Towards a Theory for Testing Non-terminating Programs

Arnaud Gotlieb and Matthieu Petit
INRIA Rennes - Bretagne Atlantique
35042 Rennes Cedex, France
Email: Arnaud.Gotlieb,Matthieu.Petit@irisa.fr

Abstract

Non-terminating programs are programs that legally perform unbounded computations. Though they are ubiquitous in real-world applications, testing these programs requires new theoretic developments as usual definitions of test data adequacy criteria ignore infinite paths. This paper develops a theory of program-based structural testing based on operational semantics. Reasoning at the program semantics level permits to cope with infinite paths (and non-feasible paths) when defining test data adequacy criteria. As a result, our criteria respect the first Weyuker's property on finite applicability, even for non-terminating programs. We discuss the consequences of this re-interpretation of test data adequacy criteria w.r.t. existing test coverage criteria.

I. Introduction

Motivations. Non-terminating programs are programs that legally perform unbounded computations. Nowadays, these programs are routinely exploited in many practical applications without provoking any damage. Just to name few, consider reactive programs that are used in nuclear safety systems and telecommunications, real time programs used in embedded critical systems, interactive programs used in operating systems and graphical user interfaces. As a more trivial example of non-terminating program, consider a program that computes all the decimals of the universal constant pi. This program must be thoroughly tested before running it to find the next unknown decimal of pi. Indeed, running this program will take a very long time and any unrevealed fault will cause it eventually to be restarted. Unfortunately, classical structural testing theory fails to provide the foundations for testing such programs as they usually consider infinite paths as erroneous computations [10]. Program-based structural testing is a popular testing method which requires selecting test data with respect to some adequacy criteria. These criteria depend on the structure of the program under test. Many distinct program-based adequacy criteria have been proposed by the past: control-flow adequacy criteria such as statement, branch or path coverage; data-flow adequacy criteria such as Rapps Weyuker's family of def/use criteria [7], etc. These criteria are usually defined in terms of finite control flow paths to execute [13]. When testing non-terminating programs, one faces the problem of considering infinite computation paths. Hence, structural testing of non-terminating programs requires a theory of infinite traces. Note that Tretmans developed in [8] a theory for testing reactive programs, called conformance testing. In this theory, the implementation is tested against a specification model defined as an input-output labelled transition system. Several conformance relations have been proposed that compare the traces of an implementation with respect to the traces of the model. However, conformance testing is essentially a functional testing theory that cannot be exploited in structural testing.

State-of-the-Art. Among the possible approaches used to compare program-based adequacy criteria, the axiomatic assessment plays a fundamental role as it looks for a consistent set of axioms able to define what an ideal test data adequacy criterion would be. The axiomatic assessment of criteria drew the attention of several researchers. Originally, Weyuker proposed a set of eleven axioms to assess several existing criteria [10] that were criticized by Zweben and Gourlay [14]. Parrish and Zweben [4], [5] formalized Weyuker's axioms, while Zhu and Hall [12] proposed an axiom system based on the mathematical theory of measurement. Among the original Weyuker's axioms, the *applicability property* was considered as the most fundamental [13]. Assuming the finiteness of representable points of input domain[1], the applicability property says that

This work is partially supported by ANR through the CAVERN project under ref. ANR-07-SESUR-003

[1] In [10], [11], the input domain is the set of all values for which the program halts in a non-erroneous state

for every program, there exists a finite adequate test set. When a test data adequacy criterion is not fulfilled by a test set, it is capital to determine whether that comes from the incompleteness of the test set, or from the fact that the program is not adequately testable. Unfortunately, most of the test data adequacy cited above fail to satisfy applicability due to the existence of infinite and non-feasible paths. To circumvent the problem of non-applicability, Frankl and Weyuker proposed the definition of an applicable family of criteria by requiring the test data to exercise only executable def-use pairs [2]. In their work, "executable" means that there exists a (finite) feasible path containing the corresponding element. The main obstacle in using such feasible criteria lies in the undecideability problem of determining whether a selected set of statements, branches or paths is feasible or not [9]. As quoted from Frankl and Weyuker's paper, "one is faced with a tradeoff between applicability and automatability".

Contributions. In this paper, we study structural testing based on trace partitioning of an operational semantics. Operational semantics describes how programs compute in stepwise fashion and the possible state transformations they perform. As program executions correspond to potentially infinite semantic traces, the proposed theorems address specifically the problem of non-terminating programs in structural testing. Reasoning at the operational semantics level allows to consider infinite paths as well as finite feasible paths when defining a set of new test data adequacy criteria. As a result, these criteria respect the Weyuker's finite applicability property and are applicable to programs that legally perform unbounded computations.

Structure of the paper. Section 2 contains the terminology and definitions required for understanding the rest of the paper. Section 3 formally defines a new family test data adequacy criteria based on operational semantics whereas section 4 compares these criteria with classical control flow criteria. Finally, Section 5 concludes this paper by discussing potiential extensions of the Theory.

II. Terminology and definitions

A. Syntax of programs under test

For the sake of clarity, all formal reasoning will be performed using a simple language with assignment, arithmetic expressions, conditional, loop, and output statements. This simple programming language abstracts real programming languages, but it is more general than it may appear. The input space of the programs of this language is not necessarily finite. The syntax of the language is given in FIG. 1.

A program of this language is a *block* that contains a finite set of statements *stmt*. Each statement or decision is

$$
\begin{array}{lll}
\textit{stmt} ::= & [X{:=}expr]_l & X \in \mathcal{V},\ l \in \mathcal{T} \\
\quad | & [\text{skip}]_l & \\
\quad | & [\text{print } expr]_l & \\
\quad | & \text{if } [test]_l \text{ then } \{block\} \text{ else } \{block\} & \\
\quad | & \text{while } [test]_l \text{ do } \{block\} & \\
block ::= & stmt\ ;\ \cdots\ ;\ stmt; &
\end{array}
$$

Figure 1. Syntax of the language

labelled with an element of $\mathcal{T}$, the set of labels. $\mathcal{V}$ denotes a finite set of variables and $\mathcal{V}_{\text{input}} \subseteq \mathcal{V}$ denotes the subset of input variables. Roughly speaking, the input domain of a program is the Cartesian product of the domains of all input variables. If k is the number of input variables and $\mathbf{D}$ denotes a domain of values, possibly unbounded such as the set of positive integers, then the input domain is the domain $\mathbf{D}^k$. Finally, $\mathcal{P}(E)$ denotes the powerset of E.

B. Operational semantics

The program semantics is given by a structural operational semantics (SOS) leading to a Labelled Transition System.

Definition 1 (LTS): A Labelled Transition System is a quadruple $\langle \Gamma, L, \longrightarrow, T \rangle$ consisting of a set Γ of states, a set L of labels, a ternary relation $\rightarrow \subseteq \Gamma \times L \times \Gamma$ of labelled transitions and a set of terminal states $T \subseteq \Gamma$.
A transition $(\gamma, \lambda, \gamma') \in \longrightarrow$ is noted infix: $\gamma \xrightarrow{\lambda} \gamma'$.

A state is a couple $\langle S, \rho \rangle$ where S denotes the remaining statements of a program execution and ρ is a total function $\mathcal{V} \rightarrow \mathbf{D} \cup \{undef\}$ which maps every variable of the program to a value. Note that $\rho(X) = undef$ when X is not defined in the program. Initial states in Γ are states that contain the entire program to be executed, noted $\langle P, \rho_0 \rangle$ where P is the program under test. The environment ρ_0 maps every input variable of the program to a value. Other variables are implicitly assigned the value $undef$ in ρ_0. As a consequence, an initial state $< P, \rho_0 >$ can be viewed as a test datum for the program P.

Definition 2 (Initial states): The set of initial states, noted Γ_{init}, is a subset of Γ, defined as follows:

$$\Gamma_{init} \stackrel{\text{def}}{=} \left\{ \langle P, \rho_0 \rangle \,|\, P \text{ is a program}, \rho_0 \in (\mathcal{V}_{\text{input}} \rightarrow \mathbf{D}) \right\}.$$

The transition relation $\rightarrow$ is defined according to the rules of FIG. 2. We assume that this SOS is deterministic, i.e. if $\gamma \xrightarrow{l} \gamma'$ and $\gamma \xrightarrow{l} \gamma''$ then $\gamma' = \gamma''$. A transfer function $[\![X := expr]\!]_\rho$ is used to model the modification of the environment by an assignment:

$$\forall V \in \mathcal{V},\ [\![X := expr]\!]_\rho(V) \stackrel{\text{def}}{=} \begin{cases} [\![expr]\!]_\rho & \text{if } X = V, \\ \rho(V) & \text{otherwise.} \end{cases}$$

$$[Assign]: \quad \langle [X := expr]_l, \rho\rangle \xrightarrow{l} [\![X := expr]\!]_\rho \qquad [Skip]: \quad \langle [\text{skip}]_l, \rho\rangle \xrightarrow{l} \rho$$

$$[Comp_1]: \quad \frac{\langle S_1, \rho_1\rangle \xrightarrow{l_1 \cdots l_n} \langle S_1', \rho_1'\rangle}{\langle S_1; S_2, \rho_1\rangle \xrightarrow{l_1 \cdots l_n} \langle S_1'; S_2, \rho_1'\rangle} \qquad [Comp_2]: \quad \frac{\langle S_1, \rho_1\rangle \xrightarrow{l} \rho_2}{\langle S_1; S_2, \rho_1\rangle \xrightarrow{l} \langle S_2, \rho_2\rangle}$$

$$[Cond^{true}]: \quad \frac{[\![test]\!]_\rho = \mathbf{T}}{\langle \text{ if } [test]_l \text{ then } \{S_1\} \text{ else } \{S_2\}, \rho\rangle \xrightarrow{l_T} \langle S_1, \rho\rangle}$$

$$[Cond^{false}]: \quad \frac{[\![test]\!]_\rho = \mathbf{F}}{\langle \text{ if } [test]_l \text{ then } \{S_1\} \text{ else } \{S_2\}, \rho\rangle \xrightarrow{l_F} \langle S_2, \rho\rangle}$$

$$[Loop^{true}]: \quad \frac{[\![test]\!]_\rho = \mathbf{T}}{\langle \text{ while } [test]_l \text{ do } \{S\}, \rho\rangle \xrightarrow{l_T} \langle S; \text{ while } [test]_l \text{ do } \{S\}, \rho\rangle}$$

$$[Loop^{false}]: \quad \frac{[\![test]\!]_\rho = \mathbf{F}}{\langle \text{ while } [test]_l \text{ do } \{S\}, \rho\rangle \xrightarrow{l_F} \rho}$$

Figure 2. SOS of the while language

$[\![expr]\!]_\rho$ denotes a function that maps to an environment where the expression $expr$ is valuated. For instance $[\![x + 3]\!]_\rho$ where $\rho(x) = 2$ maps to the constant 5.

In the LTS, a computation is called a trace.

Definition 3 (Trace): Let $\langle \Gamma, L, \rightarrow, T\rangle$ be an LTS, a trace is a sequence of successive transitions $(\gamma_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} \gamma_n \cdots)$ such that $\gamma_0 \in \Gamma_{init}$ and $\forall k, \gamma_k \xrightarrow{l_{k+1}} \gamma_{k+1}$.
A trace is finite whenever $\exists m \in \mathbb{N}$ such that $\gamma_m \in T$, the set of terminal states. It is infinite otherwise.

In the following, any of the two following notations will be used to denote a trace: (γ_0) or $\gamma_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} \gamma_n \cdots$.

The set of all traces of a program P is written as follows:

$$[\![P]\!] \stackrel{\text{def}}{=} \{(\gamma_0) | \gamma_0 = \langle P, \rho_0\rangle \in \Gamma_{init}\}$$

III. A new family of test data adequacy criteria based on operational semantics

A. Semantic paths

As exhaustive testing of all traces is usually impossible, one must resort to select a subset of traces to execute during the testing phase. This selection can be based on trace partitioning techniques [3]. The labels of the LTS permit to partition the set of traces:

Definition 4 (Traces equivalence):

$$\text{Let} \gamma_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} \gamma_n \cdots \text{ and } \gamma_0' \xrightarrow{l_1'} \cdots \xrightarrow{l_m'} \gamma_m' \cdots$$
$$\text{be two traces of } [\![P]\!] \text{ then } (\gamma_0) \sim (\gamma_0') \text{ iff } \forall i, \quad l_i = l_i'$$

With this definition, two traces having the same (infinite) sequence of labels or having the same sequence of statements are equivalent. Note that all the labels are not necessarily present on the transitions of the LTS. For example, the label associated to an unreachable statement of the program will never be used as the label of a transition. In fact, there does not exist any trace that contains a transition toward this statement in the LTS. $\sim$ is clearly an equivalence relation, and then $[\![P]\!]/\sim$ forms a partition of $[\![P]\!]$. Any element of $[\![P]\!]/\sim$, called a *semantic path*, is noted $l_1 l_2 \cdots l_n \cdots$.

Definition 5 (Semantic path):

$$l_1 l_2 \cdots l_n \cdots \stackrel{\text{def}}{=} \{(\gamma) \in [\![P]\!] | \exists \gamma_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} \gamma_n \cdots \text{ and } (\gamma) \sim (\gamma_0)\}$$

Roughly speaking, a semantic path is a subset of traces that share the same sequence of labels. It is worth noticing that semantic paths represent only executable control flow paths. In this theory, *executable* means there exists a trace (possibly infinite) that contains the considered element. Hence, semantic paths model not only executable finite control flow paths but also infinite ones. This definition differs from others [1], [7], [2], [13] where only finite paths are taken into account to determine executability.

B. The poset of all sequences

The definition of semantic paths leads to consider a language over the alphabet of labels $\mathcal{L}^*$. In other words, each semantic path $l_1 \cdots l_n \cdots$ defines a subset of finite sequences that can be executed by the program. These sequences, called k-words in the following, can be ordered as follows: $seq_1 \preceq seq_2$ denotes that seq_1 is a subword (or a *proper factor* [6]) of seq_2 where seq_1 and $seq_2 \in \mathcal{L}^*$. The rest of this section is devoted to the formal definition of a structure, noted $\mathcal{L}^\infty$, on which we define a new family of applicable test data adequacy criteria.

1) Definitions:

Definition 6 (k-word): Given an LTS associated with a program P, a sequence of k labels $l_i \cdots l_{i+k-1}$ that belongs to a semantic path of $[\![P]\!]/\sim$ is called a k-word. $\mathcal{L}^k$ denotes the set of all k-words.
By construction, k-words represent executable elements of the program as they are subpaths of execution traces. And, we turn on the definition of an order over sets of k-words.

Definition 7: (Order $\trianglelefteq$)
let S_1 be a set of k_1-words and S_2 be a set of k_2-words then $S_1 \trianglelefteq S_2$ iff for all k_1-word $s \in S_1$ there exists some k_2-word $t \in S_2$ such that $s \preceq t$.
The rationale behind this definition is that any k-word that is executed on a semantic path will also be executed by any set that include that k-word as a subpath. From the definition, it is clear that $\trianglelefteq$ is actually a partially ordered set as it is reflexive, anti-symmetric and transitive. Semantic paths themselves cannot be considered as k-words as they are of infinite length.

Definition 8: (The poset of all sequences $\mathcal{L}^\infty$)
Given an LTS associated with a program P, the poset of all sequences of P is defined as:

$$\mathcal{L}^\infty \stackrel{\text{def}}{=} \bigcup_{k=1}^{\infty} \mathcal{P}(\mathcal{L}^k) \cup \mathcal{P}([\![P]\!]/\sim)$$

In general, the poset of all sequences is not a lattice as least upper bounds are not unique. This relates to the well-known fact that it is usually not possible to find a unique set of paths of minimal size that covers a set of nodes in a control flow graph.

2) Covering k-words: A k-word $l_i l_{i+1} \cdots l_{i+k-1}$ is said to be covered by a trace $\gamma_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} \gamma_n \cdots$ iff the sequence of labels $l_i l_{i+1} \cdots l_{i+k-1} \preceq l_1 l_2 \cdots l_n \cdots$. By extension, we say that a k-word is covered by a semantic path when its traces cover the k-word. In all the cases, a semantic path covers an infinite set of k-words of increasing length: the semantic path $l_1 l_2 \cdots l_n l_\epsilon l_\epsilon l_\epsilon \cdots$ covers the following k-words:
$\{l_1, l_1 l_2, \cdots, l_1 l_2 \cdots l_n, l_1 l_2 \cdots l_n l_\epsilon, l_1 l_2 \cdots l_n l_\epsilon l_\epsilon \cdots\}$
$\cup \{l_2, l_2 l_3, \cdots, l_2 l_3 \cdots l_n, l_2 l_3 \cdots l_n l_\epsilon, l_2 \cdots l_n l_\epsilon l_\epsilon \cdots\}$
$\cup \cdots$

3) Examples: Consider first the program `foo` of Fig.3 that contains only finite traces. The decisions labelled l and l' can be instantiated to any pair of non-contradictory conditions. All the traces can be partitioned in 4 semantic paths shown in Fig.4. Let $L_1 = l_T l_2 l'_T l_6 l_\epsilon l_\epsilon \cdots$,

```
if ([test]_l) [skip]_l2; else [skip]_l4;
if ([test]_l') [skip]_l6; else [skip]_l8;
```

Figure 3. Program `foo`

$L_2 = l_T l_2 l'_F l_8 l_\epsilon l_\epsilon \cdots$, $L_3 = l_F l_4 l'_T l_6 l_\epsilon l_\epsilon \cdots$, $L_4 = l_F l_4 l'_F l_8 l_\epsilon l_\epsilon \cdots$ be the 4 semantic paths of P. The poset of all sequences associated with P is shown in Fig.5.

Suppose we look for a set of semantic paths that covers the 1-words l_2 and l'_T and the 2-word $l'_F l_8$. Upper bounds of minimal size can easily be computed:$\{L_1, L_2\}, \{L_1, L_4\}, \{L_2, L_3\}$. Picking up one of these semantic paths permits to get a covering set for l_2,l'_T and $l'_F l_8$.

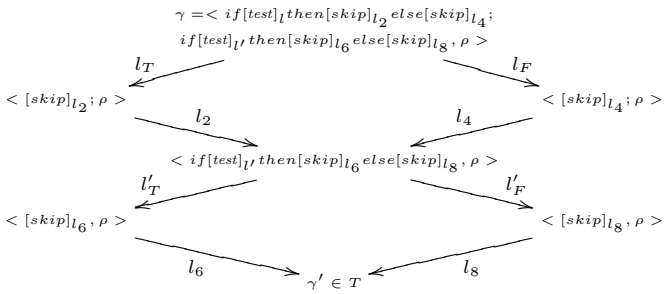


Figure 4. Abstract view of the LTS of `foo`

As second example, consider the non-terminating program `Liou` shown in Fig.6. `Liou` is intended to implement a function that computes the decimals of the Liouville number, which is defined as follows:

$$\text{Liou} = \sum_{n=1}^{\infty} \frac{1}{10^{n!}} = 0.1100010\ldots$$

This number played an important role in the theory of transcendental numbers.

```
[i := 1]a; [j := 1]a;
while ([i > 0]c){
      [i := i+1]d;
      if ([j + 1 == i]e)
            { [print 1]f; [j := j*i]g; }
      else { [print 0]h; } }
```

Figure 6. Program `Liou`

The two first statements have been assigned the same label to gain space. The `Liou` program exhibits a single infinite trace, and therefore a single semantic path noted

$$aac_T de_T fgc_T de_T fgc_T de_F hc_T d \cdots$$

The absence of 1-word c_F in the semantic path indicates that the program does not terminate, but determining such a result is undecideable in the general case.

C. The *All-k-words* test data adequacy criterion

Based on the poset of all sequences, we define a new family of test data adequacy criteria.

Definition 9 (All-k-words): A set Π of semantic paths satisfies the criterion *All-k-words* iff $\forall l_i \cdots l_{i+k-1} \in \mathcal{L}^k, \exists p \in \Pi$ such that $l_i \cdots l_{i+k-1} \preceq p$.
Covering this criterion leads to identify a set of semantic paths such that all k-words be executed. This notion of coverage is analogous to the definition of control flow coverage given in [13], but one of the difference comes from the fact that semantic paths represent only executable

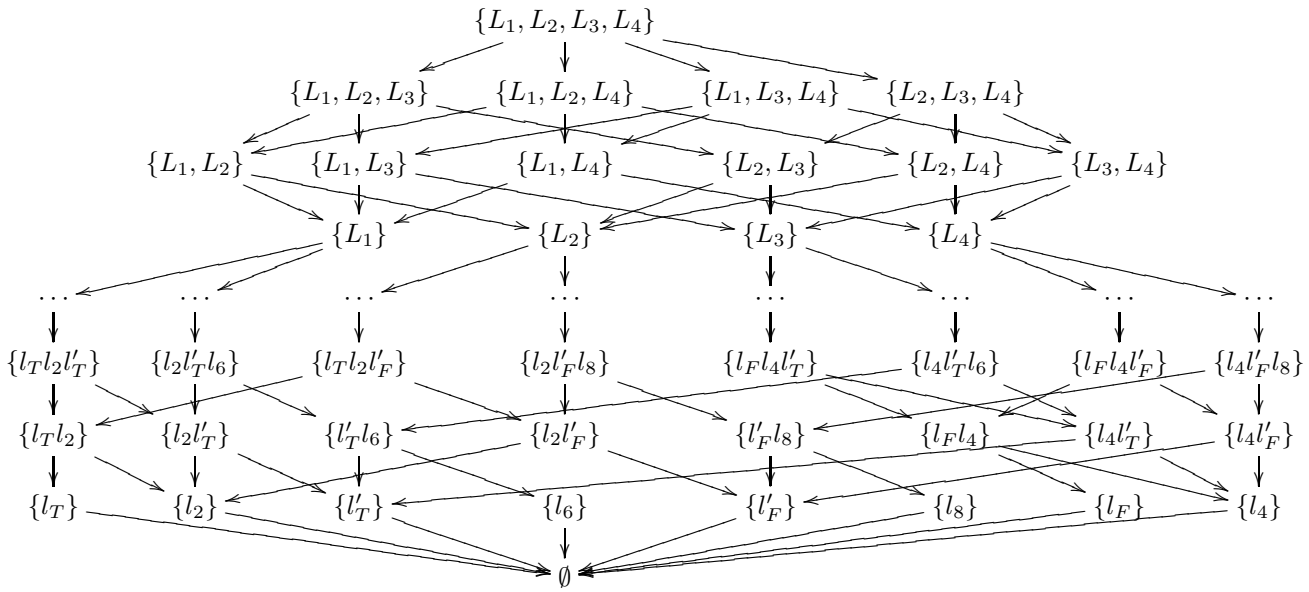


Figure 5. Poset of all sequences of program `foo`

control flow paths. Semantic paths can be understood as a dynamic flow paths. The following proposition holds:

Property 1: The adequacy criterion All-k-words respects the Weyuker's property on finite applicability
Finite applicability [10] is usually considered as the most fundamental axiom from Weyuker's system [13]. Proving this proposition requires showing that for any program P, there exists a finite set of initial states (test data) such that *All-k-words* is covered.

Proof: (Existence) Any k-word is a subword of a semantic path, hence there always exists a trace such that a given k-word is executed. Any trace is characterized by a single initial state in deterministic operational semantics, hence for any k-word one can find an initial state on which it is executed. (Finitess) The set of distinct labels is finite as the set of statements of the program is finite. Hence, the set of all label sequences of length k is finite. ■

This result is not surprising as the *All-k-words* criterion has been defined on the operational semantics of the program and k-words relate to program executions only. It is worth noticing that some k-words can sometimes be executed only on non-terminating computations. Thanks to our modification of the definition of executability, such cases are taken into account.

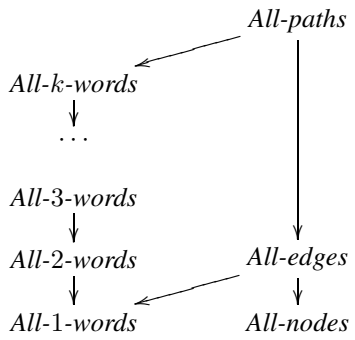
```
[i := 1]a;
while ([i >= 1]b) {
  [i := 0]c;
  if ([x == 1]d) [i := i+1]e; else [skip]f }
```

Figure 7. *All-edges* **doesn't subsume** *All-*1*-words*

IV. Comparison of test data adequacy criteria

Comparisons of test data adequacy criteria are usually based on the subsume ordering [1], [13]: criterion C_1 *subsumes* criterion C_2 iff for all programs p, all specifications s and all test sets t, t is adequate to C_1 for testing p w.r.t. s implies that t is adequate to C_2 for testing p w.r.t s. Using this oredring, we compare the *All-k-words* criteria with the three main classical control flow criteria *All-paths, All-edges* and *All-nodes*. According to the definition of *All-k-words*, a set of initial states is adequate iff its associated set of semantic paths covers all subwords of length k. Based on that, it is trivial that such a set will also cover all sequences of length $k-1, k-2, \cdots 1$. Hence, *All-k-words* subsumes *All-$k-1$-words*, which subsumes $\cdots$ which subsumes *All-*1*-words*. The comparison of the *All-k-words* criteria with the three main control flow criteria leads to a more surprising result. We found that *All-k-words* and *All-paths, All-edges* and *All-nodes* were incomparable in the general case. We explain this just by taking the comparison between *All-*1*-words* and *All-edges* as an example. Other comparisons follow the same reasoning. Firstly, a set of semantic paths covering all 1-words in a program is not guaranteed to cover all edges as there may exist non-feasible edges in the control flow graph. Hence, *All-*1*-words* does not subsume *All-edges*. Secondly, by using the definitions of [2], [13], an edge that could be executed only on infinite paths cannot be considered as covered, as control flow paths must be complete (from the entry node to the exit node of the CFG). For example, consider the program given in Fig.7. Any set of initial states that

covers *All*-1-*words* (such as $< P, \{\rho_0(x) = 0\} >, < P, \{\rho_0(x) = 1\} >$), does not cover *All-edges*. Executing the statement labelled e will indeed make the program looping forever. Hence, *All-edges* does not subsume *All*-1-*words* and both criteria are incomparable. However, if one states the hypothesis that a control flow element, such an edge or a path can also be covered by infinite control flow paths, the edge de in Fig.7 can be covered by the infinite path $abcdebcdebcde\cdots$. The *All-paths* criterion subsumes *All-k-words* for any k because covering all control flow paths (including the infinite paths) leads to cover any finite sequences of labels. On the contrary, *All-k-words* does not subsume *All-paths* as *All-k-words* covers only sequences of finite length. *All-edges* subsumes *All*-1-*words* as any 1-word is associated to an edge of the CFG. As previously said, the opposite is false as there are unreachable edges in the general case. In [2], an applicable family of data flow testing criteria is defined, including feasible version of classical control flow criteria noted *(All-paths)*, (All-edges)*, (All-nodes)**. Without the previous hypothesis, these feasible criteria are incomparable with *All-k-words* due to infinite control flow paths. On the contrary, with the hypothesis, *(All-edges)** is equivalent to *All*-1-*words* as non-feasible edges are ignored by the criterion. *(All-paths)** is not subsumed by *All-k-words* for a given k as infinite control flow paths are not be covered by *All-k-words*. So, the relations shown below remain true even if one replaces the control flow criteria with their feasible versions.



V. Conclusion

In this paper, we proposed a theory for testing non-terminating program based on trace partitioning of an operational semantics. A new family of program based adequacy criteria that cope with infinite and non-feasible paths and respect the first Weyuker's property on finite applicability was proposed. However, this theory suffers from a couple of current limitations. In particular, as test data are defined as initial states of a deterministic operational semantics, the proposed theory cannot deal with non-deterministic statements such as the parallel composition or random choice operators. These constructions require another definition of semantic paths and could compromise the finite applicability property of criteria. Moreover, our operational semantics does not contain error states and dealing with exceptions would require to adapt the definitions of semantic paths and adequacy criteria coverage.

References

[1] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil. A comparison of data flow selection criteria. In *Proc. of Int. Conf. on Soft. Eng.*, pp 244–251, London, UK, 1985.

[2] P.G. Frankl and E.J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Trans. on Soft. Eng.*, 14(10):1483–1498, 1988.

[3] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Proc. of 5th Int. Symp. on Static Analysis*, LNCS 1503, pp 200–214, Pisa, Italy, 1998.

[4] Allen Parrish and Stuart H. Zweben. Analysis and refinement of software test data adequacy properties. *IEEE Trans. on Soft. Eng.*, 17(6):565–581,1991.

[5] A.S. Parrish and S.H. Zweben. Clarifying some fundamental concepts in software testing. *IEEE Trans. on Soft. Eng.*, 19(7):742–746, 1993.

[6] D. Perrin and J.E. Pin. *Infinite words – Automata, Semigroups, Logic and Games*. Pure and Applied Mathematics Vol 141 – Elsevier, 2004.

[7] S. Rapps and E. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Trans. on Soft. Eng.*, 11(4):367–375, 1985.

[8] J. Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[9] E. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal of Computing*, 8(4):587–598, 1979.

[10] Elaine J. Weyuker. Axiomatizing software test data adequacy. *IEEE Trans. on Soft. Eng.*, 12(12):1128–1138,1986.

[11] Elaine J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Commun. ACM*, 31(6):668–675, 1988.

[12] H. Zhu and P. Hall. Test data adequacy measurement. *Soft. Eng. Journal*, 8(1):21–30, 1993.

[13] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–426, 1997.

[14] S.H. Zweben and J.S. Gourlay. On the adequacy of weyuker's test data adequacy axioms. *IEEE Trans. on Soft. Eng.*, 15(4):496–501, 1989.