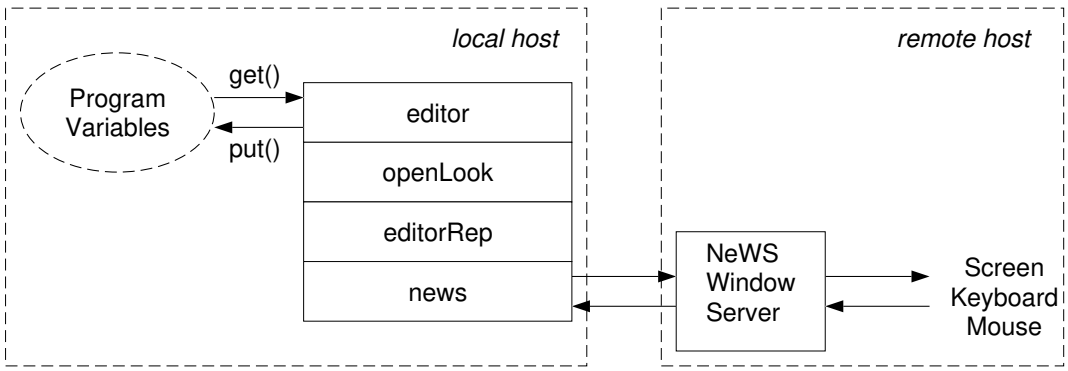
Task *Generic* tool support for the construction of typical database applications:

- Statically verified application consistency
  (database schema, program, user interface)
- Orthogonality
- "Universality"
- State-of-the-art functionality
- Extensibility

Approach A core graphical environment tailored to typical database application requirements consisting of four layers:

- Asynchronous communication with (remote) display server (module `news`);
- Access to classes, methods and display objects implementing the Sun NeWS *Open Look* user interface (module `openLook`);
- Internal, abstract representation of typed Tycoon language objects with their associated display attributes (module `editorRep`);
- External interface for the type-safe creation, display and manipulation of hierachically structured editor objects (module `editor`).

Test Data Data imported from the DBPL system demo.



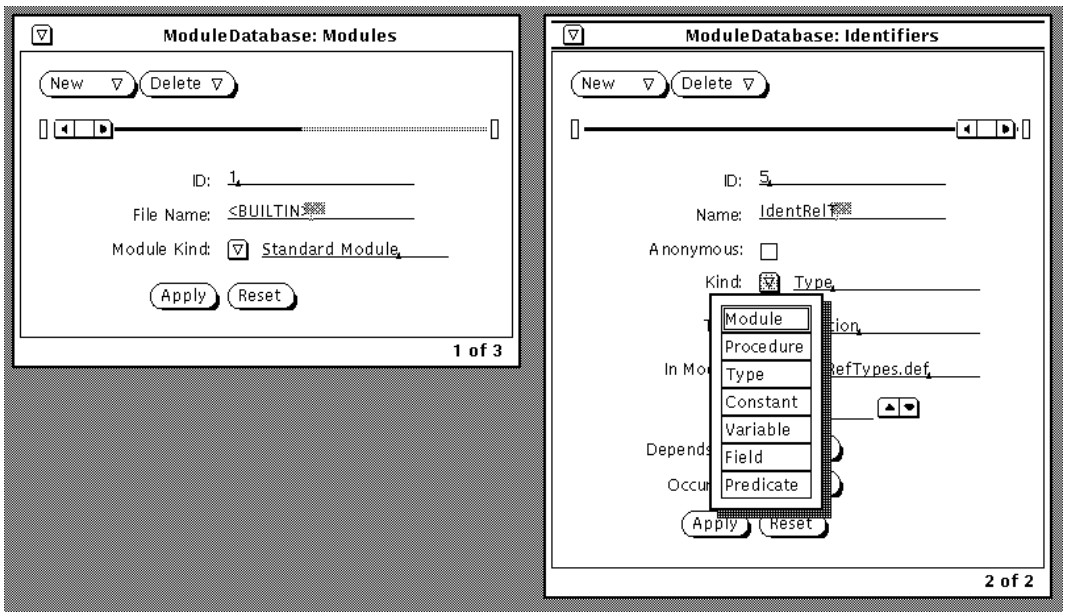
Framework for the interactive access to database variables

2.1 Display example data

After importing a pre-linked application module into the interactive top level environment and invoking it,

```
import moduleDemo :Main;
moduleDemo.go();
```

two new windows appear on the workstation screen that allow the user to browse through a database of module and identifier descriptions.[1] Text fields, numeric fields, checkboxes, buttons, drop-down menues and scrollbars provide type-specific display representations for hierarchically nested database objects.



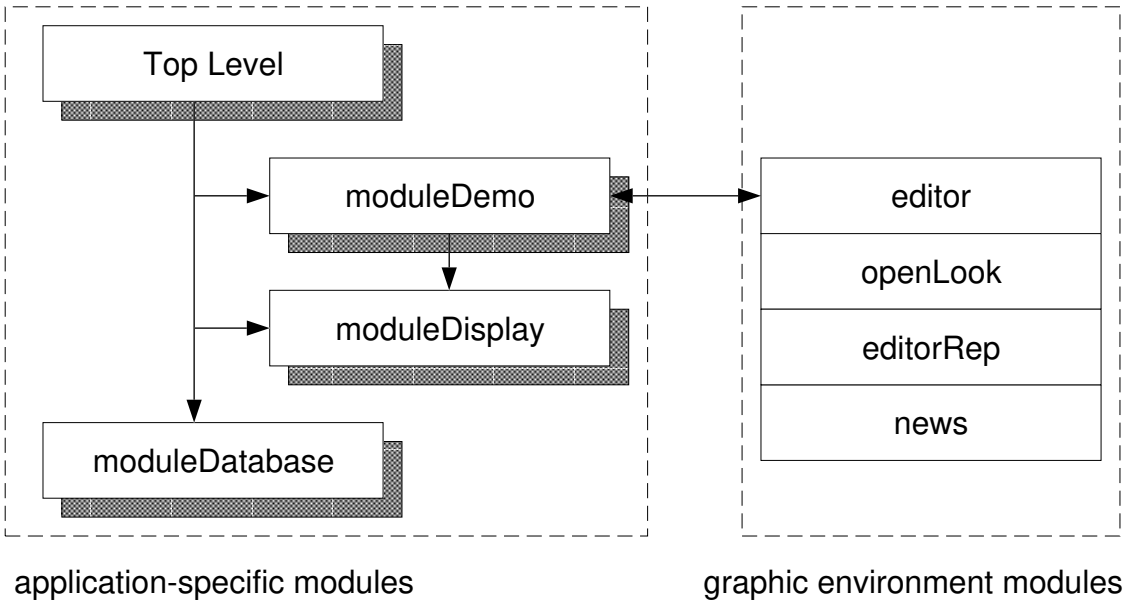
Browsing in an *Open Look* environment

Issues:

- Separation between re-usable generic functionality and application-specific functionality
- Support for complex objects
- Support for sets and lists
- Need for type abstraction
- Identity-based vs. value-based object identification

[1] The test database was initialized with data imported from the DBPL system demonstration.

2.2 Show database schema



Modules of the demo application

The interface `ModuleDatabase` exports global database types, views on the database variables and primitive update transactions.

```
interface ModuleDatabase
import
  iter :Iter
  optional :Optional
export
  Def IdKind = Option
    dbplModule, procedure, type, constant, variable, field, predicate
  end

  Def TypeKind = Option
    base, set, pointer, relation, procedure,
    enumeration, subrange, dbplArray, record, predicate, nil
  end

  Def ModuleKind = Option
    definition, implementation, main, standard
  end

  Def DeclarationKind = Option
    inModule, inRecord, inEnumeration
  end
```

The module database interface in file `ModuleDatabase.spec` (1)

```
(* --- Entity types: *)

  Def Module = Tuple
    id :Int
    var filename :String
    var kind :ModuleKind
  end
  Def Modules = Iter_T(Module)

  Def Occurence = Tuple
    var inModule :Module
    var inLine :Int
  end
  Def Occurences = Iter_T(Occurence)

  Def Rec Identifier = Tuple
    id :Int                 (* unique key issued by the DBPL compiler *)
    var name :String
    var anonymous :Bool
    var kind :IdKind
    var type :TypeKind      (* the type of a type identifier          *)
    var declModule :Module  (* compilation unit that exports the Id.  *)
    var declKind :DeclarationKind
    var declLine :Int       (* declaration line number                *)
    var scopeId :optional.T(Identifier)
                            (* identifier denoting the scope          *)
                            (* "nil" for modules                      *)
    var dependsOn :Identifiers
    var occurences :Occurences
  end
  and Identifiers = Iter_T(Identifier)

(* --- Database views: *)

  modules() :Modules
  identifiers() :Identifiers
  highestIdUsed() :Int
  standardModule :Module
  (* a pre-existing built-in module *)
```

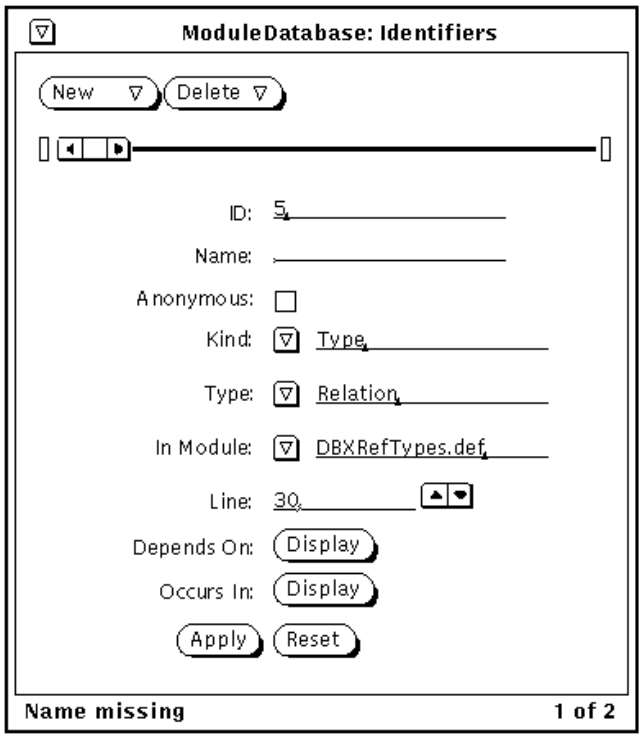
The module database interface in file `ModuleDatabase.spec` (2)

```
(* --- Basic transactions: *)

  error :Exception(String)
  (* raised when an integrity constraint is violated. *)

  setModules(:Modules) :Ok
  setIdentifiers(:Identifiers) :Ok
  addModule(:Module) :Ok
  addIdentifier(:Identifier) :Ok
  removeModule(:Module) :Ok
  removeIdentifier(:Identifier) :Ok
  newId() :Int
  (* issue a new, unique identifier *)
end;
```

The module database interface in file `ModuleDatabase.spec` (3)

2.3 Update, insert and delete example data

The user can use menues, buttons as well as cut and paste techniques to directly modify the database state. The particular implementation of the module database guarantees the uniqueness of the attribute `Id` in the set of modules and identifers, enforces domain constraints, rejects illegal "null values" and checks simple inter-attribute constraints.



Detecting a constraint violation ("Name missing")

Issues:

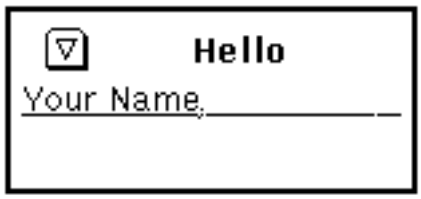
- Support for "early" database constraint validation:
  - attribute or domain constraints
  - collection constraints (e.g. key constraints)
  - subset constraints
- "Null values"

2.4 Instantiate a generic display routine

By executing the following program fragment, a simple string editor (variable `stringEd`) is created that is bound by its `put` and `get` routines to the global mutable variable `state.x`. As soon as the displayed editor window is closed by the user, the value of the variable `state.x` is returned.

```
import
  editor :Editor
  news :News;

let state = tuple let var x = ``Your Name'' end

let stringEd = editor.newString(
  let get = fun() state.x
  let put = fun(new :String) state.x:= new);

begin
  editor.display(stringEd ``Hello'')
  while editor.visible(stringEd) do
    news.waitEvent()
  end
  state.x
end;
```

Creation of a simple string editor



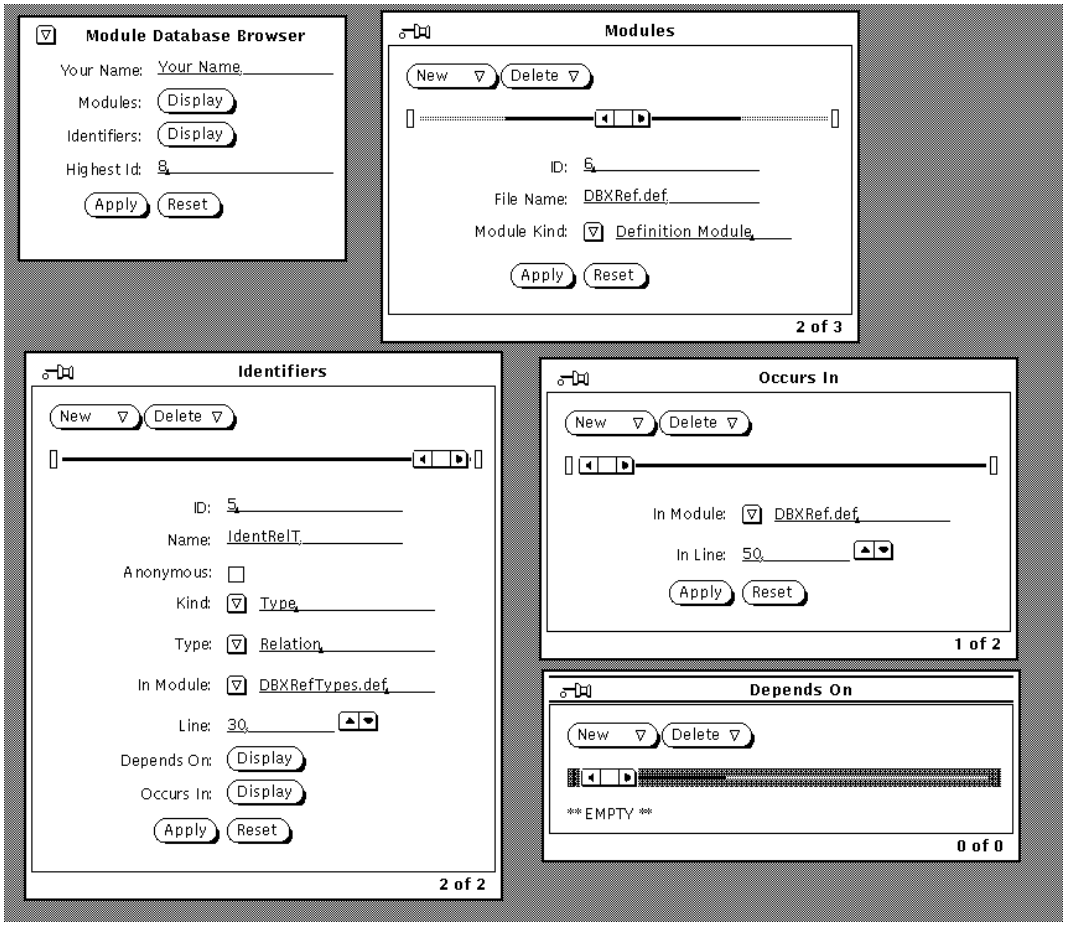
The string editor window

- Heavy use of pre-existing library functionality
- Programming with higher-order functions

2.5 Extend the example display program

Once a simple editor is created, it can be used as a component in a more complex editor. The following program fragment embeds the string editor together with two other editors imported from the module `moduleDisplay` and a simple string value into a record editor.

```
import
  moduleDisplay :ModuleDisplay
  moduleDatabase: ModuleDatabase;

let moduleEd = editor.newRecord of
  tuple "Your Name"   stringEd
  end
  tuple "Modules"     moduleDisplay.newModulesEditor(
                        moduleDatabase.modules
                        moduleDatabase.setModules)
  end
  tuple "Identifiers" moduleDisplay.newIdentifiersEditor(
                        moduleDatabase.identifiers
                        moduleDatabase.setIdentifiers)
  end
  tuple "Highest Id"  editor.newStringVal(
                        fun() fmt.int(moduleDatabase.highestIdUsed()))
  end
end;
```



The record editor window

2.6 Introduce integrity checking code

The definition of an editor for the `line` attribute of an identifier exemplifies how to check typical database integrity constraints as early as possible (i.e. at data-entry-time) by attaching them to the `put()` routine of an editor:

```
tuple
  "Line" editor.newInt(
       fun() i.x.declLine
       fun(new :Int)
         if new < 0 then
           raise editor.illegal with "Line must be positive" end
         else
           i.x.declLine:= new
         end)
end
```

Dependencies between multiple attributes of a record can also be enforced by attaching them to the `put` routine of the last attribute.

- Exception handling
- Exception typing
- Systematic approach to IC attachment

3 Built-In Persistence Management

3.1 Checkpointing the database

The lowest layer of the storage manager makes all updates to the store recoverable by transparently saving before-images of updated pages in the persistent store. The operation

```
store.stabilise();
```

exported by the module `store` that is part of the standard Tycoon environment atomically checkpoints the store. Thereby, all modifications to the store are committed and become persistent.

In the demo, this checkpointing is illustrated by undoing modifications performed via the interactive interface.

3.2 Garbage collection

The persistence model of PQuest is based on transitive reachability from the top-level interactive environment. There is no (unsafe) explicit object deletion mechanism. Garbage collection is either invoked explicitely,

```
store.garbageCollect();
```

or implicitely, as soon as the system runs out of (persistent) memory. Since the demo store has a fixed size of 5.7MB, the following program fragment triggers a garbage collection:

```
let createGarbage(count :Int) :Ok =
  (* create count arrays with 10000 integer elements *)
  for i= 1 upto count do
    let dummy = arrayOp.new(10000 0)
  end;

createGarbage(100);  (* allocate 4.000.000 bytes *)
createGarbage(100);  (* allocate 4.000.000 bytes *)
```

3.3 Program suspension

The P-Quest implementation provides a mechanism to suspend the execution of a program at arbitrary points in time and to return to the calling Unix process.

```
store.halt();
```

The command `PQuestRecover` restarts the P-Quest program with its original state (at the next statement following the `halt` function call).

The following program fragment makes use of this suspension mechanism:

```
let rec stopAfter(depth :Int) :Ok =
  if depth is 0 then
    store.halt()
  else
    print.string("Entering, depth = " <> fmt.int(depth) <> "\n")
    stopAfter(depth-1)
    print.string("Leaving, depth = " <> fmt.int(depth) <> "\n")
  end;

stopAfter(5);
..let stopAfter : All(depth:Int) Ok = <fun stopAfter()>
Entering, depth = 5
Entering, depth = 4
Entering, depth = 3
Entering, depth = 2
Entering, depth = 1
Exception:
florian@dbis1> PQuestRecover
                                        ---
     -----    ----- -- -- ----- -----|   |_
    |     |__|     |  |  |  _  |   __|    _|
    |  |  |__|  |  |     |   __|__   |   |
    |   __|  |__   |_____|_____|_____|___|
    |__|        |__|

    Hamburg University
    Tycoon Language Environment Release 1.1

    Restart from last checkpoint ...

Leaving, depth = 1
Leaving, depth = 2
Leaving, depth = 3
Leaving, depth = 4
Leaving, depth = 5
```

Resuming a suspended program execution