On Testing of Complex Hydrographic Data Processing Algorithms

B.R. Calder and M.D. Plumlee

***Abstract*—Modern hydrography relies, more and more, on complex algorithms to resolve the soundings generated by remote sensing modalities, and to process those soundings into chartable products. Algorithm quality assurance is therefore critical to the integrity of the hydrographic effort.**

At best, the algorithms used might be described in a published paper, or possibly be available as a research code-base. More often, however, they are integrated deep in proprietary code and cannot be tested or verified without great effort. For algorithms transitioned from research in one organization to operations in another, there is no guarantee that the algorithm implemented is the algorithm that was designed. And each new software release demands effectively *ab initio* testing effort.

Using the CHRT algorithm as an example, a testing structure is proposed. The structure consists of an XML-based definition which packages the required data with the desired tests, and allows for exact or approximate matching of results, with control over tolerances. The XML-based output provides hierarchically aggregated summaries of test success to assist in reporting with level-of-detail control. Although intended for end user testing, the structure has obvious benefits for developers, and acts, effectively, as the algorithm's definition: any implementation that passes the conformance suite.

***Index Terms*—Algorithm Reliability, Testability, XML, CHRT**

I. Introduction

FOR at least the last decade, hydrographic practice has become more and more reliant on complex algorithms that are used to prepare the data for processing (e.g., motion sensor trajectory estimation), assist with the data inspection and quality control (e.g., depth estimation algorithms), or to assist in chart product construction (e.g., automated contouring and sounding selection). Given the safety requirement and liability concerns that are attendant on hydrographic practice, making sure that these algorithms operate appropriately is of utmost import. Unfortunately, although these algorithms may have been tested to some extent when they were being developed in a research environment or during implementation, it is often difficult or impossible for the end user to be sure that the software that they are delivered does what the original inventor or developer intended. And with each new software release each end user (or their organization) must individually organize a repeated testing effort. This can be inefficient since they cannot readily learn from each other, so the process can be time consuming, resource hungry, or both.

In part this problem is a function of the complexity of the algorithms used, but in part it is also engendered by the closed-source ecosystem of hydrographic software. Although open-source hydrographic software exists (notably the venerable, but still actively developed, MBSystem[1] [1], although HydrOffice[2] provides smaller tools for hydrographic practice [2], [3]) and can be very powerful in tackling obscure and rare problems, it is rarely used for production hydrographic practice. Similarly, although it might be strongly encouraged, it is unlikely that an end user or user group demanding that all algorithms be published openly will meet with much success in the current environment. For better or worse, most hydrographic data processing algorithms are going to remain closed-source for the foreseeable future.

Without the visibility of an open-source code-base, end users must essentially take on trust that the algorithm is operating correctly. While most, if not all, software vendors conduct some level of testing of their source code during development, this testing is very rarely demonstrated, reported, or disclosed to the end user unless under restrictive agreements with very large customers. In the special case of algorithm developers that provide source for inclusion in a commercial product, there is also very limited ability to probe, test, and diagnose issues with the implementation after incorporation. Any ability to guarantee correctness of the "as implemented" algorithm is thereby compromised.

In addition, even when testing is done there is no guarantee that all conditions of interest to a particular end user are tested. For example, it might be time prohibitive to test with a very large dataset, which means that bugs that only manifest with large data volumes will be missed—which bugs are particularly troublesome to surveys at plausibly useful scale. Similarly, end users with more obscure sensors, or different base working environments might find that their situation is not in the software vendor's testing scheme, leading to discrepancies in behavior.

For the end user, there is a need to rapidly and easily test algorithms in a standard manner, against agreed datasets and test results, in the field. That is, that the software vendor builds in the means for the user to run standard tests outwith a GUI environment, with automated methods to capture, process, sort, and report the results, verifying correct completion, or assisting in diagnosing the location of any fault.

Such a method is proposed here. Based on the CHRT algorithm [4], a method is outlined that takes an XML description of the tests to be conducted, runs the algorithm under test through a standard set of steps, and then compares the output against the results from a reference system. The tests are structured hierarchically so that groups of tests can have their results

B.R. Calder and M.D. Plumlee are with the Center for Coastal and Ocean Mapping and NOAA-UNH Joint Hydrographic Center, University of New Hampshire, Durham, NH 03824, U.S.A. (Contact: `brc@ccom.unh.edu`, +1 603 862 0526)

[1]http://www.ldeo.columbia.edu/res/pi/MB-System
[2]http://www.hydroffice.org

aggregated, which allows detected faults to be rolled-up to the highest level, providing an overview of the results while still allowing the interested user to drill down for diagnostic purposes. The algorithms used are by design suitable for large datasets, and allow the results to be collapsed into a well-controlled summary list when many faults appear, resulting in reasonable behavior when there are numerous faults observed. The test comparison algorithm also allows the test designer to require exact matching of results between the implementation under test and the reference implementation, or to allow for small variations; warnings are issued for small differences, and faults for those over the defined threshold. The result is a scheme that allows the end user to confirm that the algorithm as implemented is equivalent to the algorithm as intended.

This paper outlines the requirements for this method, and then provides a specific example with CHRT, illustrating the benefits of the proposed method with a specific fault discovered during the development of the conformance test suite. Some implications of adopting the proposed method are then explored, in particular what it means to be a "conforming implementation" under this regime.

II. Requirements for Testable Algorithms

In order for an algorithm to be testable in the sense described above, several conditions must exist; there are also some requirements that support efficiency in testing. First and foremost, the algorithm developer and software implementor must agree that testability is a goal. It would be very difficult to extend any test regime to the end user without support within the application, for example by arranging the code so that the algorithm can be driven directly (i.e., can be scripted) rather than through a GUI interface, and that all of its inputs can be exercised. Similarly, the software vendor must make available the results of any processing in a useable format, for example by writing the outputs in a standard format, even if that is not the format normally used. In this regard, *de facto* standards such as BAG[3] files [5] (or official standards such as S-100 [7], or S-102 [8]) are to be preferred, but any well-documented open file format could be used. Since this requirement necessitates commitment of resources and dictates part of the software architecture, its significance should not be underestimated.

The algorithm designer must also provide for a testable design: for the implementation to exposure the algorithm's control interfaces, they must first be exposable. Ideally, the algorithm design would also provide for sub-modules to be tested separately, although it might be difficult to expose sub-components of the algorithm without requiring that the test environment prepare a great deal of context. Most fundamentally, however, there must be a reference implementation of the "algorithm as intended" to which the final implementation can be compared. How this is provisioned will depend on whether the algorithm is developed internally or by an external designer, and the particular form is not vital so long as it is capable of providing an example set of results for given data inputs, control parameters, and run script. Thus, a Python[4] or MATLAB[5] reference implementation would be a functional, if potentially inefficient, solution so long as the outputs can be captured and distributed. The algorithm designer's obligation is then to provide the test script, inputs, and expected outputs for the desired tests, appropriately version controlled if required so that conformance to a particular version of the algorithm can be assessed. Making these results available through a website or other electronic distribution medium is preferred, since it allows end users to access them and run their own verifications.

Simple provision of the test script, inputs, and outputs is necessary but not sufficient for a functioning test suite: there is a requirement for some software to run the tests, accumulate the results, and report them to the end user in usable fashion. Unit testing for software is a common industry practice, and many frameworks (both open source and commercial) exist. Tools such as `CppTest`[6], `GoogleTest`[7], and `Boost.Test`[8] for specific languages (in this case C++) are available, while some languages have "preferred" solutions such as `unittest`[9] for Python. In most cases, however, these are designed to provide detailed tests that are specific to developers and which rarely have meaning to end users. Consequently, such testing frameworks are not well suited to the end user-scale testing envisioned here.

For end users, building the software required to conduct the test would be difficult. It is possible to require that end users install a development environment, but doing so increases the perceived difficulty of conducting the tests and therefore lessens the likelihood that they will be conducted. A general requirement is therefore to provide pre-built tools to conduct the test, and to ensure that input and outputs are both human and machine readable. The scheme must also provide for flexible specification of tests, for example by allowing both strict checking of outputs (i.e., outputs that must be bit-wise identical in reference and test implementations) and approximate comparisons (i.e., outputs that may suffer from numerical noise, but which are required to be within some narrow range of difference between reference and test implementations).

A defining characteristic of tests conducted for hydrographic data is that they generally involve considerable data volumes. Many algorithms are therefore the same computations applied at many data points. Hydrographic data is also inherently geospatial, and a geospatially-aware testing environment allows for the test results to be summarized by area, leading to better insight into possible causes of faults when they occur. A useful end user test environment must therefore be able to provide spatial context in the results, and to cope with large numbers of near-identical output results without the reporting becoming too voluminous to be useful. It must also aggregate results in logical groups so that an overall summary can be

[3]http://www.opennavsurf.org

[4]https://www.python.org
[5]https://www.mathworks.com
[6]http://cpptest.sourceforge.net
[7]https://github.com/google/googletest
[8]http://www.boost.org/doc/libs/1_63_0/libs/test/doc/html/index.html
[9]https://docs.python.org/3.6/library/unittest.html

presented to the end user that gives pass/fail status, but still allows for more detailed analysis. This last, particularly, allows the end user to communicate observed faults to the developer, supporting diagnosis as well as detection.

Given these requirements, a separate, end user-distributable test environment is necessary.

III. The CHRT Conformance Test Suite

CHRT [4] is a computer-assisted depth estimation algorithm that is a successor to CUBE [9]. CHRT augments CUBE with large-scale data-adaptive variable resolution grids, and incorporates lessons learned from more than a decade of using CUBE in production environments.

A particular lesson from CUBE was the requirement for testability after it was incorporated into commercial software products. CUBE, first released to implementation in 2003, has been licensed by over a dozen hydrographic data processing software vendors. To spur adoption, the license for CUBE was intentionally generous: licensees gain full access to the source code without ongoing royalty costs, and can manipulate, incorporate, adapt, adopt, and derive products from it without restriction. While flexible, this model for technology transfer resulted in a number of different implementation styles from organizations that lightly wrapped the CUBE source code and provided it to their customers, to those who essentially re-implemented the algorithm fully.

A consequence of this model was that different implementations of the algorithm could, and did, provide different results over the same data. As well as being frustrating and time consuming for end users, this situation made it difficult to determine whether observed issues with a particular dataset were due to the algorithm itself, or the implementation. This made it difficult to diagnose problems, and to offer concrete advice on best practices.

To avoid this situation with CHRT, the algorithm was designed to allow for better testability and algorithm visibility after incorporation into other code, in particular through the use of a client-server design [10]. The licensing terms also ensure that vendors are restrained from labelling their implementation "CHRT" until and unless they are able to satisfy the testing requirements for a particular version. Conformance with the test suite does not guarantee that the algorithm is correct, but it does guarantee that the algorithm-as-implemented is the same as the algorithm-as-intended, at least to within the test coverage of the conformance suite. The CHRT Conformance Test Suite (CTS) was therefore designed to exercise the core elements of the algorithm while following the general design principles outlined in Section II. The basic outline of the process is shown in the flow-chart of Fig. 1.

In Fig. 1, the left-hand column represents tasks undertaken by the algorithm designer, while the right-hand column are tasks that are run automatically by the CTS when the end user starts a conformance test. When preparing the CTS, the algorithm designer specifies the tests to be run by providing a simple XML document, Fig. 2, which lists the tests required, and any modifications of their default behaviors. (The tests conducted and their relationship to the requirements outlined

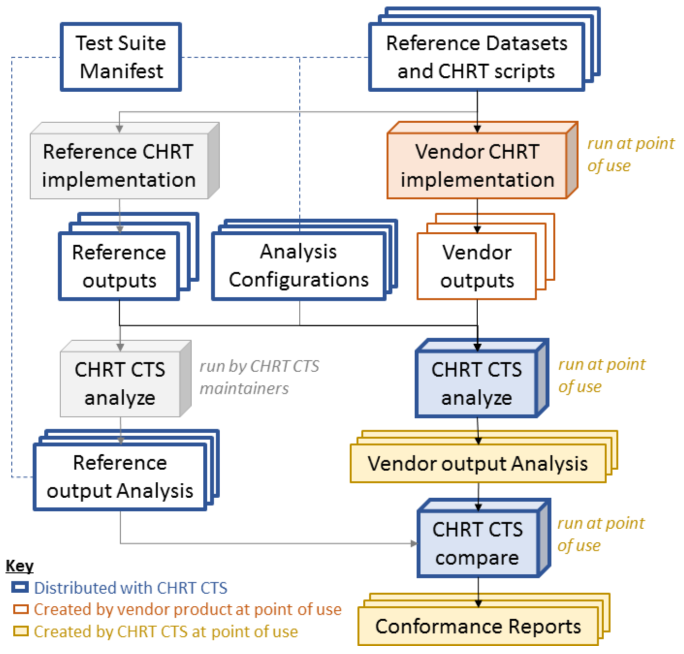

Test Suite Manifest
Reference Datasets and CHRT scripts
Reference CHRT implementation
Vendor CHRT implementation
run at point of use
Reference outputs
Analysis Configurations
Vendor outputs
CHRT CTS analyze
run by CHRT CTS maintainers
CHRT CTS analyze
run at point of use
Reference output Analysis
Vendor output Analysis
CHRT CTS compare
run at point of use
Key
Distributed with CHRT CTS
Created by vendor product at point of use
Created by CHRT CTS at point of use
Conformance Reports

Fig. 1. Flow-chart for the CHRT Conformance Test Suite. Analysis of the reference implementation is used to prepare the tests; the end user only runs analysis on the test implementation, and then compares against the outputs from the reference implementation.

```
<?xml version="1.0"?>
<CHRT_CTS>
  <AnalysisConfiguration>
    <Setup>
      <Testname>H11825</Testname>
      <Description>CHRT Conformance Test for small Pacific dataset H11825</Description>
      <Version>1.6.0</Version>
    </Setup>
    <Analysis>
      <Scope type="Global"/>
      <AnalysisComponents>
        <Statistics>
          <Origin reference="SW" x="?" y="?" units="m">
            <FieldTest field="x" tolerance_amt=".01"/>
            <FieldTest field="y" tolerance_amt=".01"/>
          </Origin>
        </Statistics>
        <Mask/>
        <DataDensity tolerance_amt="1.001e-5"/>
        <ResolutionEstimate tolerance_amt="1.001e-5"/>
        <AuxiliaryDetails/>
        <DepthDetails>
          <DepthEstimate tolerance_amt=".01"/>
          <DepthUncertainty tolerance_amt=".01"/>
        </DepthDetails>
      </AnalysisComponents>
    </Analysis>
  </AnalysisConfiguration>
</CHRT_CTS>
```

Fig. 2. Example analysis configuration XML file for a particular dataset in the CTS.

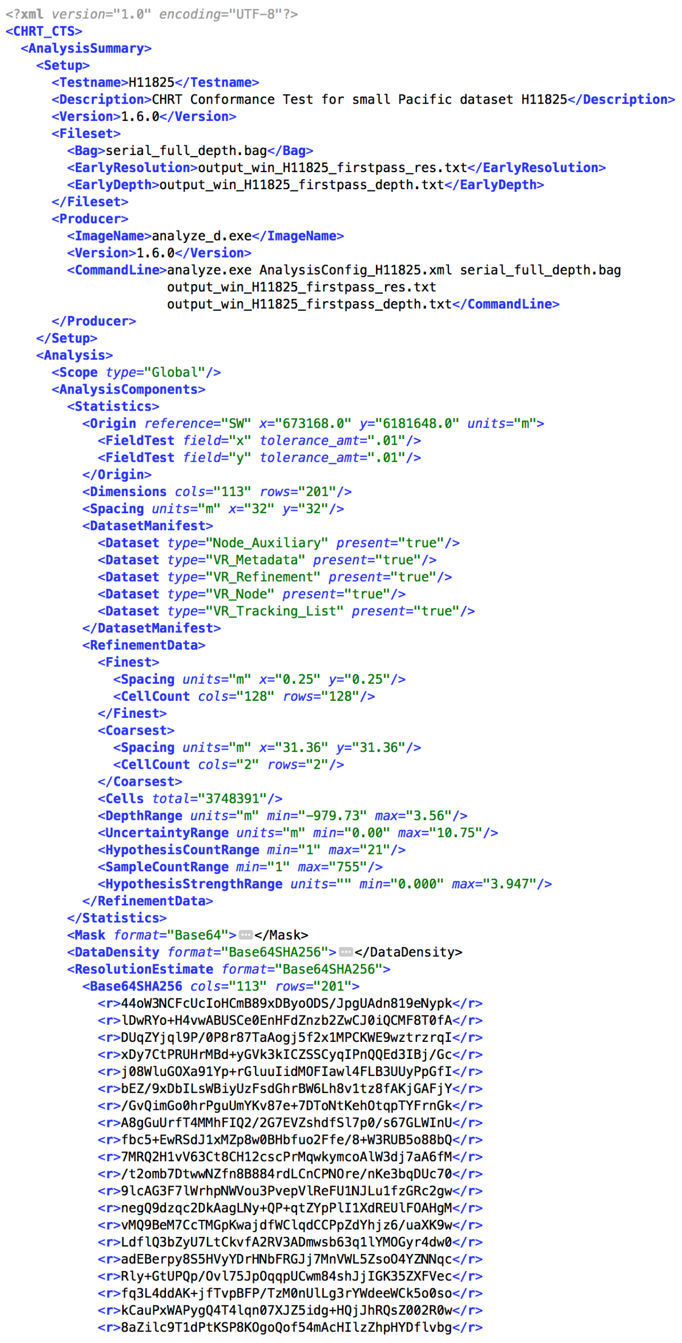
are considered following.) The reference implementation is then executed, and the outputs are sent through the analysis code to provide another XML file, Fig. 3, which specifies in detail the results of the various tests. This XML analysis file is then cached (this is the "Reference Output Analysis" in Fig. 1) and made available to all end users through the CTS as the standard to which all test implementations must demonstrate conformance.

For the end user, the CTS contains two executables (which are the same as those used by the CTS designer), one to analyze algorithm outputs according to the XML definition file and the other to compare two analysis XML files, the reference analysis file (e.g., Fig. 3) for the chosen test dataset(s), and the control files for the algorithm, which include a configuration file for CHRT, a CHRT-language command list to control the execution of the algorithm, an OS command script to run the test, and the source data. Given these, the software vendor must make it possible to run their version of the algorithm with the same inputs, generating another XML analysis file. The CTS code then compares the two results, generating a final XML document, Fig. 4, which details for each test specified whether the two analyses agree or disagree. As can be seen in Fig. 4, each entry has a binary "success" attribute indicating whether the overall test was successful (the test is considered successful if and only if all sub-tests were successful); sub-entries then provide further detail. As suggested in Section II, this allows the end user to verify a simple pass/fail for the test, but allows the developers to investigate further the details of any faults in order to diagnose the problem.

The analysis specification of Fig. 2 demonstrates a number of the outlined requirements for an end user testing system. To allow for consistency of testing, the `Setup` element provides simple versioning information, while the `AnalysisComponents` element specifies the types of tests that are to be conducted on the outputs of CHRT. Each sub-element specifies a separate set of tests, so that for example the `Statistics` test causes the code to compute basic statistics of the outputs such as the origin location, size of output, and spacing of the lowest-resolution cells in the output (in CHRT, these are known as "SuperCells"), and so on. These are very basic tests, but they are very easy to get wrong with, for example, bad georeferencing information, or a mistake in the implementation of CHRT's georeferencing model. Sub-elements can be simple, such as the `Mask` test, which simply indicates that the default testing is to be done. Alternatively, they can include modifications for the test to be done, as seen in the `DataDensity` test. This specification indicates that the data density computed in each SuperCell for the test implementation must match the reference implementation to within $10^{-5}\,\mathrm{m}^{-2}$ (essentially the floating point accuracy), but does not have to be exactly the same, as would be the default case. Similarly, the `Origin` test specifies that particular fields of the data element have to be tested separately, in this case requiring that the georeferencing location has to match to within 0.01 m. Tolerances and special conditions can be specified generally in all of the tests, allowing the algorithm designer the flexibility required to set up more nuanced test conditions.

The analysis of Fig. 3 shows both capture of new information and the ability of the algorithm to compactly represent large-scale data, as required. After the analysis code runs, information on what was done (`Fileset` and `Producer` elements) is gathered and preserved, allowing for better diagnostics; the statistics ordered in the analysis configuration of Fig. 2 are also present. For the locations of SuperCell activity (`Mask` element), estimated data density per SuperCell (`DataDensity` element), and computation resolution (`ResolutionEstimate` element), the algorithm computes a Base64[10] representation for each row of the respective 2D field in turn. In the case of the `Mask` element this is sufficiently small to simply store, but for the other fields, an SHA256 cryptographic hash [11] of the encoded data is generated, and then stored per row. This allows the algorithm to test for differences between implementations on a row-by-row basis, although in the case of the hashes it does not allow for identification of which column in the row differs if a fault is detected. Use of a hash limits the space required to store the results from a row of the grid to a fixed size, irrespective of the size of the grid.

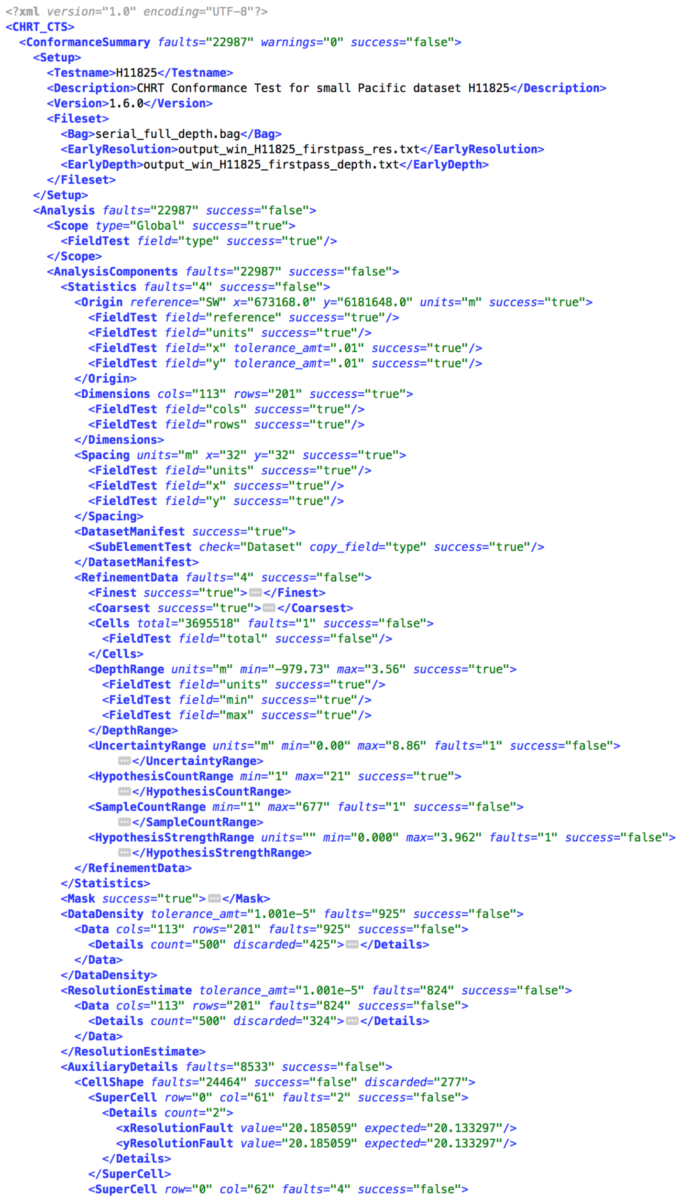
Finally, Fig. 4 illustrates how the results of the CTS are



```xml
<?xml version="1.0" encoding="UTF-8"?>
<CHRT_CTS>
  <AnalysisSummary>
    <Setup>
      <Testname>H11825</Testname>
      <Description>CHRT Conformance Test for small Pacific dataset H11825</Description>
      <Version>1.6.0</Version>
      <Fileset>
        <Bag>serial_full_depth.bag</Bag>
        <EarlyResolution>output_win_H11825_firstpass_res.txt</EarlyResolution>
        <EarlyDepth>output_win_H11825_firstpass_depth.txt</EarlyDepth>
      </Fileset>
      <Producer>
        <ImageName>analyze_d.exe</ImageName>
        <Version>1.6.0</Version>
        <CommandLine>analyze.exe AnalysisConfig_H11825.xml serial_full_depth.bag
                     output_win_H11825_firstpass_res.txt
                     output_win_H11825_firstpass_depth.txt</CommandLine>
      </Producer>
    </Setup>
    <Analysis>
      <Scope type="Global"/>
      <AnalysisComponents>
        <Statistics>
          <Origin reference="SW" x="673168.0" y="6181648.0" units="m">
            <FieldTest field="x" tolerance_amt=".01"/>
            <FieldTest field="y" tolerance_amt=".01"/>
          </Origin>
          <Dimensions cols="113" rows="201"/>
          <Spacing units="m" x="32" y="32"/>
          <DatasetManifest>
            <Dataset type="Node_Auxiliary" present="true"/>
            <Dataset type="VR_Metadata" present="true"/>
            <Dataset type="VR_Refinement" present="true"/>
            <Dataset type="VR_Node" present="true"/>
            <Dataset type="VR_Tracking_List" present="true"/>
          </DatasetManifest>
          <RefinementData>
            <Finest>
              <Spacing units="m" x="0.25" y="0.25"/>
              <CellCount cols="128" rows="128"/>
            </Finest>
            <Coarsest>
              <Spacing units="m" x="31.36" y="31.36"/>
              <CellCount cols="2" rows="2"/>
            </Coarsest>
            <Cells total="3748391"/>
            <DepthRange units="m" min="-979.73" max="3.56"/>
            <UncertaintyRange units="m" min="0.00" max="10.75"/>
            <HypothesisCountRange min="1" max="21"/>
            <SampleCountRange min="1" max="755"/>
            <HypothesisStrengthRange units="" min="0.000" max="3.947"/>
          </RefinementData>
        </Statistics>
        <Mask format="Base64">⋯</Mask>
        <DataDensity format="Base64SHA256">⋯</DataDensity>
        <ResolutionEstimate format="Base64SHA256">
          <Base64SHA256 cols="113" rows="201">
            <r>44oW3NCFcUcIoHCmB89xDByoODS/JpgUAdn819eNypk</r>
            <r>lDwRYo+H4vwABUSCe0EnHFdZnzb2ZwCJ0iQCMF8T0fA</r>
            <r>DUqZYjql9P/0P8r87TaAogj5f2x1MPCKWE9wztrzrqI</r>
            <r>xDy7CtPRUHrMBd+yGVk3kICZSSCyqIPnQQEd3IBj/Gc</r>
            <r>j08WluGOXa91Yp+rGluuIidMOFIawl4FLB3UUyPpGfI</r>
            <r>bEZ/9xDbILsWBiyUzFsdGhrBW6Lh8v1tz8fAKjGAFjY</r>
            <r>/GvQimGo0hrPguUmYKv87e+7DToNtKehOtqpTYFrnGk</r>
            <r>A8gGuUrfT4MMhFIQ2/2G7EVZshdfSl7p0/s67GLWInU</r>
            <r>fbc5+EwRSdJ1xMZp8w0BHbfuo2Ffe/8+W3RUB5o88bQ</r>
            <r>7MRQ2H1vV63Ct8CH12cscPrMqwkymcoAlW3dj7aA6fM</r>
            <r>/t2omb7DtwwNZfn8B884rdLCnCPNOre/nKe3bqDUc70</r>
            <r>9lcAG3F7lWrhpNWVou3PvepVlReFU1NJLu1fzGRc2gw</r>
            <r>negQ9dzqc2DkAagLNy+QP+qtZYpPlI1XdREUlFOAHgM</r>
            <r>vMQ9BeM7CcTMGpKwajdfWClqdCCPpZdYhjz6/uaXK9w</r>
            <r>LdflQ3bZyU7LtCkvfA2RV3ADmwsb63q1lYMOGyr4dw0</r>
            <r>adEBerpy8S5HVyYDrHNbFRGJj7MnVWL5ZsoO4YZNNqc</r>
            <r>Rly+GtUPQp/Ovl75JpOqqpUCwm84shJjIGK35ZXFVec</r>
            <r>fq3L4ddAK+jfTvpBFP/TzM0nUlLg3rYWdeeWCk5o0so</r>
            <r>kCauPxWAPygQ4T4lqn07XJZ5idg+HQjJhRQsZ002R0w</r>
            <r>8aZilc9T1dPtKSP8KOgoQof54mAcHIlzZhpHYDflvbg</r>
```

Fig. 3. Example (partial) analysis results from the CHRT reference implementation for the configuration in Figure 2. Note that many of the entries have been collapsed, shown as a horizontal grey box with white ellipses, hiding much of the detail here in order to show more of the entries.

[10]https://tools.ietf.org/html/rfc4648



```xml
<?xml version="1.0" encoding="UTF-8"?>
<CHRT_CTS>
  <ConformanceSummary faults="22987" warnings="0" success="false">
    <Setup>
      <Testname>H11825</Testname>
      <Description>CHRT Conformance Test for small Pacific dataset H11825</Description>
      <Version>1.6.0</Version>
      <Fileset>
        <Bag>serial_full_depth.bag</Bag>
        <EarlyResolution>output_win_H11825_firstpass_res.txt</EarlyResolution>
        <EarlyDepth>output_win_H11825_firstpass_depth.txt</EarlyDepth>
      </Fileset>
    </Setup>
    <Analysis faults="22987" success="false">
      <Scope type="Global" success="true">
        <FieldTest field="type" success="true"/>
      </Scope>
      <AnalysisComponents faults="22987" success="false">
        <Statistics faults="4" success="false">
          <Origin reference="SW" x="673168.0" y="6181648.0" units="m" success="true">
            <FieldTest field="reference" success="true"/>
            <FieldTest field="units" success="true"/>
            <FieldTest field="x" tolerance_amt=".01" success="true"/>
            <FieldTest field="y" tolerance_amt=".01" success="true"/>
          </Origin>
          <Dimensions cols="113" rows="201" success="true">
            <FieldTest field="cols" success="true"/>
            <FieldTest field="rows" success="true"/>
          </Dimensions>
          <Spacing units="m" x="32" y="32" success="true">
            <FieldTest field="units" success="true"/>
            <FieldTest field="x" success="true"/>
            <FieldTest field="y" success="true"/>
          </Spacing>
          <DatasetManifest success="true">
            <SubElementTest check="Dataset" copy_field="type" success="true"/>
          </DatasetManifest>
          <RefinementData faults="4" success="false">
            <Finest success="true">⋯</Finest>
            <Coarsest success="true">⋯</Coarsest>
            <Cells total="3695518" faults="1" success="false">
              <FieldTest field="total" success="false"/>
            </Cells>
            <DepthRange units="m" min="-979.73" max="3.56" success="true">
              <FieldTest field="units" success="true"/>
              <FieldTest field="min" success="true"/>
              <FieldTest field="max" success="true"/>
            </DepthRange>
            <UncertaintyRange units="m" min="0.00" max="8.86" faults="1" success="false">
                ⋯</UncertaintyRange>
            <HypothesisCountRange min="1" max="21" success="true">
                ⋯</HypothesisCountRange>
            <SampleCountRange min="1" max="677" faults="1" success="false">
                ⋯</SampleCountRange>
            <HypothesisStrengthRange units="" min="0.000" max="3.962" faults="1" success="false">
                ⋯</HypothesisStrengthRange>
          </RefinementData>
        </Statistics>
        <Mask success="true">⋯</Mask>
        <DataDensity tolerance_amt="1.001e-5" faults="925" success="false">
          <Data cols="113" rows="201" faults="925" success="false">
            <Details count="500" discarded="425">⋯</Details>
          </Data>
        </DataDensity>
        <ResolutionEstimate tolerance_amt="1.001e-5" faults="824" success="false">
          <Data cols="113" rows="201" faults="824" success="false">
            <Details count="500" discarded="324">⋯</Details>
          </Data>
        </ResolutionEstimate>
        <AuxiliaryDetails faults="8533" success="false">
          <CellShape faults="24464" success="false" discarded="277">
            <SuperCell row="0" col="61" faults="2" success="false">
              <Details count="2">
                <xResolutionFault value="20.185059" expected="20.133297"/>
                <yResolutionFault value="20.185059" expected="20.133297"/>
              </Details>
            </SuperCell>
            <SuperCell row="0" col="62" faults="4" success="false">
```

Fig. 4. Example (partial) comparison result from two implementations of CHRT for the configuration in Figure 2. Note that many of the entries have been collapsed, shown as a horizontal grey box with white ellipses, hiding much of the detail here in order to show more of the entries.

reported. First, consider the `ConformanceSummary` element at the top level. This provides the overall summary for the run, clearly indicating that the test failed, and the total number of faults to be addressed. This information is repeated from the `AnalysisComponents` element, which in turns aggregates the results of all of the sub-tests, as outlined previously. In this case, it is clear that the basic statistics for the test implementation match (e.g., the `Origin` is in the right place, and the `Dimensions`—the overall size of the output—are correct, etc.) However, there are a number of faults detected with the details of the refinements of the SuperCells: the total number of refinements is incorrect (`RefinementData/Cells` element), suggesting that the component of the algorithm that determines how to compute the refinements is not operating as expected. This is further bolstered by the faults detected

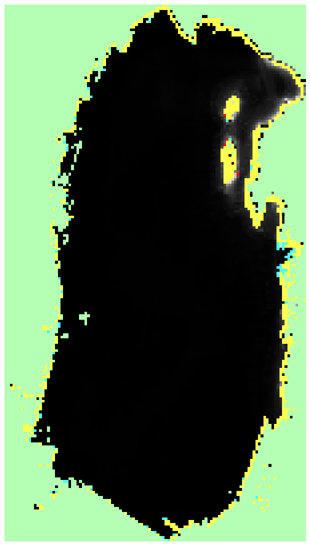


Fig. 5. Example of reported differences in computed data density between reference and test implementations of CHRT on macOS and Windows, respectively. Subtle differences in an allegedly "standard" system library cause small differences in data density computation in areas on the edge of the covered area where only noise data exists. Here, black cells indicate agreement on data density, while increasing levels of yellow indicate differences; green indicates "no information".

in the data density computed (`DataDensity` element). Note, however, that although `DataDensity` detects a total of 925 faults, only a (user defined) total of 500 are reported in detail (`DataDensity/Data/Details` element), and the remaining 425 are marked as "discarded", keeping the size of the file in check, as required.

The issue in this case was that the algorithm component that computes the data density within each SuperCell was not operating correctly in the test implementation due to subtle differences between different operating systems in an otherwise standard system library used by the code to accelerate computations. This can be seen, given some knowledge of the data, from the `CellShape` element. These can be seen starting at the bottom of the excerpt in Fig. 4, but are much more readily observed in Fig. 5 where the differences reported in this section have been displayed spatially, using the SuperCell references provided in the comparison XML document. Clearly, the majority of the problems occur at the edges of the area covered, or in the areas to the northeast, which are in fact islands where there is no other data. The issue here is overspill (noise) data outside of the planned survey area, where there is no good data to counteract it. In this case, CHRT does its best to report what is observed, but the behavior of the data precludes reliable estimates of data density. Fig. 4 readily illustrates the ability of testing schemes like this to find subtle differences in implementations, and for this type of reporting to allow for further investigation of the problem, highlighting clearly where, and therefore why, the problem occurs.

IV. Discussion

The examples shown here clearly demonstrate that it is relatively simple to provide for end user testing of even complex algorithms, although it does require commitment of some effort and resources on the part of both the algorithm designer and the software vendor. (In some cases, these might be different parts of the same organization, but that does not preclude using this method.) It is clear that even at the current level of algorithmic complexity in hydrographic practice, the ability to adequately test algorithms, and therefore provide some measure of confidence to the end user that the algorithm is as intended, is essential.

One potential concern with a scheme that tests only on the final outputs of a complex algorithm is the subtlety with which the method can identify issues in the test implementation, and therefore the precision with which these issues can be targeted. To the end user, this is irrelevant: the algorithm is either as intended, or it is not, and further subtlety is not required. To allow for debugging and development, however, this is a significant issue.

The experience with the CTS, although clearly not exploring all eventualities, seems to suggest that this may not be as significant a concern as might at first be thought. For example, although the estimation resolution predicted by CHRT is based directly on the data density estimated, by testing separately on the two outputs the CTS is able to distinguish between cases where the density is computed incorrectly and those where the translation to resolution failed, which is a subtle difference in small components of the algorithm. Similarly, the CTS counts and compares separately the number of soundings that are used to estimate the depth in the interior of each SuperCell and those on the edge between SuperCells. Although this is essentially the same comparison, separately assessing the interior estimation nodes allows the CTS to verify that the fundamental data propagation algorithms are working correctly, while comparisons of the edge estimation nodes allows for verification of the correct inter-SuperCell propagation of data. These are related, but separate components that appear at a different levels of CHRT (at least in the reference implementation). Again, this is a subtle difference that can be detected by careful control of the tests, along with some fundamental knowledge of the algorithm's implementation details. Problems in this area can be difficult to diagnose, since they may only appear in a very limited number of depth estimates, and might not be particularly evident even then, depending on the type of data.

An interesting consequence of testing in this fashion is that it might change the way that implementations of algorithms are judged. With a strong test suite, continuity of source code from reference to test implementation is not required for the test implementation to be considered "correct". In the strongest case of this paradigm, anything that meets the test suite is, *ipso facto*, a valid implementation of the algorithm. If this paradigm is adopted, however, then the design of the test suite is essential: if it does not test important behaviors of the algorithm, then they are not captured in the "definition" and variant implementations may differ significantly but still pass the test suite.

V. Conclusions

Assurances of the reliability of complex computer-assisted hydrography algorithms are very important to the end user of hydrographic software. At present, however, they are very difficult to obtain. In practice, it is difficult to test complex algorithms, particularly once they become part of a larger whole, unless there is cooperation between algorithm designer and software vendor to allow for testability. This includes designing in algorithm features to support this ideal, and implementation such that these features are maintained all the way to the end user.

The method proposed here highlights the importance of flexible human and machine readable specification for the tests required, along with the means to leverage the tests to support debugging of non-conformant implementations. In particular, the use of XML as a description medium allows for the tests specified, the analyses recorded, and the comparison against the reference implementation all to be structured in the same form. The experience with developing a test suite for CHRT has illustrated that even when simply testing the final outputs of the algorithm, relatively subtle tests on component parts of the algorithm are possible given some support from the algorithm design.

In the face of present and increasing complexity of the algorithms used to support safety of navigation hydrography, algorithm reliability and end user confidence in the same should be an urgent concern for the community.

Acknowledgements

The authors gratefully acknowledge the support of NOAA grant NA15NOS4000200.

References

[1] D. W. Caress and D. N. Chayes, "New software for processing sidescan data from sidescan-capable multibeam sonars," *Proc. IEEE*, vol. 2, pp. 997–1000, 1995.

[2] M. Wilson, G. Masetti, and B. R. Calder, "NOAA QC Tools: Origin, development, and future," in *Proc. Canadian Hydro. Conf. 2016*. Halifax, Nova Scotia: Canadian Hydro. Soc., Ottawa, Canada, May 2016.

[3] ——, "Automated tools to improve the ping-to-chart workflow," *Int. Hydro. Review*, [Submitted, 2017].

[4] B. R. Calder and G. Rice, "Computationally efficient variable resoution depth estimation," *Computers and Geosciences*, [Submitted, 2016].

[5] B. R. Calder, S. Byrne, B. Lamey, R. T. Brennan, J. D. Case, D. Fabre, B. Gallagher, R. W. Ladner, F. Moggert, and M. Paton, "The open navigation surface project," *Int. Hydro. Review*, vol. 6, no. 2, pp. 9–18, 2005.

[6] Int. Hydro. Org., "Transfer standard for digital hydrographic data (3.1)," International Hydrographic Bureau, 4, quai Antoine 1er, B.P. 445-MC 98011 MONACO Cedex, Tech. Rep. S-57, November 2000.

[7] ——, "Universal hydrographic data model (2.0.0)," International Hydrographic Bureau, 4, quai Antoine 1er, B.P. 445-MC 98011 MONACO Cedex, Tech. Rep. S-100, June 2015.

[8] ——, "Bathymetric surface product specification," International Hydrographic Bureau, 4, quai Antoine 1er, B.P. 445-MC 98011 MONACO Cedex, Tech. Rep. S-102, April 2012.

[9] B. R. Calder and L. A. Mayer, "Automatic processing of high-rate, high-density multibeam echosounder data," *Geochem., Geophys. and Geosystems (G3) DID 10.1029/2002GC000486*, vol. 4, no. 6, 2003.

[10] B. R. Calder and G. Rice, "Design and implementation of an extensible variable resolution bathymetric estimator," in *Proc. US Hydro. Conf.* Hydro. Soc. Am., April 2011.

[11] Information Technology Laboratory, "Secure hash standard (SHS)," National Institute of Standards and Technology, Tech. Rep. 180-4, August 2015.