Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT

Information and Software Technology 45 (2003) 681–689

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Operator programs and operator processes

Jan Bergstra[a,b], Pum Walters[c,*]

[a]*Programming Research Group, University of Amsterdam, The Netherlands*
[b]*Applied Logic Group, Department of Philosophy, Utrecht University, The Netherlands*
[c]*Microsoft, Amsterdam, The Netherlands*

Received 31 August 2002; revised 15 February 2003; accepted 4 April 2003

Abstract

We define a notion of program which is not a computer program but an *operator program*: a detailed description of actions performed and decisions taken by a human operator (computer user) performing a task to achieve a goal in a simple setting consisting of that user, one or more computers and a work environment.

Our definition and notations are based on the program algebra PGA: a small body of theory allowing one to reason fundamentally and practically about programs viewed as instruction sequences.

This article is entirely self-contained and introduces all concepts and notations used. We offer some small examples, and we sketch one limitation of our approach.

Keywords: Algorithms; Agent modeling; Human–computer interaction; Specification languages; User behavior; User modeling

1. Introduction

Many texts on programming and programming languages do not offer a selfcontained definition of what constitutes a computer program. Program algebra arose from an intention to have a pragmatic exposition of programs using principled definitions. In the program algebra PGA [7,1], a program is defined as a sequence of instructions, or any entity the meaning of which is defined by mapping it to a sequence of instructions.

One purpose of program algebra is to provide definitions of programs and to provide a theory of programs and programming based on this intrinsic definition of programs, leaving no room for doubt about the status of the objects classified as programs[1].

However, the theory of [1] is independent of the origin of programs and of their mode of operation, as well as of their purpose, if one exists or is known. By definition, every sequence of instructions, in the program algebra format and ignoring details of representation, qualifies as a program, and any program description which can be transformed into such a sequence of instructions also qualifies as a program. The transformation of a program description into a sequence of instructions is called a projection[2], and in order to qualify a description as a program, a projection must be known.

This definition of a program is not always satisfactory, as it may be insufficiently restrictive for purposes of computer science: things turn out to be programs, which have no reasonable bearing on computer science whatsoever.

Having a focus on computing, we will restrict ourselves, and distinguish two classes of programs: computer programs, meant to be executed by a computer (a computing device), and operator programs, meant to be executed by an operator (a computer user).

To see that these are indeed different, consider a setting with a single computer and a single operator. Let `p:on` be the instruction to switch the power supply on. It is hardly

* Corresponding author. Tel. +31-35-6221013; fax: +31-35-6221014.
E-mail addresses: pwalters@microsoft.com (P. Walters), jan.bergstra@phil.uu.nl (J. Bergstra).

[1] Logic programming, for instance, leaves open the question why logic programs are programs in the first place. Now that need not be seen as a weakness but from our point of view the theory of logic programming might be called 'Horn clause sequence processing theory' just as well. Whether a logic program constitutes a reasoning mechanism, or merely an input for it is left open in the logic programming theory. From the program algebra perspective, however, a logic program definitely lacks the imperative aspects needed to classify it as a program at all.

[2] Returning to logic programming once more: a compiler for logic programs usually has this task, in which the needed details about the reasoning strategy which will be applied are clarified.

plausible to consider `p:on` an instruction meant to be executed by the computer; most plausibly all programs including (possibly after projection) `p:on` are operator programs. Note that switching off (`p:off`) may very well be a computer instruction, so a program involving p:off may, in the absence of other information, be of either kind.

From the perspective of the program algebra PGA, the first outcome of this paper is simply the fact that there is a distinction between computer programs and non-computer programs.

In the remainder of this article we investigate the second category, noticing that an abundance of computer programs written in program algebra notation can already be found (e.g. at http://www.science.uva.nl/research/prog/projects/pga/).

1.1. Formalizing operator activity

The dogma of program algebra is that all candidate programs, written or represented in whatever program notation, may be classified as programs by an observer, only if (s)he knows how to project the candidate program on a program that fits with existing definitions, i.e. on a sequence of instructions and ultimately, but in practice this is the same, if (s)he knows how to translate the candidate program to PGLA, the simplest of the program algebra based program notations.

PGLA being very restrictive, one may debate the validity of this dogma. In Section 5 we will encounter a typical task which is easy for a programmer but not at all easy to project on PGLA (the technical complication being that the boolean values are the only built in datatype in PGLA, and that all other data handling must therefore be translated back to bit level).

It is not the purpose of this paper to either claim that all operator activity can be phrased in terms of operator programs, or that all operator programs can be naturally or straightforwardly projected onto OPNA (the subset of PGLA which is its operator program counterpart, to be introduced hereafter). However, an implicit claim of this paper is that program algebra offers a framework which allows fundamental and rigorous reasoning about a certain class of operator programs. We do not exhaustively substantiate this claim, merely offering a few examples.

In addition, the result of this paper is two-fold:

- It provides an example of a class of non-computer programs, the so-called operator programs. These operator programs match all aspects of the definition of a program, as defined in the program algebra PGA, but are clearly not meant for execution by a computer. This leads to the observation that 'computer program' is a proper subclass of 'program' (in object oriented terms).
- Secondly, a method for formalizing operator activity via operator programs is given. The formalization rests on a suitably tailored version of the so-called focus-method notation for program algebra.

Formalizing operator activity via programs is useful because operator activity often has an algorithmic nature. In addition one may very well imagine that human operators are increasingly replaced by intelligent agents. For agents it is a natural thing to describe their handling of a system by means of program execution. The ability to represent programs, reason about programs and indeed automate such reasoning, where the programs denote computer activity, human activity, or a combination thereof, appears to be of growing importance in this light.

1.2. Overview

In Section 2 we shall offer some common terminology, useful in the description of human, computer-related activities, and we shall describe the GOMS model [3] of human–computer interaction, which is widely regarded as a reference model. Then, in Section 3, we present our model. We shall briefly introduce program algebra and its notation so as to make this paper independently readable. In Section 4 we offer some examples. Section 5 sketches an example where human mental processes are not so easily expressed using program algebra. In Section 6 we present a second notation, allowing for more compact operator program notations; the ability to define more concise or more appropriate notations is one of the most useful aspects of program algebra. In Section 7 we present a final example with (limited) practical relevance. Finally, we offer some conclusions and suggestions for further research.

2. Terminology and background

2.1. Operator tasks, skills, process and process description

To achieve certain goals, an operator of a machine can perform certain tasks, and in doing so (s)he demonstrates certain skills. An operator skill is the ability to perform an operator task to achieve a goal. Operator tasks typically involve dealing with a variety of circumstances.

To express the complexity of operator task accomplishment in light of the required skills we will use the phrase *operator process*. An operator process can be observed, required, proposed, validated, falsified, forbidden, outdated, experimental, and so on.

We imagine that to achieve a certain goal, an operator needs the ability to carry through a number of operator processes. 'Operator ability' refers to the ability to carry out an operator process. Operator ability is like the ability to play music by head; the operator need not use or demonstrate any particular operator process description in order to demonstrate this ability.

How to carry out a specific operator process may be documented in a manual, it may be known to the operator through experience, it may reside in his or her genes, and so on.

Operator processes can be identified by making observation protocols of operator activity, in combination with interviews regarding the options for further action that operators claim to have considered in various stages of their activity. This approach was introduced as *contextual inquiry* in [8,9] as a system design technique to study interaction outside laboratory circumstances, in its context.

Operator processes may also have been designed as part of the computer system design. Then, the process may be exemplified by means of so-called use cases, which can be used as training materials.

Whether emerging by observation or by design, operator processes are made explicit by means of operator process descriptions. Such descriptions are phrased in an operator process description format, which may vary from completely informal (‘if you haven’t done a Linux installation before, ask a friend for help’) to a specification in meticulous detail (‘at reply “OS12.B.45” enter “yes” and press RETURN’).

Each particular form of operator process description is an add-on feature for the underlying process and the ability shown by anyone conducting the process.

Below, we will be quite specific about operator programs as a possible operator process description format. There is no implication at all that an operator availing of some ability either consciously or unconsciously must ’contain’ an operator program for the operator process (s)he is able to conduct.

What may be said is that is that operator programs are quite powerful as an operator process description format. Here, ‘powerful’ refers to the capacity of giving descriptions for a wide variety of processes. Powerful and practical are different things, however. If an operator needs to conduct a process ’by head’ (s)he will need a description with suffifcient ergonomic qualities. Specific ergonomic requirements may lead to special-purpose process description formats that depart quite a bit from conventional program representations. See, for example, [6] for steps towards formalization of tutorial process models, where tutoring systems teaching procedural skills are considered.

Finally, we will use the phrase ’operator plan’ as an alternative for operator process description. The existence of an operator plan is not implied even when faced with a most skillful computer operator. For instance, a child operating a computer game may be very skillful and completely unable to formulate its own plan at the same time.

2.2. GOMS

The GOMS model [3] is a reference model for human computer interaction. It describes the user’s cognitive structure using four notions: Goals, Operators, Methods and Selection rules, where an *operator* is an elementary act necessary to change the user’s mental state or the environment; a *method* is a ’procedure’ consisting of operators, sub-goals and conditions which accomplishes a goal; and *selection rules* describe how a user selects one of multiple available methods to achieve a goal.

In addition, [3] mentions tasks, skills and plans without formalizing them in the model.

These notions and the ones we introduced earlier are related, although we do consider the user’s actions and behavior rather than the underlying cognitive process[3].

Like a method, our use of the word ’task’ signifies both atomic actions (i.e. operators) and more complex processes involving conditions and sub-tasks. However, we distinguish rigorously the process from its description.

Not focusing on the cognitive processes, our use of the words ’skill’ and ’ability’ is broader than that of [3] and includes both the capability to perform atomic tasks and the capability to chose suitable tasks given a goal (selection). Similarly, we use the terms ’operator plan’ and ’operator process description’ interchangeably, focusing on the structure rather than the cognitive processes leading to that structure.

3. OPNA: operator program notation A

In ultimo abstracto we will distinguish three entities in the context of and in relation to an operator. Having a focus on operating a computing device, the first entity is a console, representing an interface between operator and computer[4], which allows the operator to interact with the computer.

The second entity, ‘self’, concerns the operator’s ability to make decisions about the selection of appropriate tasks to be carried out by the computer in order to accomplish certain goals. Perhaps it is surprising to distinguish an operator from this ability, but in fact a distinction between decision making and execution is quite common in process structuring. Also, it is consistent with [3] and other work in this area, where task selection and the ability to perform tasks are distinguished. (Fig. 1)

We shall call the third entity ‘principal’. It embodies the context which defines the operator’s goal.

Although it is perhaps premature to discuss the merits of this abstraction before we have even looked at its application, one comment may be in order: the entities embody the what, why and how of operator activity.

In the picture we see an operator in this context, and we see some further details about the entities mentioned.

CMR *Console method repertoire*, representing all commands the operator can give to a computer,

[3] In [4] this approach is known as *Input Output Agent Modeling*.

[4] A notion we imagine to include multiple units and entire networks.

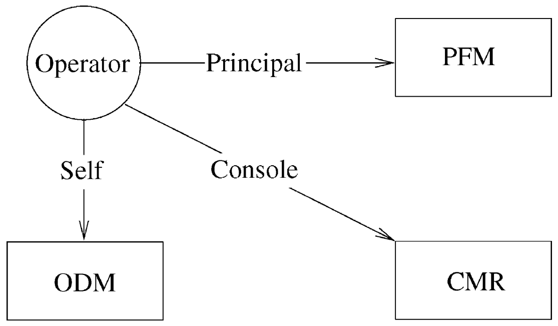

Fig. 1. OPNA setting.

and various physical actions on and around computer systems, such as turning a computer on, inserting a CD-ROM, or adding paper to a printer;

PFM *Principal feed-back methods*. The feed-back methods are replies (including implicit queries) that the operator can give to the principal;

ODM *Operator decision methods*. The complete repertoire of decisions the operator can make, based upon their skills and abilities.

3.1. PGA

The formalization of operator programs below will take place in the setting of program algebra. A full introduction to programs and related notations in the setting of program algebra can be found in [1] and [2], but a brief description may facilitate understanding.

A sequence of instructions is a program. There are two kinds of instructions: *control instructions*, solely concerned with the locus of attention, i.e. the location where subsequent relevant instructions can be found, and *basic instructions*, which require an entity in the program's context, called a *reactor*, to offer some service. Basic instructions are used in two ways: conditional, where the program prescribes to interpret a status offered by the reacting entity to determine how to proceed, and unconditional (called *void* in [1]), where no status is considered.

The entity offering the service required in a basic instruction is called a re-actor, or a co-program, or also (in simple cases) a state machine.

3.2. FMN: focus-method notation

The *focus method notation* introduced in [1] defines a specific notation for basic actions: a basic action consists of two strings separated by '·', where the first string, called the focus, identifies a reactor, and the second string, called the method, identifies a service offered by that reactor and possible arguments to that service. Focus and method consist of alpha-numeric symbols and the colon ':', which is used to separate fields. One reactor can be designated *default* for a program. Its designation and the '·' may be left out in basic actions. Obviously, the default must offer all services required (as indeed should all reactors). When describing operator console interaction it is plausible to consider the console to be the default focus.

To conclude this introduction, we mention that [1] defines a number of languages (i.e. program description formats), each being a program language by virtue of a map to sequences of instructions. Basic instructions are a parameter to the languages, so the various formats only differ in the control instructions they offer.

3.3. PGLA

The simplest language is PGLA, which offers the control instructions '!', signifying successful completion of a task, and '#n', signifying that the remainder of a task is described n instructions ahead. This is a jump instruction jumping over itself as well as the next $n-1$ instructions if $n>0$. If $n=0$ it denotes an infinite repetition or divergence. Two program composition mechanisms are available in PGLA: concatenation '-;-', and repetition '$(\text{-})^{\omega}$'. Repetition will not be used in the examples below.[5] The 'A' in OPNA stems from PGLA.

3.4. OPNA

As shown in the picture above, the entities in the operator's context are PFM, CMR and ODM. One might consider conditional instructions in relation to PFM (e.g. ask the boss) and CMR (e.g. ask your project planner). The reason not to include these is as follows. In both cases an unconditional request is presented (to the principal or to the console). As a consequence the principal or computer may produce results that the operator may inspect. The operator then has to decide what to do next on the basis of this inspection. The decision methods of the operator, accessed via the focus `self`, will usually involve an assessment of the most recent information that has been produced by principal and computer. In the case of the principal that may involve the inspection of documents, in the case of the computer it may involve the inspection of various windows on a screen, not just the window in which the commands for focus console are typed.

In fact, to ask the boss is to ask for a refinement of goals, which may lead to information that allows the operator to take further decisions. The replies of a principal do not coincide with operator decisions though a very direct translation of principal generated information into operator decisions is very well conceivable. Asking a computer concerns an unconditional request to provide additional information; how that information is presented is left unspecified, but how to proceed based upon that information remains the operator's realm, in the proposed formalization decision actions will take care of all processing of such information. Similarly, unconditional instructions concerning ODM should not be

[5] An ASCII notation for repetition is as follows: "\# n", denoting the repetition of the last n instructions, not including itself.

considered: it might represent an operator mumbling to themselves, but it has no bearing on our theory. The only reason for the operator to assign a method to self is because a decision needs to be made on how to proceed. To conclude, using the notation of [1] and [2] for basic actions:

- Basic actions concerned with *self* are conditional: i.e. $+self.p$ or $-self.p$; when a positive conditional action is processed the entity running the program will ask the appropriate reactor (here only `'self'`) to perform the action. If then a reply `true` is produced, the execution of the program continues with an execution of the next instruction, whereas if the reply `false` was produced the execution skips (i.e. jumps over) the next instruction and proceeds with the subsequent instruction.

  In the case of a negative conditional instruction, the execution of a program proceeds with the next instruction after a reply `false` was received, jumping over the next instruction in the case that `true` was received. PGLA needs only one of these conditional instructions for its expressive power; the other one has been added for symmetry reasons. Having both available, however, does simplify programming.
- Basic actions concerned with *principal* or *console* are void: i.e. *console.p* or *principal.p*. For the program notation PGLA there is no need to have these instructions used in void mode only. That restriction is merely based upon the view that the operator is the only party taking decisions on how to proceed.

3.5. Why PGLA?

When modeling operator activity via operator programs many program notations will serve equally well. The reasons to work with PGLA are these: (i) it is the simplest program notation known to us, (ii) it has complete separation between control features and state modifying actions, (iii) it can be used as a mathematical notation just as well as as a pragmatic program notation, (iv) if the description of more complex tasks requires more flexible program notations, an appropriate notation can be chosen from the hierarchy of program algebra notations as proposed in [1]. Doing so gives the guarantee that the more advanced program notation merely helps the presentation of PGLA programs in a more readable form, because the meaning of each of the more advanced notation is given in terms of a projection back to PGLA.

4. Examples

In this section we will discuss some examples.

4.1. A simple task: logging in

In this section we discuss a trivial task. Entering the office every morning, the first things to do are: turn on ones workstation, log in to the system, scan new mail, and report to the principal[6].

In this example, the CMR should offer the following services:

- `on` to turn the computer on;
- `write:`... to enter some data;
- `select:`... to start an application

The PFM is trivial:

- `ready` to report for duty;
- `need:`... to request assistance of some sort.

In this trivial task few decisions need to be made. Things that typically go wrong at the site we are thinking of are that the network or one of the servers is down. Therefore, ODM includes services

- `requesting:`... to decide whether the workstation is waiting for some kind of response;
- `alerting:`... to decide if some system message is being presented;
- `responding:`... to decide if the system is otherwise responsive;

At this point we mention the influential work on use-cases [5] used as a software engineering method, notably in information systems design. Use-cases are related to the above in that top-level processes from the perspective of a single operator are considered. The chief differences with our approach are:

- In Ref. [5] the processes are commonly described in text, rather than any formal framework. From this perspective OPNA could very well be used to define use-cases in great detail.

  Note that sequence diagrams are sometimes (incorrectly) also referred to as use-cases. A sequence diagram breaks down a process in detail. However, sequence diagrams are commonly used to break down computer programs rather than operator programs, mentioning the operator merely as the originator of the process;
- Use-cases are described from the perspective of an operator without distinguishing PFM and CMR, and often without making ODM explicit. It would be conceivable for a use-case to fail to clarify how an operator should decide how to proceed based on the result of some console command (e.g. does the result indicate what should happen or should the user make

[6] In reality the first thing to do is: get coffee. It is not our intention, however, to describe the behavior of this or any operator but rather to describe the acts and decisions needed to fulfill one task. In practice, an operator fulfills different tasks at the same time by interleaving or indeed superimposing sub-tasks related to those different tasks. For the moment we focus on individual tasks.

a decision based on some informational result). In OPNA, such an unclarity is farfetched. This is by no means criticism on [5], but rather on the phrasing of that particular use-case.

The program describing this task is:

```
console.on;
console.write:janb;
-self.requesting:password; #10;
console.write:777777;
+self.alerting:authenticationfailure; #7;
console.select:my mailer;
-self.responding;#4;
console.openlastmail;
principal.ready;
!
principal.need:help;
!
```

A verbal rephrasing of this program is as follows: switch the console on (apply the method 'on' to console), then write 'janb' on the console, then decide (self) whether a password is requested. If not, skip the next 10 instructions (arriving at the instruction principal.need:help) and signal the principal that help is needed. Then, the task is done. If a password is requested, write it to the console (an unrealistic example is used) and then decide whether the authentication procedure generates a failure. If so skip some instructions and tell the principal that help is needed, if not ask the computer (via console) to open the most recent mail. Then tell the principal that the task has been completed and terminate.

Perhaps surprisingly, this trivial example exhibits all aspects of OPNA, and although additional examples serve to further illustrate our formalization of operator activity, they are all based on the primitives shown above.

4.2. A pocket calculator

In this example we describe the use of a pocket calculator. Accordingly, it makes sense to take the console as the default focus (as mentioned, the default focus may be skipped in method denotations).

The CMR offers typical operations on a calculator. Our calculator has a number of variables a through z, b2, cf, etcetera, shown as VAR below. Commands exist to set a variable to a constant (8 decimal places), or to compute its value using an operator and one or two other variables. Operators (OP) include plus, mul, min, div, and sqrt (min can appear as monadic and as diadic operator).

Thus, CMR offers the following services: CMR = {set:VAR:VAL, set:VAR:OP:VAR, set:VAR:VAR:OP:VAR, show:VAR, show:VAR:VAR, show:txt:TEXT}.

The PFM are simple: the operator either indicates that no result is possible, or that the computation was completed.

ODM is concerned with simple properties of numbers: is a number less than zero, or equal to zero (isnegative:VAR,iszero:VAR). The tests are applied to numbers that have been rendered on the console following a show:... command. How the operator spots the proper window where to find this text and similar matters, is all hidden in the various methods for self.

The following program describes the task of producing solutions to the quadratic equation $ax^2 + bx + c = 0$, for $a = 160, b = 52.625$ and $c = -1867.3$, if such solutions exist. The following formula is implicitly used:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{\mathtt{mb} \pm \sqrt{\mathtt{det}}}{\mathtt{a2}}$$

```
set:a:160;
set:b:52.615;
set:c:minus 1867.3;
set:a2:a:plus:a;
set:mb:min:b;
set:u:b:mul:b;
set:v:a:mul:c;
set:w:4;
set:v:v:mul:w;
set:det:u:min:v;
show:det;
-self.isnegative:det; #3;
principal.no;
!;
set:d:sqrt:det;
show:d;
-self.iszero:d; #4;
set:p:mb:div:a2;
show:p;
#6;
set:p1:mb:min:d;
set:p2:mb:plus:d;
set:p1:p1:div:a2;
set:p2:p2:div:a2;
show:p1:p2;
principal.done;
!;
```

Obviously, our second example doesn't bring any great surprises, because all primitives have already been discussed. Note however, that a number of nontrivial user actions have been described in every relevant detail in a manner which is open to formal reasoning.

5. Intermezzo: finding auxiliary names

The operator program for solving the quadratic equation can be easily generated by many human operators who had some elementary training in mathematics. One may ask whether any difficult steps are involved in producing such a program, that defeat automatic program generation, and indeed there is one major difficulty: in the program several names for variables (registers) are introduced (e.g. `mb` and `det`). These names need to be new in the sense that no unwanted interference with other names take place. The (possibly artificially intelligent) program author is confronted with the so-called frame problem: what exactly is the context where the names must be new.

In ordinary mathematics this is never considered a big deal, one is simply asked to generate some new names and this is considered successful if there are no obvious collisions. A precise definition of the context where these collisions must be avoided is never provided in an elementary account of the use of auxiliary names.

In our example it could make sense to leave the task of finding new names to the principal, who may be considered an authority regarding the context of the operator activity. After asking a new name (`principal.provide:new-name`), the principal could demonstrate a piece of paper with the next new name. However, an operator program based on such a method, where the principal is used, in an interactive way, as a generator of auxiliary new names that will cause no collisions, is hard to write in PGLA.

Several solutions are possible (within the framework of OPNA) but all of them are fairly complex, as in each case the program needs to move significant information (involving the new names) from the principal to the console, while dealing with boolean data only.

Most program notations provide features for defining local names, where avoiding collision is dealt with straightforwardly, usually using scopes. Similarly, it is conceivable how OPNA could be extended to provide such a facility without the programmer having to spell it out in every detail, for example by adding an additional reactor to that end. However, this would defy a key purpose of OPNA and PGA, which is precisely to make explicit the details of these and other features assuming no more than a handful of well understood primitives.

6. OPNEc

The language PGLEc is related to PGLA by a series of maps which define the meaning of a PGLEc program as that of its PGLA image. The details of these maps are not relevant to our current discussion (they can be found in [1]).

There are some key differences between PGLA and PGLEc: the control instruction #*n* has been removed and two types of control instruction have been added. Firstly ##*Ln* (goto) signifies that a task's description continues at the first location in the program headed by the instruction *Ln* (label catch instruction). The label instruction signifies a place where a processing should continue, and signifies nothing else (it does not tell the operator to 'do' anything). These features can be easily given a semantics by projecting programs using labels and goto's into PGLA programs.

Secondly, for any basic instruction α, '$+(\alpha)\{$' and '$-(\alpha)\{$' are new control instructions designating the start of a sequence of instructions that should be performed if α succeeds, or fails, respectively. That sequence can be terminated by one of the new instructions '} {' and '}'. The former signifies the end of the sequence of instructions and the beginning of another sequence of instruction which should be performed if the reverse happened to be the case (with respect to α). In that situation the sequence of instructions is terminated with the instruction '}'. Other than the role described here, the instructions '} {' and '}' do not lead to action. Again the meaning of these instructions can be given in terms of a projection. Now it is easiest to use a program notation with labels and goto's as an intermediate stage.

The purpose of these additions to the language is of course to allow for more comprehensible program descriptions.

PGLEc gives rise to the Operator Program Notation Ec: OPNEc. In OPNEc the program presented in the Section 5 can be written as (we only show the altered part):

```
-self.isnegative:det{;
  principal.no;
  !;
};
set:d:sqrt:det;
show:d;
-self.iszero:d{;
  set:p:mb:div:a2;
  show:p;
}{;
  set:p1:mb:min:d;
  set:p2:mb:plus:d;
  set:p1:p1:div:a2;
  set:p2:p2:div:a2;
  show:p1:p2;
};
principal.done;
!;
```

7. Example

In this section we will develop an operator program which describes an operator process that occurs in day to day life, and where the availability of such a program is, in our opinion, useful. The program intrinsically involves human action.

The example concerns double sided printing using a desk-top printer such as a desk-jet or small laser printer, and pertains to printers in which a page is turned once during the printing process (or any odd number of ways, actually). This happens to be the case for almost all low-end printing systems. Most such systems do not have a built-in double sided print facility.

The program to print a file double sided is as follows. In this program the principal does not occur, and the default focus is taken to be CMR. Methods starting with `menuSelect` are taken to be commands to a computer in a printdialog; methods starting with `printer` are physical actions concerning the desktop printer.

```
menuSelect:pages:evenOnly;
menuSelect:printOrder:normal;
menuSelect:print;
printer:outputTray:extract:printed ~[7]
Pages::doNotRotate;
printer:open:container;
-self.isEven:numberOfPrintedPages{;
  printer:container:extract:blank~
  Page:1;
};
printedPages:rotateInPlane:180;
printer:container:insert:printed~
Pages::doNotRotate;
-self.isEven:numberOfPrintedPages{;
  printer:container:insert:blank~
  Page:1;
};
printer:close:container;
menuSelect:pages:oddOnly;
menuSelect:printOrder:reverse;
menuSelect:print;
!;
```

In words: print the even pages in normal order (i.e. starting with 2); extract them from the tray and insert them on top into the blank page holder, rotating them within their plane around 180°; add one blank page on top if the total number of pages to be printed is odd; and print the odd pages in reverse order (i.e. 1 last).

Although this process is simple, any deviation leads to paper waste rather than conservation, which may be the reason to print double sided. In our personal experience, attempting to determine what should happen next by reasoning through the printing process halfway through a double-sided print job, has a fair chance of failing. This is not unlike to the centipede who, when asked how on earth it keeps track of all the legs while walking, in an attempt to understand and describe the process, ends up in a horrible mess. Sticking to this program mono-maniacally succeeds (barring mechanical failure).

[7] The symbol '~' is used to signify a line-break. That symbol, and the immediately following line break are part of this presentation but are not considered to be part of the program or of the language OPNA.

8. Conclusions and further research

OPNA is in a limited sense complete. This fact is obtained as follows. For an OPNA program its external behavior can be defined as the collection of sequences of actions with focus `principal` or `console` that it can generate. The external behavior abstracts from internal (decision making) activity. The external behavior of an OPNA program may be understood as the operator process that it represents (that is, the process entails all possible rather than one particular sequence of external events).

Now let an operator process *P* be presented by means of a finite set of its traces. This involves the severe restriction that the operator process can unfold in finitely many ways only, each of them terminating after finitely many steps. Then one can find decision methods `ODM` for the operator, consisting of a set `d1,…,dn` of decision methods and an operator program using the methods occurring in the given trace sets with these operator decision methods occurring in its tests such that the program has the required trace collection. The proof is elementary. It gives no clue at all regarding which observations underly the formal decision methods that have been introduced. It may require a significant amount of empirical work to find out, in a realistic case, which observations were used by the operator for taking control decisions.

We have defined a notion of program which is not a computer program but rather a detailed description of actions performed and decisions taken by a human operator performing a task to achieve some goal.

We have also offered a model of an operator in their context which we claim to be very simple yet powerful, allowing one to model a substantial range of operator tasks; we have not substantiated this claim, offering but a few examples.

In these examples we have shown that this approach allows for a detailed and complete description of decisions, actions and communications between operators and their environment. The level of detail is in principle sufficient to model tasks such as can be produced from feature based competency modeling ([10]).

Research on program notations related to PGLA continues. Some aspects are specifically aimed at computer programs, but others are meaningful in a wider range of programs, such as the one we presented here.

References

[1] J.A. Bergstra, M.E. Loots, Program algebra for sequential code, Journal of Logic and Algebraic Programming 51 (2) (2002) 125–156.

[2] J.A. Bergstra, A. Ponse, Combining programs and state machines, Journal of Logic and Algebraic Programming 51 (2) (2002) 175–192.

[3] S.T. Card, T.P. Moran, A. Newell, The Psychology of Human–Computer Interaction, Lawrence Erlbaum Associates, London, 1983.

[4] B.C. Chiu, G.I. Webb, Using decision trees for agent modeling: Improving prediction performance, User Modeling and User Adapted Interaction 8 (1–2) (1998) 131–152.

[5] I. Jacobson, The use-case construct in object-oriented software engineering, Scenario-Based Design: Envisioning Work and Technology in System Development, Wiley, 1995, pp. 306–309.

[6] R.H. Kemp, S.P. Smith, Domain and task representation for tutorial process models, International Journal of Human–Computer Studies 41 (1994) 363–383.

[7] C.A. Middelburg (Ed.), Special Issue: Program Algebra, Journal of Logic and Algebraic Programming, 51(2) 2002

[8] J. Whiteside, J. Bennett, K. Holtzblatt, Usability engineering: our experience and evolution, in: M. Helander (Ed.), Handbook of Human–Computer Interaction, 1988, pp. 791–817.

[9] D. Wixon, K. Holtzblatt, S. Knox, Contextual design: an emergent view of system design, in: J. Carrasco-Chew, J. Whiteside (Eds.), Human Factors in Computing Systems CHI'0 Conference Proceedings, ACM, New-York, 1990, pp. 329–336.

[10] G.I. Webb, M. Kuzmycz, Feature based modelling: a methodology for producing coherent, consistent, dynamically changing models of agents' competencies, User Modeling and User Adapted Interaction 5 (1996) 117–150.