The assignment to A in the if-statement is never executed since the condition is always false. (This is obvious to an observer, but it is extremely difficult to automate this test. In fact, under rather mild assumptions, it is undecidable whether there exist values such a polynomial assumes the value 0, implying that it is in general undecidable whether a statement will be executed.) Thus, in our example the assumption that all array references are valid will impermissibly reduce the overall range for the value I. Assume that the range of the one-dimensional array A is 1..100 and that N=100. Then the first statement will imply that

\[ 1 \leq I \leq 49 \]

but the if-statement would additionally (and incorrectly!) yield the following range for I (after suitable simplification)

\[ 33 \leq I \leq 49. \]

There is another complication: In contrast to a normal for-loop, where the index will either increase or decrease by a specific increment, while-loops do not have to possess directionality. Thus, even if all values of the valid range of an index are assumed, the order in which this occurs need not be regular. Hence, it is not known in which order the array references will occur. Specificaly, in a for-loop the index either consistently increases or decreases. In a while-loop, this need not be true. Indices may be calculated in arbitrary ways. (Note that we do not specify in our examples how the indices are computed.) Consequently, it will be necessary to compute all possible dependences, assuming that any execution order of the array references is possible. However, this is frequently done anyway since dependence tests ultimately determine whether two different references to the same array may reference the same element (dependences of anti-dependences, both of which would have to be preserved by any code transformation).

Note that our technique addresses only the question of whether there exists a dependence. Thus, the problem of rewriting the code to meet some objective (e.g., vectorization, parallelization, etc.) is outside the realm of this paper. Nevertheless, it should be clear that knowing that no dependences exist allows one to vectorize code in ways that cannot be applied if this knowledge is not available. (Note that the problem is that we don’t know about it, not that it does not hold! Clearly, if there are dependences, they will not be removed by our method. However, it is quite possible that no dependences exist but we may not be able to determine this fact. Our method will reduce such cases.) The example below shows how our method may result in significant improvements over other, conventional approaches.

4. A Detailed Example

Consider the following loop where the statements S1, S2, and S3 contain no references to the array A. We further assume that the statements S1, S2, and S3 calculate the index I in some way that is of no concern (and is in fact unknown) to us:

```
while cond do
    S1;
    A(I+1) = B(2*I-1) + C(N-I);
    S2;
    A(3*I+4) = A(2*I-[N/2]) + A(N-3*I-1);
    S3
end
```

We assume that the arrays A, B, and C are defined as one-dimensional arrays of range 1..N, for some (large) positive integer N. Note that given the information indicated in this program fragment, the variable I cannot be determined to be an induction variable since we do not know how it is computed.

From this information, we obtain the following valid ranges for each of the references in the code:

- \[ 1 \leq I \leq N \]
- \[ I = 0, \ldots, (N+1)/2 \]
- \[ I = N, \ldots, N-1 \]
- \[ I = (N/2)+1, \ldots, (N-N)/2 \]
- \[ I = (N-N)/2+1, \ldots, (N+1)/2-1 \]
- \[ I = 0, \ldots, (N-3)/2 \]

Thus, the overall valid range for I is the intersection of all these individual valid ranges, namely

\[ [0..N-1] \cap [1..(N+1)/2] \cap [0..N-1] \cap [-1..(N-N)/3] \cap \ldots \cap \ldots \cap [0..(N-N)/2] \cap [0..(N-N)/3]' \]

under the assumption N>18, since

\[ \max \{ 0, 1, 0, -1, (N/2)+1/2, 2 \} = (N/2)+1/2 \]

and