DF^2
DEEP FLOW DATAFLOW FOR
QA·C 8.1 SOURCE CODE ANALYZER

USER GUIDE

February 2013

This document describes the features and configuration of the QA·C Source Code Analyser dataflow capability.

IMPORTANT NOTICE

DISCLAIMER OF WARRANTY

The staff of Programming Research Ltd have taken due care in preparing this document which is believed to be accurate at the time of printing. However, no liability can be accepted for errors or omissions nor should this document be considered as an expressed or implied warranty that the products described perform as specified within.

COPYRIGHT NOTICE

TRADEMARKS

PRQA, the PRQA logo, and QA·C are registered trademarks of Programming Research Ltd. Windows is a registered trademark of Microsoft Corporation.

CONTACTING PROGRAMMING RESEARCH LTD

For technical support, contact your nearest Programming Research Ltd authorized distributor or you can contact Programming Research's head office:

by telephone on	+44 (0) 1 932 888 080
by fax on	+44 (0) 1 932 888 081
or by e-mail on	support@programmingresearch.com
	www.programmingresearch.com

Table of Contents

1. INTRODUCTION

1.1 Terminology

The term “*dataflow analysis*” refers to a range of techniques in which static analysis of source code is used to analyse run-time behavior of a program. Dataflow analysis can identify a class of problems which may range from serious issues such as undefined behavior to conditions which are of interest simply because they are frequently associated with coding errors.

DF^2 is a *dataflow analysis* engine which functions as an integral component of QA·C static analysis. It identifies some of the following conditions:

1.1.1 Invariant Operations

An expression which always evaluates as ‘*true*’ or always evaluates as ‘*false*’ is described as *invariant*. A non-constant expression which can be identified as *invariant* is frequently associated with a mistake in the logic.

- Invariant logical operations
- Invariant controlling expressions in loop statements and ‘*if*’ statements

1.1.2 Redundant Operations

If the value assigned to a variable is never used or is always overwritten before it can be used, the operation may simply be unnecessary but may also be associated with a mistake in the logic.

- Redundant initializations
- Redundant assignments

Another kind of redundancy occurs if a binary arithmetic operation can be replaced by a constant or one of its operands.

- Redundant arithmetic operations

1.1.3 Control Flow Analysis

- Unreachable code
- Infinite loops
- Executing an implicit return statement in a function with non-void return type

1.1.4 Initialization Tracking

- Using the value of unset data
- Passing the address of unset data to a function parameter defined as a “pointer to const type”

1.1.5 Value Tracking

- Division by zero
- Arithmetic operations on signed data resulting in overflow

- Arithmetic operations on unsigned data resulting in wraparound
- Converting a value to a signed type in which it is not representable
- Converting a negative value to an unsigned type
- Converting a positive value to an unsigned type in which the value is not representable
- Performing a left shift operation on unsigned data resulting in truncation of bits
- Performing a left shift operation on signed data resulting in an implementation defined value
- Performing a shift operation with a right hand operand which is negative or too large
- Assigning a negative value which requires a "two's complement" representation.

1.1.6 Pointer Attribute Tracking

- Dereferencing a NULL pointer
- Performing arithmetic operations on a NULL pointer
- Computing an invalid pointer value – an array bounds violation
- Dereferencing an invalid pointer value – an array bounds violation
- Comparing or subtracting pointers which do not address members of the same array, struct or union

Some of the conditions listed above can be readily identified in simple situations by conventional static code analysis. More complex examples can only be identified when dataflow analysis is used to examine control flow and track the status of objects at run-time.

1.2 Analysis Limitations

Dataflow analysis does not always yield a precise result. It is a technique which can consume a significant amount of computing resources in both memory usage and processing time, and in many situations it is simply impractical to perform an exhaustive analysis. In designing analysis algorithms it becomes necessary to apply judicious simplifications and compromises in order to perform analysis within reasonable constraints of time and resources. In general, dataflow analysis is likely to take considerably more time to execute than conventional static analysis.

Design compromises mean that conditions which should be identified will sometimes go unreported (‘false negatives’) and conditions which are benign may give rise to an unnecessary diagnostic (‘false positives’). The effectiveness of dataflow analysis has to be judged by a number of conflicting criteria:

- The effectiveness in identifying genuine problems (an absence of false negatives).
- The avoidance of incorrect messages (an absence of false positives).
- The speed with which analysis can be performed.
- The ease with which the root causes of a problem can be identified.

1.3 Diagnostics

When analysing a potential problem such as dereferencing a NULL pointer, dataflow may arrive at a variety of conclusions and wish to respond accordingly. For example:

- This pointer will always be NULL.
- This pointer will never be NULL.
- This pointer could be NULL – there is no way of knowing
- This pointer will sometimes be NULL (i.e. on some, but not necessarily all iterations of a loop).
- This pointer will be NULL if a particular path elsewhere is ever executed.
- This pointer will be NULL if a loop construct elsewhere is ever bypassed.

A particular challenge in dataflow analysis is that it is frequently impossible to confirm or dismiss a potential problem without knowing the intentions of the programmer. Typically, dataflow may establish some form of anomaly or contradiction is present in the code. Such anomalies can be reported in more than one way, for example:

1. If the particular path at location A is executed, it is certain that a NULL pointer dereference will occur at point B.
2. If a NULL pointer dereference is to be avoided at point B, the path at location A must be unreachable.

The essence of problems such as this is that we have no way of knowing exactly where the root of the problem lies. Dataflow analysis is only able to point out that there are 2 potential problems and that one of them is bound to exist; either there is a redundant path in the code or there is a NULL pointer dereference.

Dataflow analysis is a technique which has inherent limitations; there will always be run-time problems which cannot be diagnosed simply because of a lack of sufficient information within the available source code. For example, dereferencing a pointer which is a function parameter without first checking that the pointer is not NULL is a *potential* vulnerability – but is not in itself evidence of a definite problem.

2. MESSAGE CATEGORIES

In the previous chapter, various classes of problem which dataflow analysis may identify were listed:

1. Invariant operations
2. Redundant operations
3. Control flow analysis
4. Initialization tracking
5. Value tracking
6. Pointer attribute tracking

Items 4, 5 and 6 in this list characterise issues which cannot be satisfactorily described in a single message. A particular issue has to be addressed by a family of related messages in order to clarify the context in which the diagnostic is being generated. In DF^2, five terms are used in a special sense to distinguish these contexts.

- *Constant*
- *Definite*
- *Apparent*
- *Suspicious*
- *Possible*

2.1 Constant Messages

Strictly speaking, a **constant** message is not a dataflow analysis message at all. It identifies an issue which can be deduced without analysis of either control flow or the run-time status of variables. For example:

```
void foo (void)
{
  int buf[25] = {0};
  ...
  buf[100] = 0;          /* Message 2840 */
}
```

An array bounds violation in this code is identified because the array subscript value is a constant expression and it can be deduced that the assignment operation addresses an array element outside the declared bounds of the array. The problem would be identified as follows:

2840: Constant: Dereference of an invalid pointer value.

2.2 Definite Messages

A **definite** message simply identifies an issue which will *definitely* occur. For example:

```
void f1 (unsigned int n)
{
  int buf[10] = {0};

  if (n > 10)
  {
    buf[n] = 100;          /* Message 2841 */
  }
}
```

Perhaps the programmer used a '>' operator by mistake instead of a '<' in the '*if*' statement. The result of this mistake is, of course, an array bounds violation; the value 100 will be assigned to an element which is outside the bounds of the array. This problem would be identified as follows:

2841: Definite: Dereference of an invalid pointer value.

Another example ...

```
void f2 (void)
{
  int buf[10];
  int *p;

  p = buf;
  ...
  p[100] = 0;          /* Message 2841 */
}
```

In this code, the array bounds violation occurs indirectly through a pointer. Dataflow analysis of the code determines that the size of the object addressed by the pointer '*p*' and deduces that the array subscript operation will result in the dereferencing of an invalid pointer value.

Notice that a **definite** message does not imply that the issue will occur every time the statement is executed (e.g. on every iteration of a loop), but it will definitely occur if the statement is ever reachable. For example, a definite array bounds violation message may be generated in the context of a loop even though the violation may not occur in every iteration (also see Section 3.4). For example:

```
void f3 (void)
{
  int buf[10];
  int i;

  for (i = 0; i <= 10; ++i)
  {
    buf[i] = i;                /* Message 2841 */
  }
}
```

2841: Definite: Dereference of an invalid pointer value.

2.3 Apparent Messages

An *apparent* message identifies an issue that will occur **unless** a specific path elsewhere in the code, associated with an '*if*' or '*switch*' statement or a *conditional operation (? :)* is always bypassed. In other words, if the issue identified by the apparent message is **not** to occur then there must be a redundant path elsewhere.

Consider the following example. The code exhibits an anomaly: the code controlled by the '*if*' statement is only reachable if the value of '*n*' is greater than or equal to 10, and if this happens, an array bounds violation will occur when referencing '*buf[n]*'.

```
void f1 (unsigned int n)
{
  int buf[10];

  if (n >= 10)                /* Redundant path ?  */
  {
    ...                      /* n is not modified */
  }
  ...
  buf[n] = 100;               /* Message 2842     */
}
```

There is therefore ***either*** an array bounds problem ***or*** a redundant path. The following message will be generated:

2842: Apparent: Dereference of an invalid pointer value.

The same message will be generated if the relational operator is changed as shown below, because a redundant path still exists.

```
void f2 (unsigned int n)
{
  int buf[10];

  if (n < 10)                  /* Redundant path ?  */
  {
    ...                       /* n is not modified */
  }
  ...
  buf[n] = 100;                /* Message 2842     */
}
```

In each of the above examples, the code can be seen to reflect 2 contradictory assumptions:

- Implicit in the '*if*' statement is the assumption that the function parameter '*n*' will sometimes have a value **greater than or equal to 10**. If this assumption is not true, the '*if*' statement is effectively redundant and we have a redundant path.
- Implicit in the array subscript operation is the assumption that the value of '*n*' must always be **less than 10**. If this assumption is not true, the array subscript operation will result in an array bounds violation.

There is therefore an array bounds problem UNLESS the '*if*' statement gives rise to a redundant path.

Both the above examples illustrate instances of what may be described as '*regular apparent*' conditions. There are other similar situations which exhibit the same anomaly but where the array reference precedes the possible redundant path condition. These are described as '*reverse apparent*' conditions.

```
void f3 (unsigned int n)
{
  int buf[10];

  buf[n] = 100;                /* Message 2842    */
  ...

  if (n < 10)                  /* Redundant path ? */
  {
    ...
  }
}
```

2.4 Suspicious Messages

A **suspicious** message identifies an issue that will occur if certain conditions are not fulfilled in the execution of a loop construct.

Notice that *apparent* and *suspicious* messages both identify issues associated with the use of paths. However,

- An *apparent* message reports that the dataflow issue will occur unless a path elsewhere is redundant, i.e. it is *never executed*.
- A *suspicious* message identifies assumptions which are implied in a loop construct if a problem is to be avoided. For example, avoidance of the problem may rest on assumptions such as:

  a) The loop is always executed at least once
  b) A certain path through the loop is always executed at least once
  c) A certain path through the loop is always executed on the final iteration

Suspicious messages are therefore informational in nature and are less specific than apparent messages. They do not necessarily identify that a real problem exists but merely attempt to expose assumptions that may have been made unwittingly.

Consider two examples which both generate message 2843:

2843: Suspicious: Dereference of an invalid pointer value.

Example 1: An array bounds violation will occur unless:

a) the loop is entered, and
b) the 'AAA' statement is executed in at least one iteration of the loop.

```
void path_never_executed (int i)
{
  int array[5];
  int j = 10;
  int k;

  for (k = 0; k < 10; ++k)
  {
    if (i > k)
    {
      j = 0;                    /* AAA */
    }
  }

  array[j] = 0;                 /* Message 2843 */
}
```

Example 2: An array bounds violation will occur unless:

a) the loop is entered, and
b) the 'AAA' statement is executed in the final iteration of the loop.

```
void path_executed_on_last_iter (int i)
{
  int array[5];
  int j;
  int k;

  for (k = 0; k < 10; ++k)
  {
    if (i > k)
    {
      j = 10;
    }
    else
    {
      j = 0;                         /* AAA */
    }
  }

  array[j] = 0;                    /* Message 2843 */
}
```

2.5 Possible Messages

A possible message identifies issues about which there is complete uncertainty, either because of the limitations of the dataflow analysis engine or because the code is not written in a way which allows a more definitive conclusion to be drawn. In practice, unless code is developed defensively with a high degree of discipline, possible messages will be encountered frequently.

'Possible' messages are not implemented for all dataflow conditions because their usefulness would be very limited. Consider the following:

```
int foo (unsigned int n)
{
  return 1000 / n;                 /* Message 2834 */
}
```

Because the value of the function parameter '*n*' on entry is considered to be unknown, DF^2 will generate message 2834 to indicate that the divisor could be zero.

2834: Possible: Division by zero.

Note that if function '*foo*' is ever called within the same *translation unit* with argument 0, a definite message will also be generated on the same line, with suitable sub-messages to indicate why the condition arises.

2831: Definite: Division by zero.

3. IMPLEMENTATION

3.1 Solver Technology

DF^2 employs an SMT solver engine to model the status of objects at run time. Code statements which impose constraints on the value of variables are translated into a series of assertions which are supplied to the solver. The solver is interrogated with queries in order to identify conditions which need to be reported.

3.1.1 Intervariable Dependency

The power of a solver engine in dataflow analysis is that it is able to provide an accurate model of the values of variables throughout the execution of a function and also model the relationships which exist between variables.

```
void foo (int i)
{
  int * p = (i > 0) ? (int *)0x1234 : 0;

  /* ... */                          /* p can be NULL here */

  if (i > 10)
  {
    *p = 1;                         /* p is never NULL here */
  }
}
```

3.1.2 Wraparound and Overflow

One aspect of solver technology which can cause some confusion is that it may sometimes arrive at conclusions which are theoretically correct but which seem at first sight to be unreasonable. Consider a trivial example.

It might appear that the control expression in the first '*if*' statement shown below will only be 'true' if x has a value of 2U. We might therefore expect that the control expression in the second '*if*' statement should be identified as *invariant* – i.e. always 'true'. In fact this is not so, and instead we find that message 2912 is generated on the first '*if*' statement:

```
void f1 (unsigned int x)
{
  if ((x * 2U) == 4U)                      /* 2912 */
  {
    if (x == 2U)                          /*      */
```

```
      {
      }
    }
  }
```

2912: Apparent: Wraparound in unsigned arithmetic operation.

The solver determines that there are 2 possible values of '*x*' which satisfy the first equality operation; 2U is the most obvious, but because unsigned wraparound may occur in the multiplication operation, a value of 32770U is also a possible solution (assuming a 16 bit integer implementation).

Message 2912 is generated because an 'apparent' anomaly is identified. Either there is an unreachable path issue or a wraparound issue. Either we have a situation where '*x*' is always equal to 2U (which results in an unreachable path), or else wraparound must occur in the first '*if*' expression when '*x*' is multiplied by 2U.

A similar situation arises with signed arithmetic but overflow is identified rather than wraparound.

```
void f2 (signed int x)
{
  if ((x * 2) == 4)                   /* 2802 */
  {
    if (x == 2)                      /*      */
    {
    }
  }
}
```

2802: Apparent: Overflow in signed arithmetic operation.

3.2 Variable tracking

A primary function of dataflow analysis is tracking the value, status and attributes of objects at run-time. Five forms of tracking are recognised:

3.2.1 Initialization Tracking

The initialization status of local automatic objects is tracked in order to identify any attempt to use an 'unset' value.

3.2.2 Value Tracking

Tracking of arithmetic values is implemented for objects of *integer* or *pointer* type. Furthermore, for pointers:

3.2.3 Pointer Attribute Tracking

Various attributes associated with an object of pointer type are tracked:

- a unique object identifier
- the size of the object into which the pointer is pointing
- the offset within that object to which the pointer is pointing

These attributes are used to identify

a) whether 2 pointers address the same object.
b) when pointer arithmetic results in the generation of an invalid pointer.
c) when an attempt is made to dereference an address outside the bounds of an object.

3.2.4 Pointer Aliasing

When an address of an object with *integer*, *class*, *struct* or *union* type is assigned to a pointer, or such a pointer is assigned to another pointer, the value of pointer dereference (object pointed to) is modelled and tracking functions described above are applied to it.

3.2.5 Inter-function Analysis

When an argument is passed to a function its value is bound to the corresponding parameter. Additionally, for a pointer parameter, the value of pointer dereference of argument is propagated to that of parameter. Similarly, the value (including pointer dereference) is propagated from return expression to call site.

3.2.5.1 Objects of Scalar Type

Initialization tracking is implemented for *automatic* objects of any *scalar* type.

Value tracking is implemented for objects of all *integer* types (so *floating* types are excluded).

Pointer attribute tracking and pointer aliasing are implemented for objects of pointer type.

Analysis of *enum* variables is complicated by the fact that their internal representation is implementation defined. An *enum* constant is always implemented in type *signed int*, but the type in which an *enum* variable is represented may vary with the implementation and may be different for each *enum* type. No options are currently available to configure the type in which an *enum* is represented. DF^2 assumes that all *enum* variables are implemented in type *signed int*.

In practice, the analysis of *enum* variables becomes less critical if coding guidelines are applied to restrict the way in which variables of *enum* type are used, for example by preventing conversion of a 'non-enum' expression to an *enum* type.

3.2.5.2 Objects of aggregate type

arrays

No tracking is implemented on individual elements of an array. For the purposes of ***initialization tracking***, an array is treated as a single discrete object. If one element of a local automatic array is initialized, DF^2 assumes that the whole array is initialized.

struct and class

Value tracking and ***initialization tracking*** are implemented for each member of a struct, but not ***pointer tracking/aliasing***.

3.2.5.3 Objects of Union Type

In order to track the behavior of individual union members correctly, it would be necessary to provide a faithful model of the storage of each member with an accurate representation of several aspects of behavior which are implementation-defined. DF^2 assumes a little-endian memory model with type sizes and alignments which reflect settings of the configuration options: *-size* and *-align*. Within the assumptions of this model, the value of one member will reflect values assigned to another member; however it must also be remembered that DF^2 will not track the value of any member of array type.

```
union U
{
  unsigned int ui;
  struct S
  {
    unsigned short s1;
    unsigned short s2;
  } s;
};

void foo (void)
{
  union U u = { 0x00020001 };
  if (u.s.s1 == 1) { }          /* 2995 & 2991 */
  if (u.s.s2 == 2) { }          /* 2995 & 2991 */
}
```

2995: The result of this logical operation is always 'true'.

2991: The value of this 'if' controlling expression is always true.

3.2.6 volatile Objects

volatile objects or members cannot modelled, as their value can change on every access.

3.3 API usage checking

3.3.1 Function Arguments

DF^2 performs analysis on the arguments to some functions defined in the C Standard Library to identify issues such as:

- an invalid NULL pointer argument
- a pointer argument which addresses an array which is of insufficient length
- the maximum number of characters to be written is larger than the target buffer size
- a copy operation being performed on overlapping objects

Such issues, which would normally result in undefined behavior inside the function, are identified with corresponding dataflow messages at the location of the function call.

For example, the argument in a call to '*atof*' cannot be a NULL pointer:

```
#include <stdlib.h>

void f1 (void)
{
  char * c = NULL;
  atof (c);                /* Message 2811 */
}
```

2811: Definite: Dereference of NULL pointer.

The third parameter in a call to strncpy should not be larger than the size of the buffer specified as the first parameter:

```
void f2 (char * src)
{
    char dst[10];
    size_t size = 20U;
    strncpy (dst, src, size);   /* Message 2846 */
}
```

2846: Definite: Maximum number of characters to be written is larger than the target buffer size.

Note that even if the actual size of the source buffer is small enough to fit in the destination buffer, message 2846 will still be issued (despite strncpy safely stopping copying characters when it reaches the null terminator in src) - there is still a mismatch between the destination buffer and number of bytes to copy.

Other standard library functions that accept a destination buffer are checked, for example:

```
void f3(void)
{
    char buf[10];
    sprintf(buf, "1234567890");         /* Message 2840 */
    strcpy(buf, "1234567890");          /* Message 2840 */
}
```

2840: Constant: Dereference of an invalid pointer value.

3.3.2 Function Return Values

Analysis can also use known characteristics of a Standard Library function when the return value is subsequently used. For example:

- *malloc* always returns either NULL, or a suitably sized buffer of bytes
- *strchr* always returns either NULL, or a pointer to an element within the array addressed by the first argument

```
#include <stdlib.h>

void f2 (void)
{
  int * m = (int *) malloc(3 * sizeof(int));
  if (m != NULL)
  {
    m[3] = 0;              /* Message 2841 */
  }
}
```

2841: Definite: Dereference of an invalid pointer value.

3.4 Modelling Limitations

There are a number of specific aspects in dataflow analysis which may give rise to ‘false positives’ and ‘false negatives’.

3.4.1 Inter-function Analysis

DF^2 currently performs dataflow analysis including modelling of function calls within a single *translation unit* at a time. For example:

```
void foo (int * ptr)
{
  *ptr = 0;
}

void bar (void)
{
  int i;
```

```
  foo (&i);
  1 / i;                /* Message 2831 */
}
```

2831: Definite: Division by zero.

This issue would not be detected if '*foo*' and '*bar*' were defined in different *translation units*.

3.4.2 Loops

Loops typically modify one or more variables but, depending on the complexity of the loop, it is generally impractical to track the values of every variable through every iteration. A loop variable is defined as any variable that is modified within the loop. The value of a loop variable is analysed in three contexts:

- the value in the first iteration
- the value in any intermediate iteration
- the value in the last iteration

If DF^2 finds an issue, sub-message 1571 is used to identify whether it occurs in the first, intermediate or last iteration. For example:

```
void loop (void)
{
  int i;

  for (i = 0; i < 10; ++i)
  {
    1 / i;              /* Message 2831 - first iteration */
    1 / (i - 3);         /* Message 2831 - intermediate iteration */
    1 / (i - 9);         /* Message 2831 - last iteration */
  }

  1 / (i - 10);          /* Message 2831 */
}
```

2831: Definite: Division by zero.

3.4.3 Variables with Unknown Value

The status of a variable often has to be classified as '*unknown*' – meaning that no relevant information is available by which its value or its attributes may be determined. This is the status, for example, of

- function parameters on entry to a function
- global variables on entry to a function

- variable assigned the result of a function call

unless the function is called within the same *translation unit*, and its body can be expanded into the caller (it's not indirectly recursive or too large for the supplied -po df:inter option, see Section 4). It is particularly difficult to track the status of a global variable because of the uncertainty introduced whenever another function is called. Whereas, local variables can be monitored with relative confidence, tracking a global variable is often a far more complex task. DF^2 tracks the status of a global variable within the current function, but, if a function call is encountered, its status immediately becomes *unknown*, unless the function call can be expanded.

3.4.4 Pointer Aliasing

Assignment involving a dereferenced return value of an expanded function call is not processed. For example:

```
int * gp;

int * foo (void)
{
  return gp;
}

void bar (void)
{
  *(foo ()) = 0;
  1 / *gp;                /* Message 2834 instead of 2831 */
}
```

Additionally, there are some cases where the value of a pointer dereference is not modelled in the entire function body:

- for a pointer to a pointer, i.e. 2 or more levels of indirection
- if the pointer is involved in any pointer arithmetic, including an array subscript operation.
- the pointer is assigned the result of an (implicit) pointer cast where at least one of the 'to' or 'from' types is a pointer to void, struct or union, or the 'to' type is a pointer to a type with a smaller value range than that for the 'from' type.
- if the pointer is assigned another pointer that is not modelled.
- if the pointer is passed as an argument to a parameter that is not modelled.

4. CONFIGURATION

4.1 Preparation

Dataflow analysis is intrinsically an intensive and, sometimes, time-consuming process and analysis will usually take significantly longer if dataflow is enabled. In practice, it cannot be conducted on code which contains certain syntax errors (level 9 messages) or even, in some cases, constraint errors (level 8 messages) and it may be aborted for the current function or even the entire translation unit if such errors are encountered. In these cases one of the following two Dataflow Recovery messages are generated:

2753: As a result of error message '%s', dataflow analysis of the remainder of this function is not possible.

2754: As a result of error message '%s', dataflow analysis of the remainder of this translation unit is not possible.

It is therefore wise to ensure that configuration and major coding problems have been addressed in standard QA·C analysis before enabling dataflow analysis.

The complete list of level 9 and level 8 messages highlighting problems with the analysed source code in the presence of which dataflow analysis cannot be conducted can be found in Appendix B.

Dataflow analysis will be most effective when examining code that is well structured. Unstructured code can present particular analysis difficulties and dataflow analysis of a function will be terminated, for example, when a *goto* statement jumping to an earlier label is encountered.

4.2 Activating Dataflow Analysis

Dataflow may be activated within the GUI. The option used to enable dataflow analysis is as follows:

```
-EnableDataflow+
```

For information on how to do it please see the QA·C 8.1 User's Guide.

4.3 Inter-function Analysis

The depth of inter-function analysis can be adjusted by specifying the following option (for example in “Dataflow Analysis Settings->Advanced”):

```
-po df::inter=X
```

where the value $2*10^X$ specifies the maximum number of simplified statements a called function can contain for it to be expanded into the caller. A simplified statement refers to an internal deconstruction of source-level statements which includes expansion of temporaries, deconstruction of compound statements according to sequence points, and other requirements of dataflow examination. Therefore, the number of explicit statements in a function will nearly always be less than the number of simple statements.

Additionally, the caller is not allowed to exceed 200,000 simplified statements as a result of inter-function analysis. Thus a subsequent function call may not be expanded if it would result in the caller exceeding this limit, even if the called function itself has fewer than $2*10^X$ simplified statements. *X* can be at most 5, and with this value message 2756 will be issued, whenever a called function cannot be expanded due to its size.

4.4 Functions Defined in Header Files

Dataflow analysis is not performed on a function located within a suppressed header file (with -Q option). Additionally, functions defined in unsuppressed header files will be analysed if the following runtime option is set:

```
-po df::analyse_header_functions+
```

The option is not set by default.

4.5 Analysis Timeout

Two options have been implemented which provide control over the time allowed for dataflow analysis.

```
-po df::query_timeout=n
-po df::function_timeout=n
```

These options are case sensitive and the numeric argument represents a maximum time measured in milliseconds.

During the course of dataflow analysis, many queries will typically be submitted to the solver engine. Most queries will be conducted within the passage of a few milliseconds, but an individual query can sometimes take an excessive amount of time to reach a conclusion. This may be indicative of some particularly complex code or a difficult sequence of operations.

The *df::query_timeout* option allows a time limit to be applied to an individual query. By default this limit is set at 10000 milliseconds (10 seconds). If such a timeout occurs on an individual query, it may result in a false negative, but in practice, this is unlikely unless the timeout value is substantially reduced. A query which results in a timeout rarely yields a result of any significance.

The *df::function_timeout* option applies a time limit to the analysis of an entire function. However checking of the function timeout limit is never applied until the current query has been completed or has itself timed out. If no function timeout value is specified, analysis will continue without limit. When a function timeout occurs, message 2755 is generated.

Note: dataflow performance is correlated to the number of queries, which is dependent on how much and what kind of analysis is being performed. Reducing the set of dataflow messages will reduce the number of queries and so will improve performance.

4.6 Flexible Array Member

If the last member of a struct is an array of 1 element, the member is treated in a similar way to a C99 *flexible array member*, and the upper bound is not checked by Dataflow by default. However, if the following option is specified, Dataflow will consider the size of this array to be 1, instead:

```
-po df:: struct_last_array_member_size_1
```

4.7 Dataflow Dependent Metric Values

QA·C generates a range of function-based, file-based and project-based metrics. Eight of the function-based metrics are computed during dataflow analysis:

- STAV1 Average size of function statements (variant 1)
- STAV2 Average size of function statements (variant 2)
- STAV3 Average size of function statements (variant 3)
- STST1 Number of statements in function (variant 1)
- STST2 Number of statements in function (variant 2)
- STST3 Number of statements in function (variant 3)
- STRET Number of function return points
- STUNR Number of unreachable statements

If dataflow is disabled, these metrics will not appear in the metric output. The same is true for the case when the function that the above metrics are based on is not analysed due to dataflow being terminated (see Appendix B).

5. UPGRADING FROM PREVIOUS VERSION OF QA·C

DF^2 is a component of QA·C versions 8.0 and 8.1 and it represents an enormous improvement over the previous dataflow implementations prior to QA·C 8.0. The section highlights some significant changes for people upgrading from versions of QA·C earlier than QA·C 8.0.

5.1 Analysis Speed

Analysis using DF^2 will usually take substantially longer than was required in versions of QA·C prior to QA·C 8.0. It is therefore important to adjust to an approach in which dataflow analysis is only used selectively. The QA·C parser is now able to run in 2 distinct modes, *with* dataflow and *without* dataflow. Analysis *without* dataflow will proceed on par with previous versions; but analysis *with* dataflow may be slower by a factor of 10 or more.

Dataflow analysis of a function cannot proceed if major problems such as syntax errors or certain constraint errors are encountered in the code. It is therefore wise to ensure that all such problems are resolved *before* enabling DF^2.

Other factors which will influence analysis speed are the analysis timeout options. These are the options used to impose a limit on the time allowed for analysis of an individual query and for analysis of a function (see Section 4). Additionally, analysis time will in general increase with higher inter-function analysis depth.

5.2 Dataflow Message System

DF^2 supports the generation of more than 100 new messages. These messages are numbered in the range 2740-2999. They are grouped under various headings which include messages of varying importance, but all have been implemented in Message Level 5, except for Dataflow Recovery messages 275x, which are associated with level 0. The message groupings are as follows:

- Conversion to signed
- Conversion to unsigned
- Shift operations
- Overflow and wraparound
- Arrays
- Pointers
- NULL pointers
- Unset data
- Redundancy
- Invariant operations
- Control flow

5.3 'Old' and 'new' Messages

27 messages associated with the '*old*' dataflow engine have been removed (see Appendix A for details). The functionality of each '*old*' message has been superseded by a '*new*' message which is functionally equivalent or superior.

5.4 Changes in the Behavior of the –o and –n Options

The visibility of messages at view time in QA·C is controlled by a number of options including:

- -n -Nomsg
- -o, -Only
- -hdr, -HDRsuppress
- -su, -SUppresslvl

In versions of QA·C earlier than QA·C 8.0, all diagnostics were output to the .err file during the code analysis phase regardless of the settings of these display options. It was therefore possible to change the setting of these options at viewing time and diagnostics which had previously been invisible would be made visible. The one exception to this approach was the –q option:

- -q -Quiet

The effect of the –q option has always been to suppress the generation of diagnostics at the analysis stage so that they are not output to the .err file and can never be viewed, regardless of the setting of other options.

In QA·C 8.0 and QA·C 8.1 a fundamental change has been implemented in the way that message diagnostics are generated. The –o and –n options now function in a similar manner to the –q option. If a message is suppressed by the –o or –n options, no diagnostics will be output to the .err file and those messages can never be visible at view time without performing a new analysis.

The functionality of options –hdr and -su remains unchanged; they continue to be view-time options which can only affect the visibility of diagnostics which have been output to the .err file.

This functional change has been implemented with 2 objectives in mind:

a) A reduction in the size of the .err file

b) Improved performance in dataflow analysis by reducing the scope of the analysis.

6. APPENDIX A: DATAFLOW MESSAGES

New Msg No.	Message Text	Old Msg No.
Constant invariants		
2740	*This loop controlling expression is a constant expression and its value is 'true'.*	3323
2741	*This 'if' controlling expression is a constant expression and its value is 'true'.*	3346
2742	*This 'if' controlling expression is a constant expression and its value is 'false'.*	3329
2743	*This 'do - while' loop controlling expression is a constant expression and its value is 'false'. The loop will only be executed once.*	3361
2744	*This 'while' or 'for' loop controlling expression is a constant expression and its value is 'false'. The loop will not be entered.*	3325
Recovery		
2750	*Internal dataflow problem. Dataflow analysis continues with the next function. Please inform Programming Research.*	0097
2751	*This function is too complex. Dataflow analysis continues with the next function.*	
2752	*This '%s' results in the function being too complex. Dataflow analysis continues with the next function.*	
2755	*Analysis time of function '%s' has exceeded the configured maximum: '%s ms'. Dataflow analysis continues with the next function.*	
2756	*Could not expand function call to '%1s' with maximum '-po df::inter' value.*	
2757	*Could not analyse function '%1s'.*	
Pointer Comparison or Subtraction		
2771	*Definite: These pointers address different objects.*	
2772	*Apparent: These pointers address different objects.*	
2773	*Suspicious: These pointers address different objects.*	
2776	*Definite: Copy between overlapping objects.*	
2777	*Apparent: Copy between overlapping objects.*	
2778	*Suspicious: Copy between overlapping objects.*	
Right operand of shift is undefined.		
2790	*Constant: Right hand operand of shift operator is negative or too large.*	0500 0501
2791	*Definite: Right hand operand of shift operator is negative or too large.*	
2792	*Apparent: Right hand operand of shift operator is negative or too large.*	
2793	*Suspicious: Right hand operand of shift operator is negative or too large.*	

New Msg No.	Message Text	Old Msg No.
Overflow		
2800	*Constant: Overflow in signed arithmetic operation.*	0278
2801	*Definite: Overflow in signed arithmetic operation.*	0296
2802	*Apparent: Overflow in signed arithmetic operation.*	0297
2803	*Suspicious: Overflow in signed arithmetic operation.*	0297
Dereference of NULL pointer		
2810	*Constant: Dereference of NULL pointer.*	0503
2811	*Definite: Dereference of NULL pointer.*	0504
2812	*Apparent: Dereference of NULL pointer.*	0505
2813	*Suspicious: Dereference of NULL pointer.*	0506
2814	*Possible: Dereference of NULL pointer.*	0506
Arithmetic operation on NULL pointer		
2820	*Constant: Arithmetic operation on NULL pointer.*	0507
2821	*Definite: Arithmetic operation on NULL pointer.*	0508
2822	*Apparent: Arithmetic operation on NULL pointer.*	0509
2823	*Suspicious: Arithmetic operation on NULL pointer.*	0510
2824	*Possible: Arithmetic operation on NULL pointer.*	0510
Division by zero		
2830	*Constant: Division by zero.*	0586
2831	*Definite: Division by zero.*	0587
2832	*Apparent: Division by zero.*	3680
2833	*Suspicious: Division by zero.*	3685
2834	*Possible: Division by zero.*	3689
Array bounds violation		
2840	*Constant: Dereference of an invalid pointer value.*	3680
2841	*Definite: Dereference of an invalid pointer value.*	3685
2842	*Apparent: Dereference of an invalid pointer value*	3689
2843	*Suspicious: Dereference of an invalid pointer value.*	
2845	*Constant: Maximum number of characters to be written is larger than the target buffer size.*	
2846	*Definite: Maximum number of characters to be written is larger than the target buffer size.*	
2847	*Apparent: Maximum number of characters to be written is larger than the target buffer size.*	
2848	*Suspicious: Maximum number of characters to be written is larger than the target buffer size.*	
Conversion to signed type which cannot represent value		
2850	*Constant: Implicit conversion to a signed integer type of insufficient size.*	0274
2851	*Definite: Implicit conversion to a signed integer type of insufficient size.*	0273

New Msg No.	Message Text	Old Msg No.
2852	*Apparent: Implicit conversion to a signed integer type of insufficient size.*	0272
2853	*Suspicious: Implicit conversion to a signed integer type of insufficient size.*	0272
2855	*Constant: Casting to a signed integer type of insufficient size.*	0974
2856	*Definite: Casting to a signed integer type of insufficient size.*	0273
2857	*Apparent: Casting to a signed integer type of insufficient size.*	0272
2858	*Suspicious: Casting to a signed integer type of insufficient size.*	0272
Left shift into sign bit		
2860	*Constant: Implementation-defined value resulting from left shift operation on expression of signed type.*	0274
2861	*Definite: Implementation-defined value resulting from left shift operation on expression of signed type.*	0273
2862	*Apparent: Implementation-defined value resulting from left shift operation on expression of signed type.*	0272
2863	*Suspicious: Implementation-defined value resulting from left shift operation on expression of signed type.*	0272
Infinite loops		
2870	*Infinite loop construct with constant control expression.*	
2871	*Infinite loop identified.*	
2872	*This loop, if entered, will never terminate.*	
2877	*This loop will never be executed more than once.*	2465
Control Flow		
2880	*This code is unreachable.*	3201
2881	*The code in this 'default' clause is unreachable.*	2018
2882	*This 'switch' statement will bypass the initialization of local variables.*	0689
2883	*This 'goto' statement will always bypass the initialization of local variables.*	1313 3311
2887	*Function 'main' ends with an implicit 'return' statement.*	0744
2888	*This function has been declared with a non-void 'return' type but ends with an implicit 'return' statement.*	0744
2889	*This function has more than one 'return' path.*	2006
Conversion of negative value to an unsigned type		
2890	*Constant: Negative value implicitly converted to an unsigned type.*	0277
2891	*Definite: Negative value implicitly converted to an unsigned type.*	0290
2892	*Apparent: Negative value implicitly converted to an unsigned type.*	0291
2893	*Suspicious: Negative value implicitly converted to an unsigned type.*	0291
2895	*Constant: Negative value cast to an unsigned type.*	0977
2896	*Definite: Negative value cast to an unsigned type.*	0290

New Msg No.	Message Text	Old Msg No.
2897	*Apparent: Negative value cast to an unsigned type.*	0291
2898	*Suspicious: Negative value cast to an unsigned type.*	0291
Truncation of positive value during implicit conversion to an unsigned type		
2900	*Constant: Positive integer value truncated by implicit conversion to a smaller unsigned type.*	3306
2901	*Definite: Positive integer value truncated by implicit conversion to a smaller unsigned type.*	3296
2902	*Apparent: Positive integer value truncated by implicit conversion to a smaller unsigned type.*	3297
2903	*Suspicious: Positive integer value truncated by implicit conversion to a smaller unsigned type.*	3297
2905	*Constant: Positive integer value truncated by cast to a smaller unsigned type.*	3290
2906	*Definite: Positive integer value truncated by cast to a smaller unsigned type.*	
2907	*Apparent: Positive integer value truncated by cast to a smaller unsigned type.*	
2908	*Suspicious: Positive integer value truncated by cast to a smaller unsigned type.*	
Wraparound in unsigned addition, subtraction or multiplication		
2910	*Constant: Wraparound in unsigned arithmetic operation.*	3302 3303 3304
2911	*Definite: Wraparound in unsigned arithmetic operation.*	3372
2912	*Apparent: Wraparound in unsigned arithmetic operation.*	3382
2913	*Suspicious: Wraparound in unsigned arithmetic operation.*	3382
Loss of bits in an unsigned left shift operation		
2920	*Constant: Left shift operation on expression of unsigned type results in loss of high order bits.*	3301
2921	*Definite: Left shift operation on expression of unsigned type results in loss of high order bits.*	3371
2922	*Apparent: Left shift operation on expression of unsigned type results in loss of high order bits.*	3381
2923	*Suspicious: Left shift operation on expression of unsigned type results in loss of high order bits.*	
Computing an invalid pointer value		
2930	*Constant: Computing an invalid pointer value.*	3680
2931	*Definite: Computing an invalid pointer value.*	3685
2932	*Apparent: Computing an invalid pointer value.*	3689
2933	*Suspicious: Computing an invalid pointer value.*	
Use of negative value which assumes two's complement representation		
2940	*Constant: Result of implicit conversion is only representable in a two's complement implementation.*	0280

New Msg No.	Message Text	Old Msg No.
2941	*Definite: Result of implicit conversion is only representable in a two's complement implementation.*	0281
2942	*Apparent: Result of implicit conversion is only representable in a two's complement implementation.*	0282
2943	*Suspicious: Result of implicit conversion is only representable in a two's complement implementation.*	0282
2945	*Constant: Result of cast is only representable in a two's complement implementation.*	0980
2946	*Definite: Result of cast is only representable in a two's complement implementation.*	0281
2947	*Apparent: Result of cast is only representable in a two's complement implementation.*	0282
2948	*Suspicious: Result of cast is only representable in a two's complement implementation.*	0282
Use of negative value in array subscript or pointer operation		
2950	*Constant: Negative value used in array subscript or pointer arithmetic operation.*	3681
2951	*Definite: Negative value used in array subscript or pointer arithmetic operation.*	3686
2952	*Apparent: Negative value used in array subscript or pointer arithmetic operation.*	3690
2953	*Suspicious: Negative value used in array subscript or pointer arithmetic operation.*	
Use of unset data		
2961	*Definite: Using value of uninitialized automatic object.*	3321
2962	*Apparent: Using value of uninitialized automatic object.*	3347
2963	*Suspicious: Using value of uninitialized automatic object.*	3353
Passing unset pointer to function which expects pointer to const		
2971	*Definite: Passing address of uninitialized object '%s' to a function parameter declared as a pointer to const.*	3348
2972	*Apparent: Passing address of uninitialized object '%s' to a function parameter declared as a pointer to const.*	3349
2973	*Suspicious: Passing address of uninitialized object '%s' to a function parameter declared as a pointer to const.*	3354
Redundant initialization or assignment		
2980	*The value of function parameter '%s' is never used before being modified.*	3195
2981	*This initialization is redundant. The value of '%s' is never used before being modified.*	3197
2982	*This assignment is redundant. The value of '%s' is never used before being modified.*	3198
2983	*This assignment is redundant. The value of '%s' is never subsequently used.*	3199

New Msg No.	Message Text	Old Msg No.
3197	*This operation is redundant. The value of the result is always '%1s'.*	
3198	*This operation is redundant. The value of the result is always that of the left-hand operand.*	
3199	*This operation is redundant. The value of the result is always that of the right-hand operand.*	
Invariant operations		
2990	*The value of this loop controlling expression is always 'true'.*	3357
2991	*The value of this 'if' controlling expression is always 'true'.*	3358
2992	*The value of this 'if' controlling expression is always 'false'.*	3359
2993	*The value of this 'do - while' loop controlling expression is always 'false'. The loop will only be executed once.*	3360
2994	*The value of this 'while' or 'for' loop controlling expression is always 'false'. The loop will not be entered.*	3359
2995	*The result of this logical operation is always 'true'.*	3355
2996	*The result of this logical operation is always 'false'.*	3356

7. APPENDIX B: MESSAGES TERMINATING DATAFLOW ANALYSIS

Msg No.	Message Text
321	*Declaration within 'for' statement defines an identifier '%s' which is not an object.*
322	*Illegal storage class specifier used in 'for' statement declaration.*
422	*Function call contains fewer arguments than prototype specifies.*
426	*Called function has incomplete return type.*
429	*Function argument is not of arithmetic type.*
430	*Function argument is not of compatible 'struct'/'union' type.*
435	*The 'struct'/'union' member '%s' does not exist.*
451	*Subscripting requires a pointer (or array lvalue).*
453	*An array subscript must have integral type.*
456	*This expression does not have an address - '&' may only be applied to an lvalue or a function designator.*
457	*The address-of operator '&' cannot be applied to a bit-field.*
468	*Unary '-' requires arithmetic operand.*
469	*Bitwise not '~' requires integral operand.*
482	*Expressions may only be cast to 'void' or scalar types.*
485	*Only integral expressions may be added to pointers.*
486	*Only integral expressions and compatible pointers may be subtracted from pointers.*
493	*Type of left operand is not compatible with this operator.*
494	*Type of right operand is not compatible with this operator.*
514	*Relational operator used to compare a pointer with an incompatible operand.*
515	*Equality operator used to compare a pointer with an incompatible operand.*
536	*First operand of '&&', '||' or '?' must have scalar (arithmetic or pointer) type.*
537	*Second operand of '&&' or '||' must have scalar (arithmetic or pointer) type.*
540	*2nd and 3rd operands of conditional operator '?' must have compatible types.*
542	*Controlling expression must have scalar (arithmetic or pointer) type.*
555	*Invalid assignment to object of void type or array type.*
556	*Left operand of assignment must be a modifiable object.*
557	*Right operand of assignment is not of arithmetic type.*
560	*Left operand of '<<=', '>>=', '&=', '|=', '^=' or '%=' must have integral type.*
561	*Right operand of assignment is not of compatible 'struct'/'union' type.*
564	*Left operand of assignment must be an lvalue (it must designate an object).*
565	*Left operand of '+=' or '-=' must be of arithmetic or pointer to object type.*
621	*The struct/union '%s' cannot be initialized because it has unknown size.*
640	*'%s' in 'struct' or 'union' type may not have 'void' type.*
643	*'%s' in 'struct' or 'union' type may not be a 'struct' or 'union' with unknown content.*
645	*A zero width bit-field cannot be given a name.*

Msg No.	Message Text
658	*Parameter cannot have 'void' type.*
671	*Initializer for object of arithmetic type is not of arithmetic type.*
682	*Initializer for object of a character type is a string literal.*
755	*'return' expression is not of arithmetic type.*
756	*'return' expression is not of compatible 'struct'/'union' type.*
1048	*Default argument values are missing for some parameters in this function declaration. This is not allowed.*
1333	*Type or number of arguments doesn't match function definition found later.*
3319	*Function called with number of arguments which differs from number of parameters in definition.*