# Data Flow Testing as Model Checking*

Hyoung Seok Hong, Sung Deok Cha
Department of Electrical Engineering and Computer Science and AITrc
Korea Advanced Institute of Science and Technology
{hshong,cha}@salmosa.kaist.ac.kr

Insup Lee, Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
{lee,sokolsky}@saul.cis.upenn.edu

Hasan Ural
School of Information Technology and Engineering
University of Ottawa
ural@site.uottawa.ca

## Abstract

*This paper presents a model checking-based approach to data flow testing. We characterize data flow oriented coverage criteria in temporal logic such that the problem of test generation is reduced to the problem of finding witnesses for a set of temporal logic formulas. The capability of model checkers to construct witnesses and counterexamples allows test generation to be fully automatic. We discuss complexity issues in minimal cost test generation and describe heurstic test generation algorithms. We illustrate our approach using CTL as temporal logic and SMV as model checker.*

## 1 Introduction

During the last two decades, there have been a number of data flow testing methods. Included are those proposed by Rapps and Weyuker[26], Ntafos[24], Ural[30], and Laski and Korel[22], which are originally developed for modules in procedural languages. These methods have been extended for interprocedural programs in procedural languages[13], object-oriented programmming languages[14], and requirements specification languages such as SDL[29, 31] and statecharts[17]. In data flow testing, we usually model a software as a flow graph which identifies the information of control flow and data flow in the software. We then establish certain associations between definitions and uses of variables required to be covered in a given coverage criterion by applying conventional data flow analysis upon the flow graph. Finally we select a finite number of paths which cover the associations as a test suite.

Model checking is a formal verification technique for determining whether a system model satisfies a property written in temporal logic and model checkers such as SMV[23] and SPIN[16] are already used on a regular basis for the verification of real-world applications. In addition to being automatic, an important feature of model checking is the ability to explain the success or failure of a temporal logic formula[5, 6, 15]. If a system model satisfies a formula, model checkers are capable of supplying an execution of the model as a witness demonstrating the success of the formula. Conversely, a counterexample is supplied when the model fails to satisfy the formula.

This paper presents a model checking-based approach to data flow testing. In our approach, the problems of data flow analysis and path selection in data flow testing are formulated in terms of model checking. We investigate four groups of coverage criteria in [26, 24, 30, 22] and characterize each coverage criterion by specifying the requirements of the coverage criterion using a set of temporal logic formulas such that the problem of test generation is reduced to the problem of finding witnesses for the set of formulas. The capability of model checkers to construct witnesses

*This research was supported in part by Advanced Information Technology Research Center at KAIST, NSF CCR-9988409, NSF CCR-0086147, NSF CCR-0209024, ARO DAAD19-01-1-0473, DARPA ITO MOBIES F33615-00-C-1707, and the Natural Sciences and Engineering Research Council of Canada under grant OGP00000976.

and counterexamples allows test generation to be fully automatic. As a by-product, the characterization enables us to discuss complexity issues in minimal cost test generation. This paper illustrates our approach using CTL[4] as temporal logic and SMV[23] as model checker. The main advantages of our approach may be summarized as follows: First, the approach enables test generation from large flow graphs whose size is limited by the capabilities of current model checkers. Second, the approach allows focusing on only high-level specifications of coverage criteria written in temporal logic. All the details about test generation algoritms and their implementations are hidden in model checkers. Third, the approach is language independent in that the temporal logic formulas employed in the approach are applicable with minor modifications to flow graphs constructed from various kinds of programming languages and requirements specification languages.

Connections between data flow analysis and model checking were made in [27, 28] which show that model checking can be used to solve various data flow analysis problems including the standard bit-vector problems. Our approach extends the work of [27, 28] in that data flow testing combines data flow analysis with the path selection problem. Recently, connections between test generation and model checking have been considered especially in specification-based testing. In [20], local and on-the-fly model checking algoritms are applied to test generation. In [32], SPIN is used for on-the-fly test generation. Test generation using the capability of model checker to construct counterexamples has been applied in several contexts. In [1], the application of model checking to mutation analysis is described. In [3, 9], tests are generated by constructing counterexamples for user-supplied temporal logic formulas. In [12], the capability of SMV and SPIN to construct counterexamples is applied to test generation for control flow oriented coverage criteria. No consideration is given to data flow testing in the above work.

In [18, 19], the authors discuss the application of model checking to test generation from requirements specifications for both control flow and data flow oriented coverage criteria. The approach in [18, 19] is based on the fact that the state space of a specification is often finite and hence one can use reachability graphs instead of flow graphs for test generation. On one hand, this paper extends [18, 19] by considering more comprehensive groups of data flow oriented coverage criteria. On the other hand, the flow-graph approach we advocate here can be seen as complementary to the reachability-graph approach in [18, 19]. In the flow-graph approach one can generate tests from programs or specifications with infinite state space because the values of variables are not expanded in flow graphs. It, however, requires posterior analysis such as symbolic execution or constraint solving to determine the executability of tests and for the selection of variable values which make tests executable. The reachability-graph approach can handle only finite state space but has the advantage that only executable tests are generated which obviates the necessity of posterior analysis.

Section 2 briefly reviews the basics of flow graph and CTL which are the model and logic employed in our approach, respectively. Section 3 characterizes the coverage criteria in [26, 24, 30, 22] by associating a CTL formula, parameterized with the propositions of a given flow graph, with each entity required to be covered in a given criterion. Each formula is defined in such a way that a flow graph satisfies the formula if and only if the flow graph has an execution covering the entity described by the formula. By finding witnesses for every formula in a given criterion, we generate a test suite satisfying the criterion. Section 4 discusses complexity issues in minimal cost test generation. Typically a CTL formula can have several executions as its witness. By selecting the right witness for each formula, one can minimize the size of the test suite. We show that two optimization problems of minimal cost test generation are NP-hard and describe heuristic test generation algorithms employing the capability of model checkers to construct counterexamples. We report the experimental results obtained by applying the heuristics to a moderate flow graph. In our experience with SMV, we were able to generate test suites from flow graphs containing dozens of variable definitions and uses in seconds. Finally, Section 5 concludes the paper with a discussion of future work.

## 2 Flow Graph and CTL

A *flow graph* $G = (V, v_s, v_f, A)$ is a directed graph where $V$ is a finite set of vertices; $v_s \in V$ is the start vertex; $v_f \in V$ is the final vertex; and $A$ is a finite set of arcs. A vertex represents a statement and an arc represents possible flow of control between statements. We adopt the following convention to decorate each vertex with data flow information. Let $x$ be a variable and $v$ be a vertex. We say that $x$ is *defined* at $v$, denoted by $d_v^x$, if $v$ represents a statement assigning a value to $x$. We say that $x$ is *used* at $v$, denoted by $u_v^x$, if $v$ represents a statement referencing $x$. We use *DEF*($v$) and *USE*($v$) to denote the sets of definitions and uses at $v$, respectively. A sequence $v_1 ... v_n$ of vertices is a *path* if $(v_i, v_{i+1}) \in A$ for $1 \leq i \leq n-1$. A path is *complete* if it starts from the start vertex $v_s$ and ends at the final vertex $v_f$. A *test sequence* is a complete path and a *test suite* is a finite set of test sequences. Figure 1 shows a program and its flow graph.

We view a flow graph as a Kripke structure $M = (Q, q_{init}, L, R)$ where $Q$ is a finite set of states; $q_{init} \in Q$ is the initial state; $L$: $Q \rightarrow 2^{AP}$ is the function labelling each state with a subset of the set $AP$ of atomic proposi-

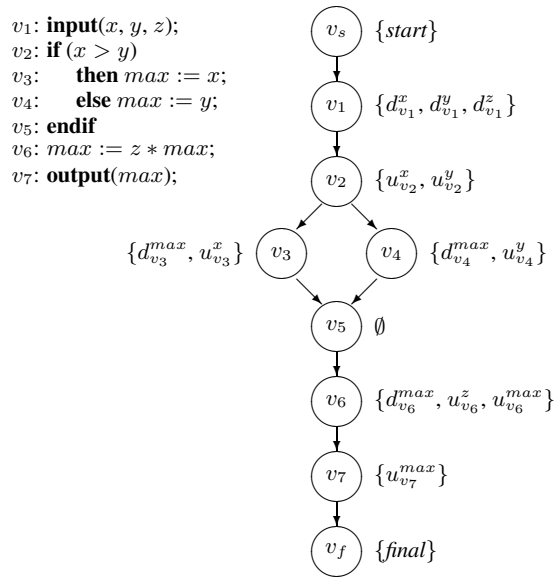



**Figure 1. An example of flow graphs**

tions; $R \subseteq Q \times Q$ is the transition relation which is total, i.e., for every state $q$, there is a state $q'$ such that $(q,q') \in R$. The Kripke structure $M(G)$ corresponding to a flow graph $G$ is $(V,\ v_s,\ L,\ A \cup \{(v_f, v_f)\})$ where $L(v_s)$ = {*start*}, $L(v_f)$ = {*final*}, and $L(v)$ = *DEF*($v$) $\cup$ *USE*($v$) for every $v \in V - \{v_s, v_f\}$. The tuple ($v_f$,$v_f$) is necessary to guarantee that the transition relation be total.

Now we give a brief and informal introduction to CTL. We refer to [4] for the formal syntax and semantics for CTL. Formulas in CTL are built from path quantifiers, modal operators, and standard logical operators. The path quantifiers are **A** (for all paths) and **E** (for some path). The modal opeators are **X** (next time), **F** (eventually), **G** (always), and **U** (until). For a CTL formula $f$ and a state $q$ of Kripke structure $M$, we write $M, q \models f$ ($q \models f$ when $M$ is understood) if $q$ satisfies $f$ and write $M \models f$ if $M, q_{init} \models f$. The meaning of CTL formulas can be understood as follows: "$q \models \mathbf{EX}p$" states that there is a path from $q$ such that $p$ holds at the next state; "$q \models \mathbf{EF}p$" states that there is a path from $q$ such that $p$ holds sometime in the future; "$q \models \mathbf{EG}p$" states that there is a path from $q$ such that $p$ holds globally in the future; "$q \models \mathbf{E}[p_1 \mathbf{U} p_2]$" states that there is a path from $q$ such that $p_1$ holds until $p_2$ holds and $p_2$ eventually holds in the future. ECTL is the exitential fragment of CTL where only the path quantifier **E** is allowed and negation is restricted to atomic propositions. ACTL is the dual universal fragment of CTL.

Symbolic model checkers for CTL such as SMV represent the state space and transition relation of Kripke structures in terms of binary decision diagrams (BDDs) and use a fixpoint characterization of CTL formulas to compute the set of states satisfying a formula. For example, the set of states satisfying $\mathbf{EF}p$ is a least fixpoint of the predicate transformer $\tau$: $2^Q \rightarrow 2^Q$ defined by $\tau(Z) = p \vee \mathbf{EX}Z$. The fixpoint computation requires standard logical operations, quantification over variables, and substitution of variables which can all be performed efficiently on BDDs.

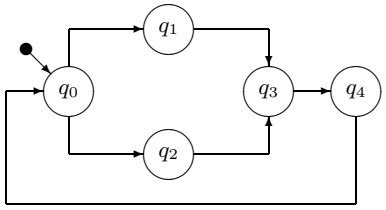
An important feature of model checking is the ability to construct witnesses and counterexamples. Algorithms for constructing linear witnesses and counterexamples, i.e., finite or infinite paths, were developed in [5, 15] and are widely used in current model checkers. Recently, Clarke *et al.*[6] made a formal definition of witnesses and counterexamples using simulation relation. Let $M$ be a Kripke structure, $f$ be a ECTL formula, and $g$ be a ACTL formula. If $M \models f$, a witness for $f$ is a Kripke structure $M'$ such that $M' \models f$ and $M$ simulates $M'$. Dually, if $M \not\models g$, a counterexample for $g$ is a Kripke structure $M''$ such that $M'' \not\models g$ and $M$ simulates $M''$. They also proposed to use tree-like structures as witnesses and counterexamples for a large class of branching-time temporal logics which do not have linear witnesses.

For the purpose of data flow testing, we are only interested in linear and finite witnesses and restrict ourselves to a subclass of ECTL, which we call WCTL, defined by: A ECTL formula $f$ is a *WCTL formula* if (i) $f$ contains only **EX**, **EF**, and **EU** and (ii) for every subformula of $f$ of the form $f_1 \wedge ... \wedge f_n$, every conjunct $f_i$ except at most one is an atomic proposition. For example, $\mathbf{EF}(p_1 \wedge \mathbf{EF}p_2)$ is in WCTL while $\mathbf{EF}p_1 \wedge \mathbf{EF}p_2$ is not. For a Kripke structure $M$ and a WCTL formula $f$ such that $M \models f$, we define the set of *witnesses* for $f$ with respect to $M$, denoted by $\mathbf{W}(M,f)$, as follows.

- $W(M,$*true*$) = Q$,
- $W(M,$*false*$) = \emptyset$,
- $W(M, p \wedge f) = \{q \mid q \models p\} * W(M,f)$,
- $W(M, f \vee g) = W(M,f) \cup W(M,g)$,
- $W(M,\mathbf{EX}f) = \{q_0 q_1 \mid q_1 \models f, (q_0, q_1) \in R\} * W(M,f)$,
- $W(M,\mathbf{EF}f) = \{q_0 ... q_n \mid q_n \models f, (q_i, q_{i+1}) \in R$ for all $0 \leq i \leq n-1\} * W(M,f)$,
- $W(M,\mathbf{E}[f\mathbf{U}g]) = \{q_0 ... q_n \mid q_i \models f$ for all $0 \leq i \leq n-1$, $q_n \models g$, $(q_j, q_{j+1}) \in R$ for all $0 \leq j \leq n-1\} * W(M,g)$,
- $\mathbf{W}(M,f) = \{\pi \in W(M,f) \mid \pi(0) = q_{init}\}$,

where $\Pi_1 * \Pi_2 = \{\pi \mid \exists i : \pi_i \in \Pi_1, \pi^i \in \Pi_2\}$, $\pi_i$ denotes the prefix of $\pi$ ending at $i$, and $\pi^i$ denotes the suffix of $\pi$ starting from $i$. Let $q_0 ... q_n$ be a witness in $\mathcal{W}(M,f)$ and $M'$ be its corresponding Kripke structure defined as ($Q$, $q_{init}$, $L$, $R - \{(q, q') \mid q = q_i \text{ for some } 1 \leq i \leq n-1\}$). It is not hard to see that $M' \models f$ and $M$ simulates $M'$.

Finally we extend the notion of witnesses to a set of WCTL formulas. Let $M$ be a Kripke structure and $F$ be a set of WCTL formulas. A *witness-set* $\Pi$ for $F$ with respect to $M$ is a set of finite paths such that, for every formula $f$ in $F$ with $M \models f$, there is a finite path $\pi$ in $\Pi$ that is a witness for $f$. It is easy to see that $\Pi$ is a witness-set for $F$ if and only if it is a witness-set for $\{f \in F \mid M \models f\}$. For example, in Figure 2 we observe that $\{q_0q_1q_3q_4q_0q_2q_3\}$, $\{q_0q_2q_3q_4q_0q_1q_3\}$, and $\{q_0q_1q_3,\ q_0q_2q_3\}$ are witness-sets for $\{\mathbf{EF}(a \wedge \mathbf{EF}c), \mathbf{EF}(b \wedge \mathbf{EF}c)\}$.




$L(q_0)$=∅, $L(q_1)$={$a$}, $L(q_2)$={$b$}, $L(q_3)$={$c$}, $L(q_4)$=∅

**Figure 2. An example of Kripke structures**

# 3 Characterizing Data Flow Oriented Coverage Criteria

This section characterizes four groups of coverage criteria[26, 24, 30, 22] in terms of witness-sets for WCTL formulas.

## 3.1 Rapps and Weyuker's Criteria

Rapps and Weyuker's criteria require certain associations between definitions and uses of the same variable be covered[26]. The criteria are extended with the notion of executability by Frankl and Weyuker[11]. We first adopt the following terminology. A path $(v, v_1, ..., v_n, v')$ is a *definition-clear path from $v$ to $v'$ with respect to variable $x$* if $n = 0$ or $x$ is not defined at $v_i$ for every $1 \leq i \leq n$. A pair ($d_v^x$, $u_{v'}^x$) is a *definition-use pair* (in short, du-pair) if there is a definition-clear path from $v$ to $v'$ with respect to $x$. For example, consider $d_{v_1}^x$ and $u_{v_3}^x$ in Figure 1. We observe that ($d_{v_1}^x$, $u_{v_3}^x$) is a du-pair through a definition-clear path $v_1v_2v_3$.

### 3.1.1 Characterization

We first describe how to generate a test sequence covering a pair ($d_v^x$, $u_{v'}^x$). The first step is to determine whether ($d_v^x$, $u_{v'}^x$) is a du-pair or not. For this, we associate the following WCTL formula with ($d_v^x$, $u_{v'}^x$).

**wctl**($d_v^x$, $u_{v'}^x$) = **EF**($d_v^x \wedge$ **EXE**[$\neg$*def*($x$) **U** ($u_{v'}^x \wedge$ **EF***final*)])

where *def*($v$) is the disjunction of all definitions of $x$. For example, in Figure 1 we have that *def*($x$) ::= $d_{v_1}^x$, *def*($y$) ::= $d_{v_1}^y$, *def*($z$) ::= $d_{v_1}^z$, and *def*($max$) ::= $d_{v_3}^{max} \vee d_{v_4}^{max} \vee d_{v_6}^{max}$. It is not hard to see that ($d_v^x$, $u_{v'}^x$) is a du-pair if and only if the Kripke structure $M(G)$ of a flow graph $G$ satisfies **wctl**($d_v^x$, $u_{v'}^x$). Hence the problem of determining whether ($d_v^x$, $u_{v'}^x$) is a du-pair is reduced to a model checking problem. After determining whether ($d_v^x$, $u_{v'}^x$) is a du-pair, we generate a test sequence covering it. It is also not hard to see that a test sequence covers a du-pair ($d_v^x$, $u_{v'}^x$) if and only if it is a witness for **wctl**($d_v^x$, $u_{v'}^x$). Hence the problem of generating a test sequence covering ($d_v^x$, $u_{v'}^x$) is reduced to the problem of finding a witness for **wctl**($d_v^x$, $u_{v'}^x$). For example, a test sequence covering the du-pair ($d_{v_1}^x$, $u_{v_3}^x$) is shown in Figure 3, which is also a witness for **EF**($d_{v_1}^x \wedge$ **EXE**[$\neg$*def*($x$) **U** ($u_{v_3}^x \wedge$ **EF***final*)]).

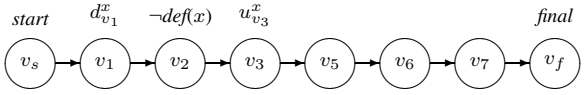



**Figure 3. A test sequence covering du-pair ($d_{v_1}^x$, $u_{v_3}^x$)**

Now we describe how to generate a set of test sequences for a set of pairs ($d_v^x$, $u_{v'}^x$) according to the criteria by Rapps and Weyuker. Basically we associate a formula **wctl**($d_v^x$, $u_{v'}^x$) with every pair ($d_v^x$, $u_{v'}^x$) and characterize each coverage criterion in terms of witness-sets for the formulas **wctl**($d_v^x$, $u_{v'}^x$). This reduces the problem of generating a test suite to the problem of finding a witness-set for a set of WCTL formulas.

A test suite $\Pi$ satisfies *all-defs coverage criterion* if, for every definition $d_v^x$ and some use $u_{v'}^x$, some definition-clear path with respect to $x$ from $v$ to $v'$ is covered by a test sequence in $\Pi$. Let *DEF*($G$) and *USE*($G$) be the sets of definitions and uses in $G$, respectively. A test suite $\Pi$ satisfies all-defs coverage criterion if and only if it is a witness-set for

$$\{ \bigvee_{u_{v'}^x \in USE(G)} \mathbf{wctl}(d_v^x, u_{v'}^x) \mid d_v^x \in DEF(G)\}.$$

A test suite $\Pi$ satisfies *all-uses coverage criterion* if, for every definition $d_v^x$ and every use $u_{v'}^x$, some definition-clear

path with respect to $x$ from $v$ to $v'$ is covered by a test sequence in $\Pi$. A test suite $\Pi$ satisfies all-uses coverage criterion if and only if it is a witness-set for

$$\{\mathbf{wctl}(d_v^x, u_{v'}^x) \mid d_v^x \in DEF(G), u_{v'}^x \in USE(G)\}.$$

In the worst case, the number of formulas can be quadractic in the size of a flow graph since the number of pairs $(d_v^x, u_{v'}^x)$ can be $O(n^2)$ in a flow graph of size $n$. For example, for all-uses coverage criterion in Figure 1 we associate 11 formulas with the pairs ($d_{v_1}^x$,$u_{v_2}^x$), ($d_{v_1}^x$,$u_{v_3}^x$), ($d_{v_1}^y$,$u_{v_2}^y$), ($d_{v_1}^y$,$u_{v_4}^y$), ($d_{v_1}^z$,$u_{v_6}^z$), ($d_{v_3}^{max}$,$u_{v_6}^{max}$), ($d_{v_3}^{max}$,$u_{v_7}^{max}$), ($d_{v_4}^{max}$,$u_{v_6}^{max}$), ($d_{v_4}^{max}$,$u_{v_7}^{max}$), ($d_{v_6}^{max}$,$u_{v_6}^{max}$), and ($d_{v_6}^{max}$,$u_{v_7}^{max}$). Among them, the formulas for ($d_{v_3}^{max}$,$u_{v_7}^{max}$), ($d_{v_4}^{max}$,$u_{v_7}^{max}$), and ($d_{v_6}^{max}$,$u_{v_6}^{max}$) are not satisfied in Figure 1, which means that the pairs are not du-pairs.

A test suite $\Pi$ satisfies *all-du-paths coverage criterion* if, for every definition $d_v^x$ and every use $u_{v'}^x$, every cycle-free definition-clear path with respect to $x$ from $v$ to $v'$ is covered by a test sequence in $\Pi$. Unlike other coverage criteria, all-du-paths coverage criterion cannot be characterized in terms of witness-sets. To generate test suites satisfying this criterion properly in our approach, we should be able to construct all cycle-free witnesses instead of only one for a given formula, which is beyond the capability of existing model checkers. In general, extending model checkers to construct all witnesses for a given formula or a subset of witnesses satisfying certain constraints is an open problem.

## 3.2 Ntafos' Criteria

Ntafos' criteria emphasize interactions between different variables[24]. Such interactions are captured in terms of sequences of alternating definitions and uses, called $k$-dr interactions. A sequence $[d_{v_1}^{x_1}\ u_{v_2}^{x_1}\ d_{v_2}^{x_2}\ u_{v_3}^{x_2}\ ...\ d_{v_n}^{x_n}\ u_{n+1}^{x_n}]$ is a *data flow chain* (df-chain) if, for every $1 \leq i \leq n$, $(d_{v_i}^{x_i}, u_{v_{i+1}}^{x_i})$ is a du-pair[30]. Note that the use $u_{v_{i+1}}^{x_i}$ and definition $d_{v_{i+1}}^{x_{i+1}}$ occur at the same vertex for every $1 \leq i \leq n$. A path $v_1\pi_1v_2\pi_2...v_{n+1}$ is an *interaction subpath* of a df-chain if, for every $1 \leq i \leq n$, $v_i\pi_iv_{i+1}$ is a definition-clear path from $v_i$ to $v_{i+1}$ with respect to $x_i$. A df-chain consisting of $k-1$ du-pairs, $k \geq 2$, is a $k$-definition/reference interaction ($k$-dr interaction) in the terminology of [24][1]. For example, in Figure 1 we observe that $[d_{v_1}^x\ u_{v_3}^x\ d_{v_3}^{max}\ u_{v_6}^{max}]$ is a 3-dr interaction which has $v_1v_2v_3v_5v_6$ as its interaction subpath.

### 3.2.1 Characterization

For a sequence $\kappa = [d_{v_1}^{x_1}\ u_{v_2}^{x_1}\ d_{v_2}^{x_2}\ u_{v_3}^{x_2}\ ...\ d_{v_{k-1}}^{x_{k-1}}\ u_k^{x_{k-1}}]$, $k \geq 2$, define $\mathbf{wctl}(\kappa)$ as follows.

[1] We do not require the variables $x_1, ..., x_n$ and the vertices $v_1, ..., v_{n+1}$ be distinct. This definition is consistent with that of Clarke et al.[7] and Ntafos[25] and is different from the original one[24] which requires the vertices to be distinct.

- if $\kappa$ is empty, then $wctl(\kappa)$ = **EF***final*,
- if $\kappa$ is $[d_{v_i}^{x_i}\ u_{v_{i+1}}^{x_i}] \cdot \kappa'$, then

  $wctl(\kappa) = d_{v_i}^{x_i} \wedge$ **EXE**$[\neg def(x_i)$ **U** $(u_{v_{i+1}}^{x_i} \wedge wctl(\kappa'))]$,
- $\mathbf{wctl}(\kappa)$ = **EF***wctl*$(\kappa)$.

By induction on the number of pairs $(d_{v_i}^{x_i},\ u_{v_{i+1}}^{x_i})$ in $\kappa$, it can be shown that $\kappa$ is a $k$-dr interaction if and only if the Kripke structure $M(G)$ of a flow graph $G$ satisfies $\mathbf{wctl}(\kappa)$. Moreover, a test sequence covers $\kappa$ if and only if it is a witness for $\mathbf{wctl}(\kappa)$. For example, a test sequence covering the 3-dr interaction $[d_{v_1}^x\ u_{v_3}^x\ d_{v_3}^{max}\ u_{v_6}^{max}]$ is shown in Figure 4, which is also a witness for **EF**$(d_{v_1}^x \wedge$ **EXE**$[\neg def(x)$ **U** $(u_{v_3}^x \wedge d_{v_3}^{max} \wedge$ **EXE**$[\neg def(max)$ **U** $(u_{v_6}^{max} \wedge$ **EF***final*$)])])$.

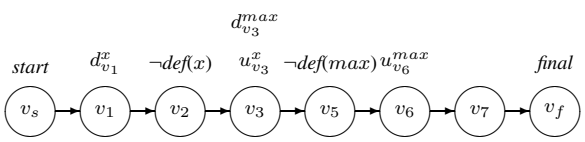



**Figure 4. A test sequence covering 3-dr interaction $[d_{v_1}^x\ u_{v_3}^x\ d_{v_3}^{max}\ u_{v_6}^{max}]$**

A test suite $\Pi$ satisfies *required k-tuples coverage criterion* if, for every $k$-dr interaction $\kappa$, some interaction subpath of $\kappa$ is covered by a test sequence in $\Pi$. A test suite $\Pi$ satisfies required $k$-tuples coverage criterion if and only if it is a witness-set for

$$\{\mathbf{wctl}([d_{v_1}^{x_1}u_{v_2}^{x_1}d_{v_2}^{x_2}u_{v_3}^{x_2}...d_{v_{k-1}}^{x_{k-1}}u_k^{x_{k-1}}])$$
$$\mid d_{v_i}^{x_i} \in DEF(G), u_{v_{i+1}}^{x_i} \in USE(G), 1 \leq i \leq k-1\}.$$

## 3.3 Ural' Criteria

Ural's criteria also emphasize interactions between different variables[30]. While Ntafos' criteria consider df-chains consisting of fixed number of du-pairs, Ural's criteria consider df-chains consisting of an arbitrary (but finite) number of du-pairs which start with inputs and end with outputs. The rationale here is to identify the functionality of a module in terms of the interactions with its environment by identifying the effects of inputs accepted from the environment on outputs offered to the environment. We say that a definition $d_v^x$ *affects* a use $u_{v'}^{x'}$ if

- either $x = x'$ and $(d_v^x,\ u_{v'}^{x'})$ is a du-pair or
- there is a use $u_{v''}^x$ such that $(d_v^x,\ u_{v''}^x)$ is a du-pair and there is a definition $d_{v''}^{x''}$, given in terms of $u_{v''}^x$, that affects $u_{v'}^{x'}$.

A pair ($d^x_v$, $u^{x'}_{v'}$) is an *affect-pair* if $d^x_v$ affects $u^{x'}_{v'}$. Among the particular affect-pairs of interest to Ural's criteria are those starting with inputs and ending with outputs, which we call *io-pairs*. We define an *input* as a definition at an **input** statement and an *output* as a use at an **output** statement. For example, $d^x_{v_1}$, $d^y_{v_1}$, $d^z_{v_1}$ are inputs and $u^{max}_{v_7}$ is an output in Figure 1. We observe that the input $d^x_{v_1}$ affects the output $u^{max}_{v_7}$ through the df-chain [$d^x_{v_1}$ $u^x_{v_3}$ $d^{max}_{v_3}$ $u^{max}_{v_6}$ $d^{max}_{v_6}$ $u^{max}_{v_7}$].

### 3.3.1 Simple Characterization

For an affect-pair ($d^x_v$, $u^{x'}_{v'}$), define *CHAIN*($d^x_v$, $u^{x'}_{v'}$) as the set of sequences $\kappa$ = [$d^{x_1}_{v_1}$ $u^{x_1}_{v_2}$ $d^{x_2}_{v_2}$ $u^{x_2}_{v_3}$ ... $d^{x_n}_{v_n}$ $u^{x_n}_{n+1}$] such that $d^{x_1}_{v_1} = d^x_v$ and $u^{x_n}_{v_{n+1}} = u^{x'}_{v'}$. In general, there may be multiple occurrences of the same pair ($d^{x_i}_{v_i}$, $u^{x_i}_{v_{i+1}}$) in $\kappa$ thereby causing the possibility of an infinite number of elements in *CHAIN*($d^x_v$, $u^{x'}_{v'}$). To ensure that *CHAIN*($d^x_v$, $u^{x'}_{v'}$) be finite, we consider its subset *SCHAIN*($d^x_v$, $u^{x'}_{v'}$) consisting of *simple* sequences in which at most one occurrence of each pair ($d^{x_i}_{v_i}$, $u^{x_i}_{v_{i+1}}$) is allowed.

A test suite $\Pi$ satisfies *all-inputs coverage criterion* if, for every input $i$ and some output $o$, an iteraction subpath of some simple df-chain in *SCHAIN*($d^x_v$, $u^{x'}_{v'}$) is covered by a test sequence in $\Pi$. Let *IN*($G$) and *OUT*($G$) be the sets of inputs and outputs in $G$, respectively. A test suite $\Pi$ satisfies all-inputs coverage criterion if and only if it is a witness-set for

$$\{\bigvee_{o\in OUT(G)}\ \bigvee_{\kappa\in SCHAIN(i,o)} \mathbf{wctl}(\kappa) \mid i \in IN(G)\}.$$

A test suite $\Pi$ satisfies *all-outputs coverage criterion* if, for every input $i$ and every output $o$, an iteraction subpath of some simple df-chain in *SCHAIN*($d^x_v$, $u^{x'}_{v'}$) is covered by a test sequence in $\Pi$. A test suite $\Pi$ satisfies all-outputs coverage criterion if and only if it is a witness-set for

$$\{\bigvee_{\kappa\in SCHAIN(i,o)} \mathbf{wctl}(\kappa) \mid i \in IN(G), o \in OUT(G)\}.$$

A test suite $\Pi$ satisfies *all-IO-df-chains coverage criterion* if, for every input $i$ and every output $o$, an iteraction subpath of every simple df-chain in *SCHAIN*($d^x_v$, $u^{x'}_{v'}$) is covered by a test sequence in $\Pi$. A test suite $\Pi$ satisfies all-IO-df-chains coverage criterion if and only if it is a witness-set for

$$\{\mathbf{wctl}(\kappa) \mid i \in IN(G), o \in OUT(G), \kappa \in SCHAIN(i,o)\}.$$

### 3.3.2 Fixpoint Characterization

The above characterization of all-inputs and all-outputs coverage criteria is naive in that we need to identify all simple sequences in *SCHAIN*($i$, $o$) for a given io-pair in order to generate a test sequence covering just one simple df-chain for the io-pair. A more faithful characterization should allow the generation of a test sequence without identifying all simple sequences in *SCHAIN*($i$, $o$) prior to test generation. Put another way, we like to model-check a new formula $\mathbf{Q}(d^x_v, u^{x'}_{v'})$ whose semantics is defined below without model-checking all formulas **wctl**($\kappa$).

> $q \models \mathbf{Q}(d^x_v, u^{x'}_{v'})$ if and only if $q \models \mathbf{wctl}(\kappa)$ for some $\kappa$ in *CHAIN*($d^x_v$, $u^{x'}_{v'}$).

We note that $\mathbf{Q}(d^x_v, u^{x'}_{v'})$ is not directly expressible in CTL because there is in general an infinite number of $\kappa$ in *CHAIN*($d^x_v$, $u^{x'}_{v'}$) and thus an infinite number of **wctl**($\kappa$).

The formula $\mathbf{Q}(d^x_v, u^{x'}_{v'})$ leads to a natural characterization of all-inputs and all-outputs coverage criteria as follows: A test suite $\Pi$ satisfies *all-inputs*$^+$ *coverage criterion* (resp. *all-outputs*$^+$ *coverage criterion*) if, for every input $i$ and some output $o$ (resp. every output $o$), an interaction subpath of some df-chain[2] in *CHAIN*($d^x_v$, $u^{x'}_{v'}$) is covered by a test sequence in $\Pi$. A test suite $\Pi$ satisfies all-inputs$^+$ coverage criterion if and only if it is a witness-set for

$$\{\bigvee_{o\in OUT(G)} \mathbf{Q}(i,o) \mid i \in IN(G)\}.$$

A test suite $\Pi$ satisfies all-outputs$^+$ coverage criterion if and only if it is a witness-set for

$$\{\mathbf{Q}(i,o) \mid i \in IN(G), o \in OUT(G)\}.$$

Finally we make a sketch of how to model-check $\mathbf{Q}(d^x_v, u^{x'}_{v'})$. Although the formula is not in CTL, it has a symbolic model checking algorithm similar to that of CTL because it can be characterized as a fixpoint of a predicate transformer. In fact, the formula is directly expressible in alternation-free mu-calculus which has a linear-time model-checking algorithm[8]. By the definition of affect-pairs, we have the following equivalence.

$$\mathbf{Q}(d^x_v,u^{x'}_{v'}) = \mathbf{EF}Q(d^x_v,u^{x'}_{v'})$$

$$Q(d^x_v,u^{x'}_{v'}) = (d^x_v \wedge \mathbf{EXE}[\neg \mathit{def}(v)\mathbf{U}(u^{x'}_{v'} \wedge \mathbf{EF}\mathit{final})]) \vee$$

$$(d^x_v \wedge \mathbf{EXE}[\neg \mathit{def}(v)\mathbf{U} \bigvee_{u^x_{v''}\in USE(G)} (u^x_{v''} \wedge Q(d^{x''}_{v''}, u^{x'}_{v'}))])$$

where $d^{x''}_{v''}$ is the definition of $x''$ occurring at $v''$ for some $x''$. Let $\tau$: $2^Q \rightarrow 2^Q$ be a predicate transformer defined by

$$\tau(Z) = (d^x_v \wedge \mathbf{EXE}[\neg \mathit{def}(v)\mathbf{U}(u^{x'}_{v'} \wedge \mathbf{EF}\mathit{final})]) \vee$$

$$(d^x_v \wedge \mathbf{EXE}[\neg \mathit{def}(v)\mathbf{U} \bigvee_{u^x_{v''}\in USE(G)} (u^x_{v''} \wedge Z[x''/x, v''/v])))$$

where $Z[x''/x, v''/v]$ is the formula obtained by replacing each occurrence of $x$ and $v$ in $Z$ by $x''$ and $v''$, respectively.

[2] We do not require a df-chain be simple here.

**Theorem 1** *$Q(d_v^x, u_{v'}^{x'})$ is a least fixpoint of $\tau$.*

PROOF Assume that $Z_1 \subseteq Z_2$. Then $\tau(Z_1) \subseteq \tau(Z_2)$ because $Z_1[x''/x, v''/v] \subseteq Z_2[x''/x, v''/v]$ and the modal operator **U** is monotonic. Hence $\tau$ is monotonic.

Let $Z_f$ be $Q(d_v^x, u_{v'}^{x'})$. It is easy to see that $Z_f = \tau(Z_f)$ and hence $Z_f$ is a fixpoint of $\tau$.

To prove that $Z_f$ is a least fixpoint of $\tau$, it is sufficient to show that $Z_f = \cup_i \tau^i$ where $\tau^0(Z) = Z$ and $\tau^{i+1}(Z) = \tau(\tau^i(Z))$. We first prove that $\tau^i$(*false*) $\subseteq Z_f$ for every $i$. Clearly, $\tau^0$(*false*) $\subseteq Z_f$. Assume that $\tau^i$(*false*) $\subseteq Z_f$. Because $\tau$ is monotonic, $\tau^{i+1}$(*false*) $\subseteq \tau(Z_f)$. Because $Z_f$ is a fixpoint of $\tau$, $\tau^{i+1}$(*false*) $\subseteq Z_f$. Hence we have the first direction $\cup_i \tau^i$(*false*) $\subseteq Z_f$. The other direction, $Z_f \subseteq \cup_i \tau^i$(*false*), is proved by induction on the number of du-pairs. Suppose that $q_0 \models Z_f$, then there is a path $q_0 q_1 ...$ covering a df-chain for $(d_v^x, u_{v'}^{x'})$. Let $j \geq 1$ be the number of du-pairs of the df-chain. We show that $q_0 \in \tau^j$(*false*) for every $j$. For the base case $j = 1$, we have that $x = x'$ and $q_0 \models d_v^x \wedge$ **EXE**[$\neg$*def*($v$) **U** ($u_{v'}^{x'} \wedge$ **EF***final*)]. Hence $q_0 \in \tau^1$(*false*). For the inductive step, suppose $q_0 \in \tau^j$(*false*) for $j = n$. Let $j = n + 1$ and $q_k$ be the state in $q_0 q_1 ...$ at which the first du-pair in the df-chain ends. Then there are $n$ du-pairs from $q_k$ and $q_k \in \tau^n$(*false*) by the induction hypothesis. Hence $q_0 \models (d_v^x \wedge$ **EXE**[$\neg$*def*($v$) **U** $\bigvee_{u_{v''}^x \in USE(G)} (u_{v''}^x \wedge \tau^n(false)[x''/x, v''/v])]$) and $q_0 \in \tau^{n+1}$(*false*). □

### 3.4 Laski and Korel's Criteria

Laski and Korel's criteria emphasize that a vertex may contain uses of several different variables in which each use may be reached by several different definitions[22]. Such definitions constitute the definition context of the vertex. Let $v$ be a vertex and $\{u_v^{x_1}, ..., u_v^{x_n}\}$ be a subset of *USE*($v$). An *ordered definition context* of $v$ with respect to $\{u_v^{x_1}, ..., u_v^{x_n}\}$ is a sequence $[d_{v_1}^{x_1} \ ... \ d_{v_n}^{x_n}]$ of definitions such that there is a subpath $v_1 \pi_1 v_2 \pi_2 ... \pi_n v$, called *orderded context subpath*, satisfying the following property: for every $1 \leq i \leq n$, $v_i \pi_i v_{i+1} ... \pi_n v$ is a definition-clear path from $v_i$ to $v$ with respect to $x_i$. A *definition context* of $v$ is a set of definitions, some permutation of which is an ordered definition context of $v$. For example, consider the vertex $v_6$ in Figure 1. $[d_{v_1}^z \ d_{v_6}^{max}]$ is an ordered definition context of $v_6$ with respect to $\{u_{v_6}^z \ u_{v_6}^{max}\}$ whose ordered context subpath is $v_1 v_2 v_3 v_5 v_6$.

#### 3.4.1 Characterization

Let $v$ be a vertex and $\{u_v^{x_1}, ..., u_v^{x_n}\}$ be a subset of *USE*($v$). For a sequence $\lambda = [d_{v_1}^{x_1} \ ... \ d_{v_n}^{x_n}]$ of definitions, define **wctl**($\lambda$) as follows.

- if $\lambda$ is empty, then

  *wctl*($\lambda$,*nodef*) = $u_v^{x_1} \wedge ... \wedge u_v^{x_n} \wedge$ **EF***final*,

- if $\lambda$ is $[d_{v_i}^{x_i}] \cdot \lambda'$, then

  *wctl*($\lambda$,*nodef*) = *nodef* $\wedge \ d_{v_i}^{x_i} \wedge$ **EXE**[*nodef'* **U** *wctl*($\lambda'$,*nodef'*)]),

  where *nodef'* = *nodef* $\wedge \neg$*def*($v_i$),

- **wctl**($\lambda$) = **EF***wctl*($\lambda$,*true*).

By induction on the number of definitions in $\lambda$, it can be shown that $\lambda$ is an ordered definition context of $v$ with respect to $\{u_v^{x_1}, ..., u_v^{x_n}\}$ if and only if the Kripke structure $M(G)$ of a flow graph $G$ satisfies **wctl**($\lambda$). Moreover, a test sequence covers $\lambda$ if and only if it is a witness for **wctl**($\lambda$). For example, a test sequence covering the ordered definition context $[d_{v_1}^z \ d_{v_3}^{max}]$ with respect to $\{u_{v_6}^x, u_{v_6}^{max}\}$ is shown in Figure 5, which is also a witness for **EF**($d_{v_1}^z \wedge$ **EXE**[$\neg$*def*($z$) **U** ($\neg$*def*($z$) $\wedge d_{v_3}^{max} \wedge$ **EXE**[($\neg$*def*($z$) $\wedge \neg$*def*($max$)) **U** ($u_{v_6}^z \wedge u_{v_6}^{max} \wedge$ **EF***final*)])]).

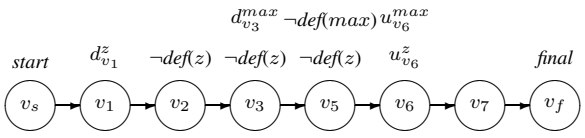



**Figure 5. A test sequence covering ordered context $[d_{v_1}^z \ d_{v_3}^{max}]$ with respect to $\{u_{v_6}^z, u_{v_6}^{max}\}$**

A test suite $\Pi$ satisfies *context coverage criterion* if, for every vertex $v$ and every definition context $dc$ of $v$, an ordered context subpath of $dc$ is covered by a test sequence in $\Pi$. A test suite $\Pi$ satisfies context coverage criterion if and only if it is a witness-set for

$$\{\mathbf{wctl}(\{d_{v_1}^{x_1}, ..., d_{v_n}^{x_n}\})$$

$$| \ v \in V, u_v^{x_i} \in USE(v), d_{v_i}^{x_i} \in DEF(G), 1 \leq i \leq n\}$$

where **wctl**($\{d_{v_1}^{x_1}, \ ..., \ d_{v_n}^{x_n}\}$) is defined as **wctl**($\lambda_1$) $\vee ... \vee$ **wctl**($\lambda_n$), where $\lambda_1, ..., \lambda_n$ are the permutations of $\{d_{v_1}^{x_1}, ..., d_{v_n}^{x_n}\}$.

A test suite $\Pi$ satisfies *ordered context coverage criterion* if, for every vertex $v$ and every ordered definition context $odc$ of $v$, an ordered context subpath of $odc$ is covered by a test sequence in $\Pi$. A test suite $\Pi$ satisfies ordered context coverage criterion if and only if it is a witness-set for

$$\{\mathbf{wctl}([d_{v_1}^{x_1}, ..., d_{v_n}^{x_n}])$$

$$| \ v \in V, u_v^{x_i} \in USE(v), d_{v_i}^{x_i} \in DEF(G), 1 \leq i \leq n\}.$$

## 4 Generating Minimal Cost Test Suites

This section discusses complexity issues in minimal cost test generation and describes heuristic algorithms and our experience with SMV for automatic test generation.

We wish to generate a minimal cost test suite $\Pi$ with respect to one of the two costs: (i) the number of test sequences in $\Pi$ or (ii) the total length of test sequences in $\Pi$. After finishing the execution of a test sequence, an implementation under test should be reset into its initial state from which another test sequence can be applied. It is appropriate to use the first cost if the reset operation is expensive, and the second one otherwise. For example, in Figure 2 we have that $\{q_0q_1q_3q_4q_0q_2q_3\}$ and $\{q_0q_2q_3q_4q_0q_1q_3\}$ are minimal in the number of test sequences, while $\{q_0q_1q_3$, $q_0q_2q_3\}$ is minimal in the total length of test sequences.

We first consider the Minimal Number Test Generation (MNTG) problem which is an optimization problem defined by: given a Kripke structure $M$ and a set $F$ of WCTL formulas, generate a minimal witness-set $\Pi$ in the number of witnesses in $\Pi$. We show this problem to be NP-hard by considering its decision problem MNTG′: given $M, F$, and a positive integer $k$, is there a witness-set $\Pi$ with $|\Pi| \leq k$?

**Theorem 2** *MNTG′ is NP-complete.*

PROOF On input $\langle\langle M, F, k\rangle, \Pi\rangle$ where $\Pi$ is a set of finite paths, we determine whether $\Pi$ is a witness-set with $|\Pi| \leq k$ for $F$ with respect to $M$ by verifying (i) $|\Pi| \leq k$, (ii) for every $\pi \in \Pi$, $\pi$ is a path of $M$, and (iii) for every $f \in F$, there is a witness $\pi \in \Pi$ for $f$. This is a polynomial time verifier and hence MNTG′ is in NP.

We next show that a NP-complete problem, called the Hitting Set problem, is polynomially reducible to MNTG′. The Hitting Set problem is defined by: given a collection of subsets $C_i$, $1 \leq i \leq n$, of a finite set $S$ and a positive integer $k$, is there a subset $S \subseteq S'$, called hitting set, such that $|S'| \leq k$ and containing at least one element from each $C_i$? Given an instance of the Hitting Set problem, we construct $M$ = ($Q$,$q_{init}$,$L$,$R$) and $F$ as follows:

- $Q = \{q_0\} \cup \{q_c \mid c \in \bigcup C_i\}$,
- $q_{init} = q_0$,
- $L(q_0) = \emptyset$ and, for every $q_c$, $i \in L(q_c)$ if and only if $q_c \in C_i$,
- $R = \{(q_0, q_c) \mid c \in \bigcup C_i\}$, and
- $F = \{\mathbf{EF}i \mid 1 \leq i \leq n\}$.

This reduction is in polynomial time. Clearly, $c \in C_i$ if and only if $q_0q_c$ is a witness for $\mathbf{EF}i$. Therefore, a subset $S'$ of $S$ is a hitting set with $|S'| \leq k$ for the collection of $C_i$ if and only if $\Pi = \{q_0q_s \mid s \in S'\}$ is a witness-set with $|\Pi| \leq k$. □

Second we consider the Minimal Length Test Generation (MNTG) problem defined by: given $M$ and $F$, generate a minimal witness-set $\Pi$ in the total length of witnesses in $\Pi$. Its decision problem MNTG′ is defined by: given $M$, $F$, and $k$, is there a witness-set $\Pi$ with $\sum_{\pi\in\Pi} |\Pi| \leq k$?

**Theorem 3** *MLTG′ is NP-complete.*

PROOF We use the same reduction as in Theorem 2. Since all paths in the Kripke structure $M$ are of length one, the minimum total length of $\Pi$ is achieved when $\Pi$ contains the minimum number of witnesses. Therefore, a solution for the MLTG′ problem in this case will yield the same witness-set which is also a solution to the MNTG′ problem. □

Because of NP-hardness, we do not expect an optimal solution to the minimal cost test generation problems. Instead we describe a heuristic algorithm which can be applied to both MNTG′ and MLTG′ problems. Figure 6 describes the algorithm in a generic fashion without being specific about any coverage criteria. We directly employ the capability of model checkers to construct counterexamples by exploiting the fact that a witness for a WCTL formula is also a counterexample for its negation.

```
INPUT: a Kripke structure M and a coverage criterion C
OUTPUT: a test suite Π satisfying C

1: Π := ∅;
2: mark every entity required to be covered in C as uncovered;
3: repeat
4:     choose an entity marked as uncovered;
5:     let f be the WCTL formula for the entity;
6:     model-check f against M;
7:     if M ⊭ f
8:         mark the entity as untestable;
9:     else
10:        let π be a witness for f
           (or equivalently a counterexample for ¬f);
11:        let En(π) be the set of entities covered by π;
12:        mark every entity in En(π) as covered;
13:        for every π′ ∈ Π such that En(π′) ⊆ En(π)
14:            Π := Π − {π′};
15:        Π := Π ∪ {π};
16: until every entity is marked as covered or untestable
17: return Π;
```

**Figure 6. A heuristic algorithm for test generation**

Basically, we construct a witness for every formula by finding a counterexample for its negation. The algorithm is locally optimal in the sense that model checkers such as SMV find a shortest counterexample for a given formula through breadth-first search of the state space. However, it

**Table 1. Results of test generation without heuristics**

| | all-defs | all-uses | required 3-tuples | all-inputs | all-outputs | all-IO-df-chains | context | orderded context |
|---|---|---|---|---|---|---|---|---|
| Number of formulas employed | 24 | 105 | 139 | 6 | 22 | 62 | 105 | 229 |
| Number of formulas satisfied | 24 | 53 | 60 | 5 | 6 | 6 | 58 | 63 |
| Number of test sequences | 24 | 53 | 60 | 5 | 6 | 6 | 58 | 63 |
| Total length of test sequences | 487 | 1210 | 2007 | 92 | 116 | 116 | 1305 | 1456 |
| Execution time (sec) | 0.44 | 0.88 | 1.56 | 0.34 | 0.31 | 0.23 | 1.48 | 1.2 |
| Number of BDD nodes | 3130 | 3268 | 3200 | 1997 | 2043 | 2000 | 3418 | 3342 |

**Table 2. Results of test generation with heuristics**

| | all-defs | all-uses | required 3-tuples | all-inputs | all-outputs | all-IO-df-chains | context | orderded context |
|---|---|---|---|---|---|---|---|---|
| Number of formulas employed | 24 | 105 | 139 | 6 | 22 | 62 | 105 | 229 |
| Number of formulas satisfied | 24 | 53 | 60 | 5 | 6 | 6 | 58 | 63 |
| Number of test sequences | 4 | 14 | 23 | 2 | 3 | 3 | 14 | 14 |
| Total length of test sequences | 106 | 385 | 878 | 39 | 63 | 63 | 385 | 385 |
| Execution time (sec) | 0.86 | 5.13 | 14.39 | 0.37 | 0.40 | 0.35 | 8.08 | 8.23 |
| Number of BDD nodes | 5823 | 21498 | 55543 | 1564 | 2185 | 2183 | 22276 | 22091 |

would generate a number of redundant witnesses because a witness may cover more than one entity at the same time. The algorithm removes such redundant witnesses by considering only uncovered states (Line 4) and by removing an existing witness if all the entities covered by it are also covered by a new witness (Line 13 and 14). Finally we note that the computation of $En(\pi)$ in Line 11 can be done by viewing a witness $q_0...q_n$ as the single-path Kripke structure $(\{q_0, ..., q_n\}, q_0, L, \{(q_0, q_1), ..., (q_{n-1}, q_n), (q_n, q_n)\})$ and model-checking the WCTL formula against the Kripke structure for every entity in the coverage criterion.

We describe the experimental results obtained by applying our approach to a moderate flow graph. In the experiment, we used SMV on a standard PC and adopted the flow graph made by Ural et al. (see Figure 1 in [31]). The flow graph consists of 39 vertices, 46 arcs, 11 variables, 24 definitions, and 36 uses. We first applied our approach to the flow graph without heuristics. Table 1 summarizes the experimental results. The second row gives the number of WCTL formulas associated with each coverage criterion and the third row gives the number of WCTL formulas satisfied in the flow graph. The number of test sequences given in the fourth column is equivalent to that of formulas satisfied, because we did not remove redundant test sequences. A test sequence has a single test purpose, that is, it is intended to cover only one entity. The fifth row gives the total length of test sequences. Finally, the sixth and seventh rows give the execution time in seconds and the number of BDD nodes, respectively. Table 2 summarizes the results of test generation with heuristics in Figure 6. We removed redundant test sequences according to the heuristics and were able to significantly reduce the number and total length of test sequences with the cost of increased execution time and BDDs. For example, only 14 test sequences are necessary for all-uses coverage criterion. The test sequences cover all entities desribed by 53 formulas. The execution time was calculated by adding up the execution time of Line 6 and Line 11 in Figure 6. The number of BDD nodes was figured out in a similar manner.

## 5 Summary and Discussion

We have showed that test generation from flow graphs for data flow oriented coverage criteria can be automated by model checking. We investigated four groups of coverage criteria in [26, 24, 30, 22]. For a given coverage criterion, a CTL formula is associated with every entity required to be covered in the coverage criterion. A witness for the CTL formula corresponds to a test sequence covering the entity described by the formula and a witness-set for the formula set corresponds to a test suite satisfying the criterion. We also discussed complexity issues in minimal cost test generation and described heuristics for automatic test generation.

As mentioned before, one of the advantages of our approach is language independence. We are currently working on both program-based and specification-based test generation for real-world applications. In our preliminary experiments for specification-based testing, we constructed flow graphs from statecharts or a set of communicating state machines by following the methods in [17, 31] and were able

to generate test suites from flow graphs with $2^{20}$ vertices and one hundred formulas in one minute and flow graphs with $2^{70}$ vertices and one hundred formulas in one hour. Of course, further experiments are compulsory to demonstrate the feasibility of our approach when applied to data flow testing with huge state space.

We are planning to extend our approach for interprocedural programs and object-oriented programs. Data flow testing of such programs is more complicated due to procedure call/return, recursion, and reference parameters as well as global variables. Data flow testing methods for such programs were proposed in [13, 14] which employ interprocedural data flow analysis. Recently, the problem of interprocedural data flow analysis has been formulated as a model checking problem[2, 10]. Combining both work together may be a starting point for developing a model checking-based approach to data flow testing of interprocedural programs and object-oriented programs.

We showed that a subclass of CTL, which we call WCTL, is expressive enough to characterize a number of data flow oriented coverage criteria except those by Ural[30]. For Ural's criteria, we extended WCTL with least fixpoints so that model checking of the resulting logic can be readily implemented in existing model checkers for CTL such as SMV. To characterize the criteria considered in this paper in a more uniform way, it is necessary to employ a logic more powerful than CTL. We are currently working with a subclass of mu-calculus[21], more specifically alternation-free mu-calculus[8], which supports the explicit use of fixpoint operators.

We cannot directly use linear time temporal logic for the characterization of data flow oriented coverage criteria, because it requires existential quantification over paths. It is, however, possible to construct a witness for a WCTL formula using linear time model checkers by exploiting the fact that a path is a witness for a WCTL formula if and only if the path is a counterexample for its negation. For example, we can construct a witness for a WCTL formula $\mathbf{EFEF}p$ by finding a counterexample for $\neg\mathbf{EFEF}p = \mathbf{AGAG}\neg p$, which is in turn equivalent to the LTL formula $\mathbf{AGG}\neg p$. This opens the possibility of applying linear time model checkers such as SPIN to data flow testing.

## References


[1] P. Ammann, P. Black, and W. Majurski, "Using Model Checking to Generate Tests from Specifications," in *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods*, pp. 46-54, 1998.

[2] T. Ball and S.K. Rajamani, "Bebop: a Symolic Model Checker for Boolean Programs," *SPIN Workshop '00*, Vol. 1885 of LNCS, pp. 113-130, Springer-Verlag, 2000.

[3] J. Callahan, F. Schneider, and S. Easterbrook, "Specification-based Testing Using Model Checking," in *Proceedings of 1996 SPIN Workshop*, also Technical Report NASA-IVV-96-022, West Virginia Univeristy, 1996.

[4] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, 8(2):244-263, Apr. 1986.

[5] E.M. Clarke, O. Grumberg, K. McMillan, and X. Zhao, "Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking," in *Proceedings of the 32nd Design Automation Conference*, pp. 427-432, 1995.

[6] E.M. Clarke, S. Jha, Y. Lu, and H. Veith, "Tree-Like Counterexamples in Model Checking," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 19-29, 2002.

[7] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Transactions on Software Engineering*, 15(11):1318-1332, Nov. 1989.

[8] R. Cleaveland and B. Steffen, "A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus," *Formal Methods in System Design*, Vol. 2, pp. 121-147, 1993.

[9] A. Engels, L. Feijs, and S. Mauw, "Test Generation for Intelligent Networks Using Model Checking," *TACAS '97*, Vol. 1217 of LNCS, pp. 384-398, Springer-Verlag, 1997.

[10] J. Esparza and J. Knoop, "An Automata-Theoretical Approach to Interprocedural Data-Flow Analysis," *FOSSACS '99*, Vol. 1578 of LNCS, pp. 14-30, Springer-Verlag, 1999.

[11] P.G. Frankl and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, 14(10):1483-1498, Oct. 1988.

[12] A. Gargantini and C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," in *Proceedings of ESEC/FSE '99* pp. 146-162, 1999.

[13] M.J. Harrold and M.L. Soffa, "Interprocedural Data Flow Testing," in *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification*, pp. 158-167, 1989.

[14] M.J. Harrold and G. Rothermel, "Performing Data Flow Testing on Classes," in *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 154-163, 1994.

[15] R. Hojati, R.K. Brayon, and R.P. Kurshan, "BDD-based Debugging of Designs Using Language Containment and Fair CTL," *CAV '99*, Vol. 697 of LNCS, pp. 41-58, Springer-Verlag, 1993.

[16] G.J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997.

[17] H.S. Hong, Y.G. Kim, S.D. Cha, D.H. Bae, and H. Ural, "A Test Sequence Selection Method for Statecharts," *Journal of Software Testing, Verification, and Reliability*, 10(4):203-227, Dec. 2000.

[18] H.S. Hong, I. Lee, O. Sokolsky, and S.D. Cha, "Automatic Test Generation from Statecharts Using Model Checking," in *Proceedings of the First Workshop on Formal Approaches to Testing of Software*, pp. 15-30, 2001.

[19] H.S. Hong, I. Lee, O. Sokolsky, and H. Ural, "A Temporal Logic Based Theory of Test Coverage and Generation," *TACAS '02*, Vol. 2280 of LNCS, pp. 327-341, Springer-Verlag, 2002.

[20] T. Jeron and P. Morel, "Test Generation Derived From Model Checking," *CAV '99*, Vol. 1633 of LNCS, pp. 108-121, Springer-Verlag, 1999.

[21] D. Kozen, "Results on the Propositional Mu-Calculus," *Theoretical Computer Science*, 27:333-354, 1983.

[22] J.W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Transactions on Software Engineering*, 9(5):347-354, May 1983.

[23] K.L. McMillan, *Symbolic Model Checking – an Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.

[24] S.C. Ntafos, "On Required Element Testing," *IEEE Transactions on Software Engineering*, 10(11):795-803, Nov. 1984.

[25] S.C. Ntafos, "A Comparison of Some Structural Testing Strategies," *IEEE Transactions on Software Engineering*, 14(6):868-874, June 1988.

[26] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, 11(4):367-375, Apr. 1985.

[27] D.A. Schmidt and B. Steffen, "Data-flow Analysis as Model Checking of Abstract Interpretations," *SAS '98*, Vol. 1503 of LNCS, pp. 351-380, Springr-Verlag, 1998.

[28] B. Steffen, "Generating Data-Flow Analysis Algorithms for Modal Specifications," *Science of Computer Programming*, 21:115-139, 1993.

[29] H. Ural and B. Yang, "A Test Sequence Generation Method for Protocol Testing," *IEEE Transactions on Communications*, 39(4):514-523, Apr. 1991.

[30] H. Ural, "IO-df-chains criterion," ISO Working Group on Formal Methods on Conformance Testing, Draft International Standard, Sept. 1993.

[31] H. Ural, K. Saleh, and A. Williams, "Test Generation Based on Control and Data Dependencies within System Specifications in SDL," *Computer Communications*, 23(7):609-627, Mar. 2000.

[32] R. de Vries and J. Tretmans, "On-the-Fly Conformance Testing Using SPIN," *International Journal on Software Tools for Technology Transfer*, 2(4):382-393, 2000.