University of Amsterdam
Programming Research Group

A Bypass of Cohen's Impossibility Result

J.A. Bergstra
A. Ponse

Report PRG0501 January 2005

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

tel. +31 20 525.7591
e-mail: janb@science.uva.nl

A. Ponse

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

tel. +31 20 525.7592
e-mail: alban@science.uva.nl

Programming Research Group Electronic Report Series

A Bypass of Cohen's Impossibility Result

Jan A. Bergstra[1,2] and Alban Ponse[1]

[1] University of Amsterdam, Programming Research Group, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
[2] Utrecht University, Department of Philosophy, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

Abstract. Detecting illegal resource access in the setting of grid computing is similar to the problem of virus detection as put forward by Fred Cohen in 1984. We discuss Cohen's impossibility result on virus detection, and introduce "risk assessment of security hazards", a notion that is decidable for a large class of program behaviors.

Keywords: Malcode, Program algebra, Thread algebra, Virus, Worm.

1 Introduction

Grid computing poses many challenges which are known from computer science, though now at an integrated level. For semantic work about the grid it is hard to reach that level of integration as well. An example of semantic work on grid computing is in [8], in which the authors Nemeth and Sunderam point out that convincing definitions of what constitutes a grid are still hard to obtain. They set out to provide a definition, and in the preparation of the formal work a table is presented with a comparison of grids and conventional distributed systems. It is stated in their Table 1 that in a grid, access to a node may not imply access to all of its resources and that users working from another node may have little information on accessible resources.

We will study this aspect in particular under the assumption that a node takes the responsibility to prevent illegal access to resources by tasks it accepts to execute. We try to find a simplest possible model for this issue and use thread algebra in combination with program algebra. These techniques were introduced in [2] and [1], respectively. (The single thread version of thread algebra is named polarized process algebra, see [1]). This leads to the preliminary conclusion that a fair amount of phenomena in concurrent processing may be understood and formalized along those lines, in particular phenomena where non-determinism is immaterial. It seems to be the case that many characteristic aspects of grid computing can be analyzed using the paradigm of strategic interleaving as put forward in [2]. That means that instead of taking a combinatorial explosion of different runs into account, a limited portfolio of interleaving strategies may be used to characterize vital phenomena, including issues concerning thread mobility and access control. Deterministic strategies inherit from planning theory as discussed in [6].

It appears that detecting illegal resource access is formally very similar to the problem of virus detection as put forward by Cohen in 1984. There is a vast amount of literature on virus detection, and opinions seem to differ wildly. Many authors agree that malcode contains all others, and that both a virus and a worm can replicate. Furthermore, a worm is more autonomous than a virus. Some authors claim that a virus can only replicate as a consequence of actions of users, and that sound education and awareness can protect users from acting with such effect. So, a virus uses a user for its replication; that user may or may not be a victim of the virus' harmful action at the same time. Unclear is if each of these users must be a human one or if background processes in a machine can also be "used" as users.

This paper focuses on virus detection and discusses two fundamental questions. First, we consider Cohen's result about the impossibility of a uniform algorithm (or tool) for detecting (forecasting) viruses in programs [5]. This is done in the setting of the program algebra PGA [1]. Then, we define a different notion of testing — *security hazard risk assessment* — with which the occurrence of security hazards is decidable for a large class of program behaviors. However, if divergence (the absence of halting) is considered also as a security hazard, decidability is lost.

The paper is organized as follows: in Sections 2 and 3 we introduce some basics of program algebra and the setting in which we will analyze code security risks. Then, in Section 4, we consider Cohen's impossibility result and some related issues. In Section 5 we introduce our notion of security hazard risk assessment. The paper is ended with some conclusions.

2 Basics of Program Algebra

Program algebra (PGA, [1]) provides a very simple notation for sequential programs and a setting in which programs can be systematically analyzed and mapped onto behaviors. Program behaviors are modeled in thread algebra. Finally, we consider some other program notations based on program algebra.

The program Algebra PGA. In PGA we consider *basic* instructions $a, b, \ldots$ given by some collection B. Furthermore, for each $a \in B$ there is a *positive test* instruction $+a$ and a *negative test* instruction $-a$. The control instructions are *termination*, notation !, and (relative) jump instructions $\#k$ ($k \in \mathbb{N}$). Program expressions in PGA, or shortly PGA-programs, have the following syntax:

- each PGA-instruction is a PGA-program,
- if X and Y are PGA-programs, so is their *concatenation* $X;Y$,
- if X is a PGA-program, so is its repetition X^{ω}.

The behavior associated with the execution of PGA-programs is explained below. Instruction congruence of programs has a simple axiomatization, given in

Table 1. Axioms for PGA's instruction sequence congruence.

$(X;Y);Z = X;(Y;Z)$	(PGA1)	$X^\omega;Y = X^\omega$	(PGA3)
$(X^n)^\omega = X^\omega$ for $n > 0$	(PGA2)	$(X;Y)^\omega = X;(Y;X)^\omega$	(PGA4)

Table 1. The axioms PGA1-4 imply *Unfolding*, i.e. the law $X^\omega = X;X^\omega$, and PGA2-4 may be replaced by Unfolding and the proof rule $Y = X;Y \Rightarrow Y = X^\omega$.

Thread Algebra. Execution of PGA-programs is modeled in thread algebra. Given B, now considered as a collection of *actions*, it is assumed that upon execution each action generates a Boolean reply (true or false). Now, behavior is specified in thread algebra by means of the following constants and operations:

Termination. The constant S represents (successful) termination.

Inaction. The constant D represents the situation in which no subsequent behavior is possible. (Sometimes the special thread D is called *deadlock* or *divergence*.)

Post conditional composition. For each action $a \in B$ and threads P and Q, the post conditional composition $P \unlhd a \unrhd Q$ describes the thread that first executes action a, and continues with P if `true` was generated, and Q otherwise.

Action prefix. For $a \in B$ and thread P, the action prefix $a \circ P$ describes the thread that first executes a and then continues with P, irrespective of the Boolean reply. Action prefix is a special case of post conditional composition: $a \circ P = P \unlhd a \unrhd P$.

Behavior Extraction: from program algebra to thread algebra. The behavior extraction operator $|X|$ assigns a behavior to program X. Instruction sequence equivalent programs have of course the same behavior. Behavior extraction is defined by the thirteen equations in Table 2, where $a \in B$ and u is a PGA-instruction.

Table 2. Equations for behavior extraction on PGA.

$\vert!\vert = S$	$\vert!;X\vert = S$	$\vert\#k\vert = D$
$\vert a\vert = a \circ D$	$\vert a;X\vert = a \circ \vert X\vert$	$\vert\#0;X\vert = D$
$\vert+a\vert = a \circ D$	$\vert+a;X\vert = \vert X\vert \unlhd a \unrhd \vert\#2;X\vert$	$\vert\#1;X\vert = \vert X\vert$
$\vert-a\vert = a \circ D$	$\vert-a;X\vert = \vert\#2;X\vert \unlhd a \unrhd \vert X\vert$	$\vert\#k{+}2;u\vert = D$
		$\vert\#k{+}2;u;X\vert = \vert\#k{+}1;X\vert$

Some examples: $|(\#0)^\omega| = |\#0;(\#0)^\omega| = D$ and, further taking action prefix to bind stronger than post conditional composition,

$$\begin{aligned}|{-a};b;c| &= |\#2;b;c| \trianglelefteq a \trianglerighteq |b;c|\\ &= |\#1;c| \trianglelefteq a \trianglerighteq b \circ |c|\\ &= |c| \trianglelefteq a \trianglerighteq b \circ c \circ D\\ &= c \circ D \trianglelefteq a \trianglerighteq b \circ c \circ D.\end{aligned}$$

In some cases, these equations can be applied (from left to right) without ever generating any behavior, e.g.,

$$|(\#1)^\omega| = |\#1;(\#1)^\omega| = |(\#1)^\omega| = \ldots$$
$$|(\#2;a)^\omega| = |\#2;a;(\#2;a)^\omega| = |\#1;(\#2;a)^\omega| = |(\#2;a)^\omega| = \ldots$$

In such cases, the extracted behavior is defined as the thread D.

It is also possible that behavioral extraction yields an infinite recursion, e.g.,

$$|a^\omega| = |a;a^\omega| = a \circ |a^\omega|,$$

and therefore, $|a^\omega| = a \circ |a^\omega| = a \circ a \circ |a^\omega| = a \circ a \circ a \circ |a^\omega| \cdots$. In such cases the behavior of X is infinite, and can be represented by a finite number of behavioral equations, e.g., $|(a;+b;\#3;-b;\#4)^\omega| = P$ and

$$\begin{aligned}P &= a \circ (P \trianglelefteq b \trianglerighteq Q),\\ Q &= P \trianglelefteq b \trianglerighteq Q.\end{aligned}$$

The program notations PGLB and PGLC. The program notation PGLB is obtained from PGA by adding backwards jumps $\backslash\#k$ and leaving out the repetition operator. For example, the thread defined by PGLB-program $+a;\backslash\#1;+b$ behaves as $(+a;\#4;+b;\#0;\#0)^\omega$, i.e., as P in $P = P \trianglelefteq a \trianglerighteq b \circ D$.
This is defined with help of a projection function `pglb2pga` that translates PGLB-programs in a context-dependent fashion. For a PGLB-program X we write $|X|_{pglb} = |\texttt{pglb2pga}(X)|$ (see further [1]).

The language PGLC is the variant of PGLB in which termination is modeled implicitly: a program terminates after its last instruction has been executed and that instruction was no jump into the program, or it terminates after a jump outside the program. The termination instruction ! is not present in PGLC. For example,

$$\begin{aligned}|+a;\#2;\backslash\#2;+b|_{pglc} &= |+a;\#2;\backslash\#2;+b;!;!|_{pglb}\\ &= |(+a;\#2;\#6;+b;!;!;\#0;\#0)^\omega|\\ &= P\end{aligned}$$

for $P = b \circ S \trianglelefteq a \trianglerighteq P$ (see [1] for precise definitions of $|X|_{pglc}$ and $|Y|_{pglb}$.)

3 Detecting Access to a Forbidden Resource

In this section we introduce the setting in which we will analyze code security risks. We now consider a thread algebra with actions in "focus-method" notation, i.e., actions of the form $f.m$ where f is the *focus* and m the *method*. A *forbidden resource* is a resource that may not be accessed by threads of ordinary security clearance. A focus containing a forbidden resource is called a high risk focus. The state of affairs in which a thread plans to access a forbidden resource is called a *security hazard*.

Let P be some thread that uses communication with the following typical resources H_e, H_f and H_g:

$$\begin{tikzcd}
\boxed{\boxed{P}} \arrow[r, "e"] \arrow[dd, "f"'] \arrow[dr, "g"'] & \boxed{H_e} \text{ (external focus)} \\
 & \boxed{H_g} \text{ (low risk focus, no security hazard)} \\
\boxed{H_f} \text{ (high risk focus, security risk)} &
\end{tikzcd}$$

The reply of a basic instruction $e.m$ will be determined by the resource H_e. Likewise, instructions with focus f or g communicate with H_f and H_g, respectively.

Now, execution is *secure* if no $f.m$ is called until termination or first call of some $e.m$ (to the external focus).

A thread can have low risk actions (secure execution expected) and high risk actions (insecure execution expected). For example,

S	— a low risk behavior (no security hazard),
$f.m \circ S$	— a high risk behavior (security hazard),
$f.m \circ S \trianglelefteq g.m \trianglerighteq g.m \circ S$	— risk depends on H_g (potential security hazard).

Suppose in some network, a site C receives the description p in some programming language PGLX of a thread $P = |p|_{pglx}$ to be run at C. Then

$$P \trianglelefteq \mathit{sctest.ok} \trianglerighteq S$$

formalizes a way for C to run P only if its security has been cleared: the test action *sctest.ok* (security clearance test) models this type of testing, yielding true if P is secure. In terms of the above modeling, such type of testing may be performed by a secure resource like H_g with focus *sctest* (thus H_{sctest}).

Alternatively, one can consider a test resource $H_{asctest}$ (alternative security clearance test) which produces true in

$$P \trianglelefteq \mathit{asctest.ok} \trianglerighteq S$$

if P has a security hazard. This resource may be implemented by always returning false. Consequently, the better option is to require that if in

$$P \trianglelefteq \mathit{asctest.ok} \trianglerighteq Q$$

the test *asctest.ok* yields false, the security of thread Q is guaranteed. In the next section we show that such a seemingly natural test action is self-contradictory; in Section 5 we propose a variant of *sctest.ok* that is not manifestly self-contradictory.

4 Security Hazard Forecasting

In this section we consider a security hazard forecasting tool and establish a formal correspondence between security hazard detection (a thread plans to access a forbidden resource) and the virus detection problem put forward by Fred Cohen in 1984.

Let SHFT be a Security Hazard Forecasting Tool with focus *shft*, thus a resource that forecasts a security hazard. As assumed earlier, a security hazard is in our simple setting a call (action) $f.m$ for some m. Furthermore, let *shft.test* be the test that uses SHFT in the following way: in

$$P \trianglelefteq \mathit{shft.test} \trianglerighteq Q,$$

the action *shft.test* returns true if P has a security hazard, and false if Q has no security hazard.

Theorem 1. *A Security Hazard Forecasting Tool cannot exist.*

Proof. Consider $S \trianglelefteq \mathit{shft.test} \trianglerighteq f.m \circ S$. If the test action *shft.test* returns false, then $f.m \circ S$ will be performed, which is a security hazard; if true is returned, then S is performed and no security hazard arises. □

The particular thread used in the proof above illustrates the impossibility of predicting that a thread (or a program) contains a virus, a general phenomenon that was described in Cohen's famous 1984-paper [5] and that will be further referred to as *Cohen's impossibility result.* For the sake of completeness, we recall Cohen's line of reasoning. In the pseudo-code below (taken from [5]), `D` is a decision procedure that determines whether a program is (contains) a virus, `~D` stands for its negation, and `next` labels the remainder of some (innocent) program:

```
program contradictory-virus:=
{1234567;

subroutine infect-executable:=
 {loop:file = get-random-executable-file;
 if first-line-of-file = 1234567 then goto loop;
 prepend virus to file;
 }

subroutine do-damage:=
 {whatever damage is to be done}
subroutine trigger-pulled:=
 {return true if some condition holds}

main-program:=
 {if ~D(contradictory-virus) then
    {infect-executable;
    if trigger-pulled then do-damage;
    }
 goto next;
 }
}
```

In PGLC, the program `contradictory-virus` can be represented by the following term `CV`:

$$\texttt{CV} = \#8; \texttt{Pre}; \#3; -\mathit{shft}.\texttt{test}(\texttt{CV}); \backslash\#8; \texttt{Next}$$

where `Pre` abbreviates the six instructions that model the security hazard:

$$\begin{aligned}\texttt{Pre} = {} & \texttt{file:=get-random-executable-file};\\ & \texttt{+first-line-of-file=1234567}; \backslash\#2; \texttt{prepend};\\ & \texttt{+trigger-pulled}; \texttt{do-damage}\end{aligned}$$

and `Next` models the remainder of the program. Applying behavior extraction on this program yields

$$\begin{aligned}|\texttt{CV}|_{pglc} &= |\texttt{Next}|_{pglc} \trianglelefteq \mathit{shft}.\texttt{test}(\texttt{CV}) \trianglerighteq |\texttt{Pre}; \#3; -\mathit{shft}.\texttt{test}(\texttt{CV}); \backslash\#8; \texttt{Next}|_{pglc}\\ &= |\texttt{Next}|_{pglc} \trianglelefteq \mathit{shft}.\texttt{test}(\texttt{CV}) \trianglerighteq |\texttt{Pre}; \texttt{Next}|_{pglc}\end{aligned}$$

So, $S \trianglelefteq \mathit{shft.test} \trianglerighteq f.m \circ S$ is indeed a faithful characterization of Cohen's impossibility result.

We note that even with the aid of universal computational power, the problem whether a thread has a security hazard (issues an $f.m$ call) is undecidable. This problem can be seen as a variant of the unsolvability of the Halting Problem, i.e., Turing's impossibility result.

Cohen's impossibility result needs the notion of a secure run (no security hazards), as well as a secure program or behavior (a thread that will have secure runs only). So, Cohen's impossibility result emerges if:

- secure runs exist,
- secure threads exist,
- there is a full match between these two,
- forecasting is possible.

Now there is a difficulty with forecasting: if *shft.test* returns false one hopes to proceed in such a way that the security hazard is avoided (why else do the test?). But that is not sound as was shown above. Thus we conclude: this type of security hazard forecasting is a problematic idea for the assessment of security hazards.

5 Security Hazard Risk Assessment

In this section we introduce a *security hazard risk assessment* tool, taking into account the above-mentioned considerations. This tool turns out to be a much more plausible modeling of testing the occurrence of security hazards. However, if we add divergence (the absence of halting) as a security risk, the tool can not exist.

The following security hazard risk assessment tool SHRAT with focus *shrat* may be conceived of as assessing a security hazard risk. In

$$P \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q$$

the test action *shrat.ok* returns true if P is secure, and false if P is insecure (then P is avoided and Q is done instead). This is a more rational test than *shft.test* because it tests only a single thread (its left argument). Using an external focus e, the test action *shrat.ok* in

$$(P_1 \trianglelefteq e.m \trianglerighteq P_2) \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q$$

yields true because $e.m$ is seen as an action that is beyond control of security hazard risk assessment.

For testing *shrat.ok* actions we can employ backtracking: at $P \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q$,

1. Temporarily remove thread (or loaded program),
2. Place P instead
3. Execute[3] until $\begin{cases} S \text{ or } D \text{ or } e.m & \Rightarrow \text{backtrack if possible, otherwise } \mathsf{true}, \\ f.m & \Rightarrow \text{backtrack if possible, otherwise } \mathsf{false}, \\ P' \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q' & \Rightarrow \text{restart 1 with } P' \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q', \end{cases}$

The backtracking in this algorithm may require the testing of threads that are no direct subthreads of the original one, e.g., in

$$(P_1 \trianglelefteq \mathit{shrat.ok} \trianglerighteq P_2) \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q$$

first the leftmost *shrat.ok* action is evaluated. If this yields false (so P_1 contains a security hazard), then $P_2 \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q$ is evaluated. For finite threads this is a terminating procedure and not problematic.

Evaluation of *shrat.ok* actions can be extended to a larger class of threads. A *regular* thread P_1 over B is defined by a finite system of equations over $\overline{P} = P_1, ..., P_n$ (for some $n \geq 1$) of the following form:

$$\begin{aligned} P_1 &= F_1(\overline{P}) \\ &\vdots \\ P_n &= F_n(\overline{P}) \end{aligned}$$

with $F_i(\overline{P}) ::= S \mid D \mid P_{i,1} \trianglelefteq a_i \trianglerighteq P_{i,2}$ for $P_{i,j} \in \{P_1, ..., P_n\}$ and $a_i \in B$. Consider $P_1 \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q$, thus $F_1(\overline{P}) \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q$. Again we can decide the outcome of the test action *shrat.ok* by doing a finite number of substitutions, linear in n. (Loops and divergence are not considered security hazards.) This leads to the following result:

[3] Here "execute" means that upon a test action a, both branches should be inspected.

Theorem 2. *For regular threads, the tool SHRAT is possible.*

We give a simple example: if

$$\begin{aligned} P_1 &= P_2 \trianglelefteq a \trianglerighteq P_1 \\ P_2 &= P_1 \trianglelefteq f.m \trianglerighteq P_1 \quad (= f.m \circ P_1), \end{aligned}$$

then *shrat.ok* in $(P_2 \trianglelefteq a \trianglerighteq P_1) \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q$ yields true if it does in both $P_1 \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q$ and $P_2 \trianglelefteq \mathit{shrat.ok} \trianglerighteq Q$. Obviously, it does not in the latter case, so this thread equals Q. A slightly more complex example (including the evaluation of the various *shrat.ok* tests):

$$\begin{aligned} P_1 &= P_2 \trianglelefteq \mathit{shrat.ok} \trianglerighteq S && (\mathsf{true}) \\ P_2 &= P_3 \trianglelefteq a \trianglerighteq P_4 \\ P_3 &= P_2 \trianglelefteq \mathit{shrat.ok} \trianglerighteq P_6 && (\mathsf{true}) \\ P_4 &= P_7 \trianglelefteq \mathit{shrat.ok} \trianglerighteq P_8 && (\mathsf{false}) \\ P_5 &= a \circ P_2 \\ P_6 &= f.m \circ P_2 \\ P_7 &= f.m \circ P_2 \\ P_8 &= a \circ S. \end{aligned}$$

Omitting the *shrat.ok*-tests, thread P_1 has behavior P as defined in

$$P = a \circ P \trianglelefteq a \trianglerighteq a \circ S.$$

Thus, evaluation of the reply of *shrat.ok* is decidable for regular threads. We conclude that Cohen's impossibility result does not apply in this case; apparently, that result is about forecasting. Of course, the decidability of the evaluation of *shrat.ok* actions is lost if a Turing Tape is used as a resource.

Divergence Risk Assessment. If we consider divergence as a security hazard, say by focus *drat* and resource DRAT (a Divergence Risk Assessment Tool), we have a totally different situation: in the thread defined by

$$P = P \trianglelefteq \mathit{drat.ok} \trianglerighteq S$$

we then obviously want that the test action *drat.ok* returns the answer false. It is well-known that in general, DRAT cannot exist, as it is equivalent with solving the Halting Problem.

Now, involving divergence as a security hazard in *shrat.ok* actions, we also find that in

$$P = P \trianglelefteq \mathit{shrat.ok} \trianglerighteq f.m \circ S$$

the test should yield false (otherwise divergence). However, this yields a problem: in

$$P = P \trianglelefteq \mathit{shrat.ok} \trianglerighteq S$$

this goes wrong: the halting problem (Turing's impossibility result) wins, and hence the backtracking model is not suitable anymore.

6 Conclusions

In [3], we provide a formal treatment of the (potential) interaction of a thread P with resources H_e, H_f and H_g. Notation for that situation is

$$((P/_e H_e)/_f H_f)/_g H_g \text{ or equivalently, } P/_e H_e/_f H_f/_g H_g.$$

In the present paper we considered all communications of P with a resource H_h implicit and wrote P instead. In other words, an expression like $P \trianglelefteq h.m \trianglerighteq Q$ as occurring in this paper is considered to abbreviate

$$(P \trianglelefteq h.m \trianglerighteq Q)/_h H_h,$$

and this type of interaction is formalized in [3]. In [4] we provide a process algebraic semantics of threads $(P \trianglelefteq h.m \trianglerighteq Q)/_h H_h$.

In [7] it is stated that in order to constitute a grid, a network must implement the Globus GRAM protocol. This is a far more constrained definition than the concept based definitions that occur in [8]. We do not use that characterization because it is too complex for a brief theoretical paper.

How do our results relate to GRAM? Secure resource allocation on the grid requires an underlying theoretical basis. What we put forward is that under a very simple definition of a resource allocation risk, Cohen's impossibility result need not constrain the options for automatic detection as much as one might think. On the contrary, if security risk avoidance is adapted as a strategy, Cohen's impossibility result disappears.

References

1. J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
2. J.A. Bergstra and C.A. Middelburg. Thread algebra for strategic interleaving. Computing Science Report 04-35, Eindhoven University of Technology, Department of Mathematics and Computing Science, November 2004.
3. J.A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.
4. J.A. Bergstra and A. Ponse. Execution architectures for program algebra. Logic Group Preprint Series 230, Dept. of Philosophy, Utrecht University, 2004.
5. F. Cohen. Computer viruses - theory and experiments, 1984. `http://vx.netlux.org/lib/afc01.html`. Version including some corrections and references: *Computers & Security* 6(1): 22-35, 1987.
6. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25-39, 2003.
7. S. Hwang and C. Kesselman. A flexible framework for fault tolerance in the grid. *Journal of Grid Computing*, 1(3):251-272, 2003.
8. Zs. Nemeth and V. Sunderam. Characterizing grids: attributes, definitions, and formalisms. *Journal of Grid Computing*, 1:9-23, 2003.