

Crystal FLOW

for Better Results in Less Time

Do you fully understand the code you are working on ?

Answer: If you understood the code **fully**, then there would be no bugs in it.

By increasing your **understanding** of the code,
you can increase the **quality** and **correctness** of your code.

To have *a better understanding* of the code in less time - *Use the right tool for the right task*:

For example:

1. To review a function and the functions called by it, view its **CallFlow**.
2. To find why a data object contains an incorrect value, examine its **Data-dependency tree**.
3. To understand a function quickly, view its **Flowchart**.
4. To examine how a data object is used in the project, view its **DataFlow**.
5. To view the calling relationship of functions within a file, view the **File's function tree**.

and so on...

Crystal FLOW

for Better Results in Less Time

By using the right tool for the right task,

you can have a better understanding of the code in less time.

It will increase the **quality** and **correctness** of your code.

Suppose you are making a **design change** that affects a particular structure member
and you need to examine **how that structure member is used** in the project.

Use the right tool: View the DataFlow of the structure member.

- ◆ **The DataFlow** shows:
 - all the places in the project where that structure member is used
 - and the control flow surrounding those places
- ◆ Its graphical view gives you a **clear understanding** of how the data is used **in much less time**



Crystal FLOW

for Better Results in Less Time

Design changes have better quality and correctness when you use Crystal FLOW.

- Before the design change, you can [analyze](#) the existing code and affected data [in less time](#).
- After making the design change, [check for correctness](#) with DataFlow, Flowcharts etc.

Team discussions are quick and more productive with Flowcharts, Rich Trees, DataFlow, etc.

When you [inherit](#) legacy code, Crystal FLOW will save you a lot of time.

Code Reviews are [quick and productive](#) with HTML documentation containing Flowcharts etc.

The graphical views of Flowcharts, DataFlow, CallFlow enable you to [think at a higher level](#).

Everyday, you spend many hours in browsing, understanding and editing code.

With highly effective tools such as: [Flowcharts](#), [Rich Trees](#), [DataFlow](#), [CallFlow](#)

[Data-dependency trees](#), [Premium browsing](#), etc.

You can save more than 1 hour every day!

Crystal FLOW

Making Better Design Changes in Less Time

◆ **Before** making changes,

Crystal FLOW will help you analyze the existing code and affected data objects in less time.

A few examples:

1. **How a global or any variable is used** in the project (Use DataFlow)
2. **What functions are callers of** a given function and
what functions are called by a given function. (Premium Browsing)
3. **What parameters are being passed** in the
call-sequence up to the current function (Use Rich Tree)

◆ **After** making code modifications:

- **Check the modified code for correctness** (Use Flowchart, DataFlow)

(every problem you detect will save much time later)

1. Use DataFlow to examine how a global variable is used in the project

The screenshot displays the Crystal REVS for C IDE with the project 'PROJECT : Demo' and file 'Jsregexp.c' open. The 'DataFlow' window is active, showing a 'DataFlow Overview' for the global variable 'reopsize'. The overview diagram illustrates the flow of data from the variable's definition to its usage in three functions: 'EmitRegExp()', 'MatchRegExp()', and 'OptimizeRegExp()'. The 'OptimizeRegExp()' function is highlighted with a red box. The 'DataFlow Overview' window also includes a text box stating: 'Overview of where the global "reopsize" is used.'

The 'Js.c' file shows the definition of 'reopsize' as a global array of type 'uint8' with a size of 256. The array is initialized with values for various character classes, including 'EMPTY', 'ALT', 'BOL', 'EOL', 'WBDY', 'WNONBDY', 'QUANT', 'STAR', 'PLUS', 'OPT', 'LPAREN', 'RPAREN', 'DOT', 'CCLASS', 'DIGIT', 'NONDIGIT', 'ALNUM', 'NONALNUM', 'SPACE', and 'NONSPACE'.

```
84 REOP_UCCLASS = 29,
85 REOP_NUCLASS = 30,
86 REOP_BACKREF = 31,
87 REOP_FLAT = 32,
88 REOP_FLAT = 33,
89 REOP_UCFLAT = 34,
90 REOP_UCFLAT = 35,
91 REOP_ANCHOR1 = 36,
92 REOP_NCLASS = 37,
93 REOP_END
94 } REOP;
95
96 #define CCLASS_CHARSET_SIZE 256
97
98 uint8 reopsize[] = {
99     /* EMPTY */ 1,
100     /* ALT */ 3,
101     /* BOL */ 1,
102     /* EOL */ 1,
103     /* WBDY */ 1,
104     /* WNONBDY */ 1,
105     /* QUANT */ 7,
106     /* STAR */ 1,
107     /* PLUS */ 1,
108     /* OPT */ 1,
109     /* LPAREN */ 3,
110     /* RPAREN */ 3,
111     /* DOT */ 1,
112     /* CCLASS */ 1 + (CCLASS_CHARSET_SIZE - 1),
113     /* DIGIT */ 1,
114     /* NONDIGIT */ 1,
115     /* ALNUM */ 1,
116     /* NONALNUM */ 1,
117     /* SPACE */ 1,
118     /* NONSPACE */ 1,
```

Here the DataFlow Overview shows: the global variable **reopsize** is used in three functions - namely, **EmitRegExp()**, **MatchRegExp()**, and **OptimizeRegExp()**

Double-click to expand "OptimizeRegExp()":

1. The DataFlow expansion shows: **how the global variable reopsize is used in OptimizeRegExp()**.

The screenshot shows the Crystal REVS for C IDE with the project "Demo" and file "Jsregexp.c". The main editor displays the source code of the function `OptimizeRegExp`. The code includes comments and logic for handling character sets and reopsize. A callout bubble points to the `reopsize` variable in the code, stating "How 'reopsize' is used".

The DataFlow - Code Only window shows a flowchart of the function's execution. The flowchart starts with a call to `OptimizeRegExp`, followed by a tracking block for `reopsize`. The flowchart then branches into a `case REOP_CCLASS:` block, which contains a decision diamond for `NO` and `YES`. The `YES` branch leads to a block where `state->progLength` is updated using `reopsize`. The `NO` branch leads to a block where `state->progLength` is updated using `reopsize` and `size`.

You can see the control flow that affects the use of "reopsize".

You get a better understanding of how data is used - and you save time.

2. View the Function Properties card to view the **Caller functions** and **Called functions**

Crystal REVS for C [PROJECT : Demo] - [Jsobj.c *]

File Edit View Tree FlowChart DataFlow CallFlow Insert Selected text Goto Project Tools BatchOutput Window Help Tutorials

(Files) js_EnterSharp reopsize

js_EnterSharp

- Called By
 - Jsarray.c
 - array_join_sub
 - Jsobj.c
 - js_obj_toSource
- Parameters
- Functions Called
 - JS_CompareValues
 - JS_DestroyIdArray
 - JS_Enumerate
 - JS_free
 - JS_hash_object
 - JS_HashTableRawAdd
 - JS_HashTableRawLookup
 - JS_InflateString
 - JS_NewHashTable
 - JS_ReportOutOfMemory
 - JS_snprintf
 - MarkSharpObjects
- Globals used
- Struct/Unions used
- Local Declarations
- Nested declarations
- #defines used
- Labels used
- Enum constants used
- Unused identifiers
- Undeclared identifiers

Go to function-call

JS_NewHashTable

JS_EnterSharp

```

251 JSHashEntry *js_EnterSharp( JSContext *cx, JSObject *obj, JSIdArray **idap, jschar **sp )
252 {
253     JSSharpObjectMap *map;
254     JSHashTable *table;
255     JSIdArray *ida;
256     JSHashNumber hash;
257     JSHashEntry *he;
258     **hep;
259     jsatomid sharpid;
260     char buf[ 20 ];
261     size_t len;
262
263     map = &cx->sharpObjectMap;
264     table = map->table;
265     if( !table )
266     {
267         table = JS_NewHashTable( 8, js_hash_object, JS_CompareValues, JS_CompareValues, NULL, NULL );
268         if( !table )
269         {
270             JS_ReportOutOfMemory( cx );
271             return NULL;
272         }
273         map->table = table;
274     }
275
276     ida = NULL;
277     if( map->depth == 0 )
278     {
279         he = MarkSharpObjects( cx, obj, &ida );
280         if( !he )
281             return NULL;
282         JS_ASSERT(( ( jsatomid )he->value ) & SHARP_BIT == 0);
283         if( !idap )
284         {
285             JS_DestroyIdArray( cx, ida );
286         }

```

Easy navigation to Caller Functions, Called Functions

Current Function : js_EnterSharp

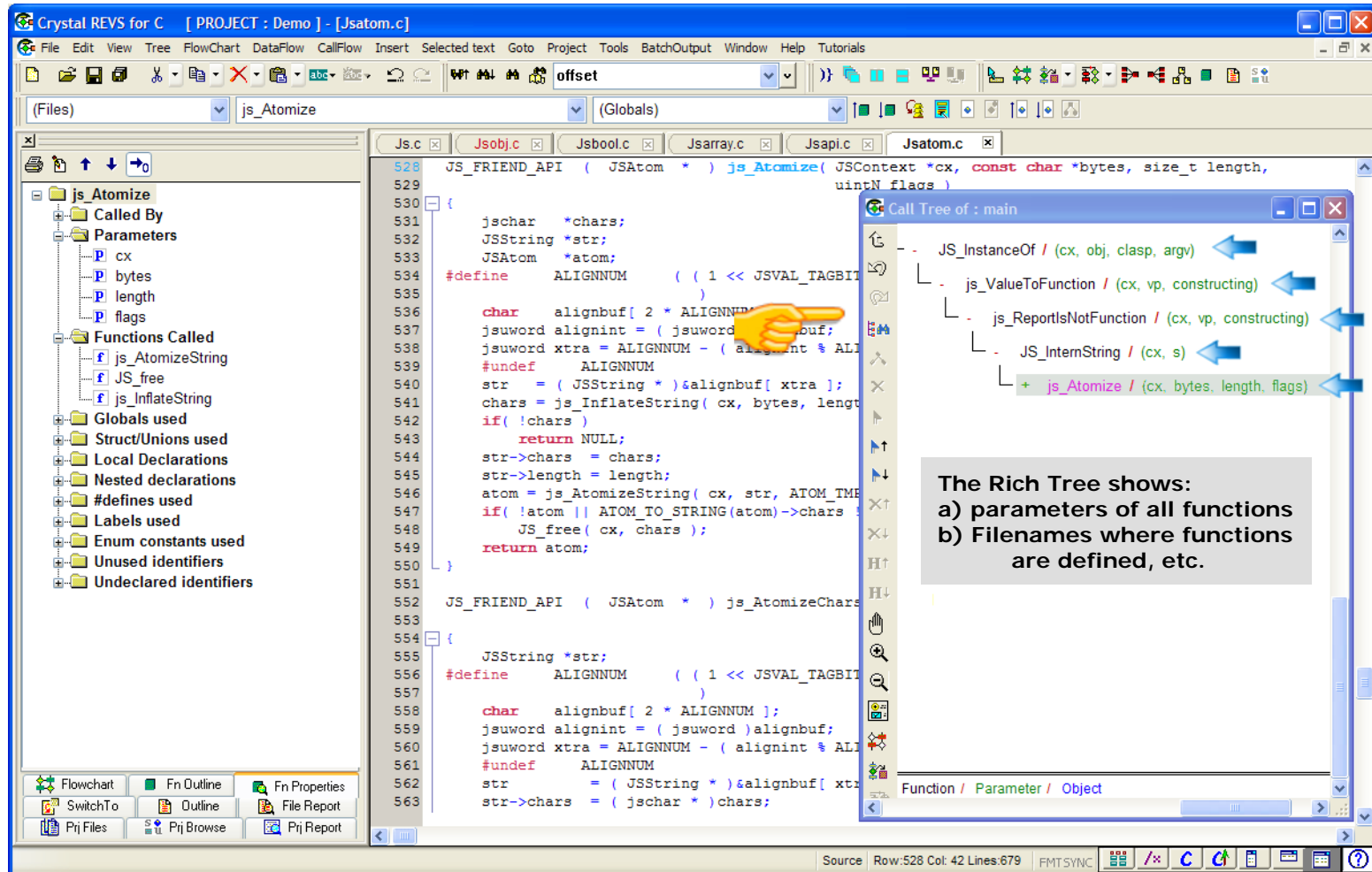
Source Row: 268 Col: 5 Lines:281

FMT SYNC

- With single-click, you can go to the function-calls in the code
- With a double-click, you can go to the function's definition

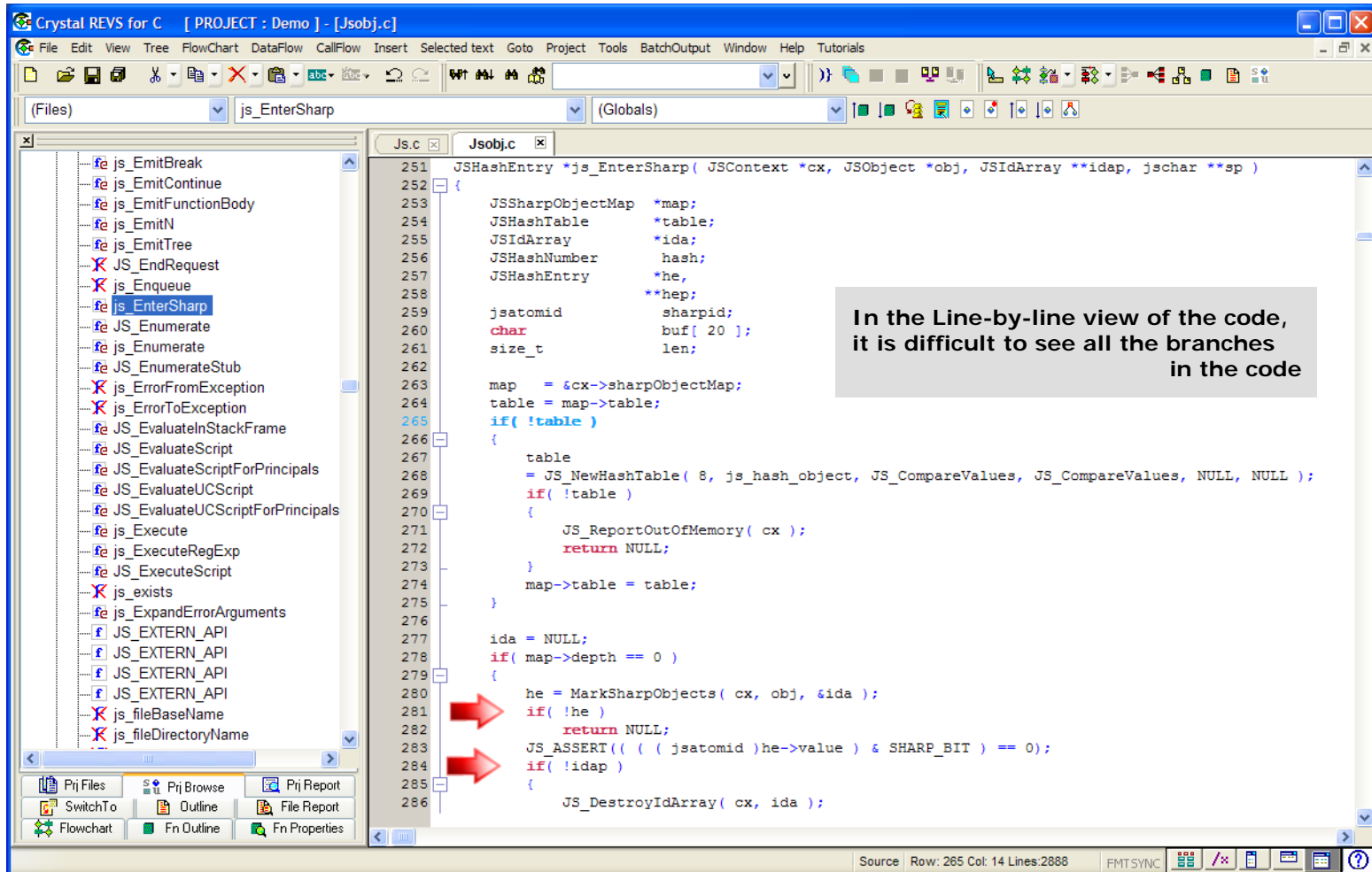
Use the Rich Tree:

View the parameters in the call-sequence



- You can view the parameters of all the functions in the call-sequence.
- You can view the filenames where the functions are defined

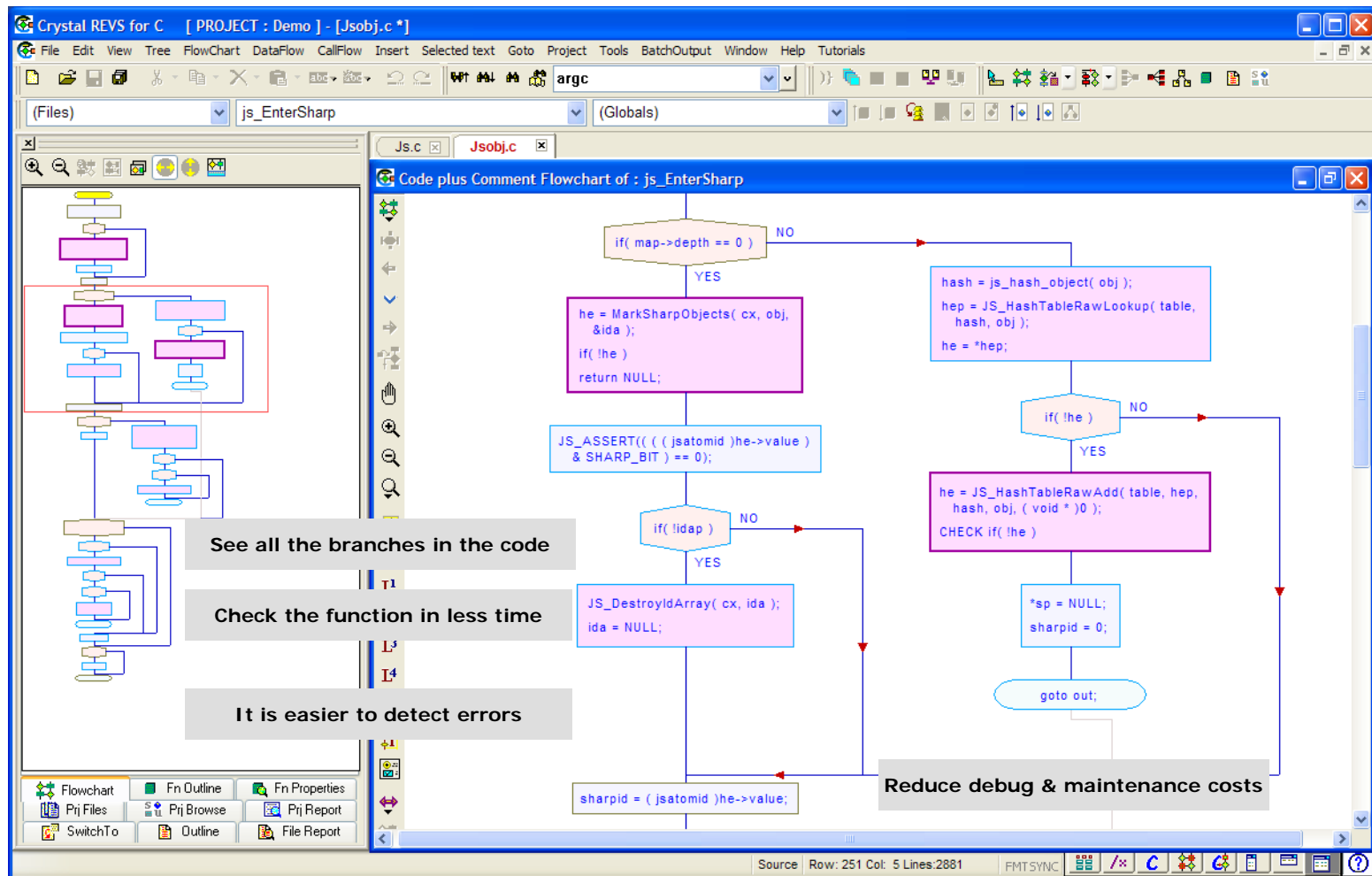
Check the modified code for correctness



The line-by-line view of code is not suitable for checking for correctness.

Here, it is difficult to see all the branches in the code. You can not see the whole function.

With Flowchart, it is easier to check the function for correctness



- You have the whole view of the function
- It is easier to detect errors
- The flowchart reduces debug and maintenance costs

In a Team Discussion,
Use Crystal Flow to make the discussion quick and productive

Suppose you are in a team discussion to solve a problem.

1) You are trying to figure out **why a data object has an incorrect value**

Use the **best tool** for the problem:

View the Data-dependency Tree of that data object

2) You are trying to examine **all paths that can reach a given point in a function**

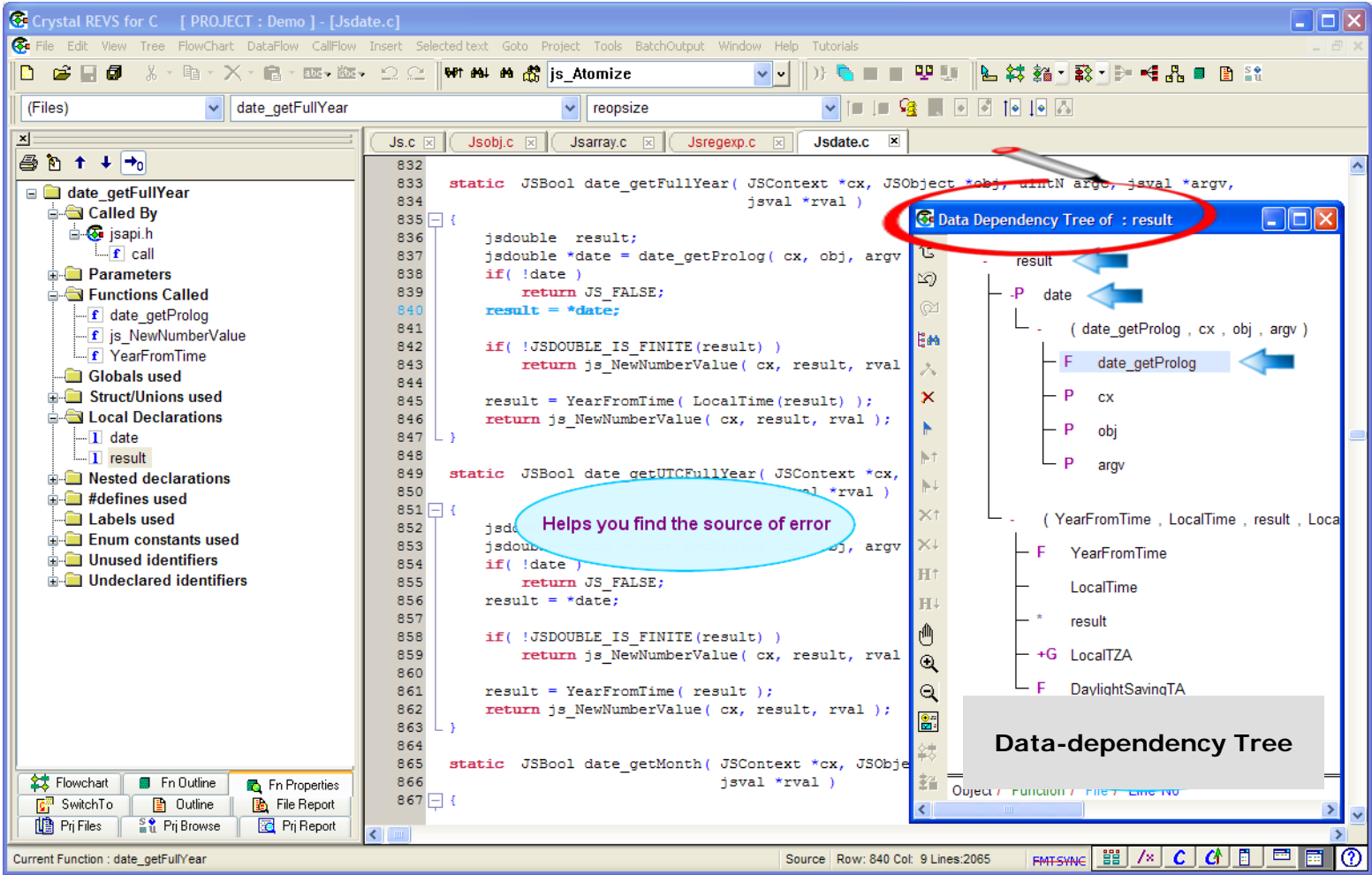
Use the **best tool** for the problem:

View the flowchart and highlight all paths to that point.

Question:

Which would you prefer for team discussion: **Graphical view** or **Line-by-line code?**

Data-dependency Tree: To find the source of incorrect data-value

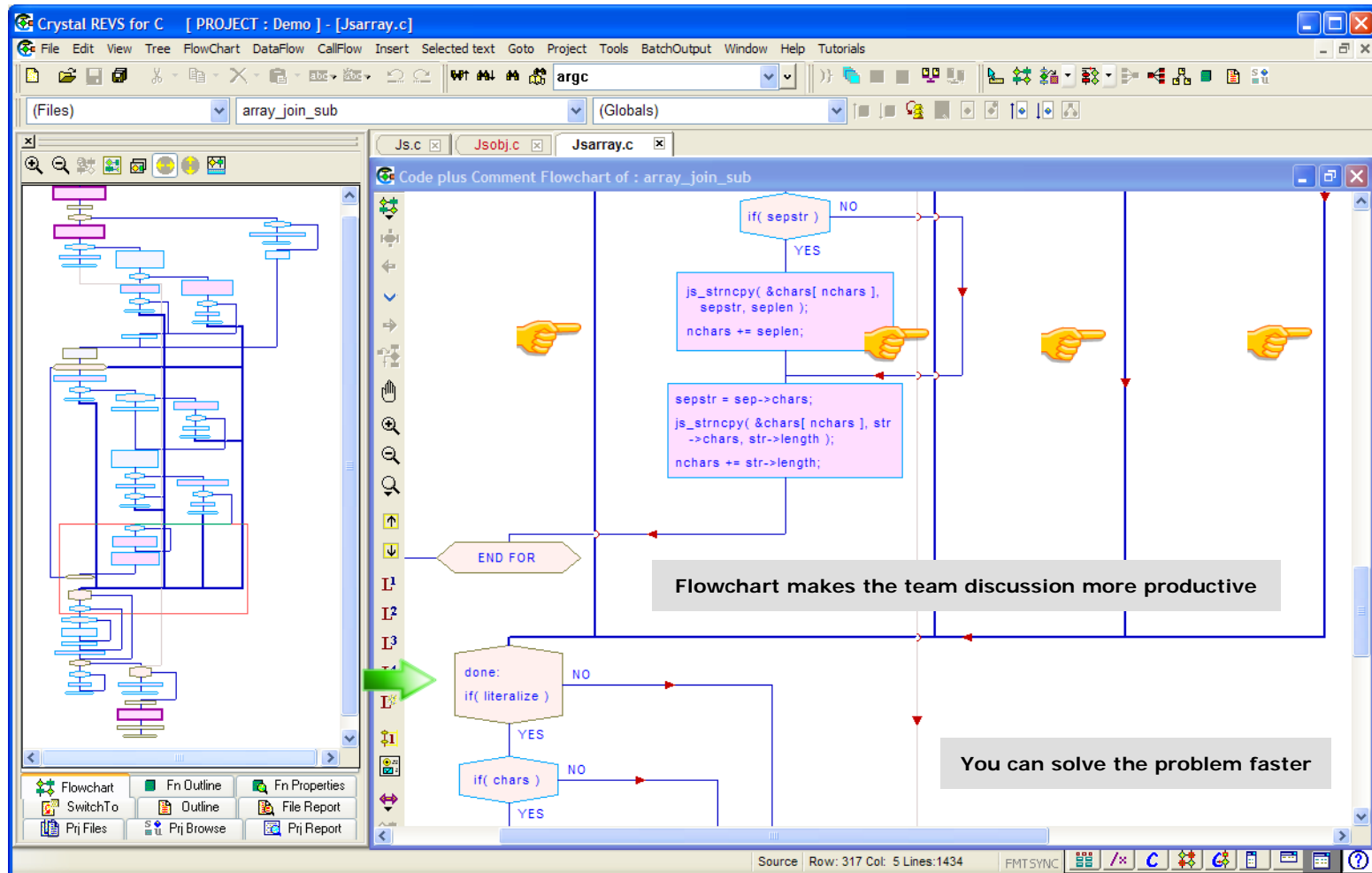


Here we see that the value of "result" **depends** on the value of "date"
which, in turn depends on the function "data_getProlog()"

The Data-dependency tree helps you go to the source of error.

During the discussion

How do you view **all the paths** that reach a given point in the function



- Use the flowchart to see all the paths that can reach the point of interest
- Flowcharts make the team discussion more effective
- You can find the best solution in less time

Explore and Analyze Code in Less Time

- ◆ Do you need to **get an overview of a function** and **the functions called by it** ?
 - **Use CallFlow** - You can expand function-calls and go deeper in the CallFlow

- ◆ **Improve readability:**
 - Crystal performs code formatting. **Makes the code easy to read**

Use CallFlow to make a fast attack on a function and the functions called by it

The screenshot displays the Crystal REVS for C interface. The main window shows the source code of the `script_toSource` function in `Jscript.c`. A callout box on the left says "Use CallFlow to make a fast attack on the functions". The `CallFlow - Code Only` window on the right shows the control flow graph for the function. A callout box on the right says "CallFlow of script_toSource". A red box highlights the `JS_DecompileScript` function call in the code and the corresponding node in the flow graph. A callout box at the bottom says "You can expand the function-calls to go deeper".

```
static JSBool script_toSource( JSContext *cx, JSObject *obj, uintN argc, jsval *argv, JSString *pval )
{
    JSString *script;
    size_t i, j, k, n;
    char buf[ 16 ];
    jschar *s, *t;
    uint32 indent;
    JSString *str;

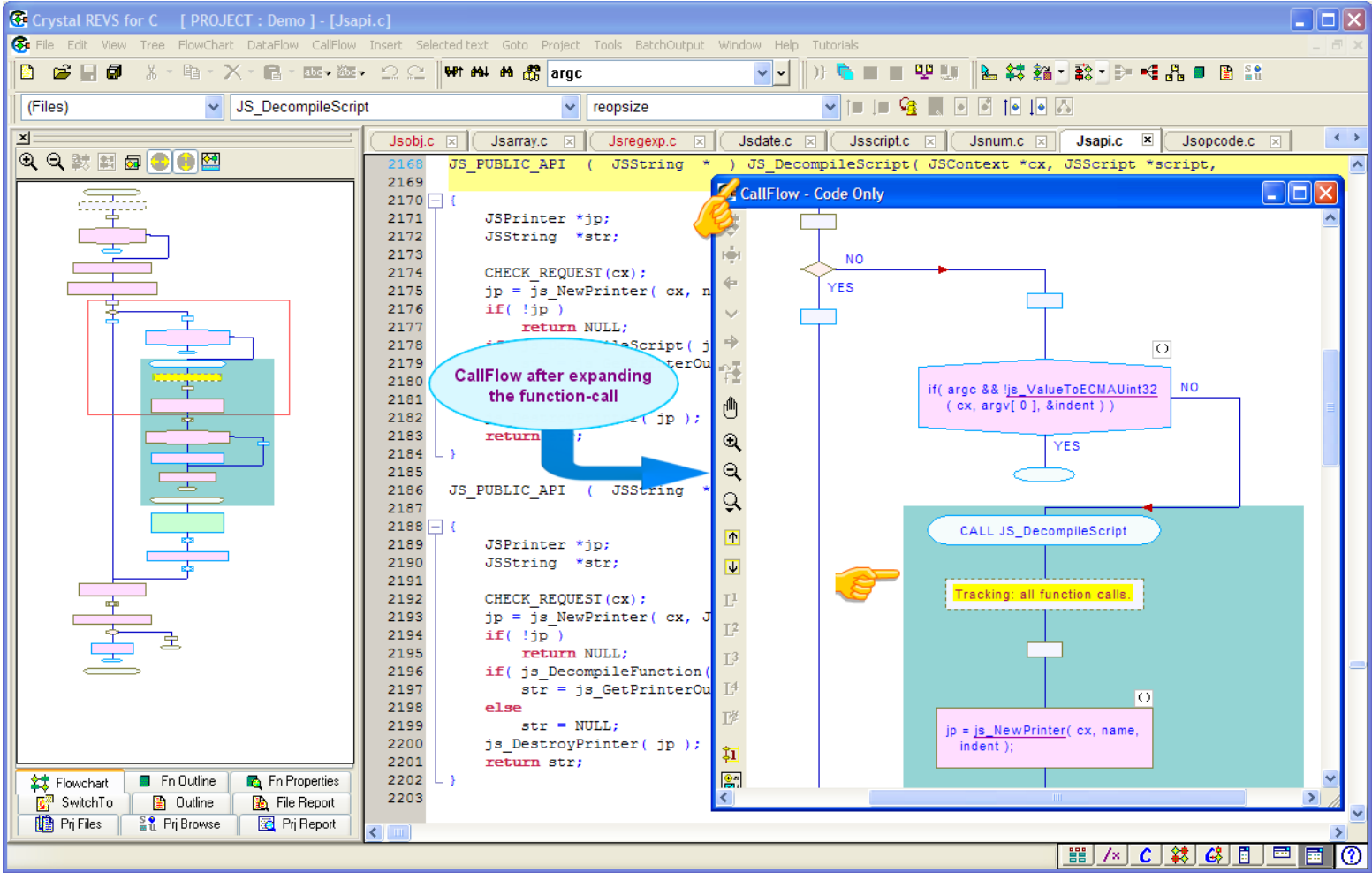
    if( !JS_InstanceOf( cx, obj, JS_SCRIPT_CLASS, &script ) )
        return JS_FALSE;
    script = JS_GetPrivate( cx, obj );

    /* Let n count the source */
    j = JS_snprintf( buf, sizeof buf, "(new %s(", js_ScriptClass.name );
    n = j + 2;
    if( !script )
    {
        /* Let k count the comment */
        k = 0;
        s = NULL;
    }
    else
    {
        indent = 0;
        if( argc && !js_ValueToECMAUint32( cx, argv[ 0 ], &indent ) )
            return JS_FALSE;
        str = JS_DecompileScript( cx, script, "Script.prototype.toSource", ( uintN ) indent );
        if( !str )
            return JS_FALSE;
        str = js_QuoteString( str );
    }
    return JS_TRUE;
}
```

Here, you see the CallFlow of the function `script_toSource()`.

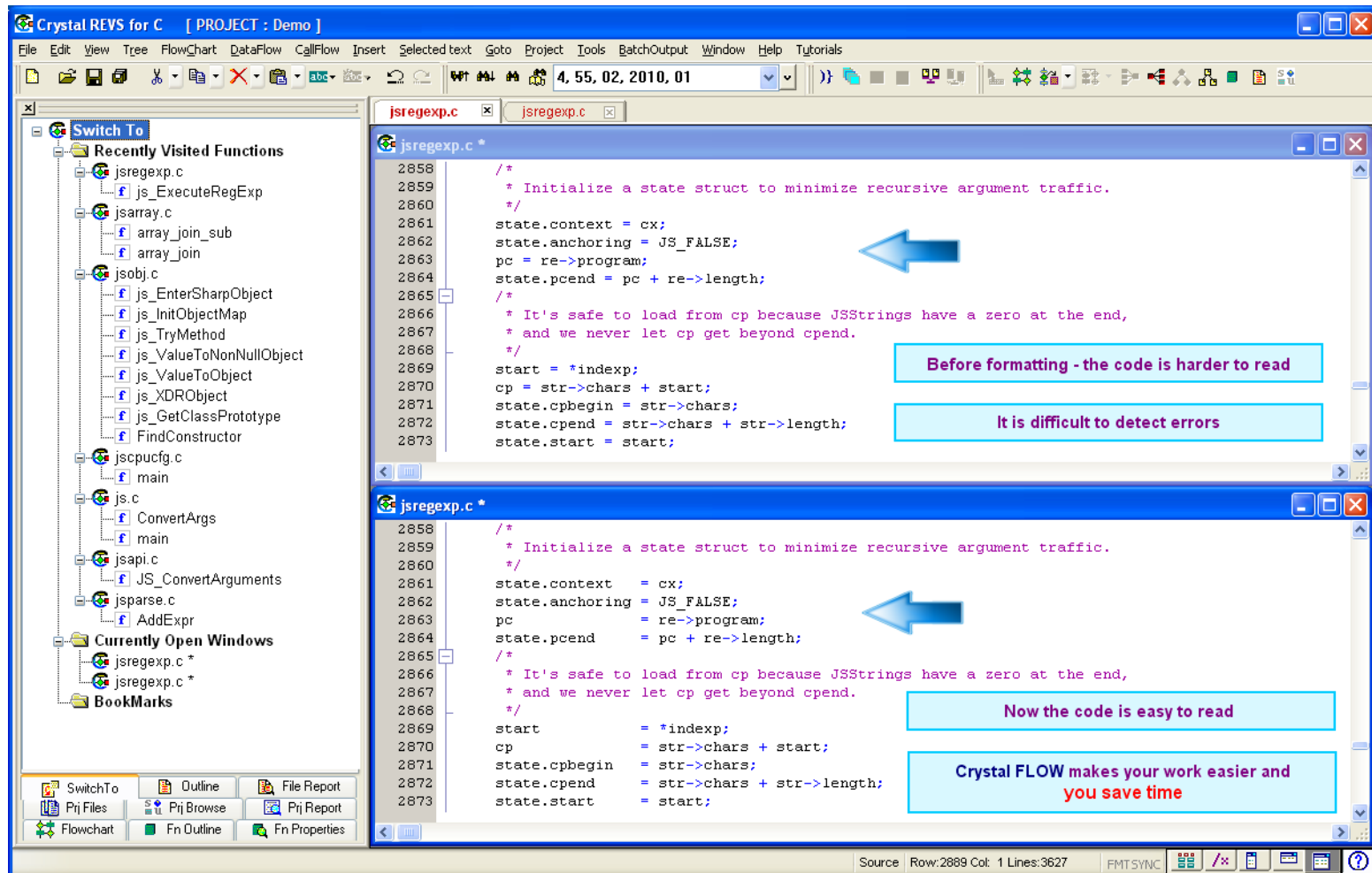
It shows the function-calls and the control flow around the function-calls.

Expand CallFlow to go deeper along the function-calls



- Navigate easily among a set of related function
- You can quickly analyze a function and the functions called by it

Crystal performs code formatting and **makes code and comments easy to read**



When you use Crystal, **you save time** at every step of your task.

Use the best tools.

Do a better quality design in less time.

Isn't it the formula to get ahead and stay ahead?

In the global marketplace,

If you lose 1 hour everyday, can you keep up with the competition?

***Get Crystal FLOW.* Be ahead of the competition.**

With Crystal Flow, you can save more than 1 hour every day!

Use Crystal Flow. Increase individual productivity. Increase team productivity...