**University of Amsterdam**
Programming Research Group

# A Toolset for PGA

B. Diertens

Report PRG0302 October 2003

B. Diertens

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7593
e-mail: bobd@science.uva.nl

Programming Research Group Electronic Report Series

# A Toolset for PGA

*Bob Diertens**

University of Amsterdam

*ABSTRACT*

We give a description of the implementation of a toolset for Program Algebra (PGA). The toolset is intended to be used in education and further research. The latter requires easy extendibility in order to support new program notations.

*Keywords:* program algebra, software engineering

## 1. Introduction

In this paper we describe the development of a toolset for Program Algebra (PGA).[1] The toolset is intended to be used in education and further research. In the following sections we describe tools such as a generic parser, a generic simulator, a parallel simulator, and a bisimulator. We also give an overview of the available primitive and basic instruction sets in the toolset. We end this paper with some concluding remarks.
In the remainder of this section we give a description of PGA and describe what our goals are for the toolset.

### *1.1 Program Algebra*

PGA is an algebraic framework for sequential programming. It is intended to contribute to a better understanding of sequential programming. A very simple program notation is used as basis for development of other program notations.

The syntax of program expressions in PGA is generated from a set of constants, the primitive instructions, and two composition mechanisms. The primitive instructions have as parameter a set of basic instructions. These basic instructions can be viewed as requests to an environment to provide some service. Upon execution, a basic instructions returns a boolean value.

Primitive instructions:

a basic instruction: $a$

After performing the basic instruction $a$ execution continues with the next instruction.

termination: $!$

Execution stops.

positive test: $+a$

If the basic instruction $a$ returns true, execution is continued with the next instruction. If it returns false, the next instruction is skipped.

negative test: $-a$

If the basic instruction $a$ returns false, execution is continued with the next instruction. If it returns true, the next instruction is skipped.

forward jump: $\#k$

The instruction itself and the following $k - 1$ instructions are skipped. If $k$ is $0$, a jump to the instruction itself is made.

---

* *E-mail* address: bobd@science.uva.nl

1. website: www.science.uva.nl/research/prog/projects/pga/

Compositions:
concatenation of X and Y: X;Y
repetition of X: $X^{\omega}$

PGLA is a program notation for representing PGA expressions. For dealing with repetition, PGLA has an additional primitive instruction, which replaces repetition.

repeat: \\#*n*
Here, *n* is a natural number greater than zero. A program text ending with this instruction will repeat its last *n* instructions, excluding the repeat instruction itself.

On top of PGLA, other program notations are designed, which can be mapped on PGLA. Such a mapping towards PGLA is called a projection. Several projections to intermediate languages may be used to get the result that is needed. A mapping away from PGLA is called an embedding.

### *1.2 Program notations*

In Figure 1 we show the hierarchy of program notations described in publications related to PGA. The arrows indicate an available projection/embedding.

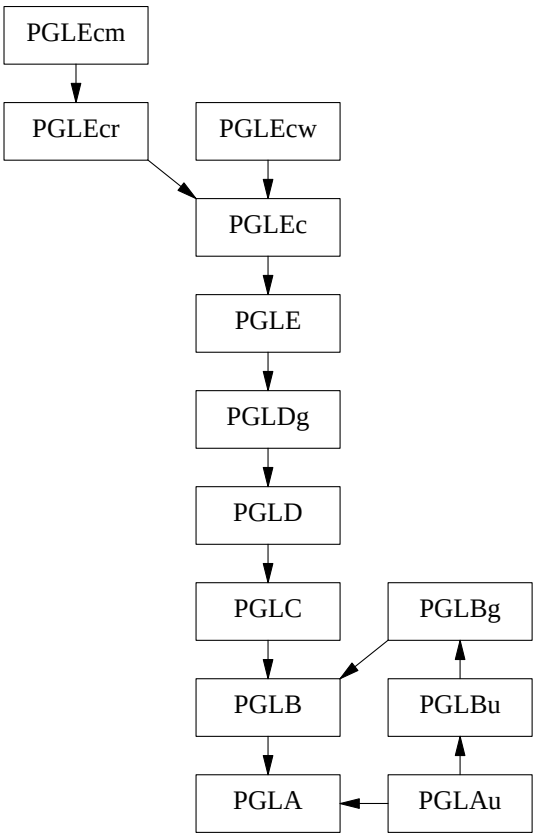


**Figure 1.** Hierarchy of program notations

In [9] the instruction sets PGLAu, PGLBu, and PGLBg are described along with the projections/embeddings pglau2pglbu, pglbu2pglbg, pglbg2pglb. The projection pglau2pgla is a composition of the four mappings pglau2pglbu, pglbu2pglbg, pglbg2pglb, and pglb2pgla.

The program notations PGLEcm and PGLEcr are introduced in [2]. PGLEcr extends PGLEc with recursion and PGLEcm is an extension of PGLEc with method calls. The projections pglecm2pglecr and pglecr2pglec must be used in combination with each other and rely on the basic instruction set MPP (see section 5.1).

The other program notations and projections are all described in [3].

### *1.3 Toolset*

The toolset should at least contain the projections and embeddings mentioned in the previous section, and the basic instruction sets described in [4], and [2] We also like to test our programs by means of simulation.

Since the toolset will be used in further research, it is likely that new program notations will be designed for which we want support from the toolset. So, it must be easy to extend our toolset.
Adding a new program notation means that we have to build parser routines. We like to be able to do this in a very easy way. This can be made possible by the right support, but also by keeping our program notations simple.
It should also be possible to run the tools on many different platforms.

As implementation language we chose Perl [10] for its regular expression and ease of manipulation. For graphical views we chose the language Tcl/Tk [7]. Tk is also available as a package for Perl, but we want to keep the code for the graphical views separated from the other code.
Perl and Tcl/Tk are both available for a variety of platforms, what makes the toolset portable.

## 2. Parsing

A language consist of a primitive instruction set, with as parameter a basic instruction set, and compositional constructs. Parsing of such a language should therefore consist of decomposing the input (program) into primitive instructions, parsing of these primitive instructions, in which parsing is passed on for the parts of the primitive instructions that are basic instructions.

We model this scheme as a module that splits the input into primitive instructions, a module that parses the instructions it gets from the input module, and passes basic instructions on to another module that will parse them.

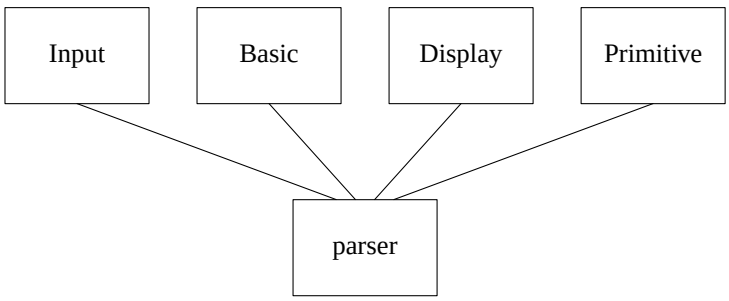


**Figure 2.** Import structure of parser

Figure 2 shows the import relation of the modules. The module Display is used to route all output through one module. The modules Input, Basic, and Display are actually used as a parameter to the initialization method of module Primitive. The relation between the modules is better shown by the use of methods of the modules (Figure 3).

The result will be stored in memory in the form of a list of instructions. The instructions are represented as a list consisting of an opcode and its arguments. Depending on the primitive instruction, a range of these arguments may represent a basic instruction in the form of an opcode and arguments.
The choice for lists is given by the ease of manipulation of lists in our implementation language (Perl).

Parsing of a program also includes some static checking and resolving of language constructs in which more than one instruction is involved, such as the existence of a label used in a goto, matching ’if’ and ’endif’, etc. So we extend our Primitive module with a ’Check’ method. The basic module does not need such a method, because there is no need for this with the basic instruction sets we deal with at the moment. To see what actually has been parsed the Primitive and Basic module are provided with a ’Print’ method.

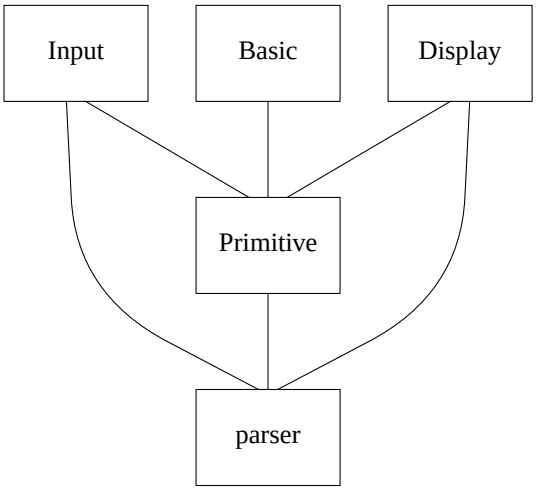


**Figure 3.** Functional model of parser

Basically, a parser is described by the following Perl-script.

```
open(FH, "$ARGV[0]") || die "could not open $ARGV[0]\n";
Input::Stream(\*FH);
Primitive->Init(\&Input::Next, 'Basic', 'Display');
$errors = Primitive->Parse();
print "parse errors: $errors\n";
Primitive->Check();
Primitive->Print();
```

Instead of making a parser for every combination of primitive and basic instruction set we use, we can make a generic parser that depending on the commandline options given to it, loads the appropriate primitive and basic instruction sets.
If the Perl-variables $primitive and $basic contain the names of the instructions sets, we can load and use them as follows.

```
if (! eval {require "$primitive.pm"}) {
    die "module $language not found\n";
}
if (! eval {require "$basic.pm"}) {
    die "module $basic not found\n";
}
open(FH, "$ARGV[0]") || die "could not open $ARGV[0]\n";
Input::Stream(\*FH);
$primitive->Init(\&Input::Next, $basic, 'Display');
$errors = $primitive->Parse();
print "parse errors: $errors\n";
$primitive->Check();
$primitive->Print();
```

### 3. Projections

We can model a projection from language B to language A by parsing a program in language B, convert its instruction to instructions of language A, and print the result with the printing method of language A (Figure 4). An embedding can be modelled in the same way.

As basic instruction set we use Generic, which accepts any string as a basic instruction, since most projections only need to convert the primitive instructions.

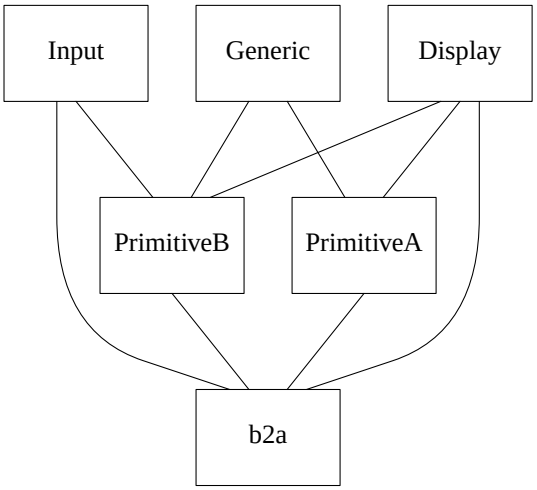


**Figure 4.** Model of a projection

Our projections look roughly like the following Perl-script.

```
PrimitiveB->Init(\&Input::Next, 'Generic', 'Display');
PrimitiveA->Init(\&Input::Next, 'Generic', 'Display');

Input::Stream(\*STDIN);
PrimitiveB->Parse();
PrimitiveB->Check();

$src = PrimitiveB->Mem();
$dest = PrimitiveA->Mem();

for ($ic = 0; $ic <= $#{$src}; $ic ++) {
    @instr = @{$$src[$ic]};
    $opcode = shift @instr;
    if ($opcode eq '...') {
        ...
    } elsif ($opcode eq '...') {
        ...
    } else {
        $$dest[$ic] = [( $opcode, @instr )];
    }
}

PrimitiveA->Print();
```

The `if` and `elsif` (zero or more) parts should contain the code for converting instructions with that particular opcode. The `else` part deals with all other instructions and just copies the original instruction.

The above code represents a simple projection. If there is no one-to-one mapping of instructions possible or if there is information needed from other instructions, the code becomes more complicated.

Our projections are modelled as filters (reading from standard input, and writing to standard output) so they can be used in a pipeline.[2] For example:

```
d2c < x.d | c2b | b2a > x.a
```

2. In Unix systems, a pipeline is a way to communicate between two processes.

Invoking a command consisting of a large number of projections is a pain. Therefore, the toolset contains a program **project** that has some knowledge about the hierarchy of the available projections, and invokes the right command. For example, the command

```
project pgld pgla
```

invokes the command

```
cat | pgld2pglc | pglc2pglb | pglb2pgla
```

**4. Simulation**

Here, we describe the implementation model for simulating PGA programs. We extend the modules for the primitive and basic instruction sets with methods for executions of these instructions. Execution of a primitive instruction may result in passing a basic instruction to the Basic module. Execution of this basic instruction results in a boolean value upon which the primitive instruction can act.

In the same way we made a generic parser, we can make a generic simulator. The model of the generic simulator is shown in Figure 5.

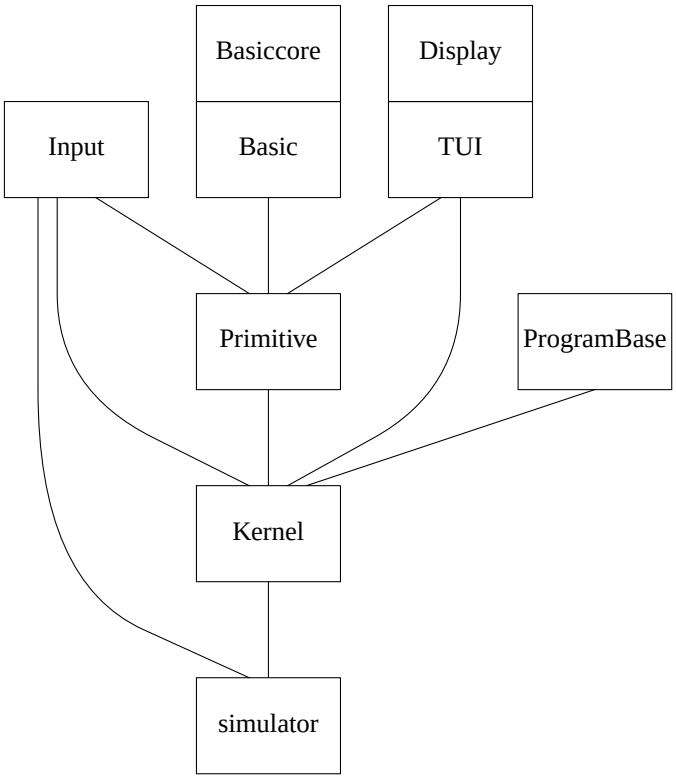


**Figure 5.** Model of the simulator

The module Basiccore is inherited by the module Basic, and contains the methods for execution of the basic instructions. The usefulness of inheritance instead of incorporation comes to light when we discuss the implementation of a parallel simulator (see Section 6). The description of the core methods for several basic instruction sets can be found in Section 5.

The module Display is inherited by module TUI (textual user interface), which is now passed as parameter of the initialization method of module Primitive instead of module Display. The module Kernel gets the commands given by the user from the textual user interface and acts upon it.

Several programs can be loaded into the simulator. Storage and switching between them is taken care of by the module ProgramBase.

The textual user interface of the generic simulator has the following commands:

| | |
|---|---|
| **l** *file* | load program from *file* and make it the current one |
| **l** | list loaded programs |
| **lc** *name* | make program *name* the current if previously loaded |
| **list** | list current program |
| **r** | run current program (stopped by Ctrl-C) |
| **r** *name* | make program *name* the current if previously loaded, and run it |
| **rr** | rerun - reset and run |
| **s** | step - execute one instruction |
| **e** *action* | execute *action* |
| **reset** | reset program counter |
| **t +/-** | trace on/off |
| **b +/-** *n* | set/unset breakpoint on line *n* |
| **b** | list breakpoints |
| **b c** | clear all breakpoints |
| **i** | initialize core |
| **d** | dump core |
| **u +/-** | enables/disables updating of core during a command |
| **q** | quit |
| **?** | displays the above list of commands |

The first three instructions make it possible to load more than one program and activate them in varying order. A collection of programs used in this way is also called a program family.[3]
The commands i, d, and u work on the core of the basic instruction set, provided that the implementation of the specific basic instruction set supports them. Within the commands t, b, and u, spaces are optional.

A session with the simulator is shown below.

```
% gensim -P PGLEc -l list.pglec
Parsing list.pglec ...
 0 1 2 3 4 5 6
Done. (errors: 0)
Checking ...
Done.
-> b+ 6
-> t+
trace on
-> r
 0: x = new    => T
 1: L0
 2: y = new    => T
 3: y.+a    => T
 4: y.a = x    => T
 5: x = y    => T
break (6)
-> q
%
```

Instead of the textual user interface, the generic simulator can be started with a graphical user interface by using a commandline option. In our model for the simulator the TUI module (and Display module) is replaced by a model for the graphical user interface shown in Figure 6. The module GUI contains the alternatives for the methods of module Display, which communicate with the **gui** program over pipelines.

---

3. A description of program families can be found in [2].

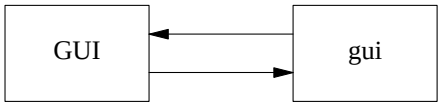


**Figure 6.** Model of the graphical user interface of the simulator

The graphical user interface, which is implemented in Tcl/Tk, is shown in Figure 7. The state of the interface is the result of the same session as with the textual user interface.

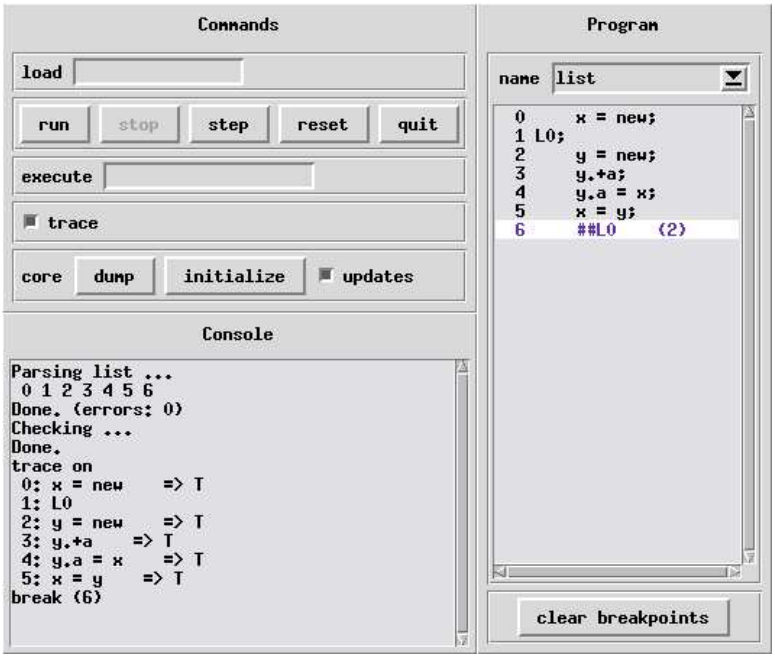


**Figure 7.** Graphical user interface

The buttons and entry fields represent commands that are described with the textual user interface. A listing of the current program is always provided in the Program window. The selection of the current program that has been previously loaded can be done in this window too. Breakpoints can be set and unset by clicking on a line in the listing of the program. Instructions on which a breakpoint is set are indicated with a different color of the background. The next instruction to be executed is indicated with a different color of the foreground of the instruction.

## 5. Basic instruction sets

We describe two types of basic instruction sets available with the toolset. The first type is based on *molecular dynamics* and the second on combining programs and state machines.

### *5.1 Molecular Dynamics*

In molecular dynamics [2], the memory state of a system is modeled as a fluid consisting of a collection of atoms which may have bindings between them to form molecules. A molecule consists of a number of atoms all reachable from one of the atoms by sequences of directed links. A directed link from one atom to another atom exists if the former has a so-called *field* containing the latter. By means of actions causing a change of state, fields can be added to and withdrawn from atoms, and contents of fields can be modified. Selected atoms can also be brought into *focus*.
The basic instruction set is called MPP (Molecular Programming Primitives). There is also a version with

values (MPPV) and a highlevel implementation in which all kinds of combination of instructions are possible (HMPPV). All three have been extended with garbage collection instruction as described in [2].

*5.1.1 Implementation of MPP*

The implementation model of MPP is given in Figure 8.

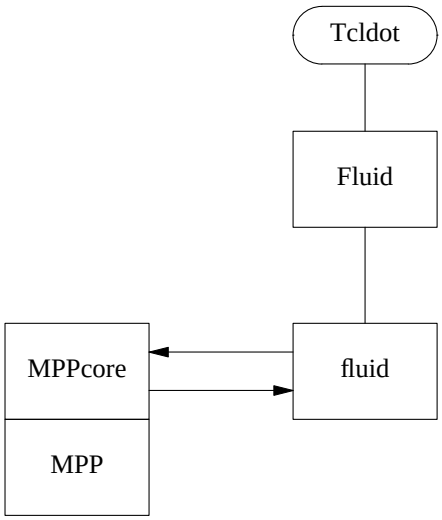


**Figure 8.** Model of MPP

The module MPPcore communicates with the **fluid** program, which is implemented in Tcl/Tk and shows a graphical view of the fluid. The module Fluid of this program makes use of a library Tcldot[4] for the layout of the graphs that represent the molecules. An example of a graphical view is shown in Figure 9.

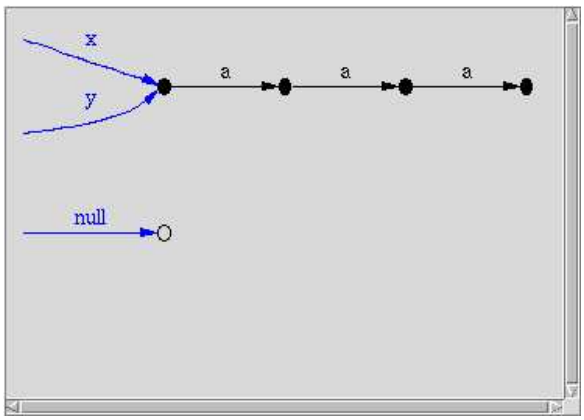


**Figure 9.** Graphical view of fluid

*5.2 Combining programs and state machines*

In [4] the interaction of programs and state machines is introduced. To interact with state machines, the basic instruction set Focus Method Notation (FMN), as described in [3], is provided. FMN instructions have a focus, which represents a part of the system able to process a basic instruction and to respond with a boolean value. Such a part is called a coprogram or state machine. A FMN instruction consist further of a

4. Tcldot is part of the software package Graphviz from AT&T Bell Laboratories. website: www.research.att.com/sw/tools/graphviz/

method separated from the focus by a '.'. FMN contains two types of instructions:

method call: `f.a:x`
: Where `f` is the focus of the instruction, and `a:x` its co-instruction part (or co-action). It returns the value replied by the co-instruction.

instance method call: `f:i.a:x`
: Here, `i` is the instance of the focus `f` of which the co-instruction part will be performed. So, it indicates a particular copy of the coprogram. It returns the value replied by the co-instruction.

Co-actions may also be written as `a(x)`.

*5.2.1 Implemented state machines*

We present here the state machines that currently are provided with the toolset. Where not mentioned, the co-actions return the value `true`.

**Console**
: Co-actions: `println(`*`s`*`)` in which *`s`* is a string of characters
The action `println` prints its argument.

**smbr** − boolean register
: Co-actions: `set(true)`, `set(false)`, `eq(true)`, `eq(false)`
The co-actions `set(true)` and `set(false)` reply `true`. The co-actions `eq(true)` and `eq(false)` reply `false` until the register is set, after which `eq(`*`b`*`)` replies `true` whenever *`b`* is the current value of the register, and `false` otherwise.

**smnnr** − natural number register
: Co-actions: `set(`*`i`*`)` , `eq(`*`i`*`)` in which *`i`* is a natural number
The replies of the co-actions are similar to that of **smbr**.

**smnnc** − natural number counter
: Co-actions: `succ()`, `pred()`, `isZero()`
The counter always starts with value zero, and the action `pred()` leaves the counter at zero if it is zero.

**smnns** − natural number stack
: Co-action: `push(`*`i`*`)`, `pop()`, `topEq(`*`i`*`)` in which *`i`* is a natural number
The stack always starts empty, and the action `pop()` leaves the empty stack empty.

For example, consider the following program, where PGLEc is used as primitive instruction set.

```
smbr:x.set(true);
smbr:y.set(false);
+ smbr:x.eq(true) {;
    Console.println(x is true);
}{;
    Console.println(x is false);
};
+ smbr:y.eq(true) {;
    Console.println(y is true);
}{;
    Console.println(y is false);
}
```

Upon execution the result will be:

```
x is true
y is false
```

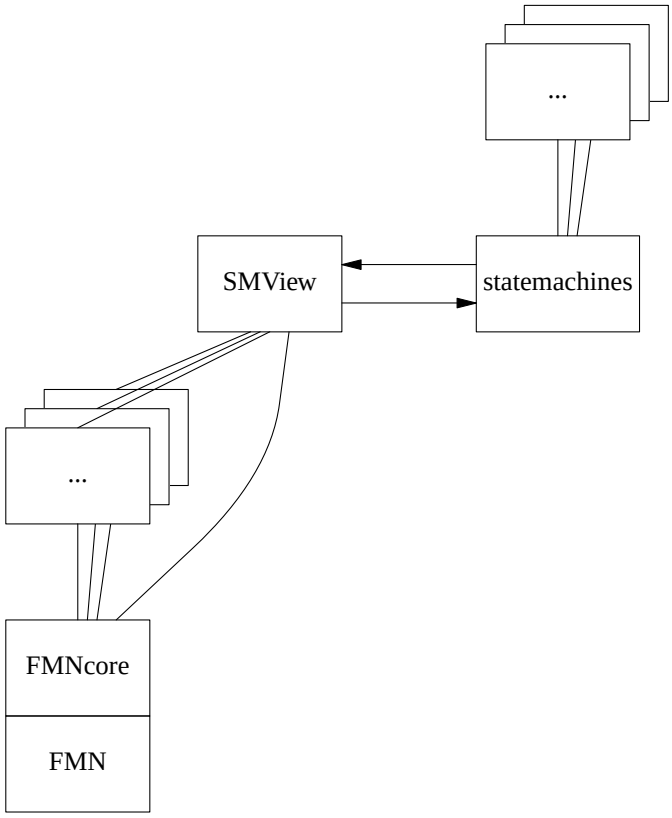


**Figure 10.** Model of FMN

*5.2.2 Implementation of FMN*

In Figure 10 the implementation model of FMN is given. A state machine module is imported by the FMNcore module the moment it is needed for the first time. An instance of a state machine is made by calling the method **new**.

For a graphical view of a state machine communication through the module SMView can be done. The module SMView communicates over pipelines with the Tcl/Tk program **statemachines**. This program interpretes commands it recieves from SMView and imports the necessary Tcl/Tk module for a graphical view of the state machine if not already loaded, and invokes the right method according to the command it received. It is not necessary for a state machine to have a graphical counterpart. A graphical view of the state machines for the above program can be seen in Figure 11.

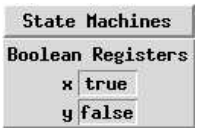


**Figure 11.** Graphical view of state machines

## 6. Parallel Simulator

The parallel simulator is a controller that can start several simulators that all work on one core of the basic instruction set. It is implemented with the use of the ToolBus.[5] An implementation model of the parallel simulator is given in Figure 12.

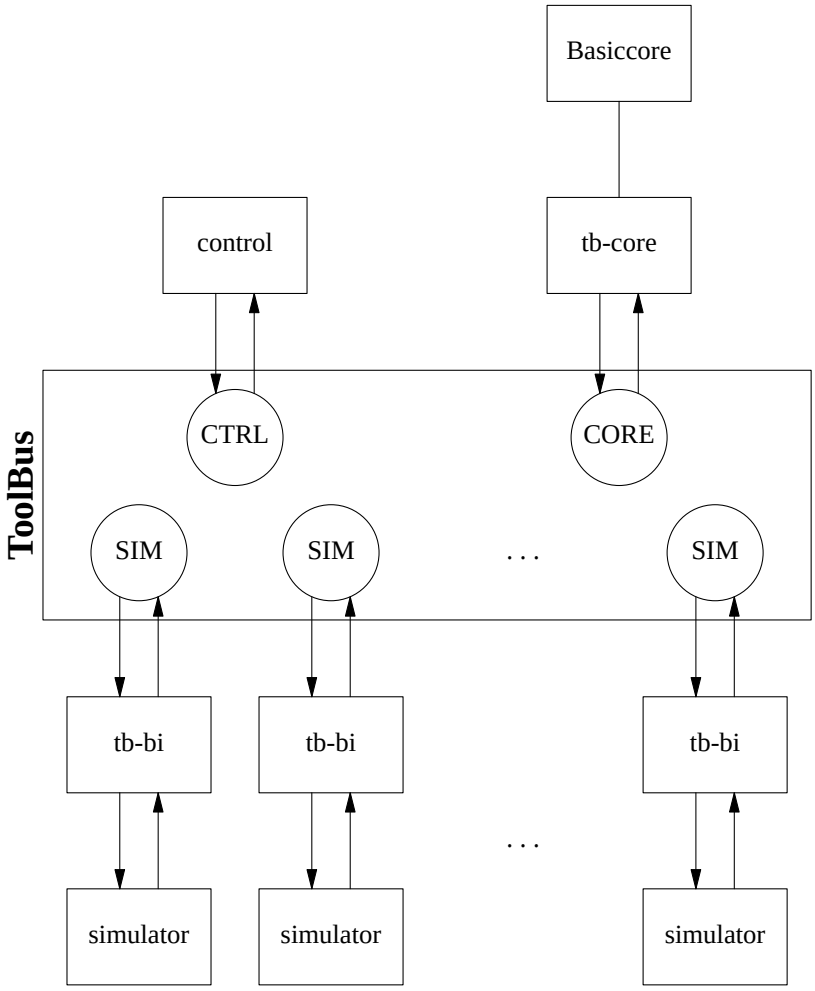


**Figure 12.** Model of the parallel simulator

The **control** program (see Figure 13) can start new simulators for a given primitive instruction set and program, and with the basic instruction set given at the start of the parallel simulator. It also provides the possibility to quit the parallel simulator as a whole. Each simulator can quit itself, but the operations that work on the core (dump, initialize, updates) are blocked. These facilities are now available in the **control** program.

Instead of importing the Basiccore module, the Basic module of a simulator now imports a module that communicates with the **tb-bi** program. The Basiccore module is imported by the **tb-core** program, which communicates with the ToolBus and invokes the right methods from the Basiccore module.

## 7. Bisimulation

For testing the equaltity of the behavior of two programs, a bisimulator has been implemented. But in order to test the bisimularity the two programs have to be converted to a labelled transition system (LTS). We describe here some implementation details.

---

5. The ToolBus is a software application architecture developed at the University of Amsterdam [5]. It utilizes a scripting language based on process algebra [1] to describe the communication between software tools. A ToolBus script describes a number of processes that can communicate with each other and with tools living outside the ToolBus.
website: www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ToolBus


**Figure 13.** Control window of the parallel simulator

*7.1 From program to labelled transition system*

As with the parser and the simulator, the program to LTS converter is also generic. The implementation model is shown in Figure 14.

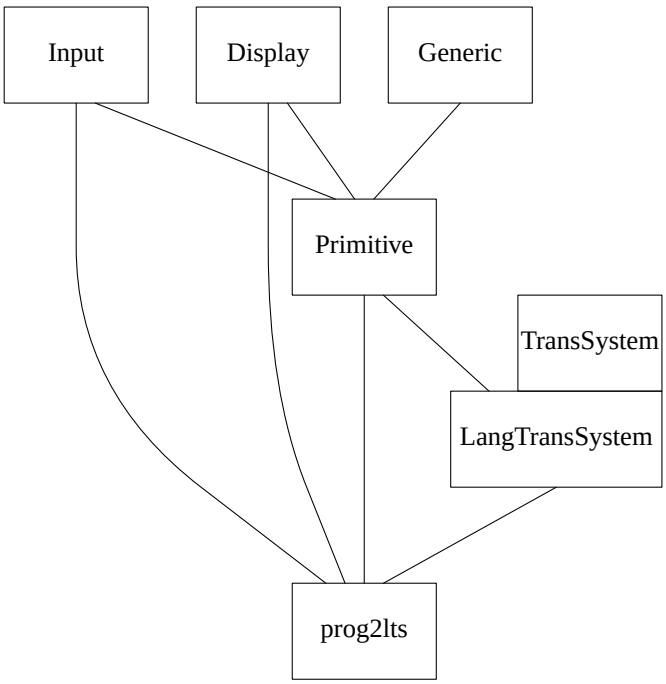


**Figure 14.** Model of program to transition system

The module TransSystem contains standard methods for adding states, adding transitions, and writing the LTS to a file. LangTransSystem is the name of the module that contains the method that converts the program in the language (primitive instruction set) 'Lang' to a LTS. Which primitive instruction set should be used, can be indicated by a commandline option. Currently, the toolset contains modules for PGLA and PGLEc (called PGLATransSystem and PGLEcTransSystem).

The numbers of the states are given by the number of the instruction in the program (starting with 0). A jump (or repeat, or label) instruction is converted to a tau transition. For every other instruction two transitions are added. One with the basic instruction without spaces and the character T (for true) as label, and one with the character F (for false) added as label. The next state indicated with a transition is the number of the next instruction that should be executed depending on the value returned by the basic instruction.

For example, the PGLA program

```
x = new;
x.+f;
```

```
y = new;
- x/f;
#4;
y.+g;
y.g = x;
#3;
y.+f;
y.f = x;
x = y;
\#9
```

is converted to the following LTS. The first line indicates the start state, the number of transitions, and the number of states. The other lines describe the transitions, consisting of the origin state, a label, and the destination state.

```
des (0, 21, 12)
(0, "x=newT", 1)
(0, "x=newF", 1)
(1, "x.+fT", 2)
(1, "x.+fF", 2)
(2, "y=newT", 3)
(2, "y=newF", 3)
(3, "x/fF", 4)
(3, "x/fT", 5)
(4, "tau", 8)
(5, "y.+gT", 6)
(5, "y.+gF", 6)
(6, "y.g=xT", 7)
(6, "y.g=xF", 7)
(7, "tau", 10)
(8, "y.+fT", 9)
(8, "y.+fF", 9)
(9, "y.f=xT", 10)
(9, "y.f=xF", 10)
(10, "x=yT", 11)
(10, "x=yF", 11)
(11, "tau", 2)
```

### *7.2 Implementation of the bisimulator*

The bisimulation tester is written in the language C [6], because of the speed needed when comparing larger programs. The algorithm used for deciding bisimularity differs in great length from the one described by Paige and Tarjan [8]. It is faster, but it should be noted that it is geared towards the transition systems produced from our programs and only checks whether the starting states (the start of the programs) are bisimilar. Research is going on currently to investigate whether the algorithm used here can be applied to transition systems in general with a better performance than existing algorithms.

## 8. Conclusions

We have described the implementation of a toolset for PGA which contains projections described in the referred articles and simulators for testing programs in the various program notations. With this toolset, new primitive or basic instruction sets can be added easily, and due to the generic nature of the parser and simulators, it is easy to test programs that use these new instruction sets.

**References**

[1] J.C.M. Baeten and W.P. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[2] J.A. Bergstra and I. Bethke, ‘‘Molecular dynamics,’’ *Journal of Logic and Algebraic Programming*, vol. 51, no. 2, pp. 193-214, 2002.

[3] J.A. Bergstra and M.E. Loots, ‘‘Program algebra for sequential code,’’ *Journal of Logic and Algebraic Programming*, vol. 51, no. 2, pp. 125-156, 2002.

[4] J.A. Bergstra and A. Ponse, ‘‘Combining programs and state machines,’’ *Journal of Logic and Algebraic Programming*, vol. 51, no. 2, pp. 175-192, 2002.

[5] J.A. Bergstra and P. Klint, ‘‘The discrete time ToolBus,’’ *Science of Computer Programming*, vol. 31, no. 2-3, pp. 205-229, July 1998.

[6] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, second edition, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[7] J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[8] R. Paige and R. Tarjan, ‘‘Three partition refi nement algorithms,’’ *SIAM Journal of Computing*, vol. 16, pp. 973-989, 1987.

[9] A. Ponse, ‘‘Program algebra with unit instruction operators,’’ *Journal of Logic and Algebraic Programming*, vol. 51, no. 2, pp. 157-174, 2002.

[10] L. Wall, T. Christiansen, and R.L. Schwartz, *Programming Perl*, O’Reilly & Associates, Inc., 1996.

## Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0302] B. Diertens, *A Toolset for PGA*, Programming Research Group - University of Amsterdam, 2003.

[PRG0301] J.A. Bergstra and P. Walters, *projection semantics for multi-file programs*, Programming Research Group - University of Amsterdam, 2003.

[PRG0201] I. Bethke and P. Walters, *Molecule-oriented Java Programs for Cyclic Sequences*, Programming Research Group - University of Amsterdam, 2002.

The above reports and more are available through the website: www.science.uva.nl/research/prog/

Electronic Report Series

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
the Netherlands

www.science.uva.nl/research/prog/