Development Testing Platform Engines for C/C++ User's Guide

Version 10.3

Parasoft Corporation
101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Phone: (888) 305-0041
Fax: (626) 305-9048
E-mail: info@parasoft.com
URL: www.parasoft.com

PARASOFT END USER LICENSE AGREEMENT

PLEASE READ THIS END USER LICENSE AGREEMENT ("AGREEMENT") CAREFULLY BEFORE USING THE SOFTWARE. PARASOFT CORPORATION ("PARASOFT") IS WILLING TO LICENSE THE SOFTWARE TO YOU, AS AN INDIVIDUAL OR COMPANY THAT WILL BE USING THE SOFTWARE ("YOU" OR "YOUR") ONLY ON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS OF THIS AGREEMENT. THIS IS A LEGALLY ENFORCEABLE CONTRACT BETWEEN YOU AND PARASOFT. BY CLICKING THE "ACCEPT" OR "YES" BUTTON, OR OTHERWISE INDICATING ASSENT ELECTRONICALLY, OR BY INSTALLING THE SOFTWARE, YOU AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT AND ALSO AGREE THAT IS IT ENFORCEABLE LIKE ANY WRITTEN AND NEGOTIATED AGREEMENT SIGNED BY YOU. IF YOU DO NOT AGREE TO THESE TERMS AND CONDITIONS, CLICK THE "I DO NOT ACCEPT" OR "NO" BUTTON AND MAKE NO FURTHER USE OF THE SOFTWARE.

1. DEFINITIONS

1.1. **"Community Edition"** means a limited version of certain Software available with a no cost license. You can execute only one Instance of a Community Edition on a single machine. You shall provide Parasoft with a valid email address at the time of installation, and such email address cannot be used by any other individual to register a Community Edition. You may not transfer the Community Edition to another machine without prior written approval from Parasoft. You may not tamper or attempt to bypass any of the installation steps for a Community Edition, and Parasoft shall terminate your right to a Community Edition in the event that you do so. Notwithstanding any other provisions herein, Parasoft (a) does not provide Maintenance for Community Editions; (b) provides no warranty for Community Editions; (c) provides no indemnification for Community Editions; and (d) accepts no liability for Community Editions.

1.2. **"Concurrent User"** means a person that has accessed the Software at any given point in time, either directly or through an application.

1.3. **"Instance"** means a single occurrence of initialization or execution of software on one machine. You are prohibited from using more than one Instance on the same machine at the same time.

1.4. **"Licensed Capacity"** means the capacity-based license pricing metrics, including, without limitation, Concurrent Users, Node Locked machines, Instances, and Community Editions.

1.5. **"Maintenance"** means the maintenance and technical support services for the Software identified in the Order Instrument and provided by Parasoft pursuant to this Agreement.

1.6. **"Node Locked"** means a license for a single machine that has been authorized to run a single Instance of the licensed Software. A Node Locked license requires that users are physically present and not accessing the machine and using the Software from a remote location.

1.7. **"Software"** means Parasoft's software products, in object code form, that are commercially available at the time of Your order and identified on the Order Instrument, and any modifications, corrections and updates provided by Parasoft in connection with Maintenance.

1.8. **"Territory"** means the country or countries in which You have a license to use the Software, as specified in Your order for the Software; or, if no Territory is specified, the country from which Your order has been issued.

1.9. **"User Documentation"** means the user's guide, installation guides, and/or on-line documentation applicable to the Software. User Documentation does not include marketing materials or responses to requests for proposals.

2. GRANT OF LICENSE AND USE OF SOFTWARE

2.1. **License Grant.** Subject to the terms and conditions of this Agreement, Parasoft grants to You a non-exclusive license to use the Software within the Territory, in accordance with the User Documentation and in compliance with the authorized Licensed Capacity. This license may be perpetual or for a limited duration term, as stated in (a) an executed agreement between You and Parasoft; (b) a sales quotation from Parasoft; (c) a purchase order that You issue to Parasoft; or (d) the online ordering process found on Parasoft's website or an authorized third party's website. You acknowledge and agree that this Agreement only grants a license to the Software as set forth herein and does not constitute a sale of the Software by Parasoft. You have no right to resell the Software, whether by contract or by operation of applicable copyright law.

2.2. **Usage Rights.** You may only use the Software and/or the User Documentation for Your internal business operations and to process Your data. You shall not (a) permit any third parties or non-licensed entities to use the Software or the User Documentation; (b) process or permit to be processed any data that is not Your data; (c) use the Software in the operation of a service bureau; (d) sublicense, rent, or lease the Software or the User Documentation to a third party; or (e) perform, publish, or release to any third parties any benchmarks or other comparisons regarding the Software or User Documentation. You shall not make simultaneous use of the Software on multiple, partitioned, virtual, or cloud hosted computers without first procuring an appropriate number of licenses from Parasoft. You shall not bypass or attempt to bypass any licensing controls either contained within the Software or imposed by Parasoft. You shall not permit a third party outsourcer to use the Software to process data on Your behalf without Parasoft's prior written consent.

2.3. **License Keys.** You acknowledge that the Software contains one or more license keys that will enable the functionality of the Software and third party software embedded in or distributed with the Software. You may only access and use the Software with license keys issued by Parasoft, and shall not attempt to modify, tamper with, reverse engineer, reverse compile, or disassemble any license key. If Parasoft issues a new license key for the Software, You shall not use the previous license key to enable the Software. If a particular license is then currently on Maintenance, You may transfer such license to a different machine and request a new license key from Parasoft.

2.4. **Archival Copies.** You may make one copy of the Software for back-up and archival purposes only. You may make a reasonable number of copies of the User Documentation for Your internal use. All copies of Software and User Documentation must include all copyright and similar proprietary notices appearing on or in the originals. Copies of the Software may be stored offsite provided that all persons having access to the Software are subject to Your obligations under this Agreement and You take reasonable precautions to ensure compliance with these obligations. Parasoft reserves the right to revoke permission to reproduce copyrighted and proprietary material if Parasoft reasonably believes that You have failed to comply with its obligations hereunder.

2.5. **Licensed Capacity.** Parasoft licenses Software based on Licensed Capacity for different types of usage, including, without limitation, Concurrent Users, Node Locked machines, and Community Editions. A Concurrent User license allows multiple Concurrent Users to share access to and use the Software, provided that the number of Concurrent Users accessing the Software at any time does not exceed the total number of licensed Concurrent Users. A Node Locked license allows a single specified machine to run a single Instance of the Software. If an application accessing the Software is a multiplexing, database, or web portal application that permits users of such application to access the Software or data processed by the Software, a separate Concurrent User license will be required for each Concurrent User of such application. Regardless of usage type, You shall immediately notify Parasoft in writing of any increase in use beyond the Licensed Capacity. You must obtain a license for any increase in

Licensed Capacity, and You agree to pay to Parasoft additional Software license fees, which will be based on Parasoft's then-current list price.

2.6. **Third Party Terms.** You acknowledge that software provided by third party vendors ("Third Party Software") may be embedded in or delivered with the Software. The terms of this Agreement and any other terms that Parasoft may specify will apply to such Third Party Software, and the Third Party Software vendors will be deemed third party beneficiaries under this Agreement. You may only use the Third Party Software with the Software. You may not use the Third Party Software on a stand-alone basis or use or integrate it with any other software or device.

2.7. **Evaluation License.** This Section 2.7 applies if Parasoft has provided the Software to You for evaluation purposes. Parasoft grants to You a thirty (30) day, limited license solely for the purpose of internal evaluation. You are strictly prohibited from using the Software for any production purpose or any purpose other than the sole purpose of determining whether to purchase a commercial license for the Software that You are evaluating. Parasoft is not obligated to provide maintenance or support for the evaluation Software. YOU ACKNOWLEDGE THAT SOFTWARE PROVIDED FOR EVALUATION MAY (A) HAVE LIMITED FEATURES; (B) FUNCTION FOR A LIMITED PERIOD OF TIME; OR (C) HAVE OTHER LIMITATIONS NOT CONTAINED IN A COMMERCIAL VERSION OF THE SOFTWARE. NOTWITHSTANDING ANYTHING TO THE CONTRARY IN THIS AGREEMENT, PARASOFT IS PROVIDING THE EVALUATION SOFTWARE TO YOU "AS IS", AND PARASOFT DISCLAIMS ANY AND ALL WARRANTIES (INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND STATUTORY WARRANTIES OF NON-INFRINGEMENT), LIABILITIES, AND INDEMNIFICATION OBLIGATIONS OF ANY KIND. In the event of any conflict between this Section 2.7 and any other provision of this Agreement, this Section 2.7 will prevail and supersede such other provision with respect to Software licensed to You for evaluation purposes.

2.8. **Education License.** If You are an educational or academic institution and are receiving a discount from Parasoft, You may use the Software solely for education or academic purposes and You may not use the Software for any commercial purpose. Parasoft may require that You provide proof of your status as an educational or academic institution.

2.9. **Audit.** You shall maintain accurate business records relating to its use and deployment of the Software. Parasoft shall have the right, not more than once every twelve (12) months and upon ten (10) business days prior written notice, to verify Your compliance with its obligations under this Agreement by auditing Your business records and Your use and deployment of the Software within Your information technology systems. Parasoft and/or a public accounting firm selected by Parasoft shall perform the audit during Your regular business hours and comply with Your reasonable safety and security policies and procedures. Any agreement You may require the public accounting firm to execute shall not prevent disclosure of the audit results to Parasoft. You shall reasonably cooperate and assist with such audit. You shall, upon demand, pay to Parasoft all license and Maintenance fees for any unauthorized deployments and/or excess usage of Software products disclosed by the audit. License fees for such unauthorized deployments and/or excess usage shall be invoiced to and paid by You at Parasoft's then-current list price, and applicable Maintenance fees shall be applied retroactively to the entire period of the unauthorized and/or excess usage. Parasoft shall be responsible for its own costs and expenses in conducting the audit, unless the audit indicates that You have exceeded its Licensed Capacity or otherwise exceeds its license restrictions, such that the then-current list price of non-compliant Software deployment exceeds five percent (5%) of the total then-current list price of the Software actually licensed by You, in which event You shall, upon demand, reimburse Parasoft for all reasonable costs and expenses of the audit.

3. **TITLE**. Parasoft retains all right, title and interest in and to the Software and User Documentation and all copies, improvements, enhancements, modifications and derivative works of the Software and User Documentation, including, without limitation, all patent, copyright, trade

secret, trademarks and other intellectual property rights. You agrees that it shall not, and shall not authorize others to, copy (except as expressly permitted herein), make modifications to, translate, disassemble, decompile, reverse engineer, otherwise decode or alter, or create derivative works based on the Software or User Documentation. Except as otherwise provided, Parasoft grants no express or implied rights under this license to any of Parasoft's patents, copyrights, trade secrets, trademarks, or other intellectual property rights..

4. TERMINATION

4.1. **Default; Bankruptcy**. Parasoft may terminate this Agreement if (a) You fail to pay any amount when due under any order You have placed with Parasoft and do not cure such non-payment within ten (10) days of receipt of written notice of non-payment; (b) You materially breach this Agreement and do not cure such breach within thirty (30) days of receipt of written notice of such breach; (c) subject to provisions of applicable bankruptcy and insolvency laws, You become the subject of any involuntary proceeding relating to insolvency and such petition or proceeding is not dismissed within sixty (60) days of filing; or (d) You become the subject of any voluntary or involuntary petition pursuant to applicable bankruptcy or insolvency laws, or request for receivership, liquidation, or composition for the benefit of creditors and such petition, request or proceeding is not dismissed within sixty (60) days of filing.

4.2. **Effect of Termination**. Upon termination of this Agreement, You shall immediately discontinue use of, and uninstall and destroy all copies of, all Software. Within ten (10) days following termination, You shall certify to Parasoft in a writing signed by an officer of Yours that all Software has been uninstalled from Your computer systems and destroyed.

5. LIMITED WARRANTY

5.1. **Performance Warranty**. Parasoft warrants that the Software, as delivered by Parasoft and when used in accordance with the User Documentation and the terms of this Agreement, will substantially perform in accordance with the User Documentation for a period of ninety (90) days from the date of initial delivery of the Software. If the Software does not operate as warranted and You have provided written notice of the non-conformity to Parasoft within the ninety (90) day warranty period, Parasoft shall at its option (a) repair the Software; (b) replace the Software with software of substantially the same functionality; or (c) terminate the license for the nonconforming Software and refund the applicable license fees received by Parasoft for the nonconforming Software. The foregoing warranty specifically excludes defects in or non-conformance of the Software resulting from (a) use of the Software in a manner not in accordance with the User Documentation; (b) modifications or enhancements to the Software made by or on behalf of You; (c) combining the Software with products, software, or devices not provided by Parasoft; or (d) computer hardware malfunctions, unauthorized repair, accident, or abuse.

5.2. **Disclaimers**. THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE AND IN LIEU OF ALL OTHER WARRANTIES, WHETHER EXPRESS OR IMPLIED, AND PARASOFT EXPRESSLY DISCLAIMS ALL OTHER WARRANTIES, INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND STATUTORY WARRANTIES OF NON-INFRINGEMENT. PARASOFT DOES NOT WARRANT THAT THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT USE OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR FREE. THE REMEDIES SET FORTH IN THIS SECTION 5 ARE YOUR SOLE AND EXCLUSIVE REMEDIES AND PARASOFT'S SOLE AND EXCLUSIVE LIABILITY REGARDING FAILURE OF ANY SOFTWARE TO FUNCTION OR PERFORM AS WARRANTED IN THIS SECTION 5.

6. INDEMNIFICATION

6.1. **Infringement**. Parasoft shall defend any claim against You that the Software infringes any intellectual property right of a third party, provided that the third party is located in a country that is a signatory to the Berne Convention, and shall indemnify You against any and all damages finally awarded against You by a court of final appeal, or agreed to in settlement by Para-

soft and attributable to such claim, so long as You (a) provide Parasoft prompt written notice of the claim; (b) provide Parasoft all reasonable assistance and information to enable Parasoft to perform its duties under this Section 6; (c) allow Parasoft sole control of the defense and all related settlement negotiations; and (d) have not compromised or settled such claim. If the Software is found to infringe, or if Parasoft determines in its sole opinion that it is likely to be found to infringe, then Parasoft may, at its option (a) obtain for You the right to continue to use the Software; (b) modify the Software to be non-infringing or replace it with a non-infringing functional equivalent, in which case You shall stop using any infringing version of the Software; or (c) terminate Your rights and Parasoft's obligations under this Agreement with respect to such Software and refund to You the unamortized portion of the Software license fee paid for the Software based on a five year straight-line depreciation schedule commencing on the date of delivery of the Software. The foregoing indemnity will not apply to any infringement resulting from (a) use of the Software in a manner not in accordance with the User Documentation; (b) modifications or enhancements to the Software made by or on behalf of You; (c) combination, use, or operation of the Software with products not provided by Parasoft; or (d) use of an allegedly infringing version of the Software if the alleged infringement could be avoided by the use of a different version of the Software made available to You.

6.2. Disclaimers. THIS SECTION 6 STATES YOUR SOLE AND EXCLUSIVE REMEDY AND PARASOFT'S SOLE AND EXCLUSIVE LIABILITY REGARDING INFRINGEMENT OR MISAPPROPRIATION OF ANY INTELLECTUAL PROPERTY RIGHTS OF A THIRD PARTY.

7. **LIMITATION OF LIABILITY**. IIN NO EVENT WILL PARASOFT OR ITS THIRD PARTY VENDORS BE LIABLE TO YOU OR ANY OTHER PARTY FOR (A) ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR (B) LOSS OF DATA, LOSS OF PROFITS, BUSINESS INTERRUPTION, OR SIMILAR DAMAGES OR LOSS, EVEN IF PARASOFT AND ITS THIRD PARTY VENDORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. EXCEPT AS LIMITED BY APPLICABLE LAW AND EXCLUDING PARASOFT'S LIABILITY TO YOU UNDER SECTION 6 (INDEMNIFICATION), AND REGARDLESS OF THE BASIS FOR YOUR CLAIM, PARASOFT'S MAXIMUM LIABILITY UNDER THIS AGREEMENT WILL BE LIMITED TO THE LICENSE OR MAINTENANCE FEES PAID FOR THE SOFTWARE OR MAINTENANCE GIVING RISE TO THE CLAIM. THE FOREGOING LIMITATIONS WILL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.
8. **CONFIDENTIAL INFORMATION**. For purposes of this Agreement, "Confidential Information" will include trade secrets contained within the Software and User Documentation, the terms and pricing of the Software and Maintenance (including any pricing proposals), and such other information (a) identified by either party as confidential at the time of disclosure or (b) that a reasonable person would consider confidential due to its nature and circumstances of disclosure ("Confidential Information"). Confidential Information will not include information that (a) is or becomes a part of the public domain through no act or omission of the receiving party; (b) was in the receiving party's lawful possession prior to receiving it from the disclosing party; (c) is lawfully disclosed to the receiving party by a third party without restriction on disclosure; or (d) is independently developed by the receiving party without breaching this Agreement. Each party agrees to maintain all Confidential Information in confidence and not disclose any Confidential Information to a third party or use the Confidential Information except as permitted under this Agreement. Each party shall take all reasonable precautions necessary to ensure that the Confidential Information is not disclosed by such party or its employees, agents or authorized users to any third party. Each party agrees to immediately notify the other party of any unauthorized access to or disclosure of the Confidential Information. The receiving party agrees that any breach of this Section 8 may cause irreparable harm to the disclosing party, and such disclosing party shall be entitled to seek equitable relief in addition to all other remedies provided by this Agreement or available at law.
9. **MAINTENANCE**

9.1. **Maintenance Period**. If You have purchased a perpetual license, You are required to purchase first year Maintenance with the Software, and the Maintenance period will commence upon the initial delivery of the Software and continue for a period of one year. If You have purchased a term license, Maintenance during the term is included at no additional charge. The Maintenance period, at Your option, may be renewed pursuant to subsequent orders. Prior to such renewal, Parasoft may, upon ten (10) business days written notice, require You to provide a report on Your use and deployment of the Software. Such report will be certified by an officer of Yours and will specify, with respect to Your Software: (a) the type and amount of Licensed Capacity; (b) the version; and (c) the Parasoft license serial number. Parasoft shall issue an annual renewal notice to You at least ninety (90) days prior to the expiration of the then-current Maintenance period. Maintenance fees will be based on the then-current list price and are subject to change without notice.

9.2. **Support Coordinators**. Maintenance will consist of support services provided by Parasoft to one designated support coordinator of Yours (and one backup coordinator) per Your location, by telephone, email, and website. Support is available during normal business hours in the applicable location within the Territory, Monday through Friday, excluding nationally observed holidays.

9.3. **Additional Licensed Capacity**. Additional Licensed Capacity. In the event that You purchases additional Licensed Capacity for the Software prior to the annual anniversary date of the Maintenance period, You agree to pay applicable Maintenance fees based on Parasoft's then-current Maintenance rates. Maintenance fees will apply from the effective date of such additional Licensed Capacity and continue for a period of one year thereafter, unless otherwise agreed to in writing by the parties, so that Maintenance for Your previously acquired Software and added Licensed Capacity is coterminous.

9.4. **New Releases**. During any period in which You are current on Maintenance, Parasoft shall provide You with any new release of the Software, which may include generally available error corrections, modifications, maintenance patch releases, enhancements (unless priced separately by Parasoft and generally not included with new licenses for the Software at that time), and the revised User Documentation, if applicable. Notwithstanding the foregoing, stand-alone error corrections that are not part of a new release will not be independently supported but will be incorporated into the next release of the Software. If You install a new release of the Software, You may continue to use the previous version of the Software for up to ninety (90) days in order to assist You in the transition to the new release. Once You complete its transition to the new release of the Software, You must discontinue use of the previous version of the Software.

9.5. **Supported Releases**. Parasoft shall continue to support the immediately preceding release of the Software for a period of twelve (12) months following the discontinuance of such Software or the date on which the new release becomes generally available, provided that You have paid applicable Maintenance fees and incorporated all Maintenance patch releases issued by Parasoft for the release of the Software.

9.6. **Reinstatement of Maintenance**. If You allow Maintenance to expire, You may, at a later date, renew Maintenance by paying the following: (a) if You have installed the current release of the Software but have failed to pay the applicable renewal fee on or before the ninetieth (90th) day following expiration of the Maintenance period, annual Maintenance fees at Parasoft's then-current rates, plus Parasoft's then-current reinstatement fee; or (b) if You have not installed the current release of the Software or have failed to pay the applicable renewal fee by the ninetieth (90th) day following expiration of the Maintenance period, annual Maintenance fees at Parasoft's then-current rates, plus Parasoft's then-current license update fee for the current release of the Software.

10. GENERAL

10.1. **Independent Contractors**. The parties acknowledge and agree that each is an independent contractor. This Agreement will not be construed to create a partnership, joint venture or agency relationship between the parties.

10.2. **Entire Agreement**. The terms and conditions of this Agreement apply to all Software licensed, all User Documentation provided, and all Maintenance purchased hereunder. This Agreement will supersede any different, inconsistent or preprinted terms and conditions in any order form of Yours, purchase order or other ordering document.

10.3. **Assignment**. You have no right to assign, sublicense, pledge, or otherwise transfer any of Your rights in and to the Software, User Documentation or this Agreement, in whole or in part (collectively, an "Assignment"), without Parasoft's prior written consent, and any Assignment without such consent shall be null and void. Any change in control of Your organization or entity, whether by merger, share purchase, asset sale, or otherwise, will be deemed an Assignment subject to the terms of this Section 10.3.

10.4. **Force Majeure**. No failure, delay or default in performance of any obligation of a party to this Agreement, except payment of license fees due hereunder, will constitute an event of default or breach of the Agreement to the extent that such failure to perform, delay or default arises out of a cause, existing or future, that is beyond the reasonable control of such party, including, without limitation, action or inaction of a governmental agency, civil or military authority, fire, strike, lockout or other labor dispute, inability to obtain labor or materials on time, flood, war, riot, theft, earthquake or other natural disaster ("Force Majeure Event"). The party affected by such Force Majeure Event shall take all reasonable actions to minimize the consequences of any Force Majeure Event.

10.5. **Severability**. If any provision of this Agreement is held to be illegal or otherwise unenforceable by a court of competent jurisdiction, that provision will be severed and the remainder of the Agreement will remain in full force and effect.

10.6. **Waiver**. The waiver of any right or election of any remedy in one instance will not affect any rights or remedies in another instance. A waiver will be effective only if made in writing and signed by an authorized representative of the applicable party.

10.7. **Notices**. All notices required by this Agreement will be in writing, addressed to the party to be notified and deemed to have been effectively given and received (a) on the fifth business day following deposit in the mail, if sent by first class mail, postage prepaid; (b) upon receipt, if sent by registered or certified U.S. mail, postage prepaid, with return receipt requested; (c) upon transmission, if sent by facsimile and confirmation of transmission is produced by the sending machine and a copy of such facsimile is promptly sent by another means specified in this Section 10.7; or (d) upon delivery, if delivered personally or sent by express courier service and receipt is confirmed by the recipient. Notices will be addressed to the parties based on the address stated in the applicable order, to the attention of the Legal Department. A change of address for notice purposes may be made pursuant to the procedures set forth above.

10.8. **Export Restrictions**. You acknowledge that the Software and certain Confidential Information (collectively "Technical Data") are subject to United States export controls under the U. S. Export Administration Act, including the Export Administration Regulations, 15 C.F.R. Parts 730 et seq. (collectively, "Export Control Laws"). Each party agrees to comply with all requirements of the Export Control Laws with respect to the Technical Data. Without limiting the foregoing, You shall not (a) export, re-export, divert or transfer any such Technical Data, or any direct product thereof, to any destination, company, or person restricted or prohibited by Export Control Laws; (b) disclose any such Technical Data to any national of any country when such disclosure is restricted or prohibited by the Export Control Laws; or (c) export or re-export the Technical Data, directly or indirectly, for nuclear, missile, or chemical/biological weaponry end uses prohibited by the Export Control Laws.

10.9. **U. S. Government Rights**. The Software and User Documentation are deemed to be "commercial computer software" and "commercial computer software documentation" as defined in FAR Section 12.212 and DFARS Section 227.7202, as applicable. Any use, modification, reproduction, release, performance, display, or disclosure of the Software and User Documentation by the United States government will be solely in accordance with the terms of this Agreement.

10.10. **Choice of Law; Jurisdiction**. This Agreement is governed by and construed in accordance with the laws of the State of California, U. S. A., exclusive of any provisions of the United Nations Convention on Contracts for the International Sale of Goods, including any amendments thereto, and without regard to principles of conflicts of law. Any suits concerning this Agreement will be brought in the federal courts for the Central District of California or the state courts in Los Angeles County, California. The parties expressly agree that the Uniform Computer Information Transactions Act, as adopted or amended from time to time, will not apply to this Agreement or the Software and Maintenance provided hereunder.

10.11. **Amendment**. This Agreement may only be modified by a written document signed by an authorized representative of Parasoft and by You.

10.12. **Survival**. Any terms of this Agreement which by their nature extend beyond the termination or expiration of this Agreement will remain in effect. Such terms will include, without limitation, all provisions herein relating to limitation of liability, title and ownership of Software, and all general provisions.

Parasoft Corporation

101 East Huntington Drive, 2nd Floor

Monrovia, CA 91016 USA

+1 (626) 256-3680

+1 (888) 305-0041 (USA only)

+1 (626) 256-9048 (Fax)

info@parasoft.com

http://www.parasoft.com

Printed in the U.S.A, May 16, 2017

Table of Contents

Reporting

Unit Test Connector

Code Coverage Engine

Customizing DTP Engines for C/C++

Integrations

Supported Compilers

Getting Help

Third-Party Content

Introduction

Parasoft Development Testing Platform (DTP) Engines for C/C++ are integrated solutions for automating a broad range of best practices to improve productivity and software quality. DTP Engines are a component of the Parasoft Development Testing Platform family of software quality solutions.Please read the following guide for additional information about how DTP Engines integrate into Parasoft's Development Testing ecosystem:

The Parasoft Development Testing Solution (PDF)

This documentation provides information on how to use the following engines:

Static Analysis Engine (SAE)

SAE enforces your coding policy with proven quality practices, such as static analysis and flow analysis, to ensure that your C and C++ applications function as expected. See "Static Analysis Engine", page 10.

Unit Test Connector (UTC)

UTC allows you to run unit tests from open format tools, and report results to Development Testing Platform (DTP) Server. See "Unit Test Connector", page 49.

Code Coverage Engine (CCE)

CCE collects coverage information during a run of the executable and generates reports that can be sent to DTP Server. See "Code Coverage Engine", page 56.

Getting Started

This chapter will help you verify that your system meets the requirements for using DTP Engines, as well as help you configure DTP Engines so you can quickly start analyzing code.

System Requirements

Windows 32-bit

- Windows 7, Windows 8
- 4GB memory minimum*
- 2GHz or faster processor (x86-compatible), multi-CPU configuration recommended
- Supported C / C++ compiler

Windows 64-bit

- Windows 7 (x64), Windows 8 (x64), Windows 10, Windows 2008 Server (x64), Windows Server 2012
- 4GB memory minimum, 8GB recommended*
- 2GHz or faster processor (x86_64-compatible), multi-CPU configuration recommended
- Supported C / C++ compiler

Linux 32-bit

- Linux kernel 2.6 (or newer) with glibc 2.9 (or newer)
- 4GB memory minimum*
- 2GHz or faster processor (x86-compatible), multi-CPU configuration recommended
- Supported C / C++ compiler

Linux 64-bit

- Linux kernel 2.6 (or newer) with glibc 2.12 (or newer)
- 4GB memory minimum, 8GB recommended*
- 2GHz or faster processor (x86_64-compatible), multi-CPU configuration recommended
- Supported C / C++ compiler

*DTP Engines for C/C++ may allocate up to 1GB RAM on 32-bit machines or up to 2GB RAM on 64-bit machines for the Java Virtual Machine process. You can change memory allocation for the JVM process in the `[INSTALL_DIR]/etc/cpptestcli.jvm` configuration file (`-Xmx` option). When running an analysis, DTP Engines' native code analyzers will require additional memory, depending on the test configuration parameters.

Installing DTP Engines

1. Unpack the distribution file. A directory ([NSTALL_DIR]) called "cpptest" that contains all DTP Engine files will be created.

2. Add [INSTALL_DIR] to $PATH to enable convenient access to the `cpptestcli` executable.
3. Add [INSTALL_DIR]/bin to $PATH to enable convenient access to utility tools, including the tool for instrumenting the code for coverage (`cpptestcc`).

Remove the installation directory from disk to uninstall DTP Engines.

If you are using Azure or AWS services, you need to configure the `cloudvm` option in the .properties configuration file. You can set the option to one of the following values:

- `azure` - Enables integration with Azure
- `aws` - Enables integration with AWS
- `true` - Enables integration with the automatically detected cloud computing platform
- `false` - Disables integration (the default value)

If you set the value to `false` or if the option is not configured, integration with Azure or AWS is disabled.

Setting the License

DTP Engines can run on either a local or a network license. There are two types of network licenses:

- `dtp`: This license is stored in DTP. Your DTP license limits analysis to the number of files specified in your licensing agreement. This is the default type when `license.use_network` is set to `true`.
- `ls`: This is a "floating" or "machine-locked" license that limits usage to a specified number of machines. This type of license is stored in DTP in License Server.

Network licenses are also available in three editions that determine what functionality is available:

- `desktop_edition`: Functionality is optimized for desktop usage.
- `server_edition`: Functionality configured for high performance usage in server command line mode.
- `custom_edition`: functionality can be customized.

Local License

In the .properties configuration file:

1. Set the `cpptest.license.use_network` property to `false`
2. Set the `cpptest.license.local.password` property with your password

Obtaining the Machine ID

If you are using a local license, you will need your machine ID to request a license from Parasoft. Run the following command from a command line window to obtain your machine ID:

```
cpptestcli -machineID
```

Network License

In the .properties configuration file:

1. Set the `cpptest.license.use_network` property to `true`

2. Set the `cpptest.license.network.type`
3. Set the `cpptest.license.network.edition`

Connecting to DTP Server

Connecting to DTP Server is required for licensing, as well as extending other team-working capabilities, such as:

- Reporting analysis to a centralized database (see “Sending Results to Development Testing Platform (DTP) Server”, page 47)
- Sharing test configurations
- Sharing static analysis rules

Modify the following settings in the `[INSTALL_DIR]\cpptestcli.properties` file to configure the connection to DTP Server.

```
dtp.server=[SERVER]
dtp.port=[PORT]
dtp.user=[USER]
dtp.password=[PASSWORD]
```

Creating an Encoded Password

DTP Engines can encrypt your password, which adds a layer of security to your interactions with DTP Server. Run the following command to print an encoded password:

```
-encodepass [MYPASSWORD]
```

Copy the encoded password that is returned and paste it into the cpptestcli.properties file.

```
dtp.password=[ENCODED PASSWORD]
```

Connecting to Source Control

Parasoft DTP Engines ship with out-of-the-box support for the following SCMs:

Brand	Tested Version
AccuRev	4.6, 5.4, 6.2
ClearCase	2003.06, 7.0, 8.0
CVS	1.1.2
Git	1.7
Mercurial	1.8.0 - 3.6.3
Perforce	2006, 2012, 2013, 2014, 2015
Serena Dimensions	9.1, 10.1, 10.3 (2009 R2), 12.2

Brand	Tested Version
Star Team	2005, 2008, 2009
Subversion (SVN)	1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9
Synergy/CM	6.4, 7.0, 7.1
Microsoft Team Foundation Server	2008, 2010, 2012, 2013, 2015
Visual SourceSafe	5.0, 6.0, 2005

Edit the `cpptestcli.properties` file located in the installation directory to connect to your SCM. Parameters will vary depending on the brand of your SCM. The following example shows the parameters required to connect to SVN:

```
scontrol.rep.type=svn
scontrol.rep.svn.url=https://svn_server/
scontrol.rep.svn.login=username
scontrol.rep.svn.password=password
scontrol.svn.exec=C:\\path\to\svn.exe
```

See “Customizing DTP Engines for C/C++”, page 76, for information about configuring your SCM connection.

If you have C++test 9.5 or later, you can use its interface to configure integration with source control systems. See “Integrating with Source Control Systems”, page 102 for details.

Static Analysis Engine

Static Analysis Engine (SAE) enforces your coding policy with proven quality practices, such as static analysis and flow analysis, to ensure that your applications function as expected. The following sections describe how to analyze code with SAE.

- Basic Analysis
- Specifying Test Configurations
- Defining Input Scope
- Code Duplicate Analysis
- Using DTP Engines in an IDE

Basic Analysis

Executable

Verify that the executable (cpptestcli) is on $PATH. See “Installing DTP Engines”, page 6.

Prerequisites

Compiler

Static Analysis Engine must be configured for use with specific C and C++ compilers and versions before you can analyze code. The configuration should reflect the original compiler and version used for building the code under test. The original compiler executable must be available on $PATH (unless it is specified with a full path).

Use the `-compiler` switch to specify the compiler configuration identifier:

```
cpptestcli -config "builtin://Recommended Rules" -compiler gcc_4_5 -input
cpptest.bdf
```

Alternatively, the `cpptest.compiler.family` entry could be added to a custom configuration file:

```
cpptest.compiler.family=gcc_4_5
```

If you are using a single compiler and version for all testing, the compiler identifier can be specified in the `cpptestcli.properties` global configuration file in either the [INSTALL_DIR] or [USER_HOME] directory.

Compiler Discovery

Perform one of the following actions to find the configuration for that compiler:

- Use the `-detect-compiler` switch to auto-detect configuration:

```
cpptestcli -detect-compiler gcc
```

- Use the `-list-compilers` switch to find the configuration in the list of all supported compilers:

```
cpptestcli -list-compilers
```

Also see “Compiler Configuration”, page 21, and “Supported Compilers”, page 104, for additional information.

About Usage Examples

The following instructions assume that:

- GNU GCC 3.4 compiler is being used (configuration identifier: gcc_3_4)
- The prerequisites discussed above have been met.
- Users are running commands in the [INSTALL_DIR]/examples/ATM directory.

Analyzing a Single File

Run an analysis and specify the original **compiler command** with the `--` switch (separator):

```
cpptestcli -config "builtin://Recommended Rules" -compiler gcc_3_4 -- gcc -I
include Bank.cxx
```

Static Analysis Engine will analyze the Bank.cxx file using the original compiler executable and compiler options and report detected violations to the output console.

All values after `--` switch will be interpreted as a compiler command, so options specific to Static Analysis Engine must be specified before `--` switch.

Only the specified source files will be analyzed. Header files included by the source files will be excluded from analysis. To broaden the scope of files tested, including header files, see “Defining Input Scope”, page 22.

Analyzing a Makefile-based Project

Run code analysis and specify the original **build command** with the `-trace` switch:

```
cpptestcli -config "builtin://Recommended Rules" -compiler gcc_3_4 -trace make
clean all
```

Static Analysis Engine will perform the following tasks:

1. Run the original build (make clean all)
2. Detect which files to test
3. Run the analysis for these files
4. Report results to the output console
5. Store all build information in the cpptest.bdf file for future runs (see “About Build Data Files”, page 12, for additional information about build data files)

All values after `-trace` will be interpreted as a build command, so options specific to Static Analysis Engine must be specified before `-trace`. Additionally, Static Analysis Engine will detect a source file for testing only if that file was compiled when running the build command. Only source files from Makefile will be analyzed. Header files included by the source files will be excluded from analysis. To broaden the scope of files tested, including header files, see “Defining Input Scope”, page 22.

About Build Data Files

You can create a build data file (.bdf), which stores information such as the working directory, command line options for the compilation, and link processes of the original build, so that Static Analysis Engine can analyze a project without having to rebuild it. The following example is a fragment from a build data file:

```
working_dir=/home/place/project/hypnos/pscom
project_name=pscom
arg=g++
arg=-c
arg=src/io/Path.cc
arg=-Iinclude
arg=-I.
arg=-o
arg=/home/place/project/hypnos/product/pscom/shared/io/Path.o
```

You can use the -trace switch to create a .bdf (see “Analyzing a Makefile-based Project”, page 12) or use the standalone `cpptestscan` or `cpptesttrace` utility located in the [INSTALL_DIR]/bin directory.

Using cpptestscan and cpptesttrace Utilities

The `cpptestscan` utility is used as a wrapper for the compiler and/or linker during the normal build. To use `cpptestscan` with an existing build, prefix the compiler/linker executable with `cpptestscan` when building the code base. This can be done in two ways:

- Modify the build command line to use `cpptestscan` as the wrapper for the compiler/linker executables
- If you aren't able to override the compiler variable on the command line, embed `cpptestscan` in the actual make file or build script.

To use `cpptesttrace` with an existing build, prefix the entire build command with `cpptesttrace` when building the code base. `cpptesttrace` will trace the compiler and linker processes executed during the build and store them in the build data file.

In both cases, you must specify the full path to either utility in your PATH environment variable. Additional options for `cpptestscan` and `cpptesttrace` are summarized in the following table. Options can be set directly for the `cpptestscan` command or via environment variables. Most options can be applied to `cpptestscan` or `cpptesttrace` by changing the prefix in command line.

Basic cpptestscan usage:

- Windows: `cpptestscan.exe [options] [compile/link command]`
- Linux and Solaris: `cpptestscan [options] [compile/link command]`

Basic cpptesttrace usage:

- Windows: `cpptesttrace.exe [options] [build command]`
- Linux, Solaris: `cpptesttrace [options] [build command]`

Option	Environment Variable	Description	Default

`--cpptestscanOutput-File=<OUTPUT_FILE>`
 	
`--cpptesttraceOutput-File=<OUTPUT_FILE>`
	CPPTEST_SCAN_OUTPUT_FILE	Defines file to append build information to.	cpptestscan.bdf

`--cpptestscanProject-Name=<PROJECT_NAME>`
 	
`--cpptesttraceProject-Name=<PROJECT_NAME>`
	CPPTEST_SCAN_PROJECT_NAME	Defines suggested name of the C++test project.	name of the current working directory

`--cpptestscanRun-OrigCmd=[yes|no]`
 	
`--cpptesttraceRun-OrigCmd=[yes|no]`
	CPPTEST_SCAN_RUN_ORIG_CMD	If set to "yes", original command line will be executed.	yes

Option	Environment Variable	Description	Default

`--cpptestscanQuoteCmd-LineMode=[all|sq|none]`
 	
`--cpptesttraceQuoteCmd-LineMode=[all|sq|none]`
	CPPTEST_SCAN_QUOTE_CMD_LINE_MODE	Determines the way C++test quotes parameters when preparing cmd line to run.
		`all`: all params will be quoted
		`none`: no params will be quoted
		`sq`: only params with space or quote character will be quoted
		cpptestscanQuoteCmdLineMode is not supported on Linux
			all

`--cpptestscanCmd-LinePrefix=<PREFIX>` `--cpptesttraceCmd-LinePrefix=<PREFIX>`	CPPTEST_SCAN_CMD_LINE_PREFIX	If non-empty and running original executable is turned on, the specified command will be prefixed to the original command line.	[empty]

`--cpptestscanEnvInOutput=[yes|no]`
 	
`--cpptesttraceEnvInOutput=[yes|no]`
	CPPTEST_SCAN_ENV_IN_OUTPUT	Enabling dumps the selected environment variables and the command line arguments that outputs the file. For advanced settings use --cpptestscanEnvFile and --cpptestscanEnvvars options	no

`--cpptestscanEnvFile=<ENV_FILE>`
 	
`--cpptesttraceEnvFile=<ENV_FILE>`
	CPPTEST_SCAN_ENV_FILE	If enabled, the specified file keeps common environment variables for all build commands; the main output file will only keep differences. Use this option to reduce the size of the main output file. Use this option with --cpptestscanEnvInOutput enabled	[empty]

Re-analyzing a Project without Re-building

Run code analysis and specify the existing **build data file** with the `-input` switch:

```
cpptestcli -config "builtin://Recommended Rules" -compiler gcc_3_4 -input
cpptest.bdf
```

Static Analysis Engine will perform the following tasks:

1. Read the information about which files to test from the existing cpptest.bdf file
2. Run the analysis for these files
3. Report results to the output console
4. The original build will not be executed

Multiple build data files can be specified using multiple `-input` switches:

```
cpptestcli -config "builtin://Recommended Rules" -compiler gcc_3_4 -input
project1.bdf -input project2.bdf
```

Only the source files defined in the build data file will be analyzed. Header files included by the source files will be excluded from analysis. To broaden the scope of files tested, including header files, see “Defining Input Scope”, page 22.

Generating a .csv Report

Be sure the project was already analyzed with Static Analysis Engine and that the cpptest.bdf file exists (see above)

Create an empty configuration file (csv.properties) and add the following line:

```
cpptest.report.csv.enabled=true
```

Run code analysis and specify the configuration file with `-settings` switch:

```
cpptestcli -config "builtin://Recommended Rules" -compiler gcc_3_4 -settings
csv.properties -input cpptest.bdf
```

Static Analysis Engine will perform the following tasks:

1. Run the analysis as described above
2. Report results to the output console
3. Create an additional report.csv result file

Testing a Microsoft Visual Studio Project or Solution

Static Analysis Engine can read Visual Studio project and solution files and analyze all source and included header files from the project or solution. Use the `-input` switch to specify a Visual Studio project or solution file:

```
cpptestcli -config "builtin://Recommended Rules" -input MyProject.vcproj
```

You can specify the build configuration and the platform you want to use during analysis of your project or solution. Append the configuration and platform names to the solution or project file name. Your command may resemble the following:

```
cpptestcli -config "builtin://Recommended Rules" -input MyProject.vcproj@Debug|x64
```

Alternatively, you can use the following properties to specify the build configuration and the platform you want to use during analysis of all Visual Studio solutions and projects:

```
cpptest.input.msvc.config=Debug
cpptest.input.msvc.platform=x64
```

For all Microsoft Visual Studio settings, see “Visual Studio Configuration Settings”, page 100.

Make sure that the correct version of Microsoft Visual C++ compiler is available on $PATH before running analysis. Microsoft Visual Studio 6 is not supported.

Specifying Test Data Location

Exclusive access to the `.cpptest` directory is required. The directory is created in the current working directory by default, which is where some of the run-specific data is stored. As a result, only one instance of Static Analysis Engine can run in a directory at a time. You can use the `-workspace` switch to change the location of the `.cpptest` directory.

```
-workspace <WORKSPACE_LOCATION>
```

Specifying Test Configurations

Test configurations define how DTP Engines test and analyze code, including which static analysis rules are enabled, which tests to run, and other analysis parameters. DTP Engines ship with built-in test configurations, but users can create and store their own test configurations in the DTP server. You can access the DTP server via the DTP plug-in. If you have administrator-level access in DTP Report Center, you can also create test configurations directly in DTP (**administration> Engines> Test Configurations**).

User-defined test configurations can be downloaded from the DTP server and stored in the `[INSTALL_DIR]/configs/user` directory as `*.properties` files.

Use the `-config` switch to specify which test configuration to run:

```
cpptestcli -config "builtin://Recommended Rules" -compiler gcc_3_4 -input
cpptest.bdf
```

The test configuration being executed can be specified in the following ways (by default, the `builtin://Recommended Rules` test configuration is used):

Built-in Configurations

```
-config "builtin://Recommended Rules"
```

User-defined Configurations

```
-config "user://Foo Configuration"
```

DTP Server-hosted Configurations

```
-config "dtp://Foo Team Configuration"
-config "dtp://FooTeamConfig.properties"
```

Test configurations can also be referenced by filename and URL:

By File Name

```
-config "C:\Devel\Configs\FooConfig.properties"
```

By URL

```
-config "http://foo.bar.com/configs/FoodConfig.properties"
```

Viewing Available Test Configurations

Use the `-listconfigs` switch to print the available test configurations. Use arguments to filter configurations; the use of "*" expresssions is supported.

```
-listconfigs
```

Built-in Test Configurations

The following table includes the test configurations shipped with DTP Engines in the `[INSTALL]/configs/builtin` directory.

Configuration Name	Description
CERT C Coding Standard	Checks rules for the CERT C Secure Coding Standard. This standard provides guidelines for secure coding. The goal is to facilitate the development of safe, reliable, and secure systems by, for example, eliminating undefined behaviors that can lead to undefined program behaviors and exploitable vulnerabilities.
CRules	Checks rules that enforce C best practices.
Effective C++	Checks rules from Scott Meyers' "Effective C++" book. These rules check the efficiency of C++ programs.
Effective STL	Checks rules from Scott Meyers' "Effective STL" book.
Ellemtel	Checks rules based on a C++ style guide that originated by Ellemtel Telecom Systems and is used widely in the telecommunications industry.
GoogleTest	Analyzes Google Test unit test results.
HIS Source Code Metrics	Checks metrics required by the Herstellerinitiative Software (HIS) group.
ISO 26262 Recommended Rules	Checks rules recommended by the ISO 26262 standard.
Joint Strike Fighter	Checks rules that enforce the Joint Strike Fighter (JSF) program coding standards.
MISRA C	Checks rules that enforce the MISRA C coding standards.
MISRA C [2004, 2012]	Checks rules that enforce the MISRA C 2004 or 2012 coding standards.
MISRA C++ 2008	Checks rules that enforce the MISRA C++ 2008 coding standards.
Parasoft's Recommended FDA C++ Phase 1	Checks the core set of rules recommended for complying with the FDA General Principles for Software Validation.
Parasoft's Recommended FDA C++ Phase 2	Checks a broader set of rules recommended for complying with the FDA General Principles for Software Validation; recommended for organizations who have already implemented the phase 1 rule set.

Configuration Name	Description
Parasoft's Recommended FDA C++ Phase 3	Checks a broader set of rules recommended for complying with the FDA General Principles for Software Validation; recommended for organizations who have already implemented the phase 2 rule set.
Sutter-Alexandrescu	Checks rules based on the book "C++ Coding Standards," by Herb Sutter and Andrei Alexandrescu.
The Power of Ten	Checks rules based on Gerard J. Holzmann's article "The Power of Ten - Rules for Developing Safety Critical Code." http://spinroot.com/gerard/pdf/Power_of_Ten.pdf
Recommended Rules	The default configuration of recommended rules. Covers most Severity 1 and Severity 2 rules. Includes rules in the Flow Analysis Fast configuration.
Parasoft's Recommended Rules (Deprecated)	The default configuration of recommended rules. Covers most Severity 1 and Severity 2 rules. Includes rules in the Flow Analysis Fast configuration.
Find Duplicated Code	Applies static code analysis rules that report duplicate code. Duplicate code may indicate poor application design and lead to maintainability issues.
Metrics	Computes values for several code metrics.
DISA-STIG Coding Standard	Includes rules that find issues identified in the DISA-STIG standard
Flow Analysis	Detects complex runtime errors without requiring test cases or application execution. Defects detected include using uninitialized or invalid memory, null pointer dereferencing, array and buffer overflows, division by zero, memory and resource leaks, and dead code. This requires a special Flow Analysis license option.
Flow Analysis Aggressive	Includes rules for deep flow analysis of code. Significant amount of time may be required to run this configuration.
Flow Analysis Fast	Includes rules for shallow depth of flow analysis, which limits the number of potentially acceptable defects from being reported.

Configuration Name	Description
CWE-SANS Top 25 Most Dangerous Programming Errors	Includes rules that find issues classified as Top 25 Most Dangerous Programming Errors of the CWE-SANS standard.
SAMATE Annex A Source Code Weaknesses	Includes rules that find issues identified in the NIST SAMATE standard
OWASP Top 10 Security Vulnerabilities	Includes rules that find issues identified in OWASP's Top 10 standard
Payment Card Industry Data Security Standard	Includes rules that find issues identified in PCI Data Security Standard
SecurityRules	General test configuration that finds security issues
Coverage	Used to generate the code coverage report.

Creating Custom Rules

Use RuleWizard to create custom rules. To use the rule in the Static Analysis Engine, it needs to be enabled in a test configuration and the custom rule file must be located in one of the following directories:

- `[INSTALL_DIR]\rules\user\`
- `[DOCUMENTS DIR]\Parasoft\[engine]\rules` where `[DOCUMENTS DIR]` refers to the "My Documents" directory in Windows

Compiler Configuration

You must configure Static Analysis Engine for specific C and C++ compilers, including compiler versions, before running the analysis. See “Compiler”, page 11, for basic information on configuring compilers. For a full list of supported compilers, see “Supported Compilers”, page 104.

Specifying Multiple Compilers

If multiple compilers need to be used during a single test run, they can be configured with the following extended syntax of the `-compiler / cpptest.compiler.family` option where `<COMPILER_COMMAND>` is the path to the original compiler executable and '`*`' is match-all character:

```
-compiler <COMPILER_ID>:<COMPILER_COMMAND>;<COMPILER_ID>:<COMPILER_COMMAND>;...
cpptest.compiler.family=<COMPILER_ID>:<COMPILER_COMMAND>;<COMPILER_ID>:<COMPILER_COMMAND>;...
```

Example

```
-compiler vc_9_0:*cl.exe;gcc_3_4:/usr/bin/gcc.exe
cpptest.compiler.family=vc_9_0:*cl.exe;gcc_3_4:/usr/bin/gcc.exe
```

Working with Custom Compiler Configurations

When working with custom compiler configurations, the configuration files should be copied into a directory defined with the `cpptest.compiler.dir.user` configuration entry:

```
cpptest.compiler.dir.user=/home/custom_compilers
```

Each custom compiler should be located in a dedicated subdirectory with unique name:

```
/home/custom_compilers (cpptest.compiler.dir.user configuration entry)
  custom_gcc
    c.psrc
    cpp.psrc
    gui.properties
  custom_iar
    c.psrc
    cpp.psrc
    gui.properties
```

Defining Input Scope

The input scope defines the C and C++ source files to test with Static Analysis Engine. The input scope also provides the full set of information about compiler options and environment, so Static Analysis Engine can re-create the original build environment to provide accurate test results. See “Compiler”, page 11, for information about defining compilers.

Analyzing a Single File

See “Analyzing a Single File”, page 11, for instructions.

Analyzing a Makefile-based Project

See “Analyzing a Makefile-based Project”, page 12, for instructions.

Analyzing Code Using Existing Build Data

Only the source files defined in the build data file will be analyzed. Header files included by the source files will be excluded from analysis. See the following sections for additional information:

- “About Build Data Files”, page 12, describes the concept of the .bdf and how to create it
- “Re-analyzing a Project without Re-building”, page 14, describes the steps for using the .bdf for analysis
- “Defining Source File Structures (Modules)”, page 22, desribes how to broaden the scope of files tested, including header files

Defining Source File Structures (Modules)

Static Analysis Engine treats the input scope as a set of unrelated source files. Defining modules allows you to introduce a well-defined source file structure and add additional files, such as header files, into the Input Scope.

Modules are defined by specifying its name and the root directory. All tested files located in the root directory or its sub-directories will belong to the module. All header files located in the root directory or its sub-directories that are included by the tested source files will also belong to the module and be analyzed with the source files.

For all files from the module, a "module-relative path" will be available. A project-relative path is computed as a relative path from the module root to the actual file location. In most cases, module-relative paths are independent from machines, so the test results can be easily shared across different machines.

Example of Module Structure

The first block of code describes a simple directory/file structure. In the second block of code, the relationships between the files and module root directory are described, as well as which files will be analyzed:

```
/home/devel_1/project/src/foo.cpp          tested file defined in bdf will be analyzed

/home/devel_1/project/includes/foo.h       #included by foo.cpp

/home/devel_1/project/includes/other.h     not #included by foo.cpp
```

```
/home/devel_1/common/common.h              #included by foo.cpp
```

Assuming module *MyApp* is defined with `/home/devel_1/project` root location, the following files will be tested as part of the module:

```
/home/devel_1/project/src/foo.cpp          belongs to MyApp as MyApp/src/foo.cpp; will be
                                           analyzed

/home/devel_1/project/includes/foo.h       belongs to MyApp as MyApp/includes/foo.h; will be
                                           analyzed

/home/devel_1/project/includes/other.h     not #included; will not be analyzed

/home/devel_1/common/common.h              does not belong to MyApp; will not be analyzed
```

Defining a Basic Module Structure

Use the `-`[<MODULE_NAME>=]<MODULE_ROOT_LOCATION> switch to define a module. If the name is unspecified, the name of the root directory will be used:

```
-module MyApp=/home/devel_1/project
-module /home/devel_1/project
-module MyModule=../projects/module1
-module .
```

Alternatively, module structures can be defined in a custom configuration file using the `cpptest.scope.module.<MODULE_NAME>=<MODULE_ROOT_LOCATION>` property:

```
cpptest.scope.module.MyApp=/home/devel_1/project
cpptest.scope.module.MyModule=../projects/module1
```

Defining a Module with Multiple Root Locations

Add a logical path to the module name that points to the appropriate root location to define multiple, non-overlapping locations:

```
-module MyApp/module1=/home/devel_1/project -module MyApp/module2=/home/external/
module2/src

cpptest.scope.module.MyApp/module1=/home/devel_1/project
cpptest.scope.module.MyApp/module2=/home/external/module2/src
```

Fine-tuning the Input Scope

Use the `-resource` switch to specify a file or set of files for testing.

```
-resource /home/cpptest/examples/ATM/ATM.cxx
-resource /home/cpptest/examples/ATM
-resource ATM.cxx
```

You can specify the following resources in the path:

- File path (only selected file will be tested)

- Directory path (only files from selected directory will be tested)
- File name (only files with selected name will be tested)

Use the `-include` and `-exclude` switches to apply additional filters to the scope.

- `-include` instructs Static Analysis Engine to test only the files that match the file system path; all other files are skipped.
- `-exclude` instructs Static Analysis Engine to test all files except for those that match the file system path.

If both switches are specified, then all files that match `-include`, but not those that match `-exclude` patterns are tested.

The `-include` and `-exclude` switches accept an absolute path to a file, with asterisk (`*`) as an accepted wildcard. .

```
-include /home/project/src/ATM.cxx
-include /home/project/CustomIncludes.lst
-exclude /home/project/src/*.cxx
-exclude /home/project/CustomExcludes.lst
```

You can specify a file system path to a list file (`*.lst`) to include or exclude files in bulk. Each item in the `*.lst` file is treated as a separate entry.

Configuring Authorship

You can configure DTP Engines to collect authorship data during analysis to facilitate task assignment. The data can be sent to the DTP server where additional analysis components, such as the Process Intelligence Engine (PIE), can be leveraged to facilitate defect remediation and development optimization.

You can configure DTP Engines to assign authorship based on information from source control, XML files that directly map sources to authors, and/or the current local user.

About Authorship Configuration Priority

Authorship priority is determined by reading the settings in the .properties configuration file from top to bottom. If multiple authorship sources are used, the following order of precedence is used:

1. information from source control
2. XML map file
3. current user

If one of the selected options does not determine an author, Authorship will be determined based on the next option selected. If an author cannot be determined, the user is set as "unknown". Likewise, if none of these options is selected, the user is set as "unknown."

Configuring How Authorship is Computed

Edit the `cpptestcli.properties` configuration file to specify how authorship is determined:

```
scope.local=[true or false]
scope.scontrol=[true or false]
scope.xmlmap=[true or false]
```

Additional Authorship Configurations

By default, author names are case-sensitive, but you can disable case sensitivity:

```
authors.ignore.case=true
```

You can set the user name, email, and full name for a user with the `authors.user[identifyer]` setting. For example:

```
authors.user1=john,john.doe@company.com,John Doe
```

If a user is no longer on team or must transfer authorship to another user, you can use the `authors.mapping[x,y]` setting:

```
authors.mapping1=old_user,new_user
```

If you are transferring authorship between users, the author-to-author mapping information can be stored locally or in an a shared XML map file:

```
authors.mappings.location=[local or shared]
```

If the mapping file is shared, you must specify the location of the shared XML file:

```
authors.shared.path=[path to file]
```

Creating Authorship XML Map Files

The `<authorship>` element contains indicates the beginning of the mapping information.

The `<file />` element is placed inside the `<authorship>` element and takes two properties, `author` and `path` to map users to files or sets of files:

```
<?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE authorship (View Source for full doctype...)>
    <authorship>
      <!-- assigns all files named: "foo/src/SomeClass.java" to "author1" -->
      <file author="author1" path="foo/src/SomeClass.java" />
```

You can use wildcards to map authors to sets of files. The following table contains examples:

Wildcard Expression	Description
`?oo/src/Foo.c`	Assigns all files that have names starting with any character (except /) and ends with "oo/src/"
`**.cs`	Assigns all *.cs files in any directory
`**/src/**`	Assigns every file whose path has a folder named "src"
`src/**`	Assigns all files located in directory "src"
`src/**/Test*`	Assigns all files in directory "src" whose name starts with "Test" (e.g., "src/some/other/dir/TestFile.c")

Mapping order matters. The mapping file is read from top to bottom, so beginning with the most specific mapping ensures that authorship will map to the correct files.

Suppressing Violations

Suppressions prevent DTP Engines from reporting additional occurrences of a specific static analysis task (multiple tasks might be reported for a single rule). Suppressions are useful when you want to follow a rule, but do not want to receive repeated messages about your intentional rule violations. If you do not want to receive error messages for any violations of a specific rule, disable the rule in the test configuration.

If you are using DTP Engines in an IDE, you can define suppressions using the GUI (see the DTP Plugin documentation for your IDE for details), otherwise suppressions are defined in the source code using the following syntax.

Line Suppression

```
<suppression keyword> [<rule category> | <rule category> . <rule id> | <rule
category > - <rule severity> | ALL ] <suppression comment>
```

Line Suppression Examples

```
// parasoft-suppress CODSTA "suppress all rules in category CODSTA"


// parasoft-suppress CODSTA.NEA "suppress rule CODSTA.NEA"


// parasoft-suppress CODSTA-1 "suppress all rules in category CODSTA with severity
level 1"


// parasoft-suppress ALL "suppress all rules"


// parasoft-suppress CODSTA FORMAT.MCH JAVADOC-3 "suppress all rules in category
CODSTA and rule FORMAT.MCH and all rules in category JAVADOC with severity level 3"
```

Block Suppression

```
<begin suppression keyword> [<rule category> | <rule category> . <rule id> | <rule
category > - <rule severity> | ALL ] <suppression comment>

 ..... source code block .....

<end suppression keyword> [<rule category> | <rule category> . <rule id> | <rule
category > - <rule severity> | ALL ] <suppression comment>
```

Block Suppression Examples

```
// parasoft-begin-suppress CODSTA "begin suppress all rules in category CODSTA"
.....
// parasoft-end-suppress CODSTA "end suppress all rules in category CODSTA"


// parasoft-begin-suppress CODSTA.NEA "begin suppress rule CODSTA.NEA"
.....
```

```
// parasoft-end-suppress CODSTA.NEA "end suppress rule CODSTA.NEA"


// parasoft-begin-suppress CODSTA-1 "begin suppress all rules in category CODSTA
with severity level 1"
......
// parasoft-end-suppress CODSTA-1 "end suppress all rules in category CODSTA with
severity level 1"


//parasoft-begin-suppress ALL "begin suppress all rules"
.....
// parasoft-end-suppress ALL "end suppress all rules"


// parasoft-begin-suppress CODSTA FORMAT.MCH "begin suppress all rules in category
CODSTA and rule FORMAT.MCH"
.....
// parasoft-end-suppress CODSTA FORMAT.MCH "end suppress all rules in category COD-
STA and rule FORMAT.MCH"


// parasoft-begin-suppress CODSTA "begin suppress all rules in category CODSTA"
.....
// parasoft-end-suppress CODSTA-1 "end suppress all rules in category CODSTA with
severity level 1; however rules with severity level 2-5 in category CODSTA are still
suppressed."
.....
// parasoft-end-suppress CODSTA "end suppress all rules in category CODSTA"


// parasoft-begin-suppress ALL "begin suppress all rules"
.....
// parasoft-end-suppress CODSTA FORMAT-1 "end suppress all rules in category CODSTA
and all rules in category FORMAT with severity level 1; however, others rules in COD-
STA and FORMAT-1 are still suppressed."
.....
// parasoft-end-suppress ALL "end suppress all rules"


//parasoft-begin-suppress ALL "begin suppress all rules, since no end suppression
comment, all rules will be suppressed starting from this line"
```

Flow Analysis

Flow Analysis is a type of static analysis technology that uses several analysis techniques, including simulation of application execution paths, to identify paths that could trigger runtime defects. Defects detected include use of uninitialized memory, null pointer dereferencing, division by zero, memory and resource leaks.

Since this analysis involves identifying and tracing complex paths, it exposes bugs that typically evade static code analysis and unit testing, and would be difficult to find through manual testing or inspection.

Flow Analysis' ability to expose bugs without executing code is especially valuable for users with legacy code bases and embedded code (where runtime detection of such errors is not effective or possible).

Run one of the Flow Analysis test configurations during analysis to execute flow analysis rules:

```
builtin://Flow Analysis Fast
builtin://Flow Analysis Standard
builtin://Flow Analysis Aggressive
```

Configuring Depth of Flow Analysis

Flow Analysis engine builds paths through the analyzed code to detect different kinds of problems. Since the analysis of all possible paths that span through the whole application may be infeasible, you can set up the desired level of depth of analysis. A deeper analysis will result in more findings, but the performance will be slower and the memory consumption will increase slightly.

You can specify the depth of analysis by using the test configuration interface in DTP. Go to **Report Center> Test Configurations> Static Analysis> Flow Analysis Advanced Settings> Performance> Depth of analysis** and choose one of the following options by selecting a radio button:

- **Shallowest (fastest)**: Finds only the most obvious problems in the source code. It is limited to cases where the cause of the problem is located close to the code where the problem occurs. The execution paths of violations found by this type of analysis normally span several lines of code in a single function. Only rarely will they span more than 3 function calls.
- **Shallow (fast):** Like the "Shallowest" analysis type, finds only the most obvious problems in the source code. However, it produces a greater overall number of findings and allows for examination of somewhat longer execution paths.
- **Standard**: Finds many complicated problems with execution paths containing tens of elements. Standard analysis goes beyond shallow analysis and also looks for more complicated problems, which can occur because of bad flow in a single function or due to improper interaction between different functions in different parts of the analyzed project. Violations found by this type of analysis often reveal non-trivial bugs in the analyzed source code and often span tens of lines of code.
- **Deep (slow)**: Allows for detection of a greater number of problems of the same complexity and nature as those defined for "Standard" depth. This type of analysis is slower than the standard one.
- **Thorough (slowest):** Finds more complicated problems. This type of analysis will perform a thorough scan of the code base; this requires more time, but will uncover many very complicated problems whose violation paths can span more than a hundred lines of code in different parts of the scanned application. This option is recommended for nightly runs.

The depth of Flow Analysis is set to **Standard** by default.

Setting Timeout Strategy

Apart from the depth of analysis, Flow Analysis engine uses an additional timeout guard to ensure the analysis completes within a reasonable time. An appropriate strategy can be set by using the test configuration interface in DTP. Go to **Report Center> Test Configurations> Static Analysis> Flow Analysis Advanced Settings> Performance> Strategy for Timeouts** and choose one of the following options by selecting a radio button:

- **time**: Analysis of the given hotspot is stopped after spending the defined amount of time on it. Note: in some cases, using this option can result in a slightly unstable number of violations being reported.
- **instructions**: Analysis of the given hotspot is stopped after executing the defined number of Flow Analysis engine instructions.Note: to determine the proper number of instructions to be set up for your environment, review information about timeouts in the Setup Problems section of the generated report.
- **off**: No timeout. Note: using this option may require significantly more time to finish the analysis.

The default timeout option is **time** set to 60 seconds. To get information about the Flow Analysis timeouts that occurred during the analysis, review the Setup Problems section of the report generated after the analysis.

Running Flow Analysis in Incremental Mode

By default, Flow Analysis performs a complete analysis of the scope it is run on. This can take considerable time when running on large code bases.

The most common way of performing Flow Analysis analysis is to run nightly tests on a single code base that changes slightly from day to day. Flow Analysis's incremental analysis mode is designed to reduce the time required to run analysis in this typical scenario. With incremental analysis mode, Analysis memorizes important analysis data during the initial run, then reuses it during the subsequent runs—rerunning analysis only for parts of the code that have been modified or are tightly connected to the modified code.

When using incremental analysis, remember that:

- The initial run of Flow Analysis may be slightly slower than running without incremental analysis. This is because Flow Analysis in addition to performing a complete analysis of the code base, Flow Analysis saves data to be reused in subsequent runs.

Disk space is required to store the necessary data.

Incremental analysis options control the incremental analysis feature. Available options are:

- **Enable incremental analysis:** Determines whether incremental analysis is used.
- **Compact incremental caches every [days]:** Determines how often compactization of incremental caches is run. Incremental analysis is optimized for speed; although Flow Analysis strives to always keep cache sizes small and remove unnecessary data, source code changes may result in these caches containing some data that will no longer be used. Compactization, which is run regularly as defined by this parameter, removes all outdated data. More precisely, if the time that has elapsed since the previous compactization is greater than the number of days specified for this option, compactization is performed immediately after the incremental run of Flow Analysis

Running Flow Analysis with Swapping of Analysis

Data Enabled

In this mode, analysis data is written to disk. Swapping of analysis data uses the same persistent storage and is done in a similar process as incremental analysis. If analysis is run on a large project, the analysis data that represents a semantical model of the analyzed source code may consume all the memory available for running Flow Analysis. If this occurs, Flow Analysis will remove from memory parts of the analysis data that are not currently necessary and reread it from disk later.

In general, we recommend running C++test in a large JVM heap configured with the Xmx JVM option. This is to minimize swapping, which results in greater performance. If sufficient memory is available, swapping of analysis data may be disabled, which may speed up code analysis. You can enable or disable the mode by using the test configuration interface in DTP:

Enable swapping of analysis data to disk: Enabled by default. If this option is disabled, it may result in faster analysis, if you are running Flow Analysis analysis on small to moderate size projects that do not require a lot of memory or when plenty of memory is available (for example, for 64-bit systems).

Configuring Verbosity of Flow Analysis

You can configure the following options by using the test configuration interface in DTP:

- **Do not report violations when cause cannot be shown:** Determines whether Flow Analysis reports violations where causes cannot be shown.

  Some Flow Analysis rules require that Flow Analysis checks all the possible paths leading to a certain point and verifies that a certain condition is met for all those paths. In such cases, a violation is associated with a set of paths (whereas in simple cases, a violation is represented by only one path). All of the paths in such a violation end with the violation point common to all the paths in the violation. However, different paths may start at different points in code. The beginning of each path is a violation cause (a point in code which stipulates a violation of a certain condition later in the code at the violation point). If a multipath violation's different paths have different causes, Flow Analysis will show only the violation point (and not the violation causes).

  Violations containing only the violation point may be difficult to understand (compared to regular cases where Flow Analysis shows complete paths starting from violation causes and leading to violation points). That’s why we provide an option to hide violations where the cause cannot be shown.
- **Do not show more than one violation per point:** Restricts reporting to one violation (for a single rule) per violation point. For example, one violation will be reported when Flow Analysis detects a potential null dereference with multiple sources of the null value. When verbosity is set to this level, Flow Analysis performance is somewhat faster.

Specifying Terminating Functions

You can define functions that terminate application execution. C/C++ developers sometimes use functions that terminate application execution in the event of a fatal error from which recovery is impossible. Examples of such functions are abort() and exit() from the standard library. Since Flow Analysis analyzes the application’s execution flow, it's important for it to be aware of the terminating functions that break execution flow by immediately stopping the application. Without such knowledge, Flow Analysis might make incorrect assumptions about the application flow.

Flow Analysis is aware of the terminating functions that are defined in the standard library. However, this is often not sufficient because non-standard libraries define their own terminating functions. If your application uses one of these functions, you should communicate that to Flow Analysis by specifying

the custom terminating function in the **Terminators** tab. Otherwise, Flow Analysis may produce false positives with execution paths passing by terminating functions.

Use the table listing supported APIs to enable/disable terminators from various APIs as well as to define your own APIs containing terminating functions. To add information about terminating functions from a certain library:

1. Click the **+** button in the top row of the table.
2. Click the arrow to expand the **Functions that terminate application execution t**ab.
3. Complete the table that opens; the table has the following columns:
    - **Enabled:** Specifies whether a built-in or custom terminator should be considered during the analysis.
    - **Fully qualified type name or namespace (wildcard):** Specifies the entity for a particular terminator. If this field is left empty, only the global function with the name specified in the 'Function name' column will be considered a terminator.
    - For example: The field value may be "myNameSpace::myClass" if the terminator is declared in 'myClass' coming from the 'myNameSpace' name space. Or, it may be "myNameSpace" if it is not declared in a type, but belongs only to the 'myNameSpace'.
    - **Function name (wildcard):** Specifies the name of the terminating function.
    - **+ definitions in subclasses:** Indicates whether the terminating function definitions in subclasses should be considered terminating functions as well. This applies to both instance and non-instance functions, and makes sense only if its fully qualified type name is specified.

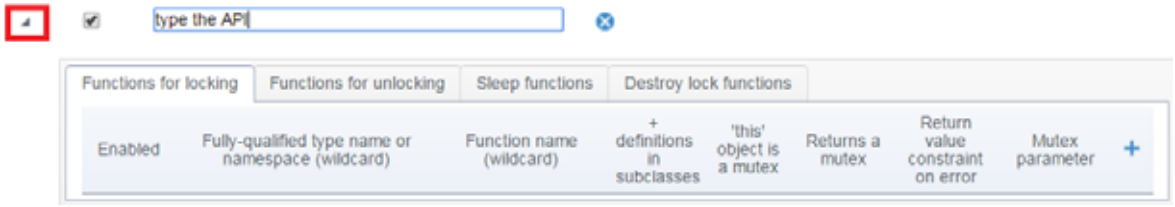
Specifying Multithreading Options

The **Multitheading** tab allows you to define functions for synchronization between threads as well as to activate/deactivate known multithreading functions. The information defined here affects the behavior of rules from the BD.TRS (Threads and Synchronization) category. These rules will check all the functions that are defined and activated on this tab.

Use the table that lists supported APIs to enable/disable synchronization functions from various APIs as well as to define your own APIs containing synchronization functions. To add information about synchronization functions from a certain library:

1. Click the **+** button in the top row of the table.
2. Type the name of the library in the **API** field.
3. Click the arrow to expand the tabs and complete the tables to define the following types of functions (details about completing the tables are provided below):
    - Functions for locking (for instance, obtaining a mutex)
    - Functions for unlocking (for instance, releasing a mutex)
    - Sleep functions

- Destroy lock functions



Functions for locking

Complete the table with the following information:

- **Enabled**: specifies whether the locking function should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the locking function. '*' can be used to denote any number of any symbols.
- **+ definitions in subclasses: a check box that indicates** whether the definitions (of the function with the given name) in subclasses should be considered locking functions as well. Note that this applies to both instance and static functions.
- **"this" object is a mutex**: a check box that indicates that the function locks a mutex in the object on which the function is called.
- **Returns a mutex:** a check box that indicates that the function returns a mutex.
- **Return value constraint on error:** specifies a return value constraint in case of allocation failure if a resource allocator returns an integral value. Enter the condition in the following format: `<comparison operator><integer value>`. For example, if the function returns non-zero value on failure, enter "!=0" (without quotes) into the field. If return code on error is -1, type "==-1" there. In addition to "!=" and "==", you can use the following operators for specifying error conditions: ">", ">=", "<", and "<=".
- **Mutex parameter**: specifies that the function locks a mutex in one of its parameters.

Functions for unlocking

Complete the table with the following information:

- **Enabled**: specifies whether the unlocking function should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the unlocking function. '*' can be used to denote any number of any symbols.
- **+ definitions in subclasses: a check box that indicates** whether the definitions (of the function with the given name) in subclasses should be considered unlocking functions as well. Note that this applies to both instance and static functions.
- **"this" object is a mutex**: a check box that indicates that a mutex in the object on which the function is called is unlocked

- **Mutex parameter:** specifies that a mutex in one of the parameters is unlocked.

Sleep functions

Complete the table with the following information:

- **Enabled**: specifies whether the sleep function should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the sleep function. '*' can be used to denote any number of any symbols.
- **+ definitions in subclasses: a check box that indicates** whether the definitions (of the function with the given name) in subclasses should be considered sleep functions as well. Note that this applies to both instance and static functions.

Destroy lock functions

Complete the table with the following information:

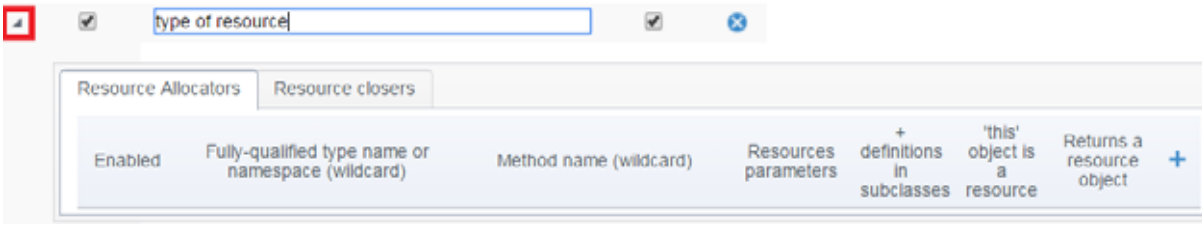
- **Enabled**: specifies whether the lock-destroying function should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the lock-destroying function. '*' can be used to denote any number of any symbols.
- **+ definitions in subclasses: a check box that indicates** whether the definitions (of the function with the given name) in subclasses should be considered lock-destroying functions as well. Note that this applies to both instance and static functions.
- **"this" object is a mutex: a check box that indicates that a mutex in the object on which the function is called is destroyed.**
- **Mutex parameter:** specifies that a mutex in one of the parameters is destroyed.

Specifying Resources

The **Resources** tab allows you to define which resources the BD.RES category (Resources) rules should check. These rules check for the correct usage of all resources that are defined and enabled on this tab.

1. Specify the **Type of resource.**
2. Select the **Enabled** checkbox.
3. If appropriate/desired, disable the **Do not report leaks at termination** option.

4. Click the arrow to expand the **Resource Allocators** and **Resource Closers** tabs and complete the tables that open with the information about allocators and closers. Details about completing these tabs are provided below. .



Configuring Resource Allocators

The **Resource allocators** table can be completed with the descriptors of functions that can produce a resource. The table has the following columns:

- **Enabled**: specifies whether the allocator should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the allocating function. '*' can be used to denote any number of any symbols.
- **Resource parameters**: specifies that the function allocates a resource in one or more of its parameters. In this case, either specify a 1-based number of the parameter that is allocated by the function, or use '*' to denote that all of the parameters are allocated.
- **+ definitions in subclasses: a check box that indicates** whether the definitions (of functions with the given name) in subclasses should be considered allocators as well. Note that this applies to both instance and static functions.
- **"this" object is a resource**: a check box that indicates that the function allocates a resource in the object on which the function is called.
- **Returns a resource object**: a check box that inticates that a the function returns an allocated resource.
- **Return value constraint on error:** specifies a return value constraint in case of allocation failure if a resource allocator returns an integral value. Enter the condition in the following format: `<comparison operator><integer value>`. For example, if the function returns non-zero value on failure, enter "!=0" (without quotes) into the field. If return code on error is -1, type "==-1" there. In addition to "!=" and "==", you can use the following operators for specifying error conditions: ">", ">=", "<", and "<=".

It is common that allocation functions return an error code to indicate allocation failure. When an allocation function returns a pointer to a resource, a NULL pointer normally indicates an allocation failure. When Flow Analysis is looking for resource leaks, it needs to understand if allocation succeeded or failed; this helps it report only missing calls to deallocation functions on paths where allocation actually occurred. In cases where a resource allocator function returns a pointer to a resource, Flow Analysis assumes that the resource is successfully allocated if the pointer is not NULL.

Configuring Resource Closers

The **Resource closers** table can be completed with the descriptors of functions that can close a resource. The table has the following columns:

- **Enabled**: specifies whether the closer should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the function is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Function name (wildcard)**: the name of the closing function. '*' can be used to denote any number of any symbols.
- **+ definitions in subclasses: a check box that indicates** whether the definitions (of functions with the given name) in subclasses should be considered closers as well. Note that this applies to both instance and static functions.
- **"this" object is a resource**: a check box that indicates that a resource in the object on which the function is called is closed.
- **Resource parameters**: specifies that a resource in one or more of its parameters is closed. In this case, either specify a 1-based number of the parameter that is closed by the function, or use '*' to denote that all of the parameters are allocated.

Reusing Flow Analysis Data for Desktop Analysis

One way to improve desktop performance with Flow Analysis is to reuse the server analysis data on the desktop. To do this, you need to define a mapping that allows Flow Analysis engine to match server file paths with corresponding desktop file paths.

Additionally, you can reuse data to run the analysis on a small scope (for example, one file) and build paths that include methods defined in files outside the defined scope of analysis, provided that these files have been analyzed.

Please, contact Parasoft Support for more information on how to use this functionality.

Compiler-specific Settings

You can configure advancesd compiler-specific settings by using the test configuration interface in DTP.

- **Internal representation of the "errno" value:** The Standard defines erno to be a modifiable lvalue of type int. It is unspecified whether errno is a macro or an identifier declared with an external linkage. Implementations may use the global variable "errno" or "__errno", or apply the "(*errno_function())" pattern with different names of the called functions. This option allows you to specify the names of these variables and functions with regular expressions.
    - **Function name pattern**: The name of the function that is called when the "errno" value is used. The name must be specified with regular expressions.
    - **Variable name pattern**: The name of the variable that is called when the "errno" value is used. The name must be specified with regular expressions.
- **Internal representation of the call to a function from the header <ctype.h>**: The Standard specifies several functions to be defined in the header <ctype.h>. Some implementations (e.g GNU GCC in the C mode) define these functions as macros that expand to the code which tests an element of the internal array against some flags. This can be either a global array or a pointer returned by a function. This option allows you to specify names of these variables and functions with regular expressions.
    - **Function name pattern:** The name of the function that is invoked internally instead of one of the functions from the header <ctype.h> (define with regular expressions). The name must be specified with regular expressions.

- **Variable name pattern**: The name of the variable that is used internally after a call to one of the functions from the header <ctype.h>. The name must be specified with regular expressions.

Variable name pattern: The name of the variable that is used internally after a call to one of the functions from the header <ctype.h>. The name must be specified with regular expressions.

Metrics Analysis

DTP Engines can compute several code metrics, such as code complexity, coupling between objects, and lack of cohesion, which can help you understand potential weak points in the code. Run the Metrics test configuration during analysis to execute metrics analysis rules:

```
builtin://Metrics
```

Metrics analysis is added to the HTML and XML report files generated by DTP Engines. See “Metrics Summary”, page 44, for information about reports.

Setting Metrics Thresholds

You can set upper and lower boundaries so that a static analysis violation is reported if a metric is calculated outside the specified value range. For example, if you want to restrict the number of logical lines in a project, you could configure the Metrics test configuration so that a violation is reported if the Number of Logical Lines metric exceeds the limit.

The Metrics test configuration shipped with DTP Engines includes default threshold values. There are some rules, such as Number of Files (METRIC.NOF), for which thresholds cannot be set.

Metric thresholds can be set using the following methods:

- By using the test configuration interface in DTP (see "Report Center> Test Configurations> Editing Test Configurations> Metrics Tab" in the Development Testing Platform user manual for details).
- By editing the test configuration using the interface in an IDE (see "Working with Test Configurations> Creating Custom Test Configurations" in the DTP Plugin manual for your IDE).
- By manually editing the test configuration file:
    1. Duplicate the built-in Metrics test configuration ([INSTALL]/configs/builtin) to the user configurations directory ([INSTALL]/configs/user)
    2. Open the duplicate configuration in an editor and set the `[METRIC.ID].ThresholdEnabled` property to `true`.
    3. Configure the lower and upper boundaries in the `[METRIC.ID].Threshold` property according to the following format:

        ```
        [METRIC.ID].Threshold=l [lower boundary value] g [upper boundary value]
        ```
    4. Save the test configuration and run the analysis using the custom metrics test configuration.

Code Duplicate Analysis

DTP Engines can check for duplicate code, which may indicate poor application design, as well as increase maintenance costs. During code duplication analysis, the code is parsed into smaller language elements (tokens). The tokens are analyzed according to a set of rules that specify what should be considered duplicate code. There are two types of rules for analyzing tokens:

- Simple rules for finding single token duplicates, e.g., string literals
- Complex rules for finding mulitple token duplicates, e.g., duplicate methods or statements

Run the Find Duplicated Code test configuration during analysis to execute code duplicates detection rules:

```
builtin://Find Duplicated Code
```

Using DTP Engines in an IDE

You can use DTP Engines within Eclipse or Visual Studio. Integrating with an IDE gives you a desktop interface for executing code analysis locally, viewing results, and leveraging the data and test configurations stored in DTP server. You can also import findings from DTP Server into your development environment.

This integration is achieved with the DTP Plugin for Eclipse or Visual Studio and the DTP Engine Plugin. See the appropriate DTP Plugin User Guide for installation, usage, and other details stored in the [INSTALL]/integration/[IDE]/ directory.

You can also leverage the IDE plug-ins to integrate the DTP Engines for C/C++ with the Wind River Workbench IDE. See the "DTP Plugin Wind River Integration" guide for more information.

Reporting

DTP Engines print results to the output console, as well as save an HTML report to the `[WORKING_DIR]/reports` directory by default. Data for the HTML report is stored in the directory as an XML file, which can be used for importing results into a supported Parasoft DTP Plugin for the IDE and Parasoft DTP Plugin for C/C++ (see "Parasoft DTP Plugin for [IDE] User's Guide" for additional information). For an overview of the HTML report structure, see "Viewing Reports", page 41.

If the engines are connected to DTP, reports are also sent to the server (see "Sending Results to Development Testing Platform (DTP) Server", page 47).

Specifying Report Output Location

You can use the `-report` switch during analysis to specify an output directory for reports.

```
cpptestcli -report /home/reports/html …
```

You can also use the `report.location` property to change the location of an HTML report.

```
report.location=<HTML_REPORT_LOCATION>
```

A simple .csv report can also be generated by enabling the CVS reporter. See "Generating a .csv Report", page 15.

Specifying Report Format

You can also generate a PDF report or a report using a custom extension to the specified directory by setting the `report.format` property. See "Report Settings", page 85, for additional information.

```
report.format=pdf
```

Viewing Reports

Open the report.html or report.pdf file saved to the working directory or location specified with the `-report` switch. Reports may contain different sections depending on the type of analysis, but the following sections are included in all static and flow analysis configurations.

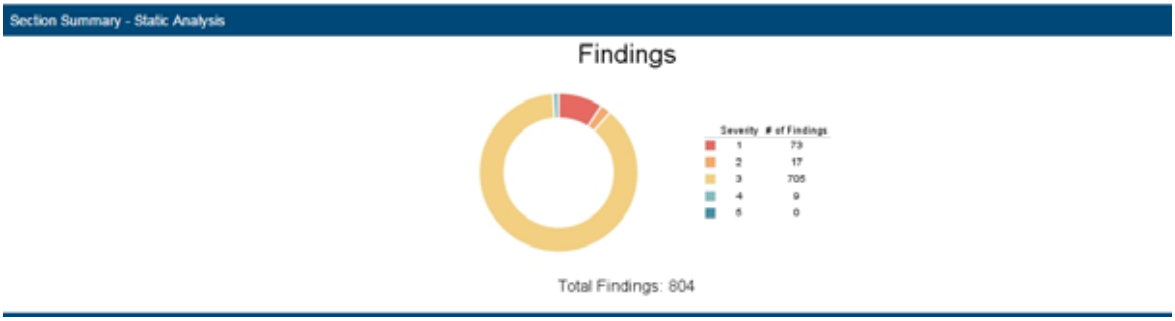
Header

The following information is included:

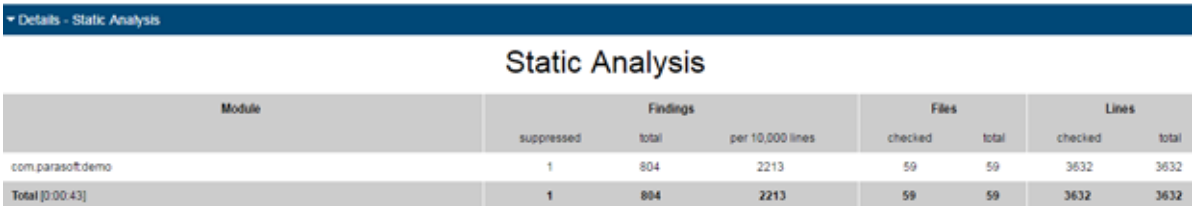
- Tool used for the analysis
- Build ID
- Test configuration
- Time stamp of the analysis
- Machine name and user name
- Session tag
- Project name
- Number of findings with the highest severity
- Number of failed tests

Static Analysis

The first part of the report covers the Static Analysis findings and is divided into two main sections.The first section is a summary which shows an overview of findings displayed as a pie chart. The colors indicate different severity types and their corresponding number of findings detected during static analysis.



The second section shows the details of static analysis findings. It starts with a table which includes static analysis results.



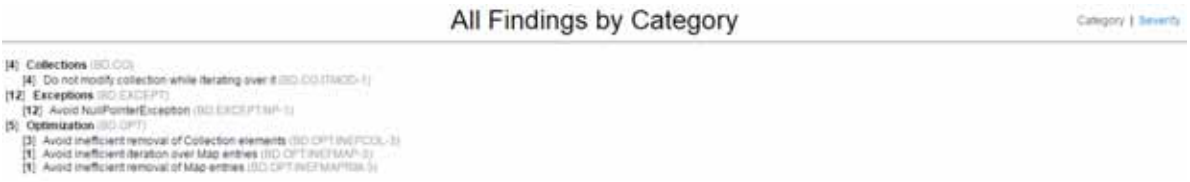
The following information is included:

- Name of module
- Number of suppressed rules
- Total number of findings
- Average number of findings per 10,000 lines
- Number of analyzed files
- Total number of files in the module
- Number of code lines analyzed
- Total number of code lines in the module

All Findings

The All Findings section displays the details of findings organized by category or severity. Click the **Severity** or **Category** link to toggle between views.

In category view, findings are reported by rule and grouped by category. A count of how many times each rule was violated in the scope of analysis is also shown.



In severity view, findings are reported and grouped by severity. A count of findings per severity is also included.

These sections are merged in PDF versions of the report.

Findings by Author

This section includes a table of authors associated with the analyzed code and a count of findings per each author. Findings are segmented into findings associated with suppressed rules and findings recommended for remediation. Click on an author link to view their finding details.

The details view includes the following information:

- File containing the finding and its location
- Violation message and rule
- Flow analysis reports also mark the cause of the violation (C), violation points (P), thrown exceptions (E), and important data flows (!)

Findings by File

You can navigate the analyzed code to the reported findings in the Findings by File section. Each node begins with a value that indicates the total number of findings in the node. The value in brackets shows the number of suppressed rules in the node. You can click nodes marked with a plus sign (+) to expand them. PDF versions of the reports are already fully expanded.

Active Rules

The rules enabled during analysis are listed in the Active Rules section.

Metrics Summary

If your test configuration includes metrics analysis, a metrics section will appear in the report. See “Metrics Analysis”, page 38, for additional information.

Metrics Summary

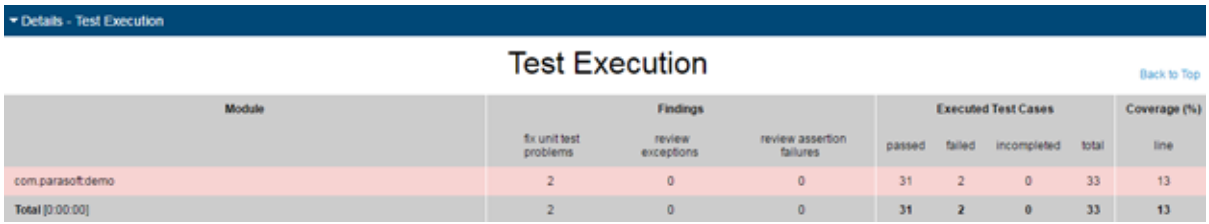
Expand All Collapse All Back to Top

Metric name	Number of items	Average	Std. Deviation	Maximum	Minimum
McCabe Cyclomatic Complexity (METRIC.CC)	**14**	**1,571**	**1,116**	**5**	**1**
ATM	14	1,571	1,116	5	1
Nested Blocks Depth (METRIC.NBD)	**14**	**0,286**	**0,452**	**1**	**0**
ATM	14	0,286	0,452	1	0
Number of Physical Lines in Files (METRIC.NOPLIF)	**4**	**31**	**15,652**	**52**	**14**
ATM	4	31	15,652	52	14
Number of Source Lines in Methods (METRIC.NOSLIM)	**14**	**6,5**	**2,897**	**13**	**3**
ATM	14	6,5	2,897	13	3

Test Execution

The second part of the report covers the Test Execution results and is divided into two sections.The first section is a summary which shows an overview of test failures and coverage displayed as pie charts.

The second section shows the details of test execution. It starts with a table which includes test execution results and coverage information.



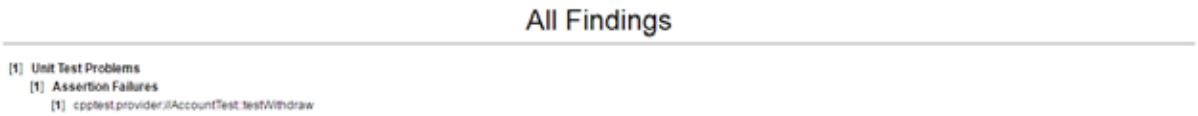
Module	Findings			Executed Test Cases				Coverage (%)
	fix unit test problems	review exceptions	review assertion failures	passed	failed	incompleted	total	line
com.parasoft.demo	2	0	0	31	2	0	33	13
Total [0:00:00]	2	0	0	**31**	**2**	**0**	**33**	**13**

The following information is included:

- Module name
- Number of unit test problems which need to be fixed
- Number of exceptions which need to be reviewed
- Number of assertion failures which need to be reviewed
- Number of unit tests successfully executed
- Number of unit tests failures
- Number of incomplete unit tests
- Total number of unit tests
- Line coverage expressed as percentage

All Findings

The All Findings section displays the details of all unit test problems detected during test execution.



Findings by Author

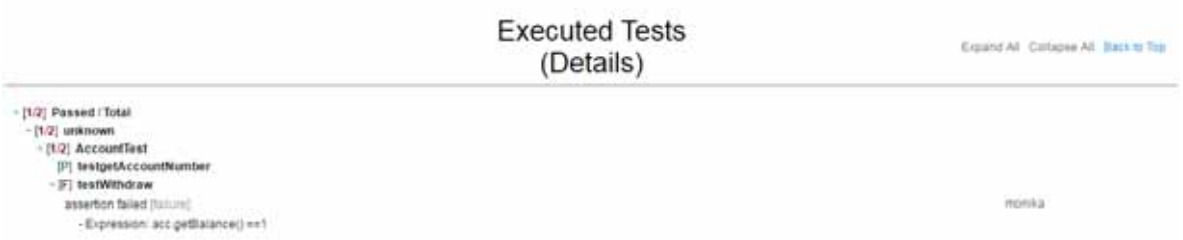
This section includes a table of authors associated with the analyzed code and shows the total number of findings for each author. Click on an author link to view their finding details.

The details view includes the following information:

- Finding location
- Test name
- Failure message

Executed Tests (Details)

You can view the findings in the Executed Tests (Details) section. The nodes where all the test passed are marked with "P" in square brackets. The nodes with test failures begin with a set of values in square brackets. The first value is a count of successfully passed tests and the second indicates the total number of tests executed in the node.The letter "F" indicates the final node where the test failed. You can click nodes marked with a plus sign (+) to expand them.



Coverage

This section shows the details of coverage collected during the test execution. Each node starts with a set of values. The first value shows coverage expressed as percentage. The second value is a count of

the number of lines in the node which were covered during the test execution. The third value indicates the total number of lines in the node. You can click nodes marked with a plus sign (+) to expand them.

Test Parameters

The arguments specified during analysis are shown in the Test Parameters section.

Sending Results to Development Testing Platform (DTP) Server

See “Connecting to DTP Server”, page 8, for information about configuring your connection to DTP Server. Use the `-publish` switch to report test results to DTP server.

```
cpptestcli -publish . . .
```

Publishing Source Code to DTP Server

By default, tested sources are sent to DTP when the report setting is enabled. This enables DTP to present source code associated with findings.

You can use the `report.dtp.publish.src` setting to disable the publishing of source code, restrict the depth of source code publishing, or enable source code publishing when sending reports to DTP Server is disabled. See “Settings Reference”, page 77, for additional information on DTP Engine settings.

The `report.dtp.publish.src` setting takes one of the following values:

- `off`: Code is not published to DTP server.
- `min`: Publishes minimal part of sources. Only source code that has no reference to source control is published.
- `full`: Publishes all sources associated with the specified scope. This is the default settings.

See the "Development Testing Platform User Guide" for additional information about viewing source code in DTP.

Publishing Sources to DTP Without Running Code Analysis

DTP Engines need to execute to send data to DTP Server, but you may want to send sources without running analysis.

1. Create an empty test configuration and save it to `[INSTALL_DIR]/configs/user` (see “Specifying Test Configurations”, page 17).
2. Run the configuration with appropriate `report.dtp.publish.src` setting.

Unit Test Connector

Unit Test Connector (UTC) allows you to run unit tests created in open source unit testing tools and report results to DTP. UTC for C/C++ currently ships with out-of-the-box support for the following unit testing tools:

- Google Test
- CppUnit
- CppUtest

Visit the Parasoft Marketplace (http://marketplace.parasoft.com) for additional unit test tool integrations.

Unit Test Connector consists of:

- a module that extracts and reports unit test case execution results
- a module which annotates code coverage results to associate test cases with code coverage to evaluate the quality of a particular unit test or collection of tests.

Integrating Unit Test Connector with a testing infrastructure requires minor modifications of the existing test harness code.

Google Test Connector

Unit Test Connector for Google Test can send test execution results to DTP server, as well as generate a local HTML report using the XML report created by the test framework.

Reporting Google Test Results

1. Add the following option to your normal Google Test test driver command:

```
--gtest_output=xml:<result_filename>.xml
```

   A Google Test result XML file is generated

2. Run the following test configuration and specify the Google Test result XML file as the input (see “Specifying Test Configurations”, page 17, for additional information about test configurations):

```
cpptestcli -config builtin://GoogleTest -input <result_filename>.xml -publish
```

   This will generate an HTML report, as well as send test results to DTP server if configured (see “Connecting to DTP Server”, page 8, for additional information).

UTE will perform the following actions:

- Associate the test results with the project name specified with the `dtp.project` property
- Associate test failures with the author specified with the `dtp.user` property

Improving the Test Case File Association

The Google Test result XML file provides limited information and may incorrectly associate passed test cases with the tested file. This may affect how test results are displayed in some DTP Test widgets.

Use the `RecordProperty` function inside a test case to force an association between a test case and a file. Use `cpptest_filename` as the attribute name and the filename as the second argument:

```
TEST(ExampleTestSuite, ExampleTestCase) {
    RecordProperty("cpptest_filename", __FILE__);
}
```

The `RecordProperty` function is provided by the Google Test framework.

Associating Tests with Code Coverage

You can automatically annotate code coverage results with test start/stop information (see “Annotating Results Stream with Test Start/Stop Information”, page 63). This will allow you to selectively analyze code coverage generated by a specific test or collection of tests.

1. Include a dedicated header file in the source file that contains Google Test main function:

```
#include "cpptest/gtest.h"
```

2. Add the following code to the main function after the InitGoogleTest() call and before tests execution:

```
CppTest_GoogleTestListener::Append();
```

   Alternatively, you can register the C++test listener using the GoogleTest API directly:

```
::testing::TestEventListeners& listeners = ::testing::UnitTest::GetInstance()->listeners();
listeners.Append(new CppTest_GoogleTestListener());
```

   For simple setups, your modified main function may resemble the following:

```
GTEST_API_ int main(int argc, char **argv)
{
  // Initializing GoogleTest
  testing::InitGoogleTest(&argc, argv);

  // Appending CppTest_GoogleTestListener
  CppTest_GoogleTestListener::Append();

  // Running tests
  return RUN_ALL_TESTS();
}
```

3. Use the `-I` option to specify the cpptest/gtest.h header file location when compiling the source file with the main function:

```
-I <Installation Directory>/runtime/include
```

4. Use Code Coverage Engine to build test executable (see “Code Coverage Engine”, page 56).

5. Run the test executable with the following option:

```
--gtest_output=xml:<result_filename>.xml
```

6. Run cpptestcli specifying the GoogleTest result .xml file and coverage log file as the inputs to generate a report:

```
cpptestcli -config builtin://GoogleTest -input <result_filename>.xml -input
cpptest_results.clog -publish
```

See “Reporting”, page 41 for details on configuring and publishing reports.

CppUnit and CppUtest Connector

Unit Test Connector for CppUnit and CppUtest can report test execution results to DTP server, as well as associate tests with code coverage. To integrate CppUnit or CppUtest with Unit Test Connector, you need to install a results listener and a coverage annotator into the existing CppUnit or CppUtest infrastructure.

The typical integration includes both the results listener and the coverage annotator, which give you complete information about tests results and coverage. You may choose to install only the results listener for lightweight tests or for comparing results with and without coverage.

Installing Unit Test Connectors into Test Setups

This section describes CppUnit and CppUtest setups with both the results listener and the coverage annotator. If you choose to install the results listener only, skip all the lines which mention the coverage annotator.

Installing Unit Test Connector into CppUnit Setup

1. Include a dedicated header file in the source file that contains the CppUnit main function.

```
#include "cpptest/extensions/cppunit/results_listener.h"
#include "cpptest/extensions/cppunit/coverage_annotator.h"
```

2. Install the results listener. The installation details will depend on the TestRunner class of the CppUnit framework you use to execute unit tests.

For simple setups, your modified main function may resemble the following:

```
     /* required header files */

     int main()
     {
        CPPUNIT_NS::Test *suite = CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest();
         CPPUNIT_NS::TextUi::TestRunner runner;
         runner.addTest( suite );

         CppTest_CppUnitResultsOutputter cpptestResListener;
         CppTest_CppUnitCoverageAnnotator cpptestCovAnnotator;

         runner.eventManager().addListener( &cpptestResListener );
         runner.eventManager().addListener( &cpptestCovAnnotator );

         runner.setOutputter( new CPPUNIT_NS::CompilerOutputter( &runner.result(),
std::cout));
         bool wasSucessful = runner.run();
         return wasSucessful ? 0 : 1;
```

Depending on the class that was used, installation of the modifications may differ. The examples below show installation for two classes of the CppUnit framework.

Installation for the CppUnit::TestRunner class:

```
     CppUnit::TestResult controller;
     CppTest_CppUnitResultsListener cpptestResListener;
     CppTest_CppUnitCoverageAnnotator cpptestCovAnnotator;
     controller.addListener( &cpptestResListener );
     controller.addListener( &cpptestCovAnnotator );

     CppUnit::TestRunner runner;
     runner.addTest(CppUnit::TestFactoryRegistry::getRegistry().makeTest() );
     runner.run( controller, testPath );
```

Installation for the CppUnit::TextTestRunner class:

```
     CppUnit::TextTestRunner runner;
     CppTest_CppUnitResultsListener cpptestResListener;
     CppTest_CppUnitCoverageAnnotator cpptestCovAnnotator;
     runner.eventManager().addListener(&cpptestResListener);
     runner.eventManager().addListener(&cpptestCovAnnotator);
```

3. Modify your build system configuration to specify the results_listener.h header file location with the `-I` option.

```
-I<C++test Installation Directory>/runtime/include
```

Installing Unit Test Connector into CppUtest Setup

1. Include a dedicated header file in the source file that contains the CppUtest main function.

```
#include "cpptest/extensions/cpputest/results_listener.h"
#include "cpptest/extensions/cpputest/coverage_annotator.h"
#include "cpptest/extensions/cpputest/test_runner.h"
```

2. Install the results listener. The installation requires registration of the coverage annotator and the results listener into the TestRegistry class. It also requires use of a C++test supplied TestRunner.

For simple setups, your modified main function may resemble the following:

```
     /* required header files */
int main()
{
    // Register C++Test cpputest plugins
    TestRegistry* registry = TestRegistry::getCurrentRegistry();
    TestPlugin* coverageAnnotator = new CppTest_CppUtestCoverageAnnotator();
    registry->installPlugin(coverageAnnotator);
    TestPlugin* resultsListener = new CppTest_CppUtestResultsListener();
    registry->installPlugin(resultsListener);
    // run the tests
    int result = CppTest_CppUtestTestRunner::RunAllTests(ac, av);
    delete coverageAnnotator;
    delete resultsListener;
    return result;
```

3. Modify your build system configuration to specify the results_listener.h header file location with the `-I` option.

```
-I<C++test Installation Directory>/runtime/include
```

Reporting CppUnit and CppUtest Test Results

The results listener can record unit test execution results and store the data in a file. By default, execution results are stored in the cpptest_results.utlog file in the current working directory. You can change the default file location by providing the path as an argument to the CppTest_CppUnitResultsListener (CppUnit) or CppTest_CppUtestResultsListener (CppUtest) constructor:

- For CppUnit:

```
CppTest_CppUnitResultsListener cpptestResListener("c:/myworkspace/cpptest_results.utlog");
```

- For CppUtest:

```
CppTest_CppUtestResultsListener cpptestResListener("c:/myworkspace/cpptest_results.utlog");
```

Alternatively, you can use the following definition to specify the location during the test harness build process:

```
-DCPPTEST_UT_LOG_FILE=\"c:/home/my_workspace/cpptest_results.utlog\"
```

If you choose to change the default file location, it is important to retain the `.utlog` file extension.

Once the test executable with the results listener is built, you can execute the scheduled unit tests. In standard setups, the cpptest_results.utlog file generated during test execution will be placed into the directory that contains the executable. If you modified the file path, the file will be created in the specified location.

Unit test execution results can be published to DTP server or a local report can be generated with the following command line:

```
cpptestcli -config "builtin://Unit Testing" -input <result_filename>.utlog -publish -report
local_report
```

See “Reporting”, page 41 for details on configuring and publishing reports.

Unit test execution results are typically combined with code coverage results.

Associating Tests with Code Coverage

You can automatically annotate code coverage results with test start/stop information (See “Annotating Results Stream with Test Start/Stop Information”, page 63). This will allow you to selectively analyze code coverage generated by a specific test or a collection of tests. Associating tests with code coverage requires the installation of both the coverage results annotator and the results listener, along with Code Coverage Engine.

The coverage annotator adds special markers into the code coverage results stream. These markers delimit coverage results for each test case. The coverage annotator adds the markers to the coverage results file, which is managed by Code Coverage Engine. The annotator does not require any input parameters.

Once the test executable with the results listener is installed, you can execute the scheduled unit tests. In standard setups, the cpptest_results.clog file and the cpptest_results.utlog file generated during test execution will be placed into the current working directory.

Run the following command to generate a local report and enable the selective analysis of code coverage on DTP server:

```
cpptestcli -config "builtin://Unit Testing" -input cpptest_results.utlog -input
cpptest_results.clog -publish -report local_report
```

Reporting Assertions

By default, UTE does not show details about specific test assertion failures in the generated HTML report. Use the following properties to display test assertion failures in the report:

- `report.contexts_details=true`
- `report.testcases_details=true`

Tagging Unique Test Runs

Use the `session.tag` property to define a tag that can be assigned to results from a specific test run. The tag is a unique identifier for the analysis process on a specific module. Using the same session tag overwrites data with results from the most recent run. By default, the tag is set to the name of the executed test configuration.

```
session.tag=[name]
```

Code Coverage Engine

The DTP Code Coverage Engine (CCE) collects coverage information from unit testing, functional testing, as well as other kinds of application execution. CCE supports a range of coverage metrics and can be used for native and cross-application development. Collecting coverage information with CCE involves three phases:

- Instrumenting and building source code
- Executing instrumented code
- Generating a report

The following instructions provide a general description of integrating with a build system. For details on integrating with Make- or MSBuild-based projects, see the following chapters: “Integrating with Make-based Build Systems”, page 59, “Integrating with MSBuild”, page 61.

Instrumenting and Building Source Code

Three steps are performed during this phase:

1. Source code is instrumented
2. Instrumented source code is compiled to the object file
3. All instrumented and non-instrumented objects are linked to the code coverage tool library, as well as any additional libraries required to form the final testable binary artifact

These steps are typically performed during the build process and require that the coverage tool be integrated with the user build system.

The `cpptestcc` tool instruments the source code and compiles it into the object file. The `cpptestcc` is designed to be used as a compiler prefix in compilation command lines.

Original compilation command line:

```
cc -I app/includes -D defines -c source.cpp
```

Coverage mode compilation command line:

```
cpptestcc -compiler gcc_3_4 -line-coverage -workspace /home/test/proj/cov
-- cc -I app/includes -D defines -c source.cpp
```

When the coverage mode compilation command line is executed, `cpptestcc` performs the following operations:

- Compilation command line is analyzed to extract information about source files
- All detected source files are parsed and instrumented for coverage metrics
- Instrumented files are reconstructed in the specified location using the `-workspace` switch; additional information about the code structure used during report generation is also stored
- Compilation command line is modified and the original source files are substituted with instrumented versions
- Compilation command line is executed, object(s) files are created in the same location as in case of original command line

You can add the `[INSTALL_DIR]/bin` directory to your PATH variable so that you do not have to use full paths when specifying `cpptestcc` command. All examples in this documentation assume this has been done.

The following pattern describes the syntax for coverage instrumentation:

```
cpptestcc -compiler <compiler configuration> <coverage metric specification> -workspace <workspace directory> -- <compilation command line>
```

- `<compiler configuration>` refers to a supported compiler configuration, e.g., `gcc_3_4`. See “Supported Compilers”, page 104, for a list of supported compilers.
- `<coverage metric specification>` refers to a supported coverage metric, e.g., line-coverage. For list of supported coverage metrics, see “Command Line Reference for cpptestcc”, page 72.
- The `cpptestcc` command line is separated from compiler command line with the `--` separator.

Linking Instrumented Code

The original linker command must be modified to include the additional library required by the code coverage instrumentation. The following example shows how this is typically accomplished:

Original compilation command line:

```
lxx -L app/lib app/source.o -lsomelib -o app.exe
```

Coverage mode compilation command line:

```
lxx -L app/lib app/source.o somelib.lib <coverage
tool>/runtime/lib/cpptest.lib -o app.exe
```

Executing Instrumented Code

Details of the execution environment depend on the application specifics, but the coverage tool imposes the following limited dependencies on execution:

- If the coverage tool library was linked as a shared (dynamic-load) library, then you must ensure that the library can be loaded when instrumented application is started. On Windows, this typically requires adding `[INSTALL_DIR]/bin` directory to the PATH environment variable. On Linux systems, add `[INSTALL_DIR]/runtime/lib` to the LD_LIBRARY_PATH variable.
- If the coverage results transport channel was modified, all requirements resulting from the modification have to be fulfilled. For example, if results are sent through TCP/IP socket or rs232 then appropriate listening agents need to be started before the instrumented application executes.

After the instrumented application finishes execution, the collected results must be stored in the file that will be used for report generation.

Generating Reports

Final coverage report is generated from two types of information:

- Code structure information generated by `cpptestcc` during build process (stored in workspace)
- Coverage results achieved from instrumented code execution

Coverage report can be generated in an HTML format or sent to DTP Server. The following example shows the command for generating a report:

```
cpptestcli -config builtin://Coverage -input cpptest_results.clog -workspace /home/test/
proj/cov
```

In order to properly merge coverage data in DTP, you must specify one or more coverage image tags in the command line or .properties settings file. The coverage image(s) is automatically sent to the connected DTP server where it can be associated with a filter.

You can specify a set of up to three tags that can be used to create coverage images in DTP Server with the `report.coverage.images` property:

```
report.coverage.images=[tag1; tag2; tag3]
```

Associate coverage images in DTP in the Report Center administration page (**administration> Projects> Filters> [click on a filter]**).

You can also use the `report.coverage.limit` property to specify a lower coverage threshold:

```
report.coverage.limit=[value]
```

Coverage results lower than this value are highlighted in the report. The default value is `40`.

CCE Usage Example

In this example, the following code is from a c++ source file called `main.c`:

```
#include <iostream>

int main(int argc, char ** argv) {
   if (argc > 1) {
      std::cout << "Thank you for arguments" << std::endl;
   } else {
      std::cout << "Provide some arguments please !" << std::endl;
   }
   return 0;
}
```

The normal file compilation command is `gcc`:

```
g++ -c main.c -o main.o
```

To instrument this file and compile the instrumented code to the object file, the compilation command line must include the `cpptestcc` command prefix:

```
cpptestcc -compiler gcc_3_4 -line-coverage -workspace /home/test/proj/cov
-- g++ -c main.c -o main.o
```

Two artifacts are created as s a result of the `cpptestcc` command invocation:

1. object with instrumented code
2. code structure information stored in the directory specified with `-workspace` option.

Once the source file is instrumented and compiled to the object file it, can be linked to form the final executable. Normally this simple example would be linked with the following command:

```
g++ main.o -o app.exe
```

Coverage instrumentation requires additional library so the linking command line needs to look as follows:

```
g++ main.o <coverage tool install dir>/runtime/lib/cpptest.a -o app.exe
```

In this example, the static version of the coverage library was used. The dynamic/shared version, as well as the source code, is also provided for building a customized version. For additional information, see “Code Coverage Engine Runtime Library”, page 64, and “Customizing the Runtime Library”, page 66.

Once the application is linked it can be executed to collect information about code coverage. Run the following command:

```
./app.exe
```

The application will output the coverage log file, named `cpptest_results.clog` by default, in the current work directory.

Finally, generate the report using the following command:

```
cpptestcli -config builtin://Coverage -workspace /home/test/proj/cov -input
cpptest_results.clog -report report_dir
```

A report directory will be created containing the HTML report with code coverage information.

Integrating with Make-based Build Systems

Integrating CCE with a projects based on a GNU Make or similar build tools typically requires modifying the build scripts. In most of the cases the command line invoked by compilation and linking rules should be altered. This may require you to modify the make variables or, in some cases, the compilation and linking rules definitions.

Integrating with Make Compilation Rule

Prefix the compiler command line with the `cpptestcc` command wrapper to integrate with the Make compilation rule. To determine the best approach, start with analyzing build scripts and locating the definition for the compilation rule. In some cases, there are different rules for specific files, such as rules to handle C or C++ files. The following example shows how the compilation rule(s) may be defined:

```
$(PRODUCT_OBJ_ROOT)/%$(EXT_OBJ) : %$(EXT_CXX)
   $(CXX) $(CXXFLAGS) $(CXXSTATICFLAGS) -DAPPNAME=product
```

In this example, the compiler is referenced with a `CXX` make variable. There are two options:

1. Add the prefix variable to the compilation rule, or

2. Overwrite the compiler variable

The following sections describe how to proceed with either approach

Adding the Prefix Variable to the Compilation Rule

Modify the compilation rule by prefixing the variable that references the compiler with an additional variable:

```
$(PRODUCT_OBJ_ROOT)/%$(EXT_OBJ) : %$(EXT_CXX)
   $(COV_TOOL) $(CXX) $(CXXFLAGS) $(CXXSTATICFLAGS) -
DAPPNAME=product
```

Additionally, assign a value to the added variable (`COV_TOOL`) either at the time of the Make invocation (example a) or in build scripts (example b):

Example a

```
make COV_TOOL="cpptestcc -compiler gcc_3_4 -line-coverage
-workspace /home/test/proj/cov -- "
```

Example b

In this option, the variable would likely be inside a condition dependent on an additional variable:

```
ifeq (COV_BUILD,$(BUILD_TYPE))
    COV_TOOL="cpptestcc -compiler gcc_3_4 -line-coverage
-workspace /home/test/proj/cov -- "
endif
```

Overwriting the Compiler Variable

In this approach, the compiler variable is used to specify the coverage tool command line. This can be done either at the time of the Make invocation (example c) or in build scripts (example d) after the original value of the `CXX` variable is specified (to avoid overriding coverage tool command with original compiler):

Example c

If your build scripts have separate compilation rules for different types of files, you may need to overwrite more than one variable, for example CC and CXX:

```
make CXX="cpptestcc -compiler gcc_3_4 -line-coverage -workspace /home/test/proj/cov -- g++"
```

Example d

In this option, the variable would likely be inside a condition dependent on an additional variable:

```
ifeq (COV_BUILD,$(BUILD_TYPE))
    CXX="cpptestcc -compiler gcc_3_4 -line-coverage -workspace /home/test/proj/cov -- g++"
endif
```

Integrating with Make Linking Rule

Modify the linking rule to include the additional library required for code instrumentation. The `cpptestcc` tool library can have different forms depending on specific project requirements. It can be a shared/dynamic library, static library, or an object file. In all cases, the particular linker options may

have a different form, but modifying the Makefiles will be done in a very similar manner regardless of the case.

For more details about using different forms of the `cpptestcc` tool runtime library, see "Using Coverage Tool for Complex Projects", page 62. This section focuses on the general approach to modifying linker command lines in Make-like environments.

To find an appropriate place for modification, begin with analyzing build scripts and locating the definition of the linking rule. The following example shows how the linking rule may be defined:

```
$(PROJ_EXECUTABLE): $(PRODUCT_OBJ)
  $(LXX)  $(PRODUCT_OBJ) $(OFLAG_EXE)$(PROJ_EXECUTABLE) $(LXXFLAGS) $(SYSLIB)
$(EXECUTABLE_LIB_LXX_OPTS)
```

You can either add a special variable for representing the `cpptestcc` tool library, or append the coverage library to one of the variables already used in the linking rule.

Adding a Variable to Represent Coverage Tool Library

The following example shows what the modified rule may look like:

```
$(PROJ_EXECUTABLE): $(PRODUCT_OBJ)
   $(LXX)  $(PRODUCT_OBJ) $(OFLAG_EXE)$(PROJ_EXECUTABLE) $(LXXFLAGS) $(SYSLIB)
$(EXECUTABLE_LIB_LXX_OPTS) $(COV_LIB)
```

Additionally, assign a value to the added variable (`COV_LIB`) either at the time of the Make invocation (example e) or in build scripts (example f):

Example e

```
make COV_LIB="<COV_TOOL_INSTALLATION>/runtime/lib/cpptest.a "
```

Example f

In this option, the variable would likely be inside a condition dependent on an additional variable:

```
ifeq (COV_BUILD,$(BUILD_TYPE))
    COV_LIB="<COV_TOOL_INSTALLATION>/runtime/lib/cpptest.a"
endif
```

Appending Coverage Library to Existing Variable in the Linking Rule

```
ifeq (COV_BUILD,$(BUILD_TYPE))
    LXXFLAGS+="<COV_TOOL_INSTALLATION>/bin/engine/lib/cpptest.a"
endif
```

Integrating with MSBuild

Your can integrate Code Coverage Engine with MSBuild. This integration is achieved with parasoft-coverage files shipped with CCE in the [INSTALL_DIR]/integration/msbuild directory.

For more details, see the parasoft_dtp_plugin_msbuild_manual.pdf stored in the [INSTALL_DIR]/integration/msbuild directory.

Using Coverage Tool for Complex Projects

The examples in this section describe several classes of projects and ways to approach integrating CCE.

- Build includes compilation of multiple source files without static or dynamic libraries

  Compilation phase:

  For all sources files or selected subsets, compilation should be prefixed with `cpptestcc` tool with options for enabling the desired coverage metrics.

  Linking phase:

  The linker command line should be modified to include the coverage tool library. In most of the cases, you will include the static library:

```
g++ source1.o source2.o <cov tool install dir>/runtime/lib/cpptest.a
```

- Build includes compilation of multiple source files and the preparation of multiple static libraries, which are linked to the main objects

  Compilation phase:

  For all sources files or selected subsets, compilation should be prefixed with `cpptestcc` tool with options for enabling desired coverage metrics. If coverage from static libraries should also be collected, they need to be instrumented, as well.

  Linking phase:

  The linker command line should be modified to include the coverage tool library. In most of the cases, you will include the static library. It is important to add the coverage tool library only once, usually during executable linking. Static libraries created during the build **should not** contain the coverage library.

- Build includes compilation of multiple source files and preparation of multiple dynamic/shared libraries which are linked with main objects

  Compilation phase:

  For all sources files or selected subsets, compilation should be prefixed with `cpptestcc` tool with options for enabling desired coverage metrics. If coverage from dynamic libraries should also be collected, they need to be instrumented, as well.

  Linking phase:

  - Linking command lines for dynamic libraries should be modified to include the coverage tool shared/dynamic-load library.
  - Linking command lines for static libraries should not be modified.
  - Linking command line for executable should be modified to include coverage tool shared/dynamic library

- Build includes the following:
  - Compilation of multiple source files
  - Preparation of multiple dynamic/shared libraries linked with main objects
  - Multiple static libraries linked with main objects

  Compilation phase:

  For all sources files or selected subsets, compilation should be prefixed with `cpptestcc` tool with options for enabling desired coverage metrics. If coverage from dynamic or shared libraries should also be collected, they need to be instrumented, as well.

Linking phase:

- Linking command lines for dynamic libraries should be modified to include the coverage tool shared/dynamic-load library.
- Linking command lines for static libraries should not be modified.
- Linking command line for executable should be modified to include coverage tool shared/dynamic library

Annotating Results Stream with Test Start/Stop Information

You can annotate code coverage results with test start/stop information to understand how a particular test scenario affects code execution. Test start notification conveys information about the test name, which can be used when processing test data and generating reports.

Test Start/Stop annotations functionality is available as an API and can be extended to many different scenarios. For example, you can associate code coverage results with unit tests or associate code coverage results with manual test scenarios performed during system testing.

Using the Test Start/Stop API

The API includes the following functions:

Function	Description
`void CDECL_CALL CppTest_TestStart(const char* testName`	Sends the notification to the results stream about the beginning of a test with specified name.
`void CDECL_CALL CppTest_TestStop(void)`	Sends the notification to the results stream about the end of previously started test.

Include a dedicated header file in the source file that will invoke the API functions:

```
#include "cpptest/cpptest.h"
```

Use the `-I` option to specify the `cpptest.h` header file location when compiling the source file:

```
-I <Installation Directory>/runtime/include
```

Specify a valid string as an argument to the `CppTest_TestStart` function. Null pointers or invalid strings will cause undefined behavior.

Common Applications of the Test Start/Stop Scenarios

The following scenarios illustrate usage of the test start/stop notification API:

- Annotating coverage results with unit test case names

  In this scenario, unit test case names are used as an argument specified to `CppTest_TestStart` function invocation. For some of popular C/C++ unit testing frameworks, dedicated connectors are provided to automate this task. For additional information about unit testing framework connectors, see “Unit Test Connector”, page 49.

To use a unit testing framework that does not have a dedicated connector, you can invoke start/stop API functions at the beginning and end of the test case:

```
#include "cpptest/cpptest.h"

TEST(TimerTest, smokeTest) {
  const char * tcName = testCaseName();
  CppTest_TestStart(tcName);
  int res = init_timer();
  ASSERT_TRUE(res != 0);
  CppTest_TestStop();
}
```

- Annotating coverage results with manual test scenario names for system testing sessions. There are a few ways to achieve this goal:
    - Calls to test start/stop API can be added directly to the tested source. They can be activated with dedicated macros in the debug or test builds of the tested application. The method for providing the name of the test scenario to the API function call depends on the type of application. In some cases you can add a special option to the menu of the tested application that is only visible in the debug build or the command line. This would enable you to specify the name of the performed test scenario and send the notification to results stream once the name is entered.
    - The names of the test cases can be also read from the environmental variable set before starting tested application.
    - Use a special module implemented as a separate thread that is started in parallel to the threads of tested application. The module could, for example, listen on the TCP/IP socket and react when test start/stop command is send from external tool.

Code Coverage Engine Runtime Library

The CCE runtime library is a collection of helper functions and services used by source code instrumentation to emit coverage information at application runtime. Instrumented applications can not be linked without the library. The runtime library can be linked to the final testable binary in multiple ways depending on tested project type.

In addition to providing basic services for instrumented code, the library is also used to adapt the code coverage solution to particular development environments, such as supporting nonstandard transport for coverage results between tested embedded device and development host.

Pre-built Versions and Customized Builds

CCE ships with pre-built versions of the runtime library, which are suitable for using on the same platform on which CCE is installed. In most of the cases, collecting code coverage information from natively developed applications (i.e., MS Windows or Linux x86 and x86-64) can use pre-built versions of runtime library.

All users developing cross-platform applications will need to prepare a custom build of the runtime library using a suitable cross compiler and possibly linker. Source code of the code coverage runtime library is shipped with CCE.

The process of preparing the runtime library custom build is typically limited to compilation of runtime library source code. In some situations you may need to install some fragments of source code to adapt code coverage to a particular development platform. This process is described in the following sections.

Using the Pre-built Runtime Library

The following binary files are included with the CCE:

Windows (x86 and x86-64)

File	Description
`[INSTALL_DIR]/runtime/lib/cpptest.a`	32-bit static archive to be used with Cygwin GNU GCC compilers. To be added to linking command line
`[INSTALL_DIR]/runtime/lib/cpptest.lib`	32-bit import library to be used with Microsoft CL compilers. To be added to linking command line
`[INSTALL_DIR]runtime/lib/cpptes64.lib`	64-bit import library to be used with Microsoft CL compilers. To be added to linking command line
`[INSTALL_DIR]/bin/cpptest.dll`	32-bit Dynamic-link library to be used with Microsoft CL compilers. `[INSTALL_DIR]/bin` should be added to PATH environmental variable
`[INSTALL_DIR]/bin/cpptest64.dll`	64-bit Dynamic-link library to be used with Microsoft CL compilers. `[INSTALL_DIR]/bin` should be added to PATH environmental variable

Linux (x86 and x86-64)

File	Description
`[INSTALL_DIR]/runtime/lib/lib-cpptest.so`	32-bit shared library. To be added linking command line. `[INSTALL_DIR]/runtime/lib` should be added to LD_LIBRARY_PATH
`[INSTALL_DIR]/runtime/lib/libcpptest64.so`	64 bit shared library. To be added linking command line. `[INSTALL_DIR]/runtime/lib` should be added to LD_LIBRARY_PATH

If you need to use the runtime library in a form not provided as an out-of-the-box solution, prepare a custom build of the runtime library that matches specific development environment requirements. For more details about this please see “Customizing the Runtime Library”, page 66.

Integrating with the Linker Command Line

Integrating the CCE runtime library with a tested application linking process usually requires modifying the linker command line and, in some cases, the execution environment. This describes how to modify the linking process when using the pre-built versions shipped with CCE.

- Static library for Windows Cygwin GNU GCC compilers:
    1. Locate the linker command line in your build scripts
    2. Modify the build scripts so that the coverage runtime library is specified somewhere in in the linker command line--preferably after all object files.
- Dynamic-link library for MS CL compilers:
    1. Locate the linker command line in your build scripts

2. Modify the build scripts so that the coverage runtime library is specified somewhere in in the linker command line--preferably after all object files. For example:

```
$(LXX) $(PRODUCT_OBJ) $(OFLAG_EXE)$(PROJ_EXECUTABLE) $(LXXFLAGS) $(SYSLIB)
$(EXECUTABLE_LIB_LXX_OPTS) <INSTALL>/runtime/lib/cpptest.lib
```

3. Make sure that the `[INSTALL_DIR]/bin` directory is added to your PATH environment variable so that the library can be located when the tested program is started. You may also consider copying `cpptest.dll` (or `cpptest64.dll`) file to the same directory as your executable file or to another location that is scanned for dynamic-link libraries during tested application startup.

- Shared library for Linux GNU GCC compilers
  1. Locate the linker command line in your build scripts
  2. Modify the build scripts so that the coverage runtime library is specified somewhere in in the linker command line--preferably after all object files. For example:

```
$(LXX) $(PRODUCT_OBJ) $(OFLAG_EXE)$(PROJ_EXECUTABLE) $(LXXFLAGS) $(SYSLIB)
$(EXECUTABLE_LIB_LXX_OPTS) -L <INSTALL>/runtime/lib -lcpptest
```

     Note the addition of the `-L [INSTALL_DIR]/runtime/lib` and `-lcpptest` options.
  3. Make sure that shared library can be found by tested executable by modifying LD_LIBRARY_PATH environmental variable to include `[INSTALL_DIR]/runtime/lib` location

Customizing the Runtime Library

You may need to customize the runtime library as a result of the following conditions:

- Different form of binary file is required
- Enabling a non-default communication channel for results transport
- Installing custom implementation of communication channel for results transport
- Enabling a non-default support for multithreaded applications
- Installing custom implementation of support for multithreaded applications

Library Source Code Structure

The runtime library source code shipped with CCE is in the `[INSTALL_DIR]/runtime` directory. The following table describes the structure:

Component	Description
`include`	Directory containing library include files. `include/cpptest.h` - library public interface `include/cpptest/*` - library private interface Content of the include directory is not designed for environment specific modifications.

Component	Description
`src`	Directory containing library source code `src/cpptest.c` - main and single source file of the runtime library This file is designed for modifications and customizations.
`Makefile`	Basic Makefile provided for building runtime library
`target`	Directory containing set of Makefile include files with compiler specific options for preparing runtime library builds for most popular development environments
`channel`	Directory containing set of Makefile include files with configuration for supported communication channels.

Switching Communication Channel Support

The runtime library supports data collection through various communication channels. The communication channel used depends on the development environment. In most cases, storing results in a file or files is appropriate, but in others TCP/IP sockets or RS232 transport may be required. Specific communication channels can be enabled by setting the value to a dedicated macro during `cpptest.c` library source file compilation. Add -D<MACRO> to the compilation command line to set the value. The following table provides the full list of communication channel control macros:

Channel	Description
`CPPTEST_NULL_COMMUNICATION`	Empty implementation. If enabled no results will be sent. Suitable for initial test builds and debugging

`CPPTEST_FILE_COMMUNICATION`	File-based implementation. ANSI C File I/O interface is used. If enabled, results will be written to a local drive file.
	The following additional configuration macros are also provided:
	`CPPTEST_LOG_FILE_NAME`: Name of the results file; default `cpptest_results.clog`
	`CPPTEST_LOG_FILE_APPEND`: Creates new results file or appends to existing. Default value is `1 -> append`, alternative `0 -> create new`

Channel	Description
CPPTEST_SPLIT_FILE_COMMUNICATION	File-based implementation. ANSI C File I/O interface is used. If enabled, results will be written into a series of local drive files. You can configure this channel with the following macros: CPPTEST_LOG_FILE_NAME: Name of the first results file in the series; the default is cpptest_results.clog. Other files will be named in sequence, for example, cpptest_results.clog.0001. To pass the series to cpptestcli, ensure that all the files in the series are placed in the same directory and provide only the name of the first file as an input. When you run the cpptestcli command, other files will be merged with the first file in the series and removed from the directory. CPPTEST_MAX_ALLOWED_NUMBER_OF_BYTES_PER_FILE: Specifies the maximum size of one file in the series; default 2000000000 bytes (2 GB).
CPPTEST_UNIX_SOCKET_COMMUNICATION	TCP/IP socket based implementation. POSIX API is used. If enabled results are sent to specified the TCP/IP port. The following additional configuration macros are provided: CPPTEST_LOG_SOCKET_HOST: Specifies host IP address string CPPTEST_LOG_SOCKET_PORT: Specifies the port number CPPTEST_GETHOSTBYNAME_ENABLED: If set to 1, the host can be specified by domain name (requires gethostbyname function to be present)
CPPTEST_WIN_SOCKET_COMMUNICATION	As above, MS Windows API is used
CPPTEST_UNIX_SOCKET_UDP_COMMUNICATION	As above, UDP based implementation

Channel	Description
CPPTEST_RS232_UNIX_COMMUNICATION	RS232 based implementation. POSIX API is used. If enabled then results are sent via the specified RS232 system device. The following additional configuration macros are provided: CPPTEST_RS232_DEVICE_NAME: System device name CPPTEST_RS232_BAUD_RATE: Transmission baud rate CPPTEST_RS232_BYTE_SIZE: Byte size CPPTEST_RS232_PARITY: Parity control CPPTEST_RS232_STOP_BIT: Stop bit usage CPPTEST_RS232_TIMEOUT: Transmission time-out value
CPPTEST_RS232_WIN_COMMUNICATION	As above. MS Windows API is used.
CPPTEST_RS232_STM32F103ZE_COMMUNICATION	STM32F103x USART based implementation. STM Cortex library interface is used (ST/STM32F10x/stm32f10x.h header file is required)
CPPTEST_HEW_SIMIO_COMMUNICATION	Renesas HEW simulator specific implementation.
CPPTEST_LAUTERBACH_FDX_COMMUNICATION	Lauterbach TRACE32 based implementation (FDX used)
CPPTEST_ITM_COMMUNICATION	ARM CoreSight ITM unit based communication. Requires CMSIS header files.
CPPTEST_CUSTOM_COMMUNICATION	Enables empty template for custom implementation

If the runtime library is being built with the provided Makefile, then one of the make configuration files provided in the [INSTALL_DIR]/runtime/channel directory can be used. For details see “Integrating with Make-based Build Systems”, page 59.

Installing Support for Custom Communication Channel

If none of the communication channel implementations fit into your development environment, then a custom implementation can be provided. The following instructions describe how to customize the runtime library so that it uses a custom implementation of a communication channel:

1. Make a copy of [INSTALL_DIR]/runtime/src/cpptest.c and open the file for editing
2. Locate the section 1.13 "Custom Communication Implementation"

The custom communication implementation section contains empty templates for four different methods:

Function	Description
`void cpptestInitializeStream(void)`	This function is responsible initializing the communication channel, for example creating and connecting to a socket or the initialization of UART device.
`void cpptestFinalizeStream(void)`	This function is responsible for finalizing the communication channel. For example, it may be responsible for closing TCP/IP socket.
`int cpptestSendData (const char *data, unsigned size)`	This function is responsible for sending size bytes from a data buffer.
`void cpptestFlushData(void)`	This function is responsible for flushing the data. Its meaning depends on the particular transport type. It may have a limited application in some impementations. In this case it should be left empty.

3. Provide the implementation for these methods that match your environment requirements.
4. Compile `cpptest.c` with the following macro definition added to compilation command line:

   `"-DCPPTEST_CUSTOM_COMMUNICATION"`
5. If the generated object file is insufficient, you can process the file even further to meet your needs (e.g., to create a shared library)

Switching Multithreading API Support

The runtime library contains support for multithreaded applications. POSIX, MS Windows, and VxWorks APIs are supported. You can enable support for a specific multithreading API by adding `-D<MACRO>` to the compilation command line during `cpptest.c` compilation. The following table describes the full list of multithreading API support control macros:

Macro	Description
`CPPTEST_NO_THREADS`	Empty implementation. Coverage runtime is not prepared to be used together with multithreaded applications
`CPPTEST_WINDOWS_THREADS`	MS Windows multithreading API implementation
`CPPTEST_UNIX_THREADS`	POSIX multithreading API implementation
`CPPTEST_VXWORKS_THREADS`	VxWorks multithreading API implementation

Installing Support for Custom Threading API

If you are using CCE with multithreaded applications that do not use a supported multithreading API, you can customize the runtime library to work with your multithreading API. There are following steps required:

1. Make a copy of `[INSTALL_DIR]/runtime/src/cpptest.c` and open the file for editing
2. Locate the section 2.5 "Custom Multithreading Implementation"

Custom multithreading implementation section contain empty templates for two different methods

Function	Description
`static int cpptestLock(void)`	This function ensures synchronized operations inside the coverage tool runtime library. If a thread locks access to the runtime library service, it means an atomic operation is in progress and no other thread can use runtime library services. Once the lock is released other threads can use runtime library services
`static int cpptestUnlock(void)`	Releases the lock on runtime library services.

3. Provide the implementation for the methods that matches your environment requirements.
4. Compile `cpptest.c` with the following macro added to compilation command line:

   `"-DCPPTEST_CUSTOM_THREADS"`
5. If the generated object file is insufficient, you can process the file even further to meet your needs (e.g., to create a shared library)

Building the Runtime Library

CCE ships with a simple Makefile (see “Library Source Code Structure”, page 66) which simplifies the process of building the runtime library. In many instances, however, the make file provided will not be required because the source code is already optimized for the building process. The only step that is always required is the compilation of the main `cpptest.c` source file. Any additional processing of the produced object file will depend on the particular development environment and its requirements, such as providing the runtime library as a shared library.

Building the Runtime Library Using the Provided Makefile

1. Change directory to `[INSTALL_DIR]/runtime`
2. If compilation flags need to be modified (e.g., adding specific cross-compiler specific or definitions to enforce runtime library reconfiguration), provide a new make configuration file in the `target` subdirectory. For convenience, copy one of the existing target configuration files and modify its contents to fit your needs.
3. Invoke the following command line:

   `make TARGET_CFG=<target config file name> CHANNEL=<channel config file name>`
4. A `build` subdirectory will be created with a single object `cpptest.<OBJ_EXT>`, which can be used to link with instrumented application.
5. If the coverage runtime library needs to be linked to form a shared library, dynamic link library, or any other type of binary, the Makefile needs to be customized for this purpose or a custom build needs to be setup.

User Build of the Runtime Library

1. To setup a user build of coverage tool runtime library perform the following steps:
2. Copy the cpptest.c file from `[INSTALL_DIR]/runtime/src/cpptest.c` to your preferred location

3. Introduce any customizations as described in “Customizing the Runtime Library”, page 66
4. Set up a build system of your preference (e.g., IAR Embedded Workbench project or any other type of source code builder)
5. Modify the compilation flags to contain the compiler include flag (typically `-I`) with the following value:

   ```
   -I[INSTALL_DIR]/runtime/include
   ```
6. Add any required configuration defines (typically `-D`), for example:

   ```
   -DCPPTEST_FILE_COMMUNICATION -DCPPTEST_NO_THREADS
   ```
7. Invoke the command to run your builder (for example, select build command in the IDE)
8. Locate the resulting object file and use it to link with your instrumented application

Command Line Reference for cpptestcc

Option name	Description	Notes

`-compiler <name|path>`	Specifies compiler configuration name to be used for code analysis and instrumentation. Compiler configuration name need to be one of the supported compilers names. See “Supported Compilers”, page 104, or use the `-listcompilers` option	Examples:
		`cpptestcc -compiler gcc_3_4`
		`cpptestcc -compiler vc_11_0`
		Configuration file format:
		`cpptestcc.compiler <name>`

`-listcompilers`	Prints out all the names of supported compiler configurations	Configuration file format: `cpptestcc.listCompilers`

Option name	Description	Notes

`-include <file|pattern>`
`-exclude <file|pattern>`
	Includes/excludes file(s) that match the specified pattern into/from instrumentation scope. Options can be specified multiple times.
	Final filtering is determined only after all include/exclude entries have been specified in the order of their specification.
	The following wildcards are supported:
	`?` - Any character
	`*` - Any sequence of characters
	To prevent shells from expanding `*` wildcards to the list of files or directories, you can use the `regex:` prefix to specify the value.
		Example 1:
		Assume the following project layout:
		`<project root>`
		`+ external_libs`
		`+ src`
		`+ include`
		The following `cpptestcc` command will cause all the files from the `external_libs` directory to be excluded from instrumentation scope.:
		`cpptestcc -include regex:*/<project root>/* -exclude regex:*/<project root>/external_libs <remaining part of cmd>`
		Example 2:
		Assume the following project:
		`<project root>`
		`<sourcefiles>.cpp`
		`<headerfiles>.hpp`
		The following command line will instrument only header files without instrumenting source files:
		`cpptestcc -include regex:* -exclude regex:*.cpp <remaining part of cmd>`
		Configuration file format:
		`cpptestcc.include <path|pattern>`

Option name	Description	Notes

`-ignore <pattern>`	Ignores all source files matching specified `<pattern>` during processing. Option can be specified multiple times.
	Files matching the specified pattern will not be parsed or instrumented, but they will be compiled in their original form.
	The following wildcards are supported:
	`?` - Any character
	`*` - Any sequence of characters
	To prevent shells from expanding `*` wildcards to the list of files or directories, you can use the `regex:` prefix to specify the value.
		Use this option to reduce build time overhead by ignoring coverage analysis on some sections of the code (e.g., external libraries) or to ignore specific file that expose problems during processing (e.g., parse error)
		Example:
		`cpptestcc -ignore "*/Lib/*" <remaining part of cmd>`
		`cpptestcc -ignore regex:*/file.c <remaining part of cmd>`
		`cpptestcc -ignore c:/proj/file.c <remaining part of cmd>`
		`cpptestcc -ignore "*/MyLib/*.cpp" -ignore file.cpp <remaining part of cmd>`
		Choosing between `-ignore` and `-include/-exclude`:
		In some cases, `-ignore` functionality may appear to overlap with the `-include`/`-exclude` options. The difference is that the `-ignore` option completely removes the specified file from processing so that it is not parsed by coverage engine, whereas the `-include`/`-exclude` filters are applied after source code is parsed, which allows you to selectively instrument or not instrument header files.
		Configuration file format: `cpptestcc.ignore <path|pattern>`

Option name	Description	Notes
`-line-coverage`	Enables line coverage metric	Runtime coverage results are written to the results log as the code is executed. This imposes some overhead on the tested code execution time, it but assures that coverage data is collected even if the application crashes. This option can not be used with `-optimized-line-coverage`. Configuration file format: `cpptestcc.lineCoverage`
`-optimized-line-coverage`	Enables optimized line coverage data	Runtime coverage results are stored in memory and written to the results log either after the application finishes or on user request. This results in better performance, but results may be lost if the application crashes. This option can not be used with `-line-coverage`. Configuration file format: `cpptestcc.optimizedLineCoverage`
`-workspace <path>`	Specifies the workspace directory to be used during code analysis and instrumentation. The workspace location is used to store important code structure information essential for generating the final coverage report. The workspace location must be the same for `cpptestcc` and `cpptestcli`. tool invocation.	Configuration file format: `cpptestcc.workspace <path>`
`-psrc <file>`	Specifies the configuration file with additional options for `cpptestcc` tool. By default, `cpptestcc` attempts to read the `.psrc` file located in either the current working directory or in user `HOME` directory. Use `-psrc` to specify options inside the configuration file.	Configuration file format: `pscom.psrc <file>`
`-help`	Prints out help message and exits.	Configuration file format: `cpptestcc.help`

Customizing DTP Engines for C/C++

Customization options for Static Analysis Engine are specified with a configuration file. The file should be passed to Static Analysis Engine using the `-settings` switch:

```
cpptestcli -settings custom.properties -config "builtin://Recommended Rules" -com-
piler gcc_3_4 -input cpptest.bdf
```

The `-settings` switch may be specified multiple times. Entries with the same key will be overwritten:

```
cpptestcli -settings team.properties -settings project.properties -settings
user.properties -config "builtin://Recommended Rules" -compiler gcc_3_4 -input
cpptest.bdf
```

General settings are applied in the following order:

1. `[INSTALL_DIR]/etc/cpptestcli.properties`; the base configuration file for Static Analysis Engine and **should not be modified**.
2. `[INSTALL_DIR]/cpptestcli.properties`; contains templates for commonly used settings (license, reporting etc.)
3. `[USER_HOME]/cpptestcli.properties`; optional
4. `[WORKING_DIR]/cpptestcli.properties;` optional
5. Custom settings passed with the command line switch `-settings path/to/settings.properties` (e.g., `-settings ../settings.properties`)

Custom settings passed with the command line switch `-property [key=value]` All of the above settings can be overridden by custom settings that are passed with command line switches (e.g. `-report`, `-config`, `-dtp.share.enabled`).

Modifying a Single Property

You can quickly modify a single property in a settings configuration file with the `-property` switch without creating a dedicated configuration:

```
cpptestcli -property dtp.server=dtp.parasoft.com.pl
-config "builtin://Recommended Rules" -compiler gcc_3_4 -input cpptest.bdf
```

Viewing Current Settings

Use the `-show-settings` switch to print the current settings and customizations.

Advanced Configuration

For advanced configuration of C++test's native code analyzers, a dedicated configuration file (in a format supported by the analyzers) can be specified with `-psrc <CONFIG_FILE>` switch. Parasoft Support will provide content for the advanced configuration file.

Using Variables

The following table shows variables that can be used in settings values.

> We recommend you avoid using spaces, +, /, or any other special characters when setting variables or values for configuration settings, as some API calls may require properly encoded URLs.

Variable	Description	Example
`analysis_type`	Outputs a comma separated list of enabled analysis types (e.g., Static, Generation and Execution)	`${analysis_type}`
`env_var`	Outputs the value of the environmental variable specified after the colon.	`${env_var:HOME}`
`config_name`	Outputs the name of executed Test Configuration.	`${config_name}`
`dtp_project`	Outputs the name of DTP project specified in the settings file using `dtp.project` option.	`${dtp_project}`
`project_module`	Outputs the name of the tested project's module. If more than one module is provided as an input, the first tested module name is output followed by an ellipsis (...). The variable can be configured in the settings file with the `project.module` option.	`${module_name}`
`host_name`	Outputs the name of the host.	`${host_name}`
`user_name`	Outputs the name of the current user.	`${user_name}`
`os`	Outputs the name of the operating system.	`${os}`
`arch`	Outputs the name of the operating system architecture	`${arch}`
`exec_env`	Outputs the execution environment. Ths variable is a concatenation of `${os}` and `${arch}` variables. It can be configured in the sittings file with the `exec.env` option.	`${exec_env}`
`scontrol_branch`	Outputs the source control branch name for the tested project. If more than one branch name is detected, the first branch name is output followed by an ellipsis (...). The variable can be configured in the settings file with the `scontrol.branch` option.	`${scontrol_branch}`
`tool_name`	Outputs the name of the tool (i.e., Jtest, C++test, dotTEST).	`${tool_name}`
`jvm_prop`	Outputs the value of the Java vm property specified after the colon.	`${jvm_prop:os.name}`

Settings Reference

The following tables contain settings that are currently supported in DTP Engines.

Base Configuration Settings

Setting	Value	Description/Notes

`console.verbosity.level`	`low`
	`normal`
	`high`
		Specifies the verbosity level for the Console
		`low`: configures the Console view to show errors and basic information about the current steps and status (done, failed, up-to-date).
		`normal`: (default) also shows command lines and issues reported during test and analysis.
		`high`: also shows warnings.

`parallel.mode`	`disabled`
	`auto`
	`manual`
		Determines which of the following modes is active:
		`disabled`: configures Parasoft Test to use only one of the available CPUs.
		`auto`: (default) allows Parasoft Test to control parallel processing settings.
		`manual`: allows you to manually configure parallel processing settings to suit your specific needs.

`parallel.no_memory_limit`	`true` `false`	Enables/disables restrictions (beyond existing system limitations) on the memory consumed by parallel processing. Default is `false`
`parallel.free_memory_limit`	`[percentage]`	Specifies the amount of memory that should be kept free in low memory conditions (expressed as a percentage of the total memory available for the application). This is used to ensure that free memory is available for other processes. Default is `25`
`parallel.max_threads`	`[number]`	Specifies the maximum number of parallel threads that can be executed simultaneously. The actual number of parallel threads is determined by the number of CPUs, available memory, and license settings. The default value is equal to the number of CPUs

Setting	Value	Description/Notes
file.encoding.mode	default auto user	Specifies how file encoding is determined. default: enables use of system properties auto: enables automatic detection of encoding for the Far-East languages specified with file.encoding.lang user: enables use of specified encoding by file.encoding.name.
file.encoding.lang	[code]	Allows specify language's numeric code when file.encoding.mode is set to auto: Japanese = 1 Chinese = 2 Simplified Chinese = 3 Traditional Chinese = 4 Korean = 5
file.encoding.name	[encoding]	Allows you to specify the encoding name when file.encoding.mode is set to user: ASCII-US UTF-8 UTF-16 UTF-16LE UTF-16BE ...
settings.validation	true false	Enables/disables settings validation.
settings.rules.file.cpptest	path	Indicates the path to a file that contains additional rules for settings validation. The file should follow the .properties format and include rules according to the following examples: engine.path=$ANY engine.enabled=$BOOLEAN engine.analysis.deep=$INTEGER engine.severity.limit=$REGEXP{[1-5]} engine.verbosity.level=$REGEXP_IC{(low)|(normal)|(high)}

Test Configuration Settings

Setting	Value	Description/Notes
`cpptest.configuration`	`[test configuration]`	Specifies the default test configuration when the `-config` switch is absent. See “Specifying Test Configurations”, page 17.
`configuration.dir.builtin`	`[path]`	Path to directory with built-in test configurations.
`configuration.dir.user`	`[path]`	Path to directory with user-defined test configurations.
`configuration.share.path`	`[path]`	Path on DTP server share with shared test configuration.
`cpptest.custom.rule.dir`	`[path to directory]`	Specifies the location of user-defined coding standard rules. Default is `[INSTALL_DIR]/rules/user`

Compiler Settings

Setting	Value	Description/Notes
`cpptest.compiler.dir.user`	`[path to directory]`	Specifies the location of custom compiler configuration files. See “Compiler Configuration”, page 21.
`cpptest.compiler.family`	`[compiler identifier]`	Specifies the default compiler configuration to use during analysis, when `-compiler` switch is absent. See “Compiler Configuration”, page 21.

Development Testing Platform Settings

Setting	Value	Description/Notes
`dtp.server`	`[host]`	Specifies the host name of the DTP server.
`dtp.port`	`[port]`	Specifies the port number on DTP server port. The default settings is `443`.
`dtp.user` `dtp.password`	`[username]` `[password]`	Specifies authentication to connect to DTP server.

Setting	Value	Description/Notes
dtp.project	[project_name]	Specifies the name of the DTP project that you want linked to. This settings is optional.
dtp.autoconfig	true false	Enables auto configuration with settings stored on the DTP server. The default is false.
report.dtp.publish	true false	Determines whether the current installation is reporting test results to DTP server. The default is false.
report.dtp.publish.src	off min full	Determines whether tested source code is published to DTP server. off: code is not published to DTP server. min: publishes minimal part of sources. In most cases, source code without references to source control, e.g., auto-generated code, is published. full: publishes all sources associated with the specified scope. The default is full if report.dtp.publish is enabled, otherwise the default is off
dtp.share.enabled	true false	Enables/disables connection to Team Server. The default is false.
cpptest.license.use_network	true false	Enables/disables license retrieval from License Service. The default setting is true.
cpptest.license.network.type	dtp ls	Sets the network license type. dtp:file count license that limits usage to a certain number of files as determined by your licensing agreement ls: floating license (machine locked) that limits usage to a certain number of machines
cpptest.license.local.password	[password]	Specifies the local license password.
cpptest.license.local.expiration	[expiration]	Specifies the local license expiration date.

Setting	Value	Description/Notes
cpptest.license.network.edition	desktop_edition server_edition custom_edition	Specifies the type of license that will be retrieve from License Service for this installation. Default is custom_edition
cpptest.license.custom_edition_features	[feature_name, ...]	Specifies active features for custom license edition.
cpptest.license.wait.for.tokens.time	[minutes]	Specifies the time that tool will wait for a license if a license is not currently available.

Scope and Authorship Settings

Setting	Value	Description/Notes
scope.local	true false	Enables/disables code authorship computation based on the local user and system files modification time. Default is true
scope.scontrol	true false	Enables/disables code authorship computation based on data from a supported source control system. Default is false
scope.xmlmap	true false	Enables/disables task assignment based on an XML mapping file that defines how tasks should be assigned for particular files or sets of files. See “Creating Authorship XML Map Files”, page 26 for syntax information. Default is false
scope.xmlmap.file	[path]	Specifies the path to XML mapping file that defines how tasks should be assigned for particular files or sets of files.
cpptest.scope.module.[MODULE_NAME]	[path to a module root location]	Defines the module within a root location. See “Defining Source File Structures (Modules)”, page 22.
authors.ignore.case	true false	Enables/disables author name case sensitivity. Example: true: David and david are considered the same user. Default is false

Setting	Value	Description/Notes
`authors.mappings.location`	`local` `shared`	Specifies where the authorship mapping file is stored. Default is `local`. See `authors.user` and `authors.mapping` options for details. When set to `shared`, mappings could be specified in file located on DTP share. See `authors.shared.path` option for details.
`authors.shared.path`	`[path]`	Specifies the location of authors mapping file in DTP share. Example: `authors.shared.path=xtest/authors_map.txt`

`authors.user{n}`	`[user_name, email, full_name]`	Specifies a specific author by user name, email, and full name.
		Example:
		`authors.user1=dan,dan@parasoft.com,Dan Stowe`
		`authors.user2=jim,jim@parasoft.com,Jim White`

`authors.mapping{n}`	`[from_user, to_user]`	Specifies a specific author mapping.
		Example:
		`authors.mapping1=old_user,new_user`
		`authors.mapping2=broken_user,correct_user`

Suppression Settings

Setting	Value	Description/Notes

`suppression{n}.file.ext`	`[ext]`	Specifies the extension of types of files that should be scanned for comment suppressions.
		Example:
		`suppression1.file.ext=xml`
		`suppression2.file.ext=java`
		Set the comment prefix with the `suppression{n}.comment` setting.

Setting	Value	Description/Notes

`suppression{n}.comment`	`[comment]`	Specifies comment prefix for types of files identified in `suppression.file.ext` setting.
		Example:
		`suppression1.comment=//`
		`suppression2.comment=<!--`

`suppression{n}.comment.suffix`	`[comment suffix]`	Defines the suppression comment suffix when file extensions has been specified with the `suppression.file.ext` setting. If not specified then suppression comments will not be suffixed. Example: `suppression1.comment.suffix=-->`
`suppression{n}.block.only`	`true|fales`	Enables/disables block-only comment suppressions support when file extensions have been specified with the `suppression.file.ext` setting. Default is `false`.

`suppression.local.dir`	`[path]`	Specifies the custom directory for storing local suppressions. An absolute path should be provided.
		Example:
		`suppression.local.dir=file:///C:/parasoft/suppression/storage`
		`suppression.local.dir=../suppression/storage`

Technical Support Settings

Setting	Value	Description/Notes
`techsupport.enabled`	`true` `false`	Enables/disables global automatic technical support data collection is globally enabled with verbose logging. Default is `false`

Setting	Value	Description/Notes
`logging.verbose`	`true` `false`	Enables/disables verbose logs. Verbose logs are stored in the `xtest.log` file in the location specified with the `local.storage.dir` setting. Verbose logging state persists across sessions (restored on application startup). The log is a rolling file with a fixed maximum size. A backup is created whenever the max size is reached . Default is `false`
`logging.scontrol.verbose`	`true` `false`	Enables/disables output from source control commands in verbose logs. Note that output from source control may include fragments of analyzed source code. Default is `false`
`techsupport.create.on.exit`	`true` `false`	Enables/disables automatic archive creation when the application is shut down. The `techsupport.enabled` setting must also be enabled for packages to be created automatically. Default is `true`.
`techsupport.archive.location`	`[path]`	Specifies the custom directory where support packages should be created.
`techsupport.include.reports`	`true` `false`	Enables/disables the inclusion of reports in the technical support package.

Report Settings

Setting	Value	Description/Notes
`session.tag`	`[name]`	Specifies a tag for signing results form the test session. The tag is a unique identifier for the specified analysis process made on a specified module. Reports for different test sessions should be marked with different session tags.

Setting	Value	Description/Notes
`build.id`	`[id]`	Specifies a build identifier used to label results. It may be unique for each build but may also label more than one test sessions that were executed during a specified build. The default settings is `build-yyyy-MM-dd HH:mm:ss`
`project.module`	`[name]`	Specifies a custom name for the project's module. The setting may be used to describe unique runs. If unspecified, the tested module is detected automatically based on code provided to analysis.
`exec.env`	`[env1;env2...]`	Specifies a list of tags that describe the environment where the run was executed. Tags may describe operating system (e.g., Windows, Linux), architecture (e.g., x86, x86_64), compiler, browser, etc. The `exec.env` tags enable the entire session to be described. A detailed description of the environment may also be included in the test suite, test, or test case levels via services API.
`report.location`	`[path]`	Specifies the directory where report should be created.

`report.format`	`xml`
	`html`
	`pdf`
	`csv`
	`custom`
		Specifies the report format. Use a comma separated list of formats to generate multiple formats.
		Default is `xml`

`report.custom.extension`	`[ext]`	Specifies the report file extension of the XSL file for a custom format. Use with `report.format=custom` and `report.custom.xsl.file`.
`report.custom.xsl.file`	`[path]`	Specifies the location of the XSL file for a custom format. Use with `report.format=custom` and `report.custom.extension`

Setting	Value	Description/Notes
`report.developer_errors`	`true` `false`	Determines whether manager reports include details about developer errors. The default is `true`.
`report.developer_reports`	`true` `false`	Determines whether the system generates detailed reports for all developers (in addition to a summary report for managers). The default is `false`.
`report.authors_details`	`true` `false`	Determines whether the report includes an overview of the number and type of tasks assigned to each developer. The default is `true`.
`report.contexts_details`	`true` `false`	Determines whether the report includes an overview of the files that were checked or executed during testing. The default is `true`.
`report.suppressed_msgs`	`true` `false`	Determines whether report includes suppressed messages. The default setting is `false`.
`report.metadata`	`true` `false`	Determines whether additional metadata about findings should be downloaded from DTP. Only findings that are already present on DTP are affected. The DTP server must also support the metadata service for this settting to have an effect. Default is `true`.

`report.scontrol`	`off`
	`min`
	`full`
		Specifies if and how much additional information from source control is included in the report:
		`min`: repositories, file paths and revisions
		`full`: includes the same information as min, as well as task revisions and comments.
		Default is `off`

`report.associations`	`true` `false`	Enables/disables showing requirements, defects, tasks, and feature requests associated with a test in the report. The default is `true`.

Setting	Value	Description/Notes
`issue.tracking.tags`	`[tag1,tag2,...]`	Specifies a list of issue tracking tags. The following tags are supported by default: pr, `fr`, `task`, `asset`, `req`.
`report.assoc.url.[tag]`	`[url]`	Generates link to association inside the HTML report. The URL is a query string containing an `[%ID%]` placeholder for the `PropertyAttribute` value.
`report.active_rules`	`true` `false`	Determines if report contains a list of the rules that were enabled for the test. The default setting is `true`.

`report.rules`	`[url]`	Specifies a directory for storing static analysis rules HTML files (retrieved by clicking the **Printable Docs** button in the Test Configuration's Static Analysis tab).
		Examples:
		`report.rules=file:///C:/parasoft/gendoc/`
		`report.rules=../gendoc/`

`report.test_params`	`true` `false`	Determines whether report includes test parameter details. The default setting is `true`.
`report.testcases_details`	`true` `false`	Enables/disabled details about specific Google Test assertion failures in the generated HTML report. Default is `false`
`report.coverage.images`	`[tag1,. . .]`	Specifies a set of tags that will be used to create coverage images in DTP Server. DTP supports up to 3 coverage images per report.
`report.coverage.limit`	`[limit]`	Value that specifies the lower coverage threshold. Coverage results lower than this value are highlighted in the report. Default is `40`
`report.metrics.attributes`	`[attr1;attr2;...]`	Specifies a list of additional attributes for metric results. The following attributes are supported by default: `module`, `namespace`, `type`, `method`.

Setting	Value	Description/Notes
`report.archive`	`true` `false`	Enables/disables archiving reports into a ZIP file.
`report.graph.start_date`	`[MM/dd/yy]`	Specifies start date for trend graphs that track static analysis task, test execution, and coverage. Use with `report.graph.period=[?d|?m|?y]`
`report.graph.period`	`[?d|?m|?y]`	Determines the duration from the start date for trend graphs that track static analysis task, test execution, and coverage. Use with `report.graph.start_date=[MM/dd/yy]`
`report.mail.enabled`	`true` `false`	Enables/disables report emails to developers and additional recipients specified with the `report.mail.cc` setting. If enabled, all developers that worked on project code will automatically be sent a report that contains the errors/results related to his or her work. The default setting is `false`.
`report.mail.server`	`[host]`	Specifies the mail server used to send reports.
`report.mail.port`	`[port]`	Specifies the port for SMTP server. The default port is 25.
`report.mail.security`	`[security]`	Specifies SMTP server connection security. STARTTLS and SSL are supported. The default is STARTTLS.
`report.mail.subject`	`[subject line]`	Specifies the subject line of the emails sent.

`report.mail.username`
`report.mail.password`
`report.mail.realm`
	`[user_name]`
	`[password]`
	`[realm]`
		Specifies the settings for SMTP server authentication. The realm value is required only for those servers that authenticate using SASL realm.

`report.mail.domain`	`[domain]`	Specifies the mail domain used to send reports.
`report.mail.time_delay`	`[time]`	Specifies a time delay between emailing reports (to avoid bulk email restrictions).

Setting	Value	Description/Notes
report.mail.from	[email|user_name]	Specifies the "from" line of the emails sent.
report.mail.attachments	true false	Enables/disables sending reports as attachments. All components are included as attachments; before you can view a report with images, all attachments must be saved to the disk. The default setting is false.
report.mail.compact	trends links	Specifies how report information is delivered in the email. trends: email contains a trend graph, summary tables, and other compact data; detailed data is not included. links: email only contains a link to a report available on DTP server. This setting is not configured by default
report.mail.format	html ascii	Specifies content type for the email. The default setting is html.
report.mail.cc	[email; ...]	Specifies email address for sending comprehensive manager reports. Multiple addresses must separated with a semicolon. This setting is commonly used to send reports to managers or architects, as well as select developers.
report.mail.include	[email, ...]	Specifies email addresses of developers that you want to receive developer reports. Multiple addresses must separated with a semicolon. This setting is commonly used to send developer reports to developers if developer reports are not sent automatically (e.g., because the team is not using a supported source control system). This setting overrides addresses specified in the 'exclude' list.
report.mail.exclude	[email; ...]	Specifies email addresses that should be excluded from automatically receiving reports.

Setting	Value	Description/Notes
report.mail.exclude.developers	true false	Enables/disables report emails to developers not explicitly listed in the report.mail.cc setting. This setting is used to prevent reports from being mailed to individual developers. The default setting is false.
report.mail.unknown	[email|user_name]	Specifies where to email reports for errors assigned to "unknown".
report.mail.on.error.only	true false	Enables/disables email reports to the manager when an error is found or a fatal exception occurs. Developer emails are not affected by this setting; developer emails are sent only to developers who are responsible for reported errors. The default setting is false.
report.setup.problems	top bottom hidden	Determines placement of setup problems section in report. The default setting is bottom.
report.setup.problems.category_limit	[numerical value]	Specifies a limit to the number of messages reported in a single setup problem category. Default is 10
report.setup.problems.display_limit	[numerical value]	Specifies a limit to the total number of messages displayed in the HTML report in the setup problem section. Default is 100
report.setup.problems.console	true false	Determines whether setup problems will be printed on the console. The default setting is true.
report.ue_coverage_details_htmls	LC DC	Specifies type of coverage included in an additional report, which includes source code annotated with line-by-line coverage details, when a test's HTML report links to it. LC: line coverage DC: decision coverage
report.separate_vm.xmx	[size]	Specifies how much memory should be used for reports generation. The default is 1024M.

Setting	Value	Description/Notes
report.separate_vm	true false	Enables/disables generating reports as a separate virtual machine. Default is false.
report.separate_vm.launch.file	[path]	Specifies path to launch file which should be used during reports generation.
dupcode.sorting.mode	oldest|newest|paths	Determines how elements in the code duplication findings are sorted. oldest: the oldest result appears at the top. newest: the newest result appears at the top. paths: sorts by full path names in ascending alphabetical order (A to Z). The default is paths.
report.coverage.version	1 2	Specifies the version of the XML coverage report: 1: the standard version will be used. 2: the size of the XML report will be optimized. The default value is 1.

General Source Control Settings

Setting Name	Value	Description/Notes
scontrol.timeout	[seconds]	Specifies timeout value for operations with source control. The default value is 60.
scontrol.branch	[name]	Enables you to specify a custom name for the tested branch. This setting may be used to describe unique runs. If it is not specified, the tested branch is detected automatically based on code provided to analysis.

AccuRev Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	accurev	AccuRev repository type identifier.

Setting Name	Value	Description/Notes
scontrol.accurev.exec	[path]	Path to external client executable (accurev).
scontrol.rep{n}.accurev.host	[host]	AccuRev server host.
scontrol.rep{n}.accurev.port	[port]	AccuRev server port. Default port is 1666.
scontrol.rep{n}.accurev.login	[login]	AccuRev user name.
scontrol.rep{n}.accurev.password	[password]	AccuRev password.

ClearCase Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	ccase	ClearCase repository type name.
scontrol.ccase.exec	[path]	Path to external client executable (cleartool).
scontrol.rep{n}.ccase.vob	[path]	Specifies the VOB's mount point - the path at which the VOB will be accessed by user. Examples: scontrol.rep.ccase.vob=X:\myvob scontrol.rep.ccase.vob=/vobs/myvob
scontrol.rep{n}.ccase.vob_tag	[tag]	The VOB's unique tag in the ClearCase network region.

CVS Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	cvs	CVS repository type identifier.
scontrol.rep{n}.cvs.root	[root]	Full CVSROOT value.

Setting Name	Value	Description/Notes
scontrol.rep{n}.cvs.pass	[password]	Plain or encoded password. The encoded password should match password in the .cvspass file. For CVS, use the value in .cvspass from within the user's home directory. For CVSNT, use the value store in the registry under HKEY_CURRENT_USER\Software\Cvsnt\cvspass The password is saved in the registry when you first log into the CVS repository from the command line using `cvs login`. To retrieve the password, go to the registry (using `regedit`) and look for the value under HKEY_CURRENT_USER->CVSNT> cvspass. This displays your entire login name (e.g., :pserver:exampleA@exampleB:/exampleC) and encrypted password value.
scontrol.rep{n}.cvs.useCustomSSHCredentials	true false	Enables/disables using the cvs login and password for EXT/SSH connections. Default is `false`.
scontrol.rep{n}.cvs.ext.server	[cvs]	Specifies which CVS application to start on the server side if connecting to a CVS server in EXT mode. Has the same meaning as the CVS_SERVER variable. Default is `cvs`.
scontrol.rep{n}.cvs.ssh.loginname	[login]	Specifies the login for SSH connections (if an external program can be used to provide the login).
scontrol.rep{n}.cvs.ssh.password	[password]	Specifies the password for SSH connection.
scontrol.rep{n}.cvs.ssh.keyfile	[file]	Specifies the private key file to establish an SSH connection with key authentication.
scontrol.rep{n}.cvs.ssh.passphrase	[passphrase]	Specifies the passphrase for SSH connections with the key authentication mechanism.
scontrol.rep{n}.cvs.useShell	true false	Enables/disables an external program (CVS_RSH) to establish a connection to the CVS repository. Default is `false`.

Setting Name	Value	Description/Notes
scontrol.rep{n}.cvs.ext.shell	[path]	Specifies the path to the executable to be used as the CVS_RSH program. Command line parameters should be specified in the cvs.ext.params property.
scontrol.rep{n}.cvs.ext.params	[parameters]	Specifies the parameters to be passed to an external program. The following case-sensitive macro definitions can be used to expand values into command line parameters: {host} repository host {port} port {user} cvs user {password} cvs password {extuser} parameter cvs.ssh.loginname {extpassword} parameter cvs.ssh.password {keyfile} parameter cvs.ssh.keyfile {passphrase} parameter cvs.ssh.passphrase

Git Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	git	Git repository type identifier.
scontrol.git.exec	[path]	Path to git executable. If not set, assumes git command is on the path.
scontrol.rep{n}.git.url	[url]	The remote repository URL (e.g., git://hostname/repo.git).
scontrol.rep{n}.git.workspace	[path]	The directory containing the local git repository.

Mercurial Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	hg	Mercurial reposity type identifyer.

Setting Name	Value	Description/Notes
scontrol.hg.exec	[path]	Path to external client executable. Devault is hg
scontrol.rep{n}.hg.url	[url]	The remote repository URL (e.g., http://hostname/path).
scontrol.rep{n}.hg.workspace	[path]	The directory containing the local Mercurial repository.

Perforce Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	perforce	Perforce repository type identifier.
scontrol.perforce.exec	[path]	Path to external client executable (p4).
scontrol.rep{n}.perforce.host	[host]	Perforce server host.
scontrol.rep{n}.perforce.port	[port]	Perforce server port. Default port is 1666.
scontrol.rep{n}.perforce.login	[login]	Perforce user name.
scontrol.rep{n}.perforce.password	[password]	Perforce password, optional if ticket is used for authentication.
scontrol.rep{n}.perforce.client	[client]	The client workspace name as specified in the P4CLIENT environment variable or its equivalents. Root directory for specified workspace should be configured correctly for local machine.

Serena Dimensions Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	serena	Serena Dimensions repository type identifier.
scontrol.serena.dmroot	[path]	Path to the Serena Dimensions executable. Example: C\:\\Program Files (x86)\\Serena\\Dimensions 2009 R2\\CM\\
scontrol.rep{n}.serena.login	[login]	Serena user name.
scontrol.rep{n}.serena.password	[password]	Password.
scontrol.rep{n}.serena.host	[host]	Serena Dimensions server host name.

Setting Name	Value	Description/Notes
scontrol.rep{n}.serena.dbname	[name]	Name of the database for the product you are working with.
scontrol.rep{n}.serena.dbconn	[connection]	Connection string for that database.
scontrol.rep{n}.serena.locale	[locale]	The language used, (e.g., en_US)
scontrol.rep{n}.serena.mapping	[mapping]	If the project has been downloaded/ moved to a location other than default work area, use this option to specify a mapping between the project or stream with the Serena repository and the local project. If you are working in the default work area, you do not need to define mappings.

StarTeam Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	starteam	StarTeam repository type identifier.
scontrol.rep{n}.starteam.host	[host]	StarTeam server host.
sscontrol.rep{n}.starteam.port	[port]	StarTeam server port. Default port is 49201.
scontrol.rep{n}.starteam.login	[login]	Login name.
scontrol.rep{n}.starteam.password	[password]	Password (not encoded).
scontrol.rep{n}.starteam.path	[path]	Specifies the project, view, or folder that you are currently working with. You can specify a project name (all views will be scanned when searching for the repository path), project/view (only the given view will scanned) or project/view/ folder (only the specified Star Team folder will be scanned). This setting is useful for working with large multi-project repositories. Examples: scontrol.rep.starteam.path=proj1 scontrol.rep.starteam.path=proj1/view1 scontrol.rep.starteam.path=proj1/ view1/folderA scontrol.rep.starteam.path=proj1/ view1/folderA/folderB

Setting Name	Value	Description/Notes
scontrol.rep{n}.starteam.workdir	[path]	Specifies a new working directory for the selected view's root folder (if the path represents a view) or a new working directory for the selected folder (if the path represents a folder) when the scontrol.rep.starteam.path setting points to a StarTeam view or folder. Examples: scontrol.rep.starteam.workdir=C:\\storage\\dv scontrol.rep.starteam.workdir=/home/storage/dv

Subversion Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	svn	Subversion repository type identifier.
scontrol.svn.exec	[path]	Path to external client executable (svn).
scontrol.rep{n}.svn.url	[url]	Subversion URL specifies protocol, server name, port and starting repository path. Example: svn://buildmachine.foobar.com/home/svn
scontrol.rep{n}.svn.login	[login]	Login name.
scontrol.rep{n}.svn.password	[password]	Password (not encoded).

Synergy/CM Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	synergy	Synergy/CM repository type identifier.
scontrol.synergy.exec	[path]	Path to external client executable (ccm).
scontrol.rep{n}.synergy.host	[host]	Computer on which synergy/cm engine runs. Local host is used when missing. For Web mode, the host must be a valid Synergy Web URL with protocol and port (e.g., http://synergy.server:8400).
scontrol.rep{n}.synergy.dbpath	[path]	Absolute synergy database path (e.g., \\host\db\name).

Setting Name	Value	Description/Notes
scontrol.rep{n}.synergy.projspec	[specification]	Synergy project specification which contains project name and its version (e.g., name-version).
scontrol.rep{n}.synergy.login	[login]	Synergy user name.
scontrol.rep{n}.synergy.password	[password]	Synergy password (not encoded).
scontrol.rep{n}.synergy.port	[port]	Synergy port.
scontrol.rep{n}.synergy.remote_client	[client]	(UNIX only) Specifies that you want to start ccm as a remote client. Default value is false. Optional. This is not used for Web mode.
scontrol.rep{n}.synergy.local_dbpath	[path]	Specifies the path name to which your database information is copied when you are running a remote client session. If null, then the default location will be used. This is not used for Web mode.

Microsoft Team Foundation Server Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	tfs	TFS repository type identifier.
scontrol.rep{n}.tfs.url	[url]	URL to TFS repository, e.g., http://localhost:8080/tfs
scontrol.rep{n}.tfs.login	[login]	TFS user name.
scontrol.rep{n}.tfs.password	[password]	TFS password.

Microsoft Visual SourceSafe Source Control Settings

Setting Name	Value	Description/Notes
scontrol.rep{n}.type	vss	Visual SourceSafe repository type identifier.
scontrol.vss.exec	[path]	Path to external client executable (ss).
scontrol.rep{n}.vss.ssdir	[path]	Path of repository database.
scontrol.rep{n}.vss.projpath	[path]	VSS project path.
scontrol.rep{n}.vss.login	[login]	VSS login.
scontrol.rep{n}.vss.password	[password]	VSS password.

Visual Studio Configuration Settings

Setting Name	Value	Description/Notes
`cpptest.input.msvc.compiler`	`[path]`	Specifies the compiler executable. The default compiler is `cl.exe`
`cpptest.input.msvc.add.compiler.options`	`[compiler_option]`	Specifies additional compiler options. If you specify more than one option, separate the values with a space character. The product's property file is preconfigured to use `-I.` for this setting. If no value is present, the setting will be ignored.
`cpptest.input.msvc.solution`	`[solution_name]`	Specifies the solution name. By default, the solution name is the same as the solution file name, but if a project file name is specified as an input parameter, this property can be used to set the solution name.
`cpptest.input.msvc.config`	`[configuration_name]`	Specifies the configuration used during the build. The default is the first configuration in the project file.
`cpptest.input.msvc.platform`	`[platform_name]`	Specifies the platform used during the build. The default is the first platform in the project file.

Integrations

- Integrating with Source Control Systems
- Using DTP Engines in an IDE
- Integrating with CI Tools

Integrating with Source Control Systems

DTP Engines can collect information from source control systems and use the data to assign ownership of violations, filter analyzed files based on time or modification history, and report information about controlled files to DTP Server. Use the C++test 9.5 or later interface to configure integration with source control systems:

1. In your IDE , choose **Parasoft > Preferences** and click **Source Controls**
2. Configure your repository and source control client and click **Apply**.
3. In the Preferences panel menu, click **Scope and Authorship**
4. Enable the **Use source control (modification author) to compute scope** option and click **Apply**.
5. In the Preferences panel menu, click Parasoft
6. Click the **share** to open the Export to localsettings file panel.
7. Select the **Source Controls**, **Scope and Authorship**, and any other options you want to save.
8. Choose a location and click **OK**.
9. Add the following line to the settings file, which ensures that information on source control details are saved to the report:

```
report.scontrol=min
```

10. Either pass the file to the command line or copy the settings in the administration panel of a project in DTP server (Parasoft Test settings tab) if applicable.
11. Run the analysis.

Integrating with CI Tools

Integrating with Jenkins

DTP Engines for C/C++ can be integrated with Jenkins continuous integration software. The Parasoft Findings Plugin for Jenkins allows you to visualize static analysis and test results as trend graphs and warnings.

Parasoft Findings Plugin is available directly in Jenkins. See Parasoft Findings Plugin for details.

You can download the plugin source files form GitHub, see Parasoft Findings Plugin Project. If you need additional information on how to rebuild the plugin, contact Parasoft Support.

Supported Compilers

The following compiler configurations are included with the DTP Engines for C/C++:

Vendor	Compiler	Host Platform	Identifier	Comments
Altrium TASKING	Altium TASKING Vx-toolset for Tri-Core C/C++ Compiler 4.0	Windows	vxtc_4_0	
ARM	ARM® Compiler 5.0	Windows	rvct_5_0	
	ARM® GNU GCC 4.5.x	Windows	armgcc_4_5	
	ARM® RealView® 3.0	Windows, Linux	rvct_3_0	
	ARM® RealView® 3.1	Windows, Linux	rvct_3_1	
	ARM® RealView® 4.0	Windows	rvct_4_0	
	ARM® RealView® 4.1	Windows	rvct_4_1	
GNU	GNU GCC 3.3.x	Windows, Linux	gcc_3_3	
	GNU GCC 3.3.x (x64)	Linux	gcc_3_3-64	
	GNU GCC 3.4x	Windows, Linux	gcc_3_4	
	GNU GCC 3.4.x (x64)	Linux	gcc_3_4-64	
	GNU GCC 4.0.x	Windows, Linux	gcc_4_0	
	GNU GCC 4.0.x (x64)	Linux	gcc_4_0-64	
	GNU GCC 4.1.x	Windows, Linux	gcc_4_1	
	GNU GCC 4.1.x (x64)	Linux	gcc_4_1-64	
	GNU GCC 4.2.x	Windows, Linux	gcc_4_2	
	GNU GCC 4.2.x (x64)	Linux	gcc_4_2-64	
	GNU GCC 4.3.x	Windows, Linux	gcc_4_3	
	GNU GCC 4.3.x (x64)	Linux	gcc_4_3-64	
	GNU GCC 4.4.x	Windows, Linux	gcc_4_4	
	GNU GCC 4.4.x (x64)	Linux	gcc_4_4-64	
	GNU GCC 4.5.x	Windows, Linux	gcc_4_5	

Vendor	Compiler	Host Platform	Identifier	Comments
	GNU GCC 4.5.x (x64)	Linux	gcc_4_5-64	
	GNU GCC 4.6.x	Windows, Linux	gcc_4_6	
	GNU GCC 4.6.x (x64)	Linux	gcc_4_6-64	
	GNU GCC 4.7.x	Windows, Linux	gcc_4_7	
	GNU GCC 4.7.x (x64)	Linux	gcc_4_7-64	
	GNU GCC 4.8.x	Windows, Linux	gcc_4_8	
	GNU GCC 4.8.x (x64)	Windows, Linux	gcc_4_8-64	
	GNU GCC 4.9.x	Windows, Linux	gcc_4_9	
	GNU GCC 4.9.x (x64)	Windows, Linux	gcc_4_9-64	
	GNU GCC 5.x	Windows, Linux	gcc_5	
	GNU GCC 5.x (x64)	Windows, Linux	gcc_5-64	
	GNU GCC 6.x	Windows, Linux	gcc_6	
	GNU GCC 6.x (x64)	Windows, Linux	gcc_6-64	
Green Hills	Green Hills® Software Compiler 5.1 V850	Windows	ghs_5_1	
	Green Hills® Software Compiler 5.0 PowerPC	Windows	ghs_5_0	
	Green Hills® Software C/C++ Compiler v2013.1.x for PowerPC	Windows	ghsppc_2013_1	
IAR	IAR Compiler 5.4x for ARM®	Windows	iccarm_5_4	
	IAR Compiler 5.5x for ARM®	Windows	iccarm_5_5	
	IAR Compiler 6.1x for ARM®	Windows	iccarm_6_1	C only
	IAR Compiler 6.3x for ARM®	Windows	iccarm_6_3	C only
	IAR Compiler 6.6x for ARM®	Windows	iccarm_6_6	
	IAR Compiler 7.4x for ARM®	Windows	iccarm_7_4	
	IAR Compiler 7.8x for ARM®	Windows	iccarm_7_8.	
	IAR Compiler 5.4x for MSP430®	Windows	icc430_5_4	SAE only

Vendor	Compiler	Host Platform	Identifier	Comments
	IAR Compiler for MSP430® v. 6.1x	Windows	icc430_6_1	SAE only
Keil	ARM® Compiler 5.0 for uVision	Windows	rvct_5_0_uV	
	ARM® RealView® 3.1 for uVision	Windows	rvct_3_1_uV	
	ARM® RealView® 4.0 for uVision	Windows	rvct_4_0_uV	
	ARM® RealView® 4.1 for uVision	Windows	rvct_4_1_uV	
Microsoft	Microsoft® Visual C++® 9.0	Windows	vc_9_0	
	Microsoft® Visual C++® 9.0 (x64)	Windows	vc_9_0-64	
	Microsoft® Visual C++® 10.0	Windows	vc_10_0	
	Microsoft® Visual C++® 10.0 (x64)	Windows	vc_10_0-64	
	Microsoft® Visual C++® 11.0	Windows	vc_11_0	
	Microsoft® Visual C++® 11.0 (x64)	Windows	vc_11_0-64	
	Microsoft® Visual C++® 12.0	Windows	vc_12_0	
	Microsoft® Visual C++® 12.0 (x64)	Windows	vc_12_0-64	
	Microsoft® Visual C++® 14.0	Windows	vc_14_0	
	Microsoft® Visual C++® 14.0 (x64)	Windows	vc_14_0-64	
National Instruments	National Instruments LabWindows/CVI 2015 Clang C/C++ Compiler v3.3 for Win32	Windows	niclang_3_3	SAE only
Renesas	Renesas RX C/C++ Compiler 2.2x	Windows	renrx_2_2	SAE only
	Renesas M16C/R8C C Compiler 5.4x	Windows	rm16c_5_4	SAE only
	Renesas SH SERIES C/C++ Compiler V.9.03.xx	Windows	hew_9_3	
	Renesas SH SERIES C/C++ Compiler V.9.04.xx	Windows	hew_9_4	
	Renesas RX V2.05.x	Wndows	renrx_2_5	
Texas Instruments	TI® ARM C/C++ Compiler v5.1.x	Linux, Windows	tiarm_5_1	
	TI® MSP430 C/C++ Compiler v3.2	Windows	timsp430_3_2	
	TI® MSP430 C/C++ Compiler v4.0	Windows	timsp430_4_0	
	TI® TMS320C2000 C/C++ Compiler v4.1	Windows	tic2000_4_1	SAE only
	TI® TMS320C2000 C/C++ Compiler v5.2	Windows	tic2000_5_2	
	TI® TMS320C2000 C/C++ Compiler v6.0	Windows	tic2000_6_0	

Vendor	Compiler	Host Platform	Identifier	Comments
	TI® TMS320C2000 C/C++ Compiler v6.2	Windows	tic2000_6_2	
	TI® TMS320C6x C/C++ Compiler v5.1	Windows	tic6000_5_1	SAE only
	TI® TMS320C6x C/C++ Compiler v6.0	Windows	tic6000_6_0	
	TI® TMS320C6x C/C++ Compiler v6.1	Windows, Linux	tic6000_6_1	
	TI® TMS320C6x C/C++ Compiler v7.3	Windows	tic6000_7_3	
	TI® TMS320C6x C/C++ Compiler v7.4	Windows	tic6000_7_4	
	TI® TMS470 C/C++ Compiler v4.9.x	Windows	tiarm_4_9	
Wind River	Wind River® Diab 5.7.x	Windows, Linux	diab_5_7	
	Wind River® Diab 5.8.x	Windows, Linux	diab_5_8	
	Wind River® Diab 5.9.x	Windows, Linux	diab_5_9	
	Wind River® GCC 3.3.x	Windows	wrgcc_3_3	
	Wind River® GCC 3.4.x	Windows, Linux	wrgcc_3_4	
	Wind River® GCC 4.1.x	Windows, Linux	wrgcc_4_1	
	Wind River® GCC 4.3.x	Windows, Linux	wrgcc_4_3	

Custom-developed and Deprecated Compilers

Configurations for the following custom-developed and deprecated compilers are available for download in the Parasoft Marketplace at https://marketplace.parasoft.com.

Vendor	Compiler	Host Platform	Identifier	Comments
Altera	Altera Nios II 5.1 b73® GCC 3.4.x	Linux	nios2gcc_3_4	SAE Only
	Altera Nios® GCC 2.9	Linux	niosgcc_2_9	SAE Only
Altium TASKING	Altium TASKING Vx-toolset for TriCore C/C++ Compiler 2.5	Windows	vxtc_2_5	C only
	Altium TASKING Vx-toolset for TriCore C/C++ Compiler 3.3	Windows	vxtc_3_3	
	Altium TASKING Vx-toolset for TriCore C/C++ Compiler 3.4	Windows	vxtc_3_4	

Vendor	Compiler	Host Platform	Identifier	Comments
	Altium TASKING Vx-toolset for TriCore C/C++ Compiler 3.5	Windows	vxtc_3_5	
	TASKING 80C196 C compiler v6.0 r1	Windows	c196_6_0	SAE only
	Altium TASKING Classic Compiler for C166/ST10 v. 6.0	Windows	tc166_6_0	SAE Only
Analog Devices	Analog Devices C/C++ Compiler 7.0 for ADSP SHARC	Windows	ad21k_7_0	SAE Only
	Analog Devices C/C++ Compiler 7.0 for ADSP TigerSHARC	Windows	adts_7_0	SAE Only
ARM	ARM® Developer Suite 1.2	Windows, Linux	ads_1_2	
	ARM® RealView® 2.2	Windows	rvct_2_2	
Embarcadero/ Borland	Borland C++ Compiler 5.6.x for Win32	Windows	bcc32_5_6	SAE only
	Embarcadero C++ Compiler 6.2x for Win32	Windows	bcc32_6_2	SAE only
	Embarcadero C++ Compiler 6.9x for Win32	Windows	bcc32_6_9	SAE only
Cosmic	COSMIC Software ® 68HC08 C Cross Compiler V4.6.x	Windows	hc08_4_6	SAE only
eCosCentric	eCosCentric® GCC 3.4.x	Linux	ecosgcc_3_4	SAE only
Freescale	Freescale CodeWarrior ANSI-C/cC++ Compiler 5.0.x for HC12	Windows	cwhc12_5_0	SAE only
	Freescale C/C++ Compiler v. 5.1 for Embedded ARM	Windows	cwarm_5_1	SAE Only
Fujitsu	FR Family SOFTUNE C/C++ Compiler V6	Windows	fr_6_5	
GNU	GNU GCC 2.9.x	Windows, Linux	gcc_2_9	
	GNU GCC 3.2.x	Windows, Linux	gcc_3_2	
Green Hills	Green Hills® Software Compiler 4.0 Native	Windows	ghs_4_0	
	Green Hills® Software Compiler 4.2 Native	Linux	ghs_4_2	
	Green Hills® Software Compiler 3.4 V850	Windows	ghs_3_4	
	Green Hills Software Compiler for PPC v. 3.5	Windows	ghsppc_3_5	

Vendor	Compiler	Host Platform	Identifier	Comments
	Green Hills Software Compiler for PPC v. 4.0.x	Windows	ghsppc_4_0	
	Green Hills Software Compiler for PPC v. 4.2.x	Windows	ghsppc_4_2	
IAR	IAR Compiler 1.4x for STM8®	Windows	iccstm8_1_40	SAE only
	IAR Compiler 5.3x for ARM®	Windows	iccarm_5_3	C only
	IAR Compiler 5.3x for MSP430®	Windows	icc430_5_3	
	IAR Compiler for MSP430 v. 4.2x	Windows	icc430_4_2	SAE only
	IAR Compiler for RX v. 2.5x	Windows	iccrx_2_50	
Keil	Keil C51 8.x	Windows	c51_8	SAE only
	Keil C166 7.0	Windows	kc166_7_0	SAE only
Mentor Graphics/ CodeSourcery	CodeSourcery Sourcery G++ Lite 2009q1-203	Windows, Linux	csgccarm_4_3	SAE Only
Microchip	Microchip MPLAB® C Compiler for dsPIC v3.2x	Windows	pic30_3_23	SAE only
	Microchip MPLAB® C32 Compiler for PIC32 v2.0x	Windows	pic32_2_0	SAE only
Microsoft	Microsoft® Visual C++® 6.0	Windows	vc_6_0	
	Microsoft® Visual C++® 7.1	Windows	vc_7_1	
	Microsoft® Visual C++® 8.0	Windows	vc_8_0	
	Microsoft® Visual C++® 8.0 (x64)	Windows	vc_8_0-64	
	Microsoft® Embedded Visual C++® 4.0	Windows	evc_4_0	
	Microsoft® Visual C++® 8.0 for Windows Mobile	Windows	evc_8_0	
	Microsoft® Visual C++® 9.0 for Windows Mobile	Windows	evc_9_0	
National Instruments	National Instruments LabWindows/CVI 9.0	Windows	nicvi_9_0	SAE Only, standard non-third party CVI compiler
	National Instruments LabWindows/CVI 2013 Clang C/C++ Compiler v2.9 for Win32	Windows	niclang_2_9	SAE only
QNX	QNX® GCC 2.9.x	Windows	qcc_2_9	
	QNX® GCC 3.3.x	Windows	qcc_3_3	
	QNX® GCC 4.2.x	Windows	qcc_4_2	

Vendor	Compiler	Host Platform	Identifier	Comments
	QNX® GCC 4.4.x	Windows	qcc_4_4	
Renesas	Renesas SH SERIES C/C++ Compiler V.5.1x.x	Windows	ew_5_1	SAE Only
STMicroelectron-ics	STMicroelectronics® ST20	Windows	st20_2_2	SAE only
	STMicroelectronics® ST40	Windows	st40_3_1	SAE only
Texas Instru-ments	TI® ARM C/C++ Compiler v5.0.x	Windows	tiarm_5_0	SAE Only
	TI® TMS320C54x C/C++ Compiler v4.2	Windows	tic54x_4_2	SAE only
	TI® TMS320C55x C/C++ Compiler v4.3	Windows	tic55x_4_3	
Wind River	Wind River® Diab 5.5.x	Windows, Linux	diab_5_5	
	Wind River® Diab 5.6.x	Windows, Linux	diab_5_6	
	Wind River® EGCS 2.9	Windows	wregcs_2_9	
	Wind River® GCC 2.9	Windows	wrgcc_2_9	

Getting Help

Use the the -help switch to access usage information on the command line.

```
cpptestcli.exe -help
```

Technical Support

You can configure DTP Engines to create package for technical support. Add the following settings to your .properties configuration file:

```
techsupport.enabled=true
techsupport.create.on.exit=true
techsupport.archive.location=[OUTPUT DIRECTORY]
```

A technical support package will be created in the output directory at the end of an analysis run.

Troubleshooting

Floating Machine ID

Changes in the network environment may affect the interface that is used to compute your machine ID and result in machine ID instability. You can use the PARASOFT_SUPPORT_NET_INTERFACES environment variable to specify a stable interface and prevent the machine ID from floating.

1. Set up the **PARASOFT_SUPPORT_NET_INTERFACES** environment variable.
2. Set the variable value to a stable Ethernet network interface. Do not use virtual, temporary or loopback interfaces.
    - On Windows: Set the value to the MAC address of your network card. You can use the `ipconfig -all` command to obtain the address. Example:

```
SET PARASOFT_SUPPORT_NET_INTERFACES=00-10-D9-27-AC-85
```

    - On Linux: Set the value to one of the network interfaces from the "inet" or "inet6" family. For example: You can use the `ifconfig` command to obtain the list of available interfaces. Example:

```
export PARASOFT_SUPPORT_NET_INTERFACES=eth1
```

If the problem persists, you can obtain diagnostic information by setting up the environment variable **PARASOFT_DEBUG_NET_INTERFACES** and setting its value to `true`. This will print to the standard output the checking procedure that can be shared with technical support , as well as the interface that is used to compute your machine ID. The interface will be marked with the `[SELECTED]` prefix.

Third-Party Content

DTP Engines for C/C++ incorporate items that have been sourced from third parties. The names of the items and their license agreements have been listed in the table. Click the license name to see the details.

Item	License
commons-collections.jar	Apache License 2.0
commons-vfs.jar	Apache License 2.0
avalon-framework.jar	Apache License 2.0
batik-all.jar	Apache License 2.0
fop.jar	Apache License 2.0
chardet.jar	Mozilla Public License
bcprov.jar	MIT License
saxon.jar	Mozilla Public License
jfreechart.jar	GNU LGPL License
jcommon.jar	GNU LGPL License
cvslib.jar	CDDL License
javax.xml.stream_1.0.1.jar	Eclipse Public License
javax.activation_1.1.1.jar	Apache License 2.0
jakarta-log4j.jar	Apache License 2.0
xmlgraphics-commons.jar	Apache License 2.0
fst.jar	Apache License 2.0
truezip.jar	Apache License 2.0
jjawin.jar	DevelopMentor OpenSource Software License
trilead-ssh2.jar	Trilead AG License
javanet.staxutils_1.0.0.jar	BSD License
commons-codec.jar	Apache License 2.0
commons-httpclient.jar	Apache License 2.0
org.apache.commons.io_1.4.0.v20081110-1000.jar	Apache License 2.0
org.apache.commons.logging_1.1.3.jar	Apache License 2.0
fluent-hc.jar	Apache License 2.0
httpclient.jar	Apache License 2.0

Item	License
httpcore.jar	Apache License 2.0
httpclient-cache.jar	Apache License 2.0
htpmime.jar	Apache License 2.0
org.apache.jcs_1.3.4.jar	Apache License 2.0
org.codehaus.stax2_3.2.4.jar	Apache License 2.0
org.json_1.0.0.v201507290100.jar	JSON License
javax.mail_1.5.0.jar	CDDL License
org.suigeneris.jrcs.diff_0.4.2.jar	GNU LGPL License
org.apache.felix.scr-1.6.2.jar	Apache License 2.0
osgi.core-5.0.0.jar	Apache License 2.0
Log4Cplus	Apache License 2.0
log4j-1.2.17.jar	Apache License 2.0
org.apache.felix.main-4.2.0.jar	Apache License 2.0
org.apache.felix.scr-1.6.2.jar	Apache License 2.0
Microsoft Visual C++ Redistributable Libraries	Microsoft Visual C++ Redistributable Libraries (Visual Studio License)
PCRE (Perl-Compatible Regular Expressions)	BSD License
Java JRE	Oracle Binary Code License
Parsifal XML Parser	Public Domain and GNU LGPL License
Python	Python Software Foundation License
ConvertUTF.cc/.h code	Unicode Consortium License
groovy-all-2.4.7.jar	Apache License 2.0