J Supercomput (2018) 74:1655–1675
https://doi.org/10.1007/s11227-017-2187-3

K-DT: a formal system for the evaluation of linear data dependence testing techniques

Jie Zhao[1] · **Rongcai Zhao**[1,2]

Published online: 15 November 2017

Abstract The power of data dependence testing techniques of a parallelizing compiler is its essence to transform and optimize programs. Numerous techniques were proposed in the past, and it is, however, still a challenging problem to evaluate the relative power of these techniques to better understand the data dependence testing problem. In the past, either empirical studies or experimental evaluation results are published to compare these data dependence testing techniques, being not able to convince the research community completely. In this paper, we show a theoretical study on this issue, comparing the power on disproving dependences of existing techniques by proving theorems in a proposed formal system K-DT. Besides, we also present the upper bounds of these techniques and introduce their minimum complete sets. To the best of our knowledge, K-DT is the first formal system used to compare the power of data dependence testing techniques, and this paper is the first work to show the upper bounds and minimum complete sets of data dependence testing techniques.

Keywords Parallelizing compiler · Data dependence testing technique · Predicate logic · Minimum complete set

National Natural Science Foundation of China (61702546).

✉ Jie Zhao
zjbc2005@163.com

[1] National Digital Switching System Engineering and Technological Research Center, Zhengzhou 450001, People's Republic of China

[2] Zhongyuan University of Technology, Zhengzhou 450007, People's Republic of China

1 Introduction

As the fundamental principle of parallelizing compilers to detect parallelism and optimize programs, dependence testing techniques began to attract programmers' attentions from the very beginning of parallelizing compilers. The semantic of the generated programs of a parallelizing compiler can be preserved only when the Fundamental Theorem of Dependence [1] holds, meaning the generated code can obtain the same result as the original program. The power of data dependence testing techniques therefore impacts heavily on the ability of a parallelizing compiler to exploit the parallelism of programs.

To transform a serial program into parallel, compiler developers proposed static parallelizing compilers at first. A static parallelizing compiler is usually composed of three parts: (1) It first uses data dependence techniques to determine the dependence relationships of the input program, (2) it then applies high-level program transformations and optimizations according to the dependences, and finally, (3) it generates parallelized code accordingly. In the whole process of a parallelizing compiler, dependence testing techniques serve as an underlying principle of a parallelizing compiler.

As both the dependence analysis and program transformations happen at compile time, a static parallelizing compiler always determines dependences by answering whether these dependences exist between two subscripted references to the same array in a loop nest. This kind of methods is referred to as data dependence testing techniques. In the full generality, dependence testing is an undecidable problem, since an array subscript can be arbitrary expressions of its enclosing loops' variables. As a result, a data dependence testing technique has to be conservative, meaning that it has to suppose the program is dependent when it cannot prove the absence of dependences. Earlier works mainly studied data dependence testing methods for linear subscripts. Since the mid-1990s, programmers began to focus on the dependence testing techniques for nonlinear subscripts. The former is referred to as linear data dependence testing techniques, while the latter is nonlinear testing techniques.

With the further development of computer architectures, static parallelizing compilers cannot meet programmers' requirement to parallel programs performance any longer. As a result, dynamic speculative parallelizing compilers and static–dynamic hybrid compilers were introduced. Since they can obtain the runtime information reflecting the relationship between two array references more exactly, dynamic compilers determine dependences by answering whether these dependences exist between two subscript values under representative inputs.

We can conclude that linear dependence testing algorithms have been developed better than the nonlinear ones. The reasons can be explained from two aspects. First, as the empirical study described in [2], linear subscripts consume more than 90% in dependence analysis system in PFC (Parallel Fortran Converter) [3], illustrating that solving a linear problem is more important than a nonlinear one, which is also supported by many existing literatures. Second, static parallelizing compilers have been developed for decades, but compiler developers pay more attention to speculative parallelizing compilers recently. So we argue that the static parallelizing compilation has encountered a bottleneck.

As a result, how to evaluate various existing data dependence testing techniques so that compiler developers can compare the relative power of these methods better, has become a challenging problem. Compiler designers would not only like to know their power and availability, but also desire to make their compilers disprove as many dependences as possible. In the early 1990s, some researchers studied and examined the impact of data dependence analysis in practice. Shen et al. [4] performed a preliminary empirical study on some numerical packages, including Linpack and Eispack,[1] comparing the relative power of some dependence testing techniques. Based on the above packages and the Perfect Benchmarks, Petersen and Padua [5] performed an experimental evaluation of a proposed sequence of dependence testing methods. Psarris and Kiriakopoulos [6] also showed the tradeoffs between the power and efficiency, and the time complexities of three representative techniques. All of these studies tried to give an empirical or experimental evaluation result, but none of these works shows a theoretical comparison on the power of data dependence testing techniques.

To enhance the ability of parallelizing compilers to determine dependences in programs, some later works combined a suite of testing methods to improve the power of dependence testing algorithms. Golf et al. [2] divided linear subscripts into zero index variable (ZIV), single index variable (SIV), multiple index variable (MIV), and proposed a sequence of dependence testing methods accordingly. They improved the power of their compiler to detect dependences by combining these testing methods. Maydan et al. [7] proposed a subscript pattern based dependence testing suite. If all these testing methods failed, Fourier–Motzkin elimination [8] is used as a time-consuming back up strategy. In fact, all these studies were trying to seek a complete set of dependence testing techniques, although none of them explained their purpose properly. However, they failed to construct such a complete set but only showed some experimental results instead.

To evaluate the power of existing testing techniques theoretically, as well as to find their upper bounds and complete sets, in this paper, we propose a formal system by restricting the first-order predicate logic system, to evaluate the relative power of different linear data dependence testing techniques. we not only show a theoretical evaluation of data dependence testing techniques, but also find some upper bounds and minimum complete sets of these dependence testing techniques. More specifically, the contributions of this work are as follows:

(1) We propose a formal system K-DT that is designed to evaluate the relative power of existing linear data dependence testing techniques by supplying necessary axioms into the first-order predicate logic formal system. The soundness, adequacy, and consistency of the K-DT system are proved, respectively, which lays the foundation for proving the theorems in K-DT.
(2) We evaluate all the linear data dependence testing techniques by proving their corresponding theorems in K-DT. The predicates of the theorems are domain-specific and the same with these methods' names.

[1] Linpack, Eispack and the Perfect Benchmarks that will be introduced later, are different sets of benchmarks targeting performance evaluation of parallel programs generated by parallelizing compilers.

(3) We show some upper bounds and minimum complete sets of linear data dependence testing techniques in different cases, and prove the completeness of these sets.

The paper is organized as follows. Section 2 introduces some basic definitions in data dependence testing area and illustrates our motivation. Section 3 presents the formal system K-DT, and proves its soundness, adequacy, and consistency, followed by some additional deductive rules. Section 4 shows the proofs of the theorems corresponding to the data dependence testing techniques. The upper bounds and minimum complete sets in different cases are presented in Sect. 5. Section 6 reviews related work, followed by the conclusion in Sect. 7.

2 Motivation

Dependence testing is the method used to determine whether dependences exist between two subscripted references to the same array in a loop nest [1]. It is a fundamental principle of a parallelizing compiler. In most cases, dependence testing problem can be illustrated by the code shown below. A dependence testing technique is used to determine whether the statement $\boldsymbol{S}_2$ depends on the statement $\boldsymbol{S}_1$ or the contrary.

```
for(i_1 = L_1; i_1 ≤ U_1; i_1 ++){
    for(i_2 = L_2; i_2 ≤ U_2; i_2 ++){
        ...
            for(i_n = L_n; i_n ≤ U_n; i_n ++){
    S_1         A[h_1(i_1, i_2, ..., i_n), ..., h_m(i_1, i_2, ..., i_n)] = ···
    S_2         ... = A[g_1(i_1, i_2, ..., i_n), ..., g_m(i_1, i_2, ..., i_n)]
            }
        ...
    }
}
```

To answer this question, a compiler must figure out whether these two statements access the same memory. Since $\boldsymbol{S}_1$ writes to array A while $\boldsymbol{S}_2$ reads the value from the same address, the problem has been transformed into determining whether the subscripts in these statements are equal to each other. In the absence of control dependences, there exist dependences between these two statements, if and only if the following system of dependence equations is satisfied:

$$\begin{cases} f_1(i_1,\ldots,i_n,j_1,\ldots,j_n) \equiv h_1(i_1,\ldots,i_n) - g_1(j_1,\ldots,j_n) = 0 \\ f_2(i_1,\ldots,i_n,j_1,\ldots,j_n) \equiv h_2(i_1,\ldots,i_n) - g_2(j_1,\ldots,j_n) = 0 \\ \qquad\qquad\qquad\qquad\vdots \\ f_m(i_1,\ldots,i_n,j_1,\ldots,j_n) \equiv h_m(i_1,\ldots,i_n) - g_m(j_1,\ldots,j_n) = 0 \end{cases} \tag{1}$$

Therefore, for a couple of different subscripted references, the essence of dependence testing is to prove or disprove the solutions to the equation system (1) in its feasible region, while the feasible region is defined by each pair of loop bounds

$$\boldsymbol{R} = \{L_k \leq i_k, j_k \leq U_k | 1 \leq k \leq n, i_k, j_k \in \mathbb{Z}\} \tag{2}$$

in which L_k and U_k can depend on $i_1, \ldots, i_{k-1}$.

If a testing technique can prove the absence of the solutions, then no dependences between these two statements exist. In fact, it is the most desired result of a dependence testing algorithm, since programs can be parallelized by compilers in this case. Otherwise, the dependence testing algorithm will try to characterize the possible dependences in some manner, usually as a minimal complete set of distance [9] and direction vectors [10,11].

A data dependence testing algorithm has to be conservative. That is to say, if it is not able to disprove the dependences, it must suppose there are dependences. In some cases, there is an additional restriction due to the direction vector

$$i_k D_k j_k (1 \leq k \leq n) \tag{3}$$

where D_k represents the kth element of the direction vector.

So a dependence testing technique is used to analyze whether the system of Eq. (1) has solutions with the restrictions (2) and (3) that form the feasible region, *i.e.*, the region of interest. Existing testing methods always determine whether the system of Eq. (1) has feasible solutions by relaxing the restriction (2), extending their feasible region. As a result, if a testing algorithm can prove the absence of feasible solutions in the extended region, then the original problem is definitely unsolvable in the region of interest. However, for parallelizing compiler designers, how to compare the power of data dependence testing methods, so that the compilers can prove or disprove more dependences, is a challenging problem. Existing studies achieved this goal by providing empirical studies or experimental evaluation results. To the best of our knowledge, no theoretical research on this issue has been published up to now. Therefore, how to compare their power theoretically and find a minimum complete set of existing dependence tests is a problem demanding prompt solutions.

As a matter of fact, the restriction (2) constructs the feasible region $\boldsymbol{R}$ of the system of Eq. (1), while the restriction (3) only defines the lexicographical order of the variables of the system of Eq. (1) in $\boldsymbol{R}$. As we described in last paragraph, existing dependence testing methods always determine whether the system of Eq. (1) has a feasible solution in an extended region ***R'***. The feasible region $\boldsymbol{R}$ transforms into the whole field of real numbers in the absence of the restriction (2); otherwise it is the whole field of integer numbers or bounded real number field when taking no account of the bound or integer constraint respectively. Here the bound is defined by the upper and lower bounds of each loop index. So we classify the region $\boldsymbol{R}$ into four categories: the unbounded real number field ***UR***, the unbounded integer number field ***UI***, the bounded real number field ***BR*** and the bounded integer number field ***BI***. Their relationship is shown in Fig. 1.

Since the restriction (2) represents a set of integer numbers bounded by the loop indexes bounds, the region of interest $\boldsymbol{R}$ is equal to ***BI***. Our purpose is to compare the power of different dependence testing techniques, and each testing algorithm corresponds to a kind of region shown in Fig. 1, so we can compare their relative power through the relationship among their corresponding regions in Fig. 1.

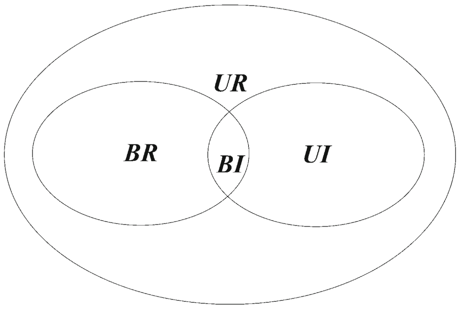

Fig. 1 The relationship between four feasible regions

At this point, for any two dependence tests D-test1 and D-test2, we are trying to answer such a question: Can D-test2 assure there is no dependence as long as D-test1 disproves the dependences? If we view D-test1 and D-test2 as two predicates in the first-order formal system $K_{\mathcal{L}}$ [12], then we only need to prove whether D-test1→D-test2 is a theorem in $K_{\mathcal{L}}$. Our intention is to construct such a formal system, and prove each theorem like D-test1→D-test2 in the system.

3 The K-DT system

A data dependence testing technique usually returns three kinds of results: *yes*, *no* or *maybe*. Accordingly, one can easily map these results to intuitionistic logic. If a data dependence testing technique returns *no*, saying there are no dependences, the truth value of its corresponding proposition is *true*. We define this way because the testing result is always expected to return *no*, so that the compiler can reorder the programs safely and more compile time optimizations can be applied. If it returns *yes*, saying there are dependences, the truth value of its corresponding proposition is *false*. If it returns *maybe*, saying it cannot determine whether there are dependences, the truth value of its corresponding proposition is *unknown*.

When parallelizing programs, safety always comes before performance. More specifically, when a parallelizing compiler translates serial programs into parallel codes, maintaining the semantics of generated codes' semantic is more important than boosting the speedup. Thus all data dependence testing techniques have to be conservative. In other words, when returning *maybe*, they have to assume conservatively the existence of dependences in case the compiler performs optimizations which will violate potential dependences. Therefore, when a testing algorithm cannot confirm whether there are dependences, we also treat it as it returns *yes*. Hence the truth value of its corresponding proposition is also *false*. We transform the intuitionistic logic problem into classical logic area by this means.

We transform the problem into classical logic based on the following considerations. First, it is reliable when a dependence testing algorithm returns *no*. Some methods do not make specific distinction between the remaining two cases. If the problem is viewed as an intuitionistic logical issue, it will be difficult to formalize the remaining truth values, *i.e.*, *false* and *unknown*. Second, if we solve the problem using classical logic, the truth value of a proposition will be either *true* or *false*, allowing us to use the reduction to absurdity, simplifying the deduction process of theorems. Finally, a classical logic based predicate calculus is more concise than that based on an intuitionistic one.

3.1 Constructing the K-DT system

We attempt to solve the problem based on the first-order formal system $K_{\mathcal{L}}$, because its soundness, adequacy, and consistency have been proved. However, there are only six basic axioms in $K_{\mathcal{L}}$, none of which has a definite relationship with the theorems we would like to prove, making $K_{\mathcal{L}}$ not suitable for our purpose. It is necessary to supply some additional axioms to form a new formal system. Since it is a $K_{\mathcal{L}}$ based system and designed to evaluate data dependence testing techniques, we name the system as K-DT.

Before adding necessary axioms to $K_{\mathcal{L}}$, we need to append some domain-specific predicate letters, including GCD, $Banerjee$, I, $EGCD$, λ, $Power$, $Omega$, $Delta$, $suff\_test$, nec_test, $Solvable$, $Include$, F_M, $complete\_test$, $single\_test$, DT_test, to this system. $Suff\_test$, representing a kind of dependence testing algorithm as well as a sufficient condition to determine the system of Eq. (1), and nec_test, representing a kind of dependence testing technique as well as a necessary condition, are unitary predicates. $Solvable$ and $Include$ are both binary predicates. One states that the tested system of equations is solvable in some feasible region, while the other tells that a feasible region includes another one. $Complete\_test$, $single\_test$, and DT_test are unitary predicates, which will be explained in the following context, while the remaining represents different kinds of dependence testing techniques.

Suppose $\mathcal{L}_{\mathcal{DT}}$ refer to a first-order language, so we can define the formal deductive system K-DT by the axioms followed.

$(D1)$	$A \rightarrow (B \rightarrow A)$
$(D2)$	$(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
$(D3)$	$(\sim B \rightarrow \sim A) \rightarrow (A \rightarrow B)$
$(D4)$	$(\forall e_i)A \rightarrow A$ (if e_i does not occur free in A)
$(D5)$	$(\forall e_i)A \rightarrow A(t)$ (if $A(e_i)$ is a $wf.$ of $\mathcal{L}_{\mathcal{DT}}$ and t is a term in $\mathcal{L}_{\mathcal{DT}}$ which is free for e in $A(e_i)$)
$(D6)$	$(\forall e_i)(A \rightarrow B) \rightarrow (A \rightarrow (\forall e_i)B)$ (if A contains no free occurrence of the variable e_i)
$(D7)$	$Suff\_test(e) \rightarrow \sim Solvable(e, r)$
$(D8)$	$\sim Solvable(e, r) \rightarrow Nec\_test(e)$
$(D9)$	$Solvable(u(e_1, e_2), r) \rightarrow Solvable(e_1, r)$
$(D10)$	$Include(r_2, r_1) \rightarrow (Solvable(e, r_1) \rightarrow Solvable(e, r_2))$

Axioms (D1)–(D6) correspond to Axioms (K1)–(K6) of $K_{\mathcal{L}}$. Axiom (D7) represents that the system of equations is unsolvable in a feasible region r provided a sufficient condition dependence testing algorithm returns no. Axiom (D8) says that if the system of equations is unsolvable in a feasible region r, a necessary condition dependence test will certainly return a result saying there is no dependence. Axiom (D9) can be understood as that if a system of equations composed of e_1 and e_2 is solvable on a feasible region r, e_1 is also solvable on r. Axiom (D10) shows that if a feasible region r_2 contains r_1 and the system of equations is solvable on r_1, it will also be solvable on r_2.

Notice that K-DT also has two deductive rules known as modus ponens and generalization, which is the same with K$_{\mathcal{L}}$.

3.2 Properties of the K-DT system

Before performing the deductive process of its theorems, we need to prove some properties of the K-DT system. The following problems should be taken into account: (1) whether K-DT is sound; (2) whether K-DT is adequate; (3) whether K-DT is consistent. Only are these problems proved true, the theorem proofs based on K-DT can be convincing. We use $\vdash A$ represents that A is a theorem in K-DT, and $\models A$ represents A is a logically valid well-formed formula (wf.) of $\mathcal{L}_{\mathcal{DT}}$.

Theorem 1 (The Soundness Theorem for K-DT) *For any wf. A of $\mathcal{L}_{\mathcal{DT}}$, if $\vdash A$ then A is logically valid.*

Proof By induction on the number s of steps in a proof of A. If $s = 1$, meaning A has a one-step proof, then A is an axiom of K-DT. Each axiom of K-DT is logically valid. Suppose that $s = n (n > 1)$ and all theorems of K-DT with proofs in fewer than n steps are logically valid. A appears in a proof, so only the following two cases may happen. (1) A is an axiom; (2) A follows from previous wfs. For case (1), A is logically valid, as above. For case (2), A is also logically valid as shown in [12]. This completes our proof by induction.

Theorem 2 (The Adequacy Theorem for K-DT) *For any wf. A of $\mathcal{L}_{\mathcal{DT}}$, if $\models A$ then A is a theorem of K-DT.*

Proof Only the predicate letters of K-DT are different from those of K$_{\mathcal{L}}$. All the wfs. in K-DT are defined in the same way as those in K$_{\mathcal{L}}$. As K$_{\mathcal{L}}$ is adequate, all the wfs. without domain-specific predicates are theorems of K-DT. For the wfs. with domain-specific predicates, their proof is similar to that of the Adequacy Theorem for K$_{\mathcal{L}}$ in [12]. So it is true in each case that if $\models A$ then A is a theorem of K-DT.

Theorem 3 (The Consistency Theorem for K-DT) *For no wf. A are both A and $\sim A$ theorems of K-DT.*

Proof Suppose that $\vdash A$ and $\vdash \sim A$ for some wf. A of $\mathcal{L}_{\mathcal{DT}}$. So both A and $\sim A$ are logically valid wfs. by Theorem 1, *i.e.*, A and $\sim A$ are both tautologies, which violates the Excluded Middle of the classical logic. So K-DT must be consistent.

Since the soundness and adequacy have been proved, we can perform the theorem proofs in K-DT. However, we find that the proofs would become very redundant with only the modus ponens and generalization rules, we therefore propose the following rules to simplify the deductive process.

Theorem 4 (The Deduction Theorem for K-DT) *Let A and B be wfs. of $\mathcal{L}_{\mathcal{DT}}$ and let $\mathcal{T}$ be a set (possibly empty) of wfs. of $\mathcal{L}_{\mathcal{DT}}$. If $\mathcal{T} \cup \{\mathcal{A}\} \vdash \mathcal{B}$, and the deduction contains no application of Generalization involving a variable which occurs free in A, then $\mathcal{T} \vdash (\mathcal{A} \rightarrow \mathcal{B})$.*

Proof As shown in Theorem 2, only the predicate letters of K-DT are different from those of $K_{\mathcal{L}}$. All the wfs. in K-DT are defined same as those in $K_{\mathcal{L}}$. So the Deduction Theorem for K-DT is same with that for $K_{\mathcal{L}}$, while the latter has been proved in [12].

Theorem 5 (The Hypothetical Syllogism Theorem for K-DT) *Let A, B and C be wfs. of $\mathcal{L}_{\mathcal{DT}}$. If* $\vdash (A \to B)$ *and* $\vdash (B \to C)$ *then* $\vdash (A \to C)$.

Proof 1. $A \to B \quad (Assumption)$
2. $B \to C \quad (Assumption)$
3. $(B \to C) \to (A \to (B \to C)) \quad (D1)$
4. $A \to (B \to C) \quad (2, 3, MP)$
5. $(A \to (B \to C)) \to ((A \to B) \to (A \to C)) \quad (D2)$
6. $(A \to B) \to (A \to C) \quad (4, 5, MP)$
7. $A \to C \quad (1, 6, MP)$

4 Theorem proofs based on the K-DT system

When determining dependences of multi-dimensional subscripted references, a subscript position is said to be separable if its indexes do not occur in the other subscripts [13,14]. If two subscripted references contain the same index, they are coupled [15]. What kind of dependence testing techniques a compiler uses is decided by whether an index of a subscript occurs in others.

Mathematically speaking, if the subscripts are separable, the system of Eq. (1) is unsolvable if and only if at least one of its equations is unsolvable. If the subscripts are coupled, the unsolvability of the system of Eq. (1) is only a necessary condition of the unsolvability of its each equation. Considering this matter, we divide existing linear dependence testing techniques into those for separated and for coupled subscripts.

4.1 Theorems of dependence tests for separated subscripts

For separable subscripts, the system of Eq. (1) is unsolvable if and only if at least one of its equations is unsolvable, as mentioned above. A dependence testing method for separated subscripts is allowed to seek solutions for a single equation rather than the whole equation system. Any equation of the system of equation system (1) can be written as

$$f(i_1, \ldots, i_n, j_1, \ldots, j_n) \equiv h(i_1, \ldots, i_n) - g(j_1, \ldots, j_n) = 0. \tag{4}$$

Without loss of generality, we can assume h and g have the following forms

$$h(i_1, \ldots, i_n) = a_1 i_1 + a_2 i_2 + \cdots + a_n i_n + a_0 \tag{5}$$
$$g(j_1, \ldots, j_n) = b_1 j_1 + b_2 j_2 + \cdots + b_n j_n + b_0. \tag{6}$$

Hence

$$f(i_1, \ldots, i_n, j_1, \ldots, j_n) \equiv a_0 - b_0 + a_1 i_1 - b_1 j_1 + \cdots + a_n i_n - b_n j_n = 0. \tag{7}$$

We first analyze the GCD (greatest common divisor) test [16]. The GCD test uses a fundamental theorem about the linear diophantine equations [17] to determine whether an equation has dependences. Rearranging Eq. (7) yields the following

$$a_1 i_1 - b_1 j_1 + \cdots + a_n i_n - b_n j_n = b_0 - a_0. \tag{8}$$

The fundamental theorem is the following.

Lemma 1 (GCD test) *Equation (8) has a solution if and only if* $gcd(a_1, \ldots, a_n, b_1, \ldots, b_n)$ *divides* $b_0 - a_0$.

That is to say, if the greatest common divisor of all the coefficients of variables does not divide the right-hand term of Eq. (8), no solution exists anywhere. Also, there are no dependences. Otherwise it has solutions somewhere. The GCD test is a necessary and sufficient condition, and its feasible region is ***UI***.

Banerjee test [18] is a dependence testing algorithm with feasible region ***BR***. Its main idea is to calculate the extreme values of the left-hand terms of Eq. (8) and determine whether the right-hand term exists in the scope bounded by these extreme values. When determining a dependence with direction vector $D = (D_1, \ldots, D_n)$, Banerjee test works based on the following.

Lemma 2 (Banerjee Inequality) *There exists a real solution to Eq. (8) for direction vector* $D = (D_1, \ldots, D_n)$ *if and only if the following inequality is satisfied on both sides:*

$$\sum_{k=1}^{n} H_k^- D_k \le b_0 - a_0 \le \sum_{k=1}^{n} H_k^+ D_k$$

where H_k^+ and H_k^- represent the maximum and minimum values of left-hand terms figured out according to D_k. A D_k can be '>', '<' '=', or '*', with '*' representing that this component is unknown. Definitions on each can be found in [1,18]. The Banerjee test is a necessary and sufficient condition and its feasible region is ***BR***.

Neither the feasible regions of GCD test nor Banerjee test is the region of interest ***BI***. Considering this problem, Kong et al. [19] proposed an improved dependence testing algorithm called I test. The I test abstracts Eq. (8) as an interval equation

$$a_1 I_1 + a_2 I_2 + \cdots + a_s I_s = [L, U] \tag{9}$$

and supposes there are $s(s = 2n)$ variables in the equation, with each coefficient is $a_k (1 \le k \le s)$. L and U are natural numbers. As a result, its determining process can be described as follows. If $|a_s| > 1$, reduce Eq. (9) with Lemma 3. Next, repeat Lemma 4 to perform equation elimination until only one variable is left in the left-hand expression. At this point, determine whether dependences exist according to the bounds of loop indexes and the GCD test. In the meanwhile of applying Lemma 4, output a result and return whether the Banerjee test can prove there are no dependences.

Lemma 3 *Let* $d = gcd(a_1, a_2, \ldots, a_s)$*, Eq. (9) is* $(M_1, N_1; M_2, N_2; \cdots; M_s, N_s)$*-integer solvable iff the interval equation* $(a_1/d)I_1 + (a_2/d)I_2 + \cdots + (a_s/d)I_s =$

Table 1 The properties and feasible regions of dependence tests for separated subscripts

Dependence tests	Property	Feasible region	Axioms
GCD test	Necessary and sufficient condition	***UI***	$GCD(e) \leftrightarrow \sim Solvable(e, UI)$ (7)(8)
Banerjee test	Necessary and sufficient condition	***BR***	$Banerjee(e) \leftrightarrow \sim Solvable(e, BR)$ (7)(8)
I test	Necessary and sufficient condition	***BI***	$I(e) \leftrightarrow \sim Solvable(e, BI)$ (7)(8)

$[\lceil L/d \rceil, \lfloor U/d \rfloor]$ is $(M_1, N_1; M_2, N_2; \cdots; M_s, N_s)$-integer solvable, where each of M_k and N_k be either an integer or the distinguished symbol "" and $M_k \leq N_k (1 \leq k \leq s)$ if they are both integers.*

Lemma 4 *If $|a_s| \leq U - L + 1$, then Eq. (9) is $(M_1, N_1; M_2, N_2; \cdots; M_s, N_s)$-integer solvable iff the interval equation $a_1 I_1 + a_2 I_2 + \cdots + a_{s-1} I_{s-1} = [L - a_s^+ N_s + a_s^- M_s, U - a_s^+ M_s + a_s^- N_s]$ is $(M_1, N_1; M_2, N_2; \cdots; M_s, N_s)$-integer solvable, where a_s^+ and a_s^- represent the positive and negative parts respectively.*

Lemma 4 reduces the number of variables of Eq. (9) and defines new bounds for the generated interval equations with the positive and negative parts. The I test is also a necessary and sufficient condition, with the feasible region ***BI***.

To sum up, for the GCD, Banerjee and I tests, their properties and feasible regions are shown in Table 1. Now we can prove the theorems whose predicates are the same with these testing techniques names in K-DT. As numerous theorems in $K_{\mathcal{L}}$ have been proved in [12] and they are also theorems in K-DT, we label these theorems with *TK* representing a Theorem in $K_{\mathcal{L}}$ in following proofs.

Theorem 6 $(\forall e)(GCD(e) \rightarrow I(e))$.

Proof 1. $Include(UI, BI)$ $\quad (Tautology)$
2. $Include(UI, BI) \rightarrow (Solvable(e, BI) \rightarrow Solvable(e, UI))$ $\quad (D10)$
3. $Solvable(e, BI) \rightarrow Solvable(e, UI)$ $\quad (1, 2, MP)$
4. $(Solvable(e, BI) \rightarrow Solvable(e, UI)) \rightarrow (\sim Solvable(e, UI) \rightarrow \sim Solvable(e, BI))$ $\quad (TK)$
5. $\sim Solvable(e, UI) \rightarrow \sim Solvable(e, BI))$ $\quad (3, 4, MP)$
6. $GCD(e) \rightarrow \sim Solvable(e, UI)$ $\quad (D7)$[2]
7. $GCD(e) \rightarrow \sim Solvable(e, BI)$ $\quad (6, 5, HS)$
8. $\sim Solvable(e, BI) \rightarrow I(e)$ $\quad (D8)$
9. $(GCD(e) \rightarrow I(e))$ $\quad (7, 8, HS)$
10. $(\forall e)(GCD(e) \rightarrow I(e))$ $\quad (Generalization)$

Theorem 7 $(\forall e)(Banerjee(e) \rightarrow I(e))$.

[2] It seems like $GCD(e) \rightarrow \sim Solvable(e, UI)$ mismatches Axiom (D7). However, we conclude in Table 1 that $GCD(e) \leftrightarrow \sim Solvable(e, UI)$, so we write them in this form to make the deductions concise. So are the situations in Theorems 7, 9 and 10.

Proof 1. $Include(BR, BI)$ $(Tautology)$
2. $Include(BR, BI) \rightarrow (Solvable(e, BI) \rightarrow Solvable(e, BR))$ $(D10)$
3. $Solvable(e, BI) \rightarrow Solvable(e, BR)$ $(1, 2, MP)$
4. $(Solvable(e, BI) \rightarrow Solvable(e, BR)) \rightarrow (\sim Solvable(e, BR) \rightarrow\sim Solvable(e, BI))$ (TK)
5. $\sim Solvable(e, BR) \rightarrow\sim Solvable(e, BI))$ $(3, 4, MP)$
6. $Banerjee(e) \rightarrow\sim Solvable(e, BR)$ $(D7)$
7. $Banerjee(e) \rightarrow\sim Solvable(e, BI)$ $(6, 5, HS)$
8. $\sim Solvable(e, BI) \rightarrow I(e)$ $(D8)$
9. $(Banerjee(e) \rightarrow I(e))$ $(7, 8, HS)$
10. $(\forall e)(Banerjee(e) \rightarrow I(e))$ $(Generalization)$

We can prove neither $(\forall e)(GCD(e) \rightarrow Banerjee(e))$ nor $(\forall e)(Banerjee(e) \rightarrow GCd(e))$, validating neither testing technique is more accurate than the other in theory, as explained by Kong et al. [19]. However, we proved formally that both the GCD and Banerjee tests can be reduced to the I test. In other words, for all system of equations, as long as the GCD test or Banerjee test can disprove dependences, the I test is certainly able to assure the absence of dependences.

4.2 Theorems of dependence tests for coupled subscripts

As described above, for coupled subscripts, the unsolvability of the system of Eq. (1) is only a necessary condition of the unsolvability of its each equation. In other words, if the m equations are solvable individually, the system of Eq. (1) is certainly solvable, which is formalized as the axiom $(D9)$. If a parallelizing compiler uses a testing algorithm for coupled subscripts (subscript-by-subscript checking), there may be no dependences even if the result is *yes*. Some newer introduced testing algorithms are usually proposed to handle the coupled subscripts.

Earlier dependence testing techniques for coupled subscripts are designed based on the GCD test and Banerjee inequality. Researchers extended these methods so as to give a more accurate result for coupled cases. Since the GCD test can only analyze separated subscripts, Knuth [20] extended its algorithm to find whether a set of linear equations with integer coefficients has any integer solutions. Banerjee provided a way to enumerate all those solutions [21]. We call it EGCD test as it derived from the GCD algorithm.

Suppose $\boldsymbol{A}$ is a $2n \times m$ coefficient matrix filled with the coefficients of the system of Eq. (1). If there are linearly dependent equations in the system of Eq. (1), eliminate the redundant ones with the GCD algorithm. The goal of the EGCD test is to seek whether there is an integer solution $\boldsymbol{x}$ for the system of Eq. (1), by checking whether a set of equations

$$\mathbf{xA} = \mathbf{c}(\mathbf{c} \text{ is a vector with } \mathbf{m} \text{ elements}) \tag{10}$$

has an integer solution.

The checking process is following. Initialize a $2n \times m$ matrix $\boldsymbol{D}$ with the elements of $\boldsymbol{A}$, and a $2n \times 2n$ matrix $\boldsymbol{U}$ with the identity matrix. Store these two matrices in one $2n \times (2n + m)$ matrix $(\boldsymbol{U} \mid \boldsymbol{D})$. Reduce the matrix $\boldsymbol{D}$ to upper triangular form, as shown below, by a series of elementary integer row operations. At this point, in the kth ($1 \leq k < 2n$) column of matrix $\boldsymbol{D}$, all the elements in rows k+1 through $2n$ are zeros. Applying such elementary operations until the identity matrix $\boldsymbol{U}$ satisfies $\boldsymbol{U}\,\boldsymbol{A}$= $\boldsymbol{D}$. Now the matrix $\boldsymbol{U}$ is transformed into a unimodular one (det($\boldsymbol{U}$) = ± 1).

Finally, if there is an integer solution $\boldsymbol{t}$ for $\boldsymbol{tD} = \boldsymbol{c}$, then $\boldsymbol{x} = \boldsymbol{tU}$ is an integer solution to the set of Eq. (10). As a result, the EGCD test is a necessary condition and its feasible region is $\boldsymbol{UI}$.

The λ test [15,22] is the first work to consider all the subscripts of coupled subscripts together. It is a multi-dimensional version of Banerjee inequalities. For the system of Eq. (1), the λ test supposes there are $s = 2n$ variables and it will transform into

$$\begin{cases} a_1^{(1)} v^{(1)} + a_1^{(2)} v^{(2)} + \cdots + a_1^{(s)} v^{(s)} + c_1 = 0 \\ a_2^{(1)} v^{(1)} + a_2^{(2)} v^{(2)} + \cdots + a_2^{(s)} v^{(s)} + c_2 = 0 \\ \vdots \\ a_m^{(1)} v^{(1)} + a_m^{(2)} v^{(2)} + \cdots + a_m^{(s)} v^{(s)} + c_m = 0 \end{cases} \tag{11}$$

Equation (11) is then linearly combined as

$$< \sum_{i=1}^{m} \lambda_i \mathbf{a_i}, \mathbf{v} > + \sum_{i=1}^{m} c_i = 0 \tag{12}$$

where $\mathbf{a_i} = (a_i^{(1)}, a_i^{(2)}, \ldots, a_i^{(s)})$, $\mathbf{v} = (v^{(1)}, v^{(2)}, \ldots, v^{(s)})$.

After such a linear combination, the λ test will eliminate one or multiple newly introduced variables λ_i with the canonical solutions defined in [15,22]. If all variables λ_i are eliminated, it then applies the Banerjee test to solve Eq. (12). However, when variables λ_i cannot be eliminated, which usually happens in cases when $m > 2$, the λ test cannot give an accurate result. Therefore, when all the introduced variables λ_i can be eliminated, the test is a necessary and sufficient condition with feasible region $\boldsymbol{BR}$. Otherwise, it will not work.

The power test [23] is a technique combining the EGCD test and Fourier–Motzkin elimination. It first tests the subscripts with the EGCD test. If the EGCD test cannot prove the system of Eq. (1) has no solution, the power test attempts to give a feasible solution. However, whether this feasible solution is in the region of interest $\boldsymbol{R}$ is not decidable. Hence the power test determines whether there are dependences according to the restrictions (2) and (3).

In the EGCD test algorithm, as the matrix $\boldsymbol{D}$ will change into upper triangular form after a series of elementary operations, its first m rows hold nonzero elements. The EGCD test is able to figure out first m elements of the integer solution $\boldsymbol{t}$ and thus $t_{m+1}, t_{m+2}, \ldots, t_{2n}$ are free elements. Next, the power test tries to compute the dependence distance. If only the coefficients of t_1 through t_m are nonzero, the dependence distance is fixed and the power test is a necessary and sufficient condition with feasible region $\boldsymbol{BI}$. If there are nonzero coefficients for any $t_v (v > m)$, then the dependence distance is not constant. At this point, the power test will apply the Fourier–Motzkin elimination, implying that the system of inequalities after eliminating a variable has solutions if and only if the original system has solutions. In this case, the power test is a sufficient condition, and its feasible region is ***BR***. It cannot be a necessary condition, since the power test may not report there is no dependence when there is no solution for the system of Eq. (1).

The omega test [24] is also a Fourier–Motzkin elimination-based dependence testing technique. It considers the system of Eq. (1) and restrictions (2) and (3) together, and transforms them all into a set of inequalities by eliminating redundant elements. As a consequence, this set of inequalities is treated as a polyhedron object. Intuitively, it finds the $n-1$-dimensional shadow cast by an n-dimensional object and calculates the "real shadow" and "dark shadow"[3] of the object respectively. When the "real shadow" and "dark shadow" are identical, there are integer solutions to the set of inequalities if and only if there are integer solutions to the shadow. Otherwise: (a) There is no integer solution to the set of inequalities if there is no integer solution to the "real shadow"; (b) there are integer solutions to the set of inequalities if there are integer solutions to the "dark shadow"; (c) otherwise, it considers a set of planes parallel to a lower bound and close to a lower bound and analyzes the problem by some expensive and complicated steps. In practice, the last case is rarely used, and hence we say the omega test is only a sufficient condition but not necessary. Although it is similar to the power test to a certain extent, its feasible region is always ***BI***.

Since most subscripts found in practice are SIV (Single Index Variable) and their tests are usually simple and accurate, the main idea behind delta test [5] is to propagate the constraints produced by the SIV subscripts to other subscripts in the same group without losing accuracy. In most cases, such propagation will simplify the testing of other subscripts and produce a precise set of direction vectors. The main process delta test can be described as follows. The delta test first tests SIV parts in the coupled subscripts. If they are independent, the test will return *no*. Otherwise, it converts the information gleaned by these SIV subscripts into constraints and propagates it to all possible MIV (Multiple Index Variable) subscripts. Repeat this phase until no constraint can be found. Then propagate all the results to coupled RDIV (restricted double-index variable, which has the form $< a_1 i_1 + c_1, a_2 i_2 + c_2 >$) subscripts. Test all the remaining MIV subscripts and intersect the results with current constraints. If the intersection set is null, return there is no dependence. Otherwise, return the produced information about the dependences.

As a matter of fact, the delta test may be viewed as a restricted form of the λ test that trades generality for greater efficiency and precision [1]. When applying the delta test in practice, one can easily find out that it is an application of the λ test on two-dimensional cases. Therefore, the delta test is a necessary and sufficient condition, with the feasible region being ***BI***.

As a result of the above discussion, we can summarize the properties and feasible regions of dependence testing techniques for coupled subscripts, which are shown in Table 2. At present we can prove the theorems in K-DT corresponding to these tests. First, as the delta test is a restricted application of the λ test, and the power test algorithm always invokes the EGCD test, we have the following

[3] A real shadow is the whole shadow cast by an object while a dark shadow is clearly dark below any part of the object that is at least one unit thick. Refer to [24] for more details.

Table 2 The properties and feasible regions of dependence tests for coupled subscripts

Dependence tests	Property	Feasible region	Axioms
EGCD test	Necessary and sufficient condition	***UI***	$EGCD(e) \leftrightarrow \sim Solvable(e, UI)$ (7)(8)
λ test	Necessary and sufficient condition	***BR***	$\lambda(e) \leftrightarrow \sim Solvable(e, BR)$ (7)(8)
Power test	Necessary and sufficient condition (when not invoking FME method)	***BI***	$Power(e) \leftrightarrow \sim Solvable(e, BI)$ (7)(8)
Power test	Sufficient condition (when invoking FME method)	***BR***	$Power(e) \rightarrow \sim Solvable(e, BR)$ (7)
Omega test	Sufficient condition	***BI***	$Omega(e) \rightarrow \sim Solvable(e, BI)$ (7)
Delta test	Necessary and sufficient condition	***BI***	$Delta(e) \leftrightarrow \sim Solvable(e, BI)$ (7)(8)

Proposition 1 $(\forall e)(Delta(e) \rightarrow \lambda(e))$.

Proposition 2 $(\forall e)(EGCD(e) \rightarrow Power(e))$.

The number of variables of the system of Eq. (1) is $2n$, and it has m equations. The power test is a necessary and sufficient condition with feasible region ***BI*** when Fourier–Motzkin elimination is not invoked. As a consequence, we have

Theorem 8 *When the power test does not invoke Fourier–Motzkin elimination,* $(\forall e)(suff\_test(e) \rightarrow Power(e))$.

Proof 1. $Include(r, BI)$ $(Tautology)$
2. $Include(r, BI) \rightarrow (Solvable(e, BI) \rightarrow Solvable(e, r))(D10)$
3. $Solvable(e, BI) \rightarrow Solvable(e, r)$ $(1, 2, MP)$
4. $(Solvable(e, BI) \rightarrow Solvable(e, r)) \rightarrow (\sim Solvable(e, r) \rightarrow \sim Solvable(e, BI))$ (TK)
5. $\sim Solvable(e, r) \rightarrow \sim Solvable(e, BI)$ $(3, 4, MP)$
6. $Suff\_test(e) \rightarrow \sim Solvable(e, r)$ $(D7)$
7. $Suff\_test(e) \rightarrow \sim Solvable(e, BI)$ $(5, 6, HS)$
8. $\sim Solvable(e, BI) \rightarrow Power(e)$ $(D8)$
9. $Suff\_test(e) \rightarrow Power(e)$ $(7, 8, HS)$
10. $(\forall e)(Suff\_test(e) \rightarrow Power(e))$ $(Generalization)$

Theorem 9 *When the power test invokes Fourier–Motzkin elimination,* $(\forall e)(Power(e) \rightarrow Omega(e))$.

Proof Before showing the proof, we should emphasize that the power test invokes Fourier–Motzkin elimination, while this method implies that, the system of inequalities after eliminating a variable has solutions if and only if the original system has solutions. Hence the following

$$\sim Solvable(e, BR) \leftrightarrow \sim Solvable(e, BR')$$

is maintained by the power test. In the above, ***BR'*** represents the region after elimination. As for the omega test, case (a) can be formalized as

$$\sim Solvable(e, BR') \leftrightarrow Omega(e))$$

Hence we have the following proof.

1. $Power(e) \rightarrow \sim Solvable(e, BR) \quad (D7)$
2. $\sim Solvable(e, BR) \rightarrow \sim Solvable(e, BR') \quad (Tautology)$
3. $Power(e) \rightarrow \sim Solvable(e, BR') \quad (1, 2, HS)$
4. $\sim Solvable(e, BR') \rightarrow Omega(e) \quad (Tautology)$
5. $(Power(e) \rightarrow Omega(e)) \quad (3, 4, HS)$
6. $(\forall e)(Power(e) \rightarrow Omega(e)) \quad (Generalization)$

It is necessary to have a discussion here. The omega test applies Fourier–Motzkin elimination as well, but the feasible region is always ***BI*** but not ***BR***. As a result, the tautology 2) in above proof is not logically valid for the omega test.

From previous two theorems we can find that, when the power test does not invoke Fourier–Motzkin elimination, for any dependence testing method discussed in this study, the power test will assure the absence of dependences provided any dependence testing algorithm returns independence. When the power test invokes Fourier–Motzkin elimination, for the dependence testing methods discussed in this study, the EGCD test and power test can be reduced to the omega test. Considering the delta test can be reduced to the λ test, we only talk about the relationship between the omega test and the λ test.

Theorem 10 $(\forall e)(\lambda(e) \rightarrow Omega(e))$.

Proof For the omega test, if we consider without the integer constraint, we will obtain.

Proposition 3 $F\_M(e) \leftrightarrow \sim Solvable(e, BR)$.

Here the $F\_M(e)$ represents that there is no real solution to e by applying Fourier–Motzkin elimination. Now consider the integer constraint again. The omega test discusses the integer solution based on the matter that the system of Eq. (1) has a real solution. Hence

Proposition 4 $F\_M(e) \rightarrow Omega(e)$.

Hence we have the following proof.

1. $\lambda(e) \rightarrow \sim Solvable(e, BR) \quad (D7)$
2. $\sim Solvable(e, BR) \rightarrow F\_M(e)$ (Proposition 3)
3. $\lambda(e) \rightarrow F\_M(e) \quad (1, 2, HS)$
4. $F\_M(e) \rightarrow Omega(e)$ (Proposition 4)
5. $(\lambda(e) \rightarrow Omega(e)) \quad (3, 4, HS)$
6. $(\forall e)(\lambda(e) \rightarrow Omega(e)) \quad (Generalization)$

Therefore, when the power test invokes Fourier–Motzkin elimination, the omega test will assure there are no dependences provided any dependence testing algorithm for coupled subscripts returns an independent result.

5 Upper bounds and minimum complete sets

When proving the above-mentioned theorems, we think about whether there is a kind of dependence testing algorithm that can imply all the remaining ones. More specifically, we say that a dependence testing technique D-test2 is more powerful than D-test1 when the theorem D-test1→D-test2 can be proved. A dependence testing technique is supposed to be a upper bound if any dependence testing techniques can be reduced to it. A set of testing techniques is defined as the minimum complete set if they can be most powerful when working together. Obviously, none of the dependence testing methods discussed in this paper satisfies the requirement of the upper bound. However, can we give an equivalent condition of such dependence testing algorithms or are there some upper bounds in different cases? Beyond that, what are the minimum complete sets of these dependence testing techniques in different cases?

5.1 Upper bounds of existing linear dependence tests

Evidently, if there is such a dependence testing technique, it must be for coupled subscripts. The reason has been explained in previous section. Hence we say

Theorem 11 *If there exists such a dependence test* $complete\_test$ *that* $complete\_test(e) \leftrightarrow \sim Solvable(e, BI)$ *is true in each case, then* $(\forall e)(suff\_test(e) \rightarrow complete\_test(e))$.

Proof As the proof of Theorem 8, substituting the $Power$ with $complete\_test$ everywhere will obtain the proof of this theorem.

Although none of the mentioned dependence testing algorithms is able to satisfy the hypothesis of Theorem 11, we can get the following conclusion from Theorem 8. In the case that Fourier–Motzkin elimination is not invoked, the power test is an upper bound of all the dependence testing methods.

In a similar way, for all the dependence testing algorithms for separated subscripts discussed in this study, the following is always preserved.

Theorem 12 *If* $single\_test$ *that is a dependence test for separated subscripts, then* $(\forall e)(single\_test(e) \rightarrow I(e))$.

Proof As the proof of Theorem 8, substituting the $Power$ with $single\_test$ everywhere will obtain the proof of this theorem.

From Theorems 6, 7 and 11, we can conclude that the I test is the upper bound of the discussed dependence tests for separated subscripts.

5.2 Minimum complete sets of existing dependence tests

Since the power test is the upper bound of the discussed dependence tests in the case that Fourier–Motzkin elimination is not invoked, the minimum complete set in this case is {power test}. The proof can be acquired from Theorem 8 and 10. However, it is

Table 3 The minimum complete sets in different cases

Cases	Minimum complete sets
$2n \leq m$	{power test}
$2n > m$ without invoking FME method in power test	{power test}
$2n > m$ with invoking FME method in power test	{I test, omega test}

not so easy to figure out whether the power test invokes Fourier–Motzkin elimination. By further analyzing the process of the power test, we find that when $2n \leq m$, saying the number of variables of the system of Eq. (1) is smaller than its equation number, the power test does not invoke it for sure.

Otherwise, if the power test invokes Fourier–Motzkin elimination, we know, from Theorem 9, that all the dependence tests for coupled subscripts can be reduced to the omega test, while each dependence testing algorithm for separated subscripts is reduced to the I test. Neither of the omega test and I test can be reduced to each other, so the minimum complete set in this case is {I test, omega test}. We should prove that

Theorem 13 *For any* $DT\_test \in Predicate\_DT$*, the following* $(\forall e)(DT\_test(e) \rightarrow (\sim I(e) \rightarrow Omega(e)))$ *is always preserved.*

Proof We should consider two cases. First, if DT_test is a dependence test for separated subscripts, then

1. $(\forall e)(DT\_test(e) \rightarrow I(e))$ (Theorem 12)
2. $(\forall e)(DT\_test(e) \rightarrow I(e)) \rightarrow (DT\_Test(e) \rightarrow I(e))$ $(D4)$
3. $(DT\_test(e) \rightarrow I(e))$ $(1, 2, MP)$
4. $I(e) \rightarrow (\sim I(e) \rightarrow Omega(e))$ (TK)
5. $DT\_test(e) \rightarrow (\sim I(e) \rightarrow Omega(e))$ $(3, 4, HS)$
6. $(\forall e)DT\_test(e) \rightarrow (\sim I(e) \rightarrow Omega(e))$ $(Generalization)$

Second, if DT_test is a dependence test for coupled subscripts, then

1. $(\forall e)(DT\_test(e) \rightarrow Omega(e))$ (Theorems 9, 10)
2. $(\forall e)(DT\_test(e) \rightarrow Omega(e)) \rightarrow (DT\_test(e) \rightarrow Omega(e))$ $(D4$
3. $(DT\_test(e) \rightarrow Omega(e))$ $(1, 2, MP)$
4. $Omega(e) \rightarrow (\sim I(e) \rightarrow Omega(e))$ $(D1)$
5. $DT\_test(e) \rightarrow (\sim I(e) \rightarrow Omega(e))$ $(3, 4, HS)$
6. $(\forall e)DT\_test(e) \rightarrow (\sim I(e) \rightarrow Omega(e))$ $(Generalization)$

Therefore, for all the discussed dependence tests, we have a conclusion as shown in Table 3.

As a matter of fact, the former two cases rarely happen in practical applications, while the last case is relatively common. Therefore, the authors are eager to use {I test, omega test} as the minimum complete set for most cases. For the remaining cases, we say {power test} is the minimum complete set of all dependence testing techniques.

6 Related work

Since the parallelizing compilation was proposed, the studies on data dependence analysis have been widely developed [2,4–8,16,18–24] up to now. To evaluate the power of the proposed data dependence testing techniques, literature [4–6] showed empirical based evaluations so as to analyze the accuracy, efficiency, time complexity and tradeoffs between these properties. None of them gave a theoretical result. Golf et al. [2] attempted to find a combination of different dependence tests based on their categories on the variable pattern. The delta test is used in the most complicated case, but we have proved that it is reduced to the omega test. Maydan et al. [7] proposed a subscript pattern based dependence test suite. If all these tests failed, Fourier–Motzkin elimination [8] is used as a time-consuming back up test. However, each test in the suite can be viewed as a restricted form of the Fourier–Motzkin elimination with less time cost. They proposed to apply the Bound and Branch Method to seek integer solutions when all simple tests fail and the Fourier–Motzkin elimination proves there are real solutions somewhere. Ideally, this technique can be transformed into a necessary and sufficient condition with the feasible region $\boldsymbol{BI}$, but it is really a nightmare in practice due to the time complexity. So is the last case of the omega test [24]. Therefore, it is only a sufficient condition in practice. The classical polyhedral compilation framework also investigated data dependence analysis [25–27]. In the polyhedral compilation framework, the dependence test is not only about finding solution to the equality system, but also satisfying the constraint including iteration space bound and dependence polyhedron.

Besides the linear dependence tests, developers proposed some nonlinear dependence tests for static parallelizing compilers in recent years. In 1995, Mohammad [28] presented a nonlinear dependence test, making the symbolic analysis serve as a fundamental step for new dependence testing techniques. Range test was proposed by Blume and Eigenmann [29]. It can handle the nonlinear expressions. Engelen et al. [30] described a unified approach, which is composed of nonlinear GCD test, nonlinear value range test and nonlinear extreme value test, for nonlinear dependence testing. The quadratic test and the QP (Quadratic Programming) test were introduced by Wu and Chu [31] and our research group [32] respectively to handle quadratic subscripts. The former is an exact test, but can only handle array subscripts with one-dimensional quadratic expression. The latter can determine multi-dimensional cases, but it has to be conservative when it cannot assure the absence of dependences. Zhou and Zeng [33] proposed Integer Interval Theory-based nonlinear test called PVI (Polynomial Variable Interval) test. It can be viewed as an extension of the I test. We adopted the quadratic case of PVI test to cooperate with the QP test [34]. The general dependence test presented by Hummel et al. [35] for dynamic, pointer-based data structures is also a nonlinear dependence testing algorithm.

7 Conclusions

We proposed a formal system K-DT that is designed to evaluate the relative power of existing linear data dependence testing techniques, and proved its soundness, adequacy, and consistency respectively. We theoretically evaluated the power of different linear dependence testing techniques, and proposed the upper bounds as well as the minimum complete sets in different cases based on the theorems in K-DT. As K-DT is a classical logic-based formal system that merges the results *yes* and *maybe* into one case, it is not able to distinguish the cases when a dependence test returns *yes* and *maybe*. As a result, it can only analyze the ability of a dependence test to disprove dependences.

There are still many aspects that need to be improved for the K-DT system. First, we analyzed the linear dependence tests and their minimum complete sets. There has been a great deal of work [28–35] for nonlinear dependence tests proposed, so we should pay our attention to these techniques for the future work. Second, K-DT is designed for the system of equations based dependence tests. For those [25–27,29,35] which are not analyzed based on the system of Eq. (1) but other techniques, it cannot analyze their power although the number of the methods like this is very limited.

Acknowledgements We would like to acknowledge anonymous referees for their invaluable comments and suggestions on this paper. Our work is supported by the National Natural Science Foundation of China under Grant No. 61702546.

References

1. Allen R, Kennedy K (2001) Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publisher
2. Goff G, Kennedy K, Tseng CW (1991) Practical dependence testing. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, pp 15–29
3. Allen R, Kennedy K (1984) PFC: a program to convert Fortran to parallel form. In: Hwang K (ed) Supercomputers: design and applications. IEEE Computer Society Press, Silver Spring, pp 186–203
4. Shen Z, Li Z, Yew P (1990) An empirical study of Fortran programs for parallelizing compilers. IEEE Trans Parallel Distrib Syst 1(3):356–364
5. Petersen P, Padua D (1996) Static and dynamic evaluation of data dependence analysis techniques. IEEE Trans Parallel Distrib Syst 7(11):1121–1132
6. Psarrisand K, Kyriakopoulos K (2004) An experimental evaluation of data dependence analysis techniques. IEEE Trans Parallel Distrib Syst 15(3):196–213
7. Maydan DE, Hennessy JL, Lam MS (1991) Efficient and exact data dependence analysis. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, pp 1–14
8. Williams HP (1976) Fourier-Motzkin elimination extension to integer programming problems. J Combin Theory (A) 21(1):118–123
9. Wolfe MJ (1982) Optimizing supercompilers for supercomputers. PhD thesis. Department of Computer Science. University of Illinois, Champaign
10. Kuck D, Muraoka Y, Chen S (1972) On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. IEEE Trans Comput 21(12):1293–1310
11. Muraoka Y (1971) Parallelism exposure and exploitation in programs. PhD thesis. Department of Computer Science. University of Illinois, Champaign
12. Hamilton AG (1988) Logic for mathematicians, 2nd edn. Cambridge University Press, Cambrige
13. Alllen JR (1983) Dependence analysis for subscripted variables and its application to program transformations. Ph.D. thesis. Department of Mathematical Sciences, Rice University
14. Callahan D (1986) Dependence testing in PFC: weak separability. Supercomputer Software Newsletter 2. Department of Computer Science, Rice University, Houston
15. Li ZY, Yew PC, Zhu CQ (1990) An efficient data dependence analysis for parallelizing compilers. IEEE Trans Parallel Distrib Syst 1(1):26–34
16. Wolfe MJ (1995) High performance compilers for parallel computing. Addison-Wesley Press, Boston

17. Shen ZY, Li ZY, Yew PC (1989) An empirical study on array subscripts and data dependencies. In: Proceedings of International Conference on Parallel Processing, pp 145–152
18. Banerjee U, Eigenmann R, Nicolau A, Padua DA (1993) Automatic program parallelization. Proc IEEE 81(2):211–243
19. Kong X, Klappholz D, Psarris K (1991) The I test: an improved dependence test for automatic parallelization and vectorization. IEEE Trans Parallel Distrib Syst 2(3):342–349
20. Knuth DE (1997) The art of computer programming, seminumerical algorithms, vol 2, 3rd edn. Addison-Wesley, Boston
21. Banerjee U (1996) Dependence analysis. Kluwer Academic Publishers, Dordrecht
22. Li ZY, Yew PC, Zhu CQ (1989) Data dependence analysis on multi-dimensional array references. In: Proceedings of the 3rd International Conference on Supercomputing, pp 215–224
23. Wolfe M, Tseng CW (1992) The power test for data dependence. IEEE Trans Parallel Distrib Syst 3(5):591–601
24. Pugh W (1991) The omega test: a fast and practical Integer Programming algorithm for dependence analysis. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, pp 4–13
25. Feautrier P (1992) Some efficient solutions to the affine scheduling problem. part I. one-dimensional time. Int J Parallel. Program 21(5):313–348
26. Feautrier P (1992) Some efficient solutions to the affine scheduling problem. part II. Multi-dimensional time. Int J Parallel. Program 21(6):389–420
27. Bastoul C (2004) Code generation in the polyhedral model is easier than you think. In: Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques, pp 7–16
28. Mohammad RH (1995) Symbolic analysis for parallelizing compilers. Springer, Berlin
29. Blume W, Eigenmann R (1998) Nonlinear and symbolic data dependence testing. IEEE Trans Parallel Distrib Syst 9(12):1180–1194
30. van Engelen RA, Birch J, Shou Y, Gallivan KA (2004) A unified framework for nonlinear dependence testing and symbolic analysis. In: Proceedings of the 18th Annual International Conference on Supercomputing, pp 106–115
31. Wu JH, Chu CP (2007) An exact data dependence test for quadratic expressions. Inf Sci 177(23):5316–5328
32. Zhao J, Zhao RC, Han L, Xu JL (2013) QP test: a dependence test for quadratic array subscripts. IET Softw 7(5):271–282
33. Zhou J, Zeng GH (2008) A general data dependence analysis for parallelizing compilers. J Supercomput 45(2):236–252
34. Zhao J, Zhao RC, Chen X, Zhao B (2015) An improved nonlinear data dependence test. J Supercomput 71(1):340–368
35. Hummel J, Hendren LJ, Nicolau A (1994) A general data dependence test for dynamic, pointer-based data structures. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, pp 218–229