A. Avron  
B. Konikowska

**Proof Systems for Reasoning about Computation Errors**

**Abstract.** In the paper we examine the use of non-classical truth values for dealing with computation errors in program specification and validation. In that context, 3-valued McCarthy logic is suitable for handling lazy sequential computation, while 3-valued Kleene logic can be used for reasoning about parallel computation. If we want to be able to deal with both strategies without distinguishing between them, we combine Kleene and McCarthy logics into a logic based on a non-deterministic, 3-valued matrix, incorporating both options as a non-deterministic choice. If the two strategies are to be distinguished, Kleene and McCarthy logics are combined into a logic based on a 4-valued deterministic matrix featuring two kinds of computation errors which correspond to the two computation strategies described above. For the resulting logics, we provide sound and complete calculi of ordinary, two-valued sequents.

**Keywords:** three-valued logics, four-valued logics, parallel computation, lazy sequential computation, computation errors, non-deterministic matrices, sequent calculi.

1. Introduction

The use of computer software is ubiquitous in the present-day world. As a result, most of everyday activities, not only in the business or public spheres, but also in our private lives, rely — directly or indirectly — on the correct operation of some software. However, to ensure the correct, reliable operation of programs, we must first specify them in a correct and precise way, and then validate them, proving that they will operate fault-free and give the expected results.

Yet, as the computing practice clearly shows, instead of meeting the objectives set for them, programs do sometimes run into error states — so any logic used for program specification and validation must take this fact into consideration too. In the existing literature, this has been done in two ways. The first is based on using a partial logic, with formulas getting no value in case of a computation error — like in [BCJ84, Ho87, Owe85]. The second employs a three-valued logic with the third, “undefined” value representing a computation error — see e.g. [MC67, Bli91, KTB91, Ko93]. The second approach has become much more popular over years, as it provides more flexibility in reasoning by allowing us to define a three-valued
semantics of the considered program logic in the way best tailored to a given
application and the intended handling of errors. However, the drawback
is that all possible computation errors in various computing scenarios are
usually bundled together under a single error value and are handled in the
same way. Yet in fact there are two distinct types of computation errors, of
inherently different characters:

- **critical errors** which make the whole computation stop, or “hang up”,
  causing a total failure of the program
- **non-critical errors** which stop only part of the computation, and can
  be remedied by a success elsewhere in it

Clearly, the difference between them is quite fundamental from the prac-
tical viewpoint, especially for mission-critical software supporting the fun-
damental business processes of an enterprise. Hence to ensure the optimum
specification and validation of programs we should distinguish between crit-
ical and non-critical errors, and treat them in different ways. The aim of
our paper is to provide logics able to achieve the above.

A typical example of a critical error occurs in lazy, sequential computa-
tion, when we proceed from left to right, and the whole computation process
stops after encountering the first error. For example, if we are computing
the value \( v(\alpha \lor \beta) \) of the disjunction of \( \alpha \) and \( \beta \), and encounter an error
in computing \( v(\alpha) \), then because of the sequential order of the computation
we cannot proceed any further. As result, even if the computation of \( v(\beta) \)
would yield \( t \) (true), we will never learn this – and so we must necessarily
assign an error value to \( v(\alpha \lor \beta) \).

In turn, a non-critical error can be encountered in a parallel computation,
where an error encountered in one branch of a computation stops this branch
only, while the computation along other branches continues, and can still
give the desired result if one of those branches is a valid alternative to the
error-involving one. Hence in the preceding example this time we would get
\( v(\alpha \lor \beta) = t \), for to compute \( v(\alpha \lor \beta) \) we compute in parallel \( v(\alpha) \) and \( v(\beta) \),
and as soon as either computation yields the value \( t \), we assign this value to
the disjunction \( \alpha \lor \beta \).

Another example of critical and non-critical errors are the so-called ma-
chine error and the error resulting from infinite computation in a sequential
computation mode. Machine error, which consists in e.g. a syntax error
in the program, or the use of an argument outside a function domain, is
immediately signalled by the computer, which allows us to undertake some
corrective actions and continue the computation. In turn, an error resulting