### Table of Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Optimizations</td>
<td>3</td>
</tr>
<tr>
<td>Fast Searching</td>
<td>7</td>
</tr>
<tr>
<td>Recursive Functions</td>
<td>11</td>
</tr>
<tr>
<td>Using the Profiler</td>
<td>14</td>
</tr>
<tr>
<td>Outdated Functions</td>
<td>16</td>
</tr>
</tbody>
</table>
Simple Optimizations

Declare the data type of local variables

If you do not explicitly declare the data type of a local variable, the data type usually will be any. Sometimes the SimTalk compiler will determine the data type implicitly, for example when assigning a constant value in the declaration.

Not using the data type any has these advantages:

• The SimTalk compiler can better optimize the code (for example perform type checks during compile time rather than during execution).
• It increases the chance to find certain programming errors during compile time.
• It may also help to understand the source code.

Simple Optimizations

Compute loop-invariant values outside of loops

If the value of a complex expression will not change inside a loop, assign it to a local variable.

Example:

```simtalk
for var x := 1 to Store.ContentsList.XDim
    for var y := 1 to Store.ContentsList.YDim
        if Store.ContentsList[x,y] /= void and Store.ContentsList[x,y].Name = "Container"
            return Store.ContentsList[x,y]
        end
    next
next
```
Simple Optimizations

Simplify expressions in waituntil-statements

An expression in a waituntil-/stopuntil-statement will possibly be evaluated many times before the expression becomes true. If you know that a part of the expression will not change, assign it to a local variable.

Example:

```plaintext
waituntil ?.cont.finished -- ?.cont might be evaluated many times

var contObj : object := ?.cont -- ?.cont is evaluated one time only
waituntil contObj.finished
```

Simple Optimizations

Returning multiple results

If you need to return multiple result values, return them in `byref` parameters instead of returning them in global variables or in an array.

Example:

```plaintext
param dividend, divisor: integer -> integer[2]  
return makeArray(dividend div divisor, dividend mod divisor)

param dividend, divisor: integer, byref quotient, remainder: integer
quotient := dividend div divisor
remainder := dividend mod divisor
```

This improves:
- Readability
- Performance
Fast Searching

Searching values in tables

- Enable the row index and put the values to search for into column 0.
- Possibly enable Fast index access.
- Use getRowNo to find the row.

Example:

```
var row := PersonTable.getRowNo("Lisa")
```

Effect of Fast index access:
- Searching is very fast
- Inserting and deleting a row is slow

Hashing (1)

- When Fast index access is disabled, searching is slow (particularly when searching for string values).
- If you need to search for more than one value, using the row index is impossible.

Solution: Hashing

Divide the search space into smaller partitions.
Compute a hash function for the search value that identifies the partition to look into.

Example:

A filing cabinet containing files of persons. Each file is put into the drawer with the first letter of the last name.
When looking for a person, you need to find the correct drawer and then you need to search inside this drawer only.
Hashing (2)

Properties of a good hash function:
- deterministic
- computes fast
- spreads well

Example: Look for a combination of a first name and a last name.

Possible hash function:
```plaintext
var HashValue: integer := (ascii(FirstName)*29 + ascii(LastName)) mod HashTable.XDim + 1
```

Alternative hash function:
```plaintext
var s: string := to_str(FirstName, ";", SecondName)
var HashValue: integer := strHash(s) mod HashTable.XDim + 1
```

Hashing (3)

Resolve collisions

“In a filing cabinet multiple files are stored in the same drawer.”

Example code:
```plaintext
param FirstName, SecondName : string -> integer

var s : string := to_str(FirstName, ";", SecondName)
var HashValue: integer := strHash(s) mod 13 + 1

HashTable.setCursor(1, 1)
if HashTable.find({HashValue, 1}..{HashValue, *}, s)
    return HashTab[HashValue+13, HashTab.CursorY]
else
    return 0
end
```
Recursive Functions (1)

Recursion in mathematics

\[ 0! = 1 \]
\[ n! = n \cdot (n - 1)! \]

Recursive SimTalk Methods

```
-- factorial(n)
param n : integer -> integer
if n = 0
  return 1
else
  return n * factorial(n - 1)
end
```

Principle idea

Simplify a problem by dividing it into sub-problems of the same type. This is called divide and conquer. It is a top-down approach, where problems are solved by solving smaller and smaller instances.

Recursive Functions (2)

Traversing a tree structure

A tree structure has a root node, and each node can have several child nodes.

If you want to visit all nodes in a tree, you can easily visit them recursively.

Visiting all objects in a simulation model:

```
param frame : object
for var i := 1 to frame.numNodes
  var obj: object := frame.node(i)
  if obj.InternalClassName = "Network"
    self.execute(obj)
  else
    -- do something with 'obj'
  end
next
```
Recursive Functions (3)

Other example for “divide and conquer”: *Towers of Hanoi*

-- moveStack(5, A, C)

param NumDiscs: integer, FromStack, ToStack: object

if NumDiscs = 1
    ToStack.push(FromStack.pop)
else
    var Other: object := getOtherStack(FromStack, ToStack)
    moveStack(numDiscs - 1, FromStack, Other)
    ToStack.push(FromStack.pop)
    moveStack(numDiscs - 1, Other, ToStack)
end

Using the Profiler (1)

Optimize the *right* methods

Example:
Method1 consumes 5% of the CPU time.
Method2 consumes 40% of the CPU time.

Which optimization is better?
- speed up Method1 by 100 times
- speed up Method2 by 30%

Answer:
- Method1: 5% * 0.99 = 4.95% less total CPU time
- Method2: 30% * 0.3 = 9% less total CPU time

Conclusion:
We need to find the Methods that consume a relevant amount of time.
**Using the Profiler (2)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Total number of calls</th>
<th>mean time a single call consumed</th>
<th>time consumed by all instances (click to expand)</th>
<th>time portions consumed by individual instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Models.Frame.SankeyDiagram1.printTrace</td>
<td>146.63s</td>
<td>96.93s</td>
<td>146.63s</td>
<td>146.63s</td>
</tr>
<tr>
<td>Tools.SankeyDiagram.SankeyDiagram.printTrace</td>
<td>60.35s</td>
<td>9.23s</td>
<td>60.35s</td>
<td>60.35s</td>
</tr>
<tr>
<td>Tools.SankeyDiagram.SankeyDiagram.logMove</td>
<td>1.11s</td>
<td>1.11s</td>
<td>1.11s</td>
<td>1.11s</td>
</tr>
<tr>
<td>Tools.SankeyDiagram.SankeyDiagram.logActivityData</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Tools.SankeyDiagram.SankeyDiagram.printGraphicalIndex</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Tools.SankeyDiagram.SankeyDiagram.logActivityData</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Tools.SankeyDiagram.SankeyDiagram.logActivityData</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Tools.SankeyDiagram.SankeyDiagram.logActivityData</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Tools.SankeyDiagram.SankeyDiagram.logActivityData</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
</tbody>
</table>

---

**Outdated Functions**

1. Look for no longer supported functions and methods in the source code

2. Double-click one of the outdated functions

3. Replace the outdated name with the new name (click “Find Next” to open the Method Editor)
Contact

Michael Joos
Team Leader and Software Engineer
Tecnomatix Plant Simulation
Weissacher Str. 11
70499 Stuttgart
E-Mail: michael.joos@siemens.com

www.siemens.com/plm/plantsimulation